

Serverless Stack

Serverless Stack is an open source guide for building full-stack production ready apps using Serverless and React on AWS. Create a note taking app from scratch using the Serverless Framework and Create React App. Follow our step-by-step tutorials with screenshots and code samples. And use our forums if you have any questions.

IMPORTANT: In some versions of Adobe Reader, copying and pasting code from this PDF might miss some hyphen or dash characters. Please, double check the code you are copying.

October 8, 2019 - v4.0

Who Is This Guide For?

This guide is meant for full-stack developers or developers that would like to build full stack serverless applications. By providing a step-by-step guide for both the frontend and the backend we hope that it addresses all the different aspects of building serverless applications. There are quite a few other tutorials on the web but we think it would be useful to have a single point of reference for the entire process. This guide is meant to serve as a resource for learning about how to build and deploy serverless applications, as opposed to laying out the best possible way of doing so.

So you might be a backend developer who would like to learn more about the frontend portion of building serverless apps or a frontend developer that would like to learn more about the backend; this guide should have you covered.

We are also catering this solely towards JavaScript developers for now. We might target other languages and environments in the future. But we think this is a good starting point because it can be really beneficial as a full-stack developer to use a single language (JavaScript) and environment (Node.js) to build your entire application.

On a personal note, the serverless approach has been a giant revelation for us and we wanted to create a resource where we could share what we've learned. You can read more about us **here** (/about.html). And check out a sample of what folks have built with Serverless Stack (/showcase.html).

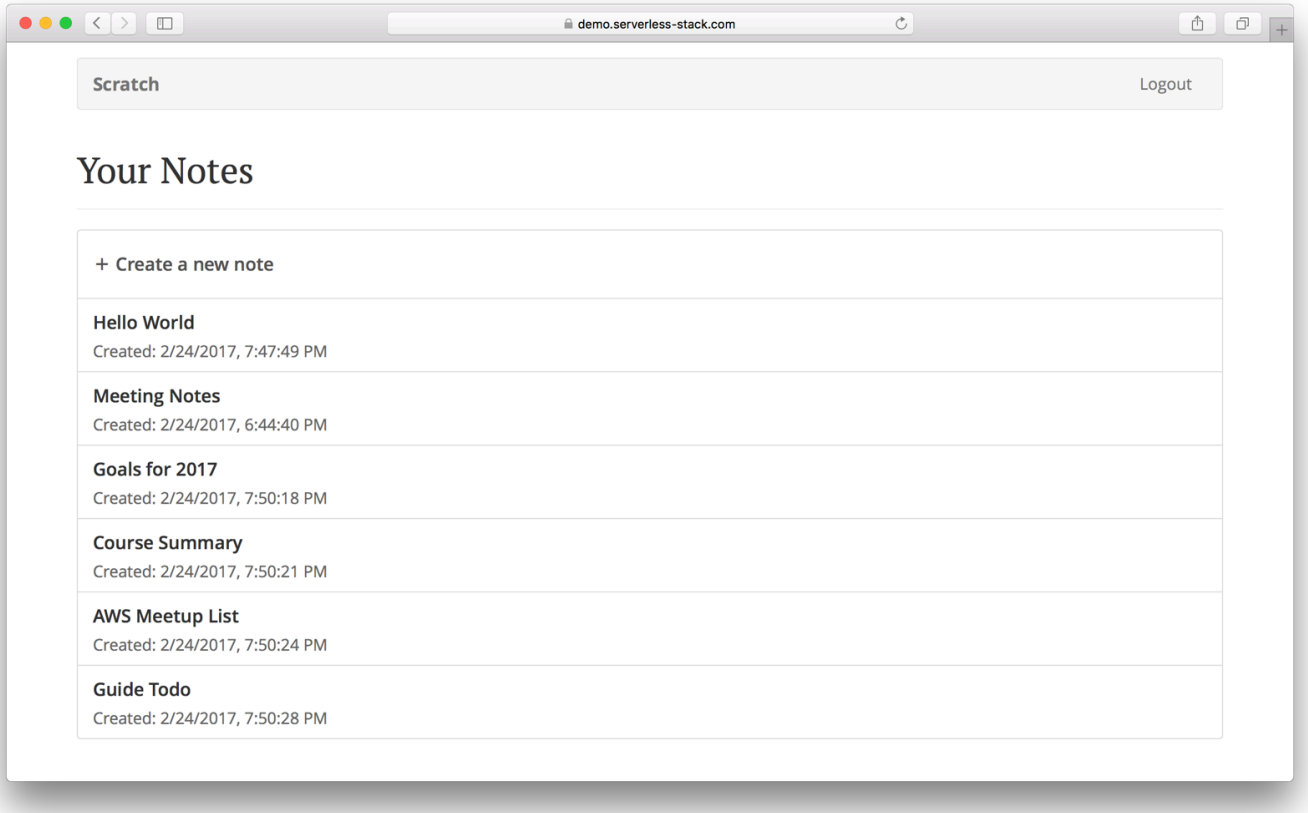
Let's start by looking at what we'll be covering.

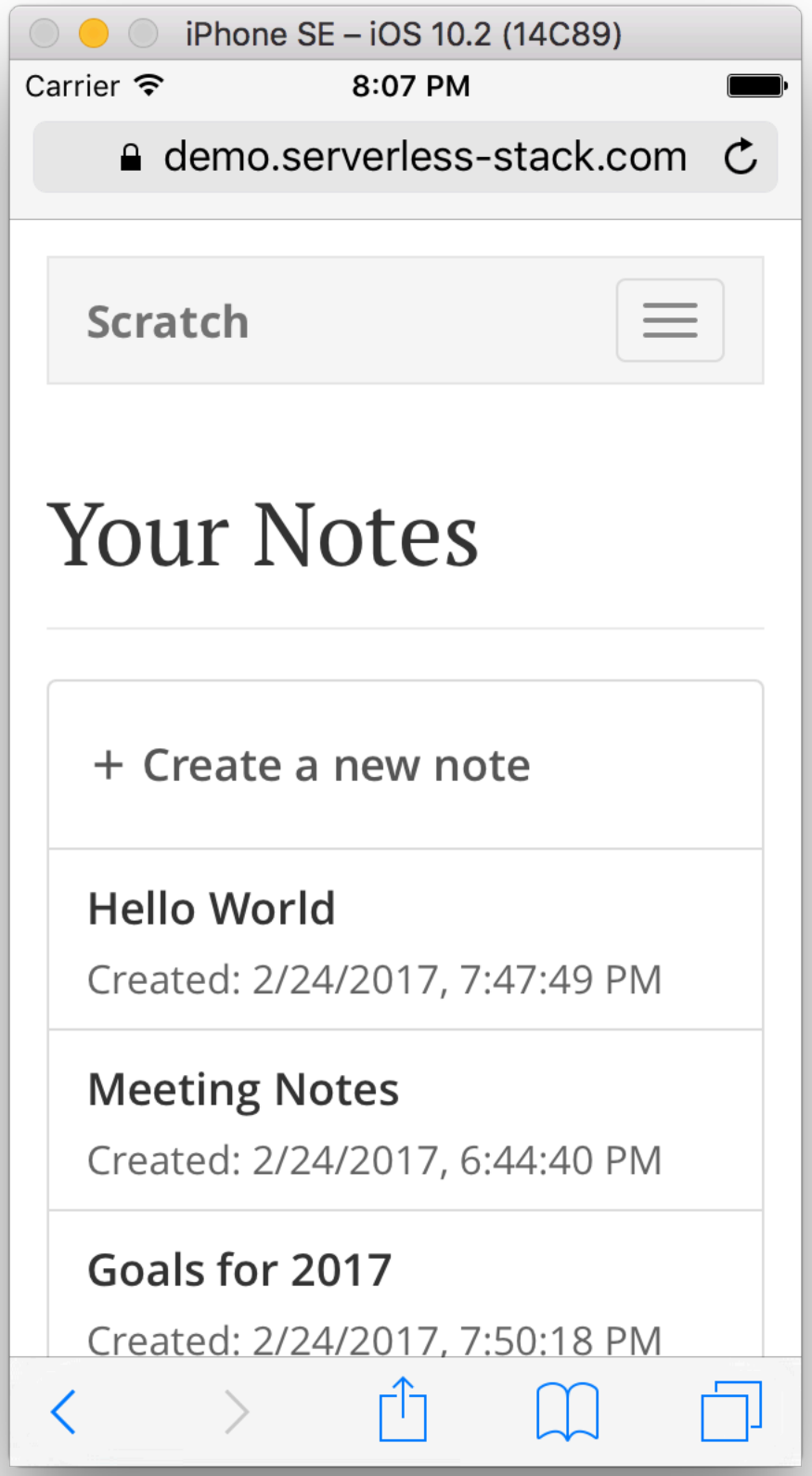
For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/who-is-this-guide-for/96)

What Does This Guide Cover?

To step through the major concepts involved in building web applications, we are going to be building a simple note taking app called **Scratch** (https://demo2.serverless-stack.com).





It is a single page application powered by a serverless API written

completely in JavaScript. Here is the complete source for the backend (https://github.com/AnomalyInnovations/serverless-stack-demo-api) and the frontend (https://github.com/AnomalyInnovations/serverless-stack-demo-client). It is a relatively simple application but we are going to address the following requirements.

- Should allow users to signup and login to their accounts
- Users should be able to create notes with some content
- Each note can also have an uploaded file as an attachment
- Allow users to modify their note and the attachment
- Users can also delete their notes
- The app should be able to process credit card payments
- App should be served over HTTPS on a custom domain
- The backend APIs need to be secure
- The app needs to be responsive

We'll be using the AWS Platform to build it. We might expand further and cover a few other platforms but we figured the AWS Platform would be a good place to start.

Technologies & Services

We'll be using the following set of technologies and services to build our serverless application.

- Lambda (https://aws.amazon.com/lambda/) & API Gateway (https://aws.amazon.com/api-gateway/) for our serverless API
- DynamoDB (https://aws.amazon.com/dynamodb/) for our database
- Cognito (https://aws.amazon.com/cognito/) for user authentication and securing our APIs
- S3 (https://aws.amazon.com/s3/) for hosting our app and file uploads

- CloudFront (https://aws.amazon.com/cloudfront/) for serving out our app
- Route 53 (https://aws.amazon.com/route53/) for our domain
- Certificate Manager (https://aws.amazon.com/certificate-manager) for SSL
- React.js (https://facebook.github.io/react/) for our single page app
- React Router (https://github.com/ReactTraining/react-router) for routing
- Bootstrap (http://getbootstrap.com) for the UI Kit
- Stripe (https://stripe.com) for processing credit card payments
- Seed (https://seed.run) for automating Serverless deployments
- Netlify (https://netlify.com) for automating React deployments
- GitHub (https://github.com) for hosting our project repos.

We are going to be using the **free tiers** for the above services. So you should be able to sign up for them for free. This of course does not apply to purchasing a new domain to host your app. Also for AWS, you are required to put in a credit card while creating an account. So if you happen to be creating resources above and beyond what we cover in this tutorial, you might end up getting charged.

While the list above might look daunting, we are trying to ensure that upon completing the guide you'll be ready to build **real-world**, **secure**, and **fully-functional** web apps. And don't worry we'll be around to help!

Requirements

You just need a couple of things to work through this guide:

- Node v8.10+ and NPM v5.5+ (https://nodejs.org/en/) installed on your machine.

- A free GitHub account (https://github.com/join).
- And basic knowledge of how to use the command line.

How This Guide Is Structured

The guide is split roughly into a couple of parts:

1. **The Basics**

   Here we go over how to create your first full-stack Serverless application. These chapters are roughly split up between the backend (Serverless) and the frontend (React). We also talk about how to dpeloy your serverless app and React app into production.

   This section of the guide is carefully designed to be completed in its entirety. We go into all the steps in detail and have tons of screenshots to help you build your first app.

2. **The Best Practices**

   We launched this guide in early 2017 with just the first part. The Serverless Stack community has grown and many of our readers have used the setup described in this guide to build apps that power their businesses. In this section, we cover the best practices of running production applications. These really begin to matter once your application codebase grows or when you add more folks to your team.

   The chapters in this section are relatively standalone and tend to revolve around specific topcis.

3. **Reference**

   Finally, we have a collection of standalone chapters on various topics.

We either refer to these in the guide or we use this to cover topics that don't necessarily belong to either of the two above sections.

Building Your First Serverless App

The first part of this guide helps you create the the notes application and deploy it to production. We cover all the basics. Each service is created by hand. Here is what is covered in order.

For the backend:

- Configure your AWS account
- Create your database using DynamoDB
- Set up S3 for file uploads
- Set up Cognito User Pools to manage user accounts
- Set up Cognito Identity Pool to secure our file uploads
- Set up the Serverless Framework to work with Lambda & API Gateway
- Write the various backend APIs
- Working with external APIs (Stripe)
- Deploy your app through the command line

For the frontend:

- Set up our project with Create React App
- Add favicons, fonts, and a UI Kit using Bootstrap
- Set up routes using React-Router
- Use AWS Cognito SDK to login and signup users
- Plugin to the backend APIs to manage our notes
- Use the AWS JS SDK to upload files
- Accepting credit card payments in React
- Environments in Create React App
- Deploy your frontend to production using Netlify

- Configure custom domains through Netlify

Automate backend deployments:

- Configure DynamoDB through code
- Configure S3 through code
- Configure Cognito User Pool through code
- Configure Cognito Identity Pool through code
- Environment variables in Serverless Framework
- Working with secrets in Serverless Framework
- Unit tests in Serverless
- Automating deployments using Seed
- Configuring custom domains through Seed
- Monitoring deployments through Seed

We think this will give you a good foundation on building full-stack production ready serverless applications. If there are any other concepts or technologies you'd like us to cover, feel free to let us know on our forums (https://discourse.serverless-stack.com).

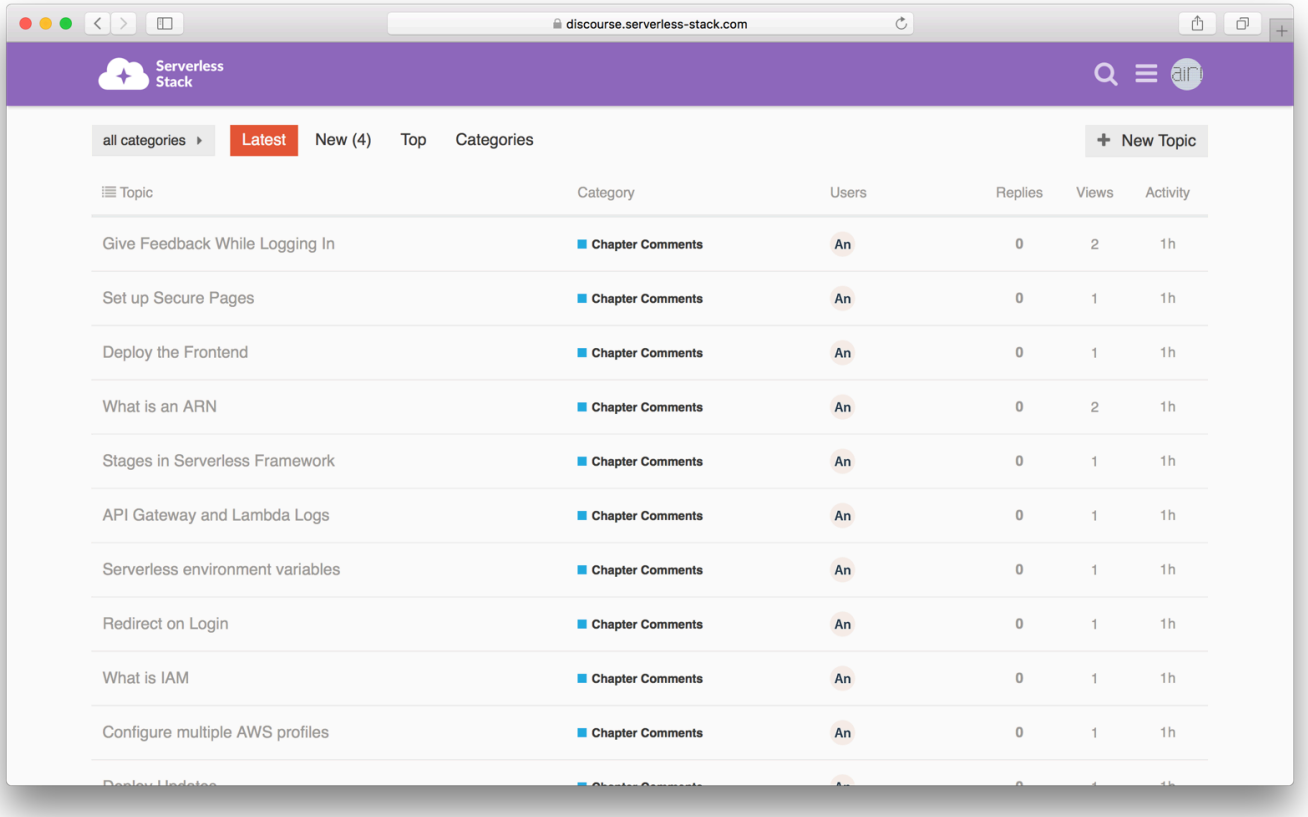
For help and discussion

💬 Comments on this chapter (https://discourse.serverless-stack.com/t/what-does-this-guide-cover/83)

How to Get Help?

In case you find yourself having problems with a certain step, we want to make sure that we are around to help you fix it and figure it out. There are a few ways to get help.

- We use Discourse forum topics (https://discourse.serverless-stack.com) as our comments and we've helped resolve quite a few issues in the past. So make sure to check the comments under each chapter to see if somebody else has run into the same issue as you have.
- Post in the comments for the specific chapter detailing your issue and one of us will respond.



This entire guide is hosted on GitHub (https://github.com/AnomalyInnovations/serverless-stack-com). So if you find an error you can always:

- Open a new issue (https://github.com/AnomalyInnovations/serverless-stack-com/issues/new)
- Or if you've found a typo, edit the page and submit a pull request!

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/how-to-get-help/95)

What is Serverless?

Traditionally, we've built and deployed web applications where we have some degree of control over the HTTP requests that are made to our server. Our application runs on that server and we are responsible for provisioning and managing the resources for it. There are a few issues with this.

1. We are charged for keeping the server up even when we are not serving out any requests.
2. We are responsible for uptime and maintenance of the server and all its resources.
3. We are also responsible for applying the appropriate security updates to the server.
4. As our usage scales we need to manage scaling up our server as well. And as a result manage scaling it down when we don't have as much usage.

For smaller companies and individual developers this can be a lot to handle. This ends up distracting from the more important job that we have; building and maintaining the actual application. At larger organizations this is handled by the infrastructure team and usually it is not the responsibility of the individual developer. However, the processes necessary to support this can end up slowing down development times. As you cannot just go ahead and build your application without working with the infrastructure

team to help you get up and running. As developers we've been looking for a solution to these problems and this is where serverless comes in.

Serverless Computing

Serverless computing (or serverless for short), is an execution model where the cloud provider (AWS, Azure, or Google Cloud) is responsible for executing a piece of code by dynamically allocating the resources. And only charging for the amount of resources used to run the code. The code is typically run inside stateless containers that can be triggered by a variety of events including http requests, database events, queuing services, monitoring alerts, file uploads, scheduled events (cron jobs), etc. The code that is sent to the cloud provider for execution is usually in the form of a function. Hence serverless is sometimes referred to as *"Functions as a Service"* or *"FaaS"*. Following are the FaaS offerings of the major cloud providers:

- AWS: AWS Lambda (https://aws.amazon.com/lambda/)
- Microsoft Azure: Azure Functions (https://azure.microsoft.com/en-us/services/functions/)
- Google Cloud: Cloud Functions (https://cloud.google.com/functions/)

While serverless abstracts the underlying infrastructure away from the developer, servers are still involved in executing our functions.

Since your code is going to be executed as individual functions, there are a couple of things that we need to be aware of.

Microservices

The biggest change that we are faced with while transitioning to a serverless world is that our application needs to be architectured in the

form of functions. You might be used to deploying your application as a single Rails or Express monolith app. But in the serverless world you are typically required to adopt a more microservice based architecture. You can get around this by running your entire application inside a single function as a monolith and handling the routing yourself. But this isn't recommended since it is better to reduce the size of your functions. We'll talk about this below.

Stateless Functions

Your functions are typically run inside secure (almost) stateless containers. This means that you won't be able to run code in your application server that executes long after an event has completed or uses a prior execution context to serve a request. You have to effectively assume that your function is invoked in a new container every single time.

There are some subtleties to this and we will discuss in the What is AWS Lambda (/chapters/what-is-aws-lambda.html) chapter.

Cold Starts

Since your functions are run inside a container that is brought up on demand to respond to an event, there is some latency associated with it. This is referred to as a *Cold Start*. Your container might be kept around for a little while after your function has completed execution. If another event is triggered during this time it responds far more quickly and this is typically known as a *Warm Start*.

The duration of cold starts depends on the implementation of the specific cloud provider. On AWS Lambda it can range from anywhere between a few hundred milliseconds to a few seconds. It can depend on the runtime

(or language) used, the size of the function (as a package), and of course the cloud provider in question. Cold starts have drastically improved over the years as cloud providers have gotten much better at optimizing for lower latency times.

Aside from optimizing your functions, you can use simple tricks like a separate scheduled function to invoke your function every few minutes to keep it warm. Serverless Framework (https://serverless.com) which we are going to be using in this tutorial has a few plugins to help keep your functions warm (https://github.com/FidelLimited/serverless-plugin-warmup).

Now that we have a good idea of serverless computing, let's take a deeper look at what a Lambda function is and how your code will be executed.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/what-is-serverless/27)

What is AWS Lambda?

AWS Lambda (https://aws.amazon.com/lambda/) (or Lambda for short) is a serverless computing service provided by AWS. In this chapter we are going to be using Lambda to build our serverless application. And while we don't need to deal with the internals of how Lambda works, it's important to have a general idea of how your functions will be executed.

Lambda Specs

Let's start by quickly looking at the technical specifications of AWS Lambda. Lambda supports the following runtimes.

- Node.js: v10.15 and v8.10
- Java 8
- Python: 3.7, 3.6, and 2.7
- .NET Core: 1.0.1 and 2.1
- Go 1.x
- Ruby 2.5
- Rust

Each function runs inside a container with a 64-bit Amazon Linux AMI. And the execution environment has:

- Memory: 128MB - 3008MB, in 64 MB increments
- Ephemeral disk space: 512MB
- Max execution duration: 900 seconds
- Compressed package size: 50MB

- Uncompressed package size: 250MB

You might notice that CPU is not mentioned as a part of the container specification. This is because you cannot control the CPU directly. As you increase the memory, the CPU is increased as well.

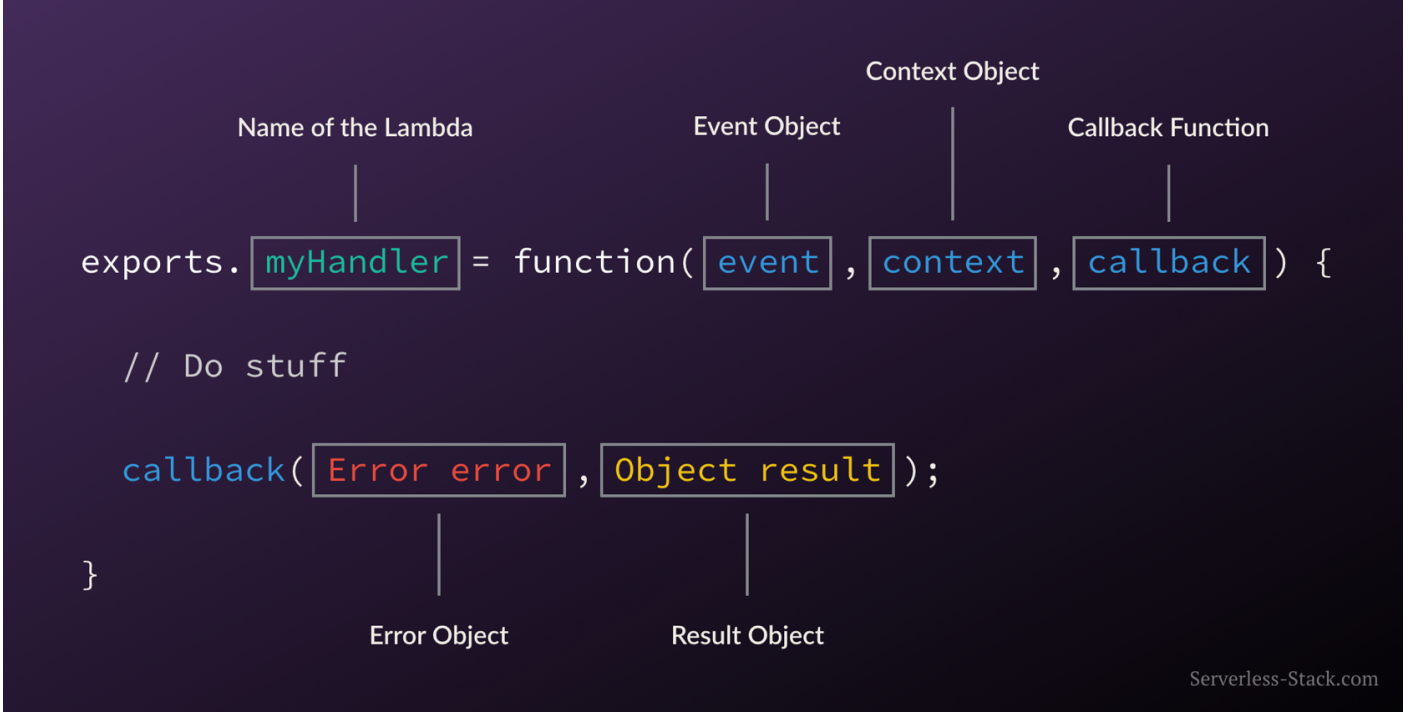
The ephemeral disk space is available in the form of the `/tmp` directory. You can only use this space for temporary storage since subsequent invocations will not have access to this. We'll talk a bit more on the stateless nature of the Lambda functions below.

The execution duration means that your Lambda function can run for a maximum of 900 seconds or 15 minutes. This means that Lambda isn't meant for long running processes.

The package size refers to all your code necessary to run your function. This includes any dependencies (`node_modules/` directory in case of Node.js) that your function might import. There is a limit of 250MB on the uncompressed package and a 50MB limit once it has been compressed. We'll take a look at the packaging process below.

Lambda Function

Finally here is what a Lambda function (a Node.js version) looks like.



Here `myHandler` is the name of our Lambda function. The `event` object contains all the information about the event that triggered this Lambda. In the case of an HTTP request it'll be information about the specific HTTP request. The `context` object contains info about the runtime our Lambda function is executing in. After we do all the work inside our Lambda function, we simply call the `callback` function with the results (or the error) and AWS will respond to the HTTP request with it.

Packaging Functions

Lambda functions need to be packaged and sent to AWS. This is usually a process of compressing the function and all its dependencies and uploading it to a S3 bucket. And letting AWS know that you want to use this package when a specific event takes place. To help us with this process we use the Serverless Framework (https://serverless.com). We'll go over this in detail later on in this guide.

Execution Model

The container (and the resources used by it) that runs our function is managed completely by AWS. It is brought up when an event takes place and is turned off if it is not being used. If additional requests are made while the original event is being served, a new container is brought up to serve a request. This means that if we are undergoing a usage spike, the cloud provider simply creates multiple instances of the container with our function to serve those requests.

This has some interesting implications. Firstly, our functions are effectively stateless. Secondly, each request (or event) is served by a single instance of a Lambda function. This means that you are not going to be handling concurrent requests in your code. AWS brings up a container whenever there is a new request. It does make some optimizations here. It will hang on to the container for a few minutes (5 - 15mins depending on the load) so it can respond to subsequent requests without a cold start.

Stateless Functions

The above execution model makes Lambda functions effectively stateless. This means that every time your Lambda function is triggered by an event it is invoked in a completely new environment. You don't have access to the execution context of the previous event.

However, due to the optimization noted above, the actual Lambda function is invoked only once per container instantiation. Recall that our functions are run inside containers. So when a function is first invoked, all the code in our handler function gets executed and the handler function gets invoked. If the container is still available for subsequent requests, your function will get invoked and not the code around it.

For example, the `createNewDbConnection` method below is called once

per container instantiation and not every time the Lambda function is invoked. The `myHandler` function on the other hand is called on every invocation.

```
var dbConnection = createNewDbConnection();

exports.myHandler = function(event, context, callback)
{
  var result = dbConnection.makeQuery();
  callback(null, result);
};
```

This caching effect of containers also applies to the `/tmp` directory that we talked about above. It is available as long as the container is being cached.

Now you can guess that this isn't a very reliable way to make our Lambda functions stateful. This is because we just don't control the underlying process by which Lambda is invoked or its containers are cached.

Pricing

Finally, Lambda functions are billed only for the time it takes to execute your function. And it is calculated from the time it begins executing till when it returns or terminates. It is rounded up to the nearest 100ms.

Note that while AWS might keep the container with your Lambda function around after it has completed; you are not going to be charged for this.

Lambda comes with a very generous free tier and it is unlikely that you will go over this while working on this guide.

The Lambda free tier includes 1M free requests per month and 400,000 GB-seconds of compute time per month. Past this, it costs $0.20 per 1 million requests and $0.00001667 for every GB-seconds. The GB-seconds is based on the memory consumption of the Lambda function. For further details check out the Lambda pricing page (https://aws.amazon.com/lambda/pricing/).

In our experience, Lambda is usually the least expensive part of our infrastructure costs.

Next, let's take a deeper look into the advantages of serverless, including the total cost of running our demo app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/what-is-aws-lambda/308)

Why Create Serverless Apps?

It is important to address why it is worth learning how to create serverless apps. There are a few reasons why serverless apps are favored over traditional server hosted apps:

1. Low maintenance
2. Low cost
3. Easy to scale

The biggest benefit by far is that you only need to worry about your code and nothing else. The low maintenance is a result of not having any servers to manage. You don't need to actively ensure that your server is running properly, or that you have the right security updates on it. You deal with your own application code and nothing else.

The main reason it's cheaper to run serverless applications is that you are effectively only paying per request. So when your application is not being used, you are not being charged for it. Let's do a quick breakdown of what it would cost for us to run our note taking application. We'll assume that we have 1000 daily active users making 20 requests per day to our API, and storing around 10MB of files on S3. Here is a very rough calculation of our costs.

Service	Rate	Cost
Cognito	Free[1]	$0.00

API Gateway	$3.5/M reqs + $0.09/GB transfer	$2.20
Lambda	Free[2]	$0.00
DynamoDB	$0.0065/hr 10 write units, $0.0065/hr 50 read units[3]	$2.80
S3	$0.023/GB storage, $0.005/K PUT, $0.004/10K GET, $0.0025/M objects[4]	$0.24
CloudFront	$0.085/GB transfer + $0.01/10K reqs	$0.86
Route53	$0.50 per hosted zone + $0.40/M queries	$0.50
Certificate Manager	Free	$0.00
Total		**$6.10**

[1] Cognito is free for < 50K MAUs and $0.00550/MAU onwards.
[2] Lambda is free for < 1M requests and 400000GB-secs of compute.
[3] DynamoDB gives 25GB of free storage.
[4] S3 gives 1GB of free transfer.

So that comes out to $6.10 per month. Additionally, a .com domain would cost us $12 per year, making that the biggest up front cost for us. But just keep in mind that these are very rough estimates. Real-world usage patterns are going to be very different. However, these rates should give you a sense of how the cost of running a serverless application is calculated.

Finally, the ease of scaling is thanks in part to DynamoDB which gives us near infinite scale and Lambda that simply scales up to meet the demand. And of course our front end is a simple static single page app that is almost guaranteed to always respond instantly thanks to CloudFront.

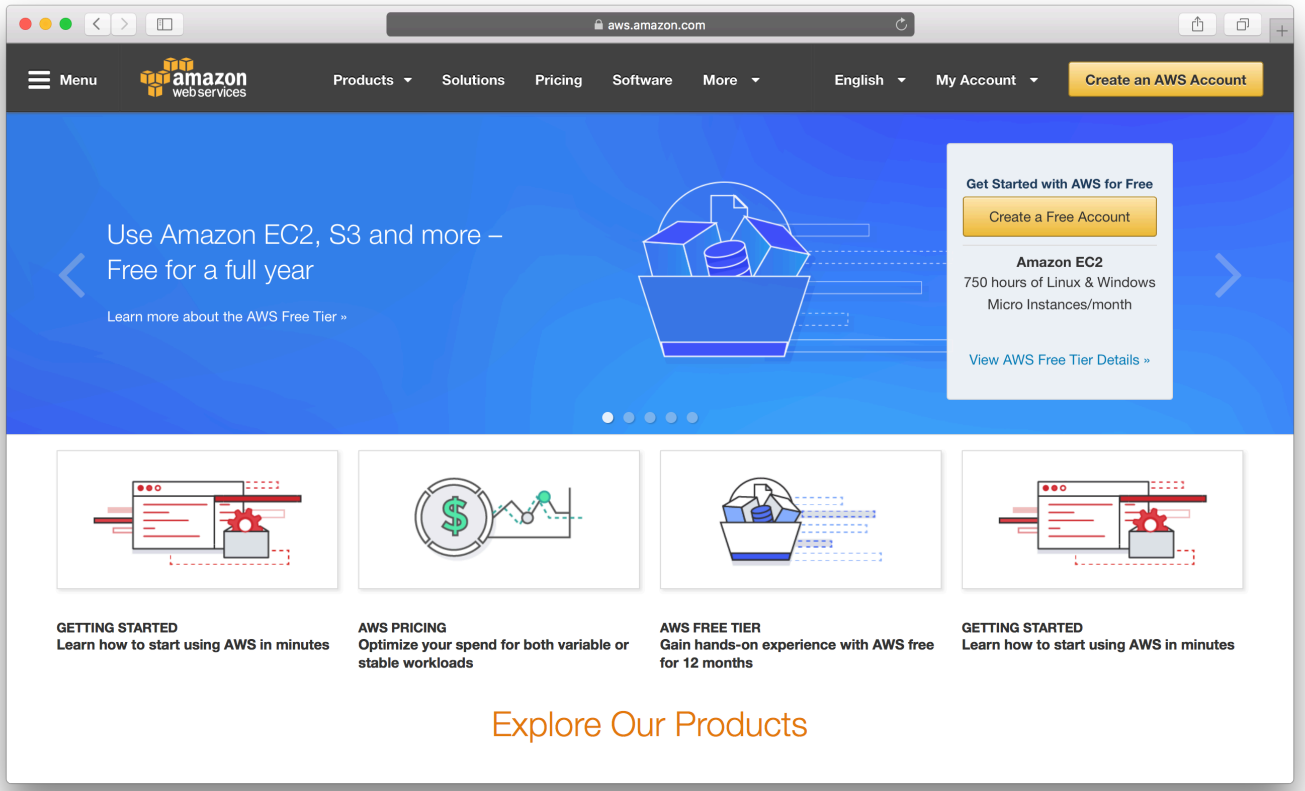
Great! Now that you are convinced on why you should build serverless apps; let's get started.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/why-create-serverless-apps/87)

Create an AWS Account

Let's first get started by creating an AWS (Amazon Web Services) account. Of course you can skip this if you already have one. Head over to the AWS homepage (https://aws.amazon.com) and hit the **Create a Free Account** and follow the steps to create your account.



Next let's configure your account so it's ready to be used for the rest of our guide.

For help and discussion

💬 Comments on this chapter (https://discourse.serverless-stack.com/t/create-an-aws-account/88)

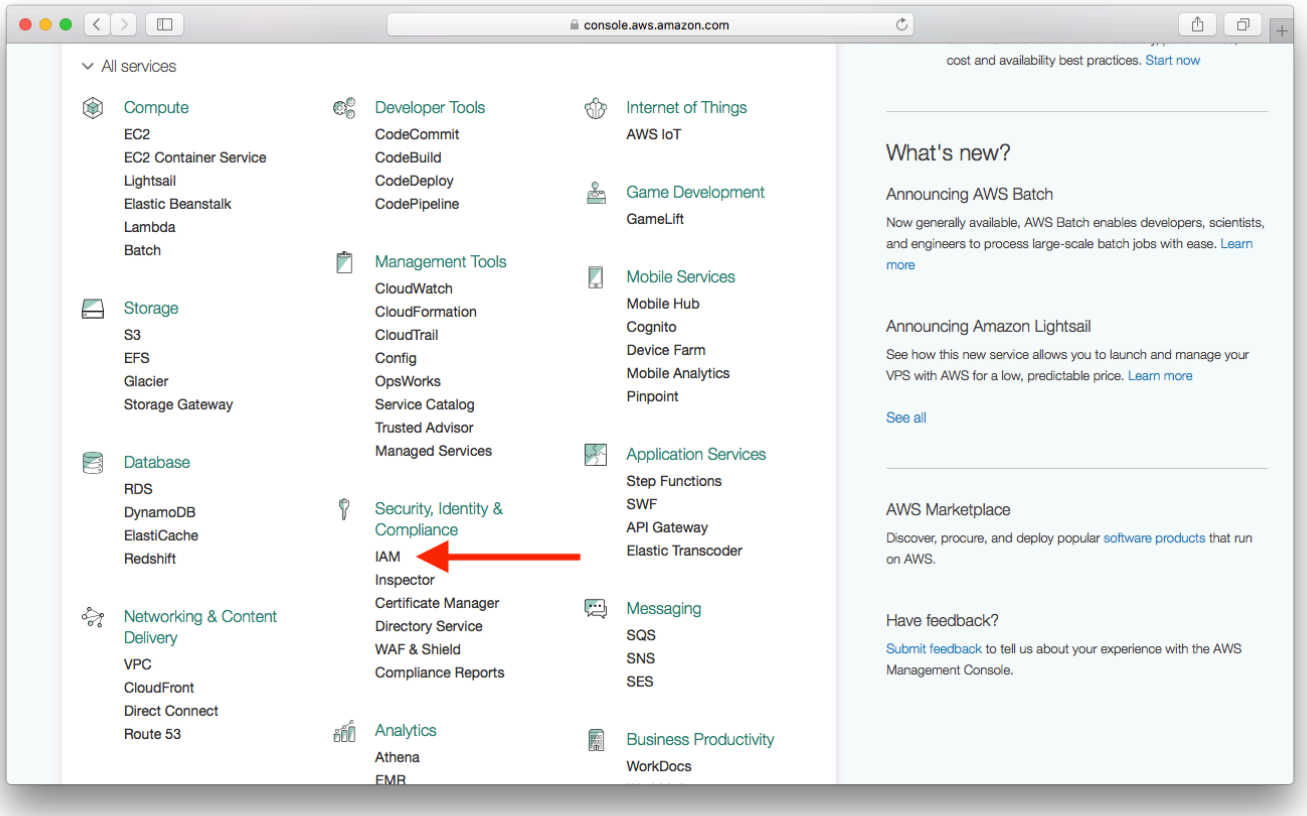
Create an IAM User

Amazon IAM (Identity and Access Management) enables you to manage users and user permissions in AWS. You can create one or more IAM users in your AWS account. You might create an IAM user for someone who needs access to your AWS console, or when you have a new application that needs to make API calls to AWS. This is to add an extra layer of security to your AWS account.

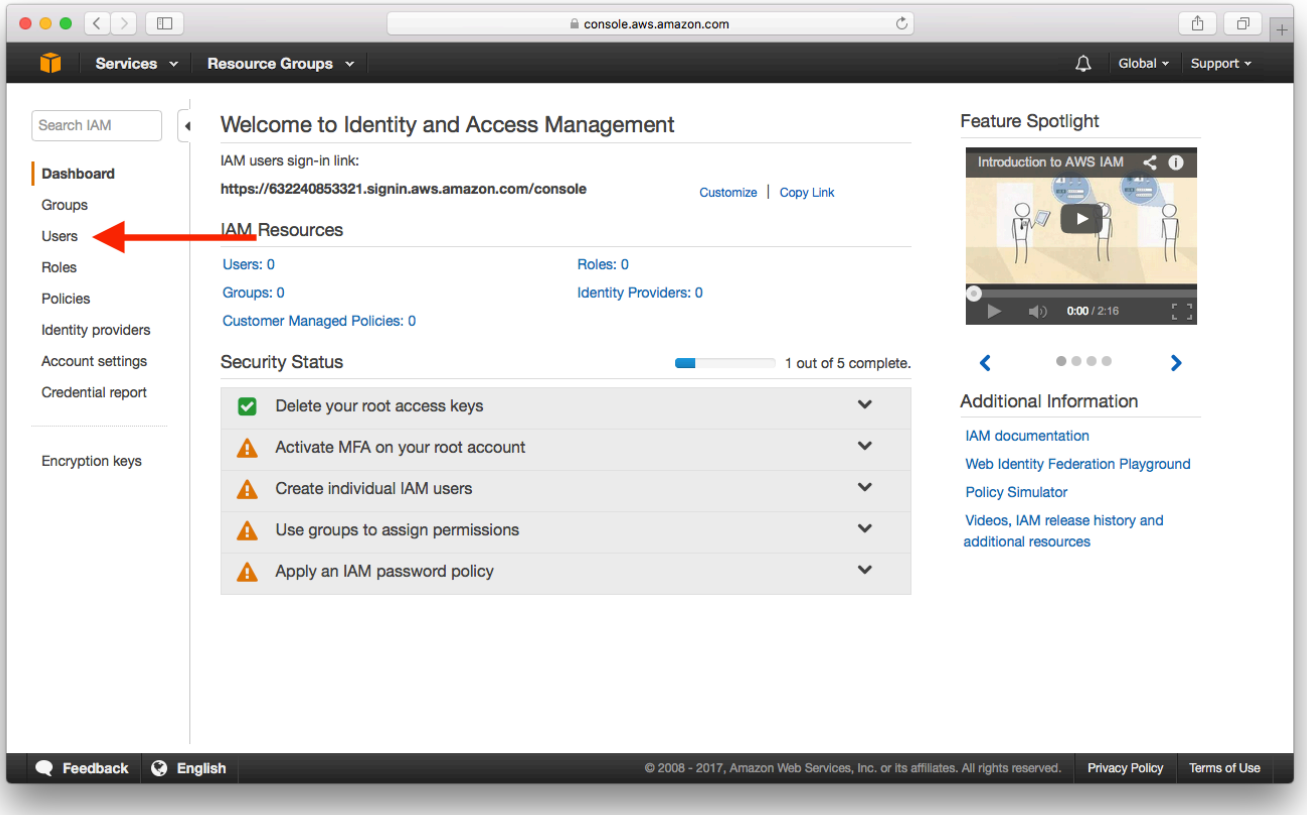
In this chapter, we are going to create a new IAM user for a couple of the AWS related tools we are going to be using later.

Create User

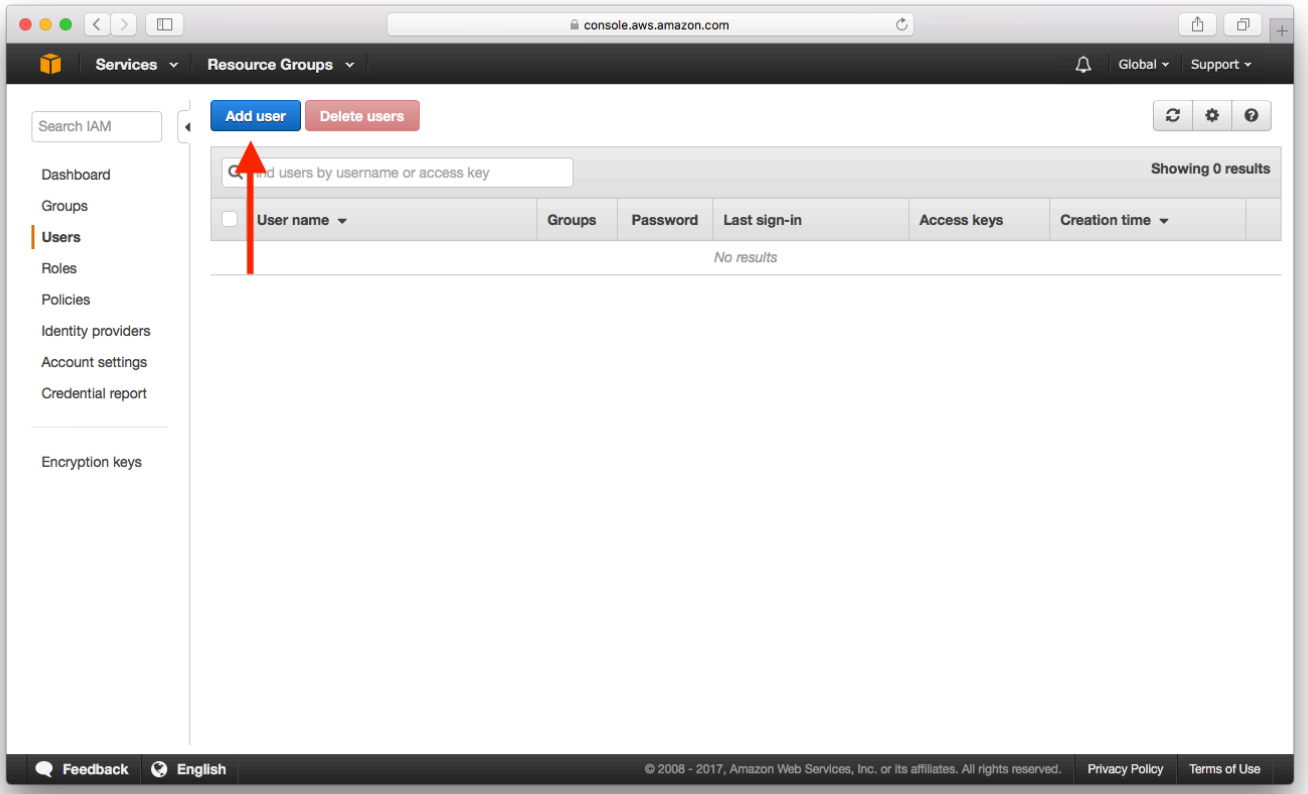
First, log in to your AWS Console (https://console.aws.amazon.com) and select IAM from the list of services.



Select **Users**.

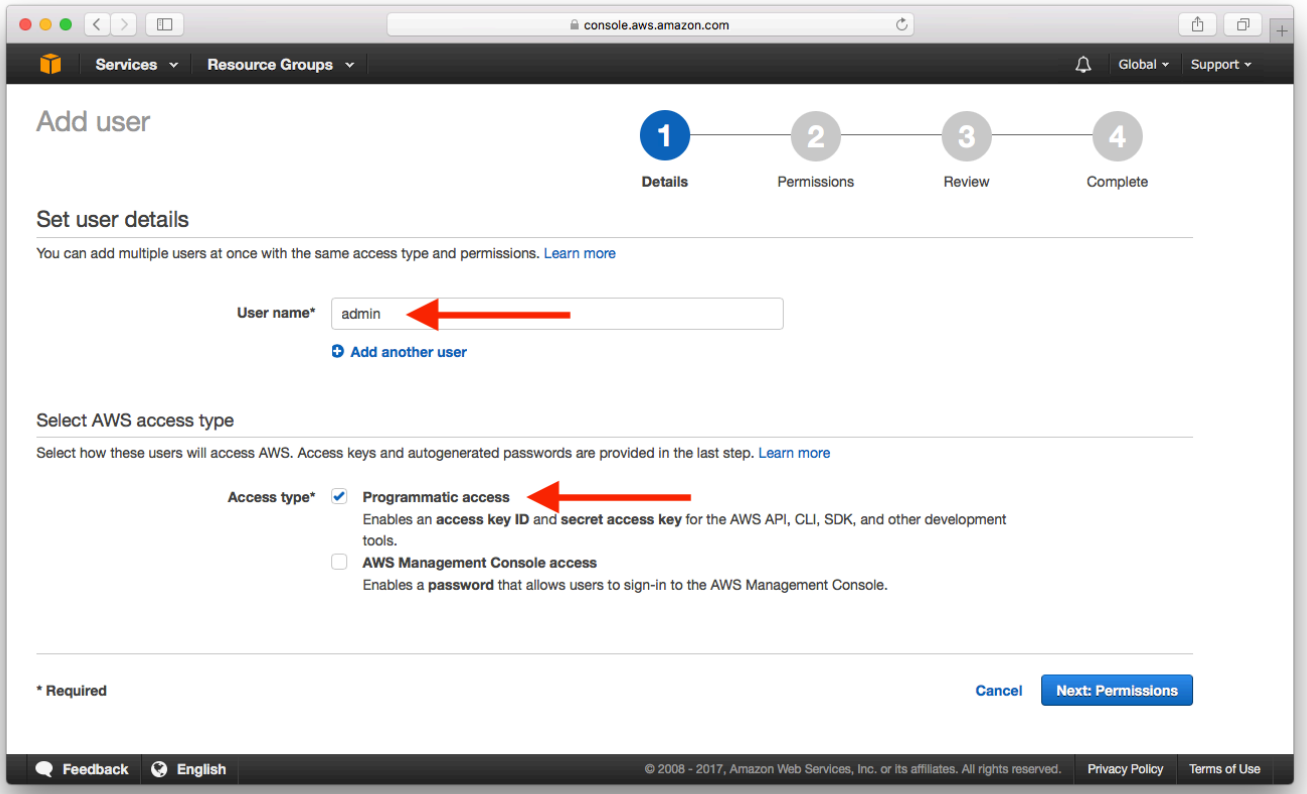


Select **Add User**.

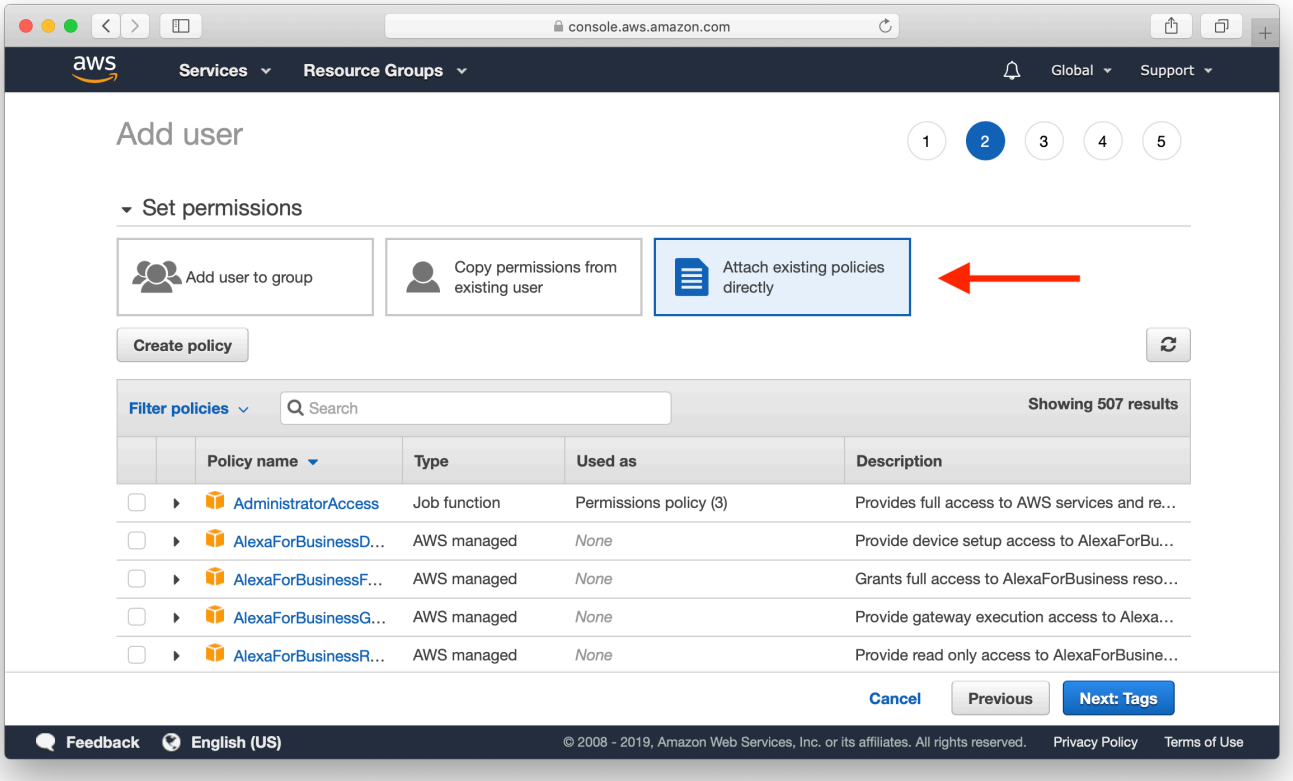


Enter a **User name** and check **Programmatic access**, then select **Next: Permissions**.

This account will be used by our AWS CLI (https://aws.amazon.com/cli/) and Serverless Framework (https://serverless.com). They'll be connecting to the AWS API directly and will not be using the Management Console.



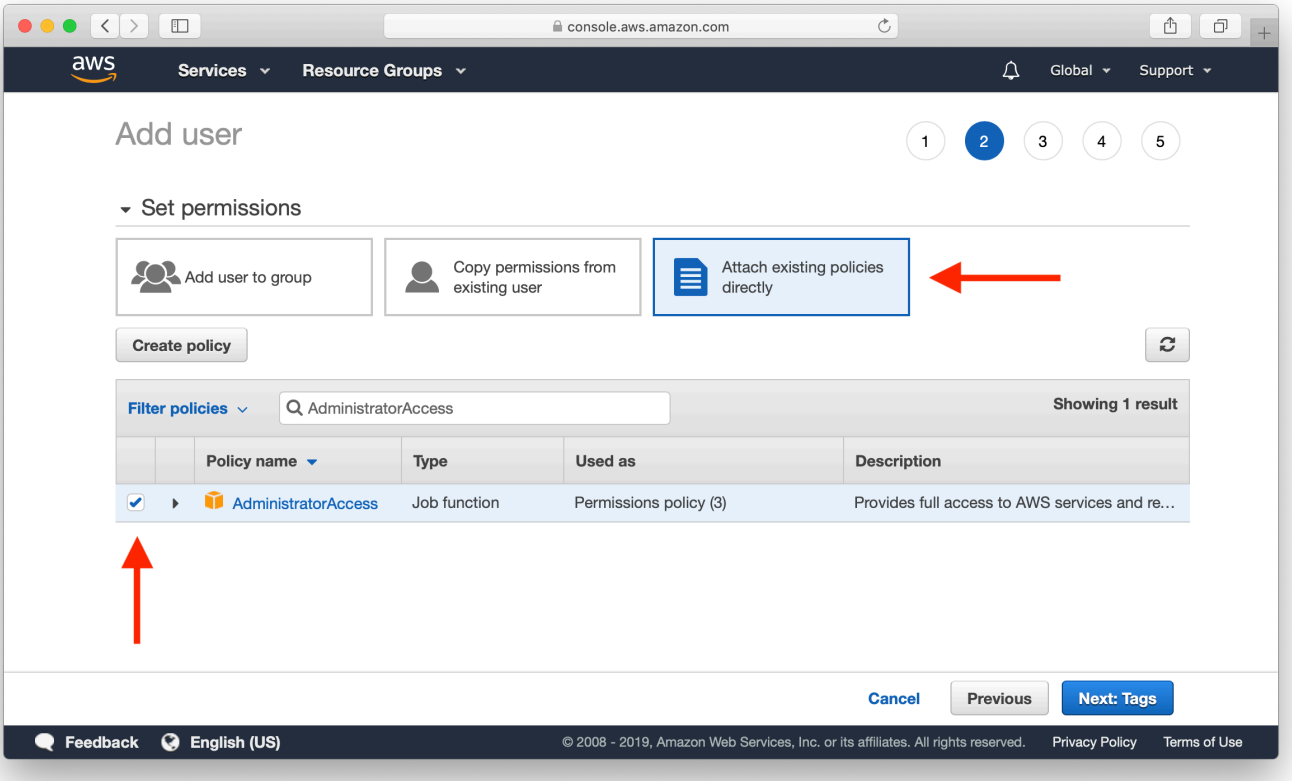
Select **Attach existing policies directly**.



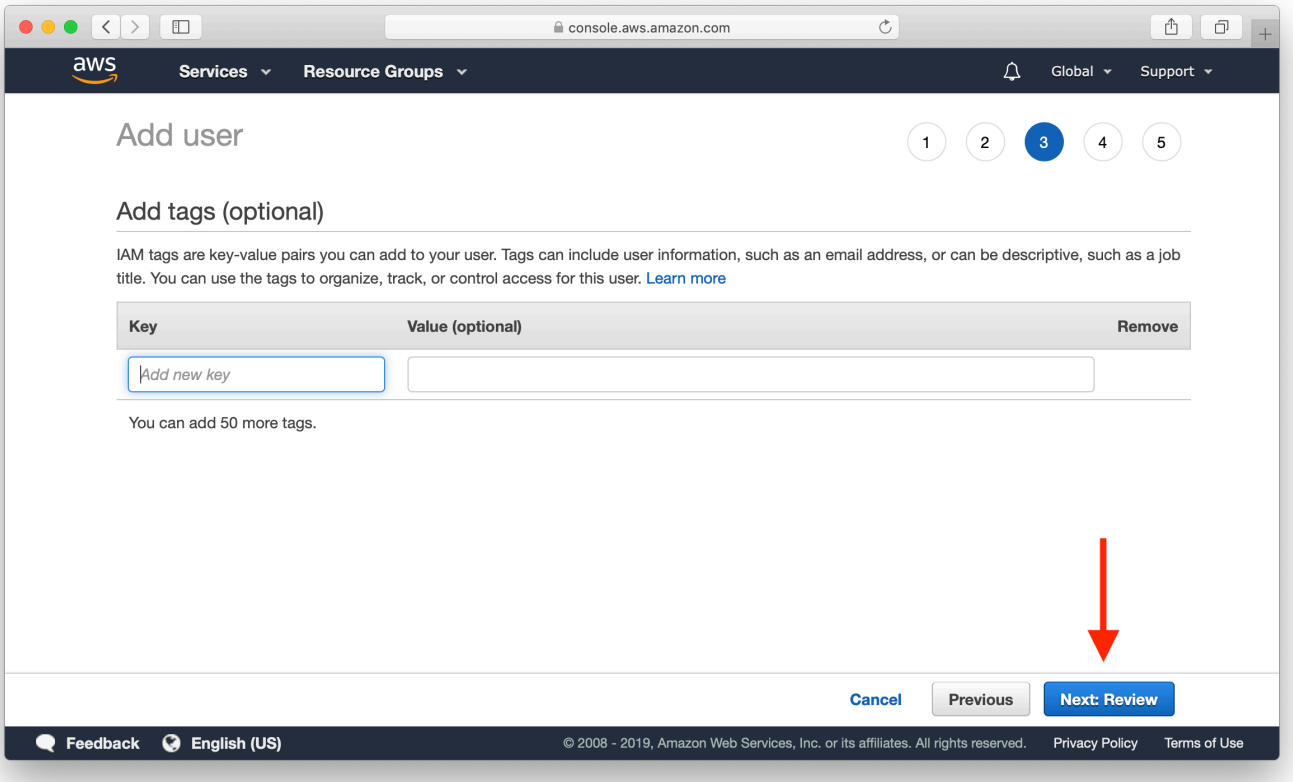
Search for **AdministratorAccess** and select the policy, then select **Next:**

Review.

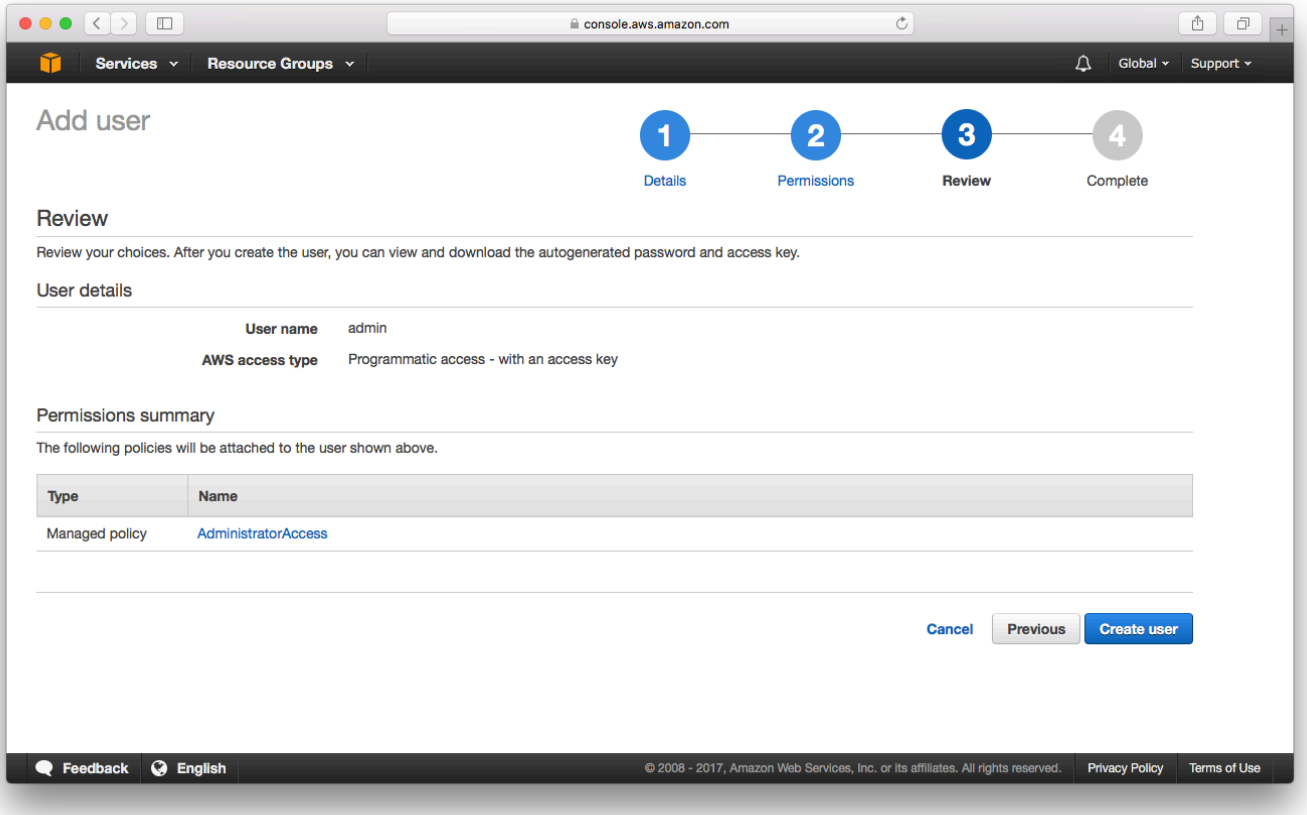
We can provide a more fine-grained policy here and we cover this later in the Customize the Serverless IAM Policy (/chapters/customize-the-serverless-iam-policy.html) chapter. But for now, let's continue with this.



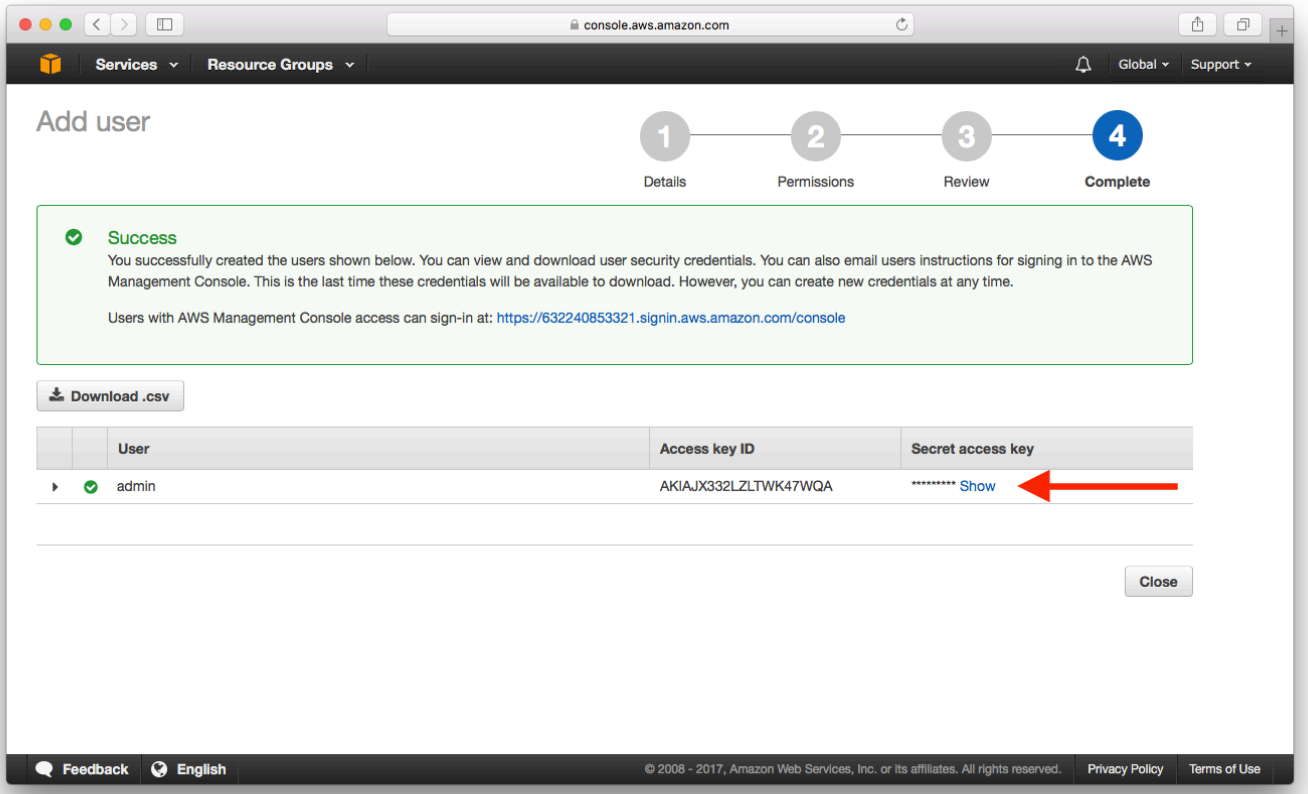
We can optionally add some info to our IAM user. But we'll skip this for now. Click **Next: Review**.



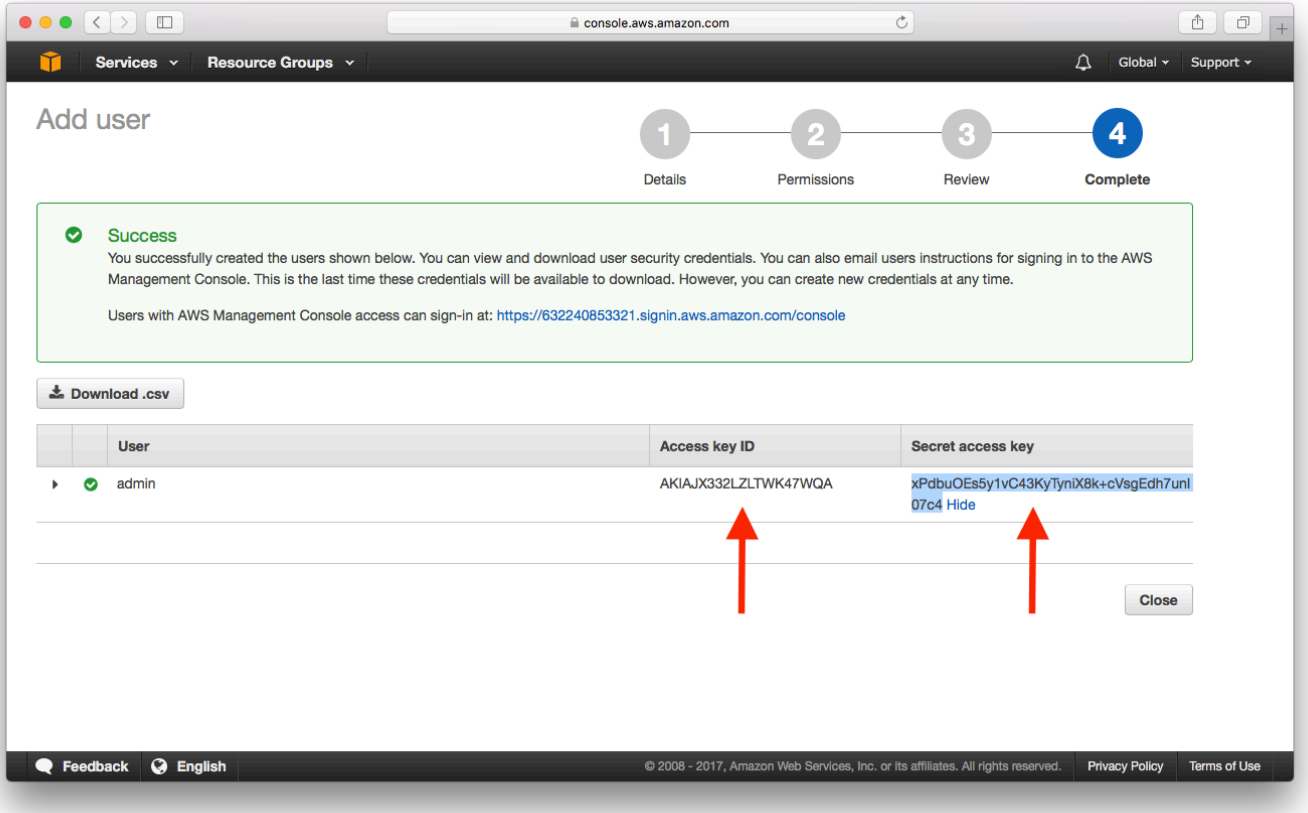
Select **Create user**.



Select **Show** to reveal **Secret access key**.



Take a note of the **Access key ID** and **Secret access key**. We will be needing this later.



The concept of IAM pops up very frequently when working with AWS services. So it is worth taking a better look at what IAM is and how it can help us secure our serverless setup.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-an-iam-user/92)

What is IAM

In the last chapter, we created an IAM user so that our AWS CLI can operate on our account without using the AWS Console. But the IAM concept is used very frequently when dealing with security for AWS services, so it is worth understanding it in a bit more detail. Unfortunately, IAM is made up of a lot of different parts and it can be very confusing for folks that first come across it. In this chapter we are going to take a look at IAM and its concepts in a bit more detail.

Let's start with the official definition of IAM.

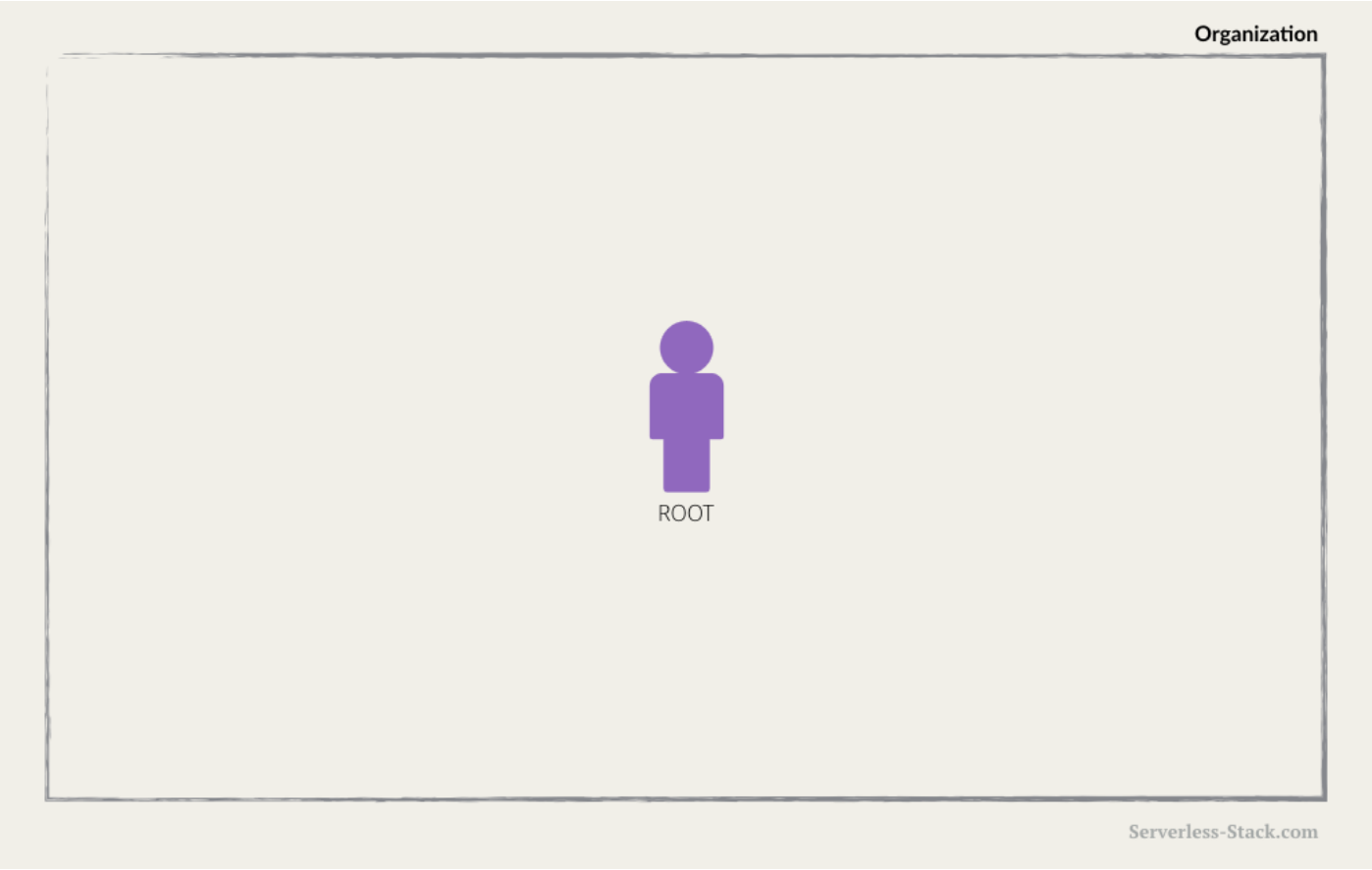
> *AWS Identity and Access Management (IAM) is a web service that helps you securely control access to AWS resources for your users. You use IAM to control who can use your AWS resources (authentication) and what resources they can use and in what ways (authorization).*

The first thing to notice here is that IAM is a service just like all the other services that AWS has. But in some ways it helps bring them all together in a secure way. IAM is made up of a few different parts, so let's start by looking at the first and most basic one.

What is an IAM User

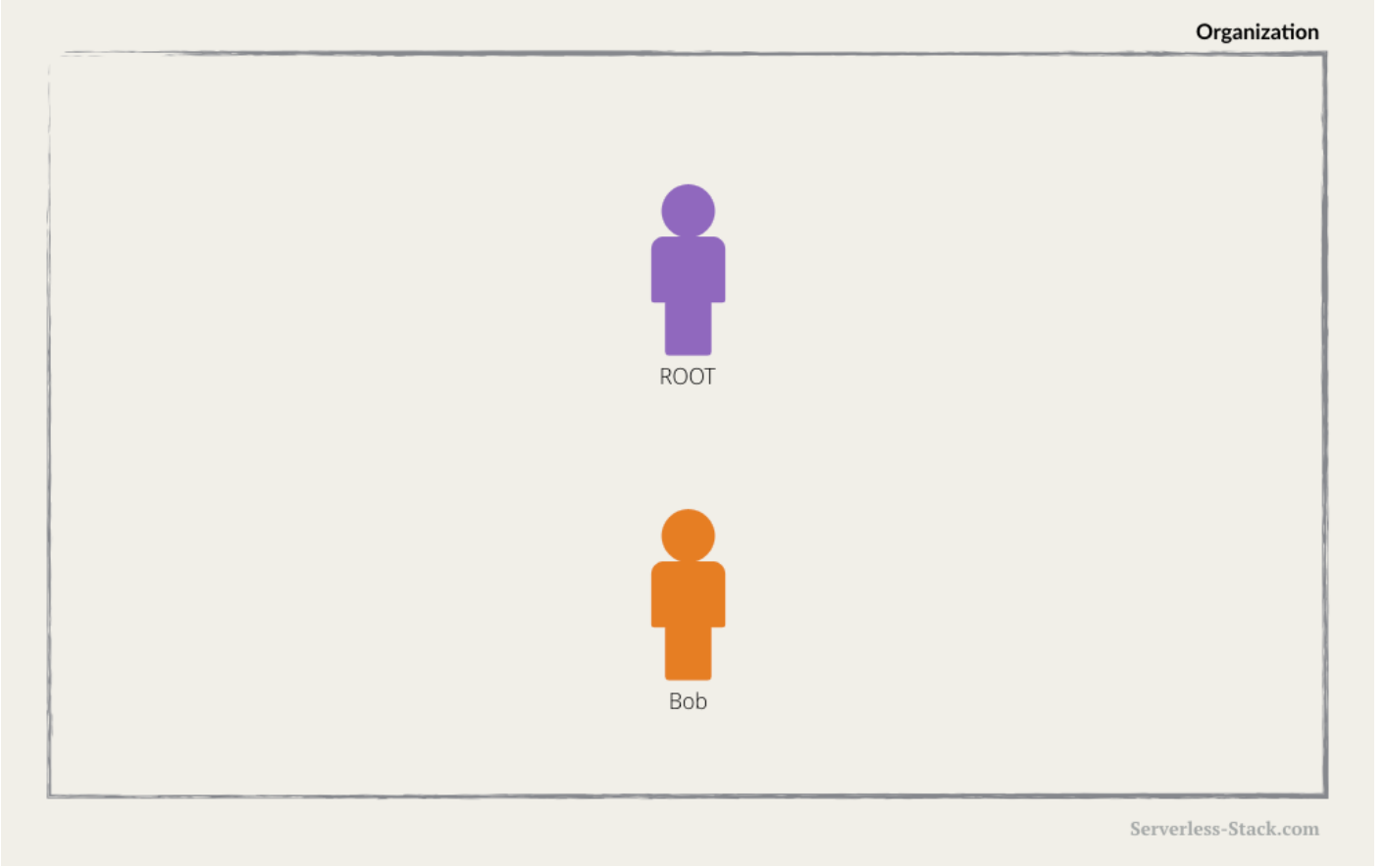
When you first create an AWS account, you are the root user. The email address and password you used to create the account is called your root account credentials. You can use them to sign in to the AWS Management

Console. When you do, you have complete, unrestricted access to all resources in your AWS account, including access to your billing information and the ability to change your password.



Though it is not a good practice to regularly access your account with this level of access, it is not a problem when you are the only person who works in your account. However, when another person needs to access and manage your AWS account, you definitely don't want to give out your root credentials. Instead you create an IAM user.

An IAM user consists of a name, a password to sign into the AWS Management Console, and up to two access keys that can be used with the API or CLI.



By default, users can't access anything in your account. You grant permissions to a user by creating a policy and attaching the policy to the user. You can grant one or more of these policies to restrict what the user can and cannot access.

What is an IAM Policy?

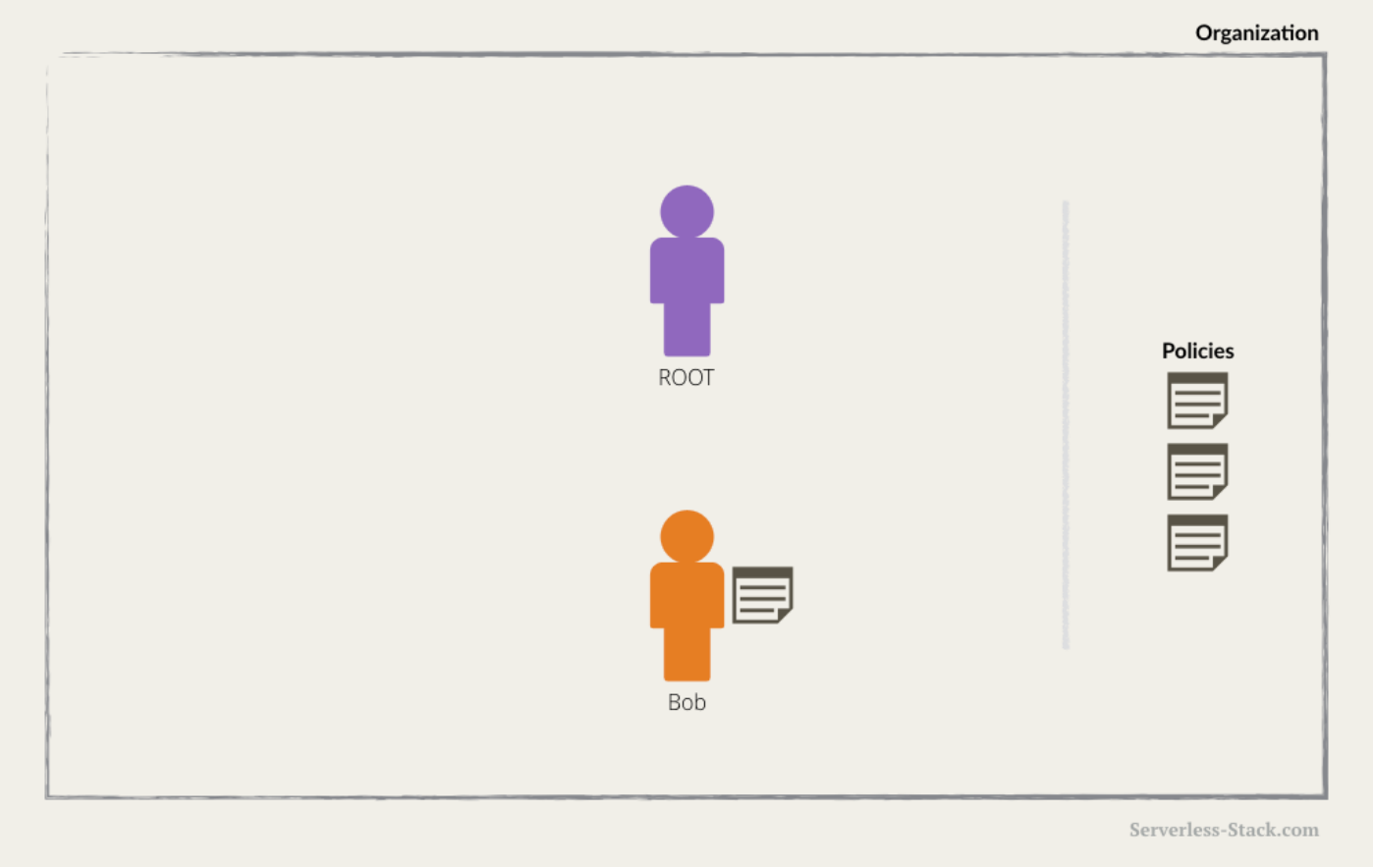
An IAM policy is a rule or set of rules defining the operations allowed/denied to be performed on an AWS resource.

Policies can be granted in a number of ways:

- Attaching a *managed policy*. AWS provides a list of pre-defined policies such as *AmazonS3ReadOnlyAccess*.
- Attaching an *inline policy*. An inline policy is a custom policy created by hand.
- Adding the user to a group that has appropriate permission policies

attached. We'll look at groups in detail below.

- Cloning the permission of an existing IAM user.



As an example, here is a policy that grants all operations to all S3 buckets.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": "s3:*",
    "Resource": "*"
  }
}
```

And here is a policy that grants more granular access, only allowing retrieval of files prefixed by the string `Bobs-` in the bucket called `Hello-bucket`.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": ["s3:GetObject"],
    "Resource": "arn:aws:s3:::Hello-bucket/*",
    "Condition": {"StringEquals": {"s3:prefix": "Bobs-
"}}
}
```

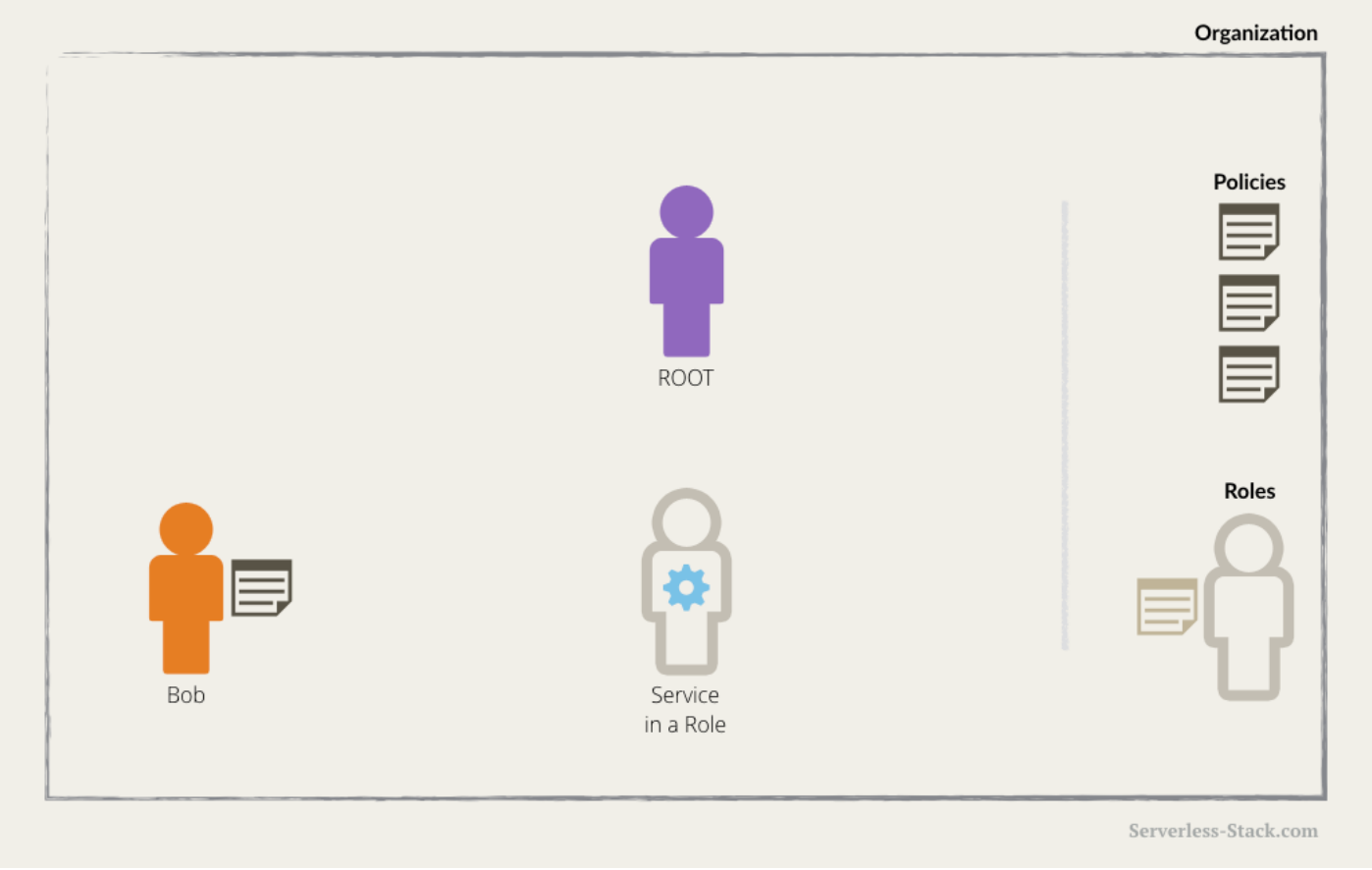
We are using S3 resources in the above examples. But a policy looks similar for any of the AWS services. It just depends on the resource ARN for `Resource` property. An ARN is an identifier for a resource in AWS and we'll look at it in more detail in the next chapter. We also add the corresponding service actions and condition context keys in `Action` and `Condition` property. You can find all the available AWS Service actions and condition context keys for use in IAM Policies here (https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_actionsconditions.html). Aside from attaching a policy to a user, you can attach them to a role or a group.

What is an IAM Role

Sometimes your AWS resources need to access other resources in your account. For example, you have a Lambda function that queries your DynamoDB to retrieve some data, process it, and then send Bob an email with the results. In this case, we want Lambda to only be able to make read queries so it does not change the database by mistake. We also want to restrict Lambda to be able to email Bob so it does not spam other people. While this could be done by creating an IAM user and putting the user's

credentials to the Lambda function or embed the credentials in the Lambda code, this is just not secure. If somebody was to get hold of these credentials, they could make those calls on your behalf. This is where IAM role comes in to play.

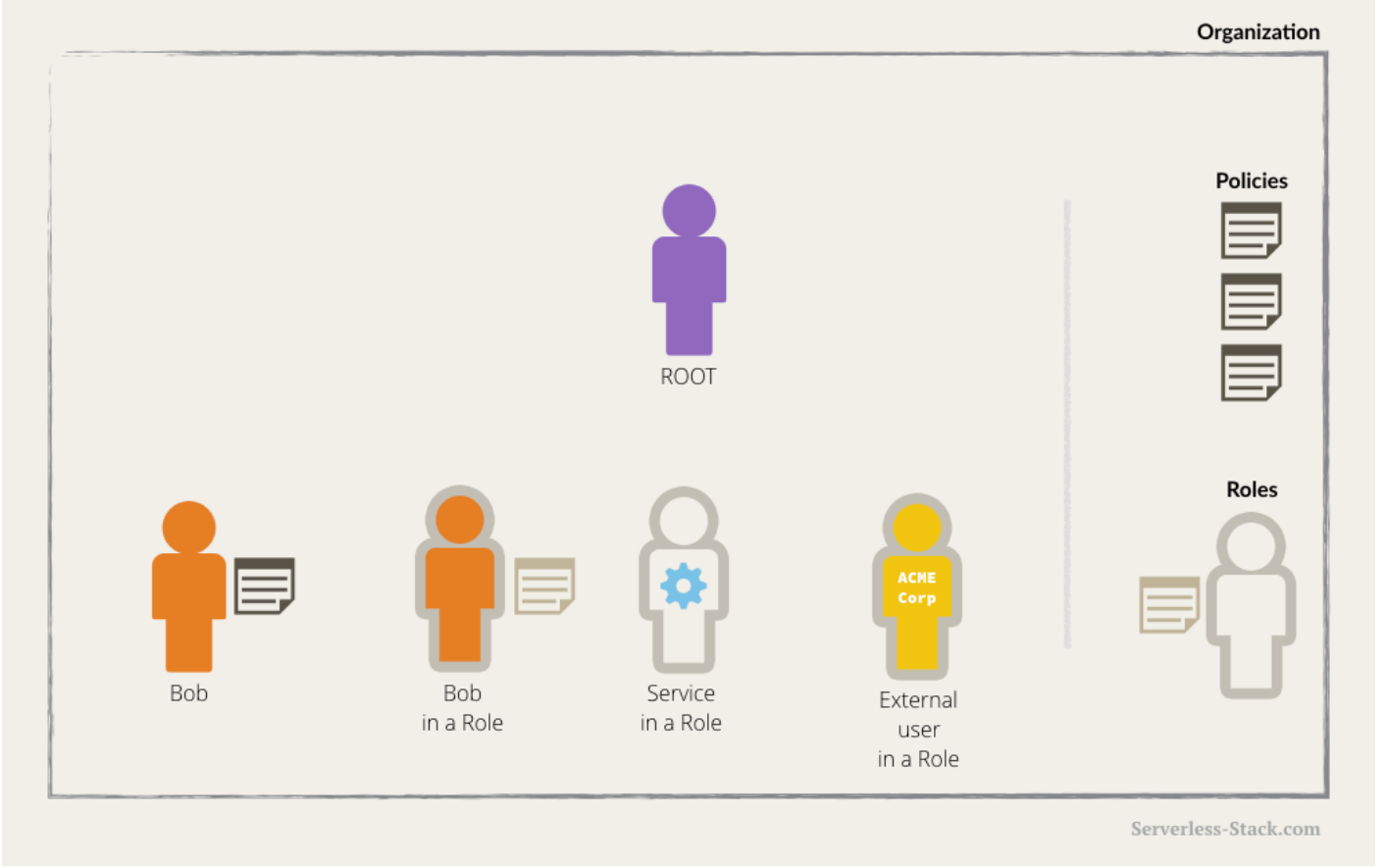
An IAM role is very similar to a user, in that it is an *identity* with permission policies that determine what the identity can and cannot do in AWS. However, a role does not have any credentials (password or access keys) associated with it. Instead of being uniquely associated with one person, a role can be taken on by anyone who needs it. In this case, the Lambda function will be assigned with a role to temporarily take on the permission.



Roles can be applied to users as well. In this case, the user is taking on the policy set for the IAM role. This is useful for cases where a user is wearing multiple “hats” in the organization. Roles make this easy since you only need to create these roles once and they can be re-used for anybody else that wants to take it on.

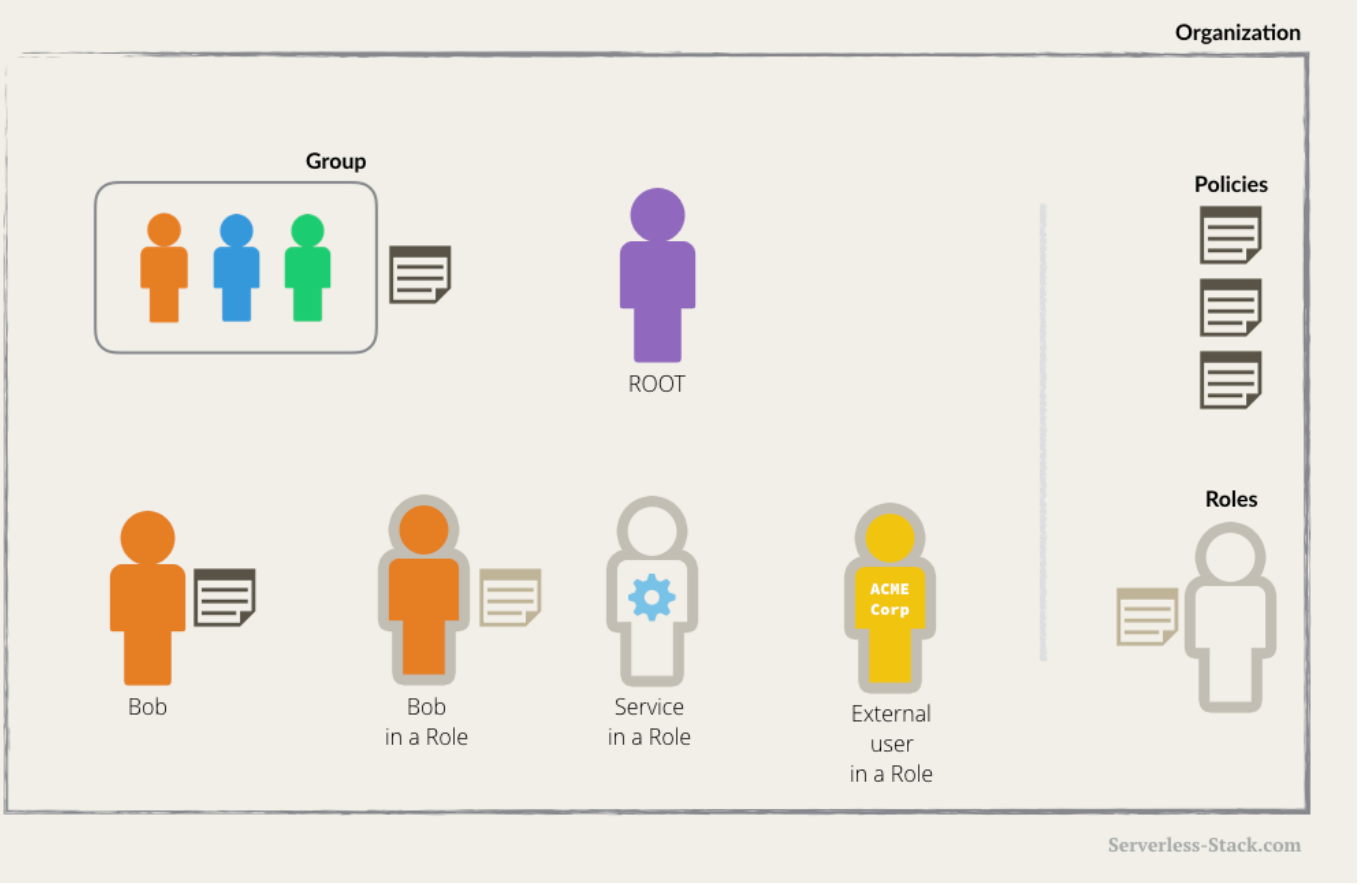


You can also have a role tied to the ARN of a user from a different organization. This allows the external user to assume that role as a part of your organization. This is typically used when you have a third party service that is acting on your AWS Organization. You'll be asked to create a Cross-Account IAM Role and add the external user as a *Trust Relationship*. The *Trust Relationship* is telling AWS that the specified external user can assume this role.



What is an IAM Group

An IAM group is simply a collection of IAM users. You can use groups to specify permissions for a collection of users, which can make those permissions easier to manage for those users. For example, you could have a group called Admins and give that group the types of permissions that administrators typically need. Any user in that group automatically has the permissions that are assigned to the group. If a new user joins your organization and should have administrator privileges, you can assign the appropriate permissions by adding the user to that group. Similarly, if a person changes jobs in your organization, instead of editing that user's permissions, you can remove him or her from the old groups and add him or her to the appropriate new groups.



This should give you a quick idea of IAM and some of its concepts. We will be referring to a few of these in the coming chapters. Next let's quickly look at another AWS concept; the ARN.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/what-is-iam/23)

What is an ARN

In the last chapter while we were looking at IAM policies we looked at how you can specify a resource using its ARN. Let's take a better look at what ARN is.

Here is the official definition:

> *Amazon Resource Names (ARNs) uniquely identify AWS resources. We require an ARN when you need to specify a resource unambiguously across all of AWS, such as in IAM policies, Amazon Relational Database Service (Amazon RDS) tags, and API calls.*

ARN is really just a globally unique identifier for an individual AWS resource. It takes one of the following formats.

```
arn:partition:service:region:account-id:resource
arn:partition:service:region:account-
id:resourcetype/resource
arn:partition:service:region:account-
id:resourcetype:resource
```

Let's look at some examples of ARN. Note the different formats used.

```
<!-- Elastic Beanstalk application version -->
arn:aws:elasticbeanstalk:us-east-
1:123456789012:environment/My App/MyEnvironment
```

```
<!-- IAM user name -->
arn:aws:iam::123456789012:user/David

<!-- Amazon RDS instance used for tagging -->
arn:aws:rds:eu-west-1:123456789012:db:mysql-db

<!-- Object in an Amazon S3 bucket -->
arn:aws:s3:::my_corporate_bucket/exampleobject.png
```

Finally, let's look at the common use cases for ARN.

1. Communication

   ARN is used to reference a specific resource when you orchestrate a system involving multiple AWS resources. For example, you have an API Gateway listening for RESTful APIs and invoking the corresponding Lambda function based on the API path and request method. The routing looks like the following.

   ```
   GET /hello_world => arn:aws:lambda:us-east-
   1:123456789012:function:lambda-hello-world
   ```

2. IAM Policy

   We had looked at this in detail in the last chapter but here is an example of a policy definition.

   ```
   {
     "Version": "2012-10-17",
     "Statement": {
       "Effect": "Allow",
       "Action": ["s3:GetObject"],
   ```

```
    "Resource": "arn:aws:s3:::Hello-bucket/*"
}
```

ARN is used to define which resource (S3 bucket in this case) the access is granted for. The wildcard `*` character is used here to match all resources inside the *Hello-bucket*.

Next let's configure our AWS CLI. We'll be using the info from the IAM user account we created previously.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/what-is-an-arn/34)

Configure the AWS CLI

To make it easier to work with a lot of the AWS services, we are going to use the AWS CLI (https://aws.amazon.com/cli/).

Install the AWS CLI

AWS CLI needs Python 2 version 2.6.5+ or Python 3 version 3.3+ and Pip (https://pypi.python.org/pypi/pip). Use the following if you need help installing Python or Pip.

- Installing Python (https://www.python.org/downloads/)
- Installing Pip (https://pip.pypa.io/en/stable/installing/)

✦ CHANGE Now using Pip you can install the AWS CLI (on Linux, macOS, or Unix) by running:

```
$ sudo pip install awscli
```

Or using Homebrew (https://brew.sh) on macOS:

```
$ brew install awscli
```

If you are having some problems installing the AWS CLI or need Windows install instructions, refer to the complete install instructions (http://docs.aws.amazon.com/cli/latest/userguide/installing.html).

Add Your Access Key to AWS CLI

We now need to tell the AWS CLI to use your Access Keys from the previous chapter.

It should look something like this:

- Access key ID **AKIAIOSFODNN7EXAMPLE**
- Secret access key **wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY**

CHANGE Simply run the following with your Secret Key ID and your Access Key.

```
$ aws configure
```

You can leave the **Default region name** and **Default output format** the way they are.

Next let's get started with setting up our backend.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/configure-the-aws-cli/86)

Create a DynamoDB Table

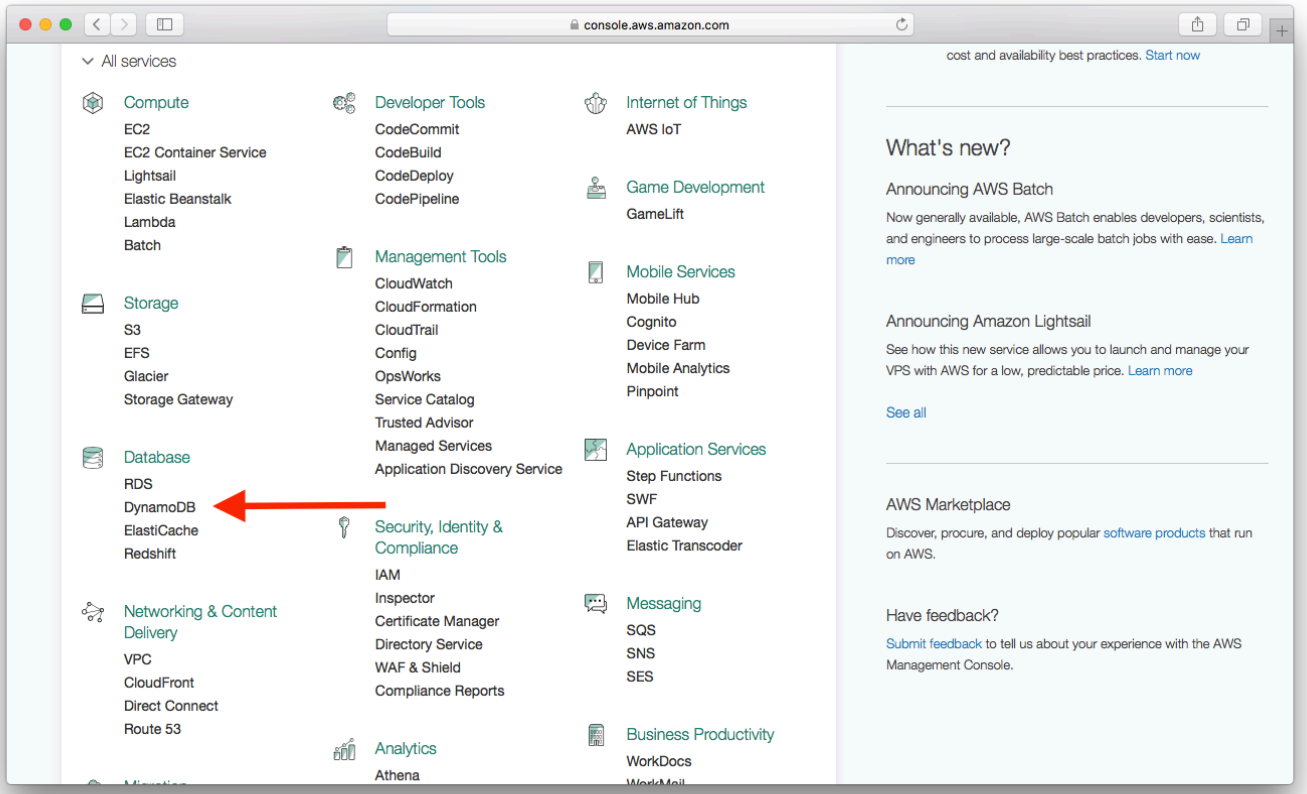
To build the backend for our notes app, it makes sense that we first start by thinking about how the data is going to be stored. We are going to use DynamoDB (https://aws.amazon.com/dynamodb/) to do this.

About DynamoDB

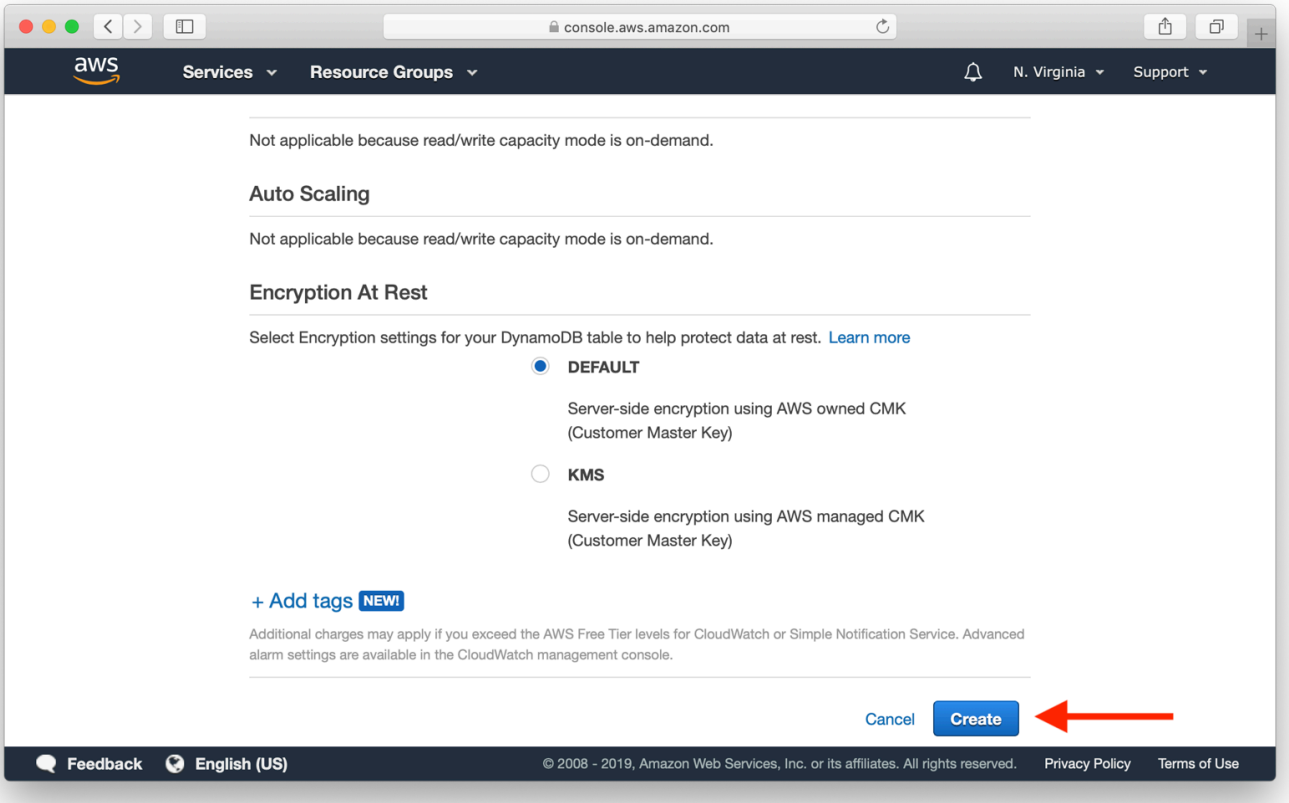
Amazon DynamoDB is a fully managed NoSQL database that provides fast and predictable performance with seamless scalability. Similar to other databases, DynamoDB stores data in tables. Each table contains multiple items, and each item is composed of one or more attributes. We are going to cover some basics in the following chapters. But to get a better feel for it, here is a great guide on DynamoDB (https://www.dynamodbguide.com).

Create Table

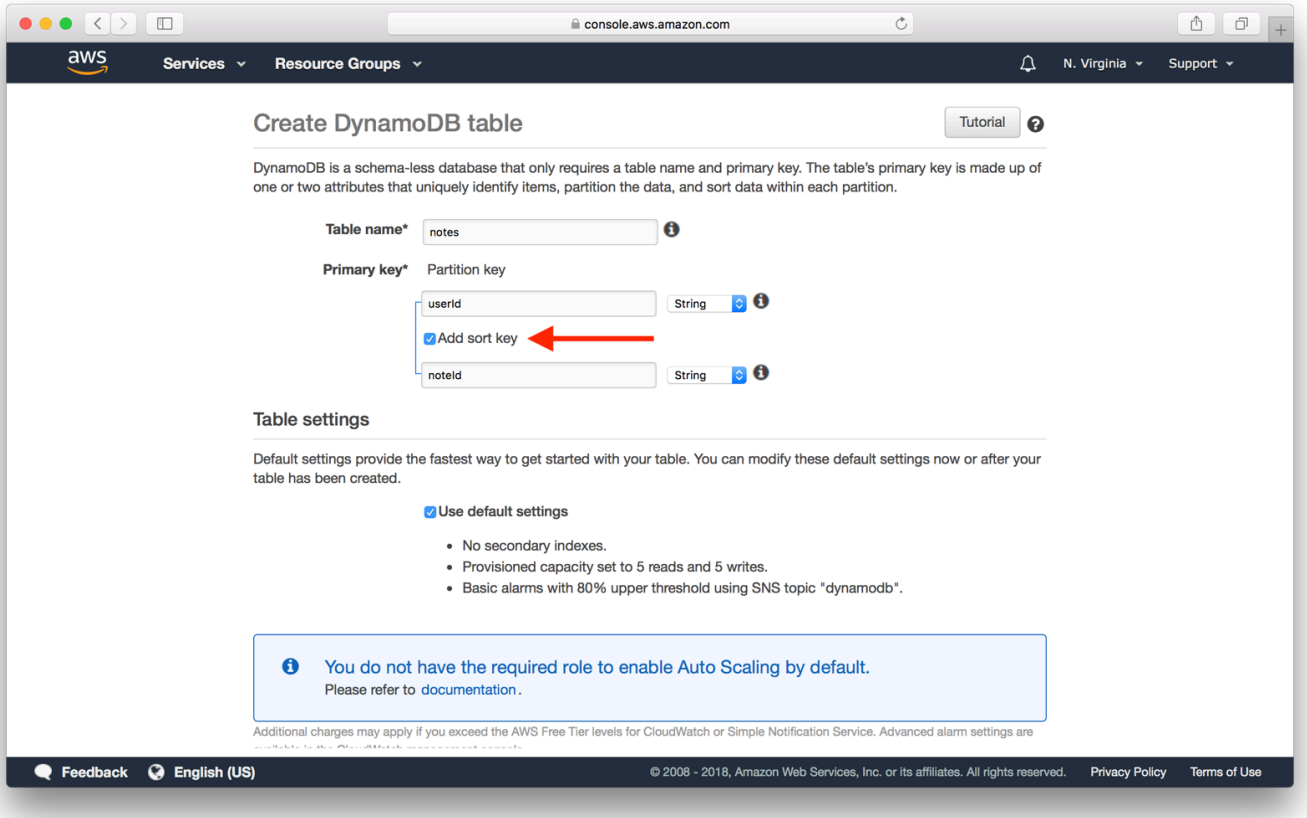
First, log in to your AWS Console (https://console.aws.amazon.com) and select **DynamoDB** from the list of services.



Select **Create table.**



Enter the **Table name** and **Primary key** info as shown below. Just make sure that `userId` and `noteId` are in camel case.



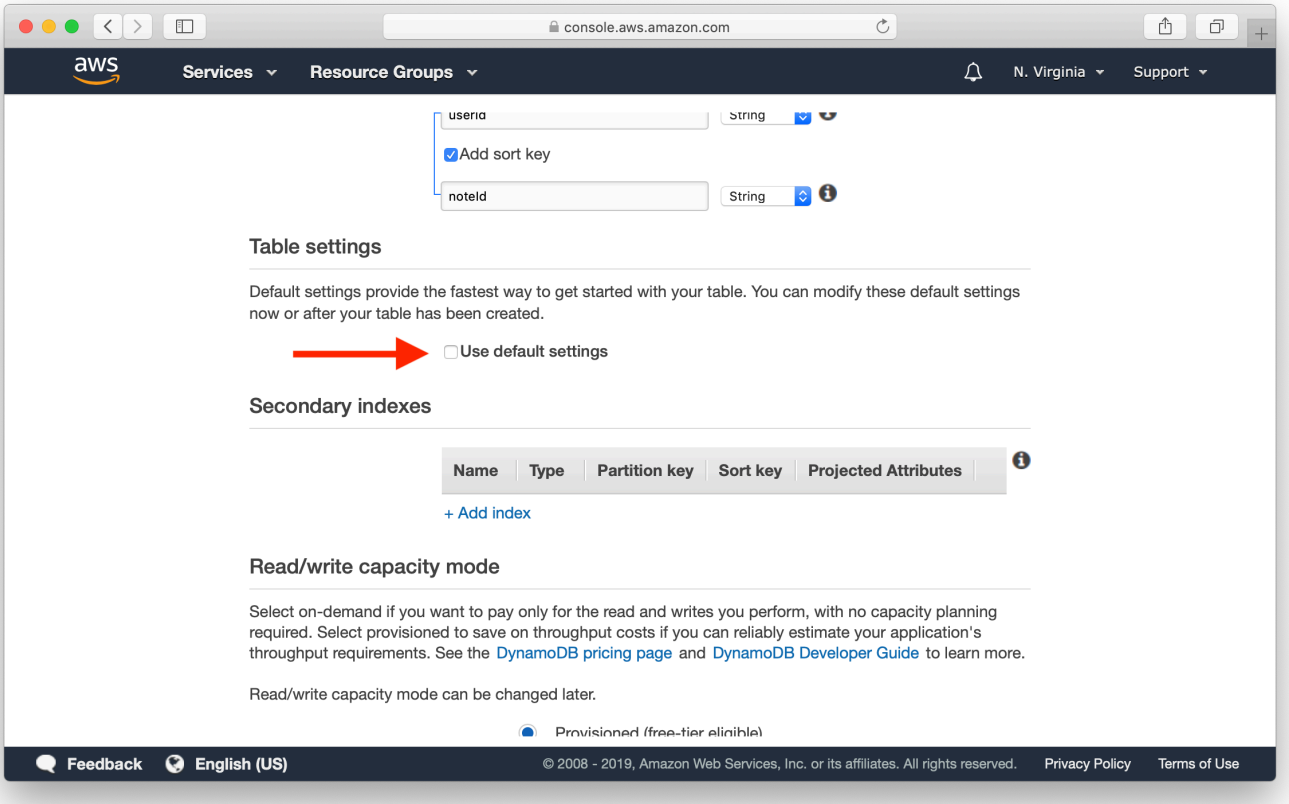
Each DynamoDB table has a primary key, which cannot be changed once set. The primary key uniquely identifies each item in the table, so that no two items can have the same key. DynamoDB supports two different kinds of primary keys:

- Partition key
- Partition key and sort key (composite)

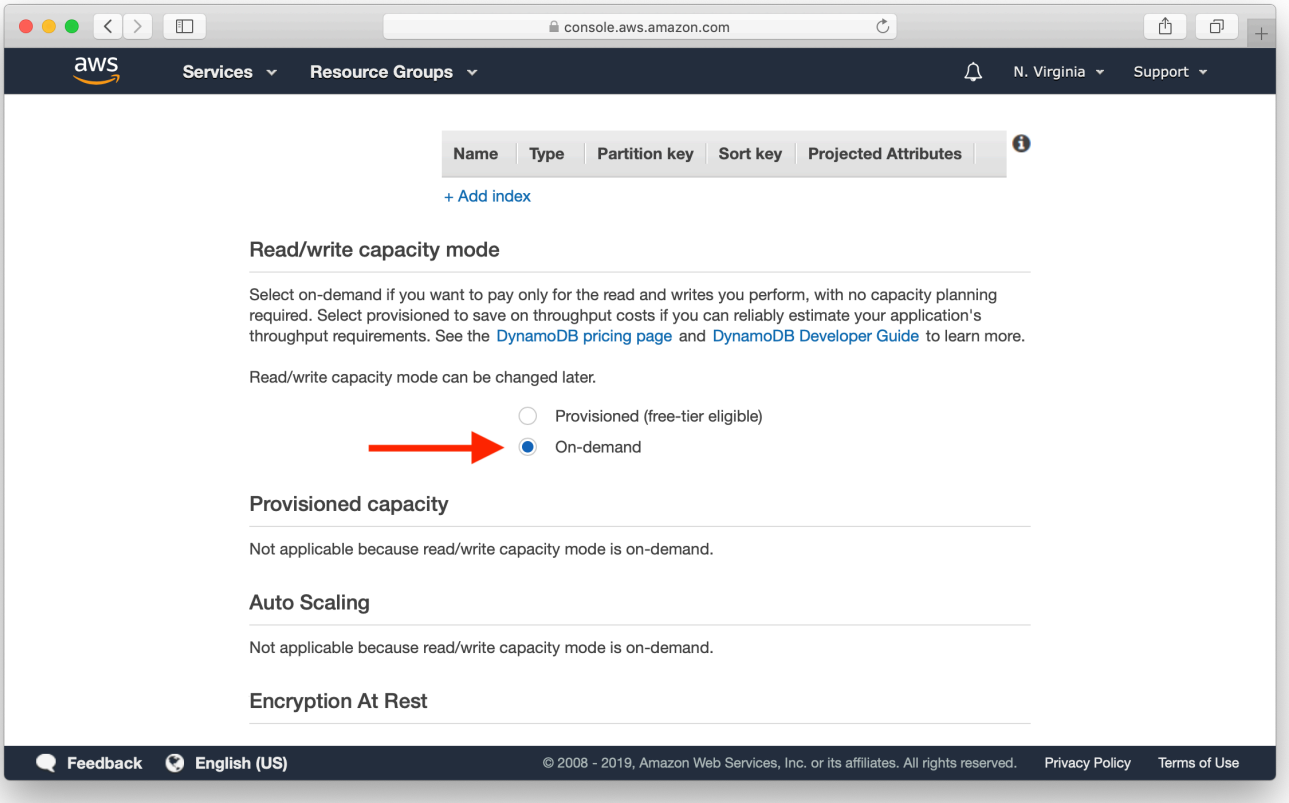
We are going to use the composite primary key which gives us additional flexibility when querying the data. For example, if you provide only the value for `userId`, DynamoDB would retrieve all of the notes by that user. Or you could provide a value for `userId` and a value for `noteId`, to retrieve a particular note.

To further your understanding of how indexes work in DynamoDB, you can read more here: DynamoDB Core Components (http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html)

Next scroll down and deselect **Use default settings**.

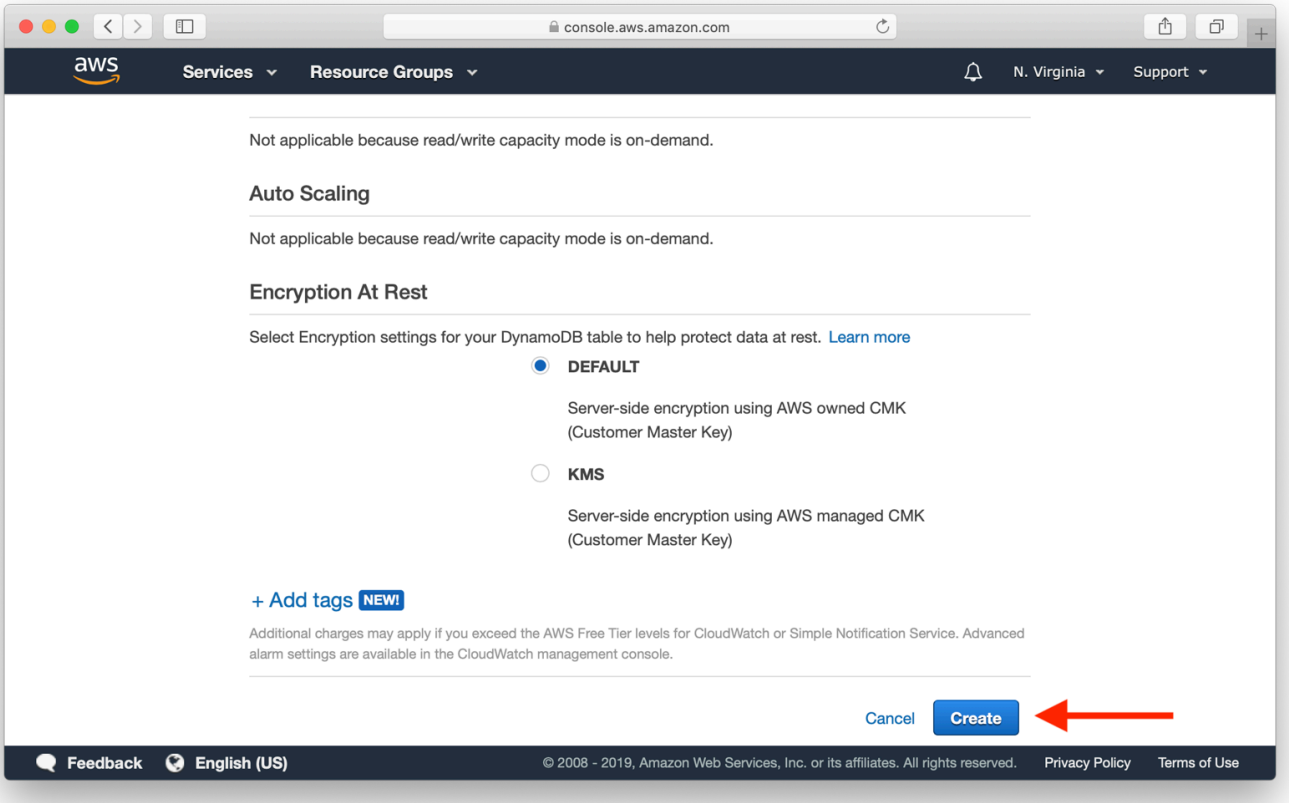


Scroll down further and select **On-demand** instead of **Provisioned**.

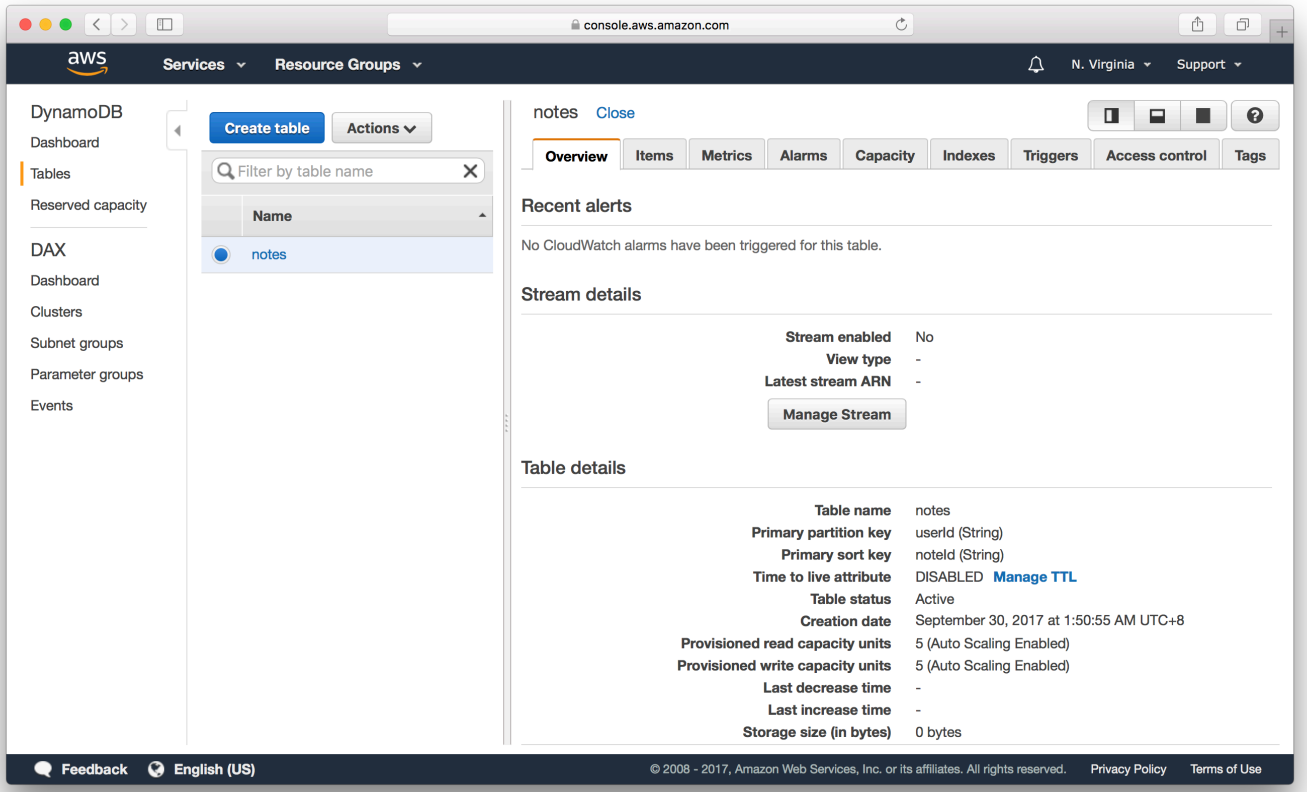


On-Demand Capacity (https://aws.amazon.com/dynamodb/pricing/on-demand/) is DynamoDB's pay per request mode. For workloads that are not predictable or if you are just starting out, this ends up being a lot cheaper than the Provisioned Capacity (https://aws.amazon.com/dynamodb/pricing/provisioned/) mode.

Finally, scroll down and hit **Create**.



The `notes` table has now been created. If you find yourself stuck with the **Table is being created** message; refresh the page manually.



It is also a good idea to set up backups for your DynamoDB table, especially if you are planning to use it in production. We cover this in an extra-credit chapter, Backups in DynamoDB (/chapters/backups-in-dynamodb.html).

Next, we'll set up an S3 bucket to handle file uploads.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-dynamodb-table/139)

Create an S3 Bucket for File Uploads

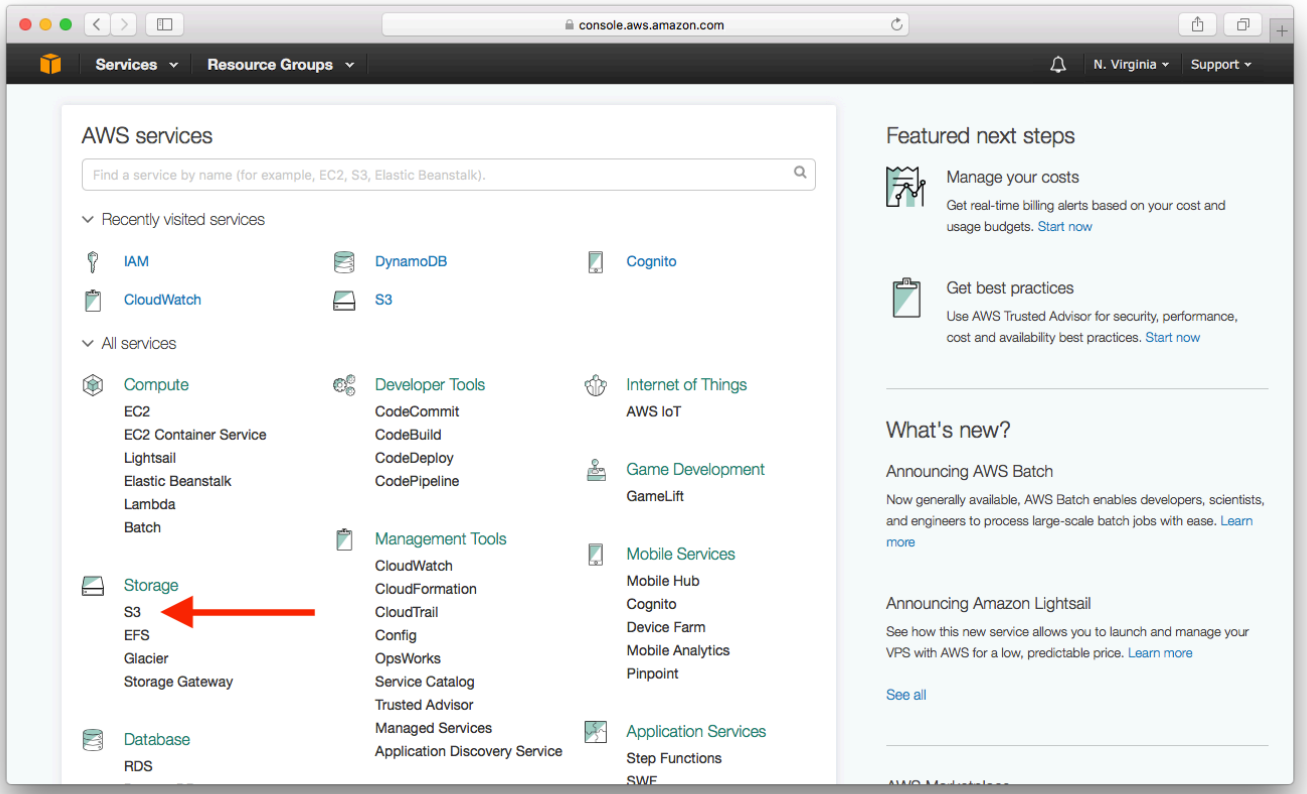
Now that we have our database table ready; let's get things set up for handling file uploads. We need to handle file uploads because each note can have an uploaded file as an attachment.

Amazon S3 (https://aws.amazon.com/s3/) (Simple Storage Service) provides storage service through web services interfaces like REST. You can store any object in S3 including images, videos, files, etc. Objects are organized into buckets, and identified within each bucket by a unique, user-assigned key.

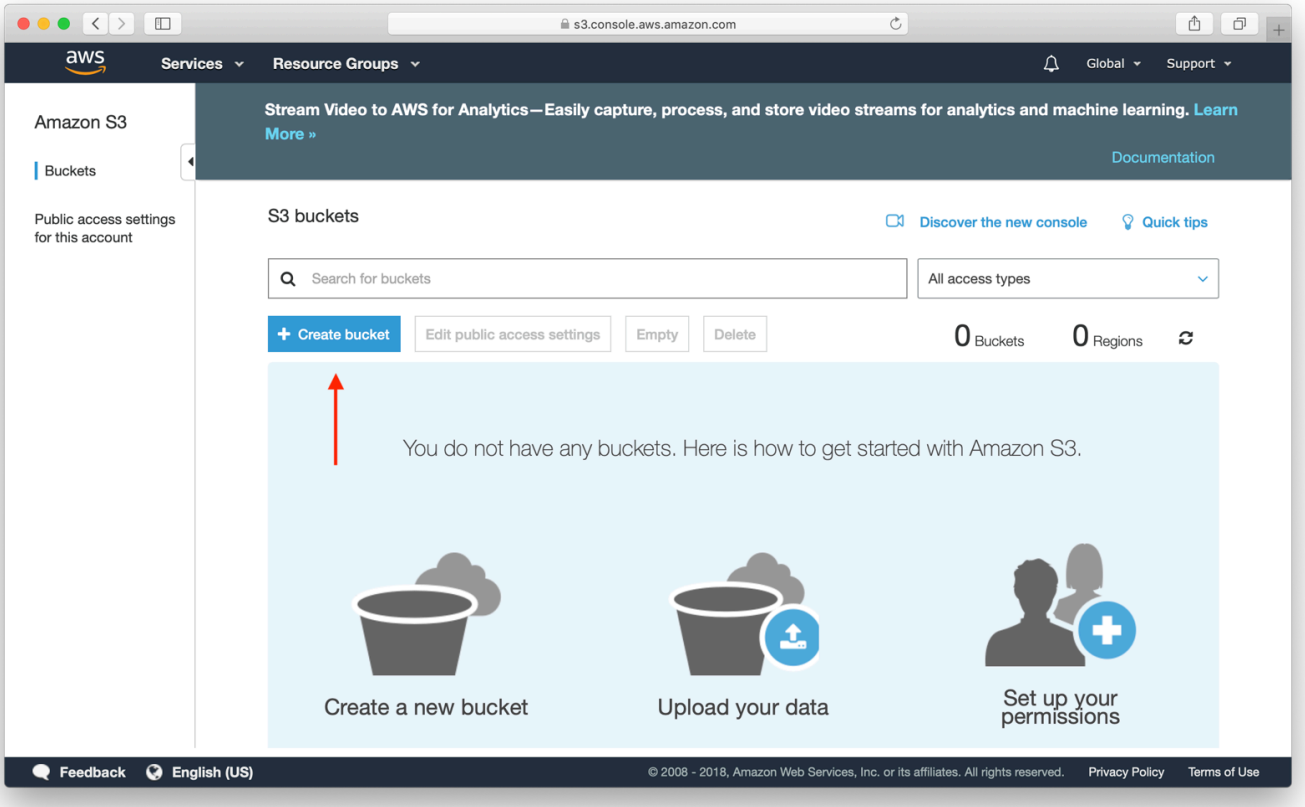
In this chapter, we are going to create an S3 bucket which will be used to store user uploaded files from our notes app.

Create Bucket

First, log in to your AWS Console (https://console.aws.amazon.com) and select **S3** from the list of services.



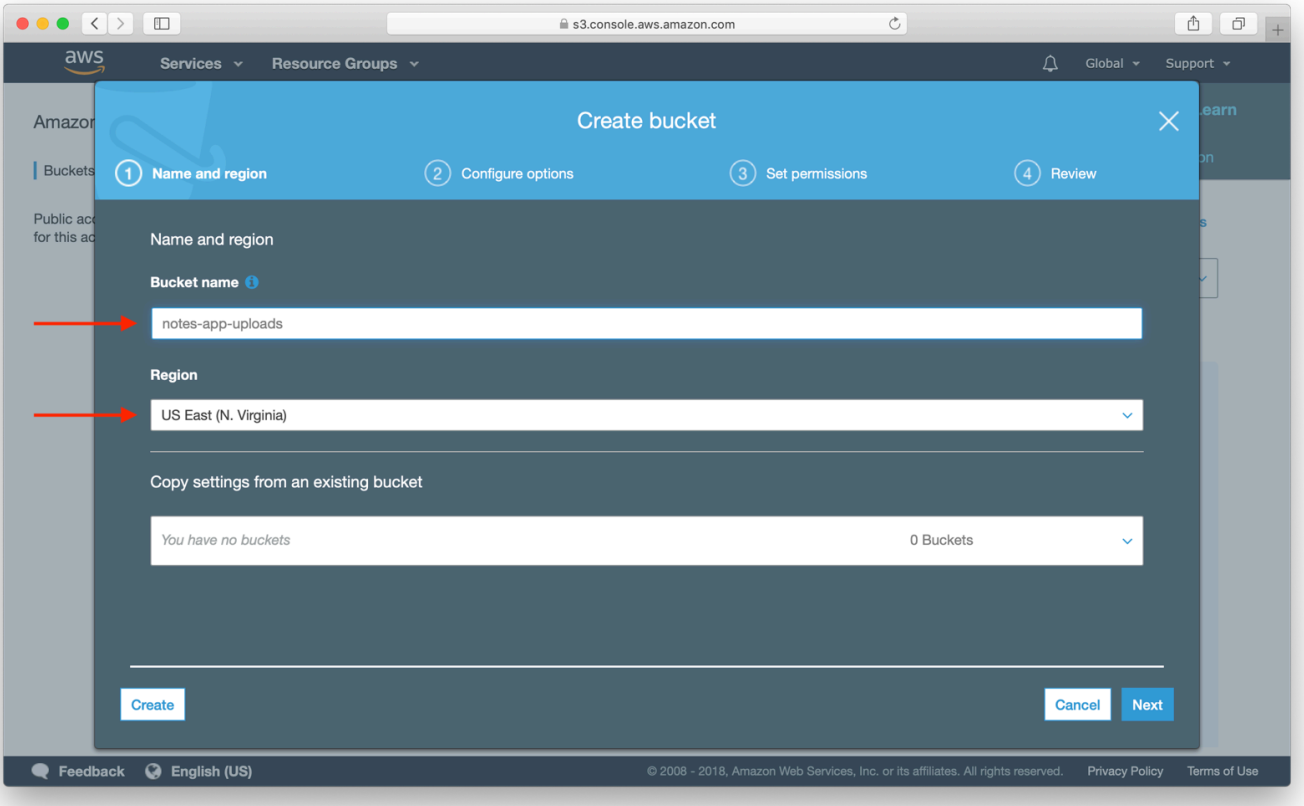
Select **Create bucket.**



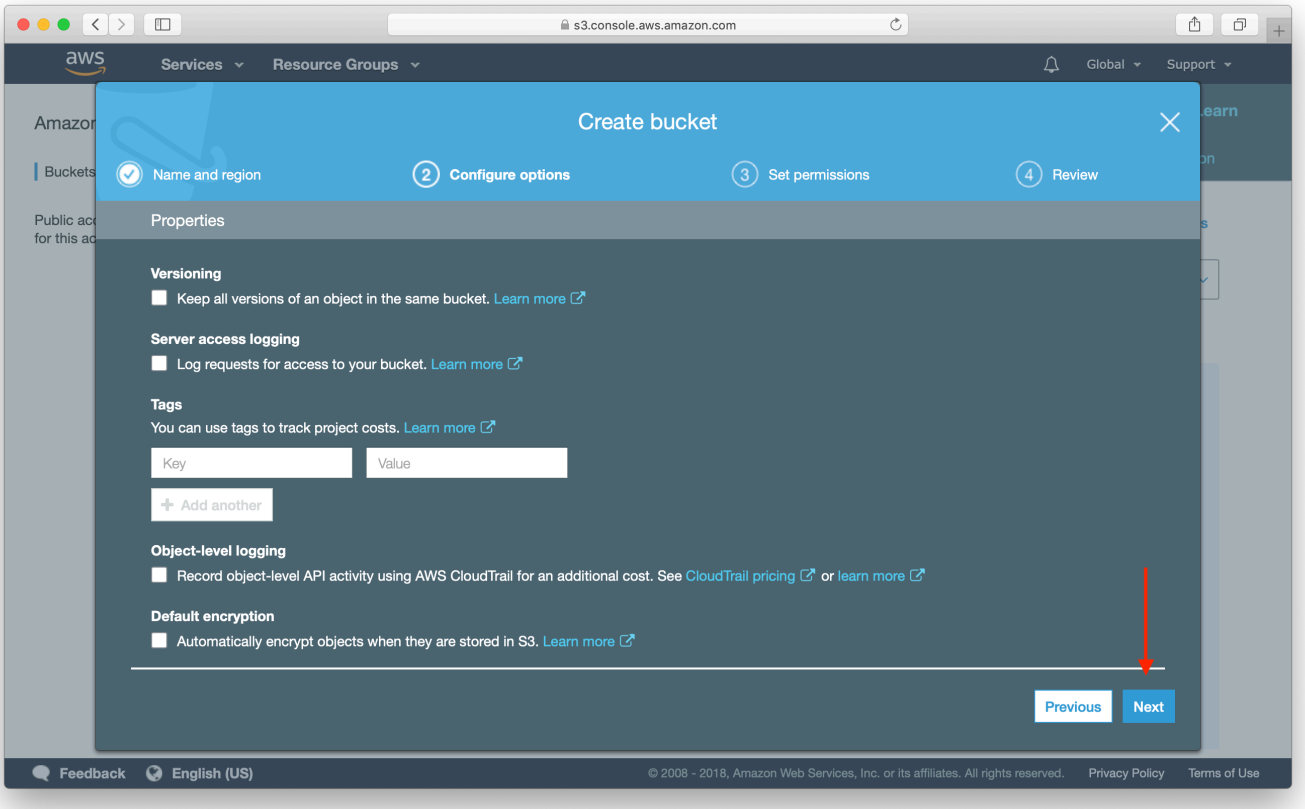
Pick a name of the bucket and select a region. Then select **Create**.

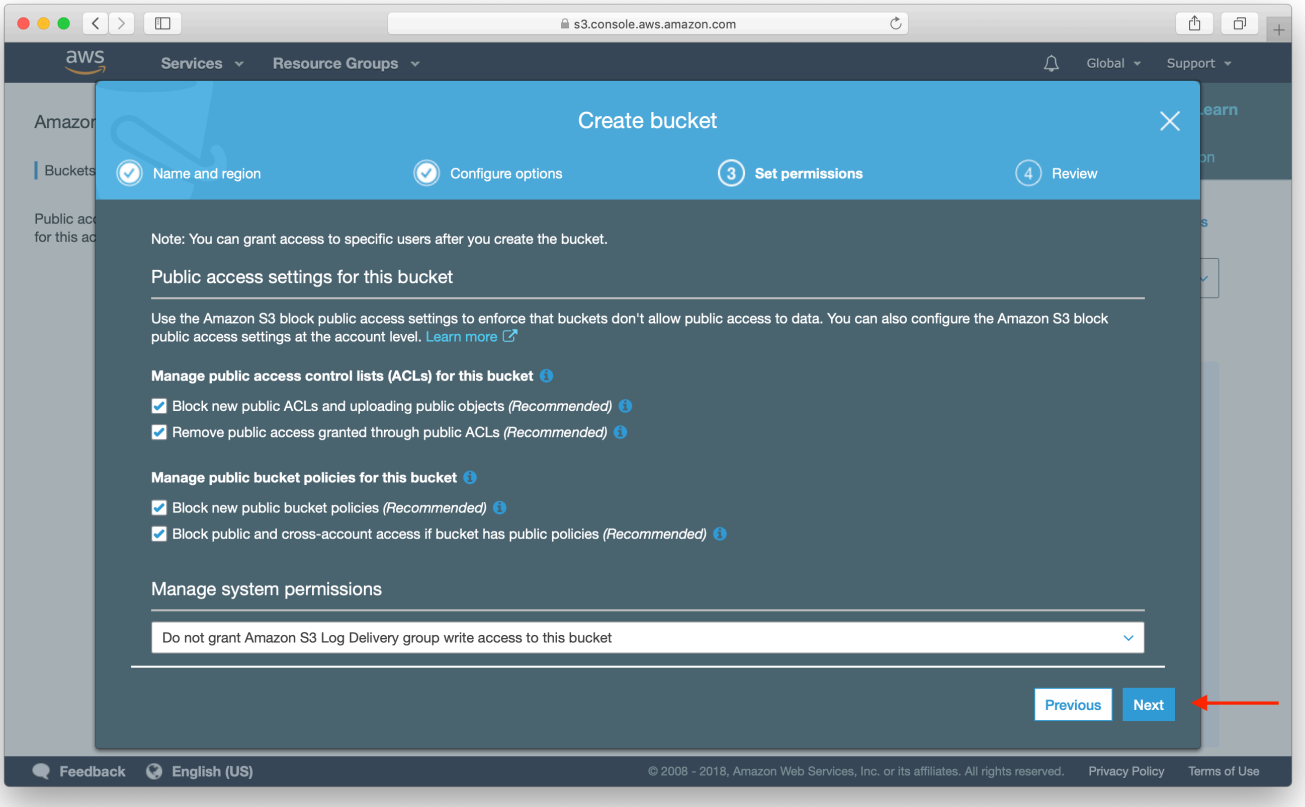
- **Bucket names** are globally unique, which means you cannot pick the same name as this tutorial.
- **Region** is the physical geographical region where the files are stored. We will use **US East (N. Virginia)** for this guide.

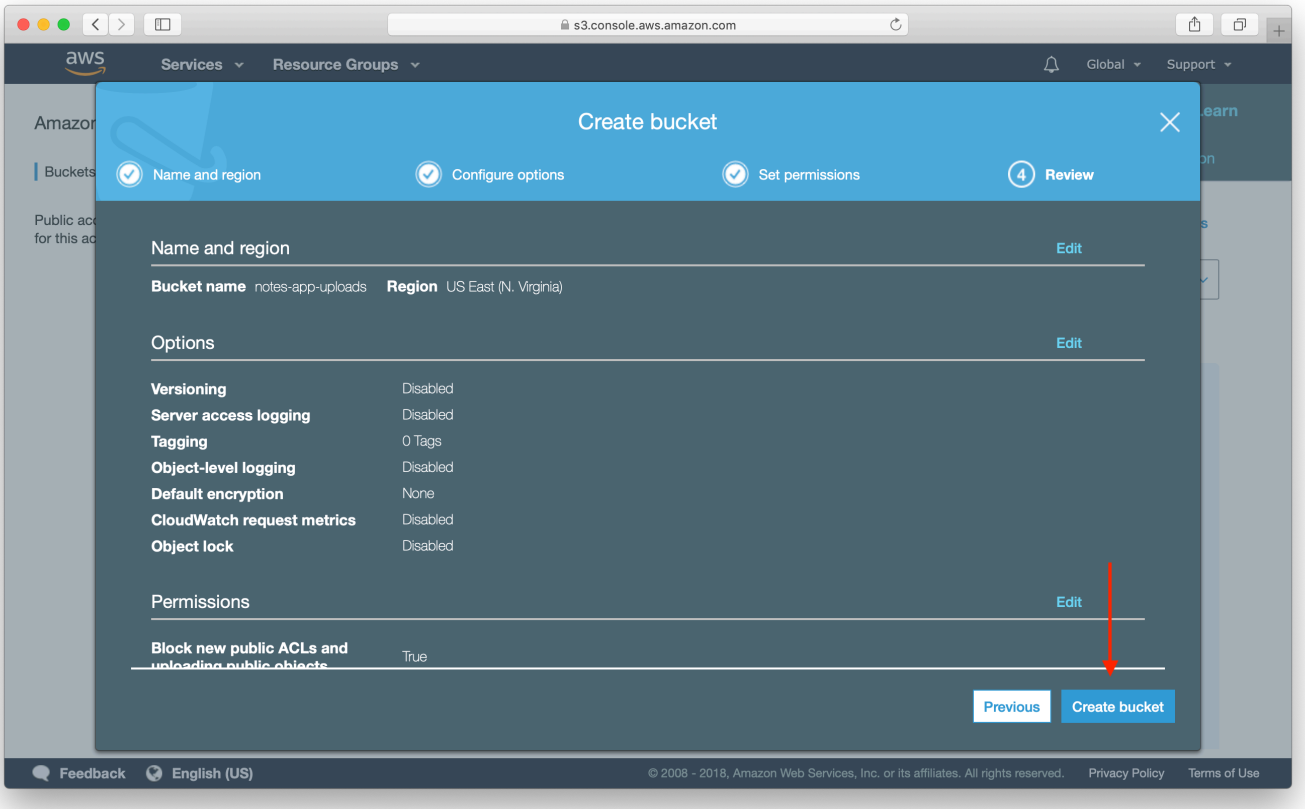
Make a note of the name and region as we'll be using it later in the guide.



Step through the next steps and leave the defaults by clicking **Next**, and then click **Create bucket** on the last step.


s3.console.aws.amazon.com
Services
Resource Groups
Global
Support
Create bucket
Name and region
2 Configure options
3 Set permissions
4 Review
Properties
Versioning
Keep all versions of an object in the same bucket. Learn more
Server access logging
Log requests for access to your bucket. Learn more
Tags
You can use tags to track project costs. Learn more
Key
Value
Add another
Object-level logging
Record object-level API activity using AWS CloudTrail for an additional cost. See CloudTrail pricing or learn more
Default encryption
Automatically encrypt objects when they are stored in S3. Learn more
Previous
Next
Feedback
English (US)
© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

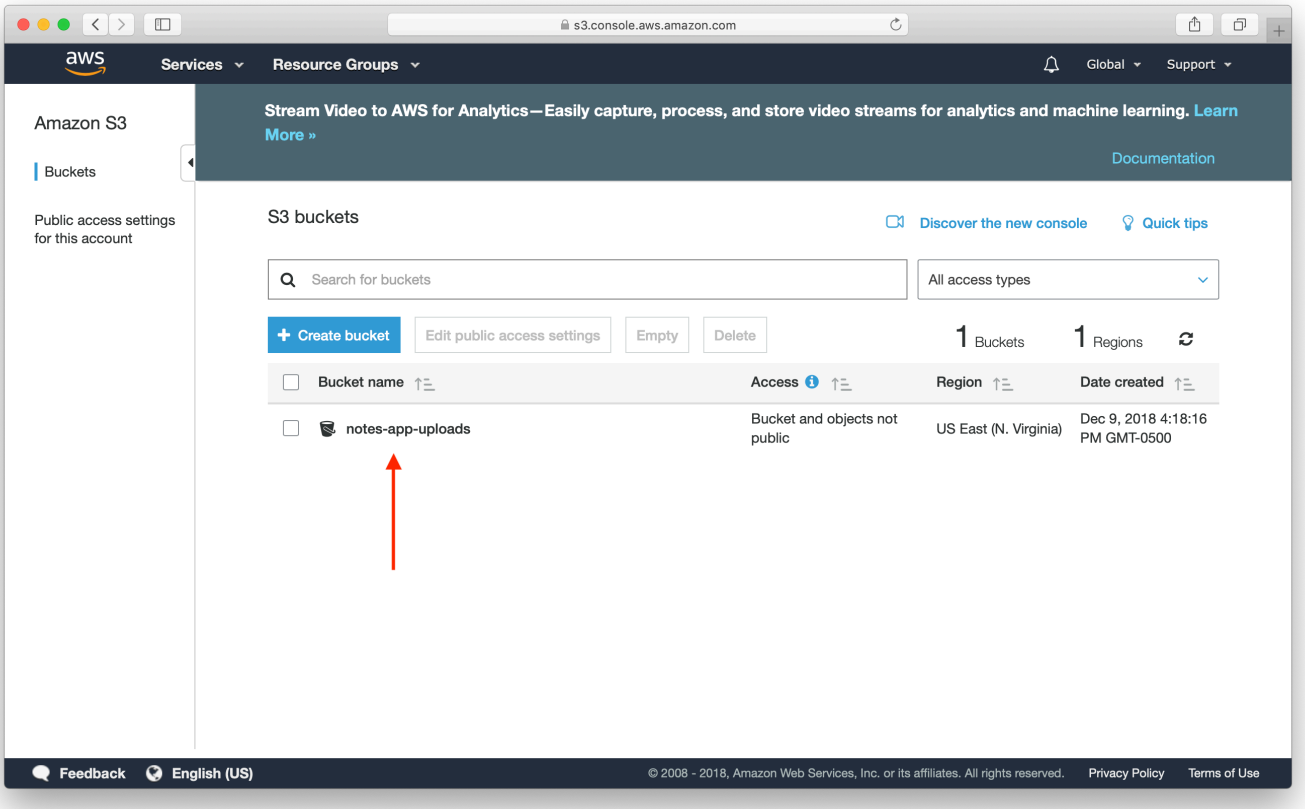

s3.console.aws.amazon.com
Services
Resource Groups
Global
Support
Create bucket
Name and region
Configure options
3 Set permissions
4 Review
Note: You can grant access to specific users after you create the bucket.
Public access settings for this bucket
Use the Amazon S3 block public access settings to enforce that buckets don't allow public access to data. You can also configure the Amazon S3 block public access settings at the account level. Learn more
Manage public access control lists (ACLs) for this bucket
Block new public ACLs and uploading public objects (Recommended)
Remove public access granted through public ACLs (Recommended)
Manage public bucket policies for this bucket
Block new public bucket policies (Recommended)
Block public and cross-account access if bucket has public policies (Recommended)
Manage system permissions
Do not grant Amazon S3 Log Delivery group write access to this bucket
Previous
Next
Feedback
English (US)
© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use



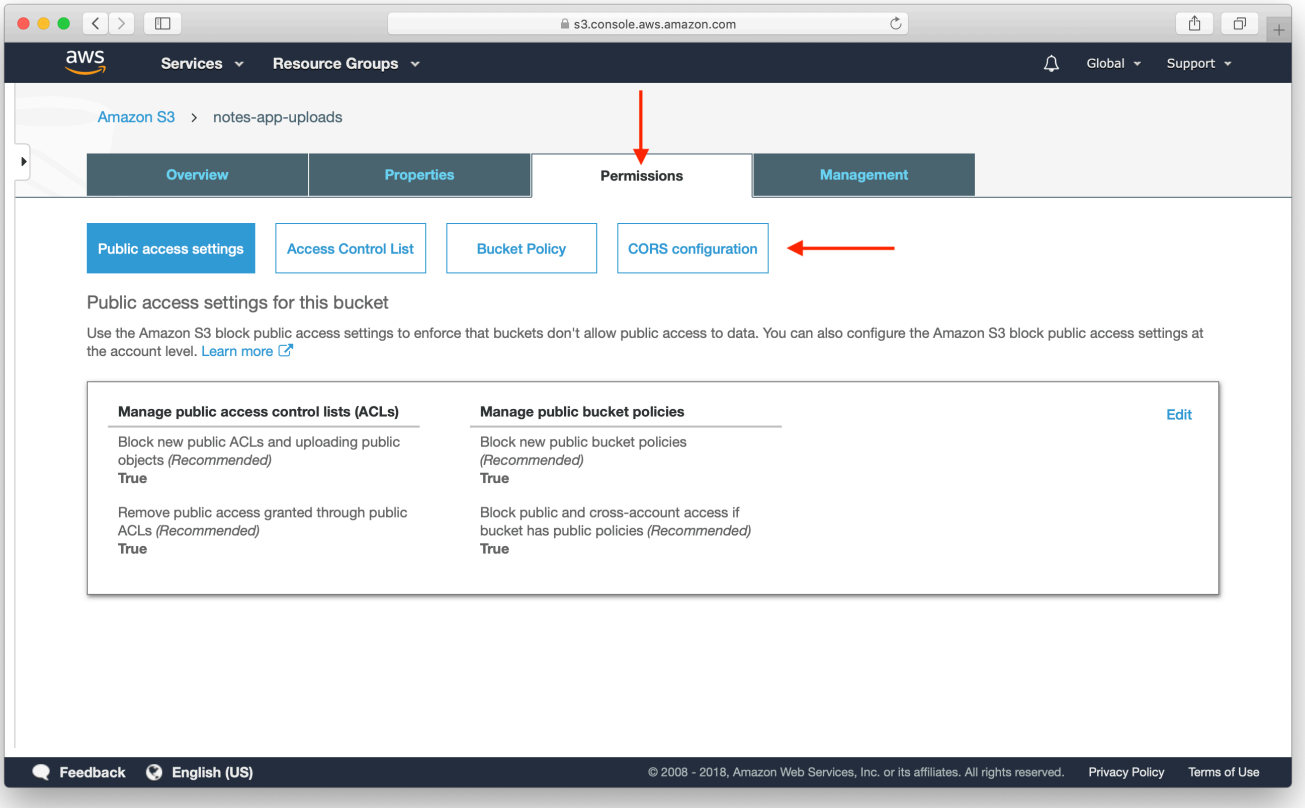
Enable CORS

In the notes app we'll be building, users will be uploading files to the bucket we just created. And since our app will be served through our custom domain, it'll be communicating across domains while it does the uploads. By default, S3 does not allow its resources to be accessed from a different domain. However, cross-origin resource sharing (CORS) defines a way for client web applications that are loaded in one domain to interact with resources in a different domain. Let's enable CORS for our S3 bucket.

Select the bucket we just created.



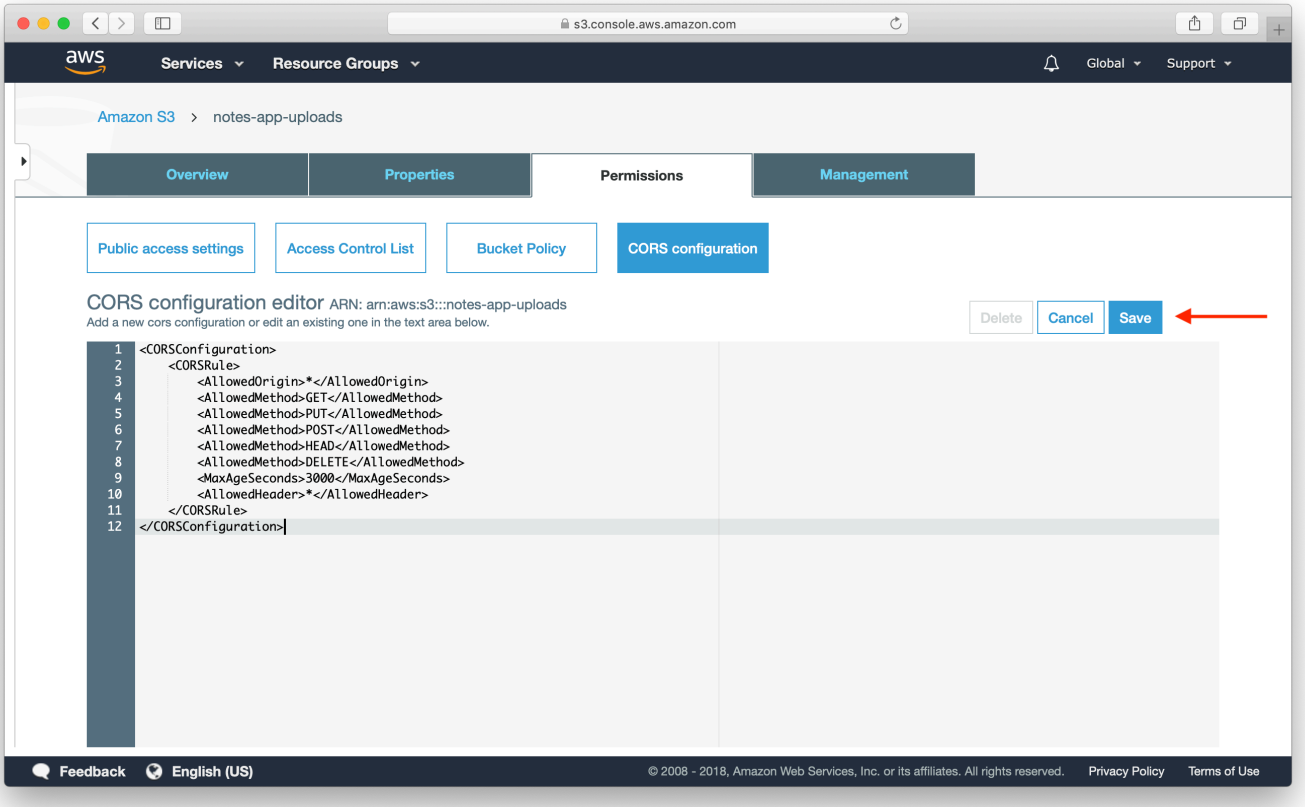
Select the **Permissions** tab, then select **CORS configuration**.



Add the following CORS configuration into the editor, then hit **Save**.

```
<CORSConfiguration>
        <CORSRule>
                <AllowedOrigin>*</AllowedOrigin>
                <AllowedMethod>GET</AllowedMethod>
                <AllowedMethod>PUT</AllowedMethod>
                <AllowedMethod>POST</AllowedMethod>
                <AllowedMethod>HEAD</AllowedMethod>
                <AllowedMethod>DELETE</AllowedMethod>
                <MaxAgeSeconds>3000</MaxAgeSeconds>
                <AllowedHeader>*</AllowedHeader>
        </CORSRule>
</CORSConfiguration>
```

Note that you can edit this configuration to use your own domain or a list of domains when you use this in production.



Now that our S3 bucket is ready, let's get set up to handle user authentication.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-an-s3-bucket-for-file-uploads/150)

Create a Cognito User Pool

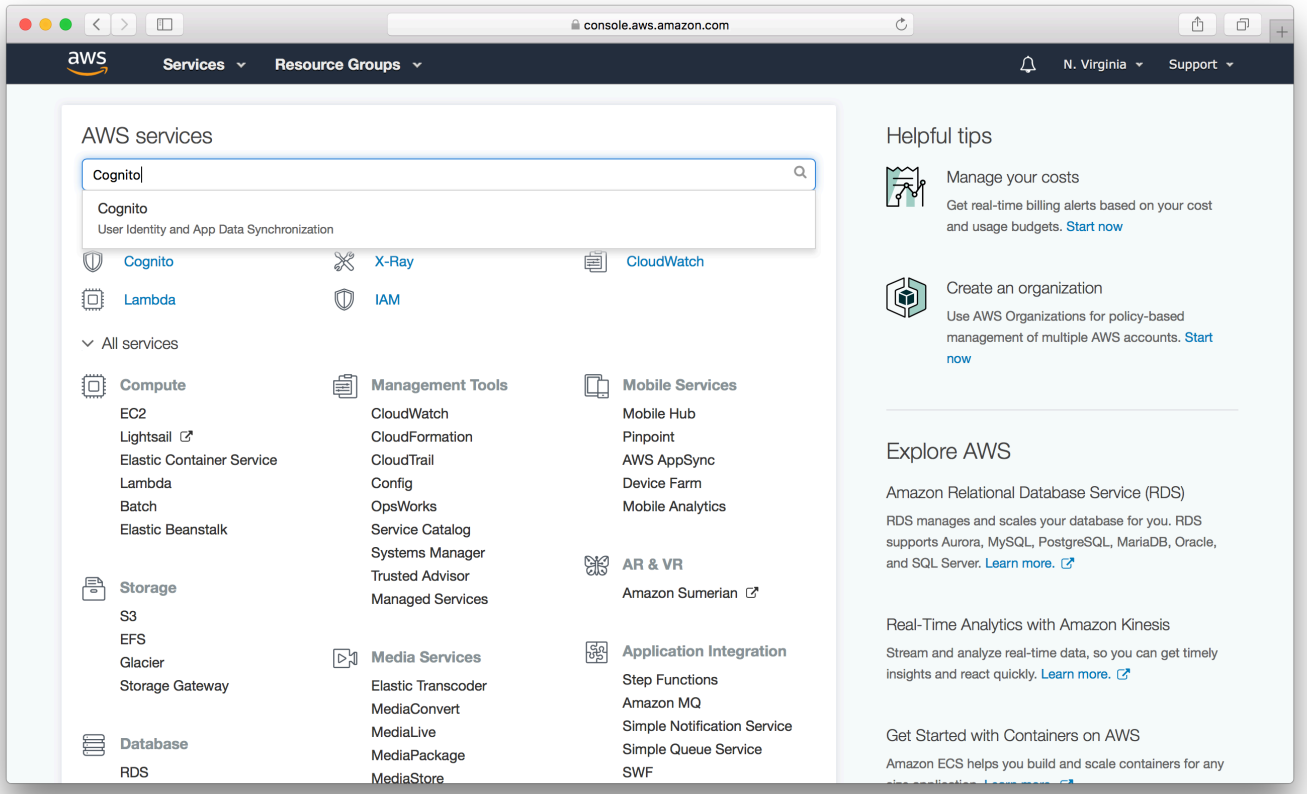
Our notes app needs to handle user accounts and authentication in a secure and reliable way. To do this we are going to use Amazon Cognito (https://aws.amazon.com/cognito/).

Amazon Cognito User Pool makes it easy for developers to add sign-up and sign-in functionality to web and mobile applications. It serves as your own identity provider to maintain a user directory. It supports user registration and sign-in, as well as provisioning identity tokens for signed-in users.

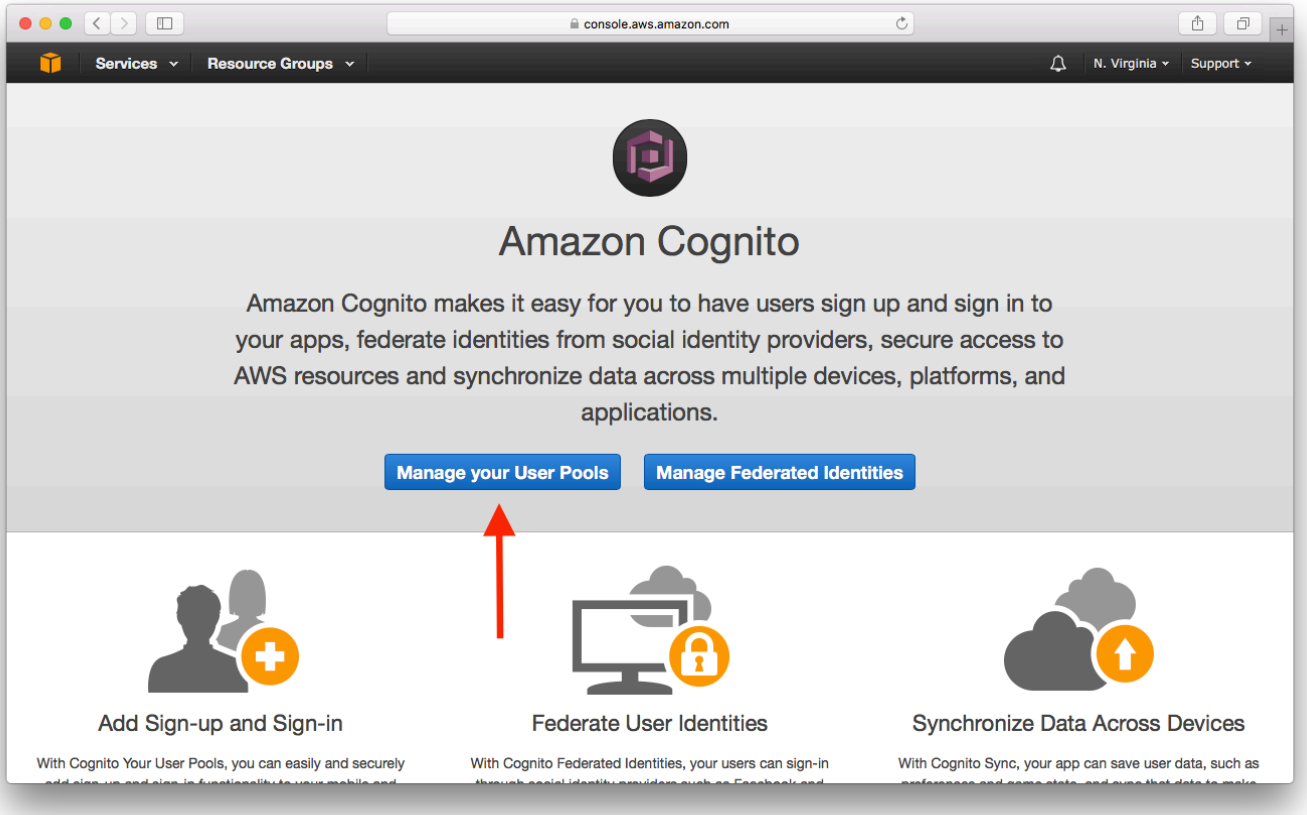
In this chapter, we are going to create a User Pool for our notes app.

Create User Pool

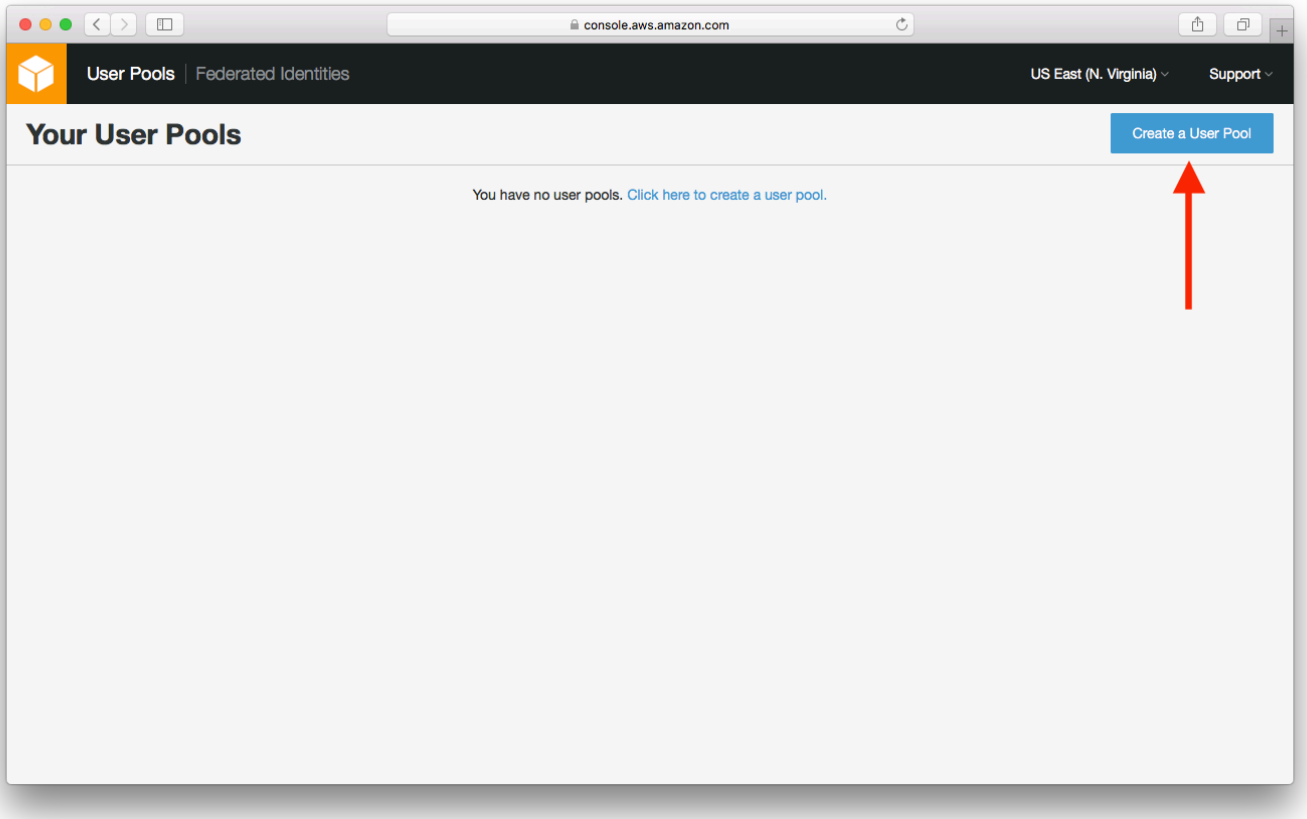
From your AWS Console (https://console.aws.amazon.com), select **Cognito** from the list of services.



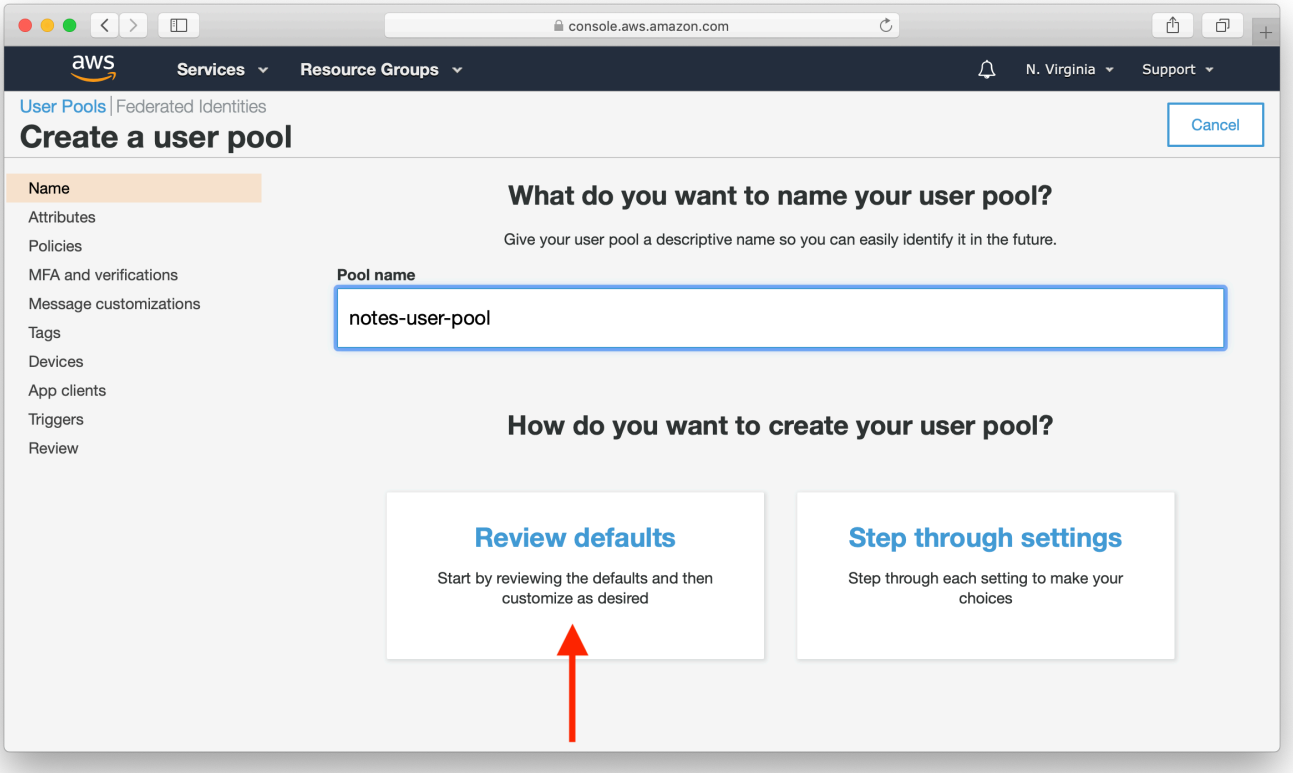
Select **Manage your User Pools.**



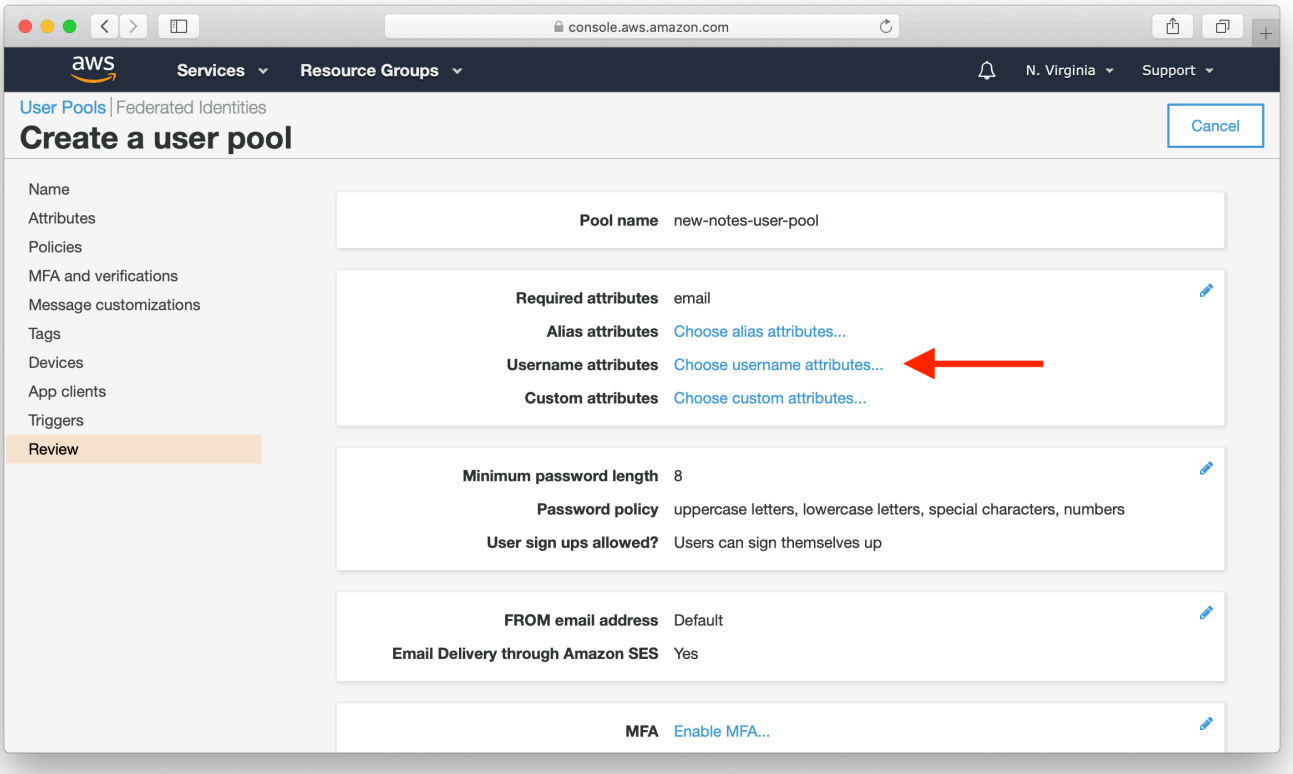
Select **Create a User Pool.**



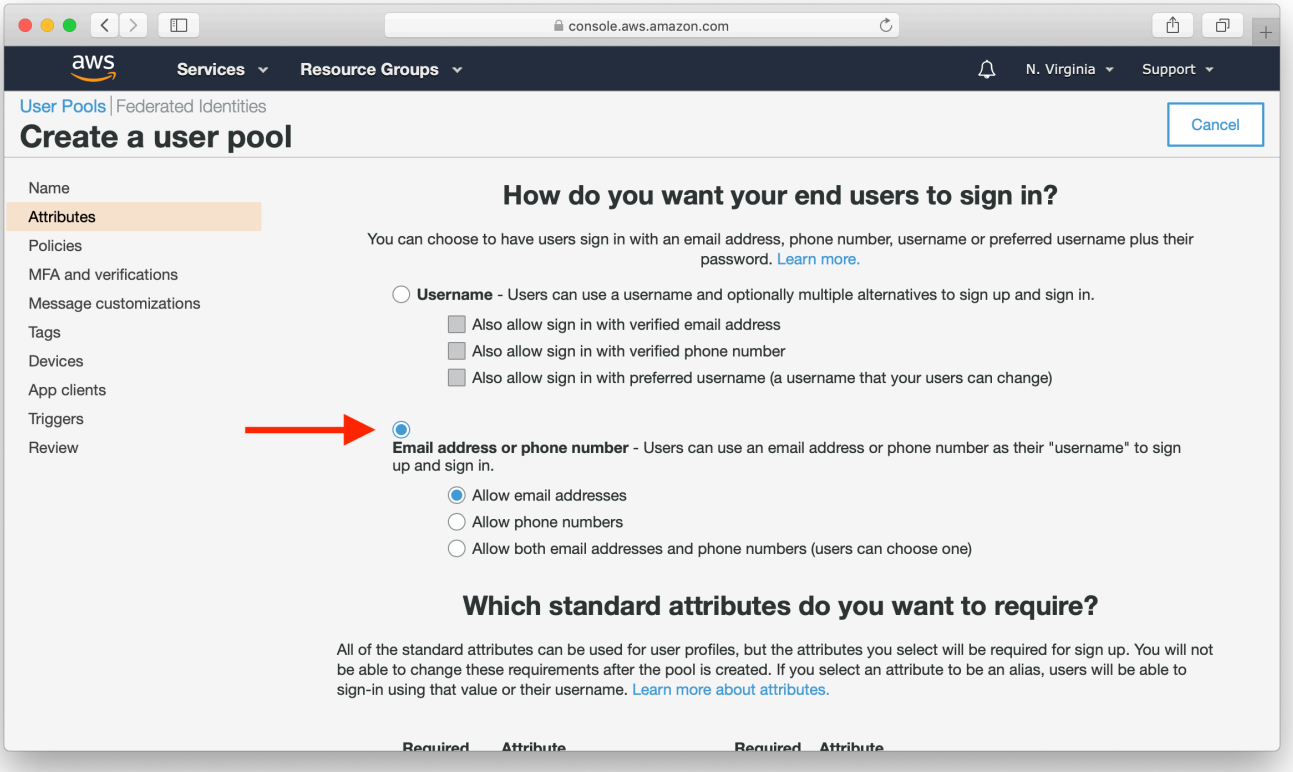
Enter **Pool name** and select **Review defaults**.



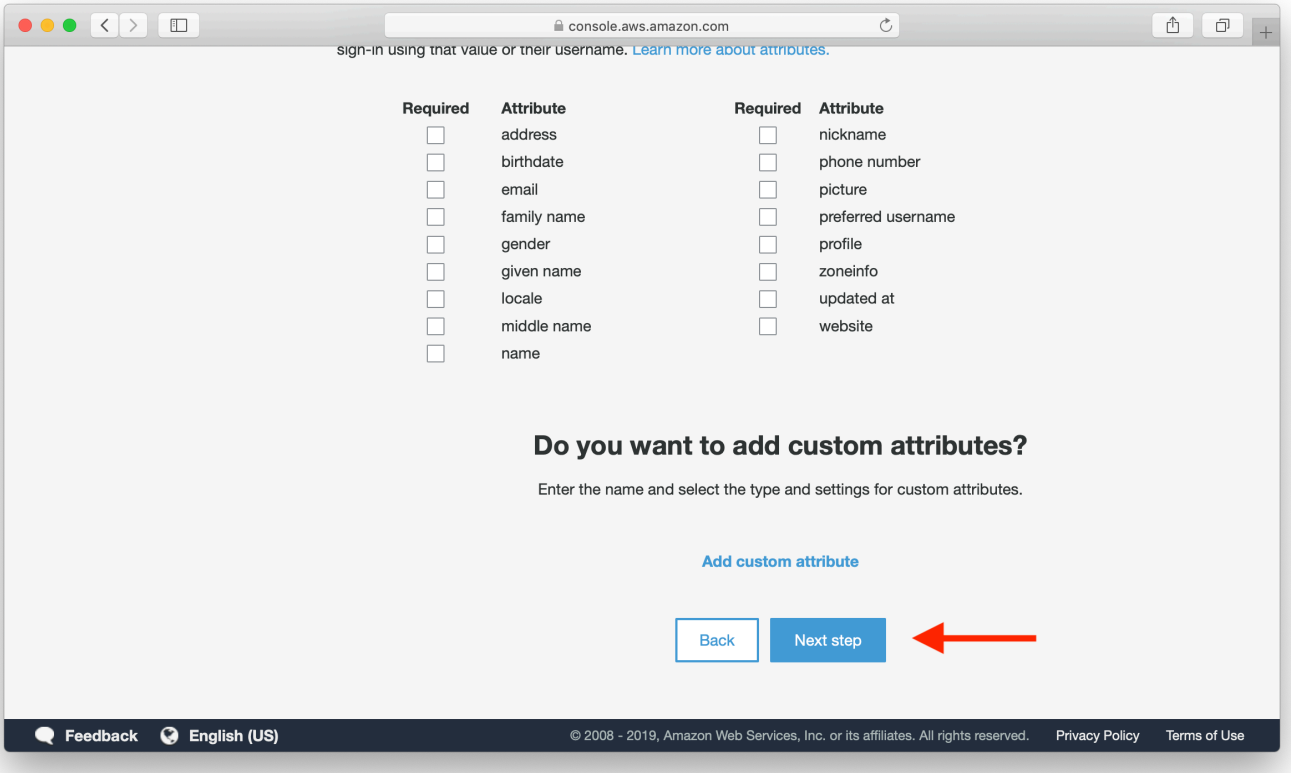
Select **Choose username attributes...**



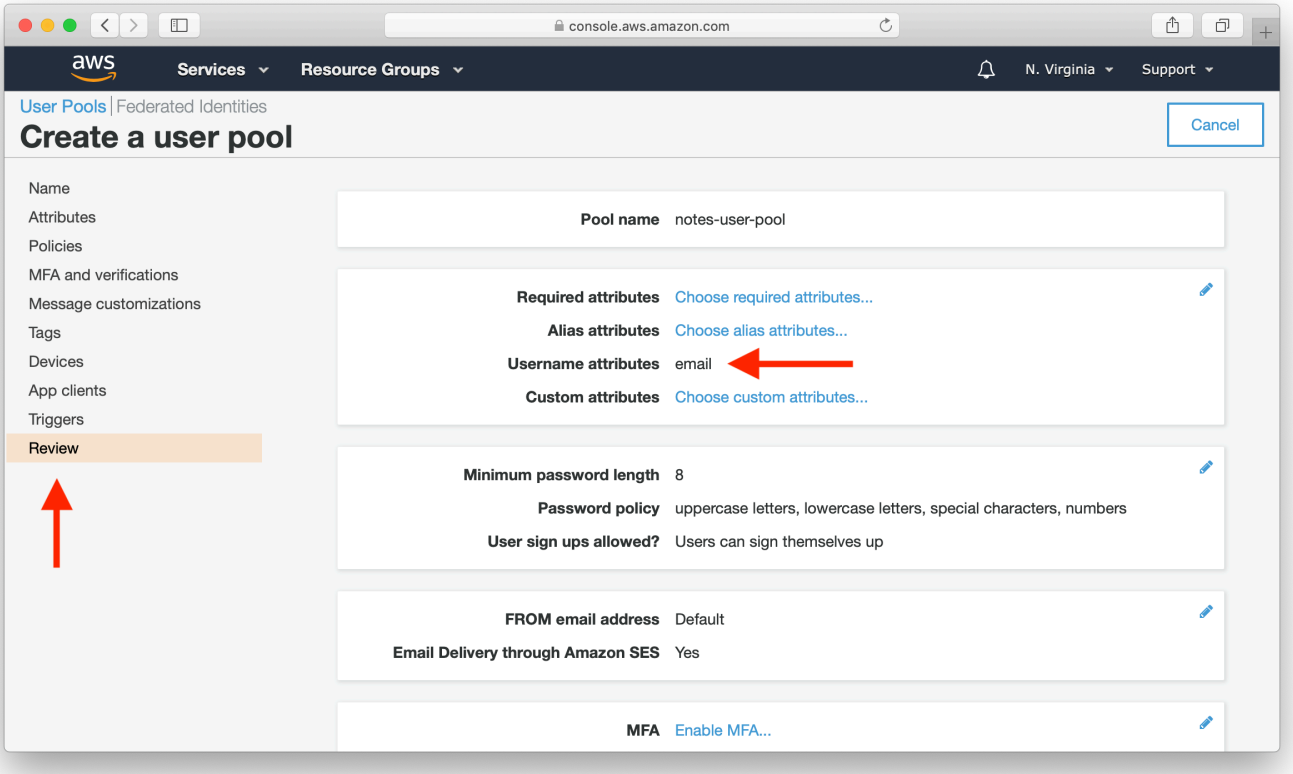
And select **Email address or phone numbers** and **Allow email addresses**. This is telling Cognito User Pool that we want our users to be able to sign up and login with their email as their username.



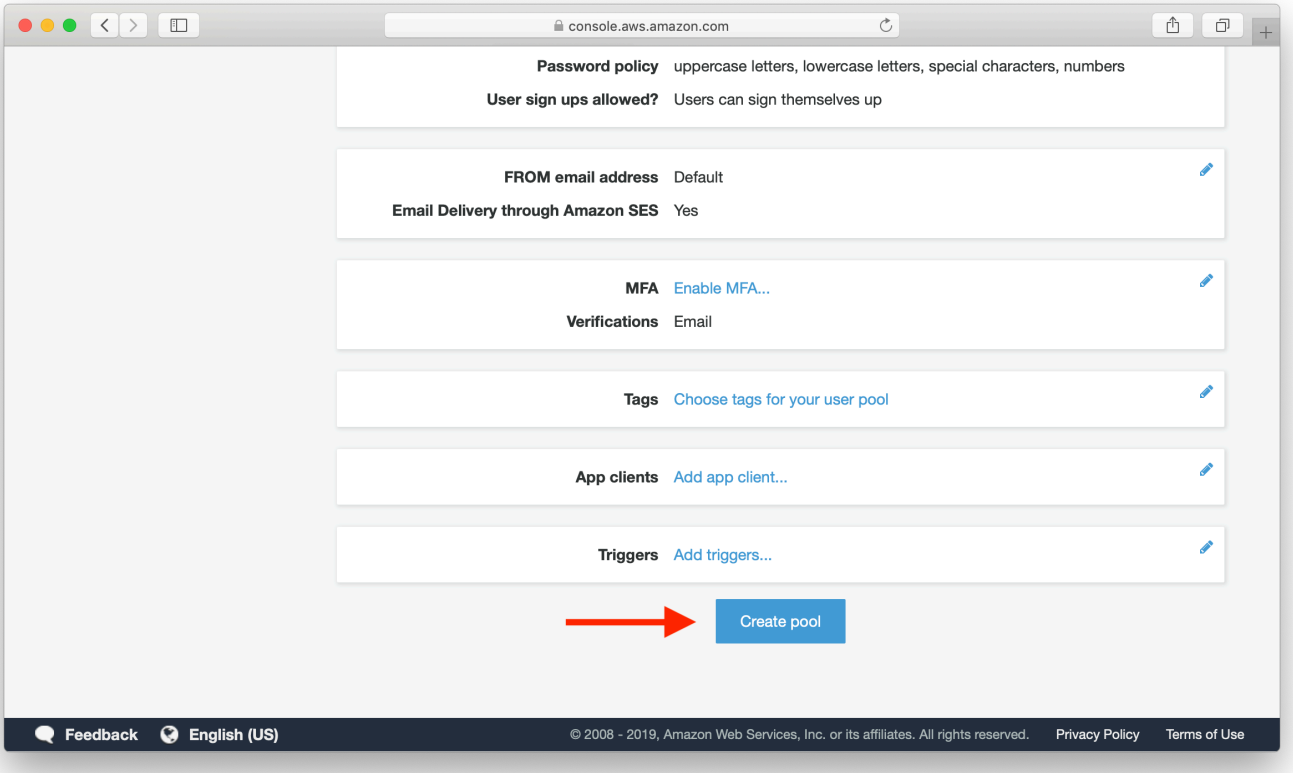
Scroll down and select **Next step**.



Hit **Review** in the side panel and make sure that the **Username attributes** is set to **email**.

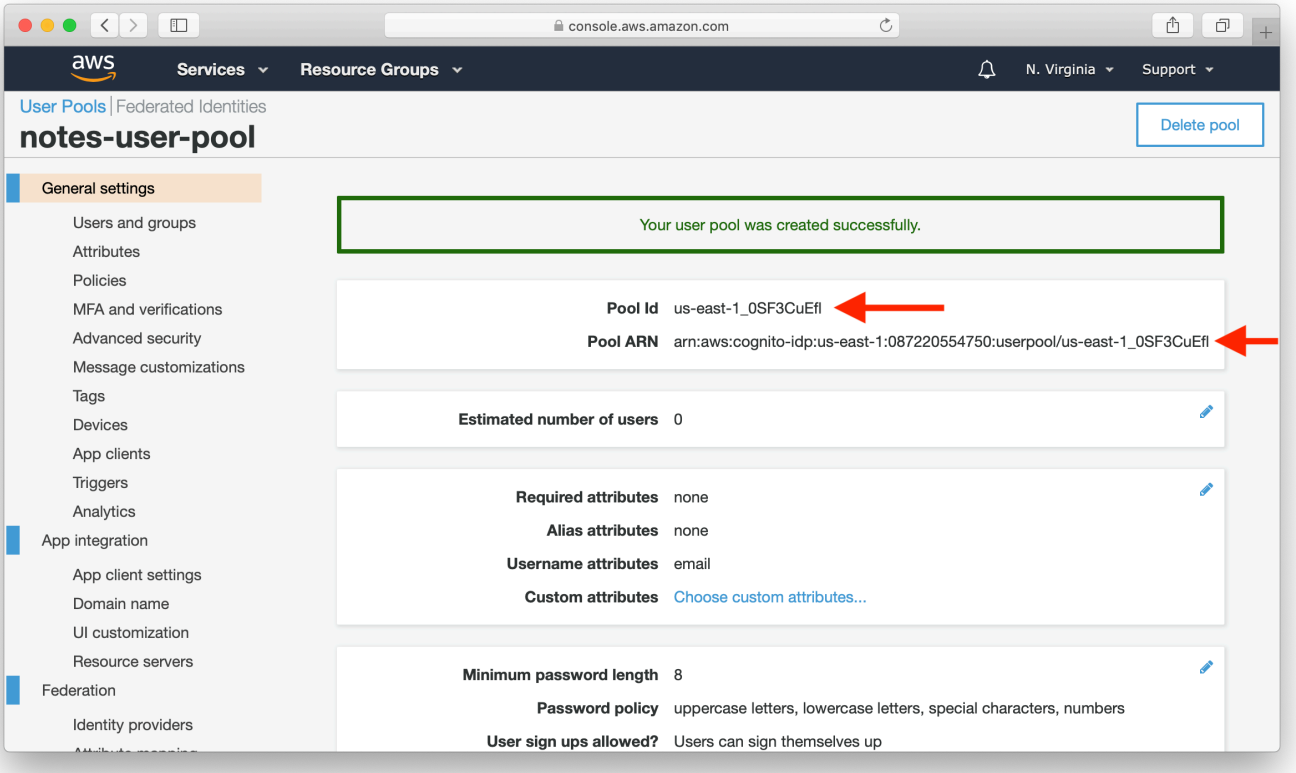


Now hit **Create pool** at the bottom of the page.



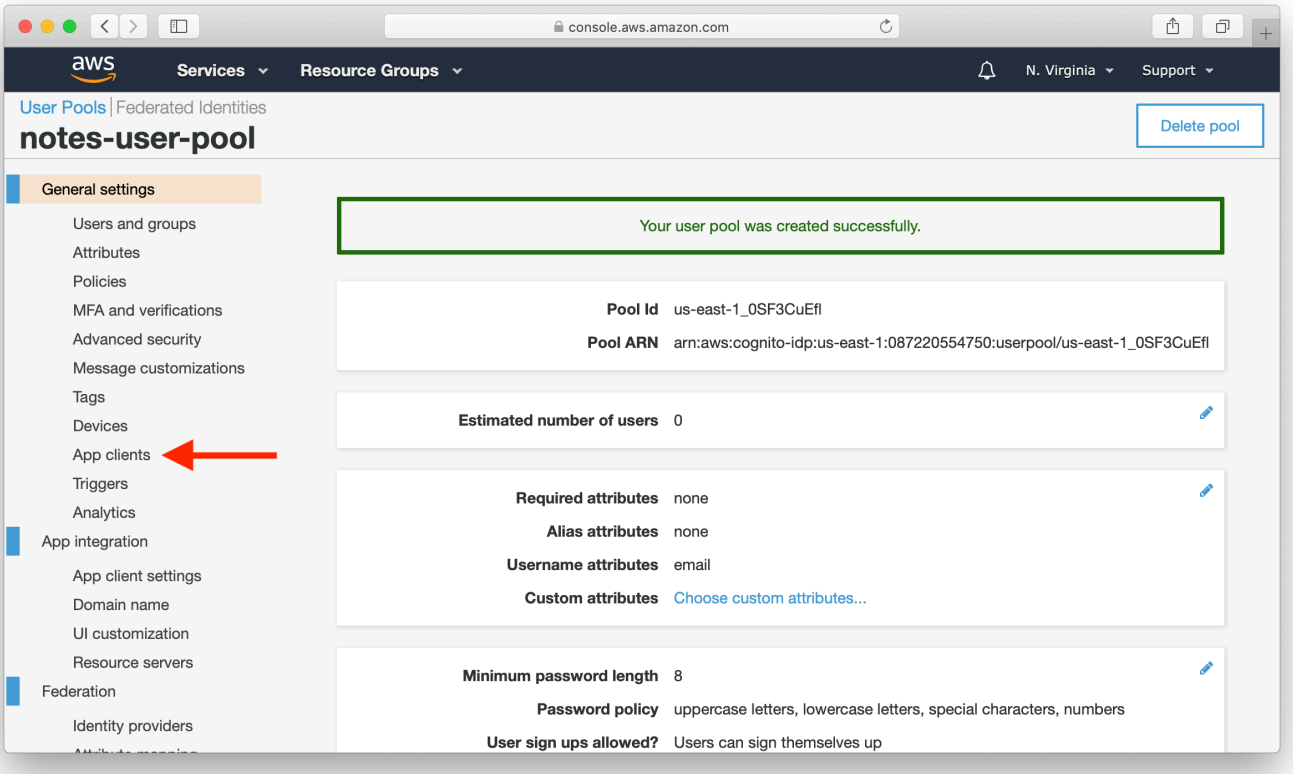
Your User Pool has been created. Take a note of the **Pool Id** and **Pool ARN**

which will be required later. Also, note the region that your User Pool is created in – in our case it's `us-east-1`.

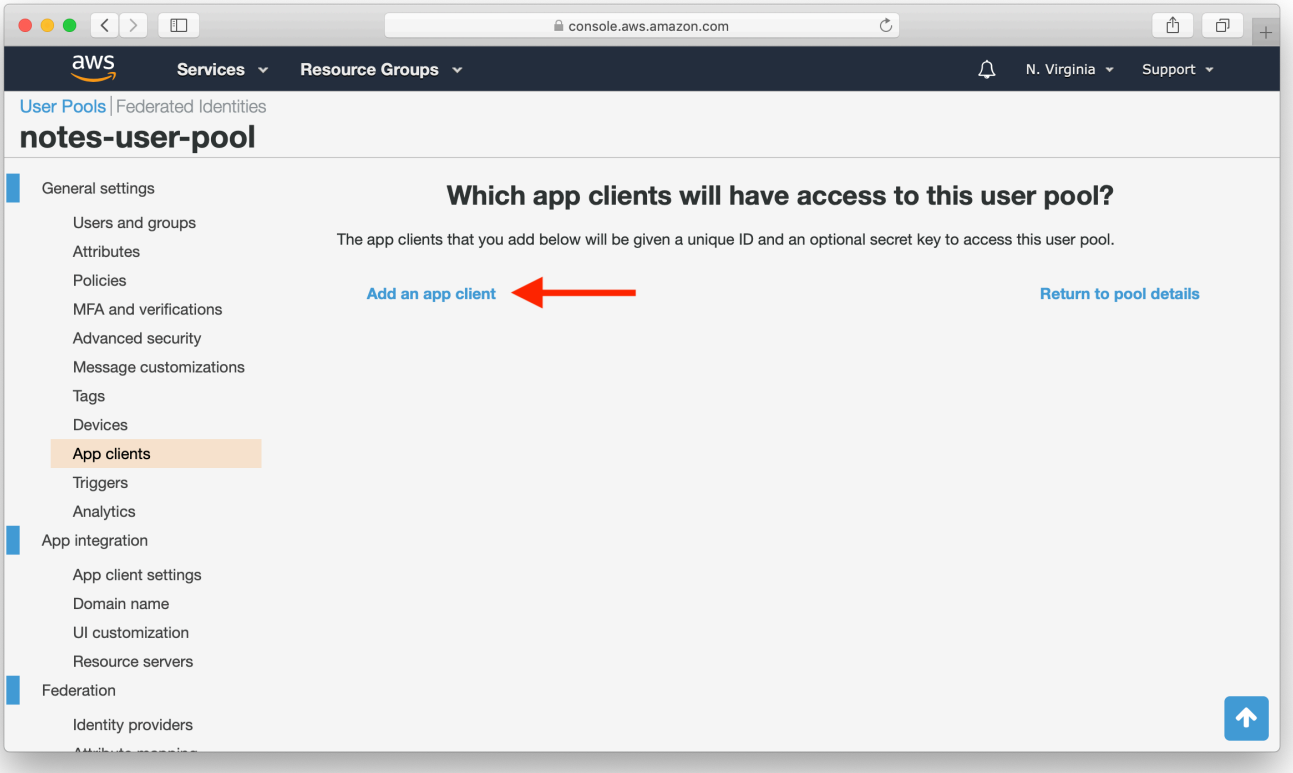


Create App Client

Select **App clients** from the left panel.



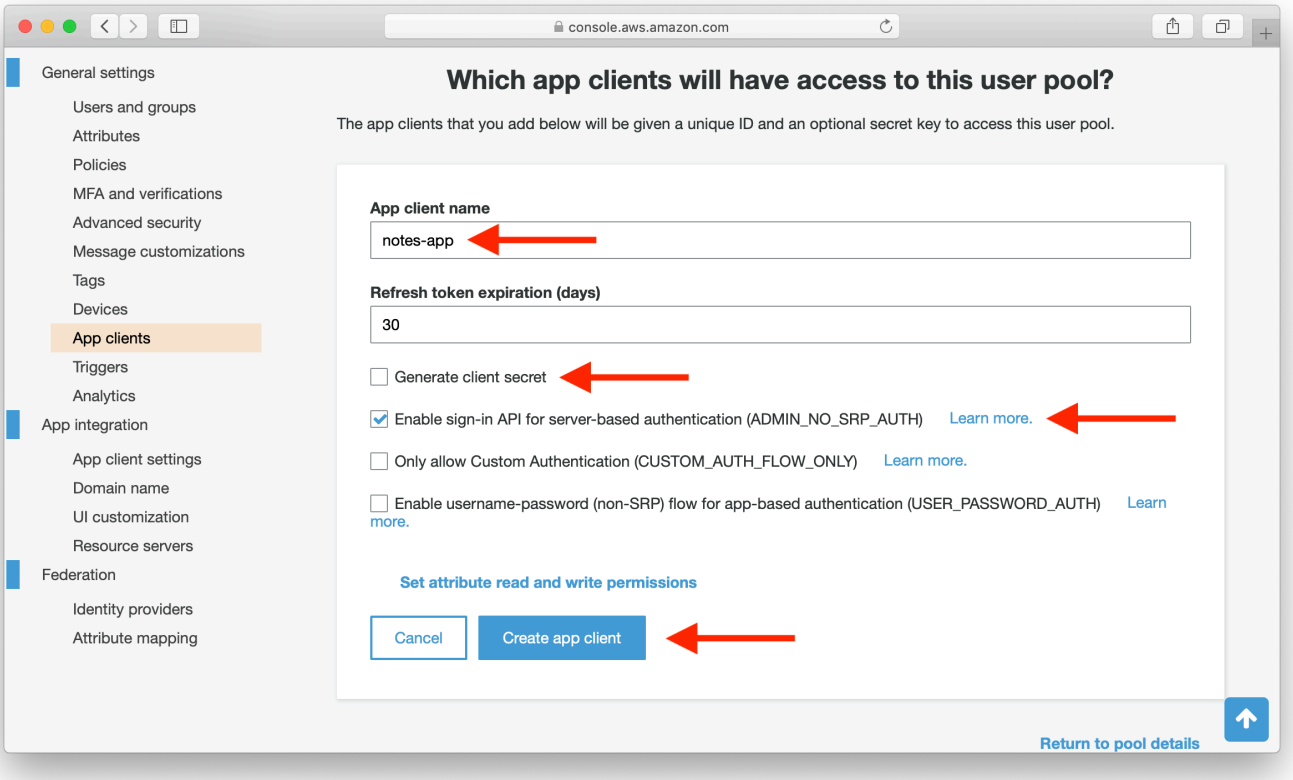
Select **Add an app client.**



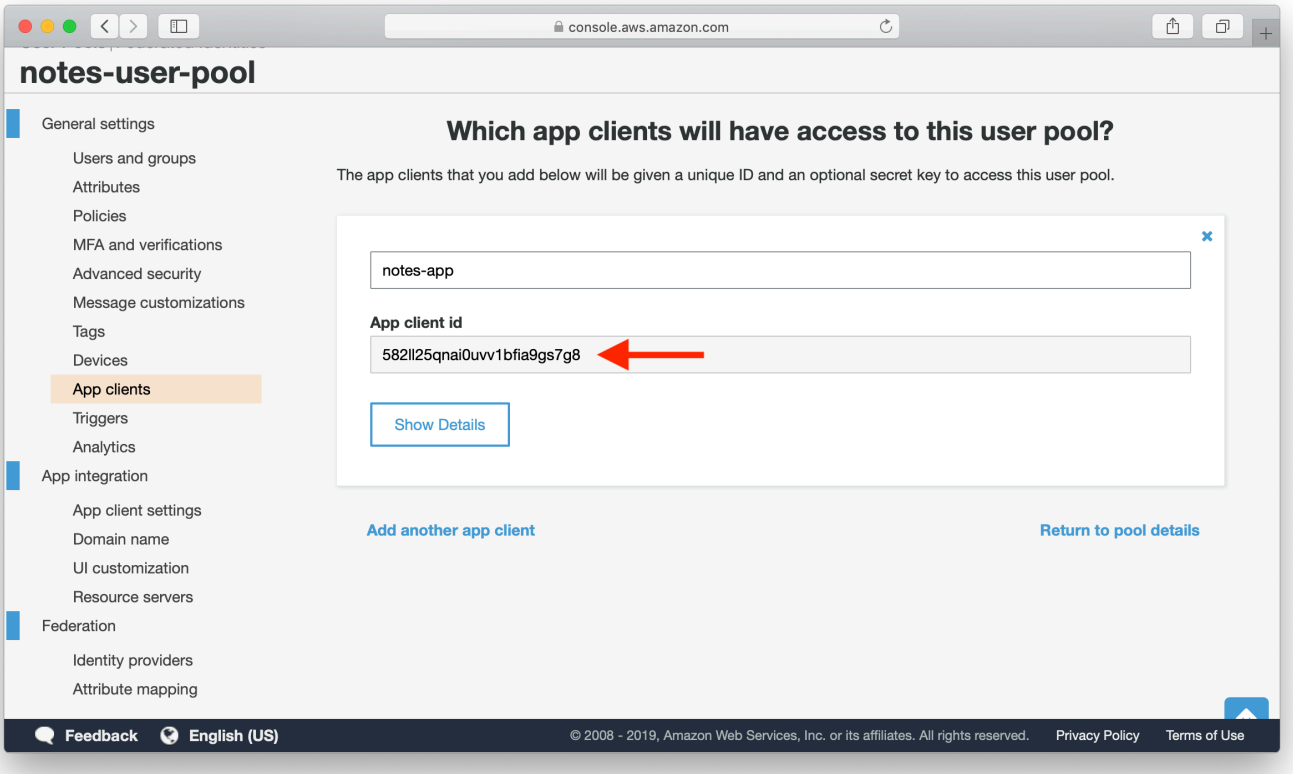
Enter **App client name**, un-select **Generate client secret**, select **Enable**

sign-in API for server-based authentication, then select **Create app client**.

- **Generate client secret**: user pool apps with a client secret are not supported by the JavaScript SDK. We need to un-select the option.
- **Enable sign-in API for server-based authentication**: required by AWS CLI when managing the pool users via command line interface. We will be creating a test user through the command line interface in the next chapter.

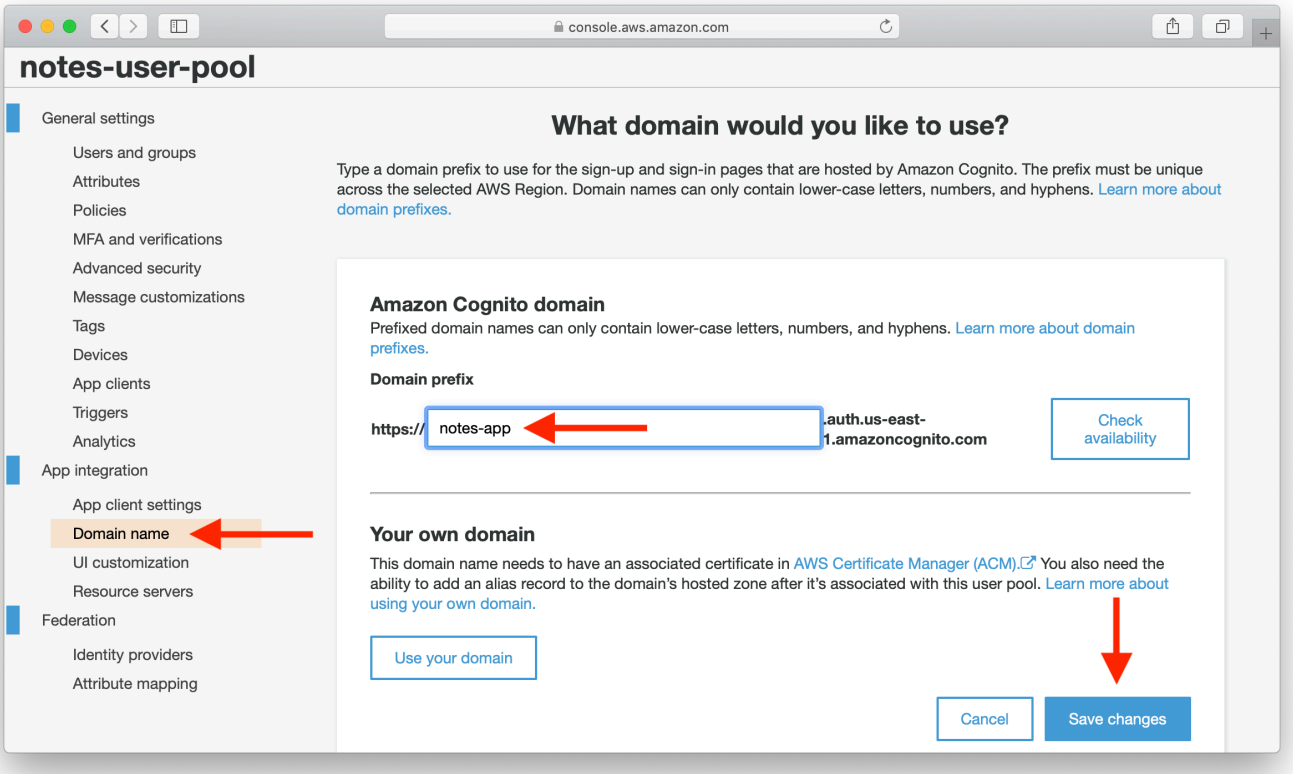


Your app client has been created. Take note of the **App client id** which will be required in the later chapters.



Create Domain Name

Finally, select **Domain name** from the left panel. Enter your unique domain name and select **Save changes**. In our case we are using `notes-app`.



Now our Cognito User Pool is ready. It will maintain a user directory for our notes app. It will also be used to authenticate access to our API. Next let's set up a test user within the pool.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-cognito-user-pool/148)

Create a Cognito Test User

In this chapter, we are going to create a test user for our Cognito User Pool. We are going to need this user to test the authentication portion of our app later.

Create User

First, we will use AWS CLI to sign up a user with their email and password.

CHANGE In your terminal, run.

```
$ aws cognito-idp sign-up \
  --region YOUR_COGNITO_REGION \
  --client-id YOUR_COGNITO_APP_CLIENT_ID \
  --username admin@example.com \
  --password Passw0rd!
```

Now, the user is created in Cognito User Pool. However, before the user can authenticate with the User Pool, the account needs to be verified. Let's quickly verify the user using an administrator command.

CHANGE In your terminal, run.

```
$ aws cognito-idp admin-confirm-sign-up \
  --region YOUR_COGNITO_REGION \
  --user-pool-id YOUR_COGNITO_USER_POOL_ID \
  --username admin@example.com
```

Now our test user is ready. Next, let's set up the Serverless Framework to create our backend APIs.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-cognito-test-user/126)

Set up the Serverless Framework

We are going to be using AWS Lambda (https://aws.amazon.com/lambda/) and Amazon API Gateway (https://aws.amazon.com/api-gateway/) to create our backend. AWS Lambda is a compute service that lets you run code without provisioning or managing servers. You pay only for the compute time you consume - there is no charge when your code is not running. And API Gateway makes it easy for developers to create, publish, maintain, monitor, and secure APIs. Working directly with AWS Lambda and configuring API Gateway can be a bit cumbersome; so we are going to use the Serverless Framework (https://serverless.com) to help us with it.

The Serverless Framework enables developers to deploy backend applications as independent functions that will be deployed to AWS Lambda. It also configures AWS Lambda to run your code in response to HTTP requests using Amazon API Gateway.

In this chapter, we are going to set up the Serverless Framework on our local development environment.

Install Serverless

✦ CHANGE Install Serverless globally.

```
$ npm install serverless -g
```

The above command needs NPM (https://www.npmjs.com), a package manager for JavaScript. Follow this (https://docs.npmjs.com/getting-started/installing-node) if you need help installing NPM.

✦ CHANGE In your working directory; create a project using a Node.js starter. We'll go over some of the details of this starter project in the next chapter.

```
$ serverless install --url
https://github.com/AnomalyInnovations/serverless-
nodejs-starter --name notes-app-api
```

✦ CHANGE Go into the directory for our backend api project.

```
$ cd notes-app-api
```

Now the directory should contain a few files including, the **handler.js** and **serverless.yml**.

- **handler.js** file contains actual code for the services/functions that will be deployed to AWS Lambda.
- **serverless.yml** file contains the configuration on what AWS services Serverless will provision and how to configure them.

We also have a `tests/` directory where we can add our unit tests.

Install Node.js packages

The starter project relies on a few dependencies that are listed in the `package.json`.

✦ CHANGE At the root of the project, run.

```
$ npm install
```

CHANGE Next, we'll install a couple of other packages specifically for our backend.

```
$ npm install aws-sdk --save-dev
$ npm install uuid --save
```

- **aws-sdk** allows us to talk to the various AWS services.
- **uuid** generates unique ids. We need this for storing things to DynamoDB.

The starter project that we are using allows us to use the version of JavaScript that we'll be using in our frontend app later. Let's look at exactly how it does this.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/set-up-the-serverless-framework/145)

Add Support for ES6/ES7 JavaScript

AWS Lambda recently added support for Node.js v8.10 and v10.x. The supported syntax is a little different when compared to the frontend React app we'll be working on a little later. It makes sense to use similar ES features across both parts of the project – specifically, we'll be relying on ES imports/exports in our handler functions. To do this we will be transpiling our code using Babel (https://babeljs.io) and Webpack 4 (https://webpack.github.io). Also, Webpack allows us to generate optimized packages for our Lambda functions by only including the code that is used in our function. This helps keep our packages small and reduces cold start times. Serverless Framework supports plugins to do this automatically. We are going to use an extension of the popular serverless-webpack (https://github.com/serverless-heaven/serverless-webpack) plugin, serverless-bundle (https://github.com/AnomalyInnovations/serverless-bundle).

All this has been added in the previous chapter using the `serverless-nodejs-starter` (/chapters/serverless-nodejs-starter.html). We created this starter for a couple of reasons:

- Generate optimized packages for our Lambda functions
- Use a similar version of JavaScript in the frontend and backend
- Ensure transpiled code still has the right line numbers for error messages

- Lint our code and add support for unit tests
- Allow you to run your backend API locally
- Not have to manage any Webpack or Babel configs

If you recall we installed this starter using the `serverless install --url https://github.com/AnomalyInnovations/serverless-nodejs-starter --name my-project` command. This is telling Serverless Framework to use the starter (https://github.com/AnomalyInnovations/serverless-nodejs-starter) as a template to create our project.

In this chapter, let's quickly go over how it's doing this so you'll be able to make changes in the future if you need to.

Serverless Webpack

The transpiling process of converting our ES code to Node v10.x JavaScript is done by the `serverless-bundle` plugin. This plugin was added in our `serverless.yml`.

✦ CHANGE Open `serverless.yml` and replace the default with the following.

```
service: notes-app-api

# Create an optimized package for our functions
package:
  individually: true

plugins:
  - serverless-bundle # Package our functions with
```

```
Webpack
  - serverless-offline
  - serverless-dotenv-plugin # Load .env as environment
variables

provider:
  name: aws
  runtime: nodejs10.x
  stage: prod
  region: us-east-1
```

The `service` option is pretty important. We are calling our service the `notes-app-api`. Serverless Framework creates your stack on AWS using this as the name. This means that if you change the name and deploy your project, it will create a completely new project.

You'll notice the plugins — `serverless-bundle`, `serverless-offline`, and `serverless-dotenv-plugin`, that we have included. The first plugin we talked about above, while the serverless-offline (https://github.com/dherault/serverless-offline) is helpful for local development and serverless-dotenv-plugin (https://github.com/colynb/serverless-dotenv-plugin) that we'll use later loads the `.env` files as Lambda environment variables.

We are also using this option:

```
# Create an optimized package for our functions
package:
  individually: true
```

By default, Serverless Framework creates one large package for all the

Lambda functions in your app. Large Lambda function packages can cause longer cold starts. By setting `individually: true`, we are telling Serverless Framework to create a single package per Lambda function. This in combination with serverless-bundle (and Webpack) will generate optimized packages. Note that, this'll slow down our builds but the performance benefit is well worth it.

Now we are ready to write our backend code. But before that, let's create a GitHub repo to store our code.

For help and discussion

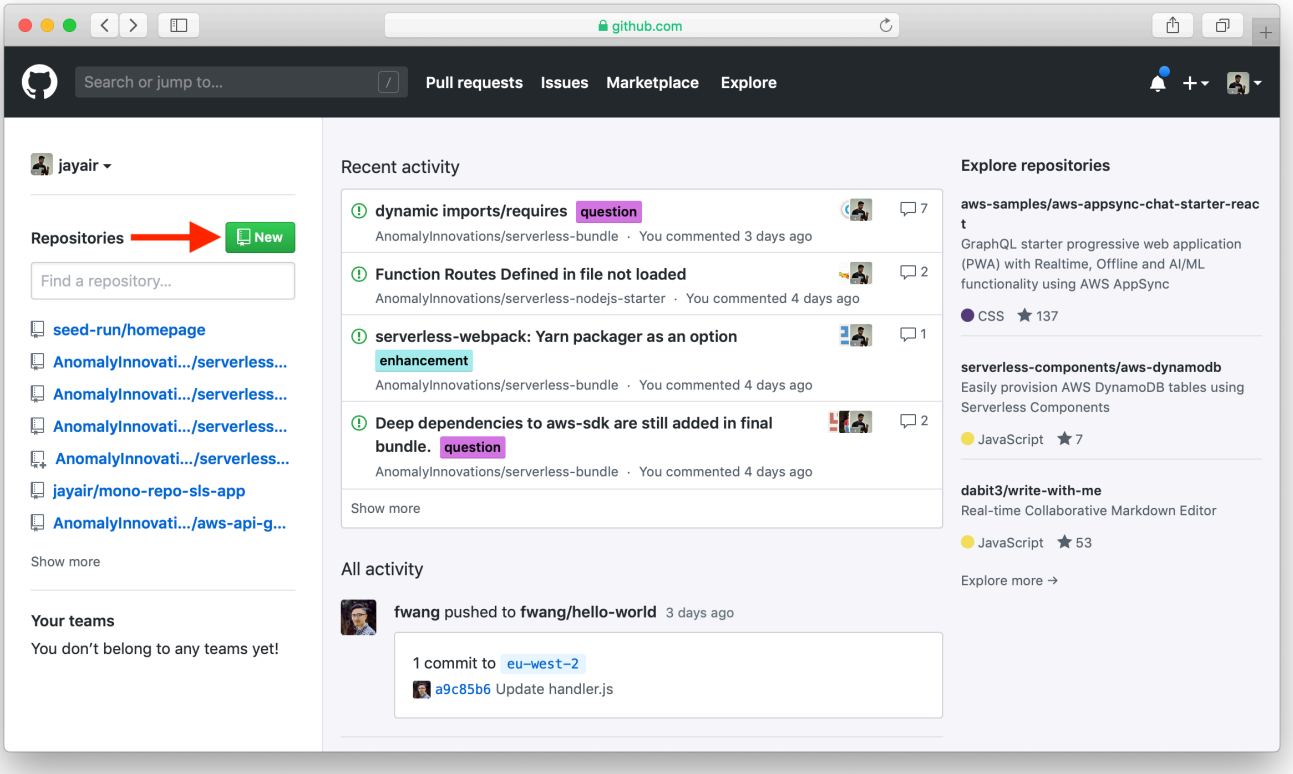
Comments on this chapter (https://discourse.serverless-stack.com/t/add-support-for-es6-es7-javascript/128)

Initialize the Backend Repo

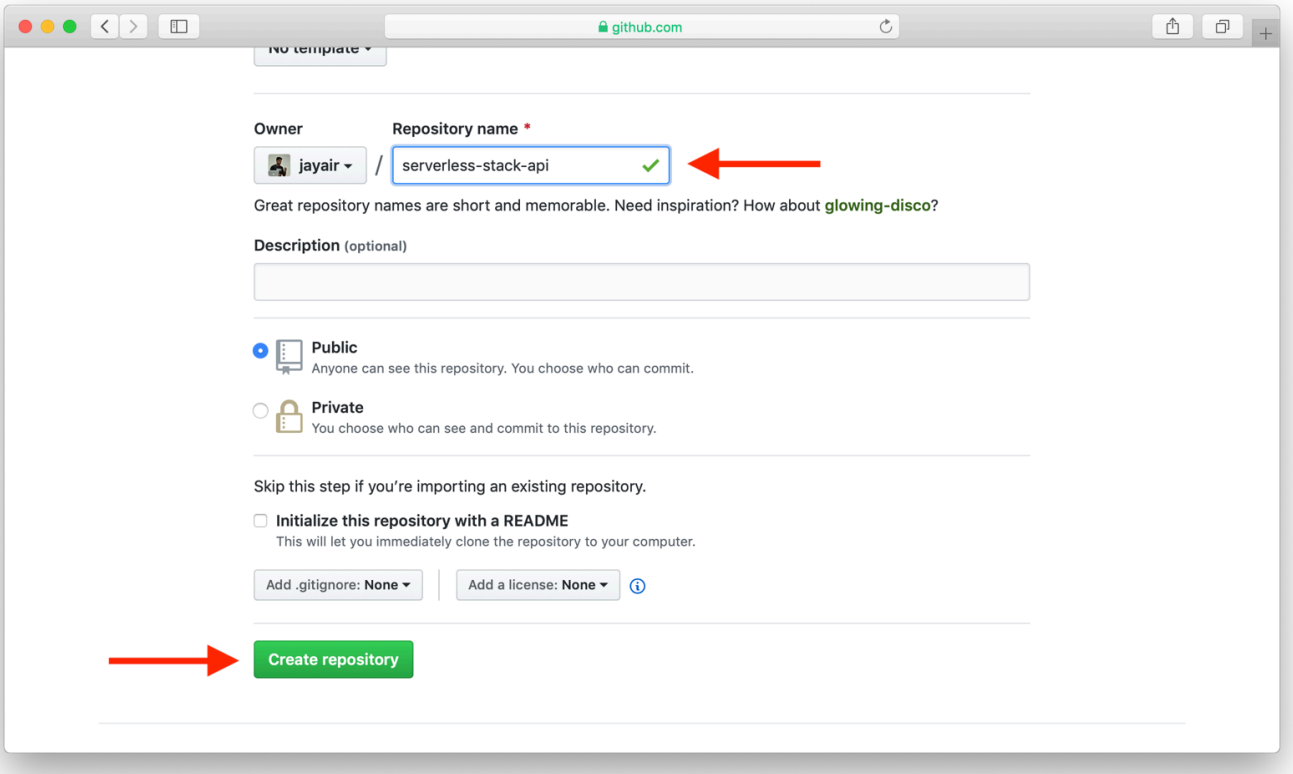
Before we start working on our app, let's create a GitHub repository for this project. It's a good way to store our code and we'll use this repository later to automate deploying our app.

Create a New Github Repo

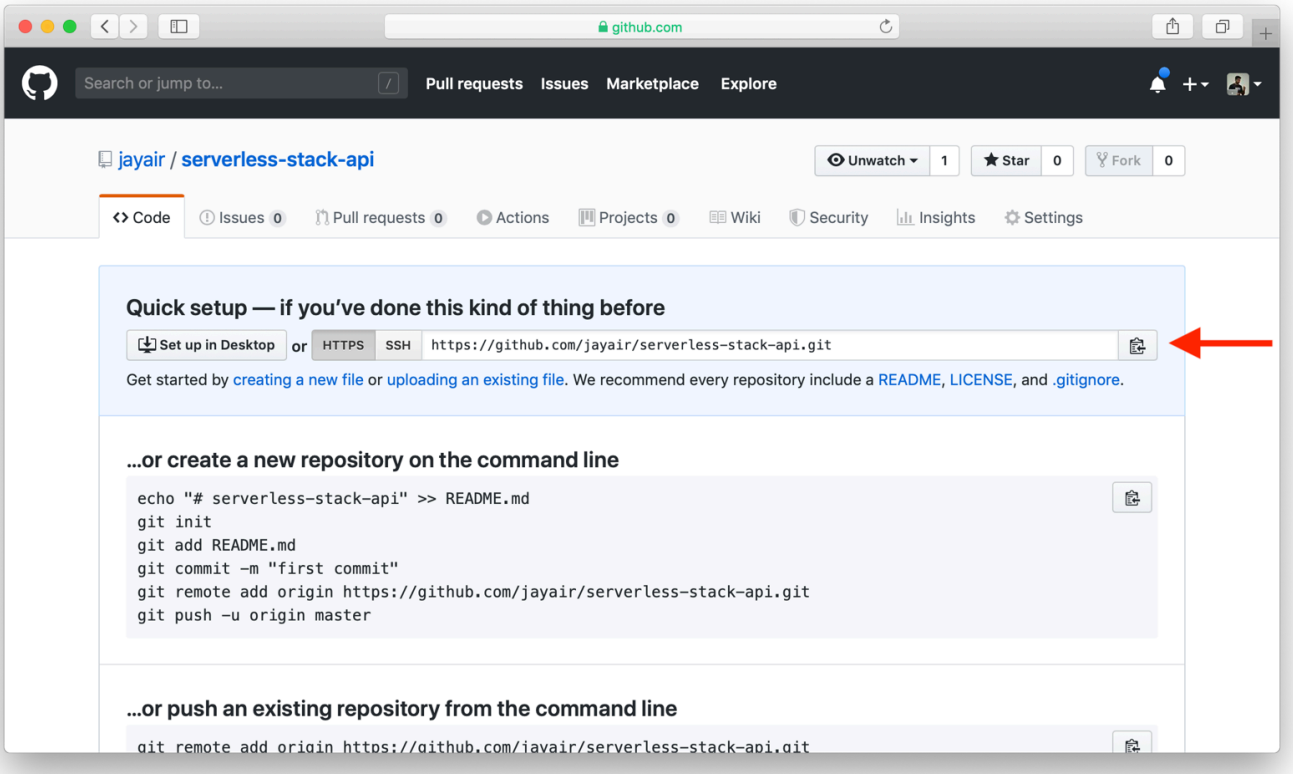
Let's head over to GitHub (https://github.com). Make sure you are signed in and hit **New repository**.



Give your repository a name, in our case we are calling it `serverless-stack-api`. Next hit **Create repository**.



Once your repository is created, copy the repository URL. We'll need this soon.



In our case the URL is:

```
https://github.com/jayair/serverless-stack-api.git
```

Initialize Your New Repo

CHANGE Now head back to your project and use the following command to initialize your new repo.

```
$ git init
```

CHANGE Add the existing files.

```
$ git add .
```

CHANGE Create your first commit.

```
$ git commit -m "First commit"
```

CHANGE Link it to the repo you created on GitHub.

```
$ git remote add origin REPO_URL
```

Here `REPO_URL` is the URL we copied from GitHub in the steps above. You can verify that it has been set correctly by doing the following.

```
$ git remote -v
```

CHANGE Finally, let's push our first commit to GitHub using:

```
$ git push -u origin master
```

Now we are ready to build our backend!

CHECKPOINT

For reference, here is the code we have so far.

Backend Source :initialize-the-backend-repo (https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/initialize-the-backend-repo)

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/initialize-the-backend-repo/159)

Add a Create Note API

Let's get started on our backend by first adding an API to create a note. This API will take the note object as the input and store it in the database with a new id. The note object will contain the `content` field (the content of the note) and an `attachment` field (the URL to the uploaded file).

Add the Function

Let's add our first function.

✦ CHANGE Create a new file called `create.js` in our project root with the following.

```
import uuid from "uuid";
import AWS from "aws-sdk";

const dynamoDb = new AWS.DynamoDB.DocumentClient();

export function main(event, context, callback) {
  // Request body is passed in as a JSON encoded string
in 'event.body'
  const data = JSON.parse(event.body);

  const params = {
    TableName: process.env.tableName,
```

```
    // 'Item' contains the attributes of the item to be
created
    // - 'userId': user identities are federated
through the
    //             Cognito Identity Pool, we will use
the identity id
    //             as the user id of the authenticated
user
    // - 'noteId': a unique uuid
    // - 'content': parsed from request body
    // - 'attachment': parsed from request body
    // - 'createdAt': current Unix timestamp
    Item: {
      userId:
event.requestContext.identity.cognitoIdentityId,
      noteId: uuid.v1(),
      content: data.content,
      attachment: data.attachment,
      createdAt: Date.now()
    }
  };

  dynamoDb.put(params, (error, data) => {
    // Set response headers to enable CORS (Cross-
Origin Resource Sharing)
    const headers = {
      "Access-Control-Allow-Origin": "*",
      "Access-Control-Allow-Credentials": true
    };

    // Return status code 500 on error
```

```
    if (error) {
      const response = {
        statusCode: 500,
        headers: headers,
        body: JSON.stringify({ status: false })
      };
      callback(null, response);
      return;
    }

    // Return status code 200 and the newly created
item
    const response = {
      statusCode: 200,
      headers: headers,
      body: JSON.stringify(params.Item)
    };
    callback(null, response);
  });
}
```

There are some helpful comments in the code but we are doing a few simple things here.

- The AWS JS SDK assumes the region based on the current region of the Lambda function. So if your DynamoDB table is in a different region, make sure to set it by calling `AWS.config.update({ region: "my-region" });` before initilizing the DynamoDB client.
- Parse the input from the `event.body`. This represents the HTTP request parameters.
- We read the name of our DynamoDB table from the environment

variable using `process.env.tableName`. We'll be setting this in our `serverless.yml` below. We do this so we won't have to hardcode it in every function.

- The `userId` is a Federated Identity id that comes in as a part of the request. This is set after our user has been authenticated via the User Pool. We are going to expand more on this in the coming chapters when we set up our Cognito Identity Pool. However, if you want to use the user's User Pool user Id; take a look at the Mapping Cognito Identity Id and User Pool Id (/chapters/mapping-cognito-identity-id-and-user-pool-id.html) chapter.
- Make a call to DynamoDB to put a new object with a generated `noteId` and the current date as the `createdAt`.
- Upon success, return the newly created note object with the HTTP status code `200` and response headers to enable **CORS (Cross-Origin Resource Sharing)**.
- And if the DynamoDB call fails then return an error with the HTTP status code `500`.

Configure the API Endpoint

Now let's define the API endpoint for our function.

✦ CHANGE Open the `serverless.yml` file and replace it with the following.

```
service: notes-app-api

# Create an optimized package for our functions
package:
  individually: true
```

```yaml
plugins:
  - serverless-bundle # Package our functions with Webpack
  - serverless-offline
  - serverless-dotenv-plugin # Load .env as environment variables

provider:
  name: aws
  runtime: nodejs10.x
  stage: prod
  region: us-east-1

  # These environment variables are made available to our functions
  # under process.env.
  environment:
    tableName: notes

  # 'iamRoleStatements' defines the permission policy for the Lambda function.
  # In this case Lambda functions are granted with permissions to access DynamoDB.
  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:DescribeTable
        - dynamodb:Query
        - dynamodb:Scan
        - dynamodb:GetItem
```

```
        - dynamodb:PutItem
        - dynamodb:UpdateItem
        - dynamodb:DeleteItem
      Resource: "arn:aws:dynamodb:us-east-1:*:*"

functions:
  # Defines an HTTP API endpoint that calls the main
function in create.js
  # - path: url path is /notes
  # - method: POST request
  # - cors: enabled CORS (Cross-Origin Resource
Sharing) for browser cross
  #     domain api call
  # - authorizer: authenticate using the AWS IAM role
  create:
    handler: create.main
    events:
      - http:
          path: notes
          method: post
          cors: true
          authorizer: aws_iam
```

Here we are adding our newly added create function to the configuration. We specify that it handles `post` requests at the `/notes` endpoint. This pattern of using a single Lambda function to respond to a single HTTP event is very much like the Microservices architecture (https://en.wikipedia.org/wiki/Microservices). We discuss this and a few other patterns in the chapter on organizing Serverless Framework projects (/chapters/organizing-serverless-projects.html). We set CORS support to true. This is because our frontend is going to be served from a different

domain. As the authorizer we are going to restrict access to our API based on the user's IAM credentials. We will touch on this and how our User Pool works with this, in the Cognito Identity Pool chapter.

The `environment:` block allows us to define environment variables for our Lambda function. These are made available under the `process.env` Node.js variable. In our specific case, we are using `process.env.tableName` to access the name of our DynamoDB table.

The `iamRoleStatements` section is telling AWS which resources our Lambda functions have access to. In this case we are saying that our Lambda functions can carry out the above listed actions on DynamoDB. We specify DynamoDB using `arn:aws:dynamodb:us-east-1:*:*`. This is roughly pointing to every DynamoDB table in the `us-east-1` region. We can be more specific here by specifying the table name but we'll leave this as an exercise for the reader. Just make sure to use the region that the DynamoDB table was created in, as this can be a common source of issues later on. For us the region is `us-east-1`.

Test

Now we are ready to test our new API. To be able to test it on our local we are going to mock the input parameters.

✦ CHANGE In our project root, create a `mocks/` directory.

```
$ mkdir mocks
```

✦ CHANGE Create a `mocks/create-event.json` file and add the following.

```
{
  "body": "{\"content\":\"hello
world\",\"attachment\":\"hello.jpg\"}",
  "requestContext": {
    "identity": {
      "cognitoIdentityId": "USER-SUB-1234"
    }
  }
}
```

You might have noticed that the `body` and `requestContext` fields are the ones we used in our create function. In this case the `cognitoIdentityId` field is just a string we are going to use as our `userId`. We can use any string here; just make sure to use the same one when we test our other functions.

And to invoke our function we run the following in the root directory.

```
$ serverless invoke local --function create --path
mocks/create-event.json
```

If you have multiple profiles for your AWS SDK credentials, you will need to explicitly pick one. Use the following command instead:

```
$ AWS_PROFILE=myProfile serverless invoke local --
function create --path mocks/create-event.json
```

Where `myProfile` is the name of the AWS profile you want to use. If you need more info on how to work with AWS profiles in Serverless, refer to our Configure multiple AWS profiles (/chapters/configure-multiple-aws-profiles.html) chapter.

The response should look similar to this.

```
{
  statusCode: 200,
  headers: {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Credentials': true
  },
  body: '{"userId":"USER-SUB-1234","noteId":"578eb840-
f70f-11e6-9d1a-1359b3b22944","content":"hello
world","attachment":"hello.jpg","createdAt":1487800950620

}
```

Make a note of the `noteId` in the response. We are going to use this newly created note in the next chapter.

Refactor Our Code

Before we move on to the next chapter, let's quickly refactor the code since we are going to be doing much of the same for all of our APIs.

✦ CHANGE In our project root, create a `libs/` directory.

```
$ mkdir libs
$ cd libs
```

✦ CHANGE And create a `libs/response-lib.js` file.

```
export function success(body) {
  return buildResponse(200, body);
```

```
}

export function failure(body) {
  return buildResponse(500, body);
}

function buildResponse(statusCode, body) {
  return {
    statusCode: statusCode,
    headers: {
      "Access-Control-Allow-Origin": "*",
      "Access-Control-Allow-Credentials": true
    },
    body: JSON.stringify(body)
  };
}
```

This will manage building the response objects for both success and failure cases with the proper HTTP status code and headers.

✦ CHANGE Again inside `libs/`, create a `dynamodb-lib.js` file.

```
import AWS from "aws-sdk";

export function call(action, params) {
  const dynamoDb = new AWS.DynamoDB.DocumentClient();

  return dynamoDb[action](params).promise();
}
```

Here we are using the promise form of the DynamoDB methods. Promises

are a method for managing asynchronous code that serve as an alternative to the standard callback function syntax. It will make our code a lot easier to read.

✦ CHANGE Now, we'll go back to our `create.js` and use the helper functions we created. Replace our `create.js` with the following.

```
import uuid from "uuid";
import * as dynamoDbLib from "./libs/dynamodb-lib";
import { success, failure } from "./libs/response-lib";

export async function main(event, context) {
  const data = JSON.parse(event.body);
  const params = {
    TableName: "notes",
    Item: {
      userId:
event.requestContext.identity.cognitoIdentityId,
      noteId: uuid.v1(),
      content: data.content,
      attachment: data.attachment,
      createdAt: Date.now()
    }
  };

  try {
    await dynamoDbLib.call("put", params);
    return success(params.Item);
  } catch (e) {
    return failure({ status: false });
  }
```

```
}
```

We are also using the `async/await` pattern here to refactor our Lambda function. This allows us to return once we are done processing; instead of using the callback function.

Next, we are going to write the API to get a note given its id.

Common Issues

- Response `statusCode: 500`

  If you see a `statusCode: 500` response when you invoke your function, here is how to debug it. The error is generated by our code in the `catch` block. Adding a `console.log` like so, should give you a clue about what the issue is.

  ```
  catch(e) {
    console.log(e);
    return failure({status: false});
  }
  ```

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/add-a-create-note-api/125)

Add a Get Note API

Now that we created a note and saved it to our database. Let's add an API to retrieve a note given its id.

Add the Function

✦ CHANGE Create a new file `get.js` and paste the following code

```
import * as dynamoDbLib from "./libs/dynamodb-lib";
import { success, failure } from "./libs/response-lib";

export async function main(event, context) {
  const params = {
    TableName: process.env.tableName,
    // 'Key' defines the partition key and sort key of the item to be retrieved
    // - 'userId': Identity Pool identity id of the authenticated user
    // - 'noteId': path parameter
    Key: {
      userId: event.requestContext.identity.cognitoIdentityId,
      noteId: event.pathParameters.id
    }
  };
```

```
  try {
    const result = await dynamoDbLib.call("get",
params);
    if (result.Item) {
      // Return the retrieved item
      return success(result.Item);
    } else {
      return failure({ status: false, error: "Item not
found." });
    }
  } catch (e) {
    return failure({ status: false });
  }
}
```

This follows exactly the same structure as our previous `create.js` function. The major difference here is that we are doing a `dynamoDbLib.call('get', params)` to get a note object given the `noteId` and `userId` that is passed in through the request.

Configure the API Endpoint

✦ CHANGE Open the `serverless.yml` file and append the following to it.

```
  get:
    # Defines an HTTP API endpoint that calls the main
function in get.js
    # - path: url path is /notes/{id}
    # - method: GET request
    handler: get.main
```

```
    events:
      - http:
          path: notes/{id}
          method: get
          cors: true
          authorizer: aws_iam
```

Make sure that this block is indented exactly the same way as the preceding `create` block.

This defines our get note API. It adds a GET request handler with the endpoint `/notes/{id}`.

Test

To test our get note API we need to mock passing in the `noteId` parameter. We are going to use the `noteId` of the note we created in the previous chapter and add in a `pathParameters` block to our mock. So it should look similar to the one below. Replace the value of `id` with the id you received when you invoked the previous `create.js` function.

✦ CHANGE Create a `mocks/get-event.json` file and add the following.

```
{
  "pathParameters": {
    "id": "578eb840-f70f-11e6-9d1a-1359b3b22944"
  },
  "requestContext": {
    "identity": {
      "cognitoIdentityId": "USER-SUB-1234"
    }
```

```
  }
}
```

And we invoke our newly created function.

```
$ serverless invoke local --function get --path
mocks/get-event.json
```

The response should look similar to this.

```
{
  statusCode: 200,
  headers: {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Credentials': true
  },
  body: '{"attachment":"hello.jpg","content":"hello
world","createdAt":1487800950620,"noteId":"578eb840-
f70f-11e6-9d1a-1359b3b22944","userId":"USER-SUB-1234"}'
}
```

Next, let's create an API to list all the notes a user has.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/add-a-get-note-api/132)

Add a List All the Notes API

Now we are going to add an API that returns a list of all the notes a user has.

Add the Function

✦ CHANGE Create a new file called `list.js` with the following.

```
import * as dynamoDbLib from "./libs/dynamodb-lib";
import { success, failure } from "./libs/response-lib";

export async function main(event, context) {
  const params = {
    TableName: process.env.tableName,
    // 'KeyConditionExpression' defines the condition
for the query
    // - 'userId = :userId': only return items with
matching 'userId'
    //   partition key
    // 'ExpressionAttributeValues' defines the value in
the condition
    // - ':userId': defines 'userId' to be Identity
Pool identity id
    //   of the authenticated user
    KeyConditionExpression: "userId = :userId",
    ExpressionAttributeValues: {
```

```
      ":userId":
event.requestContext.identity.cognitoIdentityId
    }
  };

  try {
    const result = await dynamoDbLib.call("query",
params);
    // Return the matching list of items in response
body
    return success(result.Items);
  } catch (e) {
    return failure({ status: false });
  }
}
```

This is pretty much the same as our `get.js` except we only pass in the `userId` in the DynamoDB `query` call.

Configure the API Endpoint

✦ CHANGE Open the `serverless.yml` file and append the following.

```
  list:
    # Defines an HTTP API endpoint that calls the main
function in list.js
    # - path: url path is /notes
    # - method: GET request
    handler: list.main
    events:
      - http:
```

```
        path: notes
        method: get
        cors: true
        authorizer: aws_iam
```

This defines the `/notes` endpoint that takes a GET request.

Test

CHANGE Create a `mocks/list-event.json` file and add the following.

```
{
  "requestContext": {
    "identity": {
      "cognitoIdentityId": "USER-SUB-1234"
    }
  }
}
```

And invoke our function from the root directory of the project.

```
$ serverless invoke local --function list --path
mocks/list-event.json
```

The response should look similar to this.

```
{
  statusCode: 200,
  headers: {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Credentials': true
```

```
  },
  body: '[{"attachment":"hello.jpg","content":"hello
world","createdAt":1487800950620,"noteId":"578eb840-
f70f-11e6-9d1a-1359b3b22944","userId":"USER-SUB-
1234"}]'
}
```

Note that this API returns an array of note objects as opposed to the `get.js` function that returns just a single note object.

Next we are going to add an API to update a note.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/add-a-list-all-the-notes-api/147)

Add an Update Note API

Now let's create an API that allows a user to update a note with a new note object given its id.

Add the Function

✦ CHANGE Create a new file `update.js` and paste the following code

```
import * as dynamoDbLib from "./libs/dynamodb-lib";
import { success, failure } from "./libs/response-lib";

export async function main(event, context) {
  const data = JSON.parse(event.body);
  const params = {
    TableName: process.env.tableName,
    // 'Key' defines the partition key and sort key of
the item to be updated
    // - 'userId': Identity Pool identity id of the
authenticated user
    // - 'noteId': path parameter
    Key: {
      userId:
event.requestContext.identity.cognitoIdentityId,
      noteId: event.pathParameters.id
    },
    // 'UpdateExpression' defines the attributes to be
```

```
updated
    // 'ExpressionAttributeValues' defines the value in
the update expression
    UpdateExpression: "SET content = :content,
attachment = :attachment",
    ExpressionAttributeValues: {
      ":attachment": data.attachment || null,
      ":content": data.content || null
    },
    // 'ReturnValues' specifies if and how to return
the item's attributes,
    // where ALL_NEW returns all attributes of the item
after the update; you
    // can inspect 'result' below to see how it works
with different settings
    ReturnValues: "ALL_NEW"
  };

  try {
    await dynamoDbLib.call("update", params);
    return success({ status: true });
  } catch (e) {
    return failure({ status: false });
  }
}
```

This should look similar to the `create.js` function. Here we make an `update` DynamoDB call with the new `content` and `attachment` values in the `params`.

Configure the API Endpoint

CHANGE Open the `serverless.yml` file and append the following to it.

```
  update:
    # Defines an HTTP API endpoint that calls the main
function in update.js
    # - path: url path is /notes/{id}
    # - method: PUT request
    handler: update.main
    events:
      - http:
          path: notes/{id}
          method: put
          cors: true
          authorizer: aws_iam
```

Here we are adding a handler for the PUT request to the `/notes/{id}` endpoint.

Test

CHANGE Create a `mocks/update-event.json` file and add the following.

Also, don't forget to use the `noteId` of the note we have been using in place of the `id` in the `pathParameters` block.

```
{
  "body": "{\"content\":\"new
world\",\"attachment\":\"new.jpg\"}",
  "pathParameters": {
```

```
    "id": "578eb840-f70f-11e6-9d1a-1359b3b22944"
  },
  "requestContext": {
    "identity": {
      "cognitoIdentityId": "USER-SUB-1234"
    }
  }
}
```

And we invoke our newly created function from the root directory.

```
$ serverless invoke local --function update --path
mocks/update-event.json
```

The response should look similar to this.

```
{
  statusCode: 200,
  headers: {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Credentials': true
  },
  body: '{"status":true}'
}
```

Next we are going to add an API to delete a note given its id.

For help and discussion

Comments on this chapter

```
(https://discourse.serverless-
stack.com/t/add-an-update-note-api/144)
```

Add a Delete Note API

Finally, we are going to create an API that allows a user to delete a given note.

Add the Function

✦ CHANGE Create a new file `delete.js` and paste the following code

```
import * as dynamoDbLib from "./libs/dynamodb-lib";
import { success, failure } from "./libs/response-lib";

export async function main(event, context) {
  const params = {
    TableName: process.env.tableName,
    // 'Key' defines the partition key and sort key of
the item to be removed
    // - 'userId': Identity Pool identity id of the
authenticated user
    // - 'noteId': path parameter
    Key: {
      userId:
event.requestContext.identity.cognitoIdentityId,
      noteId: event.pathParameters.id
    }
  };
```

```
  try {
    await dynamoDbLib.call("delete", params);
    return success({ status: true });
  } catch (e) {
    return failure({ status: false });
  }
}
```

This makes a DynamoDB `delete` call with the `userId` & `noteId` key to delete the note.

Configure the API Endpoint

CHANGE Open the `serverless.yml` file and append the following to it.

```
  delete:
    # Defines an HTTP API endpoint that calls the main function in delete.js
    # - path: url path is /notes/{id}
    # - method: DELETE request
    handler: delete.main
    events:
      - http:
          path: notes/{id}
          method: delete
          cors: true
          authorizer: aws_iam
```

This adds a DELETE request handler to the `/notes/{id}` endpoint.

Test

CHANGE Create a `mocks/delete-event.json` file and add the following.

Just like before we'll use the `noteId` of our note in place of the `id` in the `pathParameters` block.

```
{
  "pathParameters": {
    "id": "578eb840-f70f-11e6-9d1a-1359b3b22944"
  },
  "requestContext": {
    "identity": {
      "cognitoIdentityId": "USER-SUB-1234"
    }
  }
}
```

Invoke our newly created function from the root directory.

```
$ serverless invoke local --function delete --path
mocks/delete-event.json
```

And the response should look similar to this.

```
{
  statusCode: 200,
  headers: {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Credentials': true
```

```
  },
  body: '{"status":true}'
}
```

Now that our APIs are complete; we are almost ready to deploy them.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/add-a-delete-note-api/153)

Working with 3rd Party APIs

So far we've created a basic CRUD (create, read, update, and delete) API. We are going to make a small addition to this by adding an endpoint that works with a 3rd party API. This section is also going to illustrate how to work with environment variables and how to accept credit card payments using Stripe.

A common extension of Serverless Stack (that we have noticed) is to add a billing API that works with Stripe. In the case of our notes app we are going to allow our users to pay a fee for storing a certain number of notes. The flow is going to look something like this:

1. The user is going to select the number of notes he wants to store and puts in his credit card information.
2. We are going to generate a one time token by calling the Stripe SDK on the frontend to verify that the credit card info is valid.
3. We will then call an API passing in the number of notes and the generated token.
4. The API will take the number of notes, figure out how much to charge (based on our pricing plan), and call the Stripe API to charge our user.

We aren't going to do much else in the way of storing this info in our database. We'll leave that as an exercise for the reader.

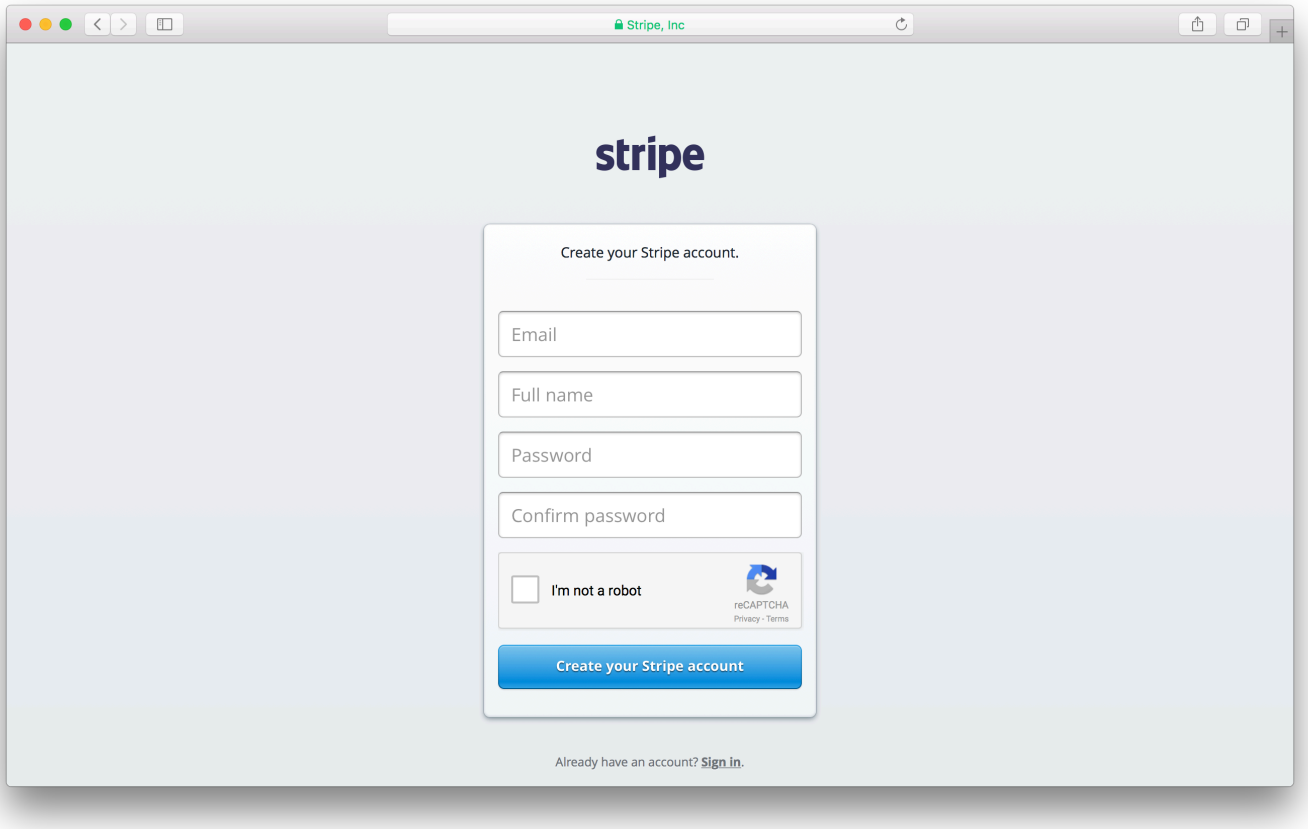
Let's get started with first setting up our Stripe account.

For help and discussion

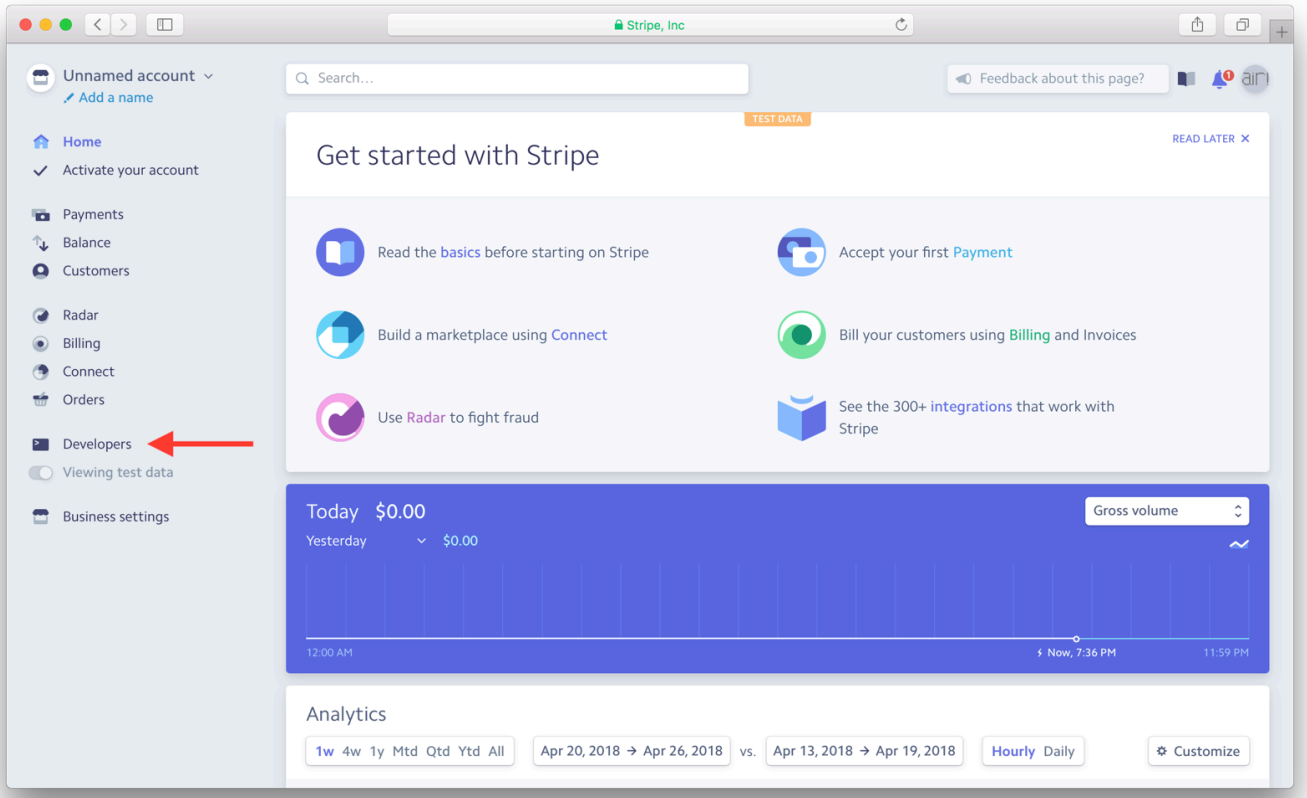
Comments on this chapter (https://discourse.serverless-stack.com/t/working-with-3rd-party-apis/168)

Setup a Stripe Account

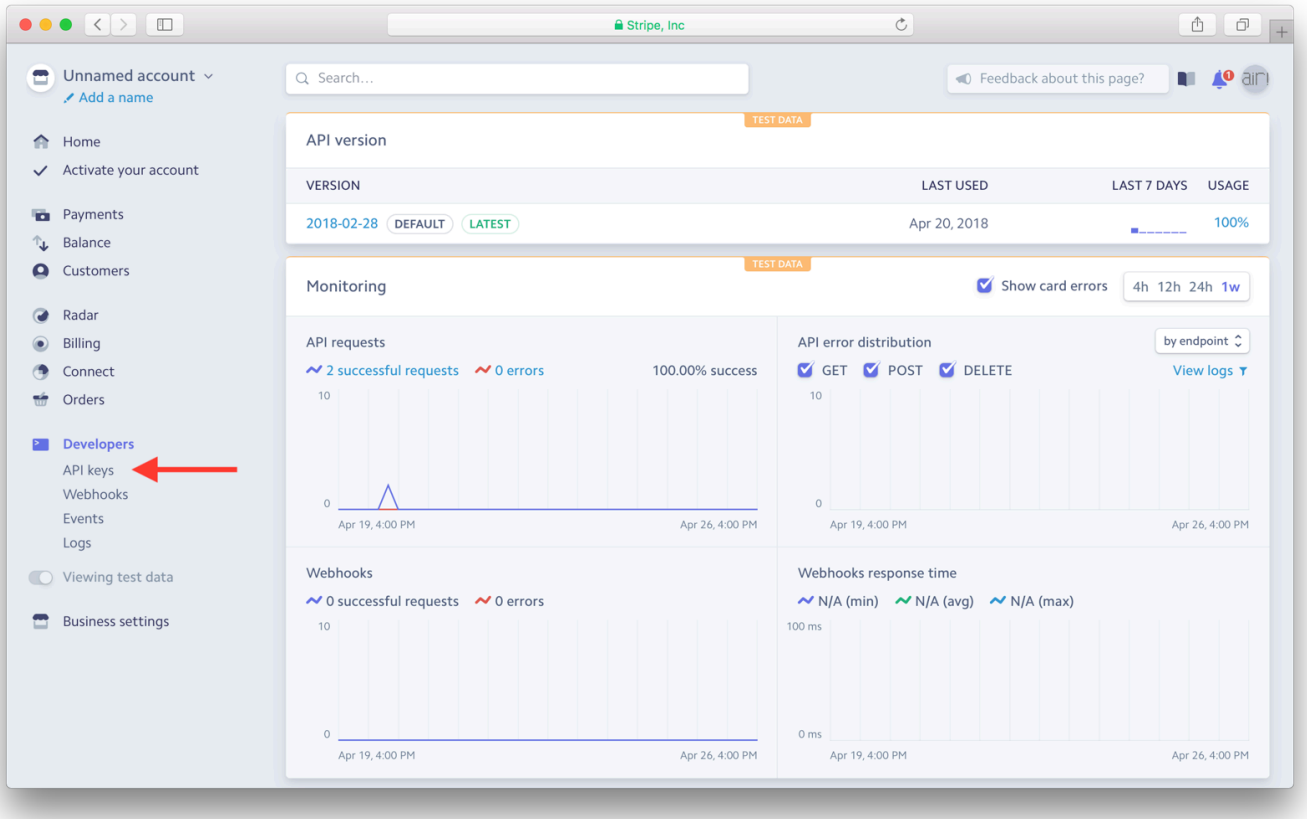
Let's start by creating a free Stripe account. Head over to Stripe (https://dashboard.stripe.com/register) and register for an account.



Once signed in, click the **Developers** link on the left.



And hit **API keys**.

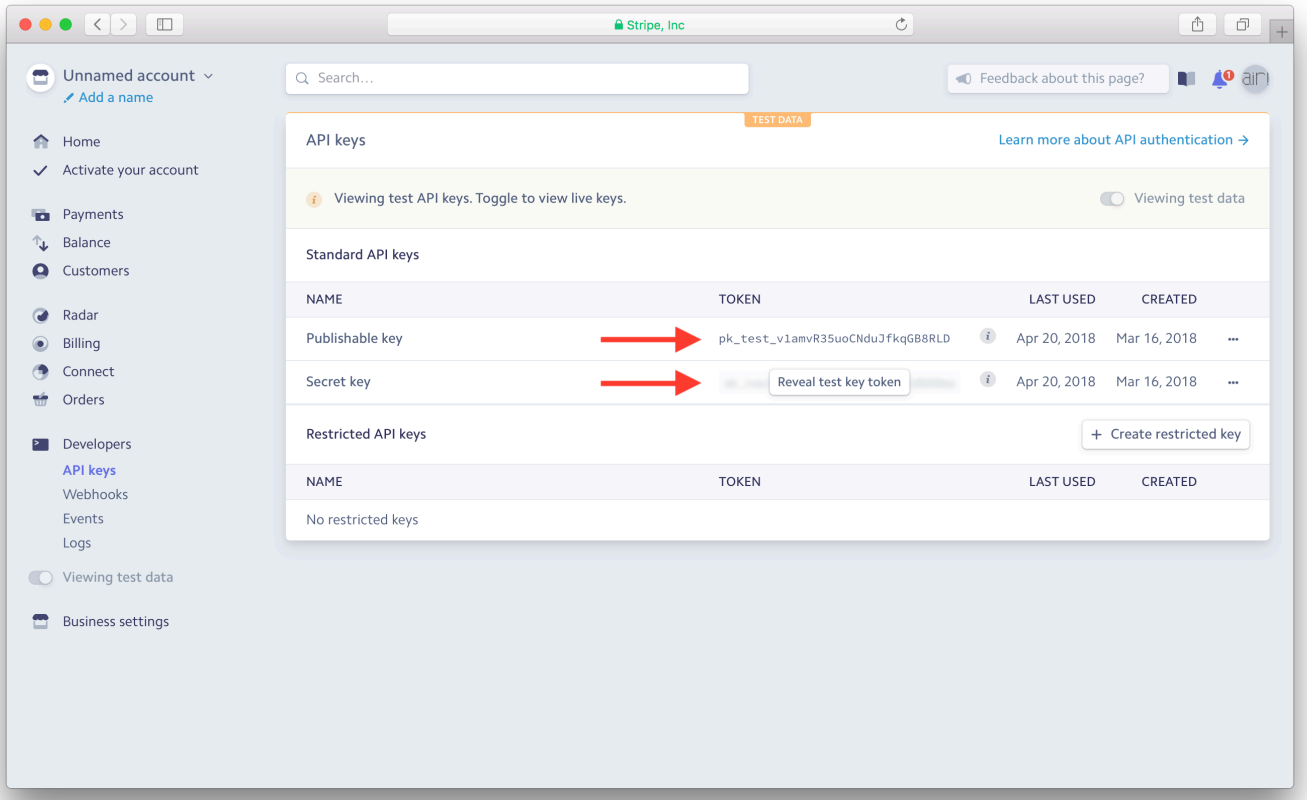


The first thing to note here is that we are working with a test version of

API keys. To create the live version, you'd need to verify your email address and business details to activate your account. For the purpose of this guide we'll continue working with our test version.

The second thing to note is that we need to generate the **Publishable key** and the **Secret key**. The Publishable key is what we are going to use in our frontend client with the Stripe SDK. And the Secret key is what we are going to use in our API when asking Stripe to charge our user. As denoted, the Publishable key is public while the Secret key needs to stay private.

Hit the **Reveal test key token**.



Make a note of both the **Publishable key** and the **Secret key**. We are going to be using these later.

Next let's create our billing API.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/setup-a-stripe-account/169)

Add a Billing API

Now let's get started with creating our billing API. It is going to take a Stripe token and the number of notes the user wants to store.

Add a Billing Lambda

✦ CHANGE Start by installing the Stripe NPM package. Run the following in the root of our project.

```
$ npm install --save stripe
```

✦ CHANGE Next, add the following to `billing.js`.

```
import stripePackage from "stripe";
import { calculateCost } from "./libs/billing-lib";
import { success, failure } from "./libs/response-lib";

export async function main(event, context) {
  const { storage, source } = JSON.parse(event.body);
  const amount = calculateCost(storage);
  const description = "Scratch charge";

  // Load our secret key from the  environment
variables
  const stripe =
stripePackage(process.env.stripeSecretKey);
```

```
  try {
    await stripe.charges.create({
      source,
      amount,
      description,
      currency: "usd"
    });
    return success({ status: true });
  } catch (e) {
    return failure({ message: e.message });
  }
}
```

Most of this is fairly straightforward but let's go over it quickly:

- We get the `storage` and `source` from the request body. The `storage` variable is the number of notes the user would like to store in his account. And `source` is the Stripe token for the card that we are going to charge.
- We are using a `calculateCost(storage)` function (that we are going to add soon) to figure out how much to charge a user based on the number of notes that are going to be stored.
- We create a new Stripe object using our Stripe Secret key. We are going to get this as an environment variable. We do not want to put our secret keys in our code and commit that to Git. This is a security issue.
- Finally, we use the `stripe.charges.create` method to charge the user and respond to the request if everything went through

successfully.

Add the Business Logic

Now let's implement our `calculateCost` method. This is primarily our *business logic*.

✦ CHANGE Create a `libs/billing-lib.js` and add the following.

```
export function calculateCost(storage) {
  const rate = storage <= 10
    ? 4
    : storage <= 100
      ? 2
      : 1;

  return rate * storage * 100;
}
```

This is basically saying that if a user wants to store 10 or fewer notes, we'll charge them $4 per note. For 11 to 100 notes, we'll charge $2 and any more than 100 is $1 per note. Since Stripe expects us to provide the amount in pennies (the currency's smallest unit) we multiply the result by 100. Clearly, our serverless infrastructure might be cheap but our service isn't!

Configure the API Endpoint

Let's add a reference to our new API and Lambda function.

✦ CHANGE Open the `serverless.yml` file and append the following to

it.

```
  billing:
    # Defines an HTTP API endpoint that calls the main
function in billing.js
    # - path: url path is /billing
    # - method: POST request
    handler: billing.main
    events:
      - http:
          path: billing
          method: post
          cors: true
          authorizer: aws_iam
```

Make sure this is **indented correctly**. This block falls under the `functions` block.

Now before we can test our API we need to load our Stripe secret key in our environment.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/add-a-billing-api/170)

Load Secrets from .env

As we had previously mentioned, we do not want to store our secret environment variables in our code. In our case it is the Stripe secret key. In this chapter, we'll look at how to do that.

We have a `env.example` file for this exact purpose.

✦ CHANGE Start by renaming the `env.example` file to `.env`.

```
$ mv env.example .env
```

✦ CHANGE Replace its contents with the following.

```
STRIPE_SECRET_KEY=STRIPE_TEST_SECRET_KEY
```

Make sure to replace the `STRIPE_TEST_SECRET_KEY` with the **Secret key** from the Setup a Stripe account (/chapters/setup-a-stripe-account.html) chapter.

We are using the serverless-dotenv-plugin (https://github.com/colynb/serverless-dotenv-plugin) to load these as an environment variable when our Lambda function runs locally. This allows us to reference them in our `serverless.yml`. We will not be commiting the `.env` file to Git as we are only going to use these locally. When we look at automating deployments, we'll be adding our secrets to the CI, so they'll be made available through there instead.

Next, let's add a reference to these.

✦ CHANGE And add the following in the `environment:` block in your `serverless.yml`.

```
    stripeSecretKey: ${env:STRIPE_SECRET_KEY}
```

Your `environment:` block should look like this:

```
  # These environment variables are made available to
our functions
  # under process.env.
  environment:
    tableName: notes
    stripeSecretKey: ${env:STRIPE_SECRET_KEY}
```

A quick explanation on the above:

- The `STRIPE_SECRET_KEY` from the `.env` file above gets loading as an environment variable when we test our code locally.
- This allows us to add a Lambda environment variables called `stripeSecretKey`. We do this using the `stripeSecretKey: ${env:STRIPE_SECRET_KEY}` line. And just like our `tableName` environment variable, we can reference it in our Lambda function using `process.env.stripeSecretKey`.

Now we need to ensure that we don't commit our `env.yml` file to git. The starter project that we are using has the following in the `.gitignore`.

```
# Env
```

```
.env
```

This will tell Git to not commit this file.

Now we are ready to test our billing API.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/load-secrets-from-env-yml/171)

Test the Billing API

Now that we have our billing API all set up, let's do a quick test in our local environment.

✦ CHANGE Create a `mocks/billing-event.json` file and add the following.

```
{
  "body": "{\"source\":\"tok_visa\",\"storage\":21}",
  "requestContext": {
    "identity": {
      "cognitoIdentityId": "USER-SUB-1234"
    }
  }
}
```

We are going to be testing with a Stripe test token called `tok_visa` and with `21` as the number of notes we want to store. You can read more about the Stripe test cards and tokens in the Stripe API Docs here (https://stripe.com/docs/testing#cards).

Let's now invoke our billing API by running the following in our project root.

```
$ serverless invoke local --function billing --path
mocks/billing-event.json
```

The response should look similar to this.

```
{
    "statusCode": 200,
    "headers": {
        "Access-Control-Allow-Origin": "*",
        "Access-Control-Allow-Credentials": true
    },
    "body": "{"status":true}"
}
```

Now that we have our new billing API ready. Let's look at how to setup unit tests to ensure that our business logic has been configured correctly.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/test-the-billing-api/172)

Unit Tests in Serverless

So we have some simple business logic that figures out exactly how much to charge our user based on the number of notes they want to store. We want to make sure that we test all the possible cases for this before we start charging people. To do this we are going to configure unit tests for our Serverless Framework project. However, if you are looking for other strategies to test your Serverless applications, we talk about them in detail here (https://seed.run/blog/testing-your-serverless-app).

We are going to use Jest (https://facebook.github.io/jest/) for this and it is already a part of our starter project (https://github.com/AnomalyInnovations/serverless-nodejs-starter).

However, if you are starting a new Serverless Framework project. Add Jest to your dev dependencies by running the following.

```
$ npm install --save-dev jest
```

And update the `scripts` block in your `package.json` with the following:

```
"scripts": {
  "test": "jest"
},
```

This will allow you to run your tests using the command `npm test`.

Alternatively, if you are using the serverless-bundle (https://github.com/AnomalyInnovations/serverless-bundle) plugin to package your functions, it comes with a built-in script to transpile your code and run your tests. Add the following to your `package.json` instead.

```
"scripts": {
  "test": "serverless-bundle test"
},
```

Add Unit Tests

CHANGE Now create a new file in `tests/billing.test.js` and add the following.

```
import { calculateCost } from "../libs/billing-lib";

test("Lowest tier", () => {
  const storage = 10;

  const cost = 4000;
  const expectedCost = calculateCost(storage);

  expect(cost).toEqual(expectedCost);
});

test("Middle tier", () => {
  const storage = 100;

  const cost = 20000;
  const expectedCost = calculateCost(storage);
```

```
  expect(cost).toEqual(expectedCost);
});

test("Highest tier", () => {
  const storage = 101;

  const cost = 10100;
  const expectedCost = calculateCost(storage);

  expect(cost).toEqual(expectedCost);
});
```

This should be straightforward. We are adding 3 tests. They are testing the different tiers of our pricing structure. We test the case where a user is trying to store 10, 100, and 101 notes. And comparing the calculated cost to the one we are expecting. You can read more about using Jest in the Jest docs here (https://facebook.github.io/jest/docs/en/getting-started.html).

You might have noticed a `handler.test.js` file in the `tests/` directory. This was a part of our starter that we can now remove.

Remove Unused Files

✦ CHANGE Remove the starter files by running the following.

```
$ rm handler.js
$ rm tests/handler.test.js
```

Run tests

And we can run our tests by using the following command in the root of our project.

```
$ npm test
```

You should see something like this:

```
 PASS  tests/billing.test.js
  ✓ Lowest tier (4ms)
  ✓ Middle tier
  ✓ Highest tier (1ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.665s
Ran all test suites.
```

And that's it! We have unit tests all configured.

Now we are almost ready to deploy our backend.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/unit-tests-in-serverless/173)

Handle API Gateway CORS Errors

Before we deploy our APIs we need to do one last thing to set them up. We need to add CORS headers to API Gateway errors. You might recall that back in the Add a create note API (/chapters/add-a-create-note-api.html) chapter, we added the CORS headers to our Lambda functions. However when we make an API request, API Gateway gets invoked before our Lambda functions. This means that if there is an error at the API Gateway level, the CORS headers won't be set.

Consequently, debugging such errors can be really hard. Our client won't be able to see the error message and instead will be presented with something like this:

```
No 'Access-Control-Allow-Origin' header is present on
the requested resource
```

These CORS related errors are one of the most common Serverless API errors. In this chapter, we are going to configure API Gateway to set the CORS headers in the case there is an HTTP error. We won't be able to test this right away, but it will really help when we work on our frontend client.

Create a Resource

To configure API Gateway errors we are going to add a few things to our

`serverless.yml`. By default, Serverless Framework (https://serverless.com) supports CloudFormation (https://aws.amazon.com/cloudformation/) to help us configure our API Gateway instance through code.

✦ CHANGE Let's create a directory to add our resources. We'll be adding to this later in the guide.

```
$ mkdir resources/
```

✦ CHANGE And add the following to `resources/api-gateway-errors.yml`.

```
Resources:
  GatewayResponseDefault4XX:
    Type: 'AWS::ApiGateway::GatewayResponse'
    Properties:
      ResponseParameters:
        gatewayresponse.header.Access-Control-Allow-
Origin: "'*'"
        gatewayresponse.header.Access-Control-Allow-
Headers: "'*'"
      ResponseType: DEFAULT_4XX
      RestApiId:
        Ref: 'ApiGatewayRestApi'
  GatewayResponseDefault5XX:
    Type: 'AWS::ApiGateway::GatewayResponse'
    Properties:
      ResponseParameters:
        gatewayresponse.header.Access-Control-Allow-
Origin: "'*'"
```

```
        gatewayresponse.header.Access-Control-Allow-
Headers: "'*'"
      ResponseType: DEFAULT_5XX
      RestApiId:
        Ref: 'ApiGatewayRestApi'
```

The above might look a little intimidating. It's a CloudFormation resource and its syntax tends to be fairly verbose. But the details here aren't too important. We are adding the CORS headers to the `ApiGatewayRestApi` resource in our app. The `GatewayResponseDefault4XX` is for 4xx errors, while `GatewayResponseDefault5XX` is for 5xx errors.

Include the Resource

Now let's include the above CloudFormation resource in our `serverless.yml`.

✦ CHANGE Add the following to the bottom of our `serverless.yml`.

```
# Create our resources with separate CloudFormation
templates
resources:
  # API Gateway Errors
  - ${file(resources/api-gateway-errors.yml)}
```

Make sure this is **indented correctly**. The `resources:` block is a top level property.

And that's it. We are ready to deploy our APIs.

Commit the Changes

CHANGE Let's commit our code so far and push it to GitHub.

```
$ git add .
$ git commit -m "Adding our Serverless API"
$ git push
```

CHECKPOINT

For reference, here is the code we have so far.

Backend Source :handle-api-gateway-cors-errors (https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/handle-api-gateway-cors-errors)

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/handle-api-gateway-cors-errors/780)

Deploy the APIs

Now that our APIs are complete, let's deploy them.

✦ CHANGE Run the following in your working directory.

```
$ serverless deploy
```

If you have multiple profiles for your AWS SDK credentials, you will need to explicitly pick one. Use the following command instead:

```
$ serverless deploy --aws-profile myProfile
```

Where `myProfile` is the name of the AWS profile you want to use. If you need more info on how to work with AWS profiles in Serverless, refer to our Configure multiple AWS profiles (/chapters/configure-multiple-aws-profiles.html) chapter.

Near the bottom of the output for this command, you will find the **Service Information**.

```
Service Information
service: notes-app-api
stage: prod
region: us-east-1
api keys:
  None
endpoints:
```

```
  POST - https://ly55wbovq4.execute-api.us-east-
1.amazonaws.com/prod/notes
  GET - https://ly55wbovq4.execute-api.us-east-
1.amazonaws.com/prod/notes/{id}
  GET - https://ly55wbovq4.execute-api.us-east-
1.amazonaws.com/prod/notes
  PUT - https://ly55wbovq4.execute-api.us-east-
1.amazonaws.com/prod/notes/{id}
  DELETE - https://ly55wbovq4.execute-api.us-east-
1.amazonaws.com/prod/notes/{id}
  POST - https://ly55wbovq4.execute-api.us-east-
1.amazonaws.com/prod/billing
functions:
  create: notes-app-api-prod-create
  get: notes-app-api-prod-get
  list: notes-app-api-prod-list
  update: notes-app-api-prod-update
  delete: notes-app-api-prod-delete
  billing: notes-app-api-prod-billing
```

This has a list of the API endpoints that were created. Make a note of these endpoints as we are going to use them later while creating our frontend. Also make a note of the region and the id in these endpoints, we are going to use them in the coming chapters. In our case, `us-east-1` is our API Gateway Region and `ly55wbovq4` is our API Gateway ID.

If you are running into some issues while deploying your app, we have a compilation of some of the most common Serverless errors (https://seed.run/docs/serverless-errors/) over on Seed (https://seed.run).

Deploy a Single Function

There are going to be cases where you might want to deploy just a single API endpoint as opposed to all of them. The `serverless deploy function` command deploys an individual function without going through the entire deployment cycle. This is a much faster way of deploying the changes we make.

For example, to deploy the list function again, we can run the following.

```
$ serverless deploy function -f list
```

Now before we test our APIs we have one final thing to set up. We need to ensure that our users can securely access the AWS resources we have created so far. Let's look at setting up a Cognito Identity Pool.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploy-the-apis/121)

Create a Cognito Identity Pool

Now that we have deployed our backend API; we almost have all the pieces we need for our backend. We have the User Pool that is going to store all of our users and help sign in and sign them up. We also have an S3 bucket that we will use to help our users upload files as attachments for their notes. The final piece that ties all these services together in a secure way is called Amazon Cognito Federated Identities.

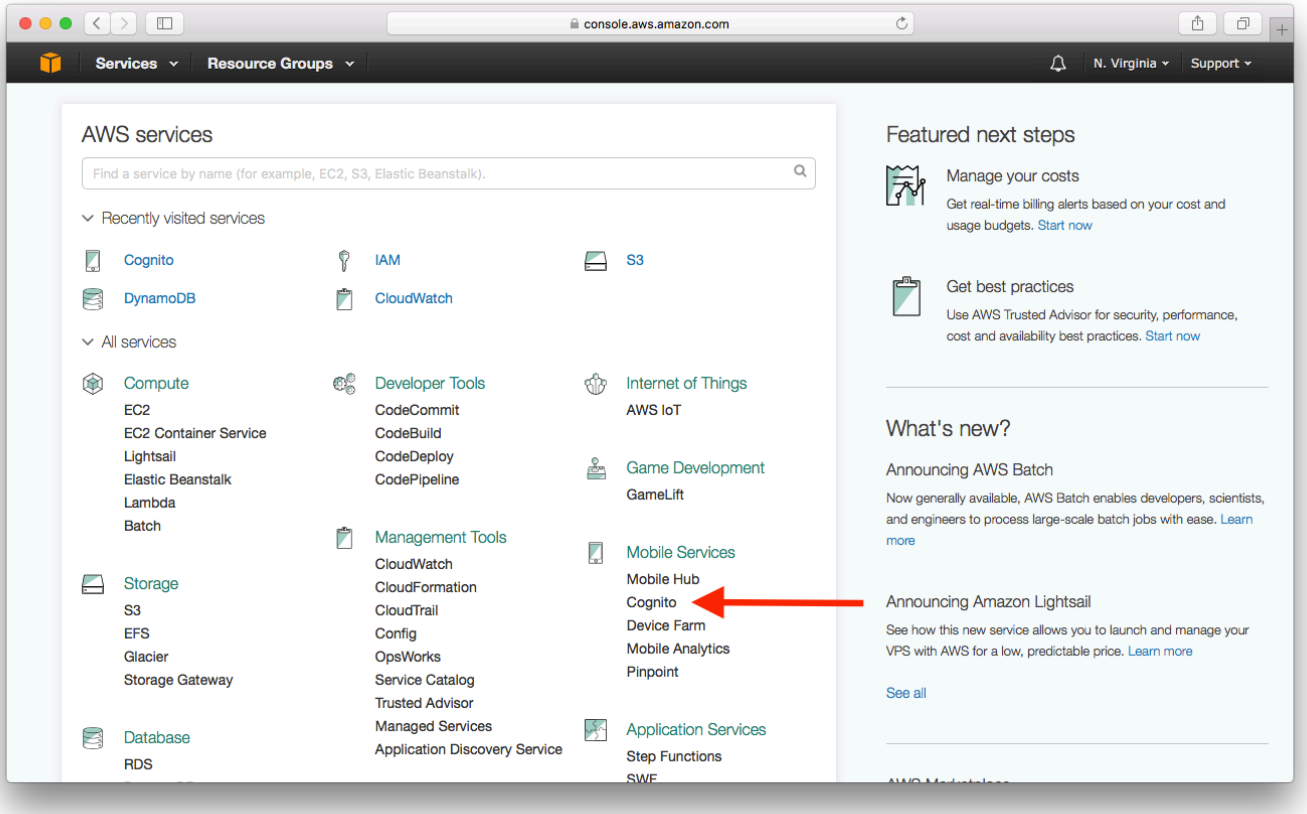
Amazon Cognito Federated Identities enables developers to create unique identities for your users and authenticate them with federated identity providers. With a federated identity, you can obtain temporary, limited-privilege AWS credentials to securely access other AWS services such as Amazon DynamoDB, Amazon S3, and Amazon API Gateway.

In this chapter, we are going to create a federated Cognito Identity Pool. We will be using our User Pool as the identity provider. We could also use Facebook, Google, or our own custom identity provider. Once a user is authenticated via our User Pool, the Identity Pool will attach an IAM Role to the user. We will define a policy for this IAM Role to grant access to the S3 bucket and our API. This is the Amazon way of securing your resources.

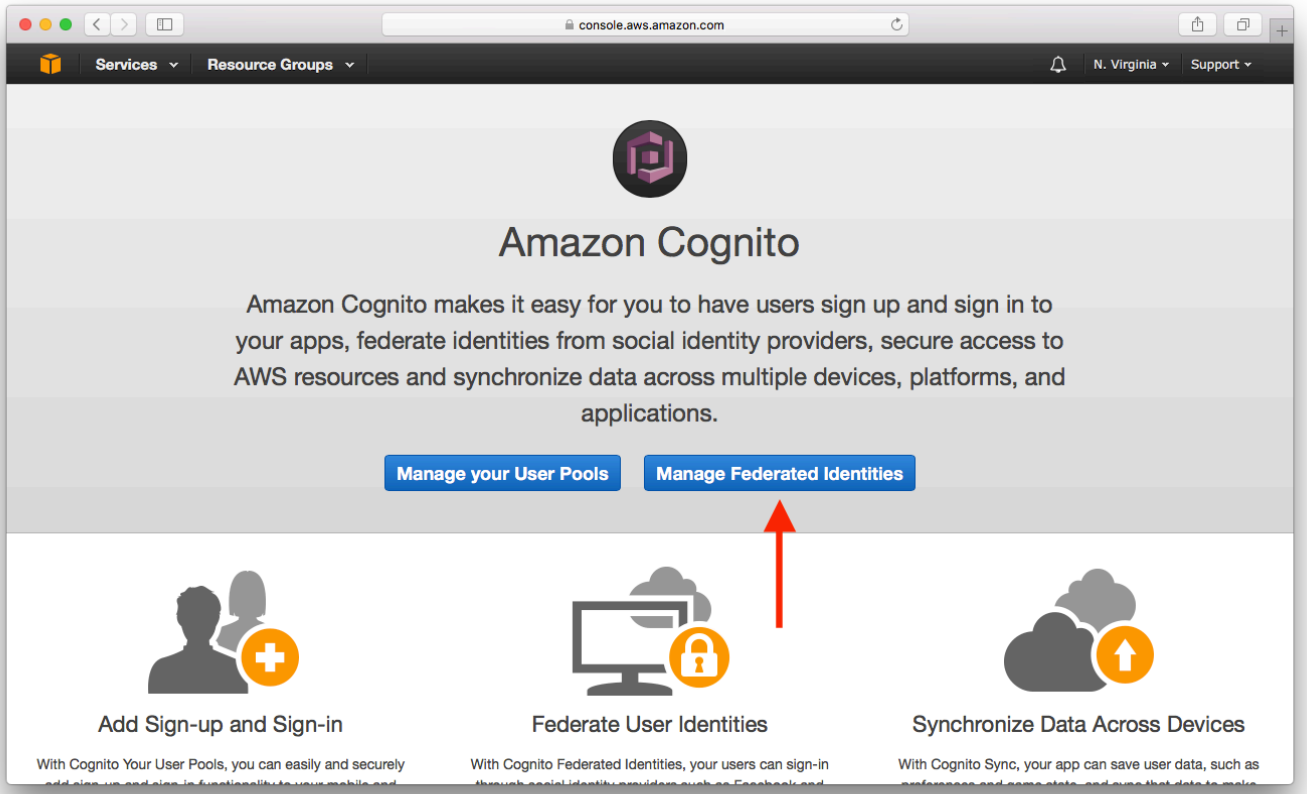
Let's get started.

Create Pool

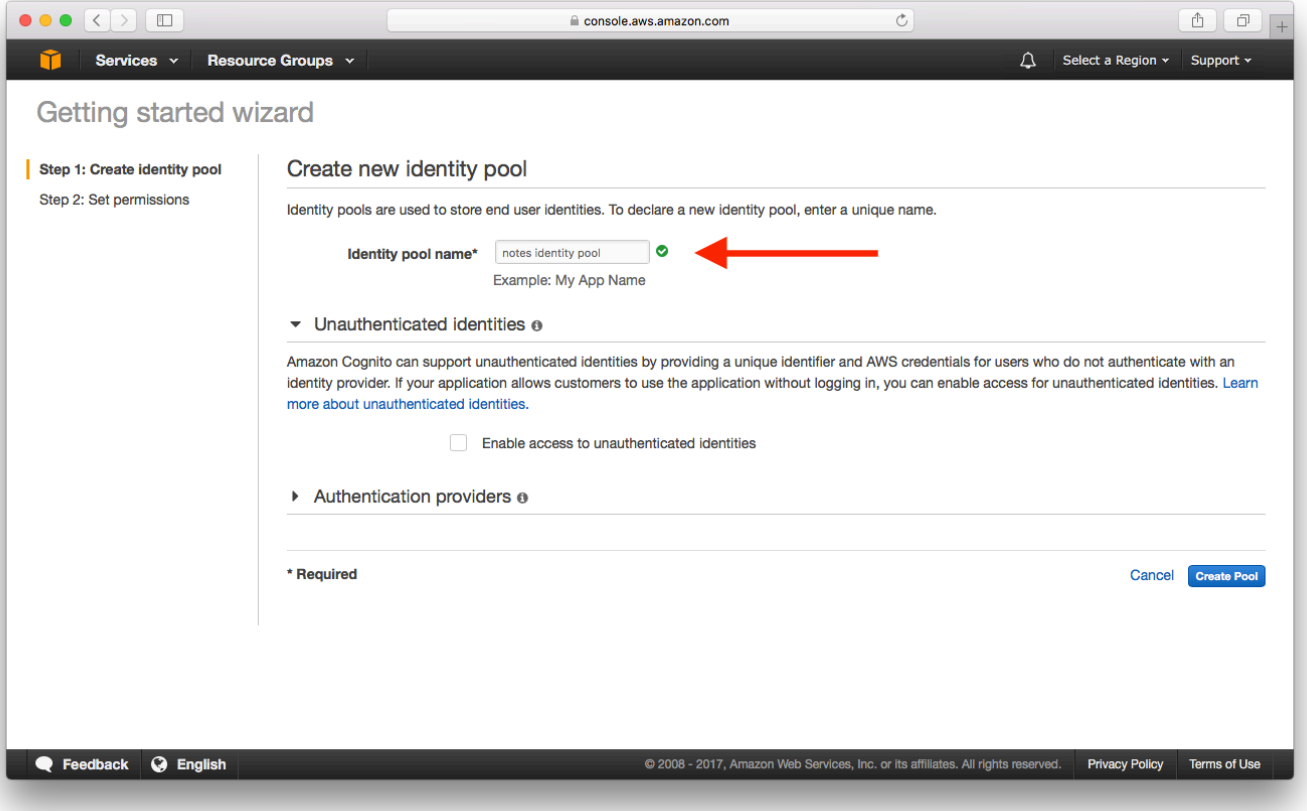
From your AWS Console (https://console.aws.amazon.com) and select **Cognito** from the list of services.



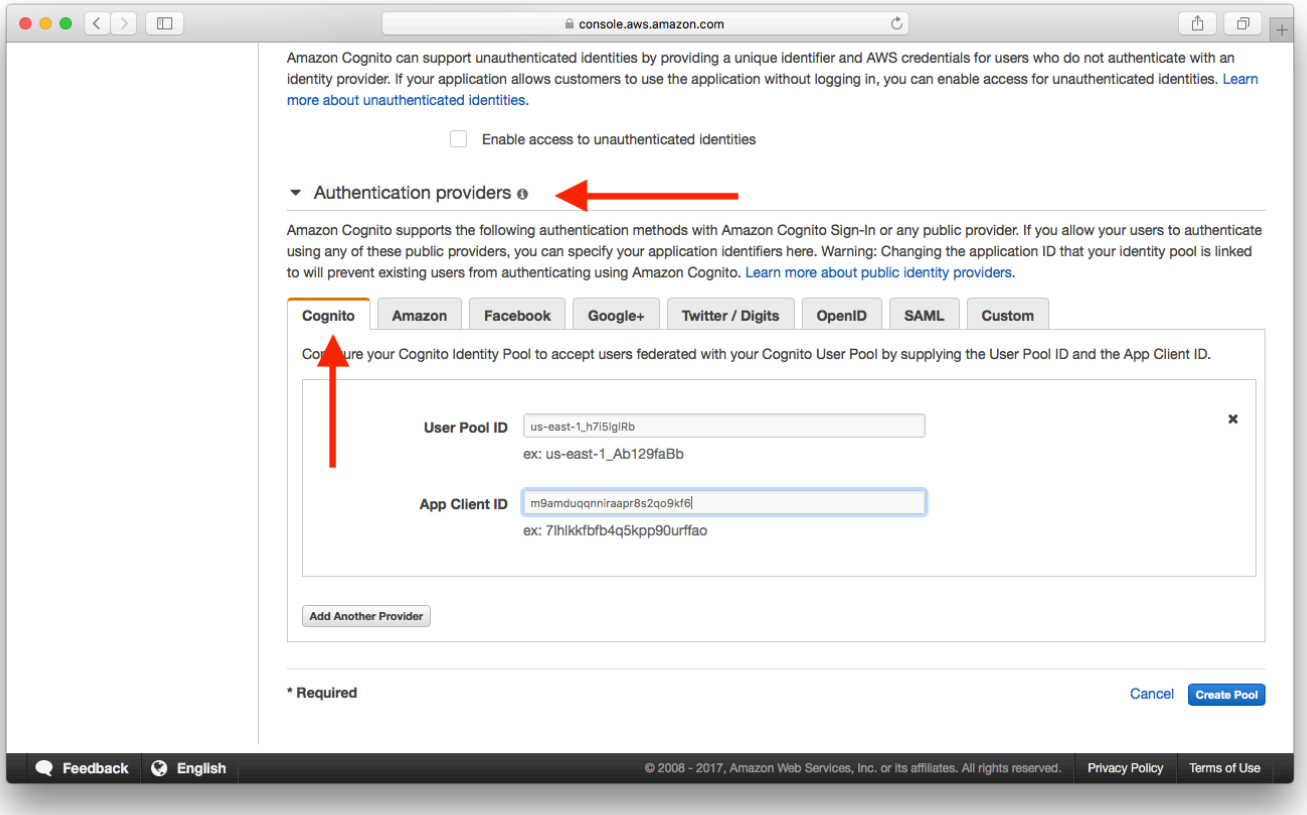
Select **Manage Federated Identities**.



Enter an **Identity pool name**. If you have any existing Identity Pools, you'll need to click the **Create new identity pool** button.



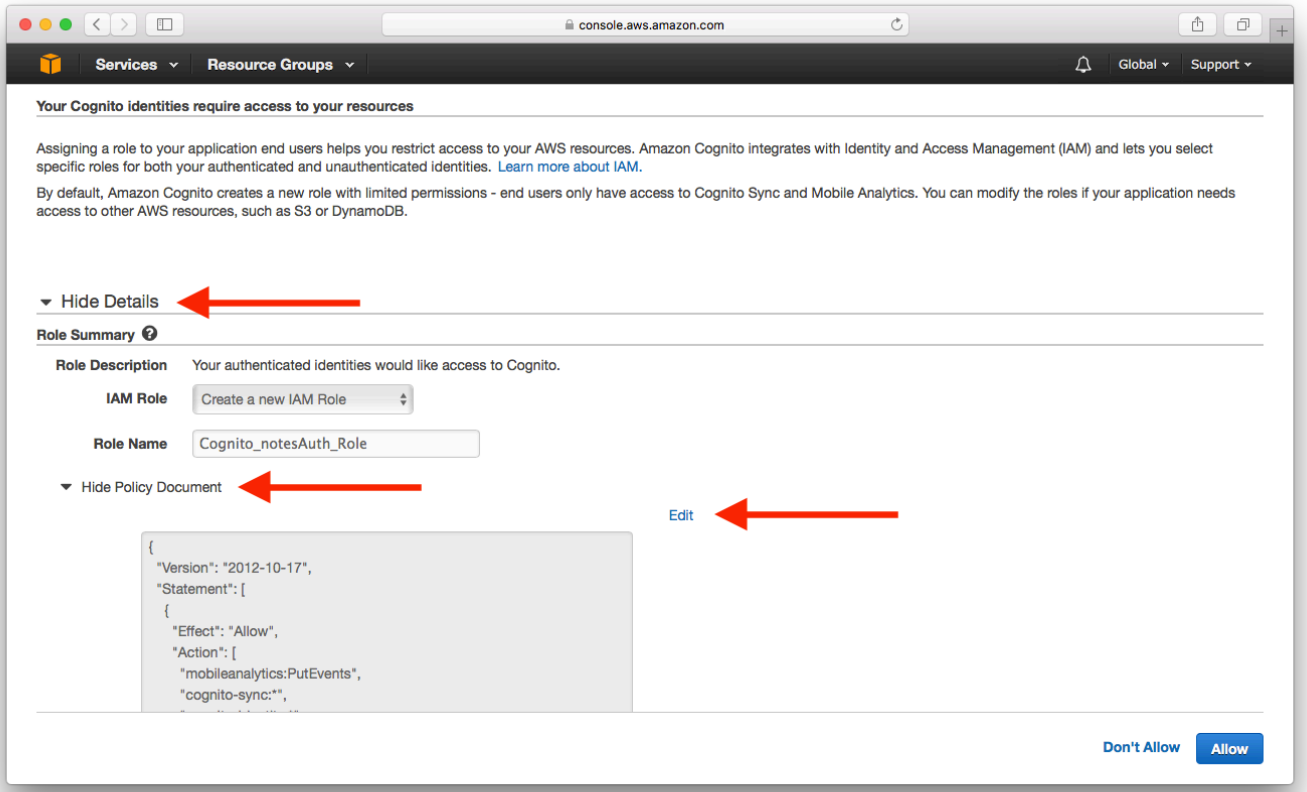
Select **Authentication providers**. Under **Cognito** tab, enter **User Pool ID** and **App Client ID** of the User Pool created in the Create a Cognito user pool (/chapters/create-a-cognito-user-pool.html) chapter. Select **Create Pool**.



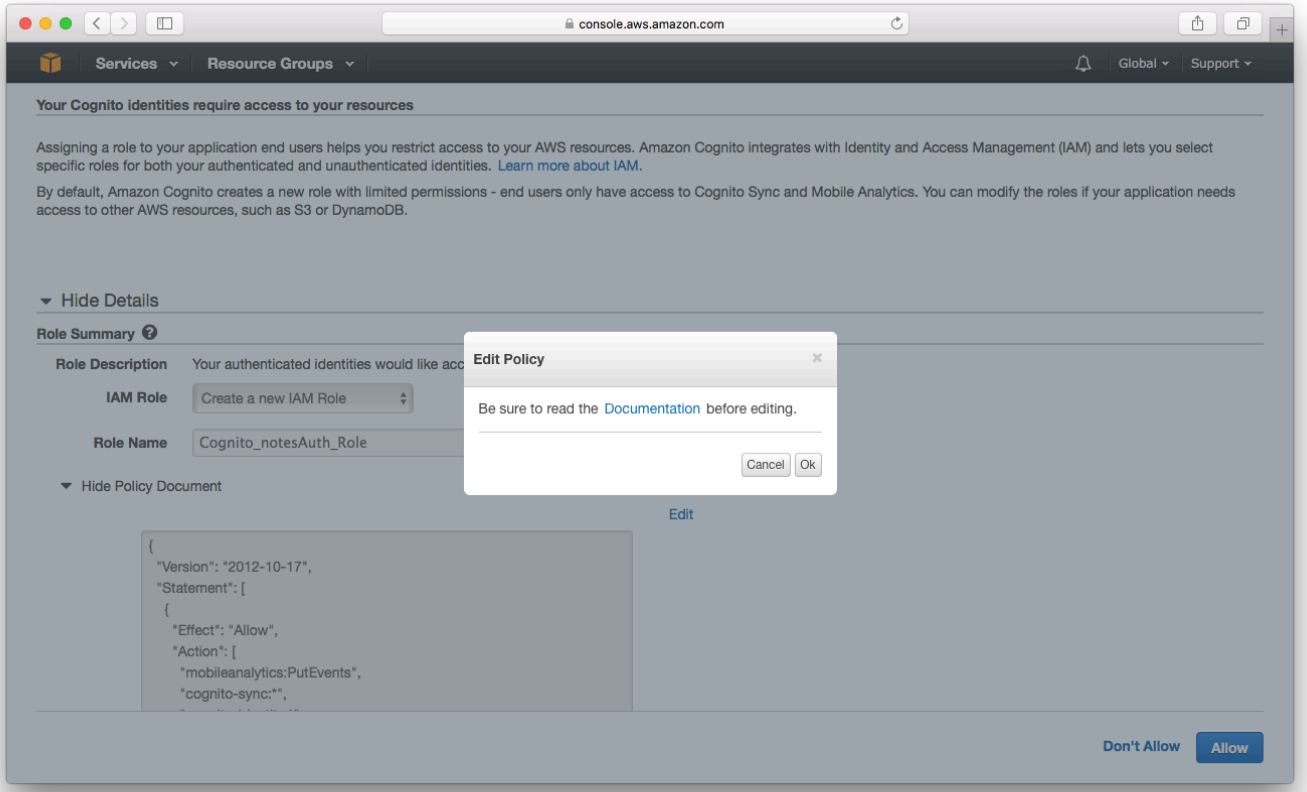
Now we need to specify what AWS resources are accessible for users with temporary credentials obtained from the Cognito Identity Pool.

Select **View Details**. Two **Role Summary** sections are expanded. The top section summarizes the permission policy for authenticated users, and the bottom section summarizes that for unauthenticated users.

Select **View Policy Document** in the top section. Then select **Edit**.



It will warn you to read the documentation. Select **Ok** to edit.



CHANGE Add the following policy into the editor. Replace

`YOUR_S3_UPLOADS_BUCKET_NAME` with the **bucket name** from the Create an S3 bucket for file uploads (/chapters/create-an-s3-bucket-for-file-uploads.html) chapter. And replace the `YOUR_API_GATEWAY_REGION` and `YOUR_API_GATEWAY_ID` with the ones that you get after you deployed your API in the last chapter.

In our case `YOUR_S3_UPLOADS_BUCKET_NAME` is `notes-app-uploads`, `YOUR_API_GATEWAY_ID` is `ly55wbovq4`, and `YOUR_API_GATEWAY_REGION` is `us-east-1`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "mobileanalytics:PutEvents",
        "cognito-sync:*",
        "cognito-identity:*"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:*"
      ],
      "Resource": [
```

```
"arn:aws:s3:::YOUR_S3_UPLOADS_BUCKET_NAME/private/${cogni
identity.amazonaws.com:sub}/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-
api:YOUR_API_GATEWAY_REGION:*:YOUR_API_GATEWAY_ID/*/*/*"

      ]
    }
  ]
}
```

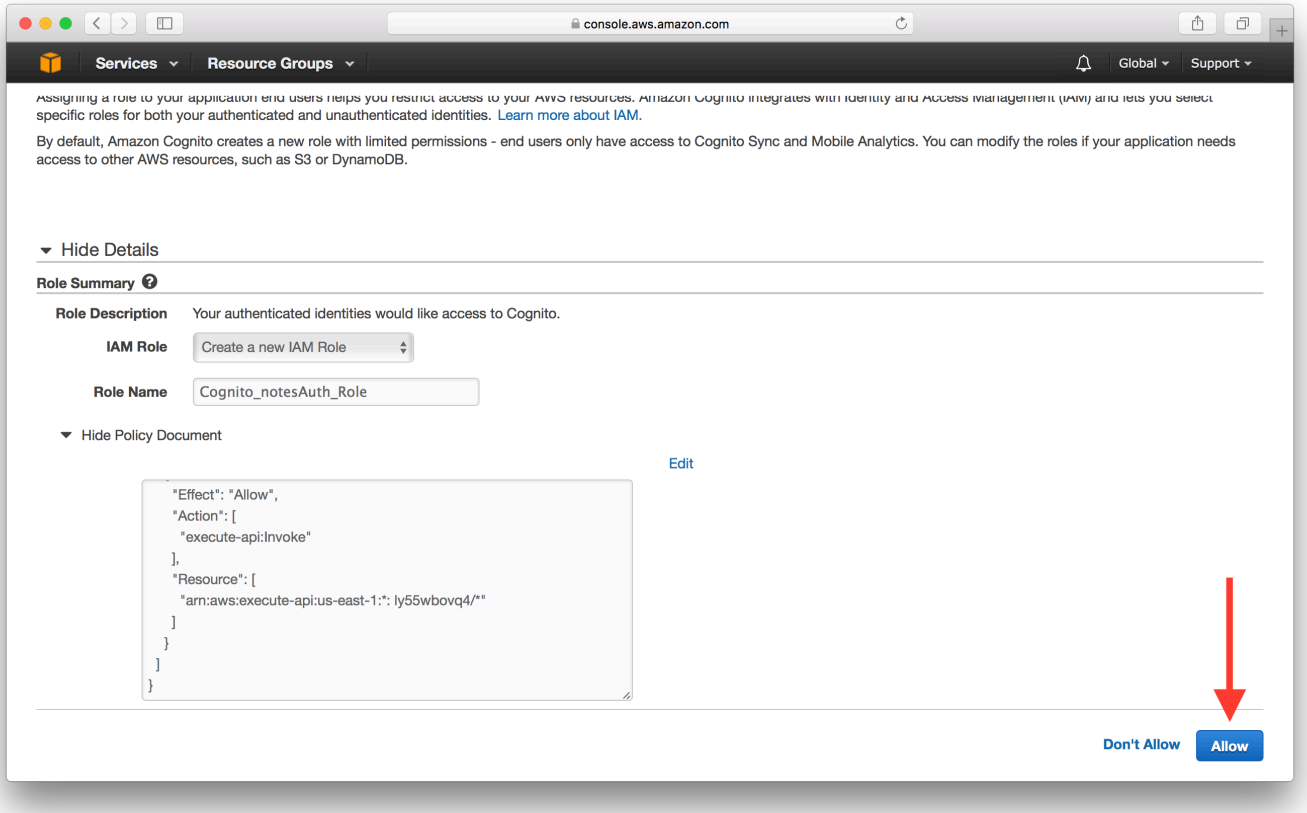
A quick note on the block that relates to the S3 Bucket. In the above policy we are granting our logged in users access to the path `private/${cognito-identity.amazonaws.com:sub}/`. Where `cognito-identity.amazonaws.com:sub` is the authenticated user's federated identity ID (their user id). So a user has access to only their folder within the bucket. This is how we are securing the uploads for each user.

So in summary we are telling AWS that an authenticated user has access to two resources.

1. Files in the S3 bucket that are inside a folder with their federated identity id as the name of the folder.
2. And, the APIs we deployed using API Gateway.

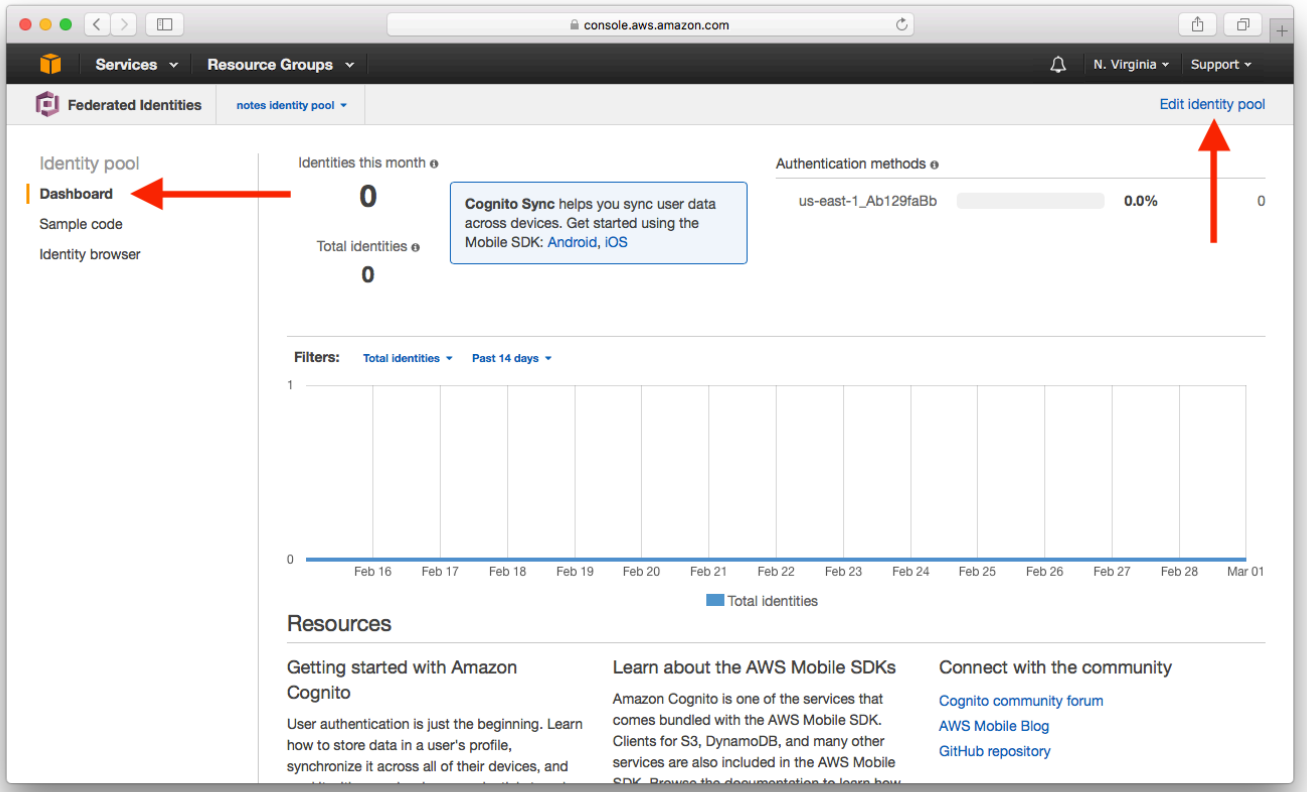
One other thing to note is that the federated identity id is a UUID that is assigned by our Identity Pool. This is the id (`event.requestContext.identity.cognitoIdentityId`) that we were using as our user id back when we were creating our APIs.

Select **Allow**.

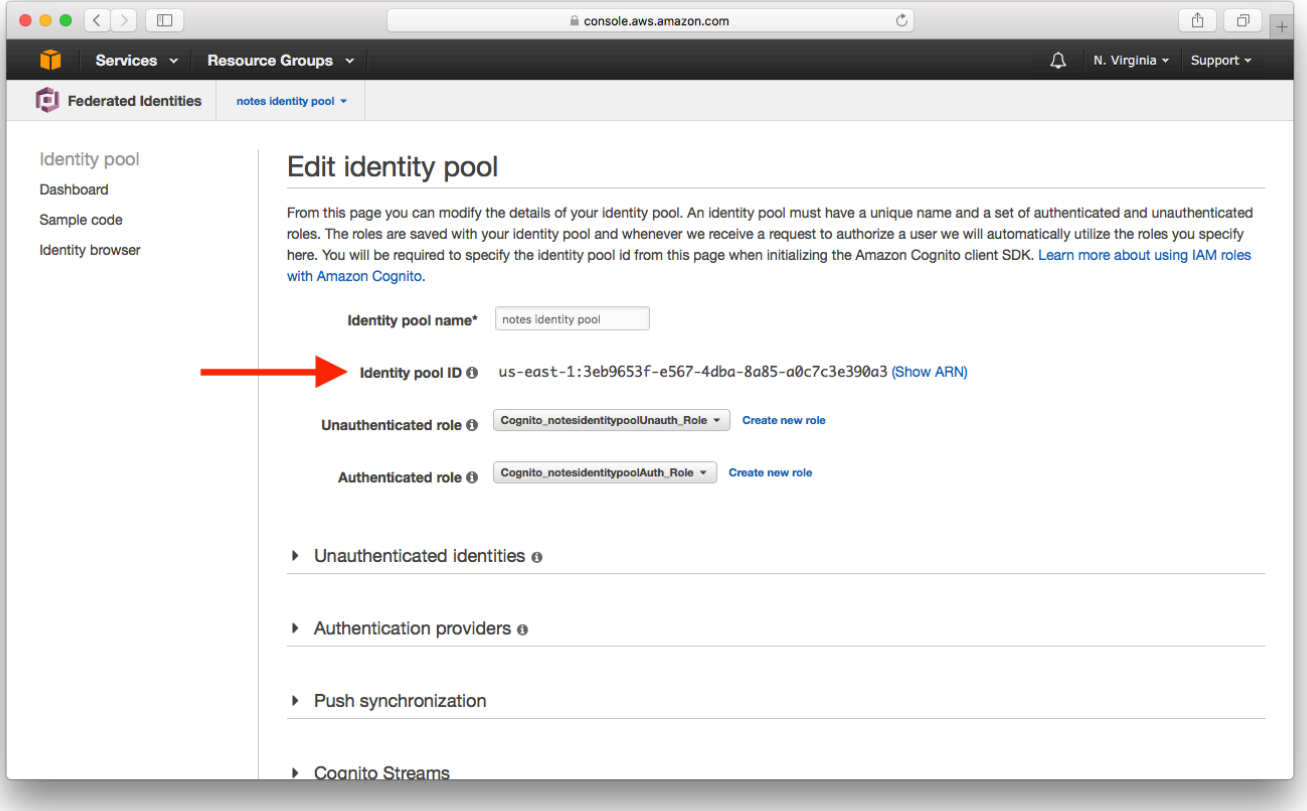


Our Cognito Identity Pool should now be created. Let's find out the Identity Pool ID.

Select **Dashboard** from the left panel, then select **Edit identity pool**.



Take a note of the **Identity pool ID** which will be required in the later chapters.



Now before we test our serverless API let's take a quick look at the Cognito User Pool and Cognito Identity Pool and make sure we've got a good idea of the two concepts and the differences between them.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-cognito-identity-pool/135)

Cognito User Pool vs Identity Pool

We often get questions about the differences between the Cognito User Pool and the Identity Pool, so it is worth covering in detail. The two can seem a bit similar in function and it is not entirely clear what they are for. Let's first start with the official definitions.

Here is what AWS defines the Cognito User Pool as:

> *Amazon Cognito User Pool makes it easy for developers to add sign-up and sign-in functionality to web and mobile applications. It serves as your own identity provider to maintain a user directory. It supports user registration and sign-in, as well as provisioning identity tokens for signed-in users.*

And the Cognito Federated Identities or Identity Pool is defined as:

> *Amazon Cognito Federated Identities enables developers to create unique identities for your users and authenticate them with federated identity providers. With a federated identity, you can obtain temporary, limited-privilege AWS credentials to securely access other AWS services such as Amazon DynamoDB, Amazon S3, and Amazon API Gateway.*

Unfortunately they are both a bit vague and confusingly similar. Here is a more practical description of what they are.

User Pool

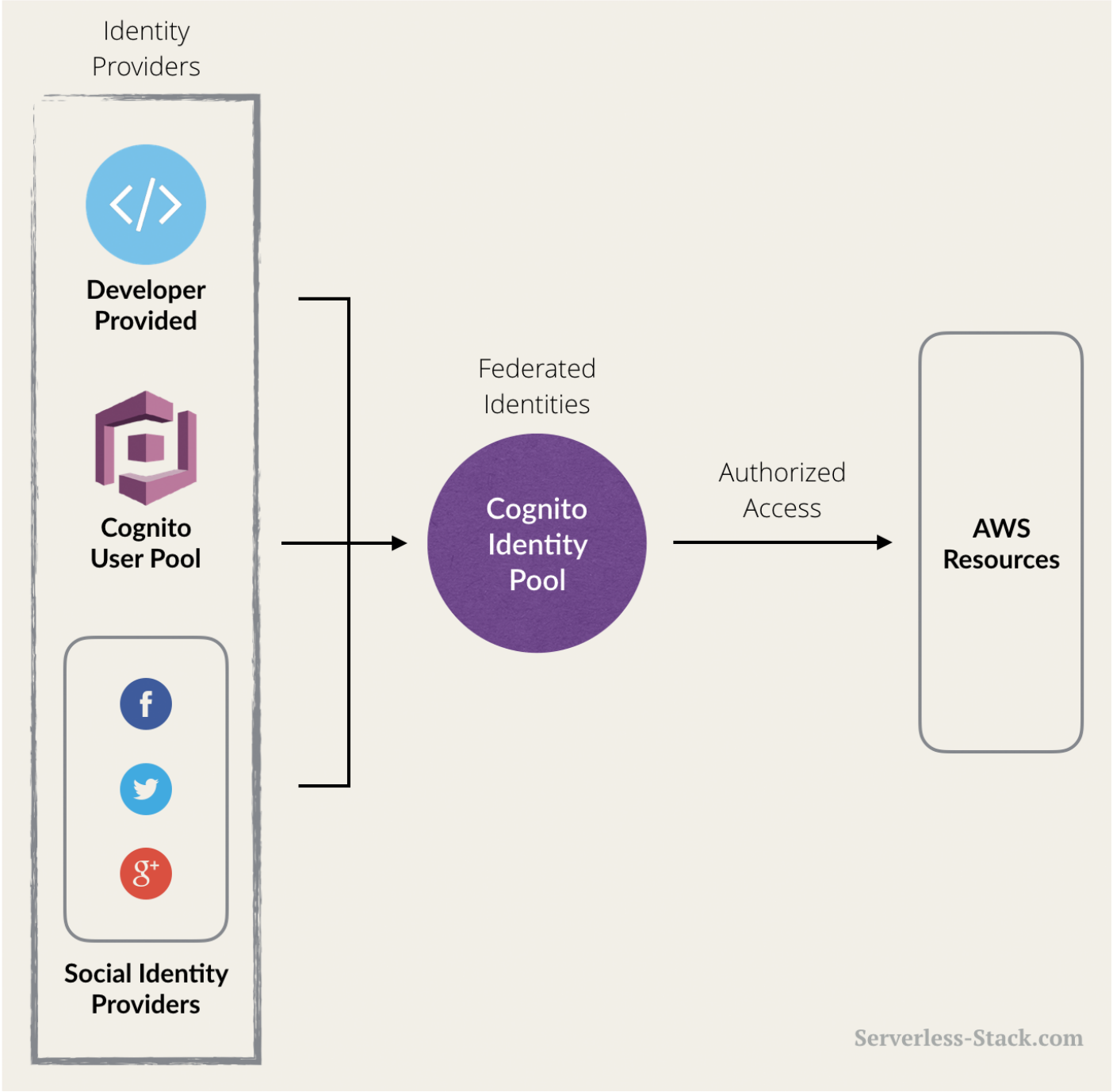
Say you were creating a new web or mobile app and you were thinking about how to handle user registration, authentication, and account recovery. This is where Cognito User Pools would come in. Cognito User Pool handles all of this and as a developer you just need to use the SDK to retrieve user related information.

Identity Pool

Cognito Identity Pool (or Cognito Federated Identities) on the other hand is a way to authorize your users to use the various AWS services. Say you wanted to allow a user to have access to your S3 bucket so that they could upload a file; you could specify that while creating an Identity Pool. And to create these levels of access, the Identity Pool has its own concept of an identity (or user). The source of these identities (or users) could be a Cognito User Pool or even Facebook or Google.

User Pool vs Identity Pool

To clarify this a bit more, let's put these two services in context of each other. Here is how they play together.



Notice how we could use the User Pool, social networks, or even our own custom authentication system as the identity provider for the Cognito Identity Pool. The Cognito Identity Pool simply takes all your identity providers and puts them together (federates them). And with all of this it can now give your users secure access to your AWS services, regardless of where they come from.

So in summary; the Cognito User Pool stores all your users which then plugs into your Cognito Identity Pool which can give your users access to

your AWS services.

Now that we have a good understanding of how our users will be handled, let's finish up our backend by testing our APIs.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/cognito-user-pool-vs-identity-pool/146)

Test the APIs

Now that we have our backend completely set up and secured, let's test the API we just deployed.

To be able to hit our API endpoints securely, we need to follow these steps.

1. Authenticate against our User Pool and acquire a user token.
2. With the user token get temporary IAM credentials from our Identity Pool.
3. Use the IAM credentials to sign our API request with Signature Version 4 (http://docs.aws.amazon.com/general/latest/gr/signature-version-4.html).

These steps can be a bit tricky to do by hand. So we created a simple tool called AWS API Gateway Test CLI (https://github.com/AnomalyInnovations/aws-api-gateway-cli-test).

You can run it using.

```
$ npx aws-api-gateway-cli-test
```

The `npx` command is just a convenient way of running a NPM module without installing it globally.

We need to pass in quite a bit of our info to complete the above steps.

- Use the username and password of the user created in the Create a

Cognito test user (/chapters/create-a-cognito-test-user.html) chapter.

- Replace **YOUR_COGNITO_USER_POOL_ID**, **YOUR_COGNITO_APP_CLIENT_ID**, and **YOUR_COGNITO_REGION** with the values from the Create a Cognito user pool (/chapters/create-a-cognito-user-pool.html) chapter. In our case the region is `us-east-1`.
- Replace **YOUR_IDENTITY_POOL_ID** with the one from the Create a Cognito identity pool (/chapters/create-a-cognito-identity-pool.html) chapter.
- Use the **YOUR_API_GATEWAY_URL** and **YOUR_API_GATEWAY_REGION** with the ones from the Deploy the APIs (/chapters/deploy-the-apis.html) chapter. In our case the URL is `https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod` and the region is `us-east-1`.

And run the following.

```
$ npx aws-api-gateway-cli-test \
--username='admin@example.com' \
--password='Passw0rd!' \
--user-pool-id='YOUR_COGNITO_USER_POOL_ID' \
--app-client-id='YOUR_COGNITO_APP_CLIENT_ID' \
--cognito-region='YOUR_COGNITO_REGION' \
--identity-pool-id='YOUR_IDENTITY_POOL_ID' \
--invoke-url='YOUR_API_GATEWAY_URL' \
--api-gateway-region='YOUR_API_GATEWAY_REGION' \
--path-template='/notes' \
--method='POST' \
--body='{"content":"hello
world","attachment":"hello.jpg"}'
```

While this might look intimidating, just keep in mind that behind the scenes all we are doing is generating some security headers before making a basic HTTP request. You'll see more of this process when we connect our React.js app to our API backend.

If you are on Windows, use the command below. The space between each option is very important.

```
$ npx aws-api-gateway-cli-test --username
admin@example.com --password Passw0rd! --user-pool-id
YOUR_COGNITO_USER_POOL_ID --app-client-id
YOUR_COGNITO_APP_CLIENT_ID --cognito-region
YOUR_COGNITO_REGION --identity-pool-id
YOUR_IDENTITY_POOL_ID --invoke-url YOUR_API_GATEWAY_URL
--api-gateway-region YOUR_API_GATEWAY_REGION --path-
template /notes --method POST --body "
{\"content\":\"hello
world\",\"attachment\":\"hello.jpg\"}"
```

If the command is successful, the response will look similar to this.

```
Authenticating with User Pool
Getting temporary credentials
Making API request
{ status: 200,
  statusText: 'OK',
  data:
   { userId: 'us-east-1:9bdc031d-ee9e-4ffa-9a2d-
123456789',
     noteId: '8f7da030-650b-11e7-a661-123456789',
     content: 'hello world',
```

```
    attachment: 'hello.jpg',
    createdAt: 1499648598452 } }
```

And that's it for the backend! Next we are going to move on to creating the frontend of our app.

Common Issues

- Response `{status: 403}`

  This is the most common issue we come across and it is a bit cryptic and can be hard to debug. Here are a few things to check before you start debugging:

  - Ensure the `--path-template` option in the `apig-test` command is pointing to `/notes` and not `notes`. The format matters for securely signing our request.
  - There are no trailing slashes for `YOUR_API_GATEWAY_URL`. In our case, the URL is `https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod`. Notice that it does not end with a `/`.
  - If you're on Windows and are using Git Bash, try adding a trailing slash to `YOUR_API_GATEWAY_URL` while removing the leading slash from `--path-template`. In our case, it would result in `--invoke-url https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/ --path-template notes`. You can follow the discussion on this here (https://github.com/AnomalyInnovations/serverless-stack-

com/issues/112#issuecomment-345996566).

There is a good chance that this error is happening even before our Lambda functions are invoked. So we can start by making sure our IAM Roles are configured properly for our Identity Pool. Follow the steps as detailed in our Debugging Serverless API Issues (/chapters/debugging-serverless-api-issues.html#missing-iam-policy) chapter to ensure that your IAM Roles have the right set of permissions.

Next, you can enable API Gateway logs (/chapters/api-gateway-and-lambda-logs.html#enable-api-gateway-cloudwatch-logs) and follow these instructions (/chapters/api-gateway-and-lambda-logs.html#viewing-api-gateway-cloudwatch-logs) to read the requests that are being logged. This should give you a better idea of what is going on.

Finally, make sure to look at the comment thread below. We've helped quite a few people with similar issues and it's very likely that somebody has run into a similar issue as you.

- Response `{status: false}`

  If instead your command fails with the `{status: false}` response; we can do a few things to debug this. This response is generated by our Lambda functions when there is an error. Add a `console.log` like so in your handler function.

```
catch(e) {
  console.log(e);
  callback(null, failure({status: false}));
}
```

And deploy it using `serverless deploy function -f create`. But we can't see this output when we make an HTTP request to it, since the console logs are not sent in our HTTP responses. We need to check the logs to see this. We have a detailed chapter (/chapters/api-gateway-and-lambda-logs.html#viewing-lambda-cloudwatch-logs) on working with API Gateway and Lambda logs and you can read about how to check your debug messages here (/chapters/api-gateway-and-lambda-logs.html#viewing-lambda-cloudwatch-logs).

A common source of errors here is an improperly indented `serverless.yml`. Make sure to double-check the indenting in your `serverless.yml` by comparing it to the one from this chapter (https://github.com/AnomalyInnovations/serverless-stack-demo-api/blob/master/serverless.yml).

- `'User: arn:aws:... is not authorized to perform: dynamodb:PutItem on resource: arn:aws:dynamodb:...'`

  This error is basically saying that our Lambda function does not have the right permissions to make a DynamoDB request. Recall that, the IAM role that allows your Lambda function to make requests to DynamoDB are set in the `serverless.yml`. And a common source of this error is when the `iamRoleStatements:` are improperly indented. Make sure to compare it to the one in the repo (https://github.com/AnomalyInnovations/serverless-stack-demo-api).

For help and discussion

Comments on this chapter (https://discourse.serverless-

```
stack.com/t/comments-for-test-the-apis/122)
```

Create a New React.js App

Let's get started with our frontend. We are going to create a single page app using React.js (https://facebook.github.io/react/). We'll use the Create React App (https://github.com/facebookincubator/create-react-app) project to set everything up. It is officially supported by the React team and conveniently packages all the dependencies for a React.js project.

✦ CHANGE Move out of the directory that we were working in for the backend.

```
$ cd ../
```

Create a New React App

✦ CHANGE Run the following command to create the client for our notes app.

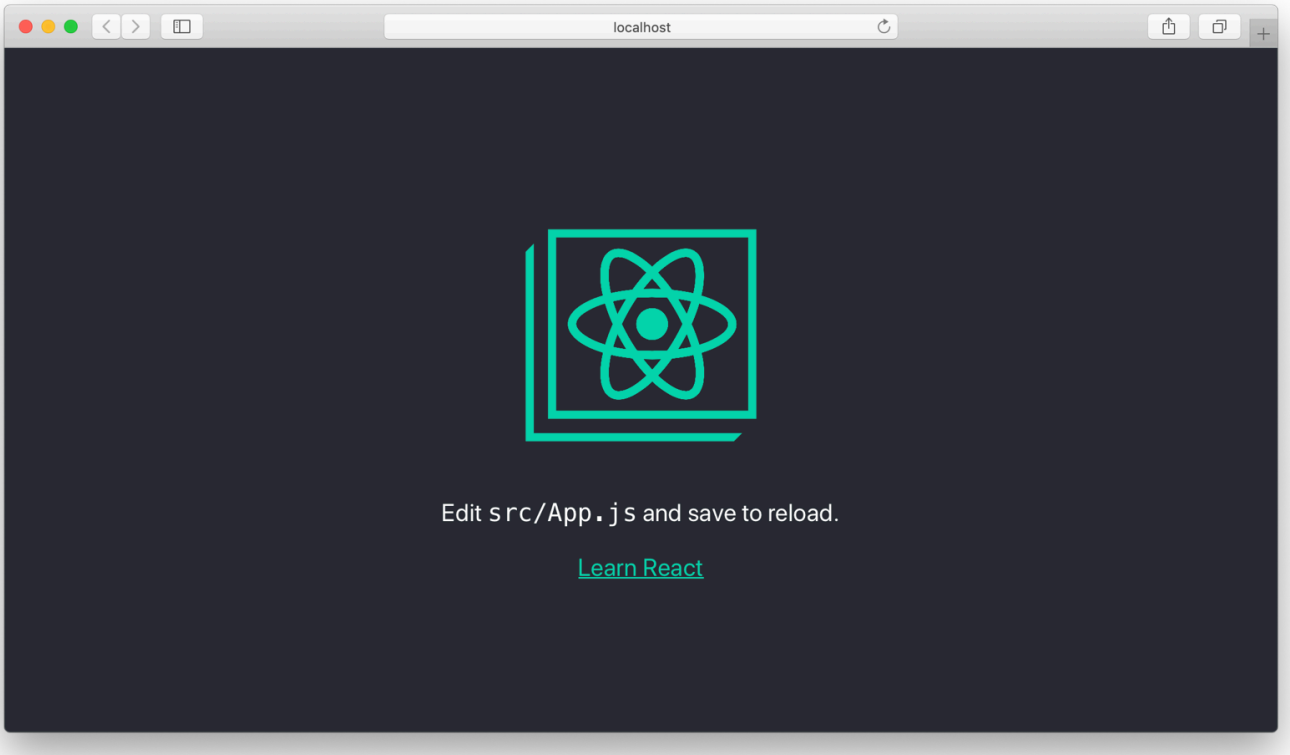
```
$ npx create-react-app notes-app-client --use-npm
```

This should take a second to run, and it will create your new project and your new working directory.

✦ CHANGE Now let's go into our working directory and run our project.

```
$ cd notes-app-client
$ npm start
```

This should fire up the newly created app in your browser.



Change the Title

✦ CHANGE Let's quickly change the title of our note taking app. Open up `public/index.html` and edit the `title` tag to the following:

```
<title>Scratch - A simple note taking app</title>
```

Create React App comes pre-loaded with a pretty convenient yet minimal development environment. It includes live reloading, a testing framework, ES6 support, and much more (https://github.com/facebookincubator/create-react-app#why-use-this).

Now we are ready to build our frontend. But just like we did with the backend, let's first create a GitHub repo to store our code.

For help and discussion

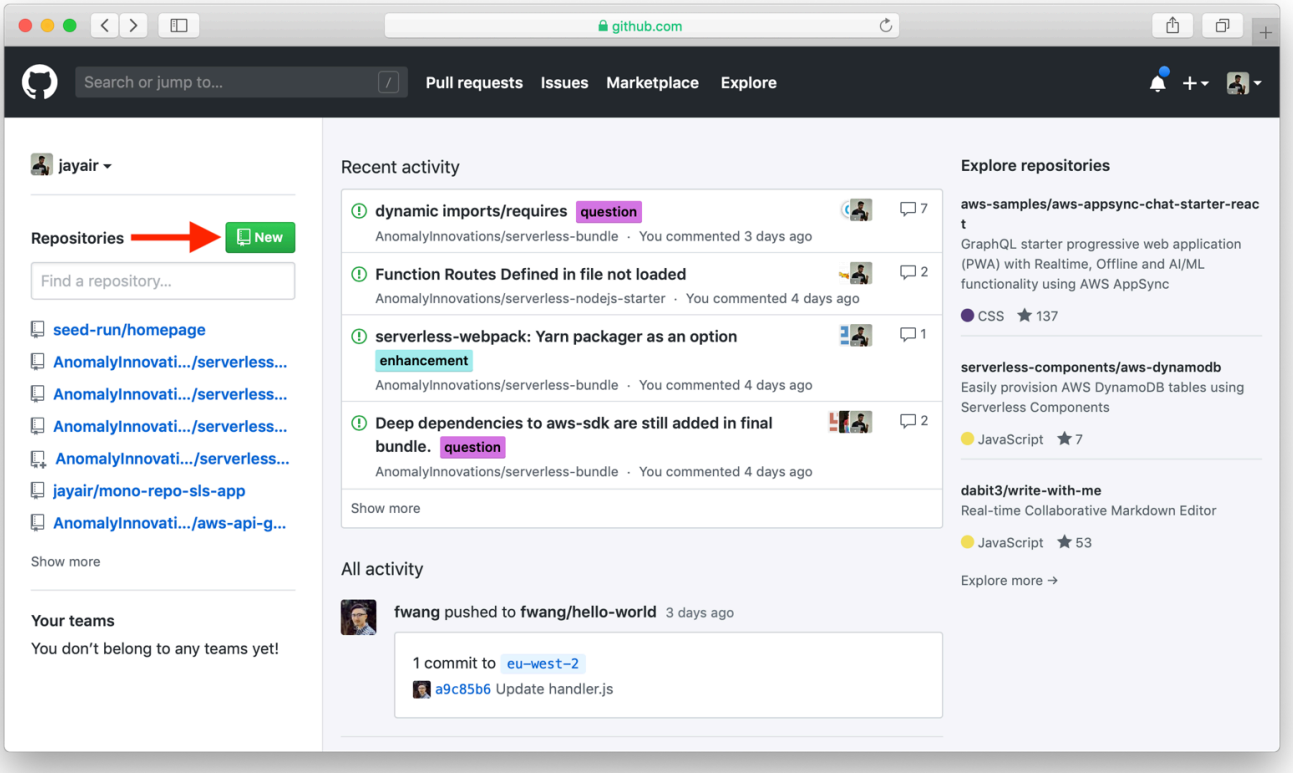
Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-new-react-js-app/68)

Initialize the Frontend Repo

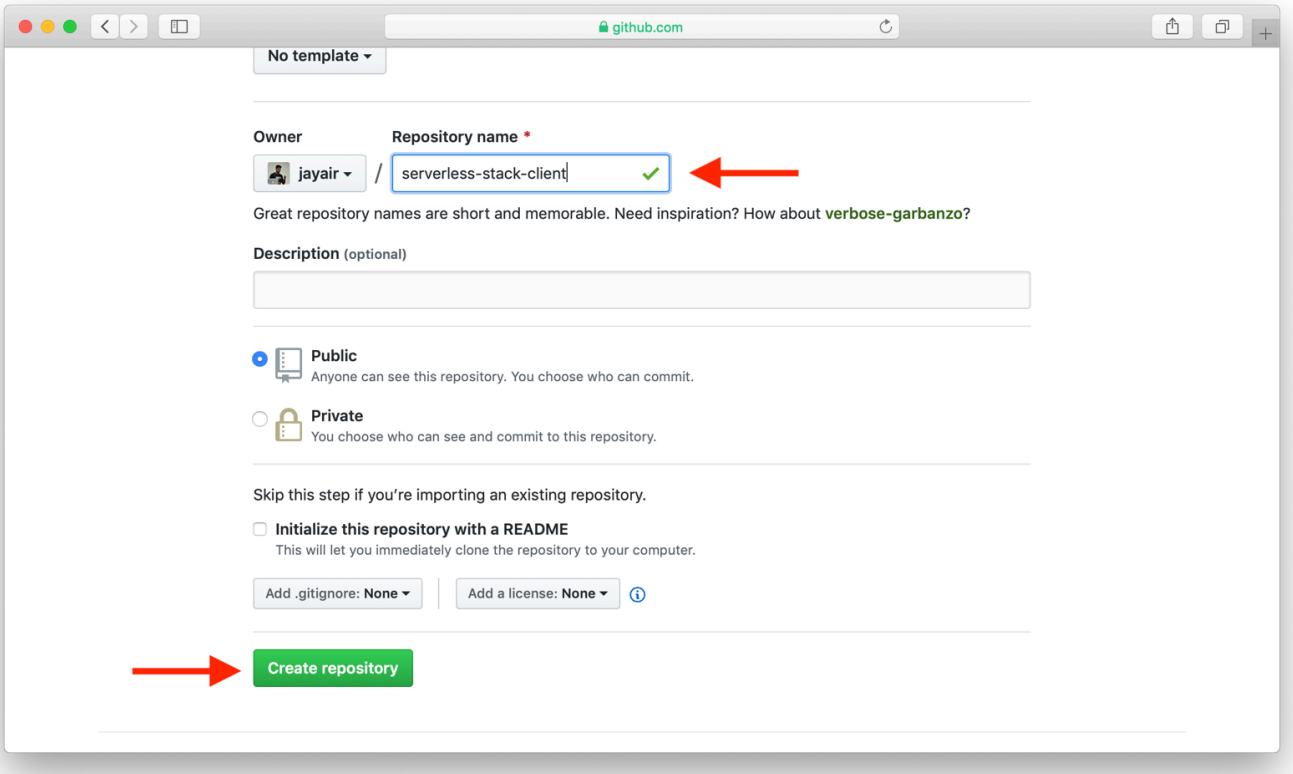
Just as we did in the backend portion, we'll start by adding our project to a GitHub repo. We need this to store our code and we'll use this later to automate our deployments.

Create a New GitHub Repo

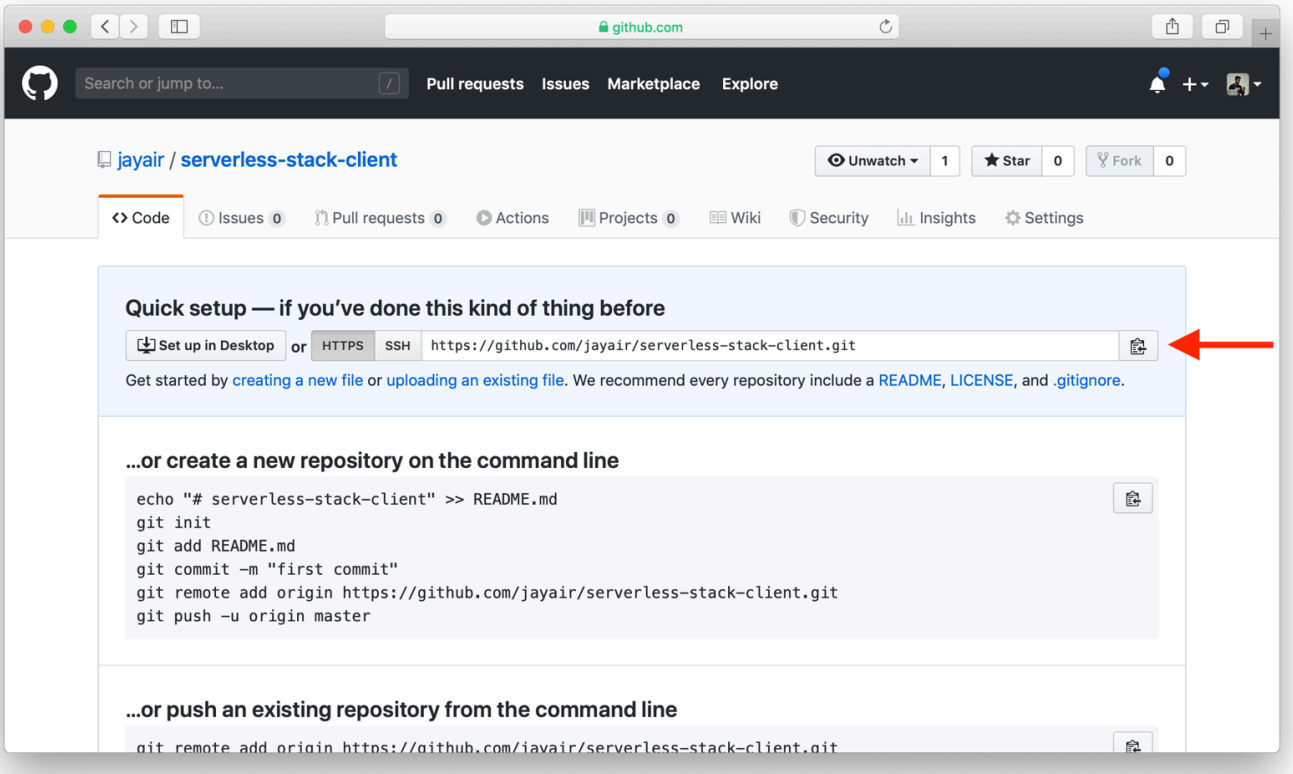
Let's head over to GitHub (https://github.com). Make sure you are signed in and hit **New repository**.



Give your repository a name, in our case we are calling it `serverless-stack-client`. And hit **Create repository**.



Once your repository is created, copy the repository URL. We'll need this soon.



In our case the URL is:

```
https://github.com/jayair/serverless-stack-client.git
```

Initialize Your New Repo

CHANGE Now head back to your project and use the following command to initialize your new repo.

```
$ git init
```

CHANGE Add the existing files.

```
$ git add .
```

CHANGE Create your first commit.

```
$ git commit -m "First commit"
```

CHANGE Link it to the repo you created on GitHub.

```
$ git remote add origin REPO_URL
```

Here `REPO_URL` is the URL we copied from GitHub in the steps above. You can verify that it has been set correctly by doing the following.

```
$ git remote -v
```

CHANGE Finally, let's push our first commit to GitHub using:

```
$ git push -u origin master
```

Now we are ready to build our frontend! We are going start by creating our app icon and updating the favicons.

CHECKPOINT

For reference, here is the code we have so far.

Frontend Source :initialize-the-frontend-repo (https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/initialize-the-frontend-repo)

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/initialize-the-frontend-repo/181)

Add App Favicons

Create React App generates a simple favicon for our app and places it in `public/favicon.ico`. However, getting the favicon to work on all browsers and mobile platforms requires a little more work. There are quite a few different requirements and dimensions. And this gives us a good opportunity to learn how to include files in the `public/` directory of our app.

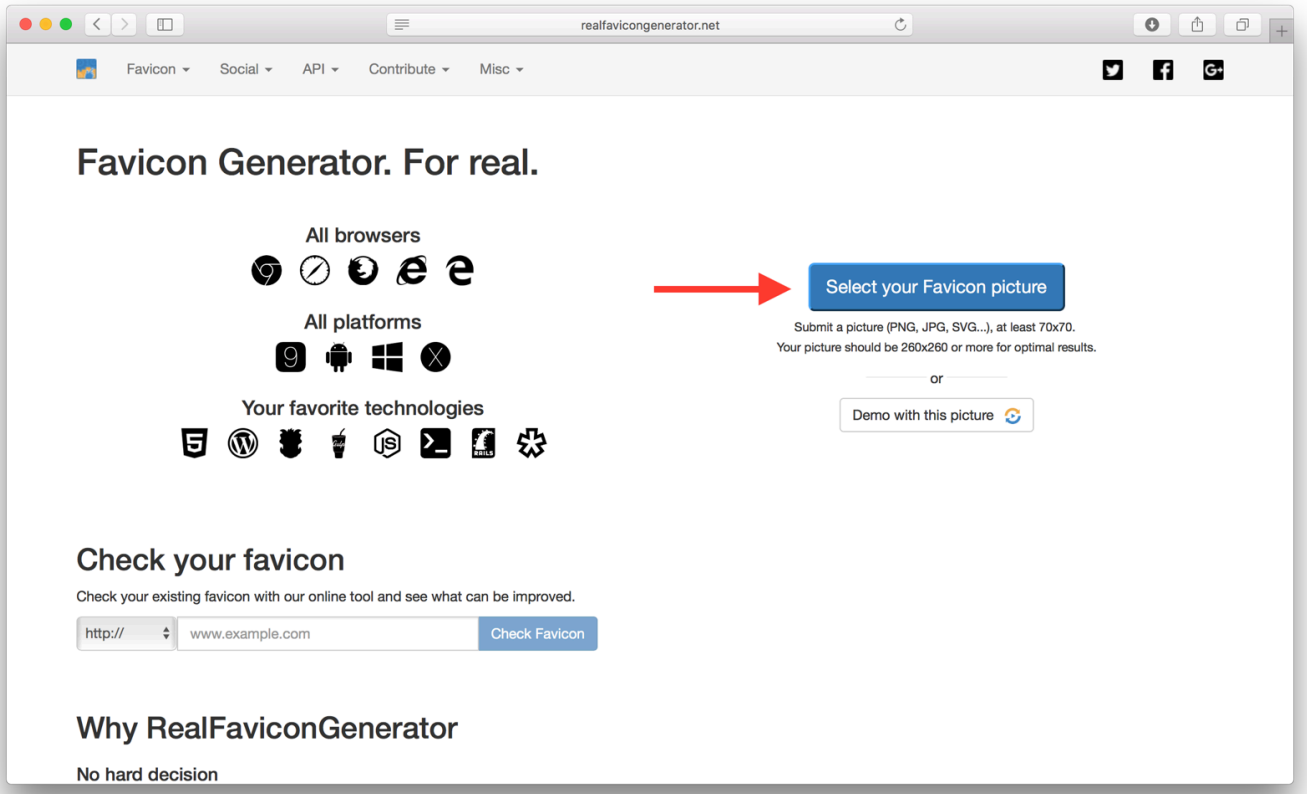
For our example, we are going to start with a simple image and generate the various versions from it.

Right-click to download the following image.

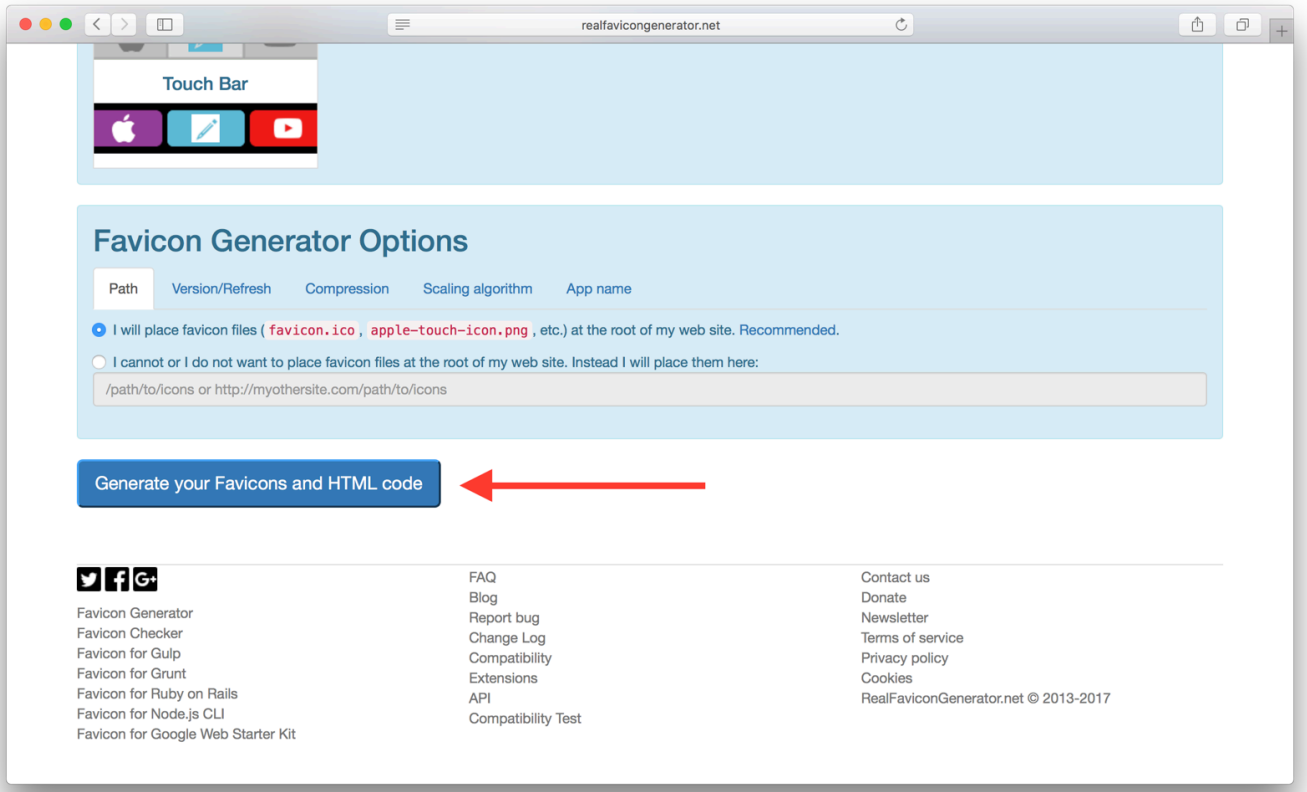


To ensure that our icon works for most of our targeted platforms we'll use a service called the Favicon Generator (http://realfavicongenerator.net).

Click **Select your Favicon picture** to upload our icon.

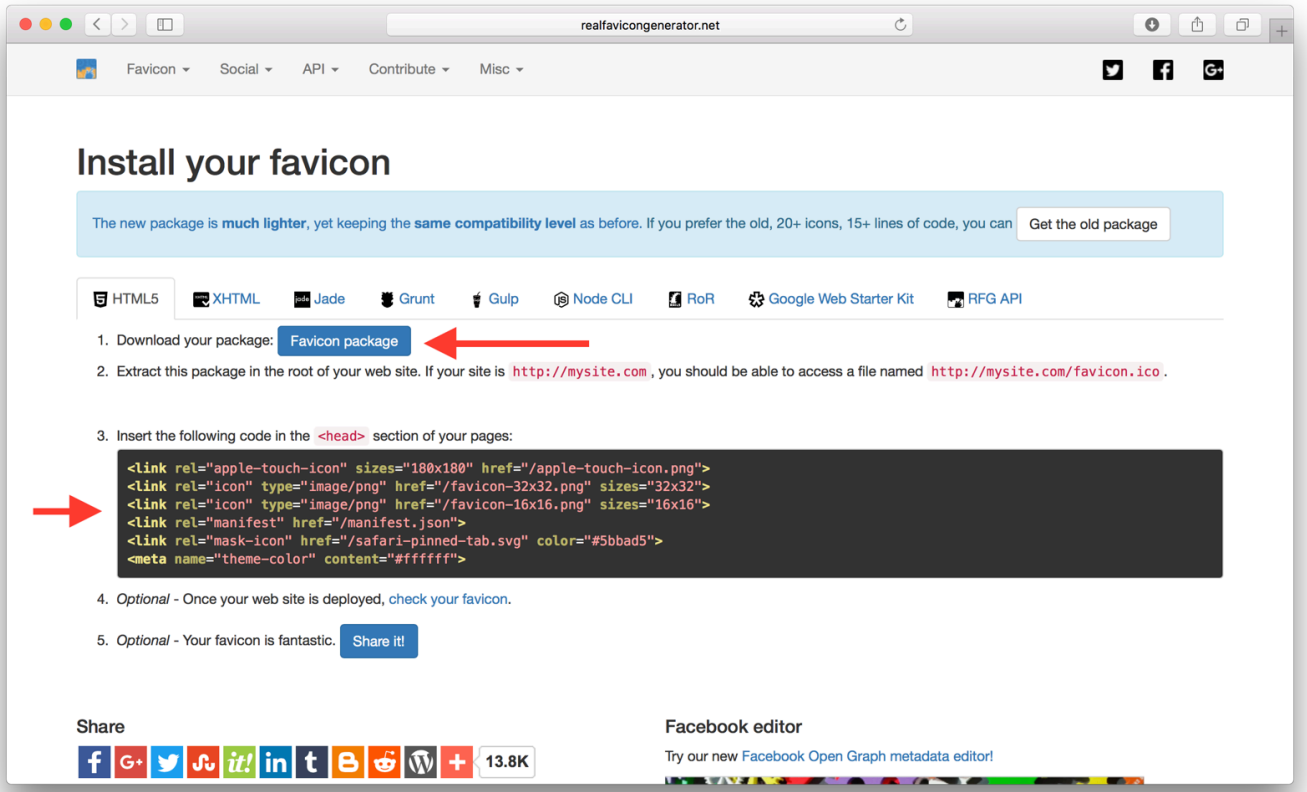


Once you upload your icon, it'll show you a preview of your icon on various platforms. Scroll down the page and hit the **Generate your Favicons and HTML code** button.



This should generate your favicon package and the accompanying code.

✦ CHANGE Click **Favicon package** to download the generated favicons. And copy all the files over to your `public/` directory.



✦ CHANGE Remove the `public/logo192.png` and `public/logo512.png` files.

```
$ rm public/logo192.png
$ rm public/logo512.png
```

✦ CHANGE Then replace the contents of `public/manifest.json` with the following:

```
{
  "short_name": "Scratch",
  "name": "Scratch Note Taking App",
  "icons": [
    {
      "src": "android-chrome-192x192.png",
      "sizes": "192x192",
```

```
      "type": "image/png"
    },
    {
      "src": "android-chrome-256x256.png",
      "sizes": "256x256",
      "type": "image/png"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#ffffff",
  "background_color": "#ffffff"
}
```

To include a file from the `public/` directory in your HTML, Create React App needs the `%PUBLIC_URL%` prefix.

✦ CHANGE Add this to your `public/index.html`.

```
<link rel="apple-touch-icon" sizes="180x180"
href="%PUBLIC_URL%/apple-touch-icon.png">
<link rel="icon" type="image/png"
href="%PUBLIC_URL%/favicon-32x32.png" sizes="32x32">
<link rel="icon" type="image/png"
href="%PUBLIC_URL%/favicon-16x16.png" sizes="16x16">
<link rel="mask-icon" href="%PUBLIC_URL%/safari-pinned-
tab.svg" color="#5bbad5">
<meta name="description" content="A simple note taking
app" />
<meta name="theme-color" content="#ffffff">
```

CHANGE And **remove** the following lines that reference the original favicon and theme color.

```
<meta name="theme-color" content="#000000">
<link rel="shortcut icon"
href="%PUBLIC_URL%/favicon.ico">
<link rel="apple-touch-icon" href="logo192.png" />
<meta
  name="description"
  content="Web site created using create-react-app"
/>
```

Finally head over to your browser and try the `/favicon-32x32.png` path to ensure that the files were added correctly.

Next we are going to look into setting up custom fonts in our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/add-app-favicons/155)

Set up Custom Fonts

Custom Fonts are now an almost standard part of modern web applications. We'll be setting it up for our note taking app using Google Fonts (https://fonts.google.com).

This also gives us a chance to explore the structure of our newly created React.js app.

Include Google Fonts

For our project we'll be using the combination of a Serif (PT Serif (https://fonts.google.com/specimen/PT+Serif)) and Sans-Serif (Open Sans (https://fonts.google.com/specimen/Open+Sans)) typeface. They will be served out through Google Fonts and can be used directly without having to host them on our end.

Let's first include them in the HTML. Our React.js app is using a single HTML file.

✦ CHANGE Go ahead and edit `public/index.html` and add the following line in the `<head>` section of the HTML to include the two typefaces.

```
<link rel="stylesheet" type="text/css"
href="https://fonts.googleapis.com/css?
family=PT+Serif|Open+Sans:300,400,600,700,800">
```

Here we are referencing all the 5 different weights (300, 400, 600, 700, and 800) of the Open Sans typeface.

Add the Fonts to the Styles

Now we are ready to add our newly added fonts to our stylesheets. Create React App helps separate the styles for our individual components and has a master stylesheet for the project located in `src/index.css`.

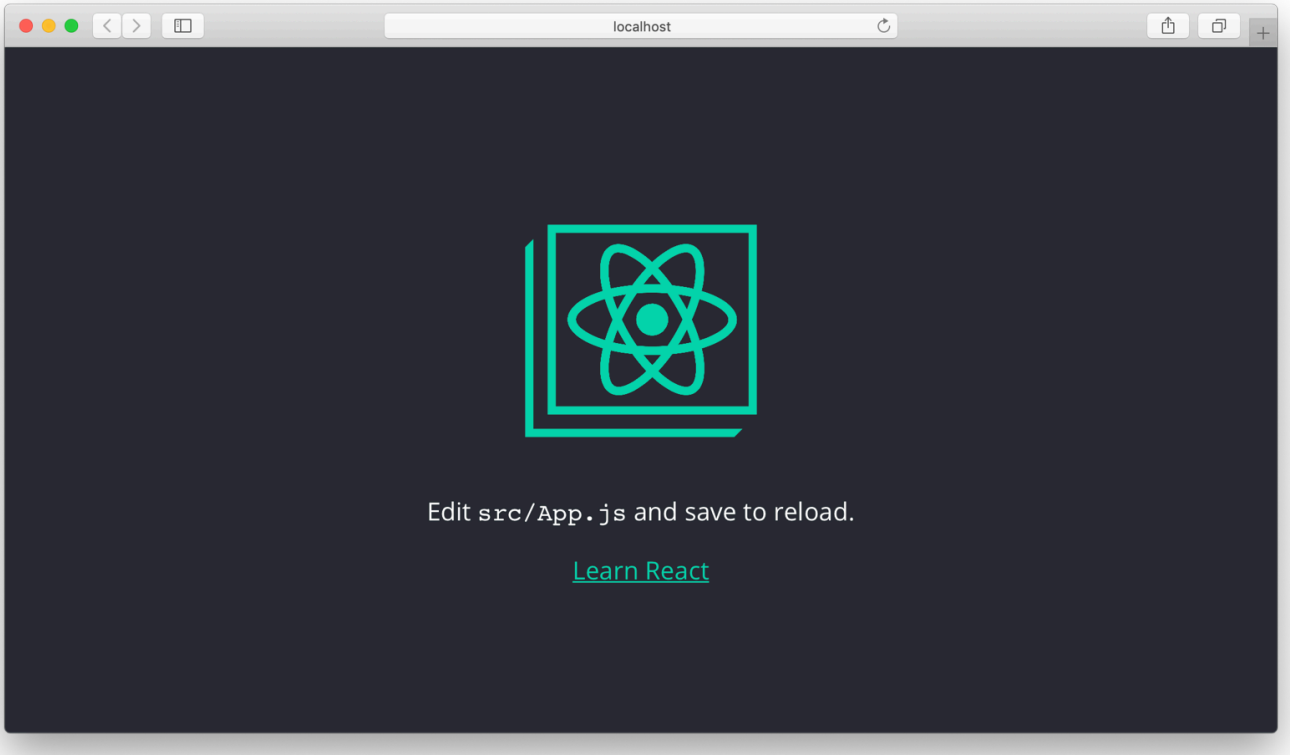
✦ CHANGE Let's change the current font in `src/index.css` for the `body` tag to the following.

```
body {
  margin: 0;
  padding: 0;
  font-family: "Open Sans", sans-serif;
  font-size: 16px;
  color: #333;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}
```

✦ CHANGE And let's change the fonts for the header tags to our new Serif font by adding this block to the css file.

```
h1, h2, h3, h4, h5, h6 {
  font-family: "PT Serif", serif;
}
```

Now if you just flip over to your browser with our new app, you should see the new fonts update automatically; thanks to the live reloading.



We'll stay on the theme of adding styles and set up our project with Bootstrap to ensure that we have a consistent UI Kit to work with while building our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/set-up-custom-fonts/81)

Set up Bootstrap

A big part of writing web applications is having a UI Kit to help create the interface of the application. We are going to use Bootstrap (http://getbootstrap.com) for our note taking app. While Bootstrap can be used directly with React; the preferred way is to use it with the React-Bootstrap (https://react-bootstrap.github.io) package. This makes our markup a lot simpler to implement and understand.

Installing React Bootstrap

✦ CHANGE Run the following command in your working directory.

```
$ npm install react-bootstrap@0.32.4 --save
```

This installs the NPM package and adds the dependency to your `package.json`.

Add Bootstrap Styles

✦ CHANGE React Bootstrap uses the standard Bootstrap v3 styles; so just add the following styles to your `public/index.html`.

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css
```

We'll also tweak the styles of the form fields so that the mobile browser does not zoom in on them on focus. We just need them to have a minimum font size of `16px` to prevent the zoom.

✦ CHANGE To do that, let's add the following to our `src/index.css`.

```
select.form-control,
textarea.form-control,
input.form-control {
  font-size: 16px;
}
input[type=file] {
  width: 100%;
}
```

We are also setting the width of the input type file to prevent the page on mobile from overflowing and adding a scrollbar.

Now if you head over to your browser, you might notice that the styles have shifted a bit. This is because Bootstrap includes Normalize.css (http://necolas.github.io/normalize.css/) to have a more consistent styles across browsers.

Next, we are going to create a few routes for our application and set up the React Router.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/set-up-bootstrap/118)

Handle Routes with React Router

Create React App sets a lot of things up by default but it does not come with a built-in way to handle routes. And since we are building a single page app, we are going to use React Router (https://reacttraining.com/react-router/) to handle them for us.

Let's start by installing React Router. We are going to be using the React Router v4, the newest version of React Router. React Router v4 can be used on the web and in native. So let's install the one for the web.

Installing React Router v4

✦ CHANGE Run the following command in your working directory.

```
$ npm install react-router-dom@5.0.1 --save
```

This installs the NPM package and adds the dependency to your `package.json`.

Setting up React Router

Even though we don't have any routes set up in our app, we can get the basic structure up and running. Our app currently runs from the `App` component in `src/App.js`. We are going to be using this component as

the container for our entire app. To do that we'll encapsulate our `App` component within a `Router`.

✦ CHANGE Replace the following code in `src/index.js`:

```
ReactDOM.render(<App />,
document.getElementById('root'));
```

✦ CHANGE With this:

```
ReactDOM.render(
  <Router>
    <App />
  </Router>,
  document.getElementById('root')
);
```

✦ CHANGE And import this in the header of `src/index.js`.

```
import { BrowserRouter as Router } from 'react-router-dom';
```

We've made two small changes here.

1. Use `BrowserRouter` as our router. This uses the browser's History (https://developer.mozilla.org/en-US/docs/Web/API/History) API to create real URLs.
2. Use the `Router` to render our `App` component. This will allow us to create the routes we need inside our `App` component.

Now if you head over to your browser, your app should load just like

before. The only difference being that we are using React Router to serve out our pages.

Next we are going to look into how to organize the different pages of our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/handle-routes-with-react-router/116)

Create Containers

Currently, our app has a single component that renders our content. For creating our note taking app, we need to create a few different pages to load/edit/create notes. Before we can do that we will put the outer chrome of our app inside a component and render all the top level components inside them. These top level components that represent the various pages will be called containers.

Add a Navbar

Let's start by creating the outer chrome of our application by first adding a navigation bar to it. We are going to use the Navbar (https://react-bootstrap.github.io/components/navbar/) React-Bootstrap component.

✦ CHANGE To start, you can go remove the `src/logo.svg` that is placed there by Create React App.

```
$ rm src/logo.svg
```

✦ CHANGE And go ahead and remove the code inside `src/App.js` and replace it with the following.

```
import React from "react";
import { Link } from "react-router-dom";
import { Navbar } from "react-bootstrap";
import "./App.css";
```

```
function App(props) {
  return (
    <div className="App container">
      <Navbar fluid collapseOnSelect>
        <Navbar.Header>
          <Navbar.Brand>
            <Link to="/">Scratch</Link>
          </Navbar.Brand>
          <Navbar.Toggle />
        </Navbar.Header>
      </Navbar>
    </div>
  );
}

export default App;
```

We are doing a few things here:

1. Creating a fixed width container using Bootstrap in `div.container`.
2. Adding a Navbar inside the container that fits to its container's width using the attribute `fluid`.
3. Using `Link` component from the React-Router to handle the link to our app's homepage (without forcing the page to refresh).

Note that React Bootstrap might give you a couple of warnings in your browser console.

```
Warning: componentWillMount has been renamed, and is
not recommended for use.
```

We can ignore them for now while we wait for React Bootstrap to fix them. Also, we'll be upgrading the guide to React Bootstrap v1 once it is out of beta.

Let's also add a couple of line of styles to space things out a bit more.

✦ CHANGE Remove all the code inside `src/App.css` and replace it with the following:

```
.App {
  margin-top: 15px;
}

.App .navbar-brand {
  font-weight: bold;
}
```

Add the Home container

Now that we have the outer chrome of our application ready, let's add the container for the homepage of our app. It'll respond to the `/` route.

✦ CHANGE Create a `src/containers/` directory by running the following in your working directory.

```
$ mkdir src/containers/
```

We'll be storing all of our top level components here. These are components that will respond to our routes and make requests to our API. We will be calling them *containers* through the rest of this tutorial.

✦ CHANGE Create a new container and add the following to

`src/containers/Home.js`.

```
import React from "react";
import "./Home.css";

export default function Home() {
  return (
    <div className="Home">
      <div className="lander">
        <h1>Scratch</h1>
        <p>A simple note taking app</p>
      </div>
    </div>
  );
}
```

This simply renders our homepage given that the user is not currently signed in.

Now let's add a few lines to style this.

✦ CHANGE Add the following into `src/containers/Home.css`.

```
.Home .lander {
  padding: 80px 0;
  text-align: center;
}

.Home .lander h1 {
  font-family: "Open Sans", sans-serif;
  font-weight: 600;
}
```

```
.Home .lander p {
  color: #999;
}
```

Set up the Routes

Now we'll set up the routes so that we can have this container respond to the `/` route.

✦ CHANGE Create `src/Routes.js` and add the following into it.

```
import React from "react";
import { Route, Switch } from "react-router-dom";
import Home from "./containers/Home";

export default function Routes() {
  return (
    <Switch>
      <Route path="/" exact component={Home} />
    </Switch>
  );
}
```

This component uses this `Switch` component from React-Router that renders the first matching route that is defined within it. For now we only have a single route, it looks for `/` and renders the `Home` component when matched. We are also using the `exact` prop to ensure that it matches the `/` route exactly. This is because the path `/` will also match any route that starts with a `/`.

Render the Routes

Now let's render the routes into our App component.

✦ CHANGE Add the following to the header of your `src/App.js`.

```
import Routes from "./Routes";
```

✦ CHANGE And add the following line below our `Navbar` component inside the `render` of `src/App.js`.

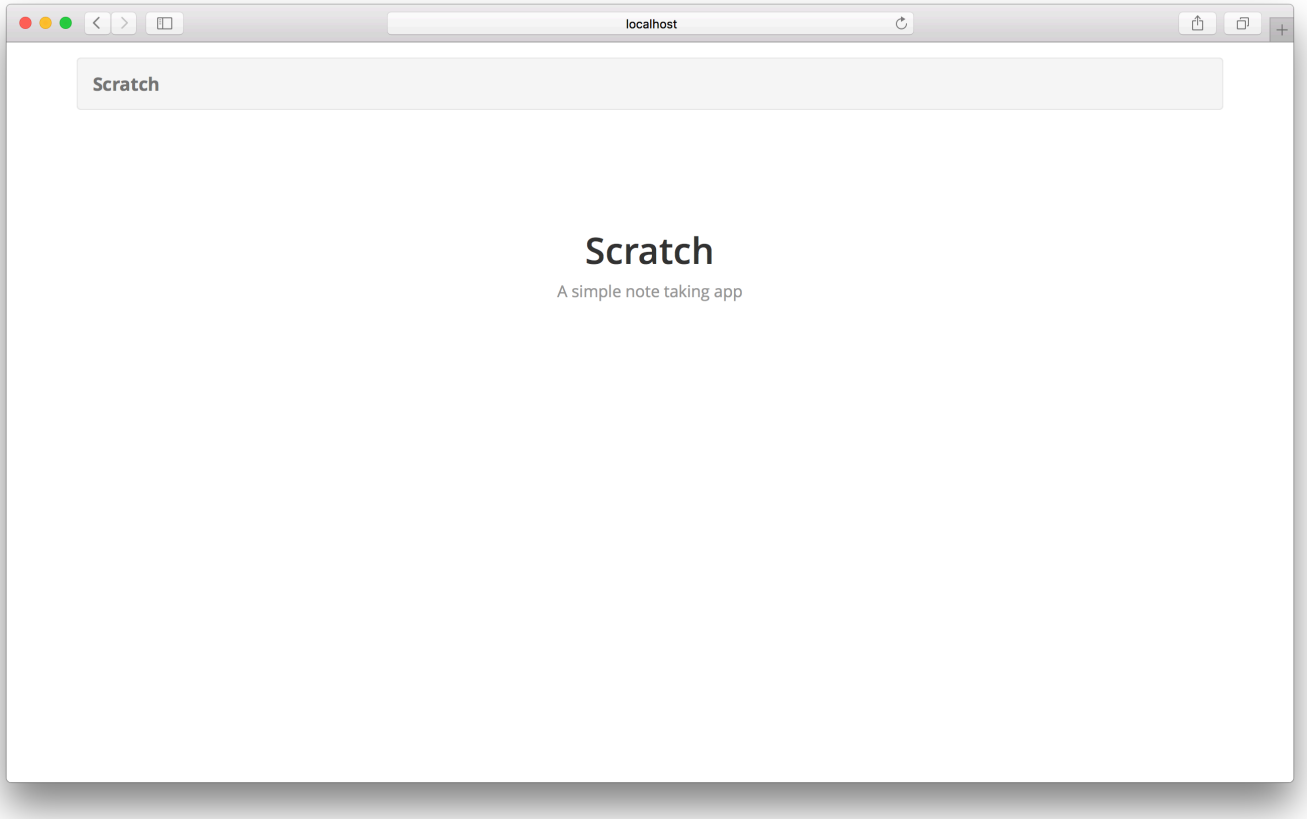
```
<Routes />
```

So the `App` function component of our `src/App.js` should now look like this.

```
function App(props) {
  return (
    <div className="App container">
      <Navbar fluid collapseOnSelect>
        <Navbar.Header>
          <Navbar.Brand>
            <Link to="/">Scratch</Link>
          </Navbar.Brand>
          <Navbar.Toggle />
        </Navbar.Header>
      </Navbar>
      <Routes />
    </div>
  );
}
```

This ensures that as we navigate to different routes in our app, the portion below the navbar will change to reflect that.

Finally, head over to your browser and your app should show the brand new homepage of your app.



Next we are going to add login and signup links to our navbar.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-containers/62)

Adding Links in the Navbar

Now that we have our first route set up, let's add a couple of links to the navbar of our app. These will direct users to login or signup for our app when they first visit it.

✦ CHANGE Replace the `App` function component in `src/App.js` with the following.

```
function App(props) {
  return (
    <div className="App container">
      <Navbar fluid collapseOnSelect>
        <Navbar.Header>
          <Navbar.Brand>
            <Link to="/">Scratch</Link>
          </Navbar.Brand>
          <Navbar.Toggle />
        </Navbar.Header>
        <Navbar.Collapse>
          <Nav pullRight>
            <NavItem href="/signup">Signup</NavItem>
            <NavItem href="/login">Login</NavItem>
          </Nav>
        </Navbar.Collapse>
      </Navbar>
      <Routes />
    </div>
```

```
    );
  }
```

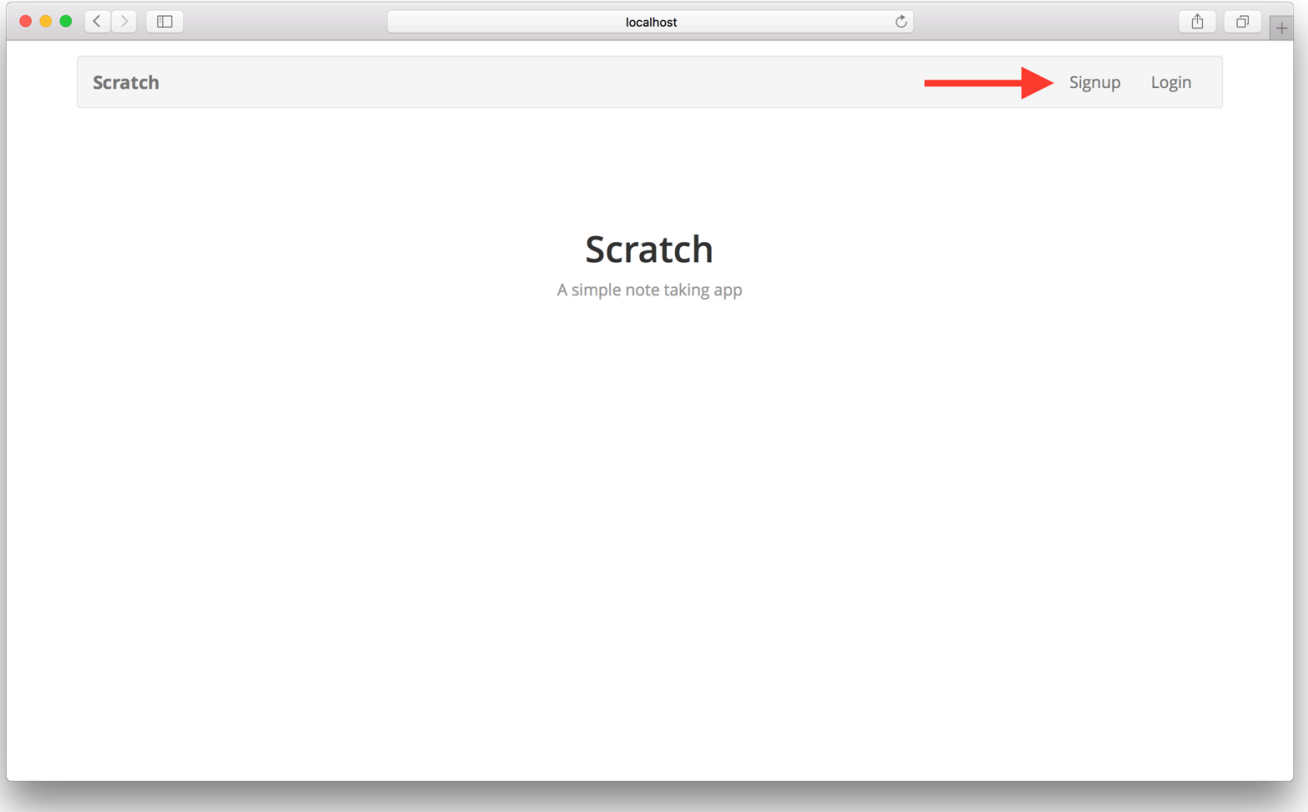
This adds two links to our navbar using the `NavItem` Bootstrap component. The `Navbar.Collapse` component ensures that on mobile devices the two links will be collapsed.

And let's include the necessary components in the header.

✦ CHANGE Replace the `react-router-dom` and `react-bootstrap` import in `src/App.js` with this.

```
import { Link } from "react-router-dom";
import { Nav, Navbar, NavItem } from "react-bootstrap";
```

Now if you flip over to your browser, you should see the two links in our navbar.



Unfortunately, when you click on them they refresh your browser while redirecting to the link. We need it to route it to the new link without refreshing the page since we are building a single page app.

To fix this we need a component that works with React Router and React Bootstrap called React Router Bootstrap (https://github.com/react-bootstrap/react-router-bootstrap). It can wrap around your `Navbar` links and use the React Router to route your app to the required link without refreshing the browser.

✦ CHANGE Run the following command in your working directory.

```
$ npm install react-router-bootstrap --save
```

✦ CHANGE And include it at the top of your `src/App.js`.

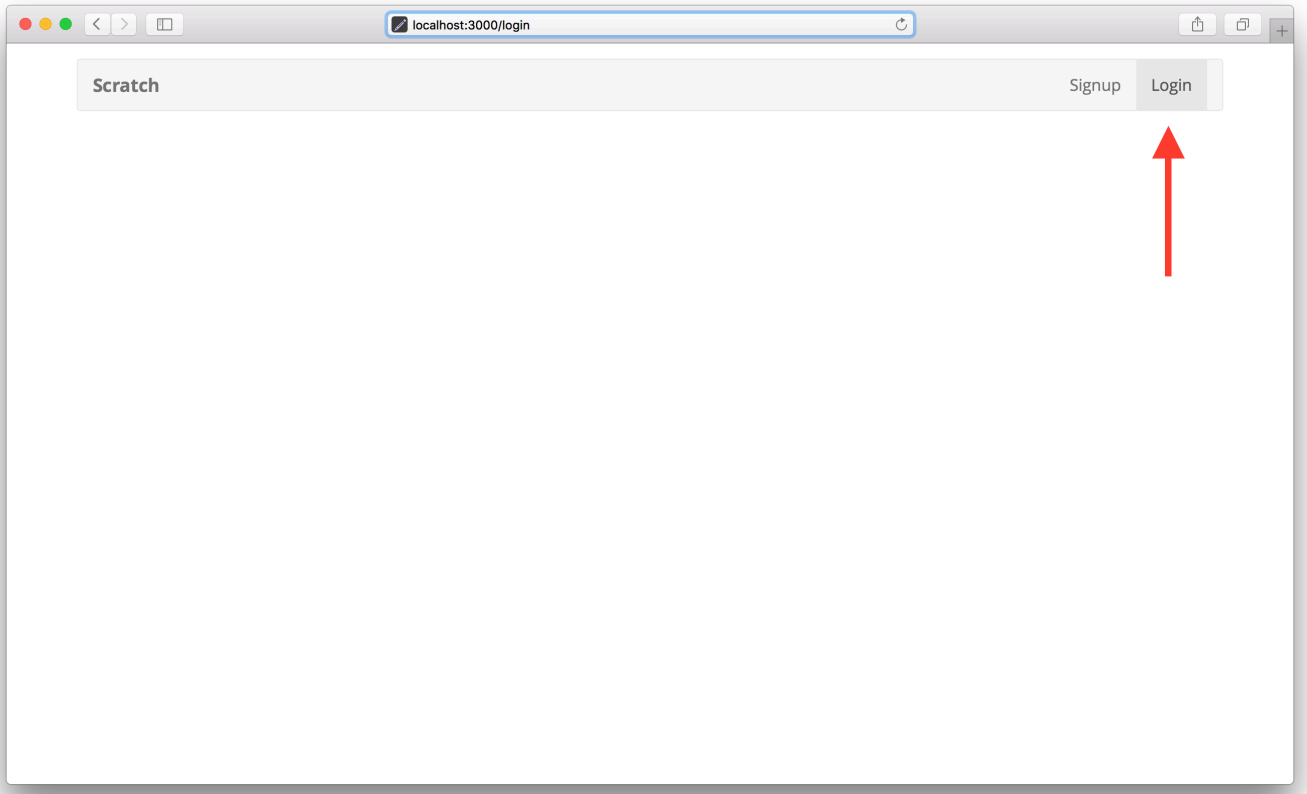
```
import { LinkContainer } from "react-router-bootstrap";
```

✦ CHANGE We will now wrap our links with the `LinkContainer`. Replace the `App` function component in your `src/App.js` with this.

```
function App(props) {
  return (
    <div className="App container">
      <Navbar fluid collapseOnSelect>
        <Navbar.Header>
          <Navbar.Brand>
            <Link to="/">Scratch</Link>
          </Navbar.Brand>
          <Navbar.Toggle />
        </Navbar.Header>
```

```
          <Navbar.Collapse>
            <Nav pullRight>
              <LinkContainer to="/signup">
                <NavItem>Signup</NavItem>
              </LinkContainer>
              <LinkContainer to="/login">
                <NavItem>Login</NavItem>
              </LinkContainer>
            </Nav>
          </Navbar.Collapse>
        </Navbar>
        <Routes />
      </div>
    );
}
```

And that's it! Now if you flip over to your browser and click on the login link, you should see the link highlighted in the navbar. Also, it doesn't refresh the page while redirecting.



You'll notice that we are not rendering anything on the page because we don't have a login page currently. We should handle the case when a requested page is not found.

Next let's look at how to tackle handling 404s with our router.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/adding-links-in-the-navbar/141)

Handle 404s

Now that we know how to handle the basic routes; let's look at handling 404s with the React Router.

Create a Component

Let's start by creating a component that will handle this for us.

✦ CHANGE Create a new component at `src/containers/NotFound.js` and add the following.

```
import React from "react";
import "./NotFound.css";

export default function NotFound() {
  return (
    <div className="NotFound">
      <h3>Sorry, page not found!</h3>
    </div>
  );
}
```

All this component does is print out a simple message for us.

✦ CHANGE Let's add a couple of styles for it in `src/containers/NotFound.css`.

```
.NotFound {
  padding-top: 100px;
  text-align: center;
}
```

Add a Catch All Route

Now we just need to add this component to our routes to handle our 404s.

✦ CHANGE Find the `<Switch>` block in `src/Routes.js` and add it as the last line in that section.

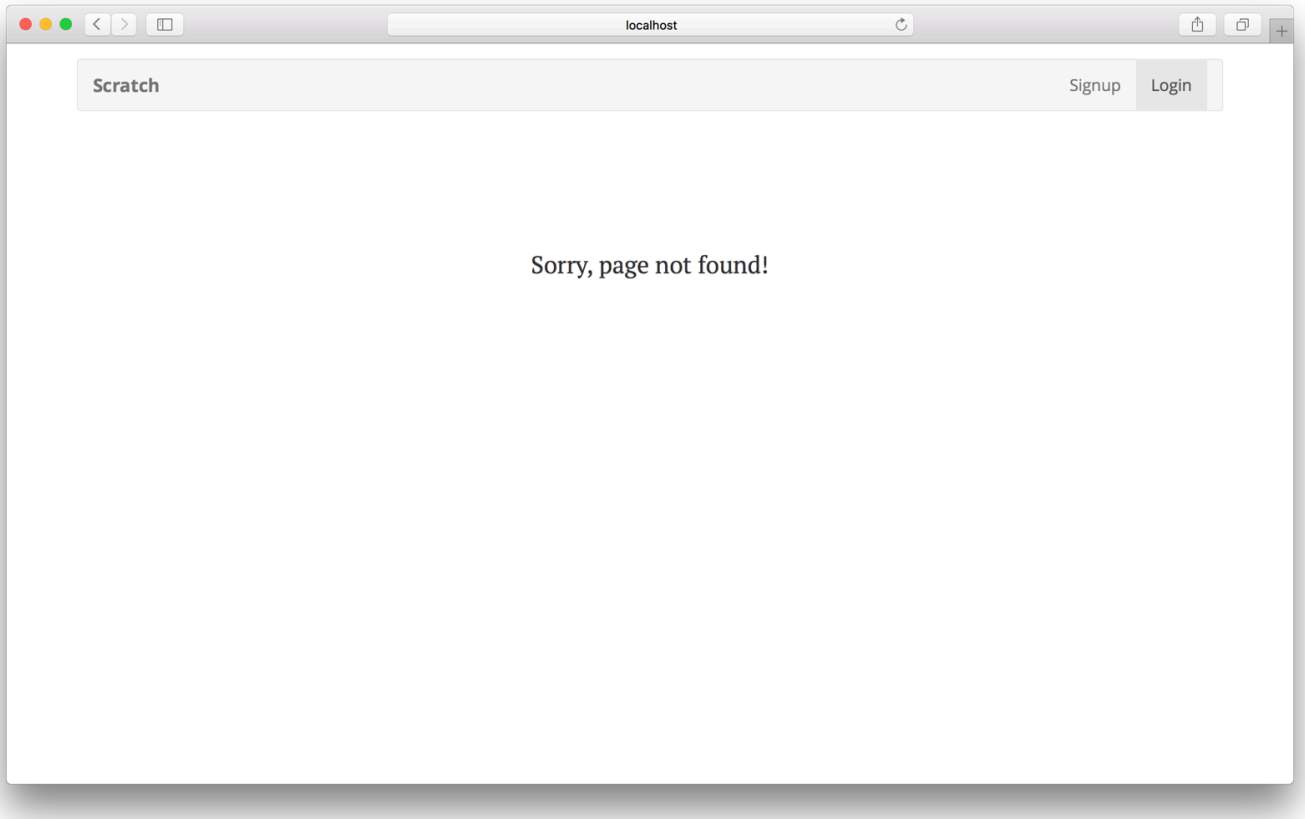
```
{ /* Finally, catch all unmatched routes */ }
<Route component={NotFound} />
```

This needs to always be the last line in the `<Route>` block. You can think of it as the route that handles requests in case all the other routes before it have failed.

✦ CHANGE And include the `NotFound` component in the header by adding the following:

```
import NotFound from "./containers/NotFound";
```

And that's it! Now if you were to switch over to your browser and try clicking on the Login or Signup buttons in the Nav you should see the 404 message that we have.



Next up, we are going to configure our app with the info of our backend resources.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/handle-404s/75)

Configure AWS Amplify

To allow our React app to talk to the AWS resources that we created (in the backend section of the tutorial), we'll be using a library called AWS Amplify (https://github.com/aws/aws-amplify).

AWS Amplify provides a few simple modules (Auth, API, and Storage) to help us easily connect to our backend. Let's get started.

Install AWS Amplify

✦ CHANGE Run the following command in your working directory.

```
$ npm install aws-amplify --save
```

This installs the NPM package and adds the dependency to your `package.json`.

Create a Config

Let's first create a configuration file for our app that'll reference all the resources we have created.

✦ CHANGE Create a file at `src/config.js` and add the following.

```
export default {
  s3: {
    REGION: "YOUR_S3_UPLOADS_BUCKET_REGION",
```

```
    BUCKET: "YOUR_S3_UPLOADS_BUCKET_NAME"
  },
  apiGateway: {
    REGION: "YOUR_API_GATEWAY_REGION",
    URL: "YOUR_API_GATEWAY_URL"
  },
  cognito: {
    REGION: "YOUR_COGNITO_REGION",
    USER_POOL_ID: "YOUR_COGNITO_USER_POOL_ID",
    APP_CLIENT_ID: "YOUR_COGNITO_APP_CLIENT_ID",
    IDENTITY_POOL_ID: "YOUR_IDENTITY_POOL_ID"
  }
};
```

Here you need to replace the following:

1. `YOUR_S3_UPLOADS_BUCKET_NAME` and `YOUR_S3_UPLOADS_BUCKET_REGION` with the your S3 Bucket name and region from the Create an S3 bucket for file uploads (/chapters/create-an-s3-bucket-for-file-uploads.html) chapter. In our case it is `notes-app-uploads` and `us-east-1`.

2. `YOUR_API_GATEWAY_URL` and `YOUR_API_GATEWAY_REGION` with the ones from the Deploy the APIs (/chapters/deploy-the-apis.html) chapter. In our case the URL is `https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod` and the region is `us-east-1`.

3. `YOUR_COGNITO_USER_POOL_ID`, `YOUR_COGNITO_APP_CLIENT_ID`, and `YOUR_COGNITO_REGION` with the Cognito **Pool Id**, **App Client id**, and region from the Create a Cognito user pool (/chapters/create-a-

cognito-user-pool.html) chapter.

4. `YOUR_IDENTITY_POOL_ID` with your **Identity pool ID** from the Create a Cognito identity pool (/chapters/create-a-cognito-identity-pool.html) chapter.

Add AWS Amplify

Next we'll set up AWS Amplify.

✦ CHANGE Import it by adding the following to the header of your `src/index.js`.

```
import Amplify from 'aws-amplify';
```

And import the config we created above.

✦ CHANGE Add the following, also to the header of your `src/index.js`.

```
import config from './config';
```

✦ CHANGE And to initialize AWS Amplify; add the following above the `ReactDOM.render` line in `src/index.js`.

```
Amplify.configure({
  Auth: {
    mandatorySignIn: true,
    region: config.cognito.REGION,
    userPoolId: config.cognito.USER_POOL_ID,
    identityPoolId: config.cognito.IDENTITY_POOL_ID,
    userPoolWebClientId: config.cognito.APP_CLIENT_ID
  },
```

```
  Storage: {
    region: config.s3.REGION,
    bucket: config.s3.BUCKET,
    identityPoolId: config.cognito.IDENTITY_POOL_ID
  },
  API: {
    endpoints: [
      {
        name: "notes",
        endpoint: config.apiGateway.URL,
        region: config.apiGateway.REGION
      },
    ]
  }
});
```

A couple of notes here.

- Amplify refers to Cognito as `Auth`, S3 as `Storage`, and API Gateway as `API`.
- The `mandatorySignIn` flag for `Auth` is set to true because we want our users to be signed in before they can interact with our app.
- The `name: "notes"` is basically telling Amplify that we want to name our API. Amplify allows you to add multiple APIs that your app is going to work with. In our case our entire backend is just one single API.
- The `Amplify.configure()` is just setting the various AWS resources that we want to interact with. It isn't doing anything else special here beside configuration. So while this might look

intimidating, just remember this is only setting things up.

Commit the Changes

CHANGE Let's commit our code so far and push it to GitHub.

```
$ git add .
$ git commit -m "Setting up our React app"
$ git push
```

Next up, we are going to work on creating our login and sign up forms.

CHECKPOINT

For reference, here is the code we have so far.

Frontend Source :configure-aws-amplify (https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/configure-aws-amplify)

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/configure-aws-amplify/151)

Create a Login Page

Let's create a page where the users of our app can login with their credentials. When we created our User Pool we asked it to allow a user to sign in and sign up with their email as their username. We'll be touching on this further when we create the signup form.

So let's start by creating the basic form that'll take the user's email (as their username) and password.

Add the Container

✦ CHANGE Create a new file `src/containers/Login.js` and add the following.

```
import React, { useState } from "react";
import { Button, FormGroup, FormControl, ControlLabel }
from "react-bootstrap";
import "./Login.css";

export default function Login(props) {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");

  function validateForm() {
    return email.length > 0 && password.length > 0;
  }
```

```
  function handleSubmit(event) {
    event.preventDefault();
  }

  return (
    <div className="Login">
      <form onSubmit={handleSubmit}>
        <FormGroup controlId="email" bsSize="large">
          <ControlLabel>Email</ControlLabel>
          <FormControl
            autoFocus
            type="email"
            value={email}
            onChange={e => setEmail(e.target.value)}
          />
        </FormGroup>
        <FormGroup controlId="password" bsSize="large">
          <ControlLabel>Password</ControlLabel>
          <FormControl
            value={password}
            onChange={e => setPassword(e.target.value)}
            type="password"
          />
        </FormGroup>
        <Button block bsSize="large" disabled=
{!validateForm()} type="submit">
          Login
        </Button>
      </form>
    </div>
```

```
    );
}
```

We are introducing a couple of new concepts in this.

1. Right at the top of our component, we are using the useState hook (https://reactjs.org/docs/hooks-state.html) to store what the user enters in the form. The `useState` hook just gives you the current value of the variable you want to store in the state and a function to set the new value. If you are transitioning from Class components to using React Hooks, we've added a chapter to help you understand how Hooks work (/chapters/understanding-react-hooks.html).

2. We then connect the state to our two fields in the form using the `setEmail` and `setPassword` functions to store what the user types in — `e.target.value`. Once we set the new state, our component gets re-rendered. The variables `email` and `password` now have the new values.

3. We are setting the form controls to show the value of our two state variables `email` and `password`. In React, this pattern of displaying the current form value as a state variable and setting the new one when a user types something, is called a Controlled Component (https://reactjs.org/docs/forms.html#controlled-components).

4. We are setting the `autoFocus` flag for our email field, so that when our form loads, it sets focus to this field.

5. We also link up our submit button with our state by using a validate function called `validateForm`. This simply checks if our fields are non-empty, but can easily do something more complicated.

6. Finally, we trigger our callback `handleSubmit` when the form is submitted. For now we are simply suppressing the browsers default behavior on submit but we'll do more here later.

CHANGE Let's add a couple of styles to this in the file `src/containers/Login.css`.

```
@media all and (min-width: 480px) {
  .Login {
    padding: 60px 0;
  }

  .Login form {
    margin: 0 auto;
    max-width: 320px;
  }
}
```

These styles roughly target any non-mobile screen sizes.

Add the Route

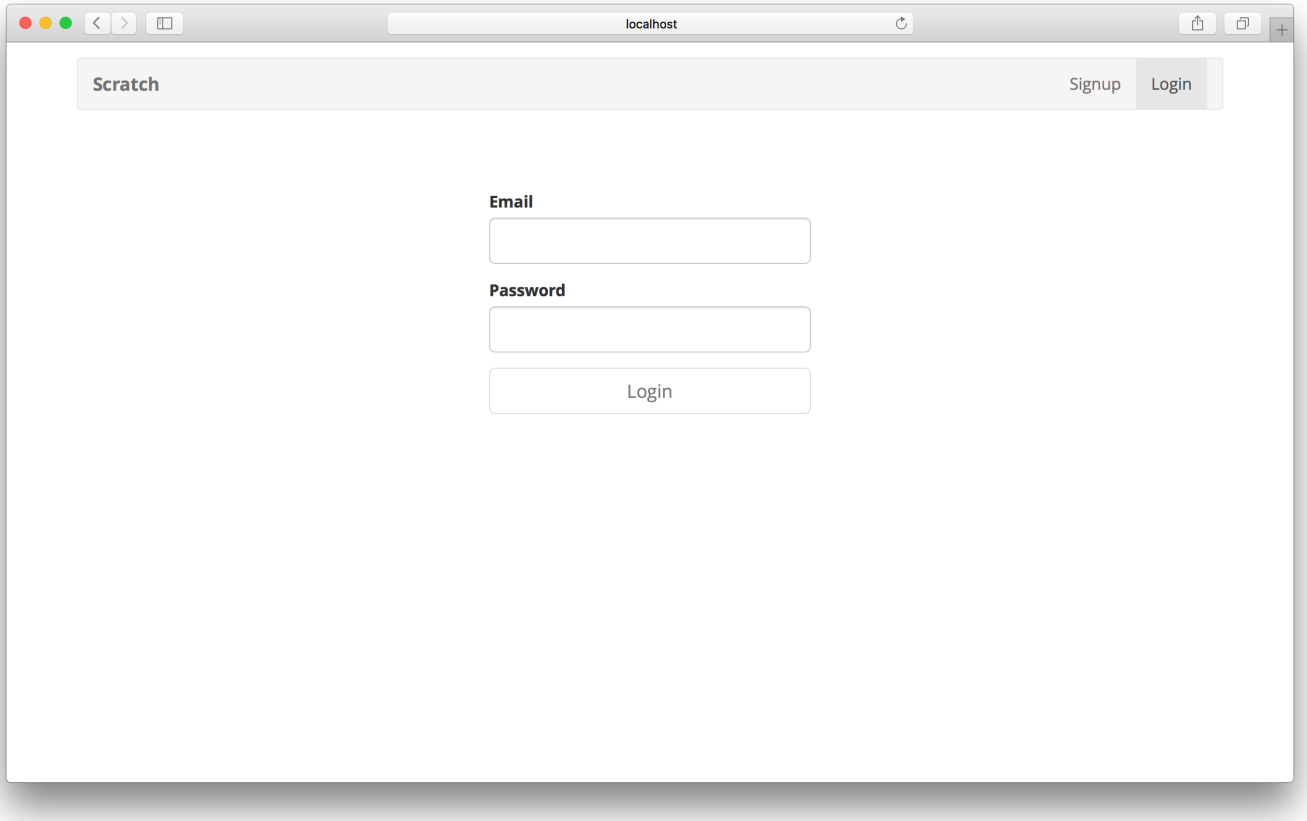
CHANGE Now we link this container up with the rest of our app by adding the following line to `src/Routes.js` below our home `<Route>`.

```
<Route path="/login" exact component={Login} />
```

CHANGE And include our component in the header.

```
import Login from "./containers/Login";
```

Now if we switch to our browser and navigate to the login page we should see our newly created form.



Next, let's connect our login form to our AWS Cognito set up.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-login-page/71)

Login with AWS Cognito

We are going to use AWS Amplify to login to our Amazon Cognito setup. Let's start by importing it.

Import Auth from AWS Amplify

✦ CHANGE Add the following to the header of our Login container in `src/containers/Login.js`.

```
import { Auth } from "aws-amplify";
```

Login to Amazon Cognito

The login code itself is relatively simple.

✦ CHANGE Simply replace our placeholder `handleSubmit` method in `src/containers/Login.js` with the following.

```
async function handleSubmit(event) {
  event.preventDefault();

  try {
    await Auth.signIn(email, password);
    alert("Logged in");
  } catch (e) {
    alert(e.message);
```

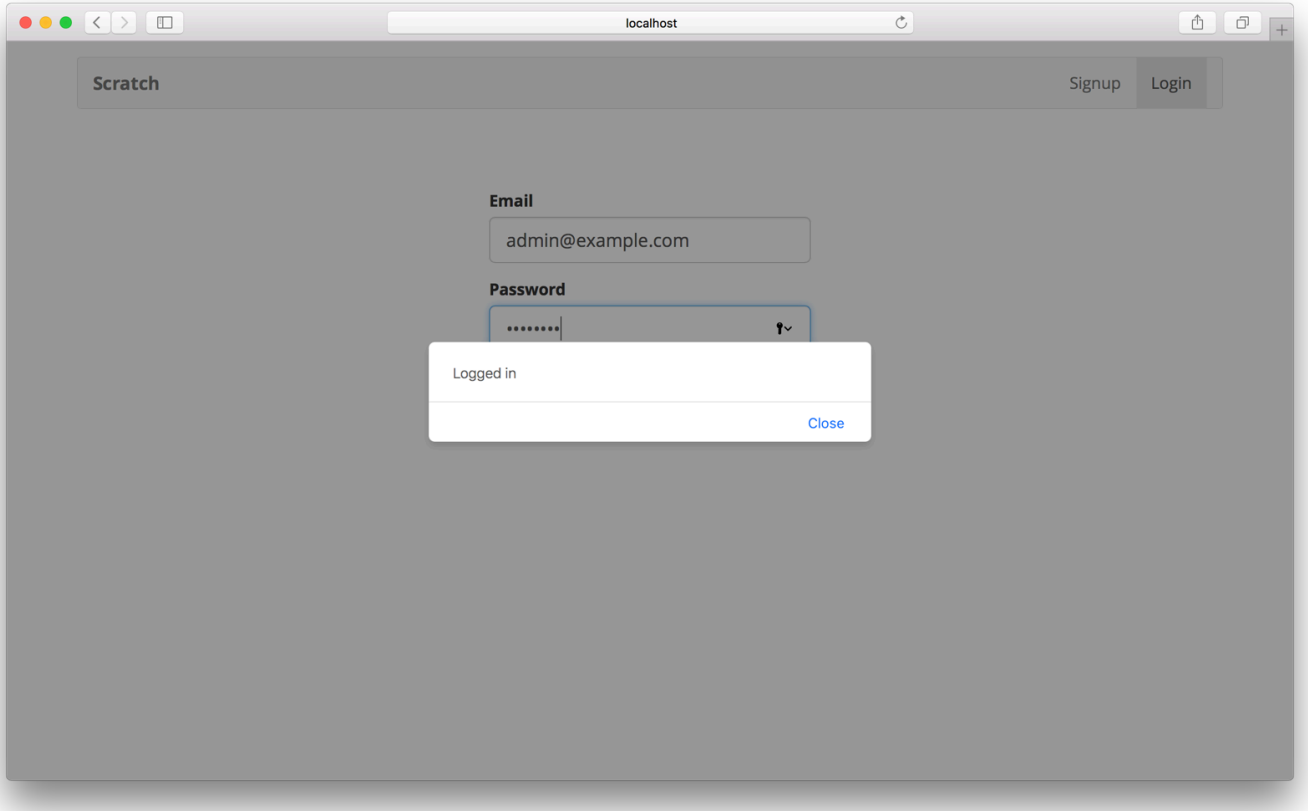
```
    }
  }
```

We are doing two things of note here.

1. We grab the `email` and `password` and call Amplify's `Auth.signIn()` method. This method returns a promise since it will be logging in the user asynchronously.

2. We use the `await` keyword to invoke the `Auth.signIn()` method that returns a promise. And we need to label our `handleSubmit` method as `async`.

Now if you try to login using the `admin@example.com` user (that we created in the Create a Cognito Test User (/chapters/create-a-cognito-test-user.html) chapter), you should see the browser alert that tells you that the login was successful.



Next, we'll take a look at storing the login state in our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/login-with-aws-cognito/129)

Add the Session to the State

To complete the login process we would need to update the app state with the session to reflect that the user has logged in.

Update the App State

First we'll start by updating the application state by setting that the user is logged in. We might be tempted to store this in the `Login` container, but since we are going to use this in a lot of other places, it makes sense to lift up the state. The most logical place to do this will be in our `App` component.

To save the user's login state, let's include the `useState` hook in `src/App.js`.

✦ CHANGE Replace, the `React` import:

```
import React from "react";
```

✦ CHANGE With the following:

```
import React, { useState } from "react";
```

✦ CHANGE Add the following to the top of our `App` component function.

```
const [isAuthenticated, userHasAuthenticated] =
```

```
useState(false);
```

This initializes the `isAuthenticated` state variable to `false`, as in the user is not logged in. And calling `userHasAuthenticated` updates it. But for the `Login` container to call this method we need to pass a reference of this method to it.

Pass the Session State to the Routes

We can do this by passing in a couple of props to the child component of the routes that the `App` component creates.

✦ CHANGE Replace our `Routes` component by replacing the following line in the `return` statement of `src/App.js`.

```
<Routes />
```

✦ CHANGE With this.

```
<Routes appProps={{ isAuthenticated,
userHasAuthenticated }} />
```

Currently, our `Routes` component don't do anything with the passed in `appProps`. We need it to apply these props to the child component it is going to render. In this case we need it to apply them to our `Login` component.

To do this we are going to create a new component that'll extend the standard Route component and apply the `appProps`.

✦ CHANGE Create a `src/components/` directory by running this

command in your working directory.

```
$ mkdir src/components/
```

Here we'll be storing all our React components that are not dealing directly with our API or responding to routes.

✦ CHANGE Create a new component in `src/components/AppliedRoute.js` and add the following.

```
import React from "react";
import { Route } from "react-router-dom";

export default function AppliedRoute({ component: C,
appProps, ...rest }) {
  return (
    <Route {...rest} render={props => <C {...props}
{...appProps} />} />
  );
}
```

This simple component creates a `Route` where the child component that it renders contains the passed in props. Let's take a quick look at how this being done.

- The `Route` component takes a prop called `component` that represents the component that will be rendered when a matching route is found. We want our `appProps` to be applied to this component.
- The `Route` component can also take a `render` method in place of

the `component`. This allows us to control what is passed in to our component.

- Based on this we can create a component that returns a `Route` that takes a `component` and `appProps` prop. This allows us to pass in the component we want rendered and the props that we want applied.
- Finally, we take `component` (set as `C`) and `appProps` and render inside our `Route` using the inline function; `props => <C {...props} {...appProps} />`. Note, the `props` variable in this case is what the Route component passes us. Whereas, the `appProps` are the props that we are trying to set in our `App` component.

Now to use this component, we are going to include it in the routes where we need to have the `appProps` passed in.

✦ CHANGE Replace the `Routes` function in `src/Routes.js` with the following.

```
export default function Routes({ appProps }) {
  return (
    <Switch>
      <AppliedRoute path="/" exact component={Home}
appProps={appProps} />
      <AppliedRoute path="/login" exact component=
{Login} appProps={appProps} />
      { /* Finally, catch all unmatched routes */ }
      <Route component={NotFound} />
    </Switch>
  );
}
```

✦ CHANGE And import the new component in the header of `src/Routes.js`.

```
import AppliedRoute from "./components/AppliedRoute";
```

Now in the `Login` container we'll call the `userHasAuthenticated` method.

✦ CHANGE Replace the `alert('Logged in');` line with the following in `src/containers/Login.js`.

```
props.userHasAuthenticated(true);
```

Create a Logout Button

We can now use this to display a Logout button once the user logs in. Find the following in our `src/App.js`.

```
<LinkContainer to="/signup">
  <NavItem>Signup</NavItem>
</LinkContainer>
<LinkContainer to="/login">
  <NavItem>Login</NavItem>
</LinkContainer>
```

✦ CHANGE And replace it with this:

```
{isAuthenticated
  ? <NavItem onClick={handleLogout}>Logout</NavItem>
  : <>
```

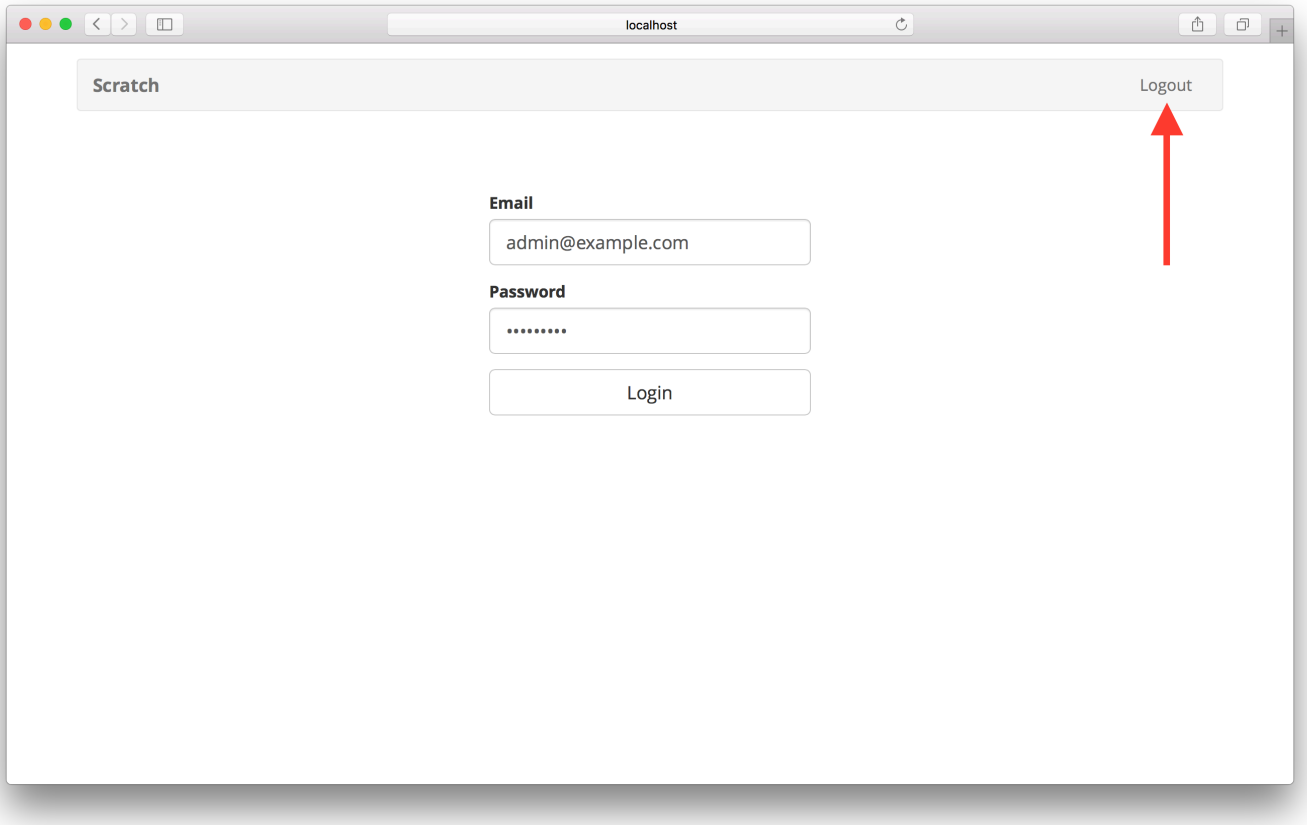
```
        <LinkContainer to="/signup">
          <NavItem>Signup</NavItem>
        </LinkContainer>
        <LinkContainer to="/login">
          <NavItem>Login</NavItem>
        </LinkContainer>
      </>
    }
```

The `<>` or Fragment component (https://reactjs.org/docs/fragments.html) can be thought of as a placeholder component. We need this because in the case the user is not logged in, we want to render two links. To do this we would need to wrap it inside a single component, like a `div`. But by using the Fragment component it tells React that the two links are inside this component but we don't want to render any extra HTML.

✦ CHANGE And add this `handleLogout` method to `src/App.js` above the `render() {` line as well.

```
function handleLogout() {
  userHasAuthenticated(false);
}
```

Now head over to your browser and try logging in with the admin credentials we created in the Create a Cognito Test User (/chapters/create-a-cognito-test-user.html) chapter. You should see the Logout button appear right away.



Now if you refresh your page you should be logged out again. This is because we are not initializing the state from the browser session. Let's look at how to do that next.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/add-the-session-to-the-state/136)

Load the State from the Session

To make our login information persist we need to store and load it from the browser session. There are a few different ways we can do this, using Cookies or Local Storage. Thankfully the AWS Amplify does this for us automatically and we just need to read from it and load it into our application state.

Amplify gives us a way to get the current user session using the `Auth.currentSession()` method. It returns a promise that resolves to the session object (if there is one).

Load User Session

Let's load this when our app loads. To do this we are going to use another React hook, called useEffect (https://reactjs.org/docs/hooks-effect.html). Since `Auth.currentSession()` returns a promise, it means that we need to ensure that the rest of our app is only ready to go after this has been loaded.

✦ CHANGE To do this, let's add another state variable to our `src/App.js` state called `isAuthenticating`. Add it to the top of our `App` function.

```
const [isAuthenticating, setIsAuthenticating] =
useState(true);
```

We start with the value set to `true` because as we first load our app, it'll start by checking the current authentication state.

✦ CHANGE Let's include the `Auth` module by adding the following to the header of `src/App.js`.

```
import { Auth } from "aws-amplify";
```

✦ CHANGE Now to load the user session we'll add the following to our `src/App.js` right below our variable declarations.

```
useEffect(() => {
  onLoad();
}, []);

async function onLoad() {
  try {
    await Auth.currentSession();
    userHasAuthenticated(true);
  }
  catch(e) {
    if (e !== 'No current user') {
      alert(e);
    }
  }

  setIsAuthenticating(false);
}
```

Let's understand how this and the `useEffect` hook works.

The `useEffect` hook take a function and an array of variables. The function will be called every time the component is rendered. And the array of variables tell React to only re-run our function if the passed in array of variables have changed. This allows us to control when our function gets run. This has some neat consequences:

1. If we don't pass in an array of variables, our hook get's executed everytime our component is rendered.
2. If we pass in some variables, on every render React will first check if those variables have changed, before running our function.
3. If we pass in an empty list of variables, then it'll only run our function on the FIRST render.

In our case, we only want to check the user's authentication state when our app first loads. So we'll use the third option; just pass in an empty list of variables — `[]`.

When our app first loads, it'll run the `onLoad` function. All this does is load the current session. If it loads, then it updates the `isAuthenticating` state variable once the process is complete. It does so by calling `setIsAuthenticating(false)`. The `Auth.currentSession()` method throws an error `No current user` if nobody is currently logged in. We don't want to show this error to users when they load up our app and are not signed in. Once `Auth.currentSession()` runs successfully, we call `userHasAuthenticated(true)` to set that the user is logged in.

So the top of our `App` function should now look like this:

```
function App(props) {
  const [isAuthenticating, setIsAuthenticating] =
```

```
useState(true);
  const [isAuthenticated, userHasAuthenticated] =
useState(false);

  useEffect(() => {
    onLoad();
  }, []);

  ...
```

✦ CHANGE Let's make sure to include the `useEffect` hook by replacing the React import in the header of `src/App.js` with:

```
import React, { useState, useEffect } from "react";
```

Render When the State Is Ready

Since loading the user session is an asynchronous process, we want to ensure that our app does not change states when it first loads. To do this we'll hold off rendering our app till `isAuthenticating` is `false`.

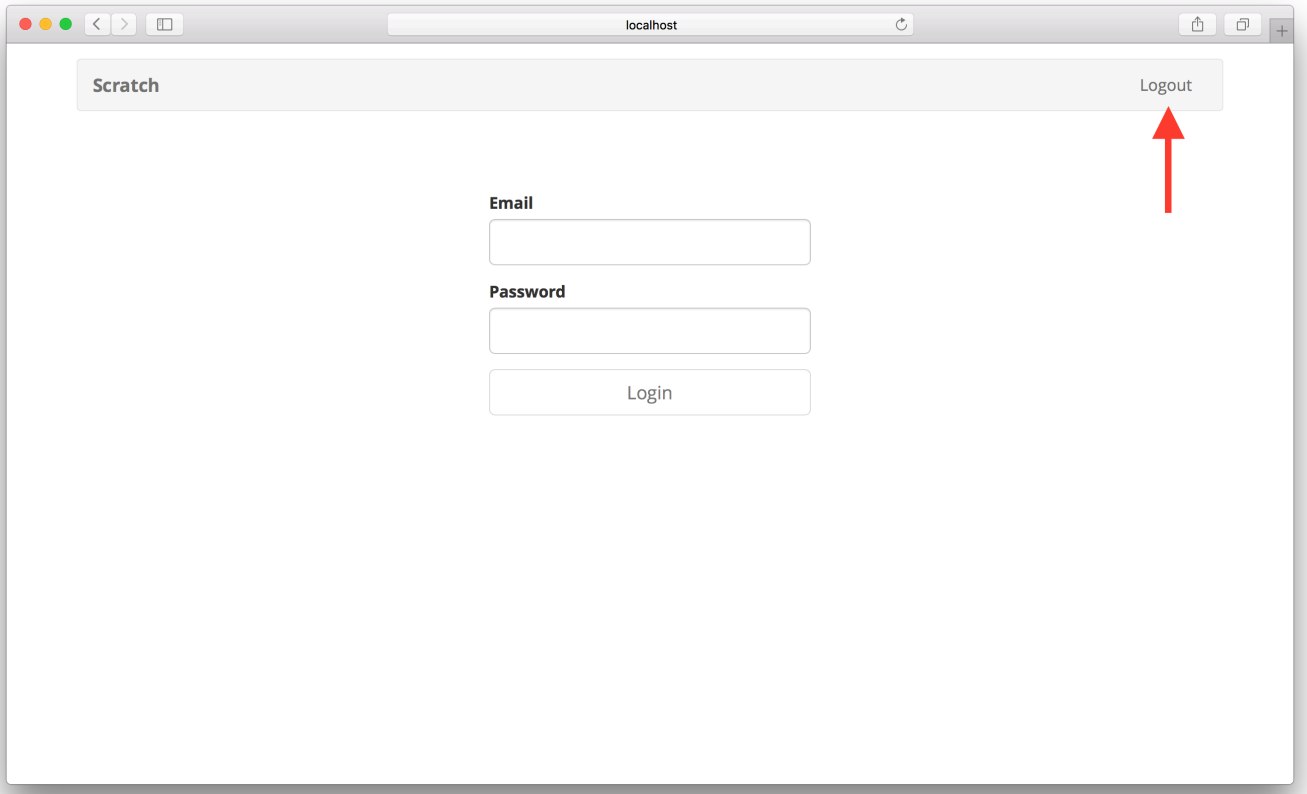
We'll conditionally render our app based on the `isAuthenticating` flag.

✦ CHANGE Our `return` statement in `src/App.js` should be as follows.

```
return (
  !isAuthenticating &&
  <div className="App container">
    <Navbar fluid collapseOnSelect>
      <Navbar.Header>
        <Navbar.Brand>
```

```
            <Link to="/">Scratch</Link>
          </Navbar.Brand>
          <Navbar.Toggle />
        </Navbar.Header>
        <Navbar.Collapse>
          <Nav pullRight>
            {isAuthenticated
              ? <NavItem onClick=
{handleLogout}>Logout</NavItem>
              : <>
                  <LinkContainer to="/signup">
                    <NavItem>Signup</NavItem>
                  </LinkContainer>
                  <LinkContainer to="/login">
                    <NavItem>Login</NavItem>
                  </LinkContainer>
                </>
            }
          </Nav>
        </Navbar.Collapse>
      </Navbar>
      <Routes appProps={{ isAuthenticated,
userHasAuthenticated }} />
    </div>
);
```

Now if you head over to your browser and refresh the page, you should see that a user is logged in.



Unfortunately, when we hit Logout and refresh the page; we are still logged in. To fix this we are going to clear the session on logout next.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/load-the-state-from-the-session/157)

Clear the Session on Logout

Currently we are only removing the user session from our app's state. But when we refresh the page, we load the user session from the browser Local Storage (using Amplify), in effect logging them back in.

AWS Amplify has a `Auth.signOut()` method that helps clear it out.

✦ CHANGE Let's replace the `handleLogout` function in our `src/App.js` with this:

```
async function handleLogout() {
  await Auth.signOut();

  userHasAuthenticated(false);
}
```

Now if you head over to your browser, logout and then refresh the page; you should be logged out completely.

If you try out the entire login flow from the beginning you'll notice that, we continue to stay on the login page through out the entire process. Next, we'll look at redirecting the page after we login and logout to make the flow make more sense.

For help and discussion

💬 Comments on this chapter (https://discourse.serverless-stack.com/t/clear-the-session-on-logout/70)

Redirect on Login and Logout

To complete the login flow we are going to need to do two more things.

1. Redirect the user to the homepage after they login.
2. And redirect them back to the login page after they logout.

We are going to use the `history.push` method that comes with React Router v4.

Redirect to Home on Login

Since our `Login` component is rendered using a `Route`, it adds the router props to it. So we can redirect using the `this.props.history.push` method.

```
props.history.push("/");
```

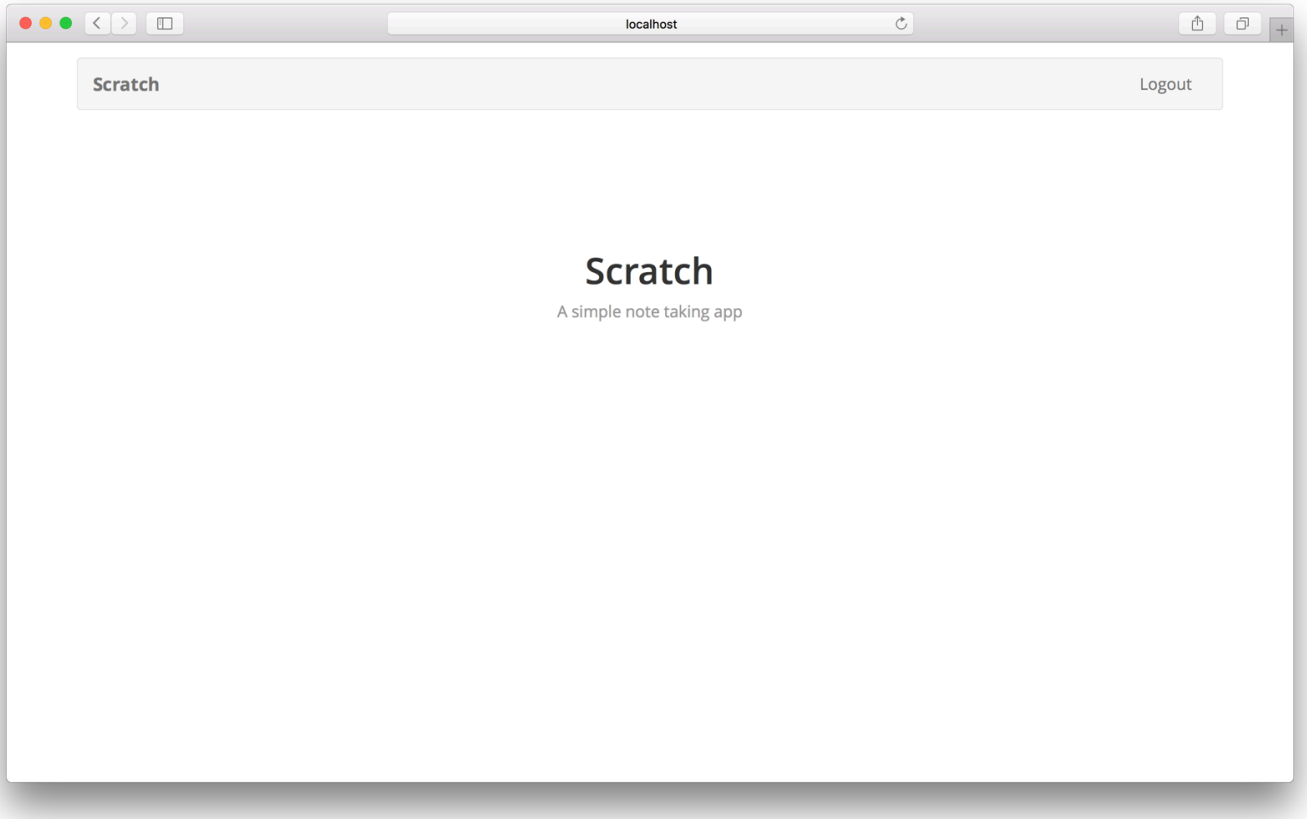
✦ CHANGE Update the `handleSubmit` method in `src/containers/Login.js` to look like this:

```
async function handleSubmit(event) {
  event.preventDefault();

  try {
    await Auth.signIn(email, password);
    props.userHasAuthenticated(true);
```

```
      props.history.push("/");
    } catch (e) {
      alert(e.message);
    }
  }
```

Now if you head over to your browser and try logging in, you should be redirected to the homepage after you've been logged in.



Redirect to Login After Logout

Now we'll do something very similar for the logout process. However, the `App` component does not have access to the router props directly since it is not rendered inside a `Route` component. To be able to use the router props in our `App` component we will need to use the `withRouter` Higher-Order Component (https://facebook.github.io/react/docs/higher-

order-components.html) (or HOC). You can read more about the `withRouter` HOC here (https://reacttraining.com/react-router/web/api/withRouter).

To use this HOC, we'll change the way we export our App component.

✦ CHANGE Replace the following line in `src/App.js`.

```
export default App;
```

✦ CHANGE With this.

```
export default withRouter(App);
```

✦ CHANGE And import `withRouter` by replacing the `import { Link }` line in the header of `src/App.js` with this:

```
import { Link, withRouter } from "react-router-dom";
```

✦ CHANGE Add the following to the bottom of the `handleLogout` function in our `src/App.js`.

```
props.history.push("/login");
```

So our `handleLogout` method should now look like this.

```
async function handleLogout() {
  await Auth.signOut();

  userHasAuthenticated(false);
```

```
  props.history.push("/login");
}
```

This redirects us back to the login page once the user logs out.

Now if you switch over to your browser and try logging out, you should be redirected to the login page.

You might have noticed while testing this flow that since the login call has a bit of a delay, we might need to give some feedback to the user that the login call is in progress. Let's do that next.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/redirect-on-login-and-logout/154)

Give Feedback While Logging In

It's important that we give the user some feedback while we are logging them in. So they get the sense that the app is still working, as opposed to being unresponsive.

Use an isLoading Flag

✦ CHANGE To do this we are going to add an `isLoading` flag to the state of our `src/containers/Login.js`. Add the following to the top of our `Login` function component.

```
const [isLoading, setIsLoading] = useState(false);
```

✦ CHANGE And we'll update it while we are logging in. So our `handleSubmit` function now looks like so:

```
async function handleSubmit(event) {
  event.preventDefault();

  setIsLoading(true);

  try {
    await Auth.signIn(email, password);
    props.userHasAuthenticated(true);
```

```
    props.history.push("/");
  } catch (e) {
    alert(e.message);
    setIsLoading(false);
  }
}
```

Create a Loader Button

Now to reflect the state change in our button we are going to render it differently based on the `isLoading` flag. But we are going to need this piece of code in a lot of different places. So it makes sense that we create a reusable component out of it.

✦ CHANGE Create a new file and add the following in `src/components/LoaderButton.js`.

```
import React from "react";
import { Button, Glyphicon } from "react-bootstrap";
import "./LoaderButton.css";

export default function LoaderButton({
  isLoading,
  className = "",
  disabled = false,
  ...props
}) {
  return (
    <Button
      className={`LoaderButton ${className}`}
      disabled={disabled || isLoading}
```

```
      {...props}
    >
      {isLoading && <Glyphicon glyph="refresh"
classNameName="spinning" />}
      {props.children}
    </Button>
  );
}
```

This is a really simple component that takes an `isLoading` flag and the text that the button displays in the two states (the default state and the loading state). The `disabled` prop is a result of what we have currently in our `Login` button. And we ensure that the button is disabled when `isLoading` is `true`. This makes it so that the user can't click it while we are in the process of logging them in.

And let's add a couple of styles to animate our loading icon.

✦ CHANGE Add the following to `src/components/LoaderButton.css`.

```
.LoaderButton .spinning.glyphicon {
  margin-right: 7px;
  top: 2px;
  animation: spin 1s infinite linear;
}
@keyframes spin {
  from { transform: scale(1) rotate(0deg); }
  to { transform: scale(1) rotate(360deg); }
}
```

This spins the refresh Glyphicon infinitely with each spin taking a second.

And by adding these styles as a part of the `LoaderButton` we keep them self contained within the component.

Render Using the isLoading Flag

Now we can use our new component in our `Login` container.

CHANGE In `src/containers/Login.js` find the `<Button>` component in the `return` statement.

```
<Button block bsSize="large" disabled={!validateForm()}
type="submit">
  Login
</Button>
```

CHANGE And replace it with this.

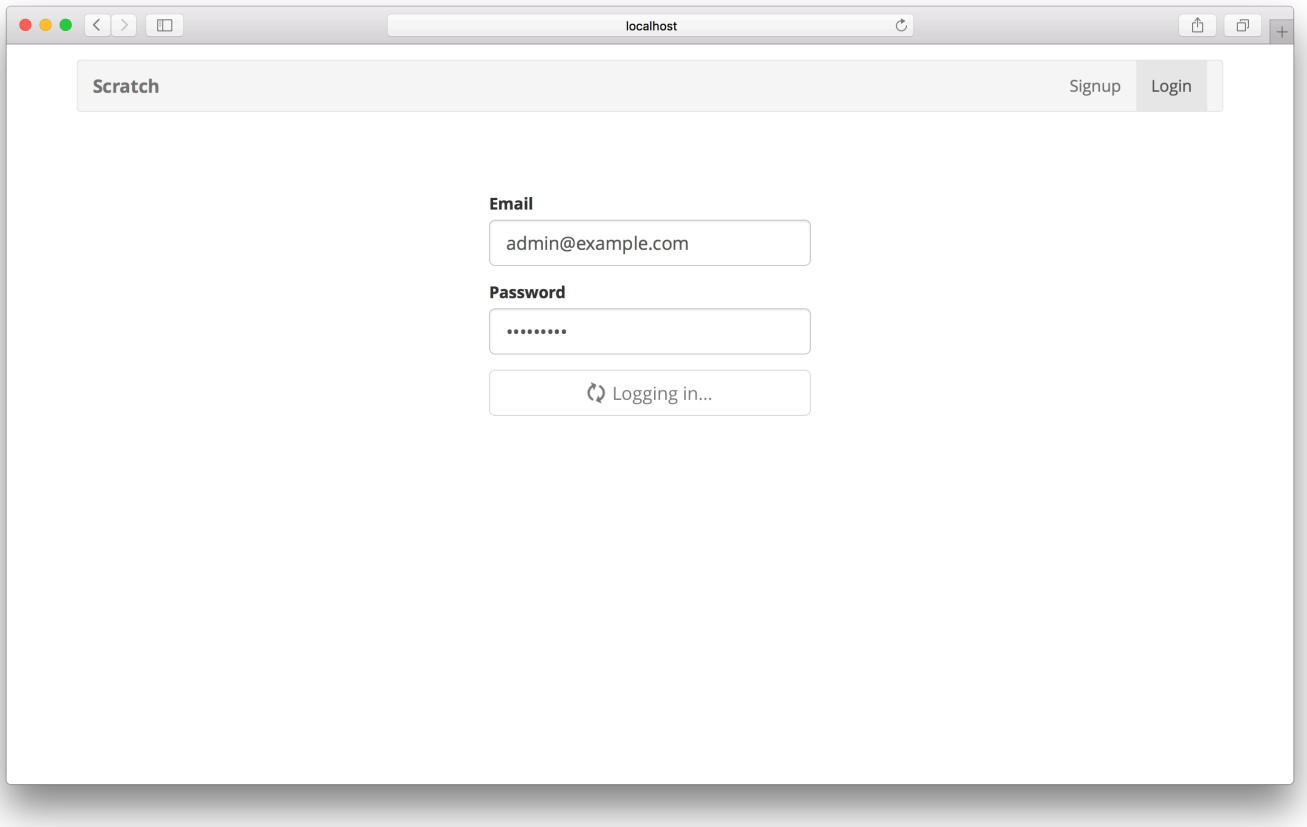
```
<LoaderButton
  block
  type="submit"
  bsSize="large"
  isLoading={isLoading}
  disabled={!validateForm()}
>
  Login
</LoaderButton>
```

CHANGE Also, import the `LoaderButton` in the header. And remove the reference to the `Button` component.

```
import { FormGroup, FormControl, ControlLabel } from
```

```
"react-bootstrap";
import LoaderButton from "../components/LoaderButton";
```

And now when we switch over to the browser and try logging in, you should see the intermediate state before the login completes.



If you would like to add *Forgot Password* functionality for your users, you can refer to our Extra Credit series of chapters on user management (/chapters/manage-user-accounts-in-aws-amplify.html).

Next let's implement the sign up process for our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-

```
stack.com/t/give-feedback-while-logging-
in/46)
```

Create a Custom React Hook to Handle Form Fields

Now before we move on to creating our sign up page, we are going to take a short detour to simplify how we handle form fields in React. We built a form as a part of our login page and we are going to do the same for our sign up page. You'll recall that in our login component we were creating two state variables to store the username and password.

```
const [email, setEmail] = useState("");
const [password, setPassword] = useState("");
```

And we also use something like this to set the state:

```
onChange={e => setEmail(e.target.value)}
```

Now we are going to do something similar for our sign up page and it'll have a few more fields than the login page. So it makes sense to simplify this process and have some common logic that all our form related components can share. Plus this is a good way to introduce the biggest benefit of React Hooks — reusing stateful logic between components.

Creating a Custom React Hook

Let's pull the above logic into a custom React Hook.

✦ CHANGE Create a `src/libs/` directory to store all our common code.

```
$ mkdir src/libs/
```

✦ CHANGE Add the following to `src/libs/hooksLib.js`.

```
import { useState } from "react";

export function useFormFields(initialState) {
  const [fields, setValues] = useState(initialState);

  return [
    fields,
    function(event) {
      setValues({
        ...fields,
        [event.target.id]: event.target.value
      });
    }
  ];
}
```

Creating a custom hook is amazingly simple. Let's go over how this works:

1. A custom React Hook starts with the word `use` in its name. So ours is called `useFormFields`.
2. Our Hook takes the initial state of our form fields as an object and saves it as a state variable called `fields`.
3. It returns an array with `fields` and a function that sets the new state based on the event object. The only difference here is that we

are using `event.target.id` (which contains the id of our form field) to store the value (`event.target.value`).

And that's it! We can now use this in our Login component.

Using Our Custom Hook

✦ CHANGE Replace our `src/containers/Login.js` with the following:

```
import React, { useState } from "react";
import { Auth } from "aws-amplify";
import { FormGroup, FormControl, ControlLabel } from
"react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import { useFormFields } from "../libs/hooksLib";
import "./Login.css";

export default function Login(props) {
  const [isLoading, setIsLoading] = useState(false);
  const [fields, handleFieldChange] = useFormFields({
    email: "",
    password: ""
  });

  function validateForm() {
    return fields.email.length > 0 &&
fields.password.length > 0;
  }

  async function handleSubmit(event) {
    event.preventDefault();
```

```
    setIsLoading(true);

    try {
      await Auth.signIn(fields.email, fields.password);
      props.userHasAuthenticated(true);
      props.history.push("/");
    } catch (e) {
      alert(e.message);
      setIsLoading(false);
    }
  }

  return (
    <div className="Login">
      <form onSubmit={handleSubmit}>
        <FormGroup controlId="email" bsSize="large">
          <ControlLabel>Email</ControlLabel>
          <FormControl
            autoFocus
            type="email"
            value={fields.email}
            onChange={handleFieldChange}
          />
        </FormGroup>
        <FormGroup controlId="password" bsSize="large">
          <ControlLabel>Password</ControlLabel>
          <FormControl
            type="password"
            value={fields.password}
            onChange={handleFieldChange}
```

```
          />
        </FormGroup>
        <LoaderButton
          block
          type="submit"
          bsSize="large"
          isLoading={isLoading}
          disabled={!validateForm()}
        >
          Login
        </LoaderButton>
      </form>
    </div>
  );
}
```

You'll notice that we are using our `useFormFields` Hook. A good way to think about custom React Hooks is to simply replace the line where we use it, with the Hook code itself. So instead of this line:

```
const [fields, handleFieldChange] = useFormFields({
  email: "",
  password: ""
});
```

Simply imagine the code for the `useFormFields` function instead!

Finally, we are setting our fields using the function our custom Hook is returning.

```
onChange={handleFieldChange}
```

Now we are ready to tackle our sign up page.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-custom-react-hook-to-handle-form-fields/1316)

Create a Signup Page

The signup page is quite similar to the login page that we just created. But it has a couple of key differences. When we sign the user up, AWS Cognito sends them a confirmation code via email. We also need to authenticate the new user once they've confirmed their account.

So the signup flow will look something like this:

1. The user types in their email, password, and confirms their password.
2. We sign them up with Amazon Cognito using the AWS Amplify library and get a user object in return.
3. We then render a form to accept the confirmation code that AWS Cognito has emailed to them.
4. We confirm the sign up by sending the confirmation code to AWS Cognito.
5. We authenticate the newly created user.
6. Finally, we update the app state with the session.

So let's get started by creating the basic sign up form first.

For help and discussion

Comments on this chapter

```
(https://discourse.serverless-
stack.com/t/create-a-signup-page/65)
```

Create the Signup Form

Let's start by creating the signup form that'll get the user's email and password.

Add the Container

✦ CHANGE Create a new container at `src/containers/Signup.js` with the following.

```
import React, { useState } from "react";
import {
  HelpBlock,
  FormGroup,
  FormControl,
  ControlLabel
} from "react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import { useFormFields } from "../libs/hooksLib";
import "./Signup.css";

export default function Signup(props) {
  const [fields, handleFieldChange] = useFormFields({
    email: "",
    password: "",
    confirmPassword: "",
    confirmationCode: ""
```

```
});
const [newUser, setNewUser] = useState(null);
const [isLoading, setIsLoading] = useState(false);

function validateForm() {
  return (
    fields.email.length > 0 &&
    fields.password.length > 0 &&
    fields.password === fields.confirmPassword
  );
}

function validateConfirmationForm() {
  return fields.confirmationCode.length > 0;
}

async function handleSubmit(event) {
  event.preventDefault();

  setIsLoading(true);

  setNewUser("test");

  setIsLoading(false);
}

async function handleConfirmationSubmit(event) {
  event.preventDefault();

  setIsLoading(true);
}
```

```
  function renderConfirmationForm() {
    return (
      <form onSubmit={handleConfirmationSubmit}>
        <FormGroup controlId="confirmationCode"
bsSize="large">
          <ControlLabel>Confirmation
Code</ControlLabel>
          <FormControl
            autoFocus
            type="tel"
            onChange={handleFieldChange}
            value={fields.confirmationCode}
          />
          <HelpBlock>Please check your email for the
code.</HelpBlock>
        </FormGroup>
        <LoaderButton
          block
          type="submit"
          bsSize="large"
          isLoading={isLoading}
          disabled={!validateConfirmationForm()}
        >
          Verify
        </LoaderButton>
      </form>
    );
  }

  function renderForm() {
```

```
    return (
      <form onSubmit={handleSubmit}>
        <FormGroup controlId="email" bsSize="large">
          <ControlLabel>Email</ControlLabel>
          <FormControl
            autoFocus
            type="email"
            value={fields.email}
            onChange={handleFieldChange}
          />
        </FormGroup>
        <FormGroup controlId="password" bsSize="large">
          <ControlLabel>Password</ControlLabel>
          <FormControl
            type="password"
            value={fields.password}
            onChange={handleFieldChange}
          />
        </FormGroup>
        <FormGroup controlId="confirmPassword"
bsSize="large">
          <ControlLabel>Confirm Password</ControlLabel>
          <FormControl
            type="password"
            onChange={handleFieldChange}
            value={fields.confirmPassword}
          />
        </FormGroup>
        <LoaderButton
          block
          type="submit"
```

```
          bsSize="large"
          isLoading={isLoading}
          disabled={!validateForm()}
        >
          Signup
        </LoaderButton>
      </form>
    );
  }

  return (
    <div className="Signup">
      {newUser === null ? renderForm() :
renderConfirmationForm()}
    </div>
  );
}
```

Most of the things we are doing here are fairly straightforward but let's go over them quickly.

1. Since we need to show the user a form to enter the confirmation code, we are conditionally rendering two forms based on if we have a user object or not.
2. We are using the `LoaderButton` component that we created earlier for our submit buttons.
3. Since we have two forms we have two validation functions called `validateForm` and `validateConfirmationForm`.
4. We are setting the `autoFocus` flags on the email and the

confirmation code fields.

5. For now our `handleSubmit` and `handleConfirmationSubmit` don't do a whole lot besides setting the `isLoading` state and a dummy value for the `newUser` state.

6. And you'll notice we are using the `useFormFields` custom React Hook that we previously created (/chapters/create-a-custom-react-hook-to-handle-form-fields.html) to handle our form fields.

CHANGE Also, let's add a couple of styles in `src/containers/Signup.css`.

```
@media all and (min-width: 480px) {
  .Signup {
    padding: 60px 0;
  }

  .Signup form {
    margin: 0 auto;
    max-width: 320px;
  }
}

.Signup form span.help-block {
  font-size: 14px;
  padding-bottom: 10px;
  color: #999;
}
```

Add the Route

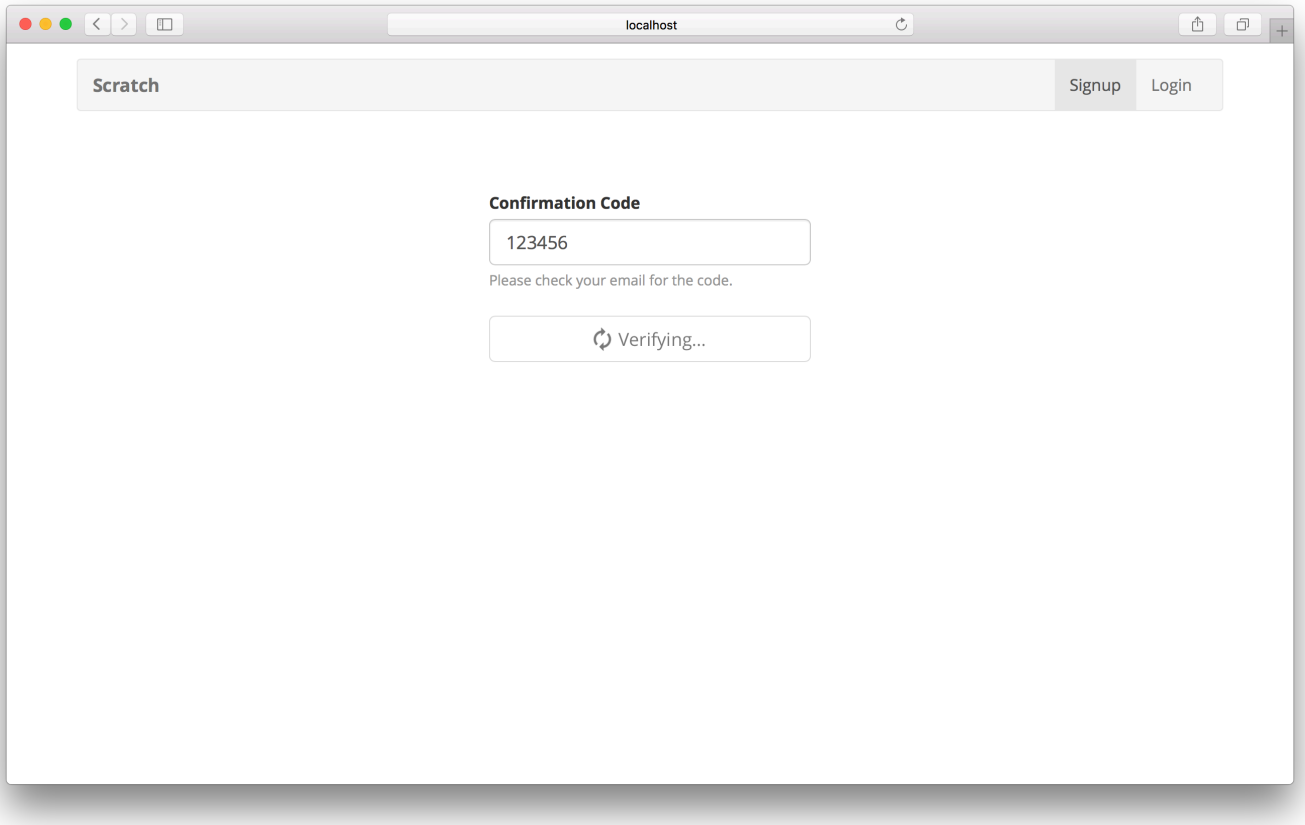
CHANGE Finally, add our container as a route in `src/Routes.js` below our login route. We are using the `AppliedRoute` component that we created in the Add the session to the state (/chapters/add-the-session-to-the-state.html) chapter.

```
<AppliedRoute path="/signup" exact component={Signup}
appProps={appProps} />
```

CHANGE And include our component in the header.

```
import Signup from "./containers/Signup";
```

Now if we switch to our browser and navigate to the signup page we should see our newly created form. Our form doesn't do anything when we enter in our info but you can still try to fill in an email address, password, and the confirmation code. It'll give you an idea of how the form will behave once we connect it to Cognito.



Next, let's connect our signup form to Amazon Cognito.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-the-signup-form/52)

Signup with AWS Cognito

Now let's go ahead and implement the `handleSubmit` and `handleConfirmationSubmit` functions and connect it up with our AWS Cognito setup.

✦ CHANGE Replace our `handleSubmit` and `handleConfirmationSubmit` functions in `src/containers/Signup.js` with the following.

```
async function handleSubmit(event) {
  event.preventDefault();

  setIsLoading(true);

  try {
    const newUser = await Auth.signUp({
      username: fields.email,
      password: fields.password
    });
    setIsLoading(false);
    setNewUser(newUser);
  } catch (e) {
    alert(e.message);
    setIsLoading(false);
  }
}
```

```
async function handleConfirmationSubmit(event) {
  event.preventDefault();

  setIsLoading(true);

  try {
    await Auth.confirmSignUp(fields.email,
fields.confirmationCode);
    await Auth.signIn(fields.email, fields.password);

    props.userHasAuthenticated(true);
    props.history.push("/");
  } catch (e) {
    alert(e.message);
    setIsLoading(false);
  }
}
```

✦ CHANGE Also, include the Amplify Auth in our header.

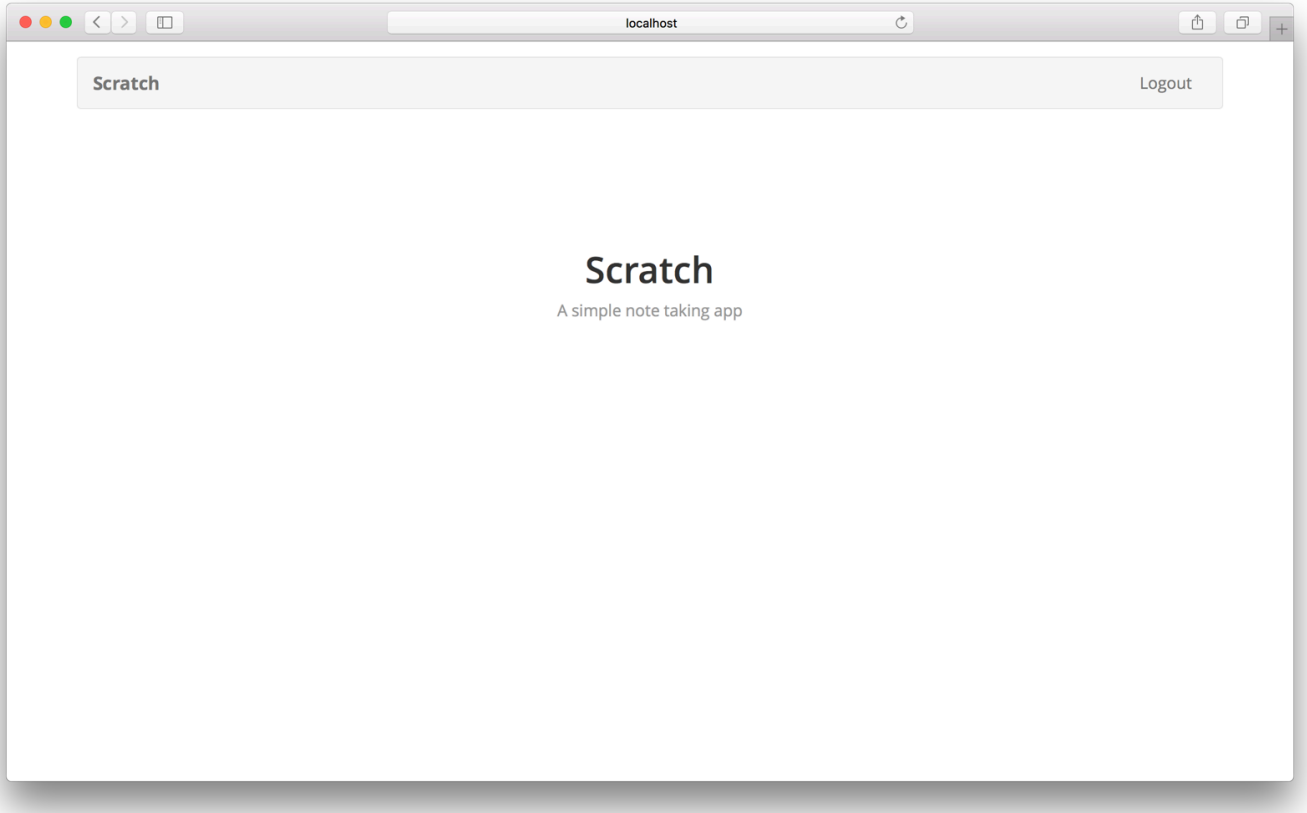
```
import { Auth } from "aws-amplify";
```

The flow here is pretty simple:

1. In `handleSubmit` we make a call to signup a user. This creates a new user object.
2. Save that user object to the state using `setNewUser`.
3. In `handleConfirmationSubmit` use the confirmation code to confirm the user.

4. With the user now confirmed, Cognito now knows that we have a new user that can login to our app.
5. Use the email and password to authenticate exactly the same way we did in the login page.
6. Update the App's state using the `userHasAuthenticated` function.
7. Finally, redirect to the homepage.

Now if you were to switch over to your browser and try signing up for a new account it should redirect you to the homepage after sign up successfully completes.



A quick note on the signup flow here. If the user refreshes their page at the confirm step, they won't be able to get back and confirm that account. It forces them to create a new account instead. We are keeping things intentionally simple but here are a couple of hints on how to fix it.

1. Check for the `UsernameExistsException` in the `handleSubmit` function's `catch` block.
2. Use the `Auth.resendSignUp()` method to resend the code if the user has not been previously confirmed. Here is a link to the Amplify API docs (https://aws.github.io/aws-amplify/api/classes/authclass.html#resendsignup).
3. Confirm the code just as we did before.

Give this a try and post in the comments if you have any questions.

Now while developing you might run into cases where you need to manually confirm an unauthenticated user. You can do that with the AWS CLI using the following command.

```
aws cognito-idp admin-confirm-sign-up \
   --region YOUR_COGNITO_REGION \
   --user-pool-id YOUR_COGNITO_USER_POOL_ID \
   --username YOUR_USER_EMAIL
```

Just be sure to use your Cognito User Pool Id and the email you used to create the account.

If you would like to allow your users to change their email or password, you can refer to our Extra Credit series of chapters on user management (/chapters/manage-user-accounts-in-aws-amplify.html).

Next up, we are going to create our first note.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/signup-with-aws-cognito/130)

Add the Create Note Page

Now that we can signup users and also log them in. Let's get started with the most important part of our note taking app; the creation of a note.

First we are going to create the form for a note. It'll take some content and a file as an attachment.

Add the Container

✦ CHANGE Create a new file `src/containers/NewNote.js` and add the following.

```
import React, { useRef, useState } from "react";
import { FormGroup, FormControl, ControlLabel } from
"react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import config from "../config";
import "./NewNote.css";

export default function NewNote(props) {
  const file = useRef(null);
  const [content, setContent] = useState("");
  const [isLoading, setIsLoading] = useState(false);

  function validateForm() {
    return content.length > 0;
  }
```

```
  function handleFileChange(event) {
    file.current = event.target.files[0];
  }

  async function handleSubmit(event) {
    event.preventDefault();

    if (file.current && file.current.size >
config.MAX_ATTACHMENT_SIZE) {
      alert(
        `Please pick a file smaller than
${config.MAX_ATTACHMENT_SIZE /
          1000000} MB.`
      );
      return;
    }

    setIsLoading(true);
  }

  return (
    <div className="NewNote">
      <form onSubmit={handleSubmit}>
        <FormGroup controlId="content">
          <FormControl
            value={content}
            componentClass="textarea"
            onChange={e => setContent(e.target.value)}
          />
        </FormGroup>
```

```
        <FormGroup controlId="file">
          <ControlLabel>Attachment</ControlLabel>
          <FormControl onChange={handleFileChange}
type="file" />
        </FormGroup>
        <LoaderButton
          block
          type="submit"
          bsSize="large"
          bsStyle="primary"
          isLoading={isLoading}
          disabled={!validateForm()}
        >
          Create
        </LoaderButton>
      </form>
    </div>
  );
}
```

Everything is fairly standard here, except for the file input. Our form elements so far have been controlled components (https://facebook.github.io/react/docs/forms.html), as in their value is directly controlled by the state of the component. However, in the case of the file input we want the browser to handle this state. So instead of `useState` we'll use the `useRef` hook. The main difference between the two is that `useRef` does not cause the component to re-render. It simply tells React to store a value for us so that we can use it later. We can set/get the current value of a ref by using its `current` property. Just as we do when the user selects a file.

```
file.current = event.target.files[0];
```

Currently, our `handleSubmit` does not do a whole lot other than limiting the file size of our attachment. We are going to define this in our config.

✦ CHANGE So add the following to our `src/config.js` below the `export default {` line.

```
MAX_ATTACHMENT_SIZE: 5000000,
```

✦ CHANGE Let's also add the styles for our form in `src/containers/NewNote.css`.

```
.NewNote form {
  padding-bottom: 15px;
}

.NewNote form textarea {
  height: 300px;
  font-size: 24px;
}
```

Add the Route

✦ CHANGE Finally, add our container as a route in `src/Routes.js` below our signup route. We are using the `AppliedRoute` component that we created in the Add the session to the state (/chapters/add-the-session-to-the-state.html) chapter.

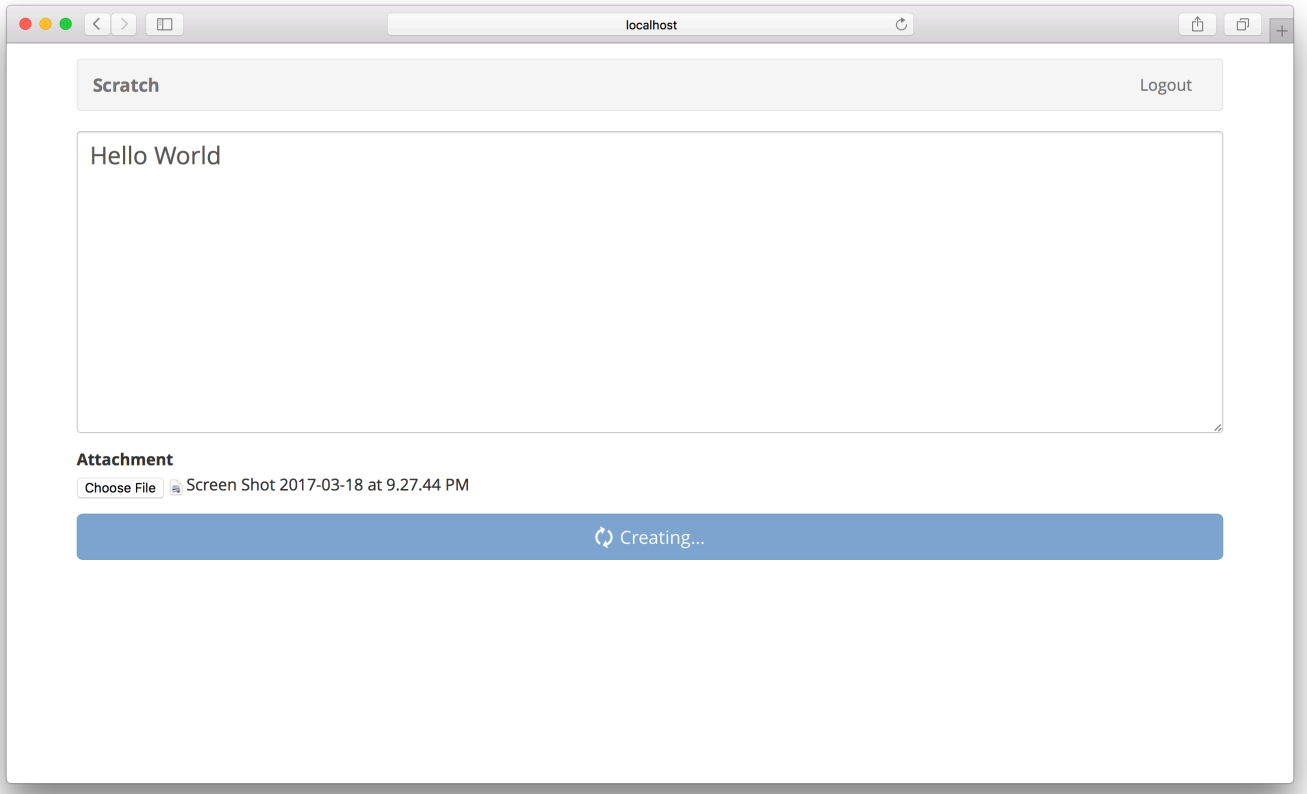
```
<AppliedRoute path="/notes/new" exact component=
```

```
{NewNote} appProps={appProps} />
```

CHANGE And include our component in the header.

```
import NewNote from "./containers/NewNote";
```

Now if we switch to our browser and navigate `http://localhost:3000/notes/new` we should see our newly created form. Try adding some content, uploading a file, and hitting submit to see it in action.



Next, let's get into connecting this form to our API.

For help and discussion

Comments on this chapter

(https://discourse.serverless-stack.com/t/add-the-create-note-page/107)

Call the Create API

Now that we have our basic create note form working, let's connect it to our API. We'll do the upload to S3 a little bit later. Our APIs are secured using AWS IAM and Cognito User Pool is our authentication provider. Thankfully, Amplify takes care of this for us by using the logged in user's session.

We just need to use the `API` module that AWS Amplify has.

✦ CHANGE Let's include the `API` module by adding the following to the header of `src/containers/NewNote.js`.

```
import { API } from "aws-amplify";
```

✦ CHANGE And replace our `handleSubmit` function with the following.

```
async function handleSubmit(event) {
  event.preventDefault();

  if (file.current && file.current.size >
config.MAX_ATTACHMENT_SIZE) {
    alert(
      `Please pick a file smaller than
${config.MAX_ATTACHMENT_SIZE /
        1000000} MB.`
    );
```

```
    return;
  }

  setIsLoading(true);

  try {
    await createNote({ content });
    props.history.push("/");
  } catch (e) {
    alert(e);
    setIsLoading(false);
  }
}

function createNote(note) {
  return API.post("notes", "/notes", {
    body: note
  });
}
```

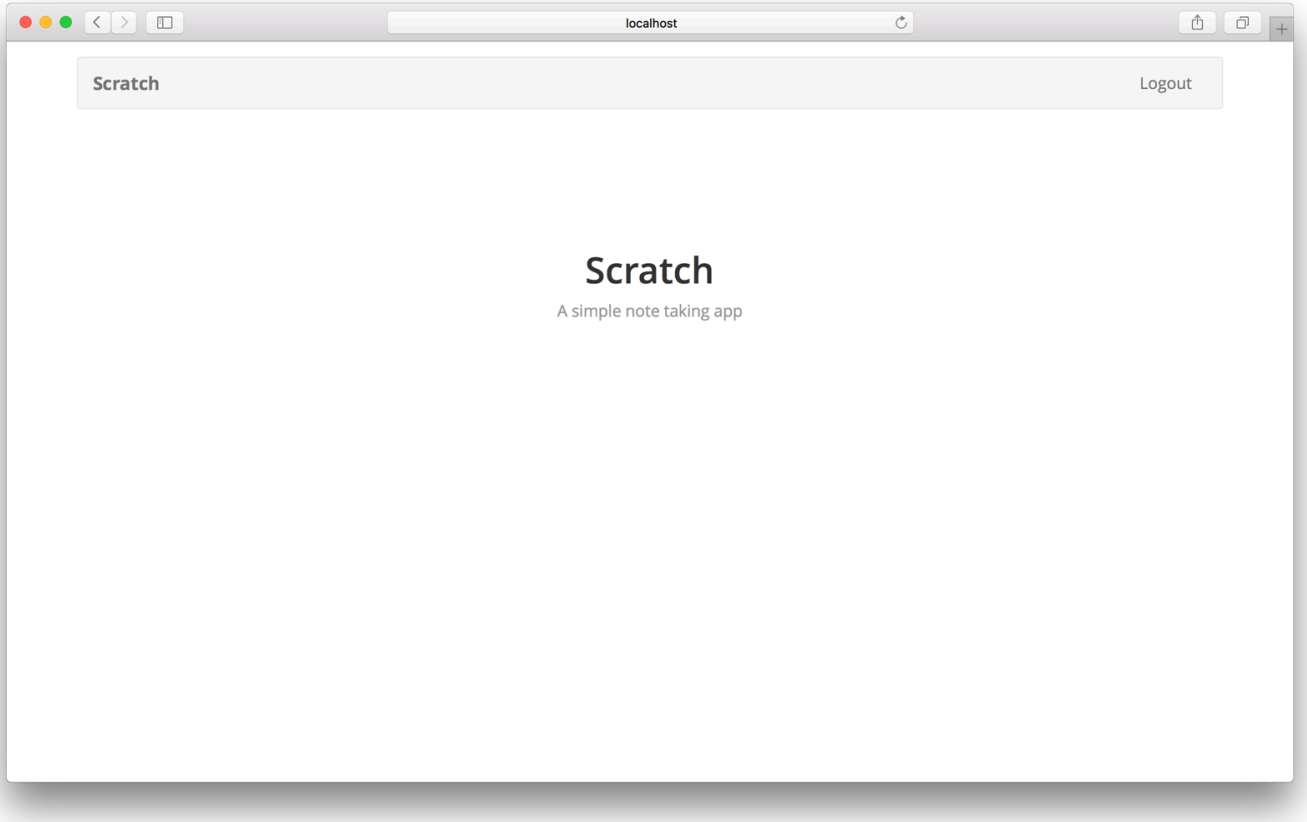
This does a couple of simple things.

1. We make our create call in `createNote` by making a POST request to `/notes` and passing in our note object. Notice that the first two arguments to the `API.post()` method are `notes` and `/notes`. This is because back in the Configure AWS Amplify (/chapters/configure-aws-amplify.html) chapter we called these set of APIs by the name `notes`.

2. For now the note object is simply the content of the note. We are creating these notes without an attachment for now.

3. Finally, after the note is created we redirect to our homepage.

And that's it; if you switch over to your browser and try submitting your form, it should successfully navigate over to our homepage.



Next let's upload our file to S3 and add an attachment to our note.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/call-the-create-api/124)

Upload a File to S3

Let's now add an attachment to our note. The flow we are using here is very simple.

1. The user selects a file to upload.
2. The file is uploaded to S3 under the user's folder and we get a key back.
3. Create a note with the file key as the attachment.

We are going to use the Storage module that AWS Amplify has. If you recall, that back in the Create a Cognito identity pool (/chapters/create-a-cognito-identity-pool.html) chapter we allow a logged in user access to a folder inside our S3 Bucket. AWS Amplify stores directly to this folder if we want to *privately* store a file.

Also, just looking ahead a bit; we will be uploading files when a note is created and when a note is edited. So let's create a simple convenience method to help with that.

Upload to S3

✦ CHANGE Add the following to `src/libs/awsLib.js`.

```
import { Storage } from "aws-amplify";

export async function s3Upload(file) {
  const filename = `${Date.now()}-${file.name}`;
```

```
  const stored = await Storage.vault.put(filename,
file, {
    contentType: file.type
  });

  return stored.key;
}
```

The above method does a couple of things.

1. It takes a file object as a parameter.
2. Generates a unique file name using the current timestamp (`Date.now()`). Of course, if your app is being used heavily this might not be the best way to create a unique filename. But this should be fine for now.
3. Upload the file to the user's folder in S3 using the `Storage.vault.put()` object. Alternatively, if we were uploading publicly you can use the `Storage.put()` method.
4. And return the stored object's key.

Upload Before Creating a Note

Now that we have our upload methods ready, let's call them from the create note method.

✦ CHANGE Replace the `handleSubmit` method in `src/containers/NewNote.js` with the following.

```
async function handleSubmit(event) {
  event.preventDefault();

  if (file.current && file.current.size >
config.MAX_ATTACHMENT_SIZE) {
    alert(
      `Please pick a file smaller than
${config.MAX_ATTACHMENT_SIZE /
        1000000} MB.`
    );
    return;
  }

  setIsLoading(true);

  try {
    const attachment = file.current
      ? await s3Upload(file.current)
      : null;

    await createNote({ content, attachment });
    props.history.push("/");
  } catch (e) {
    alert(e);
    setIsLoading(false);
  }
}
```

✦ CHANGE And make sure to include `s3Upload` by adding the following to the header of `src/containers/NewNote.js`.

```
import { s3Upload } from "../libs/awsLib";
```

The change we've made in the `handleSubmit` is that:

1. We upload the file using the `s3Upload` method.
2. Use the returned key and add that to the note object when we create the note.

Now when we switch over to our browser and submit the form with an uploaded file we should see the note being created successfully. And the app being redirected to the homepage.

Next up we are going to allow users to see a list of the notes they've created.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/comments-for-upload-a-file-to-s3/123)

List All the Notes

Now that we are able to create a new note, let's create a page where we can see a list of all the notes a user has created. It makes sense that this would be the homepage (even though we use the `/` route for the landing page). So we just need to conditionally render the landing page or the homepage depending on the user session.

Currently, our Home container is very simple. Let's add the conditional rendering in there.

✦ CHANGE Replace our `src/containers/Home.js` with the following.

```
import React, { useState, useEffect } from "react";
import { PageHeader, ListGroup } from "react-
bootstrap";
import "./Home.css";

export default function Home(props) {
  const [notes, setNotes] = useState([]);
  const [isLoading, setIsLoading] = useState(true);

  function renderNotesList(notes) {
    return null;
  }

  function renderLander() {
    return (
```

```
      <div className="lander">
        <h1>Scratch</h1>
        <p>A simple note taking app</p>
      </div>
    );
  }

  function renderNotes() {
    return (
      <div className="notes">
        <PageHeader>Your Notes</PageHeader>
        <ListGroup>
          {!isLoading && renderNotesList(notes)}
        </ListGroup>
      </div>
    );
  }

  return (
    <div className="Home">
      {props.isAuthenticated ? renderNotes() :
renderLander()}
    </div>
  );
}
```

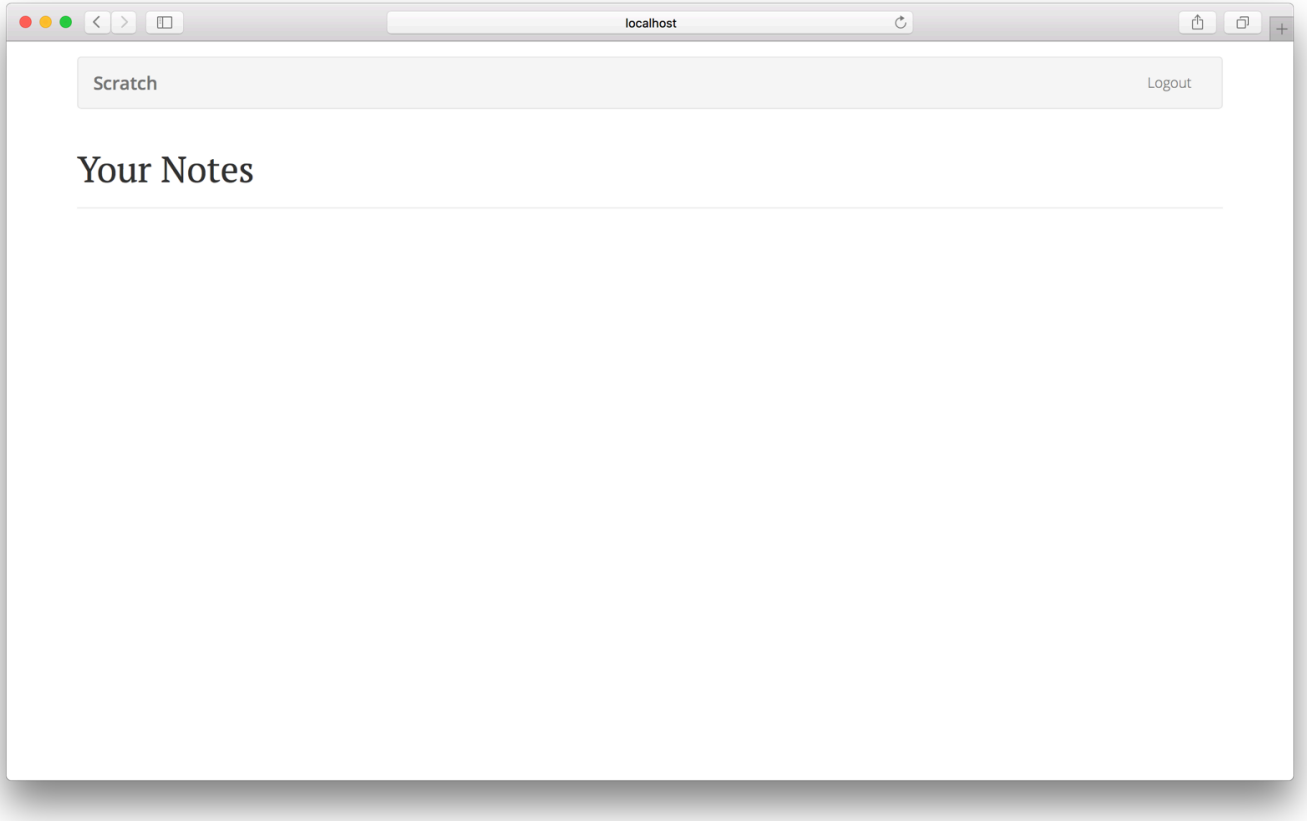
We are doing a few things of note here:

1. Rendering the lander or the list of notes based on `props.isAuthenticated`.
2. Store our notes in the state. Currently, it's empty but we'll be calling

our API for it.

3. Once we fetch our list we'll use the `renderNotesList` method to render the items in the list.

And that's our basic setup! Head over to the browser and the homepage of our app should render out an empty list.



Next we are going to fill it up with our API.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/list-all-the-notes/156)

Call the List API

Now that we have our basic homepage set up, let's make the API call to render our list of notes.

Make the Request

✦ CHANGE Add the following right below the state variable declarations in `src/containers/Home.js`.

```
useEffect(() => {
  async function onLoad() {
    if (!props.isAuthenticated) {
      return;
    }

    try {
      const notes = await loadNotes();
      setNotes(notes);
    } catch (e) {
      alert(e);
    }

    setIsLoading(false);
  }

  onLoad();
```

```
}, [props.isAuthenticated]);

function loadNotes() {
  return API.get("notes", "/notes");
}
```

We are using the useEffect React Hook (https://reactjs.org/docs/hooks-effect.html). We covered how this works back in the Load the State from the Session (/chapters/load-the-state-from-the-session.html) chapter.

Let's quickly go over how we are using it here. We want to make a request to our `/notes` API to get the list of notes when our component first loads. But only if the user is authenticated. Since our hook relies on `props.isAuthenticated`, we need to pass it in as the second argument in the `useEffect` call as an element in the array. This is basically telling React that we only want to run our Hook again when the `props.isAuthenticated` value changes.

✦ CHANGE And include our Amplify API module in the header.

```
import { API } from "aws-amplify";
```

Now let's render the results.

Render the List

✦ CHANGE Replace our `renderNotesList` placeholder method with the following.

```
function renderNotesList(notes) {
  return [{}].concat(notes).map((note, i) =>
```

```
    i !== 0 ? (
      <LinkContainer key={note.noteId} to=
{`/notes/${note.noteId}`}>
        <ListGroupItem header=
{note.content.trim().split("\n")[0]}>
          {"Created: " + new
Date(note.createdAt).toLocaleString()}
        </ListGroupItem>
      </LinkContainer>
    ) : (
      <LinkContainer key="new" to="/notes/new">
        <ListGroupItem>
          <h4>
            <b>{"\uFF0B"}</b> Create a new note
          </h4>
        </ListGroupItem>
      </LinkContainer>
    )
  );
}
```

✦ CHANGE And include the `ListGroupItem` in the header so that our `react-bootstrap` import looks like so.

```
import { PageHeader, ListGroup, ListGroupItem } from
"react-bootstrap";
```

✦ CHANGE Also include the `LinkContainer` from `react-router-bootstrap`.

```
import { LinkContainer } from "react-router-bootstrap";
```

The code above does a few things.

1. It always renders a **Create a new note** button as the first item in the list (even if the list is empty). We do this by concatenating an array with an empty object with our `notes` array.

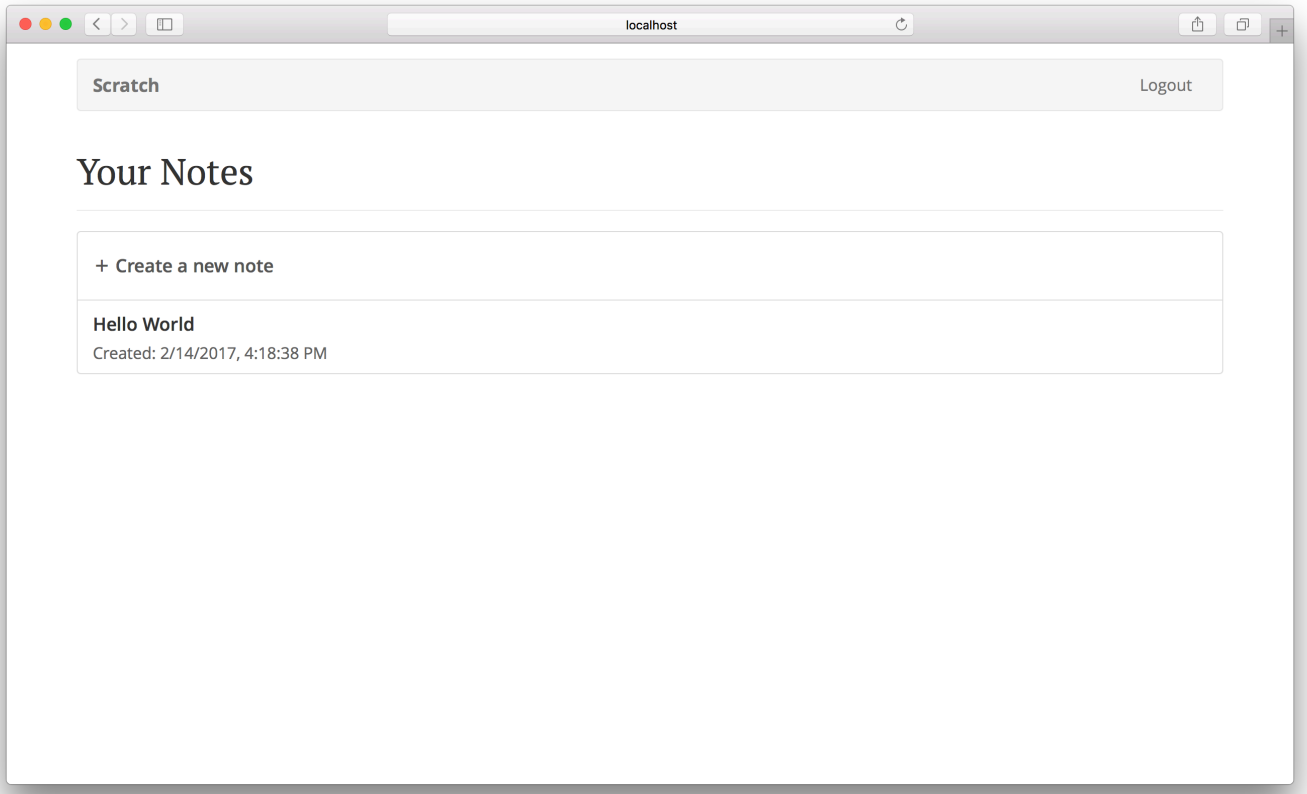
2. We render the first line of each note as the `ListGroupItem` header by doing `note.content.trim().split('\n')[0]`.

3. And the `LinkContainer` component directs our app to each of the items.

✦ CHANGE Let's also add a couple of styles to our `src/containers/Home.css`.

```
.Home .notes h4 {
  font-family: "Open Sans", sans-serif;
  font-weight: 600;
  overflow: hidden;
  line-height: 1.5;
  white-space: nowrap;
  text-overflow: ellipsis;
}
.Home .notes p {
  color: #666;
}
```

Now head over to your browser and you should see your list displayed.



If you click on each entry, the links should generate URLs with appropriate *noteIds*. For now, these URLs will take you to our 404 page. We'll fix that in the next section.

Next up we are going to allow users to view and edit their notes.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/call-the-list-api/127)

Display a Note

Now that we have a listing of all the notes, let's create a page that displays a note and lets the user edit it.

The first thing we are going to need to do is load the note when our container loads. Just like what we did in the `Home` container. So let's get started.

Add the Route

Let's add a route for the note page that we are going to create.

✦ CHANGE Add the following line to `src/Routes.js` **below** our `/notes/new` route. We are using the `AppliedRoute` component that we created in the Add the session to the state (/chapters/add-the-session-to-the-state.html) chapter.

```
<AppliedRoute path="/notes/:id" exact component={Notes}
appProps={appProps} />
```

This is important because we are going to be pattern matching to extract our note id from the URL.

By using the route path `/notes/:id` we are telling the router to send all matching routes to our component `Notes`. This will also end up matching the route `/notes/new` with an `id` of `new`. To ensure that doesn't

happen, we put our `/notes/new` route before the pattern matching one.

✦ CHANGE And include our component in the header.

```
import Notes from "./containers/Notes";
```

Of course this component doesn't exist yet and we are going to create it now.

Add the Container

✦ CHANGE Create a new file `src/containers/Notes.js` and add the following.

```
import React, { useRef, useState, useEffect } from
"react";
import { API, Storage } from "aws-amplify";

export default function Notes(props) {
  const file = useRef(null);
  const [note, setNote] = useState(null);
  const [content, setContent] = useState("");

  useEffect(() => {
    function loadNote() {
      return API.get("notes",
`/notes/${props.match.params.id}`);
    }

    async function onLoad() {
      try {
```

```
        const note = await loadNote();
        const { content, attachment } = note;

        if (attachment) {
          note.attachmentURL = await
Storage.vault.get(attachment);
        }

        setContent(content);
        setNote(note);
      } catch (e) {
        alert(e);
      }
    }

    onLoad();
  }, [props.match.params.id]);

  return (
    <div className="Notes"></div>
  );
}
```

We are doing a couple of things here.

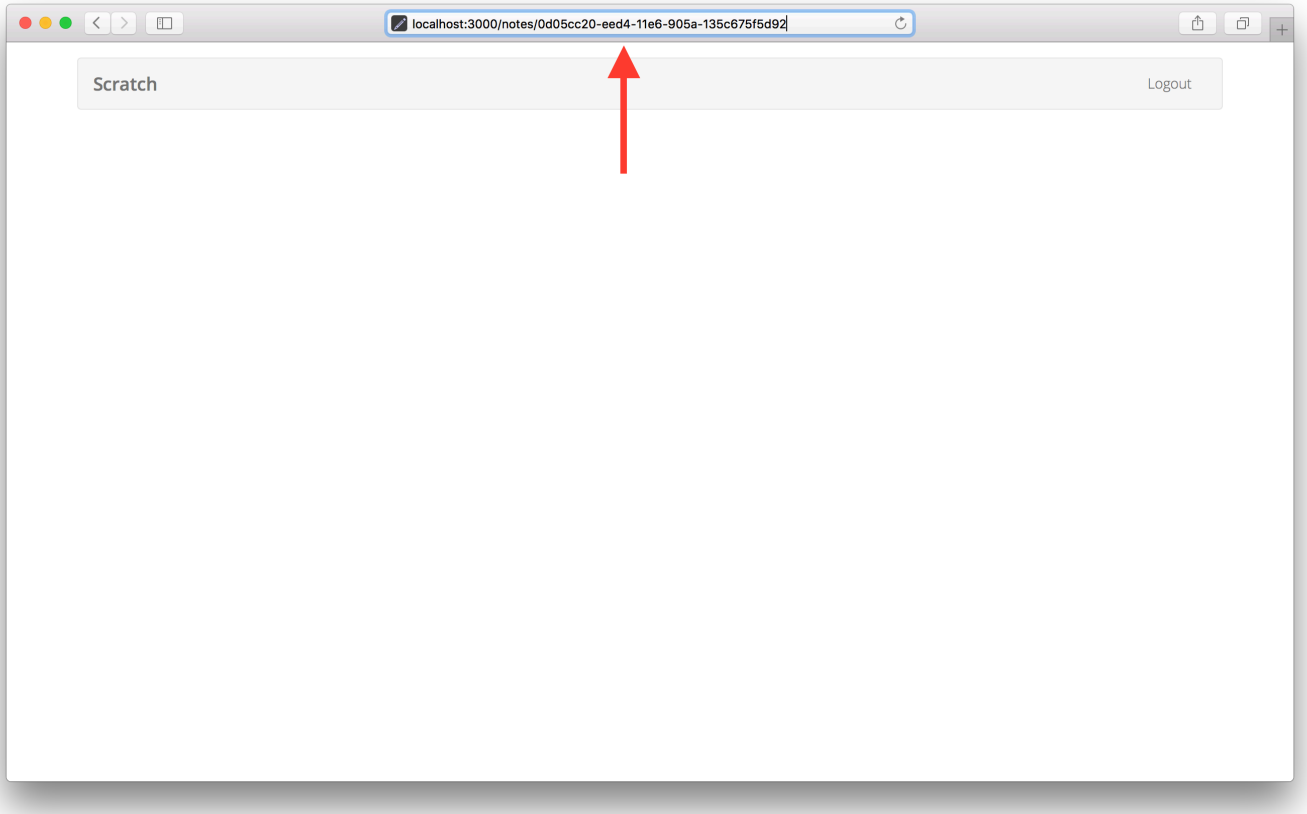
1. We are using the `useEffect` Hook to load the note when our component first loads. We then save it to the state. We get the `id` of our note from the URL using the props automatically passed to us by React-Router in `props.match.params.id`. The keyword `id` is a part of the pattern matching in our route (`/notes/:id`).

2. If there is an attachment, we use the key to get a secure link to the file

we uploaded to S3. We then store this in the new note object as `note.attachmentURL`.

3. The reason why we have the `note` object in the state along with the `content` and the `attachmentURL` is because we will be using this later when the user edits the note.

Now if you switch over to your browser and navigate to a note that we previously created, you'll notice that the page renders an empty container.



Next up, we are going to render the note we just loaded.

For help and discussion

Comments on this chapter (https://discourse.serverless-

```
stack.com/t/display-a-note/112)
```

Render the Note Form

Now that our container loads a note on `componentDidMount`, let's go ahead and render the form that we'll use to edit it.

✦ CHANGE Replace our placeholder `return` statement in `src/containers/Notes.js` with the following.

```
function validateForm() {
  return content.length > 0;
}

function formatFilename(str) {
  return str.replace(/^\w+-/, "");
}

function handleFileChange(event) {
  file.current = event.target.files[0];
}

async function handleSubmit(event) {
  let attachment;

  event.preventDefault();

  if (file.current && file.current.size >
config.MAX_ATTACHMENT_SIZE) {
```

```
    alert(
      `Please pick a file smaller than
${config.MAX_ATTACHMENT_SIZE /
        1000000} MB.`
    );
    return;
  }

  setIsLoading(true);
}

async function handleDelete(event) {
  event.preventDefault();

  const confirmed = window.confirm(
    "Are you sure you want to delete this note?"
  );

  if (!confirmed) {
    return;
  }

  setIsDeleting(true);
}

return (
  <div className="Notes">
    {note && (
      <form onSubmit={handleSubmit}>
        <FormGroup controlId="content">
          <FormControl
```

```
            value={content}
            componentClass="textarea"
            onChange={e => setContent(e.target.value)}
          />
        </FormGroup>
        {note.attachment && (
          <FormGroup>
            <ControlLabel>Attachment</ControlLabel>
            <FormControl.Static>
              <a
                target="_blank"
                rel="noopener noreferrer"
                href={note.attachmentURL}
              >
                {formatFilename(note.attachment)}
              </a>
            </FormControl.Static>
          </FormGroup>
        )}
        <FormGroup controlId="file">
          {!note.attachment &&
<ControlLabel>Attachment</ControlLabel>}
          <FormControl onChange={handleFileChange}
type="file" />
        </FormGroup>
        <LoaderButton
          block
          type="submit"
          bsSize="large"
          bsStyle="primary"
          isLoading={isLoading}
```

```
          disabled={!validateForm()}
        >
          Save
        </LoaderButton>
        <LoaderButton
          block
          bsSize="large"
          bsStyle="danger"
          onClick={handleDelete}
          isLoading={isDeleting}
        >
          Delete
        </LoaderButton>
      </form>
    )}
  </div>
);
```

We are doing a few things here:

1. We render our form only when the `note` state variable is set.
2. Inside the form we conditionally render the part where we display the attachment by using `note.attachment`.
3. We format the attachment URL using `formatFilename` by stripping the timestamp we had added to the filename while uploading it.
4. We also added a delete button to allow users to delete the note. And just like the submit button it too needs a flag that signals that the call is in progress. We call it `isDeleting`.
5. We handle attachments with a file input exactly like we did in the

`NewNote` component.

6. Our delete button also confirms with the user if they want to delete the note using the browser's `confirm` dialog.

To complete this code, let's add `isLoading` and `isDeleting` to the state.

✦ CHANGE So our new state and ref declarations at the top of our `Notes` component function look like this.

```
const file = useRef(null);
const [note, setNote] = useState(null);
const [content, setContent] = useState("");
const [isLoading, setIsLoading] = useState(false);
const [isDeleting, setIsDeleting] = useState(false);
```

✦ CHANGE Let's also add some styles by adding the following to `src/containers/Notes.css`.

```
.Notes form {
  padding-bottom: 15px;
}

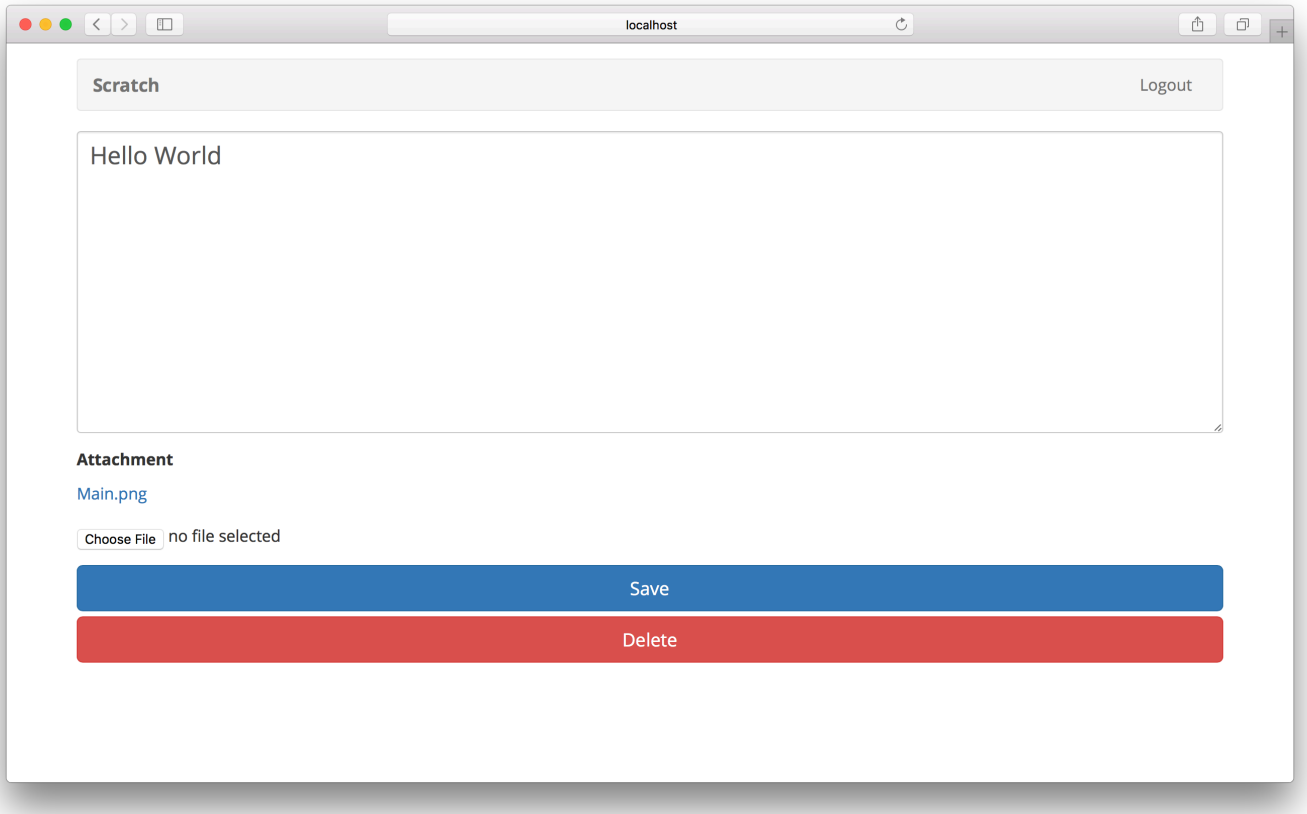
.Notes form textarea {
  height: 300px;
  font-size: 24px;
}
```

✦ CHANGE Also, let's include the React-Bootstrap components that we are using here by adding the following to our header. And our styles, the

`LoaderButton`, and the `config`.

```
import { FormGroup, FormControl, ControlLabel } from
"react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import config from "../config";
import "./Notes.css";
```

And that's it. If you switch over to your browser, you should see the note loaded.



Next, we'll look at saving the changes we make to our note.

For help and discussion

Comments on this chapter

```
(https://discourse.serverless-
stack.com/t/render-the-note-form/140)
```

Save Changes to a Note

Now that our note loads into our form, let's work on saving the changes we make to that note.

✦ CHANGE Replace the `handleSubmit` function in `src/containers/Notes.js` with the following.

```
function saveNote(note) {
  return API.put("notes",
`/notes/${props.match.params.id}`, {
    body: note
  });
}

async function handleSubmit(event) {
  let attachment;

  event.preventDefault();

  if (file.current && file.current.size >
config.MAX_ATTACHMENT_SIZE) {
    alert(
      `Please pick a file smaller than
${config.MAX_ATTACHMENT_SIZE /
        1000000} MB.`
    );
```

```
    return;
  }

  setIsLoading(true);

  try {
    if (file.current) {
      attachment = await s3Upload(file.current);
    }

    await saveNote({
      content,
      attachment: attachment || note.attachment
    });
    props.history.push("/");
  } catch (e) {
    alert(e);
    setIsLoading(false);
  }
}
```

✦ CHANGE And include our `s3Upload` helper method in the header:

```
import { s3Upload } from "../libs/awsLib";
```

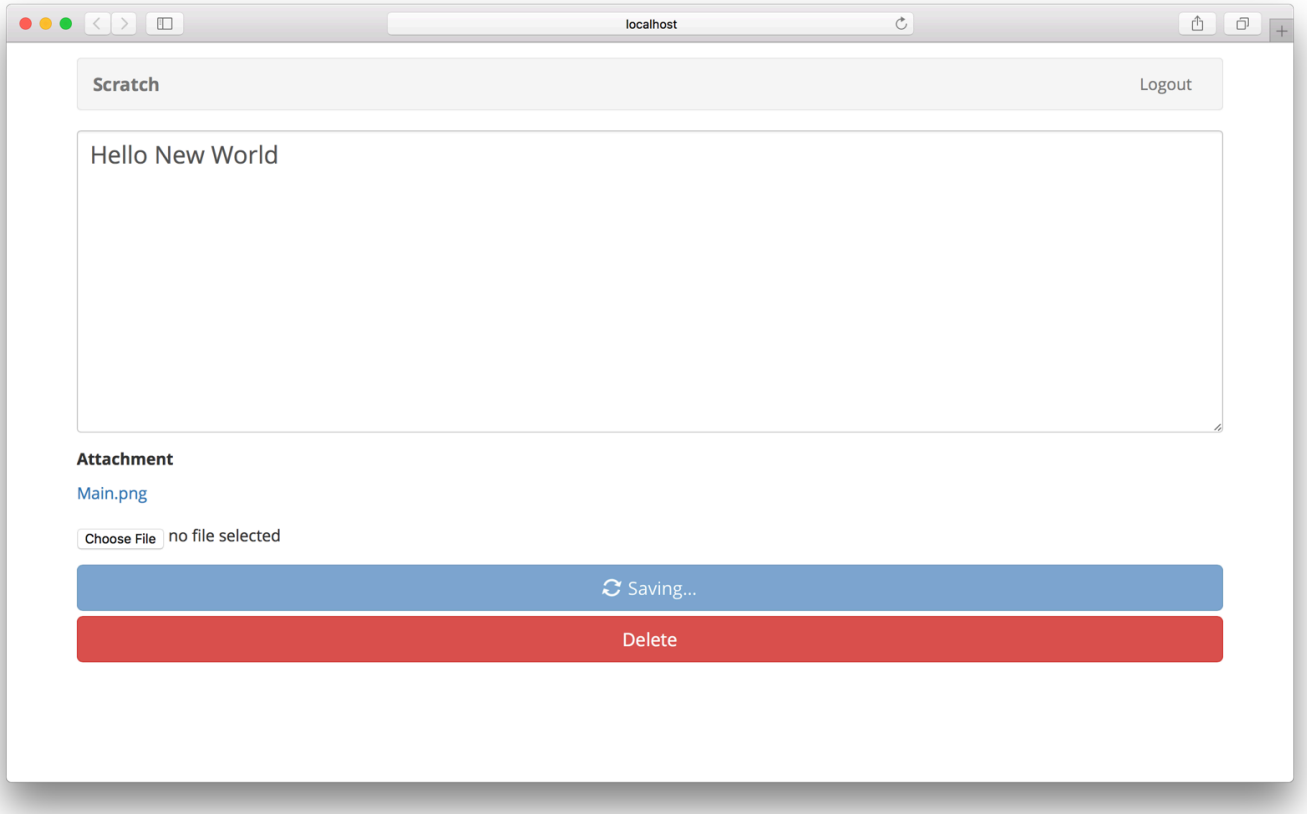
The code above is doing a couple of things that should be very similar to what we did in the `NewNote` container.

1. If there is a file to upload we call `s3Upload` to upload it and save the key we get from S3. If there isn't then we simply save the existing attachment object, `note.attachment`.

2. We save the note by making a `PUT` request with the note object to `/notes/:id` where we get the `id` from `props.match.params.id`. We use the `API.put()` method from AWS Amplify.

3. And on success we redirect the user to the homepage.

Let's switch over to our browser and give it a try by saving some changes.



You might have noticed that we are not deleting the old attachment when we upload a new one. To keep things simple, we are leaving that bit of detail up to you. It should be pretty straightforward. Check the AWS Amplify API Docs (https://aws.github.io/aws-amplify/api/classes/storageclass.html#remove) on how to a delete file from S3.

Next up, let's allow users to delete their note.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/save-changes-to-a-note/131)

Delete a Note

The last thing we need to do on the note page is allowing users to delete their note. We have the button all set up already. All that needs to be done is to hook it up with the API.

✦ CHANGE Replace our `handleDelete` function in `src/containers/Notes.js`.

```
function deleteNote() {
  return API.del("notes",
`/notes/${props.match.params.id}`);
}

async function handleDelete(event) {
  event.preventDefault();

  const confirmed = window.confirm(
    "Are you sure you want to delete this note?"
  );

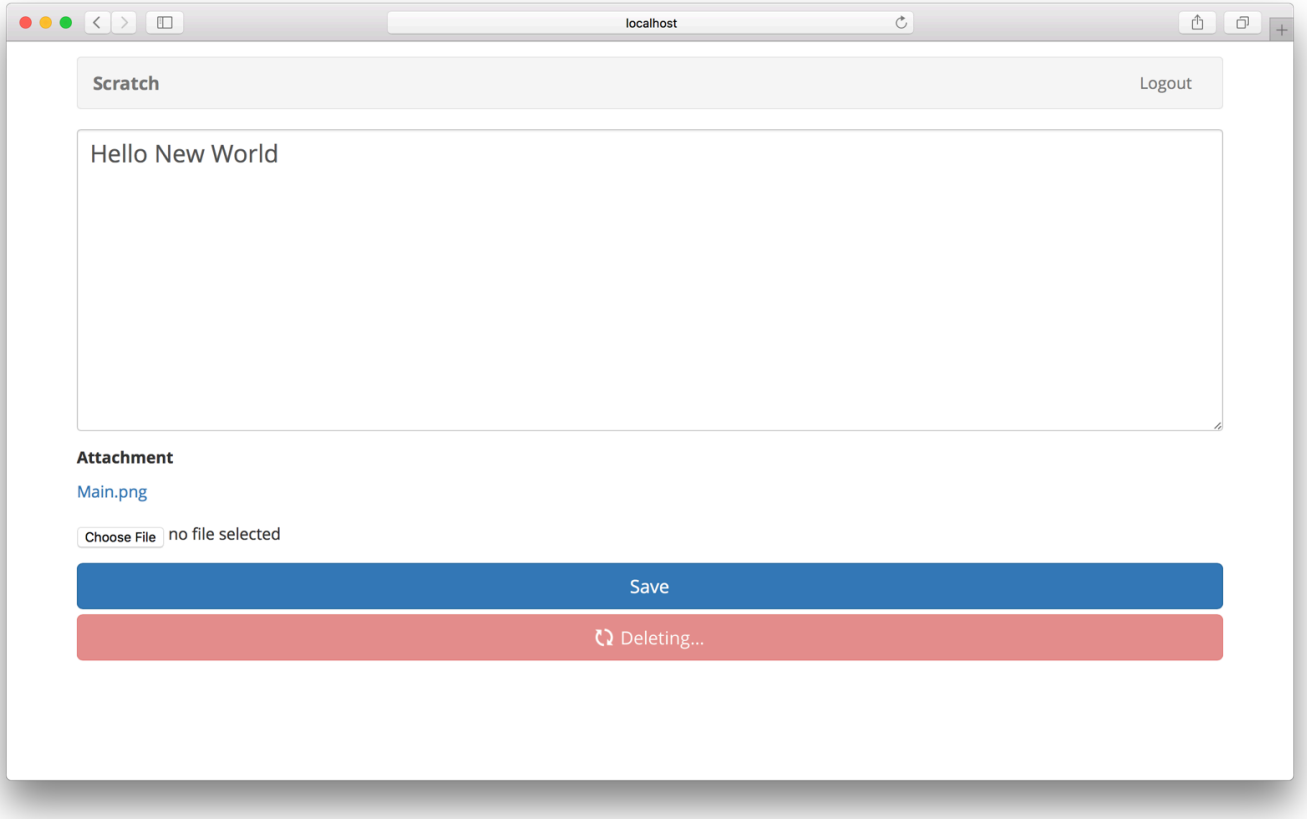
  if (!confirmed) {
    return;
  }

  setIsDeleting(true);
```

```
  try {
    await deleteNote();
    props.history.push("/");
  } catch (e) {
    alert(e);
    setIsDeleting(false);
  }
}
```

We are simply making a `DELETE` request to `/notes/:id` where we get the `id` from `props.match.params.id`. We use the `API.del` method from AWS Amplify to do so. This calls our delete API and we redirect to the homepage on success.

Now if you switch over to your browser and try deleting a note you should see it confirm your action and then delete the note.



Again, you might have noticed that we are not deleting the attachment when we are deleting a note. We are leaving that up to you to keep things simple. Check the AWS Amplify API Docs (https://aws.github.io/aws-amplify/api/classes/storageclass.html#remove) on how to a delete file from S3.

Next, let's add a settings page to our app. This is where a user will be able to pay for our service!

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/comments-for-delete-a-note/137)

Create a Settings Page

We are going to add a settings page to our app. This is going to allow users to pay for our service. The flow will look something like this:

1. Users put in their credit card info and the number of notes they want to store.
2. We call Stripe on the frontend to generate a token for the credit card.
3. We then call our billing API with the token and the number of notes.
4. Our billing API calculates the amount and bills the card!

To get started let's add our settings page.

✦ CHANGE Create a new file in `src/containers/Settings.js` and add the following.

```
import React, { useState } from "react";
import { API } from "aws-amplify";

export default function Settings(props) {
  const [isLoading, setIsLoading] = useState(false);

  function billUser(details) {
    return API.post("notes", "/billing", {
      body: details
    });
  }
```

```
  return (
    <div className="Settings">
    </div>
  );
}
```

✦ CHANGE Next import this component in the header of `src/Routes.js`.

```
import Settings from "./containers/Settings";
```

✦ CHANGE Add the following below the `/signup` route in our `<Switch>` block in `src/Routes.js`.

```
<AppliedRoute path="/settings" exact component=
{Settings} appProps={appProps} />
```

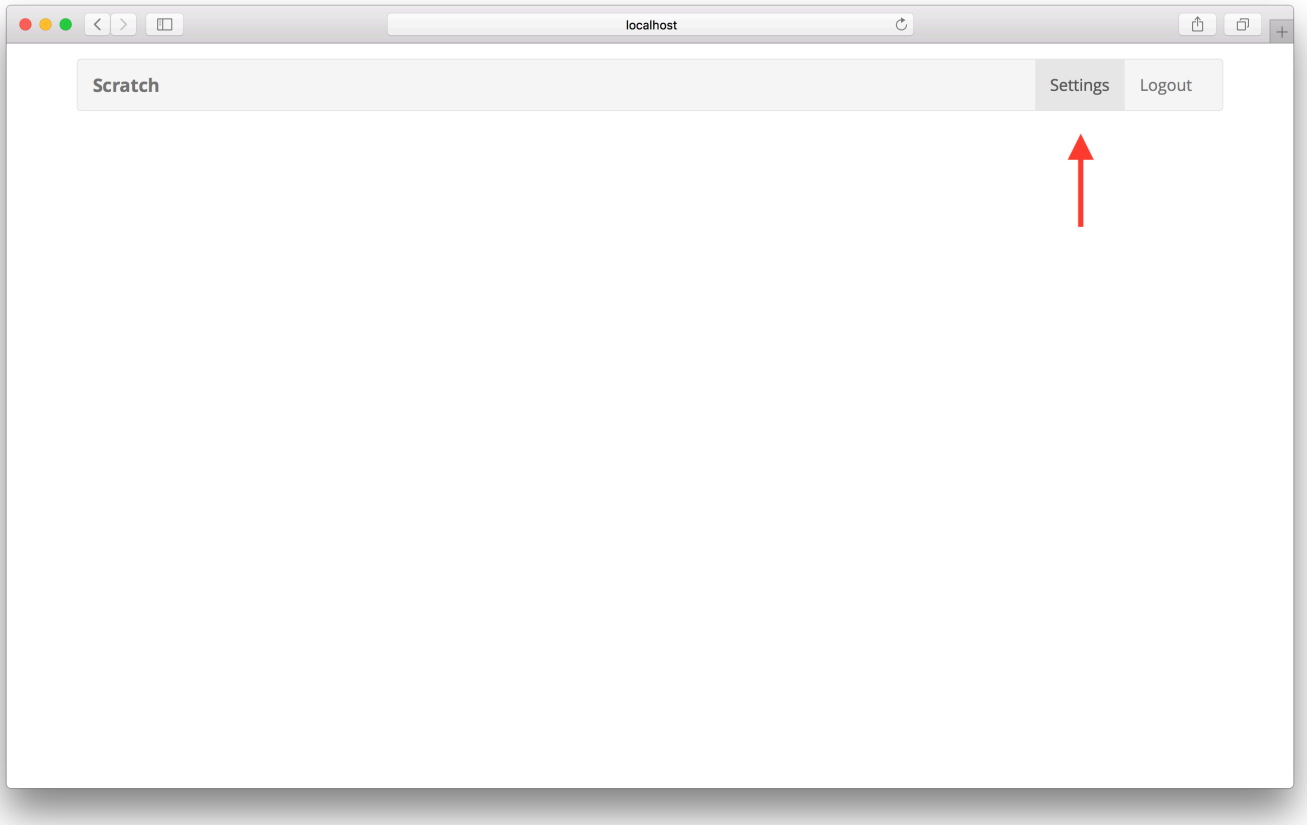
✦ CHANGE Next add a link to our settings page in the navbar by replacing the `return` statement in `src/App.js` with this.

```
return (
  !isAuthenticating && (
    <div className="App container">
      <Navbar fluid collapseOnSelect>
        <Navbar.Header>
          <Navbar.Brand>
            <Link to="/">Scratch</Link>
          </Navbar.Brand>
          <Navbar.Toggle />
        </Navbar.Header>
        <Navbar.Collapse>
          <Nav pullRight>
```

```
            {isAuthenticated ? (
              <>
                <LinkContainer to="/settings">
                  <NavItem>Settings</NavItem>
                </LinkContainer>
                <NavItem onClick=
{handleLogout}>Logout</NavItem>
              </>
            ) : (
              <>
                <LinkContainer to="/signup">
                  <NavItem>Signup</NavItem>
                </LinkContainer>
                <LinkContainer to="/login">
                  <NavItem>Login</NavItem>
                </LinkContainer>
              </>
            )}
          </Nav>
        </Navbar.Collapse>
      </Navbar>
      <Routes appProps={{ isAuthenticated,
userHasAuthenticated }} />
    </div>
  )
);
```

You'll notice that we added another link in the navbar that only displays when a user is logged in.

Now if you head over to your app, you'll see a new **Settings** link at the top. Of course, the page is pretty empty right now.



Next, we'll add our Stripe SDK keys to our config.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-settings-page/184)

Add Stripe Keys to Config

Back in the Setup a Stripe account (/chapters/setup-a-stripe-account.html) chapter, we had two keys in the Stripe console. The **Secret key** that we used in the backend and the **Publishable key**. The **Publishable key** is meant to be used in the frontend.

✦ CHANGE Add the following line to the `export` block of `src/config.js`.

```
STRIPE_KEY: "YOUR_STRIPE_PUBLIC_KEY",
```

Make sure to replace, `YOUR_STRIPE_PUBLIC_KEY` with the **Publishable key** from the Setup a Stripe account (/chapters/setup-a-stripe-account.html) chapter.

Next, we'll build our billing form.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/add-stripe-keys-to-config/185)

Create a Billing Form

Now our settings page is going to have a form that will take a user's credit card details, get a stripe token and call our billing API with it. Let's start by adding the Stripe React SDK to our project.

CHANGE From our project root, run the following.

```
$ npm install --save react-stripe-elements
```

Next let's create our billing form component.

CHANGE Add the following to a new file in `src/components/BillingForm.js`.

```
import React, { useState } from "react";
import { FormGroup, FormControl, ControlLabel } from
"react-bootstrap";
import { CardElement, injectStripe } from "react-
stripe-elements";
import LoaderButton from "./LoaderButton";
import { useFormFields } from "../libs/hooksLib";
import "./BillingForm.css";

function BillingForm({ isLoading, onSubmit, ...props })
{
  const [fields, handleFieldChange] = useFormFields({
    name: "",
```

```
    storage: ""
  });
  const [isProcessing, setIsProcessing] =
useState(false);
  const [isCardComplete, setIsCardComplete] =
useState(false);

  isLoading = isProcessing || isLoading;

  function validateForm() {
    return (
      fields.name !== "" &&
      fields.storage !== "" &&
      isCardComplete
    );
  }

  async function handleSubmitClick(event) {
    event.preventDefault();

    setIsProcessing(true);

    const { token, error } = await
props.stripe.createToken({ name: fields.name });

    setIsProcessing(false);

    onSubmit(fields.storage, { token, error });
  }

  return (
```

```
    <form className="BillingForm" onSubmit=
{handleSubmitClick}>
      <FormGroup bsSize="large" controlId="storage">
        <ControlLabel>Storage</ControlLabel>
        <FormControl
          min="0"
          type="number"
          value={fields.storage}
          onChange={handleFieldChange}
          placeholder="Number of notes to store"
        />
      </FormGroup>
      <hr />
      <FormGroup bsSize="large" controlId="name">
        <ControlLabel>Cardholder&amp;apos;s
name</ControlLabel>
        <FormControl
          type="text"
          value={fields.name}
          onChange={handleFieldChange}
          placeholder="Name on the card"
        />
      </FormGroup>
      <ControlLabel>Credit Card Info</ControlLabel>
      <CardElement
        className="card-field"
        onChange={e => setIsCardComplete(e.complete)}
        style={{
          base: { fontSize: "18px", fontFamily: '"Open
Sans", sans-serif' }
        }}
```

```
        />
        <LoaderButton
          block
          type="submit"
          bsSize="large"
          isLoading={isLoading}
          disabled={!validateForm()}
        >
          Purchase
        </LoaderButton>
      </form>
    );
}

export default injectStripe(BillingForm);
```

Let's quickly go over what we are doing here:

- To begin with we are going to wrap our component with a Stripe module using the `injectStripe` HOC. This gives our component access to the `props.stripe.createToken` method.
- As for the fields in our form, we have input field of type `number` that allows a user to enter the number of notes they want to store. We also take the name on the credit card. These are stored in the state through the `handleFieldChange` method that we get from our `useFormFields` custom React Hook.
- The credit card number form is provided by the Stripe React SDK through the `CardElement` component that we import in the header.
- The submit button has a loading state that is set to true when we call

Stripe to get a token and when we call our billing API. However, since our Settings container is calling the billing API we use the `props.isLoading` to set the state of the button from the Settings container.

- We also validate this form by checking if the name, the number of notes, and the card details are complete. For the card details, we use the CardElement's `onChange` method.

- Finally, once the user completes and submits the form we make a call to Stripe by passing in the credit card name and the credit card details (this is handled by the Stripe SDK). We call the `props.stripe.createToken` method and in return we get the token or an error back. We simply pass this and the number of notes to be stored to the settings page via the `onSubmit` method. We will be setting this up shortly.

You can read more about how to use the React Stripe Elements here (https://github.com/stripe/react-stripe-elements).

Also, let's add some styles to the card field so it matches the rest of our UI.

✦ CHANGE Create a file at `src/components/BillingForm.css`.

```
.BillingForm .card-field {
  margin-bottom: 15px;
  background-color: white;
  padding: 11px 16px;
  border-radius: 6px;
  border: 1px solid #CCC;
  box-shadow: inset 0 1px 1px rgba(0, 0, 0, .075);
  line-height: 1.3333333;
```

```
}

.BillingForm .card-field.StripeElement--focus {
  box-shadow: inset 0 1px 1px rgba(0, 0, 0, .075), 0 0
8px rgba(102, 175, 233, .6);
  border-color: #66AFE9;
}
```

Next we'll plug our form into the settings page.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-billing-form/186)

Connect the Billing Form

Now all we have left to do is to connect our billing form to our billing API.

Let's start by including Stripe.js in our HTML.

✦ CHANGE Append the following to the `<head>` block in our `public/index.html`.

```
<script src="https://js.stripe.com/v3/"></script>
```

✦ CHANGE Replace our `return` statement in `src/containers/Settings.js` with this.

```
async function handleFormSubmit(storage, { token, error
}) {
  if (error) {
    alert(error);
    return;
  }

  setIsLoading(true);

  try {
    await billUser({
      storage,
      source: token.id
    });
```

```
    alert("Your card has been charged successfully!");
    props.history.push("/");
  } catch (e) {
    alert(e);
    setIsLoading(false);
  }
}

return (
  <div className="Settings">
    <StripeProvider apiKey={config.STRIPE_KEY}>
      <Elements>
        <BillingForm
          isLoading={isLoading}
          onSubmit={handleFormSubmit}
        />
      </Elements>
    </StripeProvider>
  </div>
);
```

CHANGE And add the following to the header.

```
import { Elements, StripeProvider } from "react-stripe-elements";
import BillingForm from "../components/BillingForm";
import config from "../config";
import "./Settings.css";
```

We are adding the `BillingForm` component that we previously created

here and passing in the `isLoading` and `onSubmit` prop that we referenced in the previous chapter. In the `handleFormSubmit` method, we are checking if the Stripe method returned an error. And if things looked okay then we call our billing API and redirect to the home page after letting the user know.

An important detail here is about the `StripeProvider` and the `Elements` component that we are using. The `StripeProvider` component let's the Stripe SDK know that we want to call the Stripe methods using `config.STRIPE_KEY`. And it needs to wrap around at the top level of our billing form. Similarly, the `Elements` component needs to wrap around any component that is going to be using the `CardElement` Stripe component.

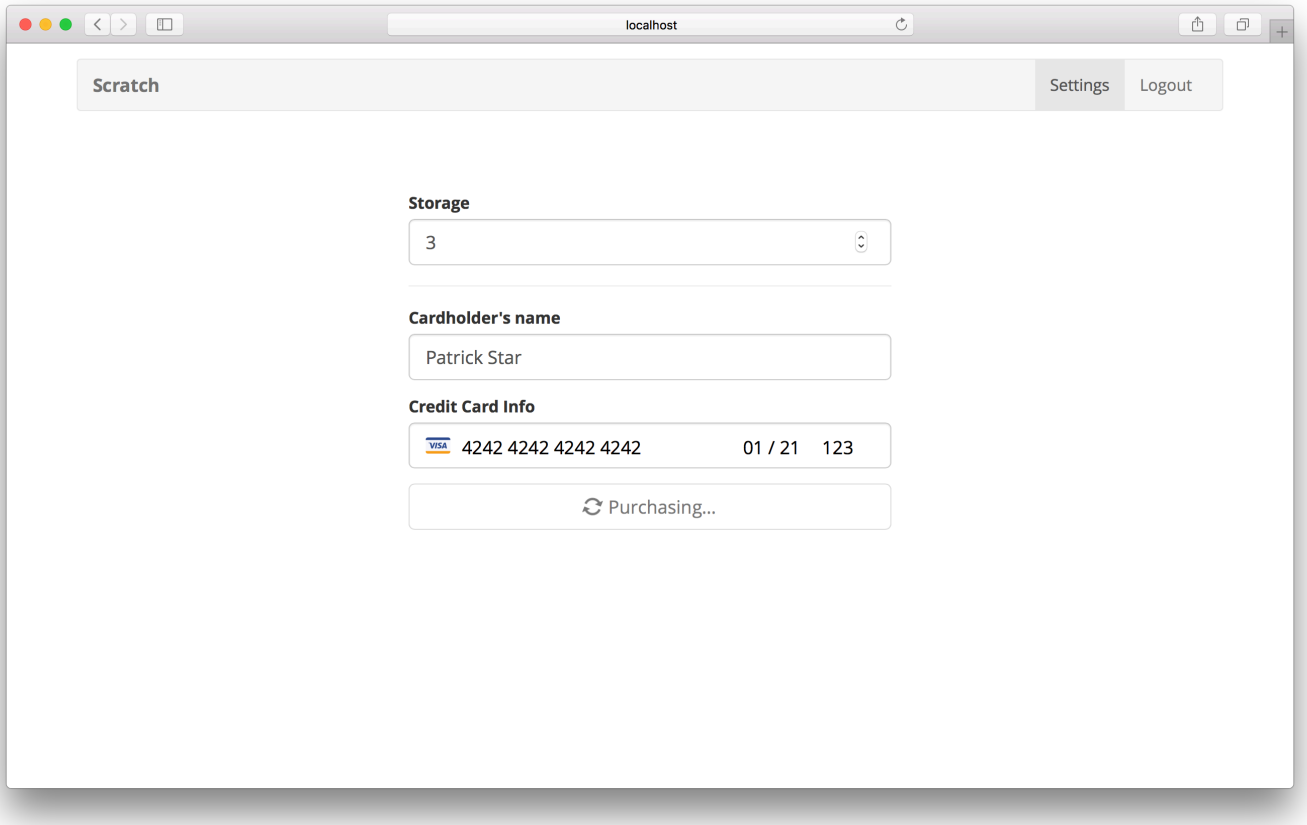
Finally, let's handle some styles for our settings page as a whole.

✦ CHANGE Create a file named `src/containers/Settings.css` and add the following.

```
@media all and (min-width: 480px) {
  .Settings {
    padding: 60px 0;
  }

  .Settings form {
    margin: 0 auto;
    max-width: 480px;
  }
}
```

This ensures that our form displays properly for larger screens.

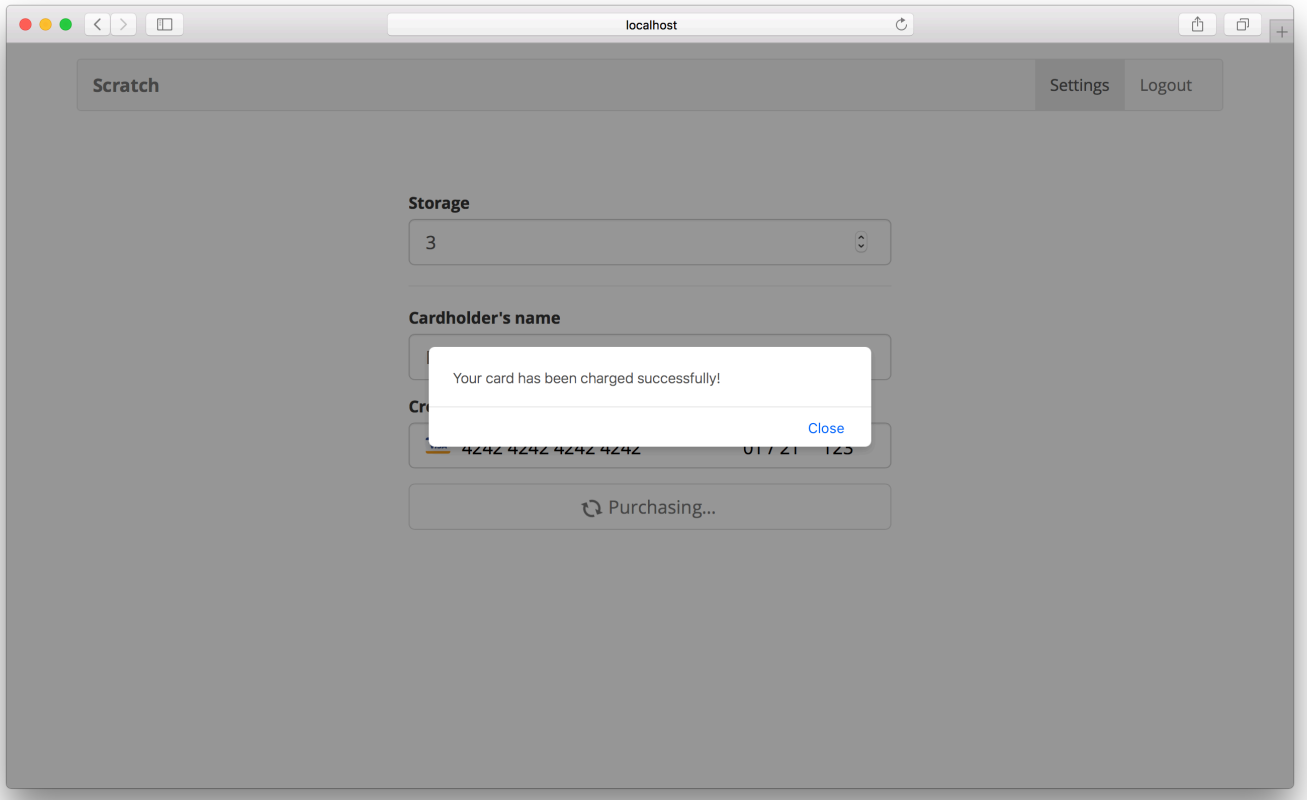


And that's it. We are ready to test our Stripe form. Head over to your browser and try picking the number of notes you want to store and use the following for your card details:

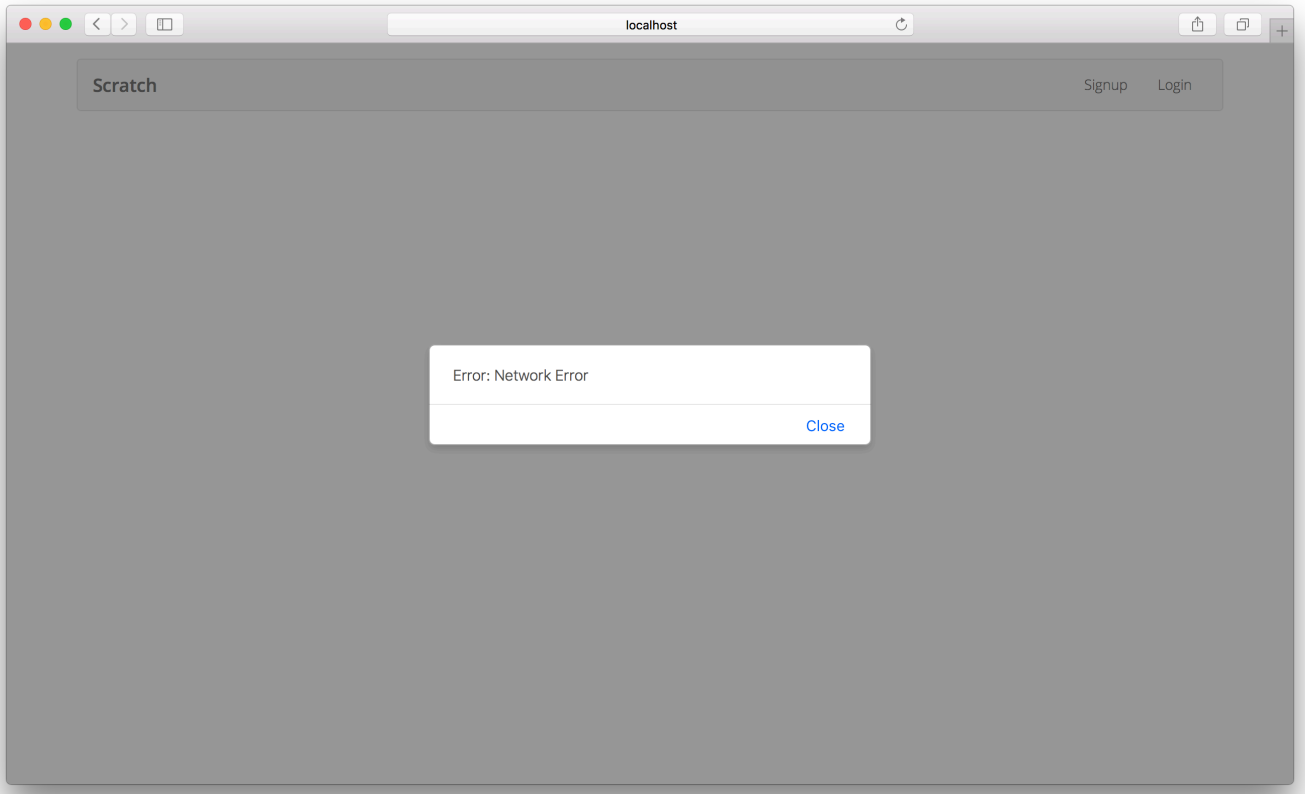
- A Stripe test card number is `4242 4242 4242 4242`.
- You can use any valid expiry date, security code, and zip code.
- And set any name.

You can read more about the Stripe test cards in the Stripe API Docs here (https://stripe.com/docs/testing#cards).

If everything is set correctly, you should see the success message and you'll be redirected to the homepage.



Now with our app nearly complete, we'll look at securing some the pages of our app that require a login. Currently if you visit a note page while you are logged out, it throws an ugly error.



Instead, we would like it to redirect us to the login page and then redirect us back after we login. Let's look at how to do that next.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/connect-the-billing-form/187)

Set up Secure Pages

We are almost done putting together our app. All the pages are done but there are a few pages that should not be accessible if a user is not logged in. For example, a page with the note should not load if a user is not logged in. Currently, we get an error when we do this. This is because the page loads and since there is no user in the session, the call to our API fails.

We also have a couple of pages that need to behave in sort of the same way. We want the user to be redirected to the homepage if they type in the login (`/login`) or signup (`/signup`) URL. Currently, the login and sign up page end up loading even though the user is already logged in.

There are many ways to solve the above problems. The simplest would be to just check the conditions in our containers and redirect. But since we have a few containers that need the same logic we can create a special route (like the `AppliedRoute` from the Add the session to the state (/chapters/add-the-session-to-the-state.html) chapter) for it.

We are going to create two different route components to fix the problem we have.

1. A route called the AuthenticatedRoute, that checks if the user is authenticated before routing.
2. And a component called the UnauthenticatedRoute, that ensures the user is not authenticated.

Let's create these components next.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/set-up-secure-pages/42)

Create a Route That Redirects

Let's first create a route that will check if the user is logged in before routing.

✦ CHANGE Add the following to `src/components/AuthenticatedRoute.js`.

```
import React from "react";
import { Route, Redirect } from "react-router-dom";

export default function AuthenticatedRoute({ component:
C, appProps, ...rest }) {
  return (
    <Route
      {...rest}
      render={props =>
        appProps.isAuthenticated
          ? <C {...props} {...appProps} />
          : <Redirect
              to={`/login?
redirect=${props.location.pathname}${props.location
                .search}`}
            />}
    />
```

```
  );
}
```

This component is similar to the `AppliedRoute` component that we created in the Add the session to the state (/chapters/add-the-session-to-the-state.html) chapter. The main difference being that we look at the props that are passed in to check if a user is authenticated. If the user is authenticated, then we simply render the passed in component. And if the user is not authenticated, then we use the `Redirect` React Router v4 component to redirect the user to the login page. We also pass in the current path to the login page (`redirect` in the querystring). We will use this later to redirect us back after the user logs in.

We'll do something similar to ensure that the user is not authenticated.

✦ CHANGE Add the following to `src/components/UnauthenticatedRoute.js`.

```
import React from "react";
import { Route, Redirect } from "react-router-dom";

export default function UnauthenticatedRoute({
component: C, appProps, ...rest }) {
  return (
    <Route
      {...rest}
      render={props =>
        !appProps.isAuthenticated
          ? <C {...props} {...appProps} />
          : <Redirect to="/" />}
    />
```

```
  );
}
```

Here we are checking to ensure that the user is not authenticated before we render the component that is passed in. And in the case where the user is authenticated, we use the `Redirect` component to simply send the user to the homepage.

Next, let's use these components in our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-route-that-redirects/47)

Use the Redirect Routes

Now that we created the `AuthenticatedRoute` and `UnauthenticatedRoute` in the last chapter, let's use them on the containers we want to secure.

✦ CHANGE First import them in the header of `src/Routes.js`.

```
import AuthenticatedRoute from
"./components/AuthenticatedRoute";
import UnauthenticatedRoute from
"./components/UnauthenticatedRoute";
```

Next, we simply switch to our new redirect routes.

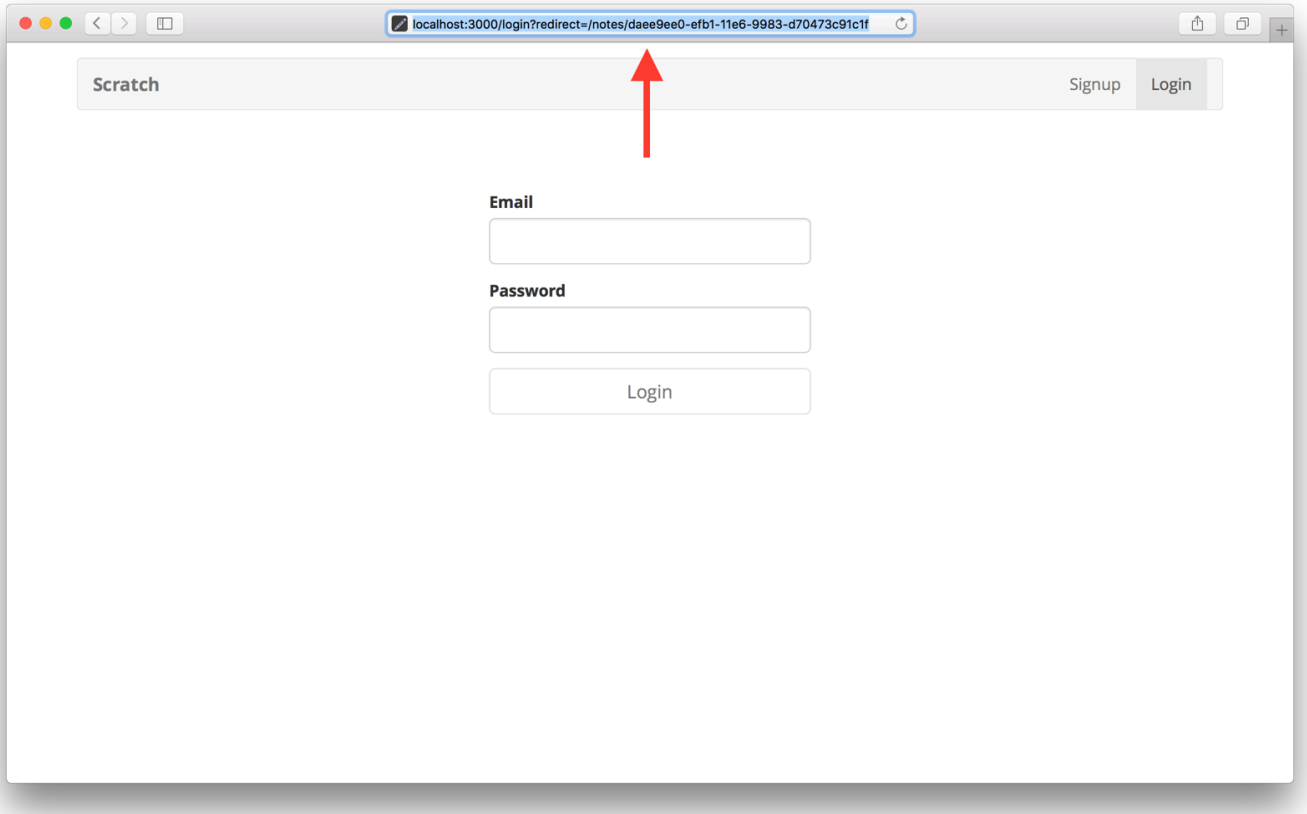
So the following routes in `src/Routes.js` would be affected.

```
<AppliedRoute path="/login" exact component={Login}
appProps={appProps} />
<AppliedRoute path="/signup" exact component={Signup}
appProps={appProps} />
<AppliedRoute path="/settings" exact component=
{Settings} appProps={appProps} />
<AppliedRoute path="/notes/new" exact component=
{NewNote} appProps={appProps} />
<AppliedRoute path="/notes/:id" exact component={Notes}
appProps={appProps} />
```

CHANGE They should now look like so:

```
<UnauthenticatedRoute path="/login" exact component=
{Login} appProps={appProps} />
<UnauthenticatedRoute path="/signup" exact component=
{Signup} appProps={appProps} />
<AuthenticatedRoute path="/settings" exact component=
{Settings} appProps={appProps} />
<AuthenticatedRoute path="/notes/new" exact component=
{NewNote} appProps={appProps} />
<AuthenticatedRoute path="/notes/:id" exact component=
{Notes} appProps={appProps} />
```

And now if we tried to load a note page while not logged in, we would be redirected to the login page with a reference to the note page.



Next, we are going to use the reference to redirect to the note page after

we login.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/use-the-redirect-routes/152)

Redirect on Login

Our secured pages redirect to the login page when the user is not logged in, with a referral to the originating page. To redirect back after they login, we need to do a couple of more things. Currently, our `Login` component does the redirecting after the user logs in. We are going to move this to the newly created `UnauthenticatedRoute` component.

Let's start by adding a method to read the `redirect` URL from the querystring.

✦ CHANGE Add the following method to your `src/components/UnauthenticatedRoute.js` below the imports.

```
function querystring(name, url = window.location.href)
{
  name = name.replace(/[[]]/g, "\\$&");

  const regex = new RegExp("[?&]" + name + "(=
([^&#]*)|&|#|$)", "i");
  const results = regex.exec(url);

  if (!results) {
    return null;
  }
  if (!results[2]) {
    return "";
```

```
  }

  return decodeURIComponent(results[2].replace(/\+/g, "
"));
}
```

This method takes the querystring param we want to read and returns it.

Now let's update our component to use this parameter when it redirects.

✦ CHANGE Replace our current `UnauthenticatedRoute` function component with the following.

```
export default function UnauthenticatedRoute({
component: C, appProps, ...rest }) {
  const redirect = querystring("redirect");
  return (
    <Route
      {...rest}
      render={props =>
        !appProps.isAuthenticated
          ? <C {...props} {...appProps} />
          : <Redirect
              to={redirect === "" || redirect === null
? "/" : redirect}
            />}
    />
  );
}
```

✦ CHANGE And remove the following from the `handleSubmit` method in `src/containers/Login.js`.

```
props.history.push("/");
```

Now our login page should redirect after we login. And that's it! Our app is ready to go live.

Commit the Changes

CHANGE Let's commit our code so far and push it to GitHub.

```
$ git add .
$ git commit -m "Building our React app"
$ git push
```

Next we'll be looking at how to automate this stack so we can use it for our future projects. You can also take a look at how to add a Login with Facebook option in the Facebook Login with Cognito using AWS Amplify (/chapters/facebook-login-with-cognito-using-aws-amplify.html) chapter. It builds on what we have covered so far.

CHECKPOINT

For reference, here is the code we have so far.

Backend Source :redirect-on-login (https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/redirect-on-login)

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/redirect-on-login/24)

Getting Production Ready

Now that we've gone through the basics of creating a Serverless app, you are ready to deploy your app to production. This means that we would like to have a couple of environments (development and production) and we want to be able to automate our deployments. While setting up the backend we did a bunch of manual work to create all the resources. And you might be wondering if you need to do that everytime you create a new environment or app. Thankfuly, there is a better way!

Over the next few chapters we will look at how to get your app ready for production, starting with:

- **Infrastructure as code**

  Currently, you go through a bunch of manual steps with a lot of clicking around to configure the backend. This makes it pretty tricky to recreate this stack for a new project. Or to configure a new environment for the same project. Serverless Framework is really good for converting this entire stack into code. This means that it can automatically recreate the entire project from scratch without ever touching the AWS Console.

- **Automating deployments**

  So far you've had to deploy through your command line using the `serverless deploy` command. When you have a team working on your project, you want to make sure the deployments to production

are centralized. This ensures that you have control over what gets deployed to production. We'll go over how to automate your deployments using Seed (https://seed.run) (for the backend) and Netlify (https://netlify.com) (for the frontend).

- **Configuring environments**

  Typically while working on projects you end up creating multiple environments. For example, you'd want to make sure not to make changes directly to your app while it is in use. Thanks to the Serverless Framework and Seed we'll be able to do this with ease for the backend. And we'll do something similar for our frontend using React and Netlify.

- **Custom domains**

  Once your app is in production, you want it hosted under your domain name. This applies both to the React app (my-domain.com) and backend APIs (api.my-domain.com).

The goal of next few sections is to make sure that you have a setup that you can easily replicate and use for your future projects. This is almost exactly what we and a few of our readers have been using.

Let's get started by first converting our backend infrastructure into code.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/getting-production-ready/158)

What Is Infrastructure as Code

Serverless Framework (https://serverless.com) converts your `serverless.yml` into a CloudFormation (https://aws.amazon.com/cloudformation) template. This is a description of the infrastructure that you are trying to configure as a part of your serverless project. In our case we were describing the Lambda functions and API Gateway endpoints that we were trying to configure.

However, in earlier part of this guide we created our DynamoDB table, Cognito User Pool, S3 uploads bucket, and Cognito Identity Pool through the AWS Console. You might be wondering if this too can be configure programmatically, instead of doing them manually through the console. It definitely can!

This general pattern is called **Infrastructure as code** and it has some massive benefits. Firstly, it allows us to simply replicate our setup with a couple of simple commands. Secondly, it is not as error prone as doing it by hand. We know a few of you have run into configuration related issues by simply following the steps in the tutorial. Additionally, describing our entire infrastructure as code allows us to create multiple environments with ease. For example, you can create a dev environment where you can make and test all your changes as you work on it. And this can be kept separate from your production environment that your users are interacting with.

In the next few chapters we are going to configure our various infrastructure pieces through our `serverless.yml`. Note that, since we had previously created our resources using the console, we will not be able to configure them through code. To do this, we'll create a new project.

Serverless Framework uses the `service` name to identify projects. Since we are creating a new project we want to ensure that we use a different name from the original. Now we could have simply overwritten the existing project but the resources were previously created by hand and will conflict when we try to create them through code.

Update the serverless.yml

✦ CHANGE Open the `serverless.yml` and find the following line:

```
service: notes-app-api
```

✦ CHANGE And replace it with this:

```
service: notes-app-2-api
```

✦ CHANGE Also, find this line in the `serverless.yml`:

```
  stage: prod
```

✦ CHANGE And replace it with:

```
  stage: dev
```

We are defaulting the stage to `dev` instead of `prod`. This will become clear later when we create multiple environments.

Let's start by configuring our DynamoDB in our `serverless.yml`.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/what-is-infrastructure-as-code/161)

Configure DynamoDB in Serverless

We are now going to start creating our resources through our `serverless.yml`. Starting with DynamoDB.

Create the Resource

✦ CHANGE Add the following to `resources/dynamodb-table.yml`.

```
Resources:
  NotesTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: ${self:custom.tableName}
      AttributeDefinitions:
        - AttributeName: userId
          AttributeType: S
        - AttributeName: noteId
          AttributeType: S
      KeySchema:
        - AttributeName: userId
          KeyType: HASH
        - AttributeName: noteId
          KeyType: RANGE
      # Set the capacity to auto-scale
```

```
      BillingMode: PAY_PER_REQUEST
```

Let's quickly go over what we are doing here.

1. We are describing a DynamoDB table resource called `NotesTable`.
2. We get the table name from the custom variable `${self:custom.tableName}`. This is generated dynamically in our `serverless.yml`. We will look at this in detail below.
3. We are also configuring the two attributes of our table as `userId` and `noteId` and specifying them as our primary key.
4. Finally, we are provisioning the read/write capacity for our table through a couple of custom variables as well. We will be defining this shortly.

Add the Resource

Now let's add a reference to this resource in our project.

✦ CHANGE Replace the `resources:` block at the bottom of our `serverless.yml` with the following:

```
# Create our resources with separate CloudFormation
templates
resources:
  # API Gateway Errors
  - ${file(resources/api-gateway-errors.yml)}
  # DynamoDB
  - ${file(resources/dynamodb-table.yml)}
```

✦ CHANGE Add the following `custom:` block at the top of our `serverless.yml` above the `provider:` block.

```
custom:
  # Our stage is based on what is passed in when
running serverless
  # commands. Or fallsback to what we have set in the
provider section.
  stage: ${opt:stage, self:provider.stage}
  # Set the table name here so we can use it while
testing locally
  tableName: ${self:custom.stage}-notes
```

We added a couple of things here that are worth spending some time on:

- We first create a custom variable called `stage`. You might be wondering why we need a custom variable for this when we already have `stage: dev` in the `provider:` block. This is because we want to set the current stage of our project based on what is set through the `serverless deploy --stage $STAGE` command. And if a stage is not set when we deploy, we want to fallback to the one we have set in the provider block. So `${opt:stage, self:provider.stage}`, is telling Serverless to first look for the `opt:stage` (the one passed in through the command line), and then fallback to `self:provider.stage` (the one in the provider block.

- The table name is based on the stage we are deploying to - `${self:custom.stage}-notes`. The reason this is dynamically set is because we want to create a separate table when we deploy to a new stage (environment). So when we deploy to `dev` we will create a

DynamoDB table called `dev-notes` and when we deploy to `prod`, it'll be called `prod-notes`. This allows us to clearly separate the resources (and data) we use in our various environments.

- Finally, we are using the `PAY_PER_REQUEST` setting for the `BillingMode`. This tells DynamoDB that we want to pay per request and use the On-Demand Capacity (https://aws.amazon.com/dynamodb/pricing/on-demand/) option. With DynamoDB in On-Demand mode, our database is now truly Serverless. This option can be very cost-effective, especially if you are just starting out and your workloads are not very predictable or stable. On the other hand, if you know exactly how much capacity you need, the Provisioned Capacity (https://aws.amazon.com/dynamodb/pricing/provisioned/) mode would work out to be cheaper.

A lot of the above might sound tricky and overly complicated right now. But we are setting it up so that we can automate and replicate our entire setup with ease. Note that, Serverless Framework (and CloudFormation behind the scenes) will be completely managing our resources based on the `serverless.yml`. This means that if you have a typo in your table name, the old table will be removed and a new one will be created in place. To prevent accidentally deleting serverless resources (like DynamoDB tables), you need to set the `DeletionPolicy: Retain` flag. We have a detailed post on this over on the Seed blog (https://seed.run/blog/how-to-prevent-accidentally-deleting-serverless-resources).

We are also going to make a quick tweak to reference the DynamoDB resource that we are creating.

✦ CHANGE Update our environment variables with the new generated table

name. Replace the `environment:` block with the following:

```
  # These environment variables are made available to
our functions
  # under process.env.
  environment:
    tableName: ${self:custom.tableName}
    stripeSecretKey: ${env:STRIPE_SECRET_KEY}
```

CHANGE Replace the `iamRoleStatements:` block in your `serverless.yml` with the following.

```
  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:DescribeTable
        - dynamodb:Query
        - dynamodb:Scan
        - dynamodb:GetItem
        - dynamodb:PutItem
        - dynamodb:UpdateItem
        - dynamodb:DeleteItem
      # Restrict our IAM role permissions to
      # the specific table for the stage
      Resource:
        - "Fn::GetAtt": [ NotesTable, Arn ]
```

Make sure to **copy the indentation properly**. These two blocks fall under the `provider` block and need to be indented as such.

A couple of interesting things we are doing here:

1. The `environment:` block here is basically telling Serverless Framework to make the variables available as `process.env` in our Lambda functions. For example, `process.env.tableName` would be set to the DynamoDB table name for this stage. We will need this later when we are connecting to our database.

2. For the `tableName` specifically, we are getting it by referencing our custom variable from above.

3. For the case of our `iamRoleStatements:` we are now specifically stating which table we want to connect to. This block is telling AWS that these are the only resources that our Lambda functions have access to.

Next, let's add our S3 bucket for file uploads.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/configure-dynamodb-in-serverless/162)

Configure S3 in Serverless

Now that we have DynamoDB configured, let's look at how we can configure the S3 file uploads bucket through our `serverless.yml`.

Create the Resource

✦ CHANGE Add the following to `resources/s3-bucket.yml`.

```
Resources:
  AttachmentsBucket:
    Type: AWS::S3::Bucket
    Properties:
      # Set the CORS policy
      CorsConfiguration:
        CorsRules:
          -
            AllowedOrigins:
              - '*'
            AllowedHeaders:
              - '*'
            AllowedMethods:
              - GET
              - PUT
              - POST
              - DELETE
              - HEAD
```

```
            MaxAge: 3000

# Print out the name of the bucket that is created
Outputs:
  AttachmentsBucketName:
    Value:
      Ref: AttachmentsBucket
```

If you recall from the Create an S3 bucket for file uploads (/chapters/create-an-s3-bucket-for-file-uploads.html) chapter, we had created a bucket and configured the CORS policy for it. We needed to do this because we are going to be uploading directly from our frontend client. We configure the same policy here.

S3 buckets (unlike DynamoDB tables) are globally named, so it is not really possible for us to know what our bucket is going to be called beforehand. Hence, we let CloudFormation generate the name for us and we just add the `Outputs:` block to tell it to print it out so we can use it later.

Add the Resource

✦ CHANGE Let's reference the resource in our `serverless.yml`. Replace your `resources:` block with the following.

```
# Create our resources with separate CloudFormation
templates
resources:
  # API Gateway Errors
  - ${file(resources/api-gateway-errors.yml)}
  # DynamoDB
  - ${file(resources/dynamodb-table.yml)}
```

```
    # S3
    - ${file(resources/s3-bucket.yml)}
```

And that's it. Next let's look into configuring our Cognito User Pool.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/configure-s3-in-serverless/163)

Configure Cognito User Pool in Serverless

Now let's look into setting up Cognito User Pool through the `serverless.yml`. It should be very similar to the one we did by hand in the Create a Cognito user pool (/chapters/create-a-cognito-user-pool.html) chapter.

Create the Resource

✦ CHANGE Add the following to `resources/cognito-user-pool.yml`.

```
Resources:
  CognitoUserPool:
    Type: AWS::Cognito::UserPool
    Properties:
      # Generate a name based on the stage
      UserPoolName: ${self:custom.stage}-user-pool
      # Set email as an alias
      UsernameAttributes:
        - email
      AutoVerifiedAttributes:
        - email

  CognitoUserPoolClient:
    Type: AWS::Cognito::UserPoolClient
```

```
    Properties:
      # Generate an app client name based on the stage
      ClientName: ${self:custom.stage}-user-pool-client
      UserPoolId:
        Ref: CognitoUserPool
      ExplicitAuthFlows:
        - ADMIN_NO_SRP_AUTH
      GenerateSecret: false

# Print out the Id of the User Pool that is created
Outputs:
  UserPoolId:
    Value:
      Ref: CognitoUserPool

  UserPoolClientId:
    Value:
      Ref: CognitoUserPoolClient
```

Let's quickly go over what we are doing here:

- We are naming our User Pool (and the User Pool app client) based on the stage by using the custom variable `${self:custom.stage}`.
- We are setting the `UsernameAttributes` as email. This is telling the User Pool that we want our users to be able to log in with their email as their username.
- Just like our S3 bucket, we want CloudFormation to tell us the User Pool Id and the User Pool Client Id that is generated. We do this in the `Outputs:` block at the end.

Add the Resource

✦ CHANGE Let's reference the resource in our `serverless.yml`. Replace your `resources:` block with the following.

```
# Create our resources with separate CloudFormation templates
resources:
  # API Gateway Errors
  - ${file(resources/api-gateway-errors.yml)}
  # DynamoDB
  - ${file(resources/dynamodb-table.yml)}
  # S3
  - ${file(resources/s3-bucket.yml)}
  # Cognito
  - ${file(resources/cognito-user-pool.yml)}
```

And next let's tie all of this together by configuring our Cognito Identity Pool.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/configure-cognito-user-pool-in-serverless/164)

Configure Cognito Identity Pool in Serverless

If you recall from the first part of this tutorial, we use the Cognito Identity Pool as a way to control which AWS resources our logged in users will have access to. We also tie in our Cognito User Pool as our authentication provider.

Create the Resource

✦ CHANGE Add the following to `resources/cognito-identity-pool.yml`.

```
Resources:
  # The federated identity for our user pool to auth
with
  CognitoIdentityPool:
    Type: AWS::Cognito::IdentityPool
    Properties:
      # Generate a name based on the stage
      IdentityPoolName:
${self:custom.stage}IdentityPool
      # Don't allow unathenticated users
      AllowUnauthenticatedIdentities: false
      # Link to our User Pool
      CognitoIdentityProviders:
```

```
        - ClientId:
            Ref: CognitoUserPoolClient
          ProviderName:
            Fn::GetAtt: [ "CognitoUserPool",
"ProviderName" ]

  # IAM roles
  CognitoIdentityPoolRoles:
    Type: AWS::Cognito::IdentityPoolRoleAttachment
    Properties:
      IdentityPoolId:
        Ref: CognitoIdentityPool
      Roles:
        authenticated:
          Fn::GetAtt: [CognitoAuthRole, Arn]

  # IAM role used for authenticated users
  CognitoAuthRole:
    Type: AWS::IAM::Role
    Properties:
      Path: /
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: 'Allow'
            Principal:
              Federated: 'cognito-
identity.amazonaws.com'
            Action:
              - 'sts:AssumeRoleWithWebIdentity'
            Condition:
```

```
            StringEquals:
              'cognito-identity.amazonaws.com:aud':
                Ref: CognitoIdentityPool
            'ForAnyValue:StringLike':
              'cognito-identity.amazonaws.com:amr':
authenticated
      Policies:
        - PolicyName: 'CognitoAuthorizedPolicy'
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: 'Allow'
                Action:
                  - 'mobileanalytics:PutEvents'
                  - 'cognito-sync:*'
                  - 'cognito-identity:*'
                Resource: '*'

              # Allow users to invoke our API
              - Effect: 'Allow'
                Action:
                  - 'execute-api:Invoke'
                Resource:
                  Fn::Join:
                    - ''
                    -
                      - 'arn:aws:execute-api:'
                      - Ref: AWS::Region
                      - ':'
                      - Ref: AWS::AccountId
                      - ':'
```

```
                    - Ref: ApiGatewayRestApi
                    - '/*'

              # Allow users to upload attachments to
their
              # folder inside our S3 bucket
              - Effect: 'Allow'
                Action:
                  - 's3:*'
                Resource:
                  - Fn::Join:
                    - ''
                    -
                      - Fn::GetAtt: [AttachmentsBucket,
Arn]
                      - '/private/'
                      - '$'
                      - '{cognito-
identity.amazonaws.com:sub}/*'

# Print out the Id of the Identity Pool that is created
Outputs:
  IdentityPoolId:
    Value:
      Ref: CognitoIdentityPool
```

While it looks like there's a whole lot going on here, it's pretty much exactly what we did back in the Create a Cognito identity pool (/chapters/create-a-cognito-identity-pool.html) chapter. It's just that CloudFormation can be a bit verbose and can end up looking a bit intimidating.

Let's quickly go over the various sections of this configuration:

1. First we name our Identity Pool based on the stage name using `${self:custom.stage}`.
2. We specify that we only want logged in users by adding `AllowUnauthenticatedIdentities: false`.
3. Next we state that we want to use our User Pool as the identity provider. We are doing this specifically using the `Ref: CognitoUserPoolClient` line. If you refer back to the Configure Cognito User Pool in Serverless (/chapters/configure-cognito-user-pool-in-serverless.html) chapter, you'll notice we have a block under `CognitoUserPoolClient` that we are referencing here.
4. We then attach an IAM role to our authenticated users.
5. We add the various parts to this role. This is exactly what we use in the Create a Cognito identity pool (/chapters/create-a-cognito-identity-pool.html) chapter. It just needs to be formatted this way to work with CloudFormation.
6. The `ApiGatewayRestApi` ref that you might notice is generated by Serverless Framework when you define an API endpoint in your `serverless.yml`. So in this case, we are referencing the API resource that we are creating.
7. For the S3 bucket the name is generated by AWS. So for this case we use the `Fn::GetAtt: [AttachmentsBucket, Arn]` to get it's exact name.
8. Finally, we print out the generated Identity Pool Id in the `Outputs:` block.

Add the Resource

✦ CHANGE Let's reference the resource in our `serverless.yml`. Replace your `resources:` block with the following.

```
# Create our resources with separate CloudFormation
templates
resources:
  # API Gateway Errors
  - ${file(resources/api-gateway-errors.yml)}
  # DynamoDB
  - ${file(resources/dynamodb-table.yml)}
  # S3
  - ${file(resources/s3-bucket.yml)}
  # Cognito
  - ${file(resources/cognito-user-pool.yml)}
  - ${file(resources/cognito-identity-pool.yml)}
```

Now we are ready to deploy our new Serverless infrastructure.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/configure-cognito-identity-pool-in-serverless/165)

Deploy Your Serverless Infrastructure

Now that we have all our resources configured, let's go ahead and deploy our entire infrastructure.

We should mention though that our current project has all of our resources and the Lambda functions that we had created in the first part of our tutorial. This is a common trend in serverless projects. Your *code* and *infrastructure* are not treated differently. Of course, as your projects get larger, you end up splitting them up. So you might have a separate Serverless Framework project that deploys your infrastructure while a different project just deploys your Lambda functions.

Deploy Your Project

Deploying our project is fairly straightforward thanks to our `serverless deploy` command. So go ahead and run this from the root of your project.

```
$ serverless deploy -v
```

Your output should look something like this:

```
Serverless: Stack update finished...
Service Information
service: notes-app-2-api
```

```
stage: dev
region: us-east-1
stack: notes-app-2-api-dev
api keys:
  None
endpoints:
  POST - https://8lugdaec03.execute-api.us-east-
1.amazonaws.com/dev/notes
  GET - https://8lugdaec03.execute-api.us-east-
1.amazonaws.com/dev/notes/{id}
  GET - https://8lugdaec03.execute-api.us-east-
1.amazonaws.com/dev/notes
  PUT - https://8lugdaec03.execute-api.us-east-
1.amazonaws.com/dev/notes/{id}
  DELETE - https://8lugdaec03.execute-api.us-east-
1.amazonaws.com/dev/notes/{id}
  POST - https://8lugdaec03.execute-api.us-east-
1.amazonaws.com/dev/billing
functions:
  create: notes-app-2-api-dev-create
  get: notes-app-2-api-dev-get
  list: notes-app-2-api-dev-list
  update: notes-app-2-api-dev-update
  delete: notes-app-2-api-dev-delete
  billing: notes-app-2-api-dev-billing
layers:
  None

Stack Outputs
AttachmentsBucketName: notes-app-2-api-dev-
attachmentsbucket-1fs6m3jt1vyjd
```

```
UserPoolClientId: 3j27ho9tmja8r3irrv6eh514fn
UserPoolId: us-east-1_bNip3CCUi
DeleteLambdaFunctionQualifiedArn: arn:aws:lambda:us-
east-1:232771856781:function:notes-app-2-api-dev-
delete:1
CreateLambdaFunctionQualifiedArn: arn:aws:lambda:us-
east-1:232771856781:function:notes-app-2-api-dev-
create:1
GetLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-
1:232771856781:function:notes-app-2-api-dev-get:1
UpdateLambdaFunctionQualifiedArn: arn:aws:lambda:us-
east-1:232771856781:function:notes-app-2-api-dev-
update:1
IdentityPoolId: us-east-1:346c5a95-467b-47a9-ab2b-
059fb3c31215
BillingLambdaFunctionQualifiedArn: arn:aws:lambda:us-
east-1:232771856781:function:notes-app-2-api-dev-
billing:1
ListLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-
1:232771856781:function:notes-app-2-api-dev-list:1
ServiceEndpoint: https://8lugdaec03.execute-api.us-
east-1.amazonaws.com/dev
ServerlessDeploymentBucketName: notes-app-2-api-dev-
serverlessdeploymentbucket-heebnceeapbg
```

A couple of things to note here:

- We are deploying to a stage called `dev`. This has been set in our `serverless.yml` under the `provider:` block. We can override this by explicitly passing it in by running the `serverless deploy --stage $STAGE_NAME` command instead.

- Our deploy command (with the `-v` option) prints out the output we had requested in our resources. For example, `AttachmentsBucketName` is the S3 file uploads bucket that was created and the `UserPoolId` is the Id of our User Pool.

- Finally, you can run the deploy command and CloudFormation will only update the parts that have changed. So you can confidently run this command without worrying about it re-creating your entire infrastructure from scratch.

And that's it! Our entire infrastructure is completely configured and deployed automatically.

Next, we will look at how we can automate our deployments. We want to set it up so that when we `git push` our changes and our app will deploy automatically. We'll also be setting up our environments so that when we work on our app, it does not affect our users.

Commit the Changes

CHANGE Let's commit our code so far and push it to GitHub.

```
$ git add .
$ git commit -m "Setting up our Serverless
infrastructure"
$ git push
```

CHECKPOINT

For reference, here is the code we have so far.

Backend Source :deploy-your-serverless-infrastructure (https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/deploy-your-serverless-infrastructure)

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploy-your-serverless-infrastructure/167)

Automating Serverless Deployments

So to recap, this is what we have so far:

- A serverless project that has all it's infrastructure completely configured in code
- A way to handle secrets locally
- And finally, a way to run unit tests to test our business logic

All of this is neatly committed in a Git repo.

Next we are going to use our Git repo to automate our deployments. This essentially means that we can deploy our entire project by simply pushing our changes to Git. This can be incredibly useful since you won't need to create any special scripts or configurations to deploy your code. You can also have multiple people on your team deploy with ease.

Along with automating deployments, we are also going to look at working with multiple environments. We want to create clear separation between our production environment and our dev environment. We are going to create a workflow where we continually deploy to our dev (or any non-prod) environment. But we will be using a manual promotion step when we promote to production. We'll also look at configuring custom domains for APIs.

For automating our serverless backend, we are going to be using a service

called Seed (https://seed.run). Full disclosure, we also built Seed. You can replace most of this section with a service like Travis CI (https://travis-ci.org) or CircleCI (https://circleci.com). It's a bit more cumbersome and needs some scripting. We have a couple of posts on this over on the Seed (https://seed.run) blog:

- Configure a CI/CD pipeline for Serverless apps on CircleCI (https://seed.run/blog/how-to-build-a-cicd-pipeline-for-serverless-apps-with-circleci)
- Configure a CI/CD pipeline for Serverless apps on Travis CI (https://seed.run/blog/how-to-build-a-cicd-pipeline-for-serverless-apps-with-travis-ci)

Let's get started with setting up your project on Seed.

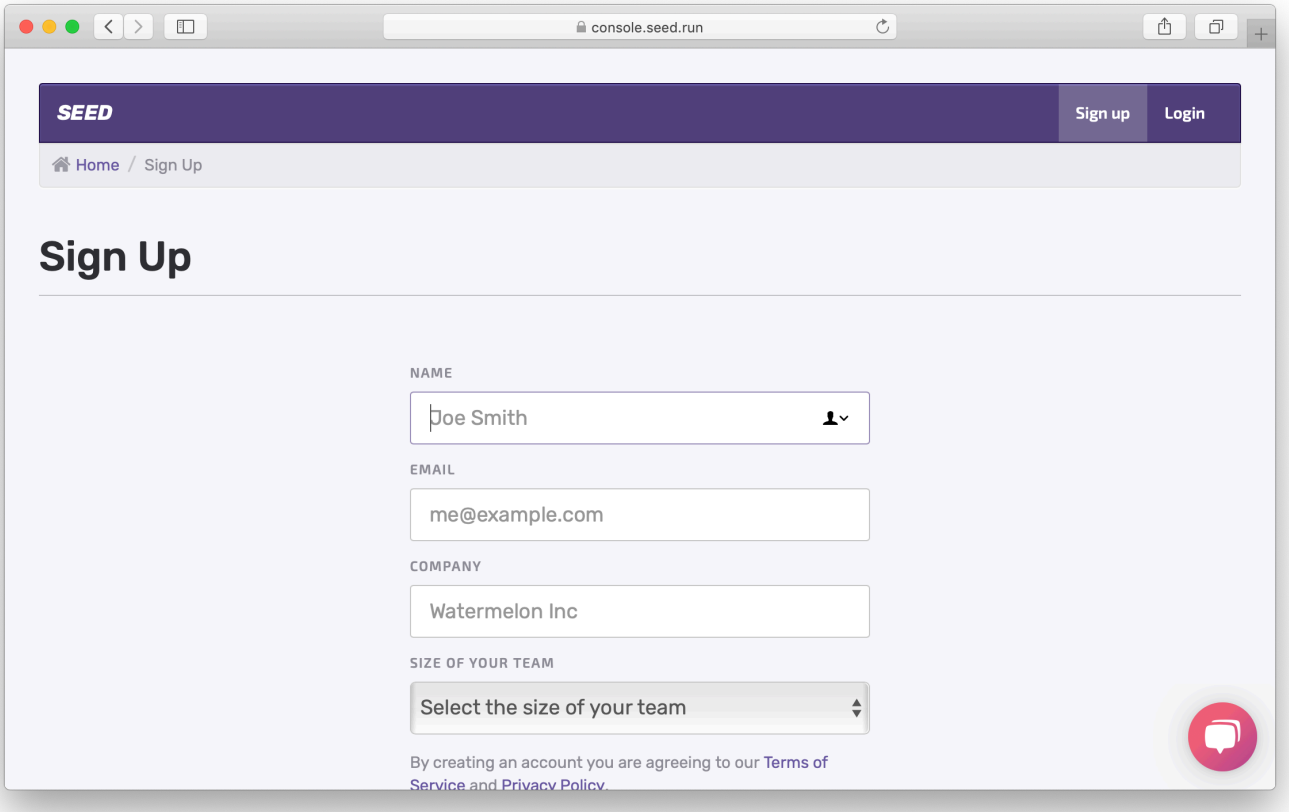
For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/automating-serverless-deployments/174)

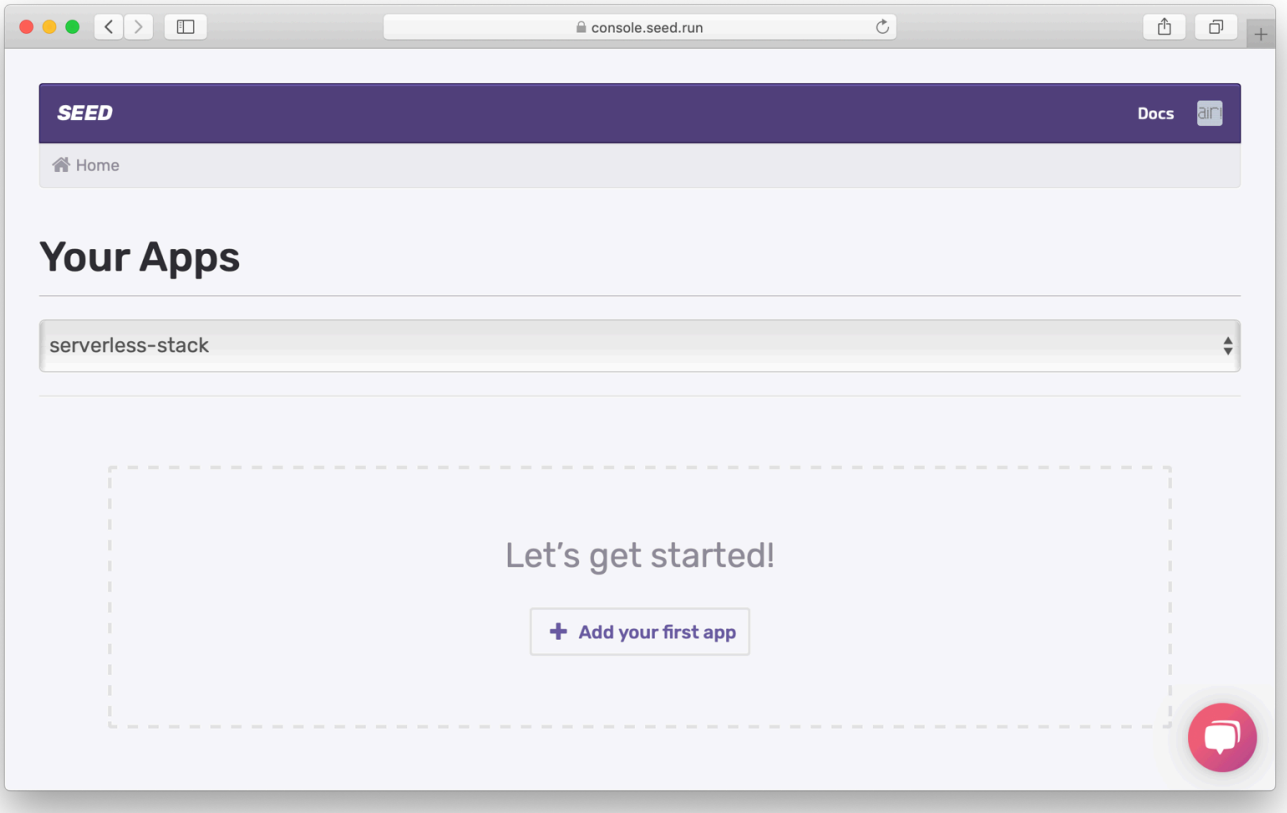
Setting up Your Project on Seed

We are going to use Seed (https://seed.run) to automate our serverless deployments and manage our environments.

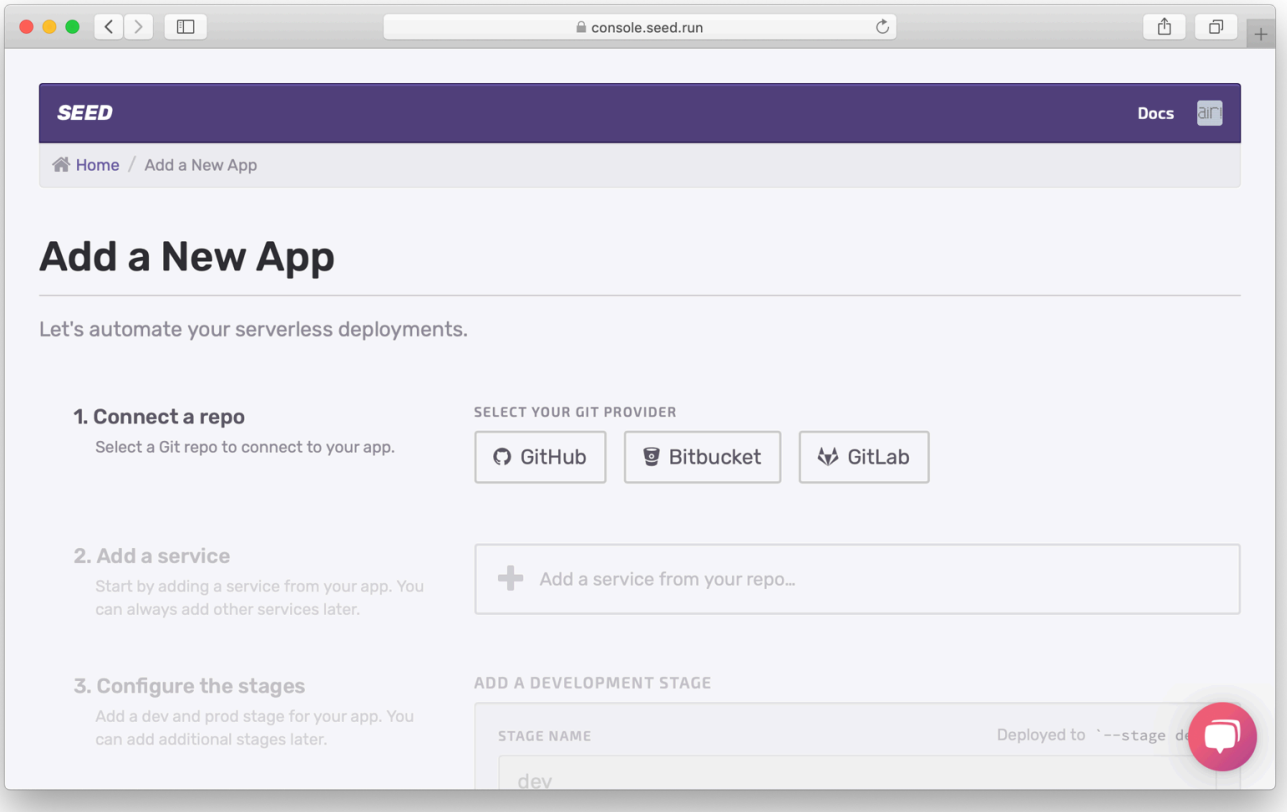
Start by signing up for a free account here (https://console.seed.run/signup).



Let's **Add your first app.**

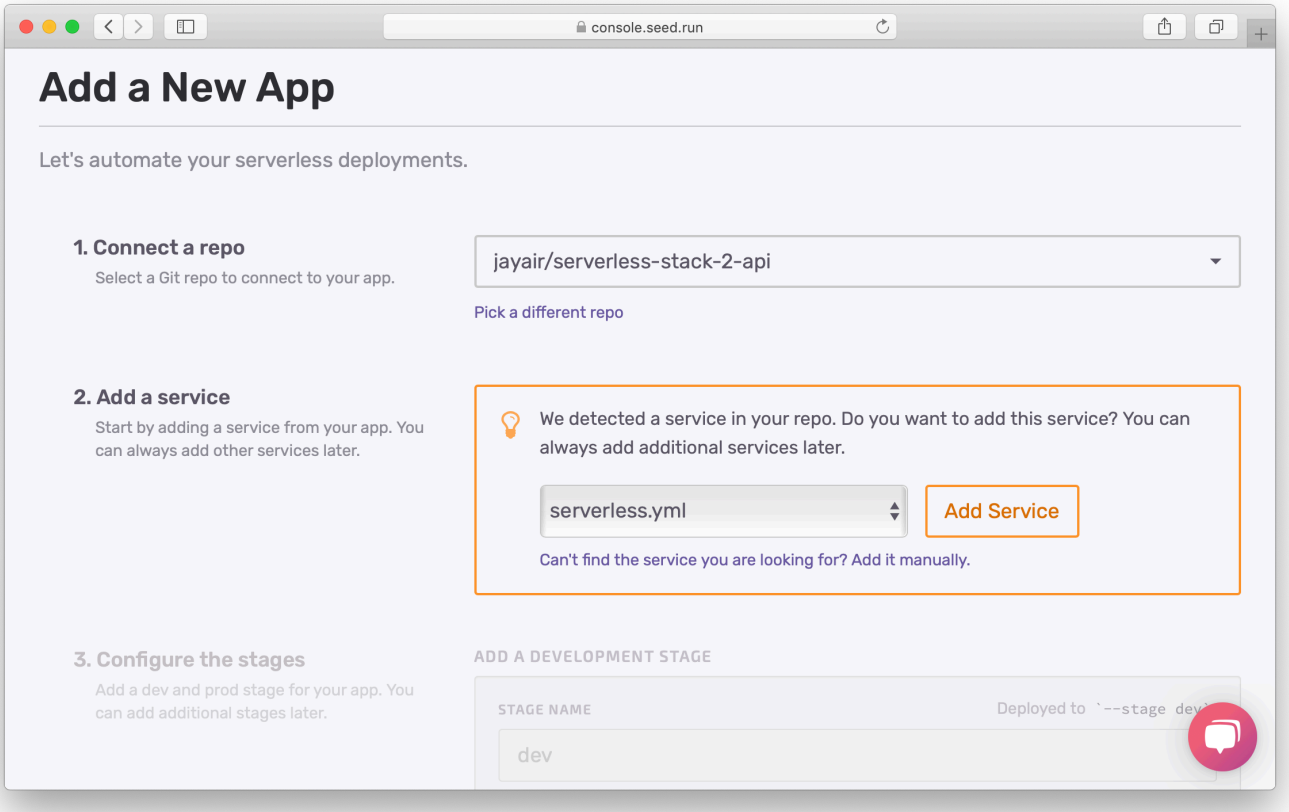


Now to add your project, select **GitHub** as your git provider. You'll be asked to give Seed permission to your GitHub account.



Select the repo we've been using so far.

Next, Seed will scan your repos for a `serverless.yml`. Hit **Add Service** to confirm this.



Note that, if your `serverless.yml` is not in your project root, you will need to change the path.

Seed deploys to your AWS account on your behalf. You should create a separate IAM user with exact permissions that your project needs. You can read more about this here (https://seed.run/docs/customizing-your-iam-policy). But for now we'll simply use the one we've used in this tutorial.

✦ CHANGE Run the following command.

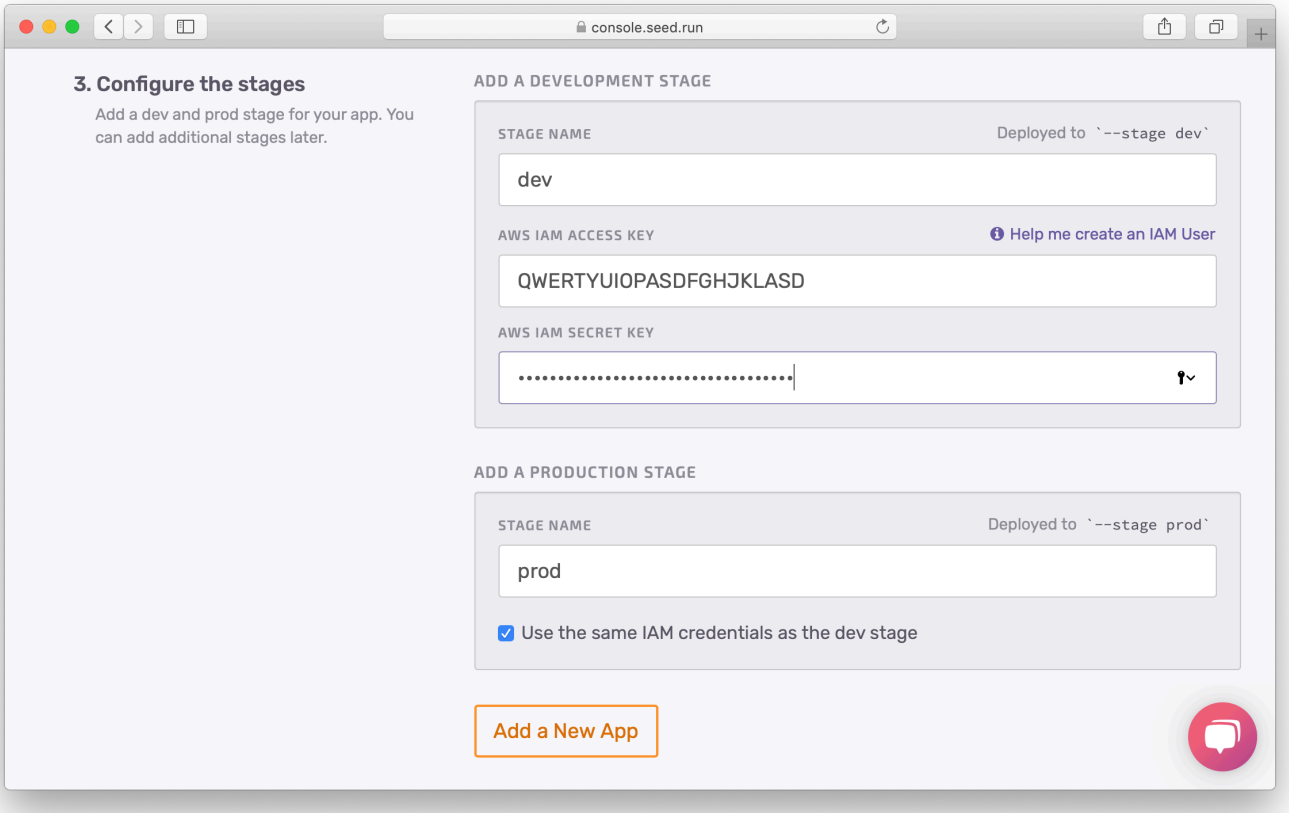
```
$ cat ~/.aws/credentials
```

The output should look something like this.

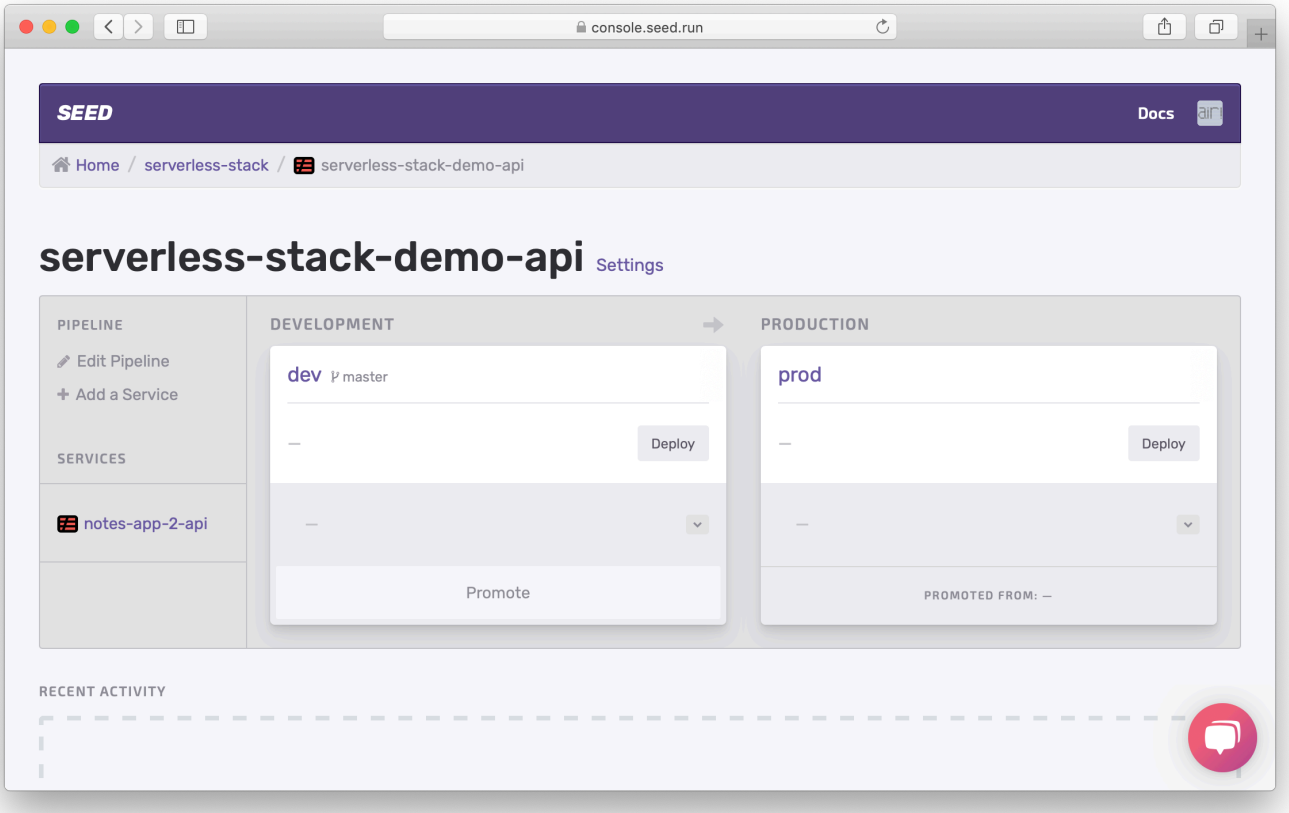
```
[default]
aws_access_key_id = YOUR_IAM_ACCESS_KEY
aws_secret_access_key = YOUR_IAM_SECRET_KEY
```

Seed will also create a couple of stages (or environments) for you. By default, it'll create a **dev** and a **prod** stage using the same AWS credentials. You can customize these but for us this is perfect.

Fill in the credentials and click **Add a New App**.



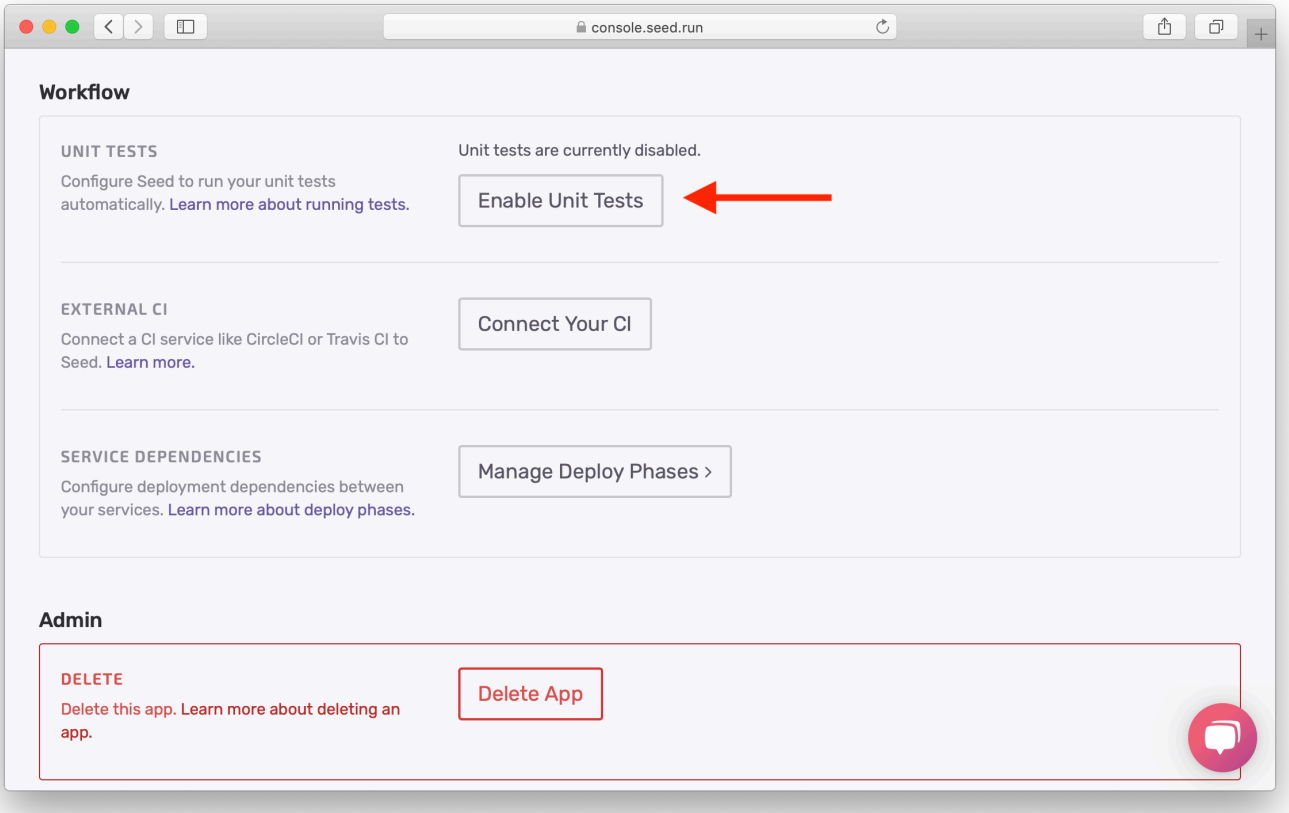
You new app is ready to go!



You'll notice a few things here. First, we have a service called **notes-app-2-api**. It's picking up the service name from our `serverless.yml`. You can choose to change this by clicking on the service and editing its name. A Serverless app can have multiple services within it. A service (roughly speaking) is a reference to a `serverless.yml` file. In our case we have one service in the root of our repo. You'll also notice the two stages that have been created.

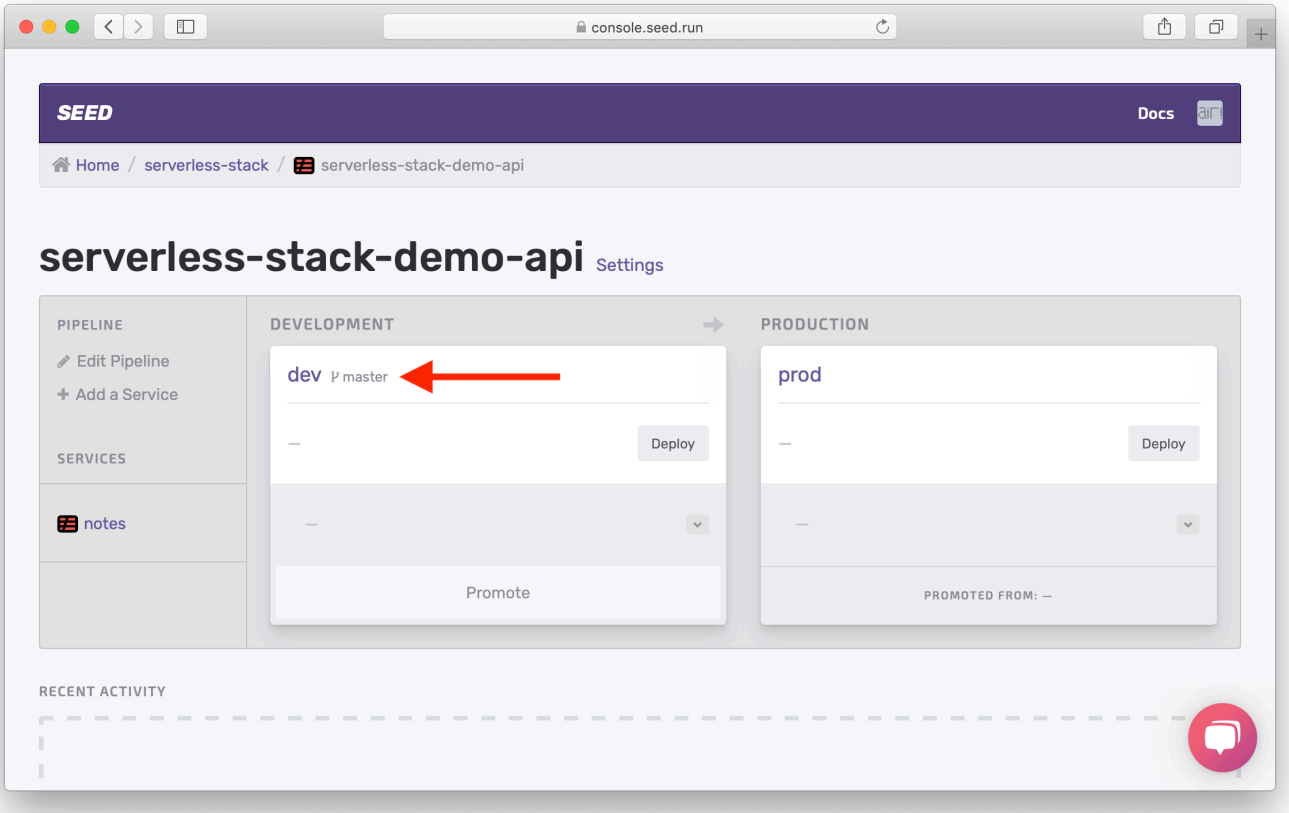
Now before we proceed to deploying our app, we need to enable running unit tests as a part of our build process. You'll recall that we had added a couple of tests back in the unit tests (/chapters/unit-tests-in-serverless.html) chapter. And we want to run those before we deploy our app.

To do this, hit the **Settings** link and click **Enable Unit Tests**.

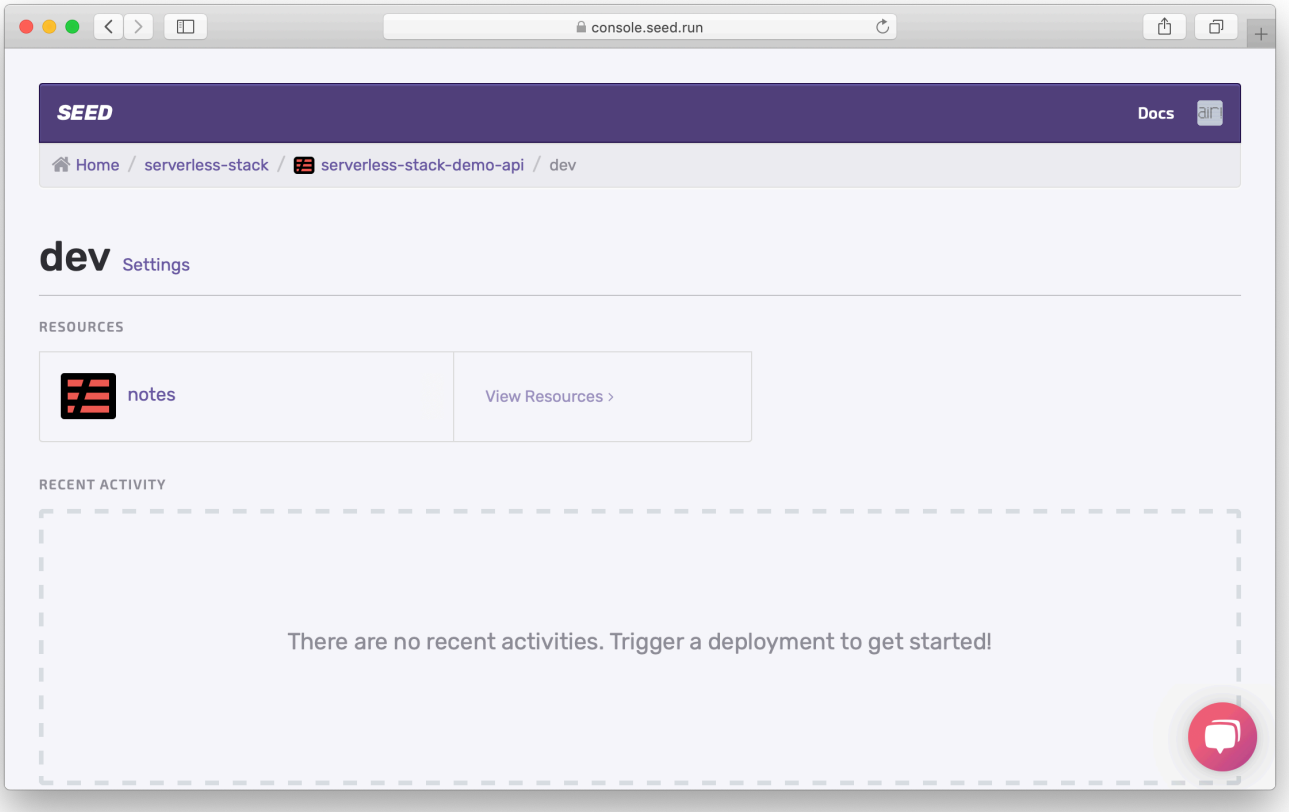


Back in our pipeline, you'll notice that our **dev** stage is hooked up to master. This means that any commits to master will trigger a build in dev.

Click on **dev**.



You'll see that we haven't deployed to this stage yet.



However, before we do that, we'll need to add our secret environment variables.

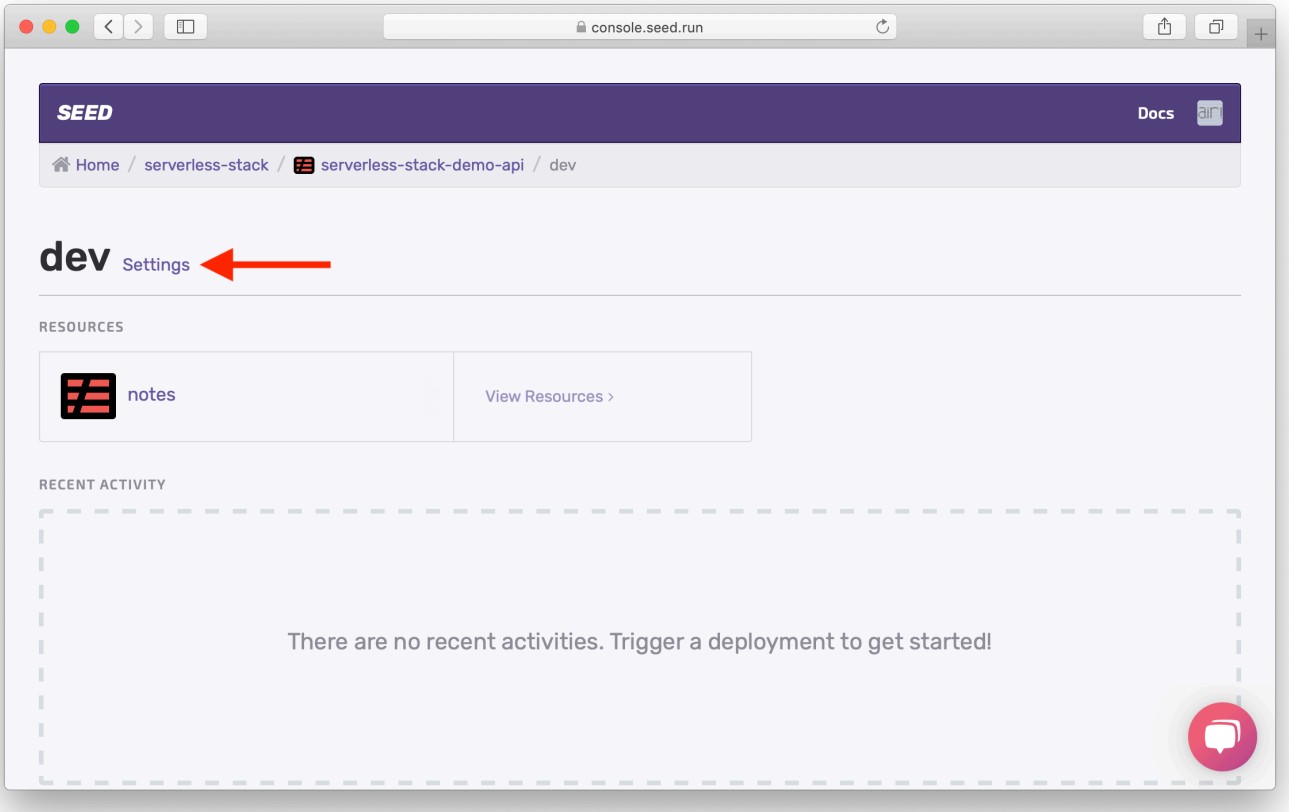
For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/setting-up-your-project-on-seed/175)

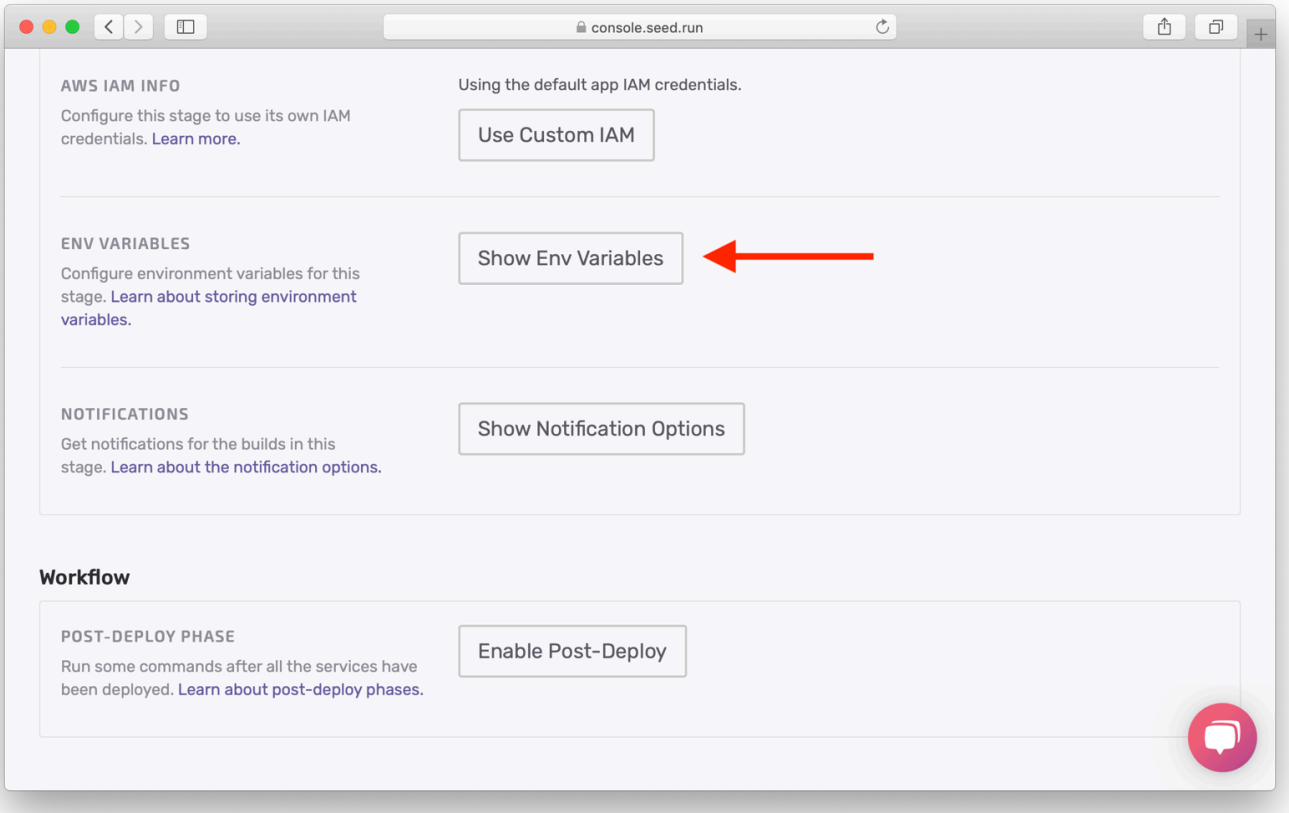
Configure Secrets in Seed

Before we can do our first deployment, we need to make sure to configure our secret environment variables. If you'll recall, we have explicitly not stored these in our code (or in Git). This means that if somebody else on our team needs to deploy, we'll need to pass the `.env` file around. Instead we'll configure Seed (https://seed.run) to deploy with our secrets for us.

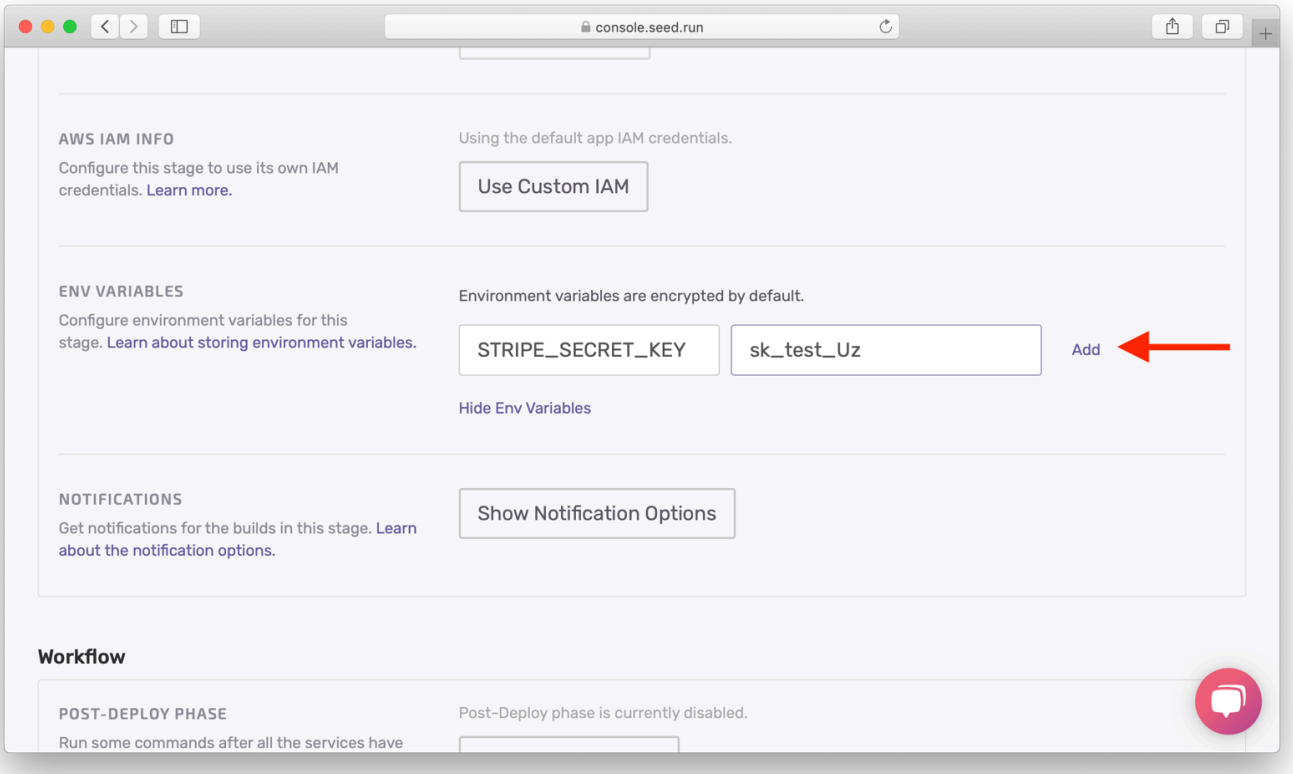
To do that, hit the **Settings** button in our **dev** stage.



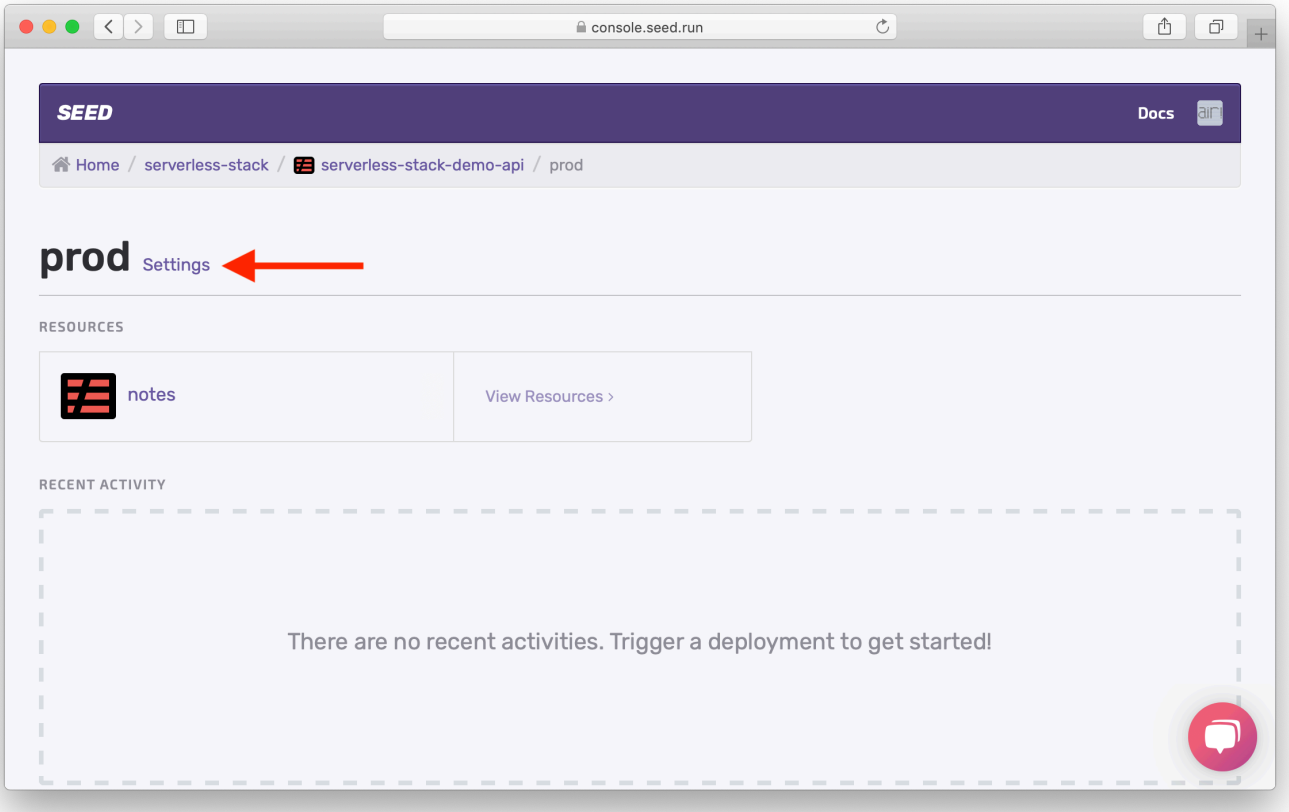
Here click **Show Env Variables**.



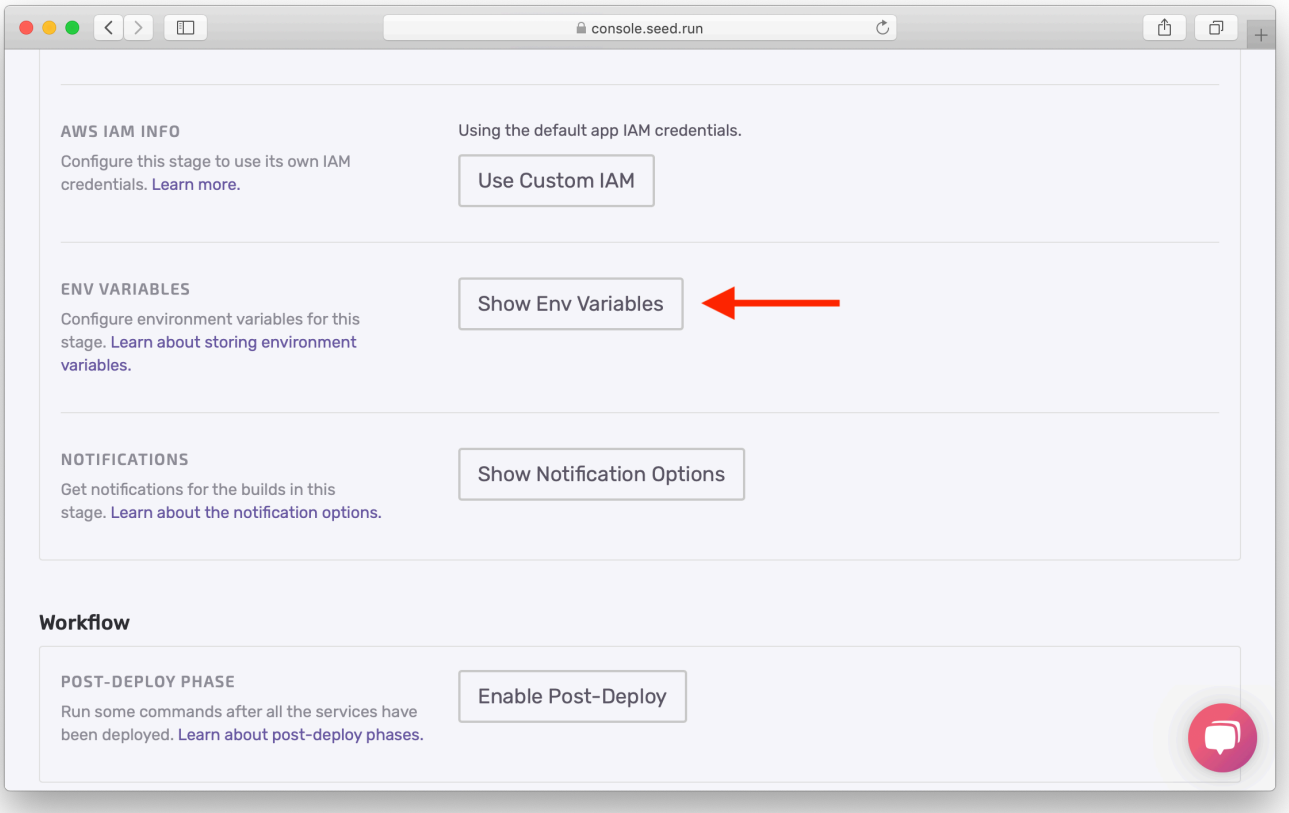
And type in `STRIPE_SECRET_KEY` as the **Key** and the value should be the `STRIPE_TEST_SECRET_KEY` back from the Load secrets from env.yml (/chapters/load-secrets-from-env.html) chapter. Hit **Add** to save your secret key.



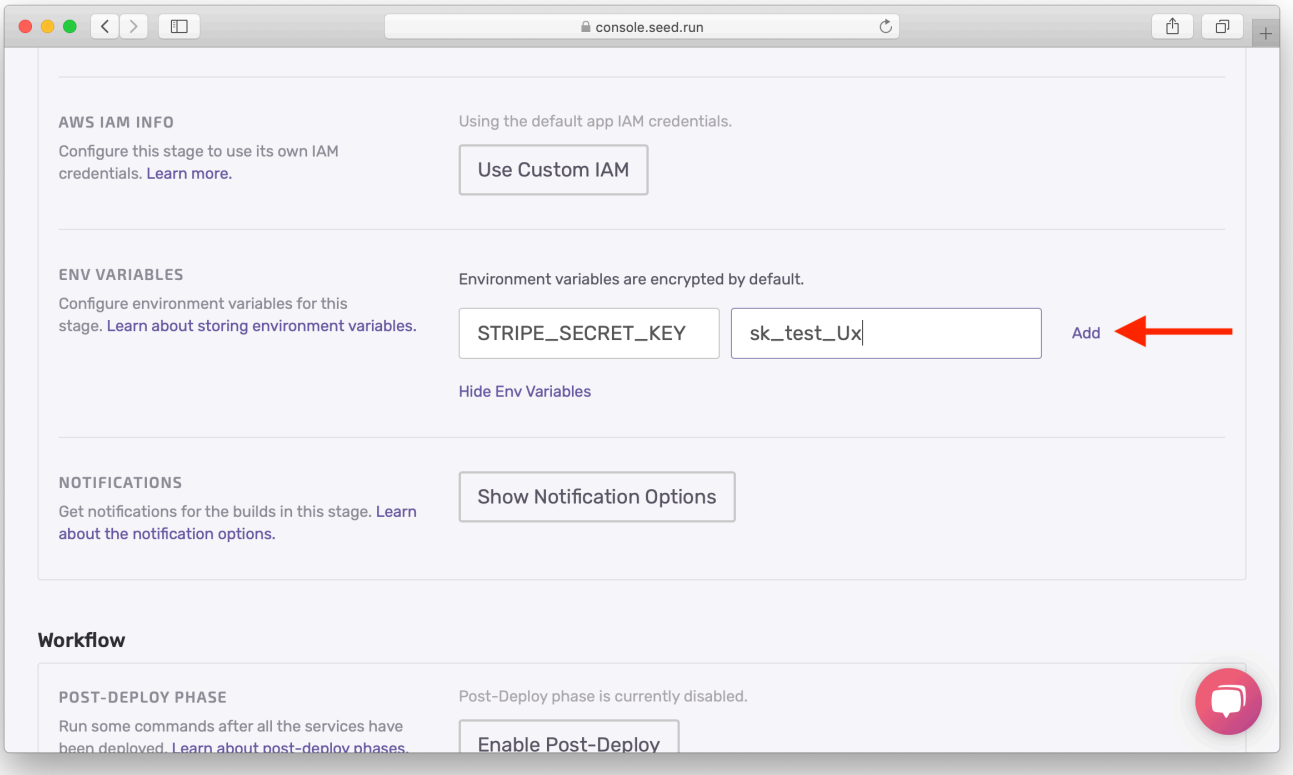
Next we need to configure our secrets for the `prod` stage. Head over there and hit the **Settings** button.



Click **Show Env Variables**.



And type in `STRIPE_SECRET_KEY` as the **Key** and the value should be the `STRIPE_TEST_SECRET_KEY` back from the Load secrets from env.yml (/chapters/load-secrets-from-env.html) chapter. Hit **Add** to save your secret key.



Next, we'll trigger our first deployment on Seed.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/configure-secrets-in-seed/176)

Deploying Through Seed

Now, we are ready to make our first deployment. You can either Git push a new change to master to trigger it. Or we can just go into the **dev** stage and hit the **Trigger Deploy** button.

Let's do it through Git.

✦ CHANGE Go back to your project root and run the following.

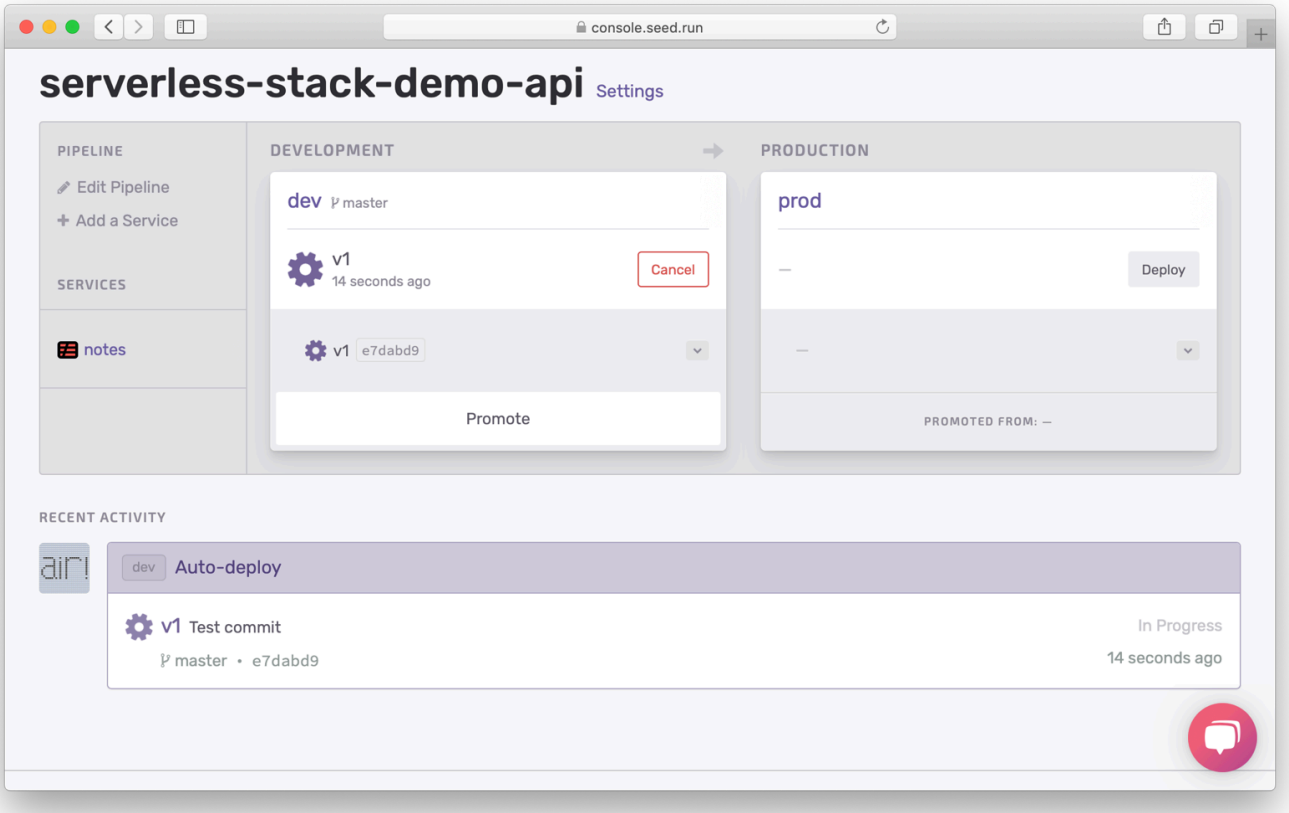
```
$ npm version patch
```

This is simply updating the NPM version for your project. It is a good way to keep track of the changes you are making to your project. And it also creates a quick Git commit for us.

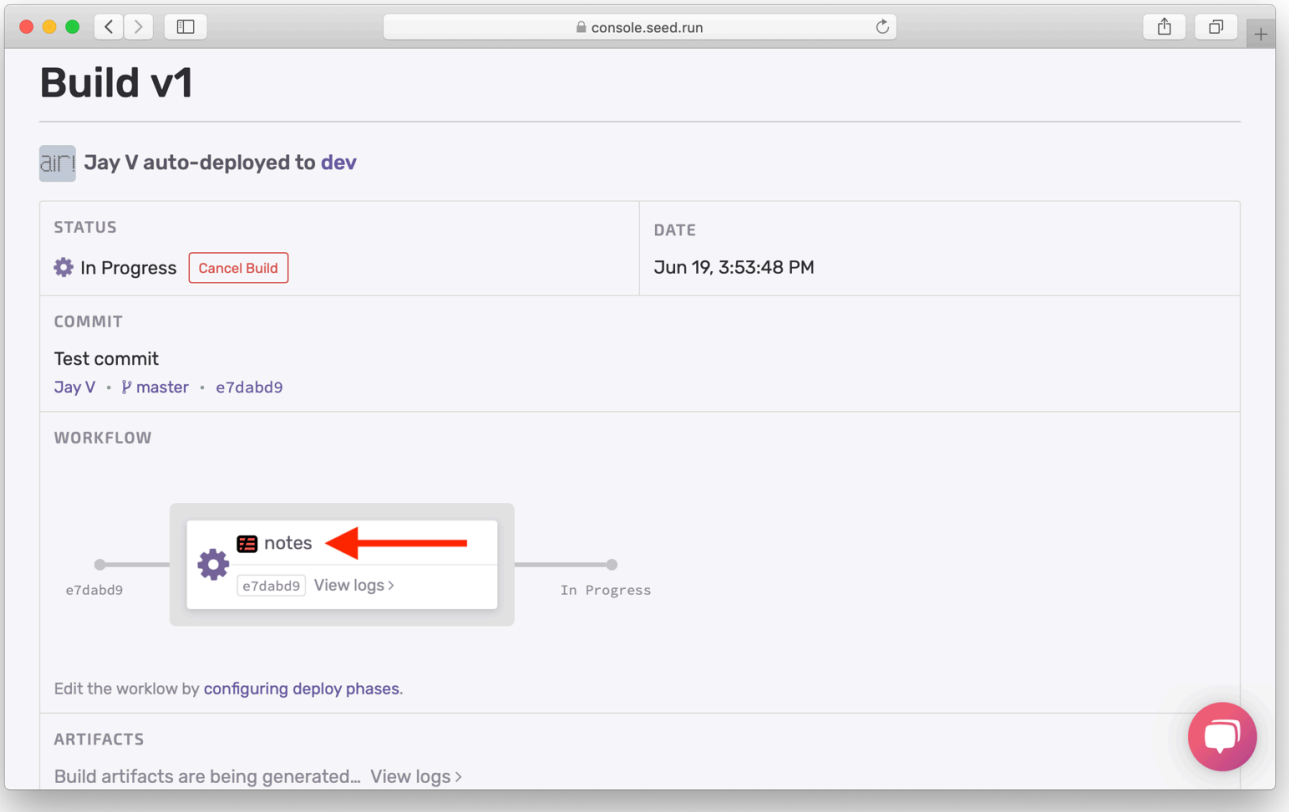
✦ CHANGE Push the change using.

```
$ git push
```

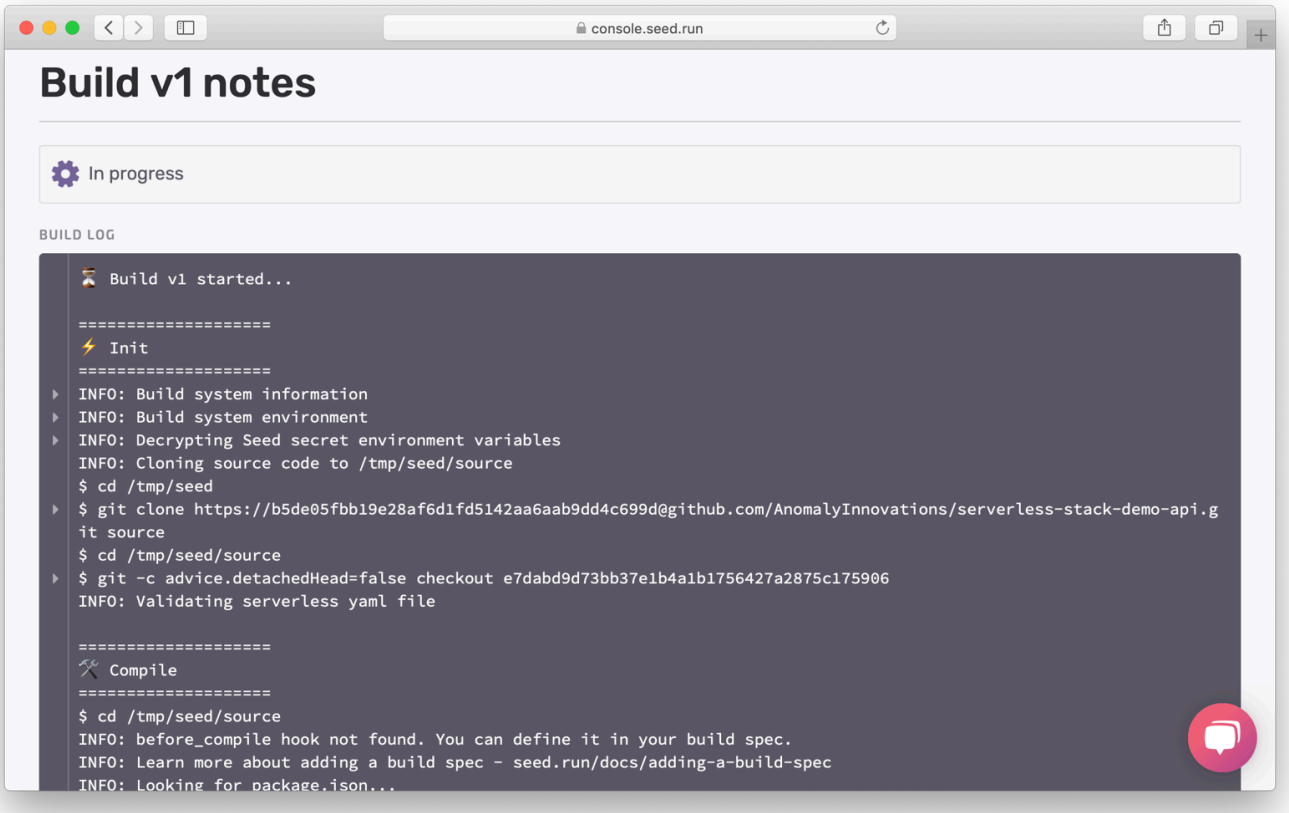
Now if you head into the **dev** stage in Seed, you should see a build in progress. Now to see the build logs, you can hit **Build v1**.



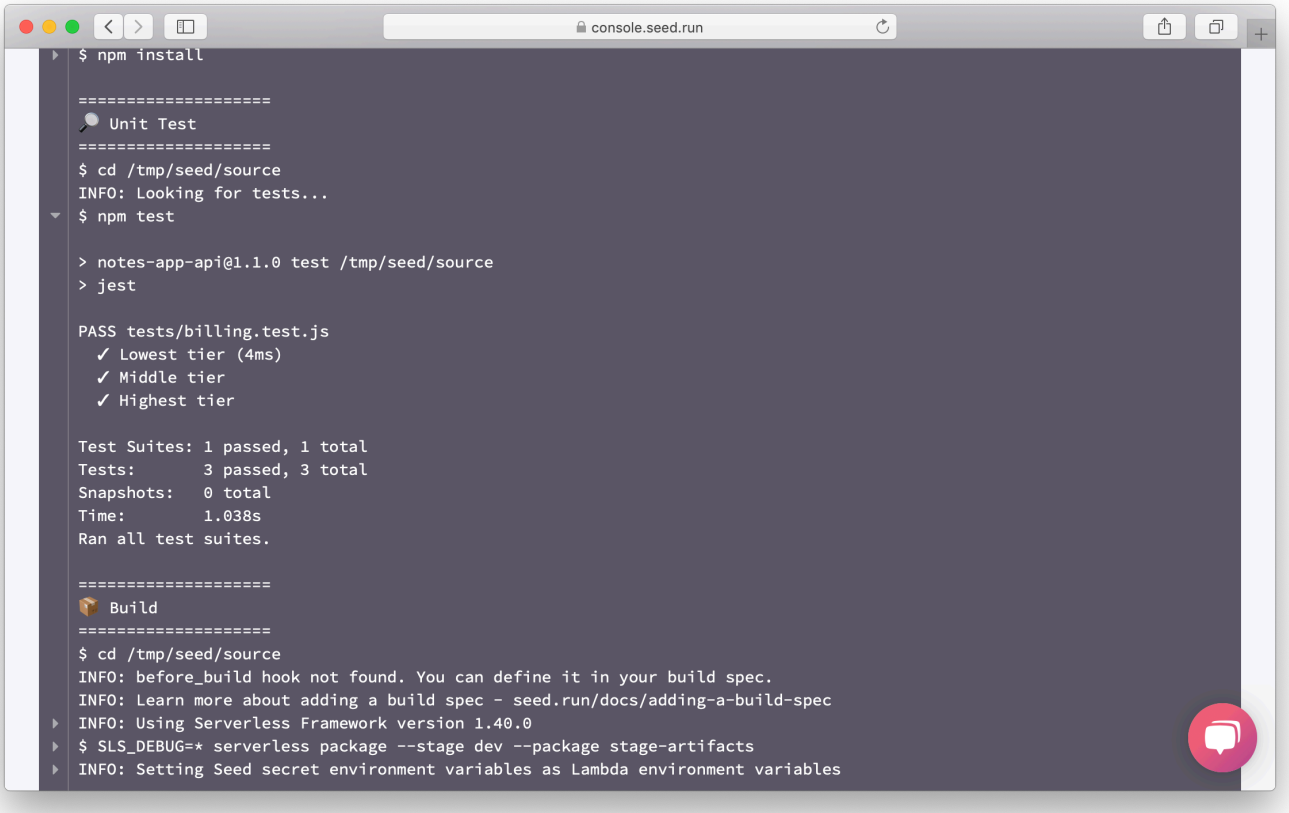
Here you'll see the build taking place live. Click on the service that is being deployed. In this case, we only have one service.



You'll see the build logs for the in progress build here.



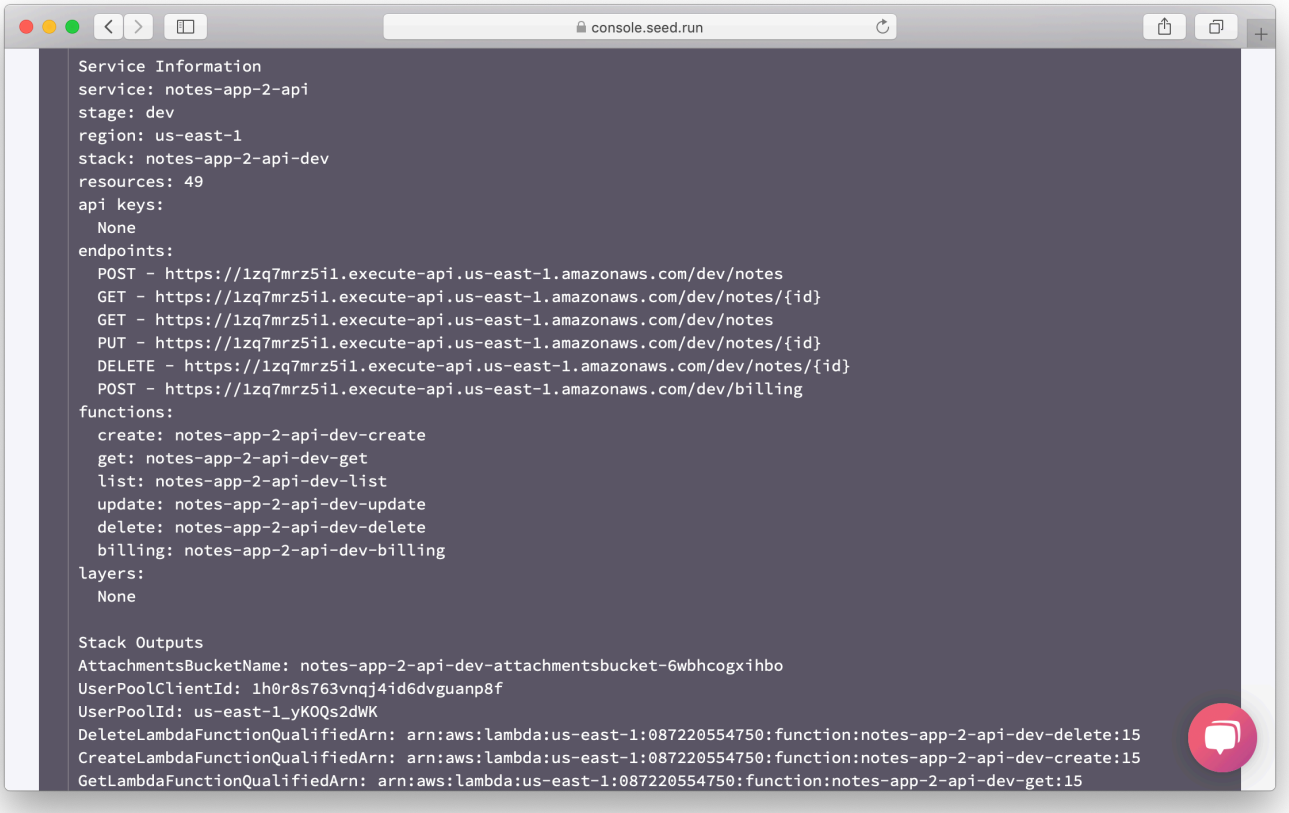
Notice the tests are being run as a part of the build.



Once the build is complete, take a look at the build log and make a note of the following:

- Region: `region`
- Cognito User Pool Id: `UserPoolId`
- Cognito App Client Id: `UserPoolClientId`
- Cognito Identity Pool Id: `IdentityPoolId`
- S3 File Uploads Bucket: `AttachmentsBucketName`
- API Gateway URL: `ServiceEndpoint`

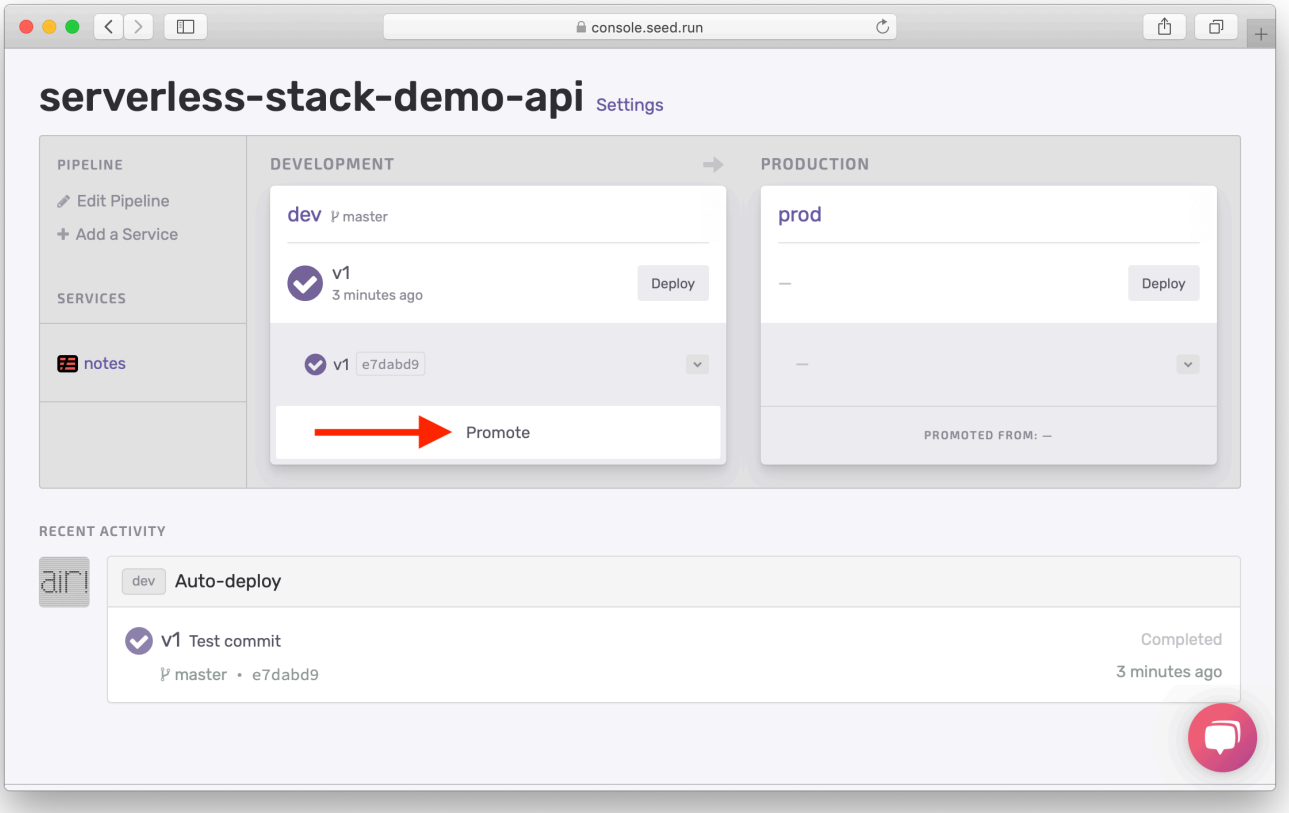
We'll be needing these later in our frontend and when we test our APIs.



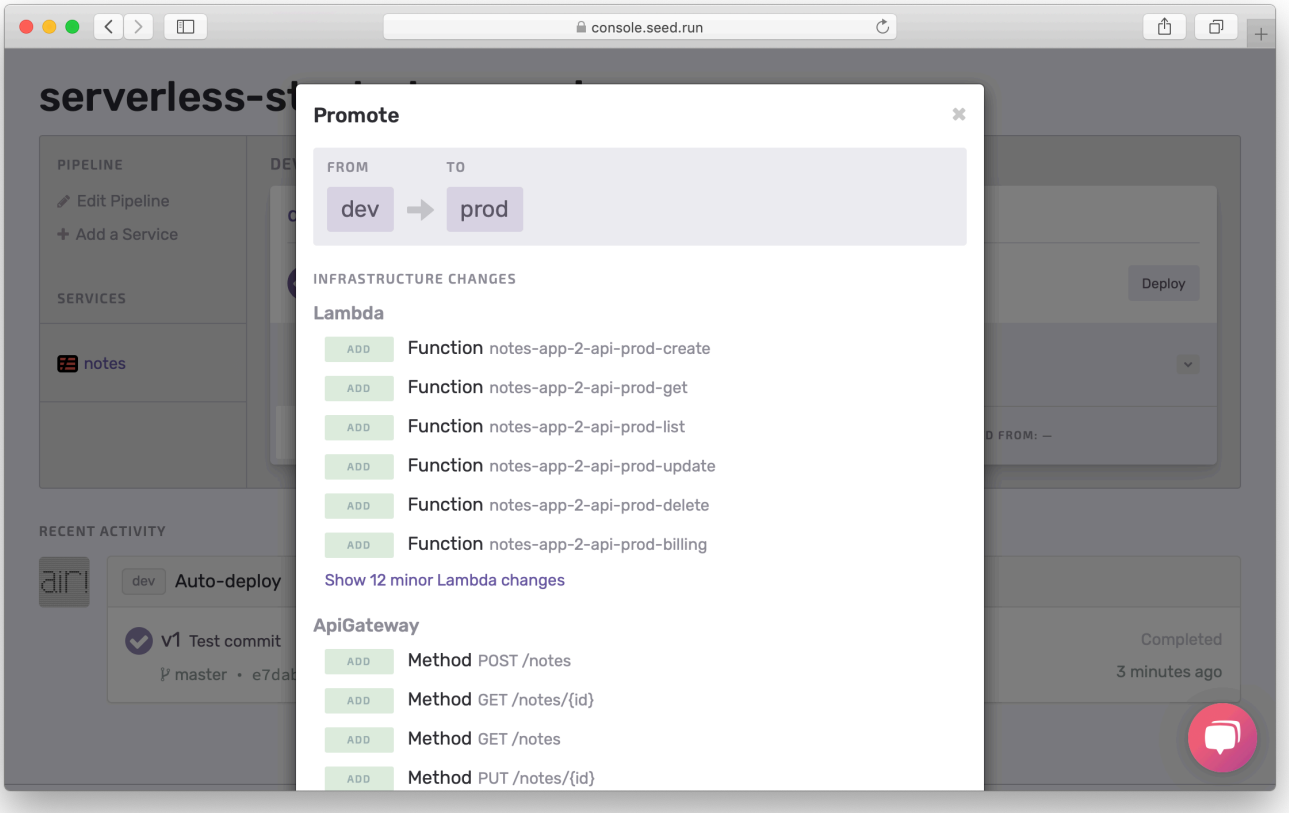
Now head over to the app home page. You'll notice that we are ready to promote to production.

We have a manual promotion step so that you get a chance to review the changes and ensure that you are ready to push to production.

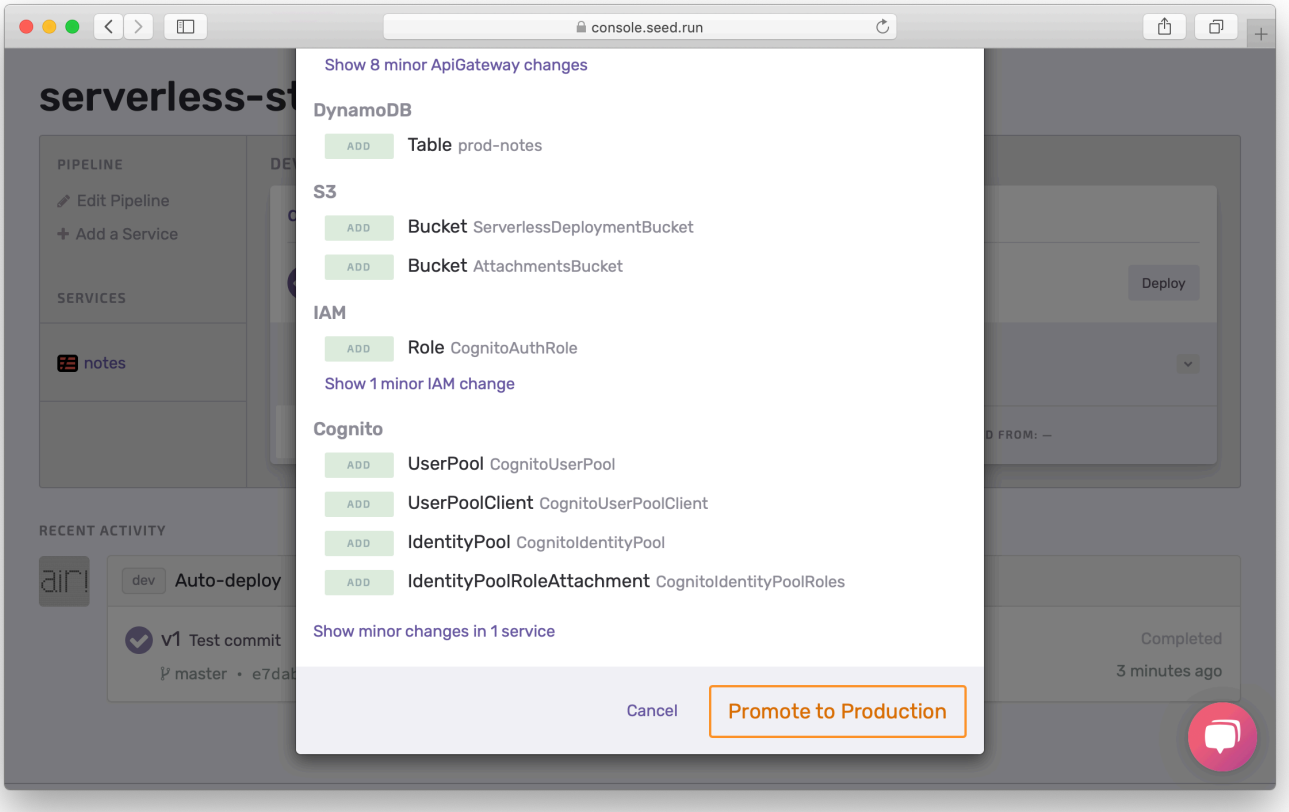
Hit the **Promote** button.



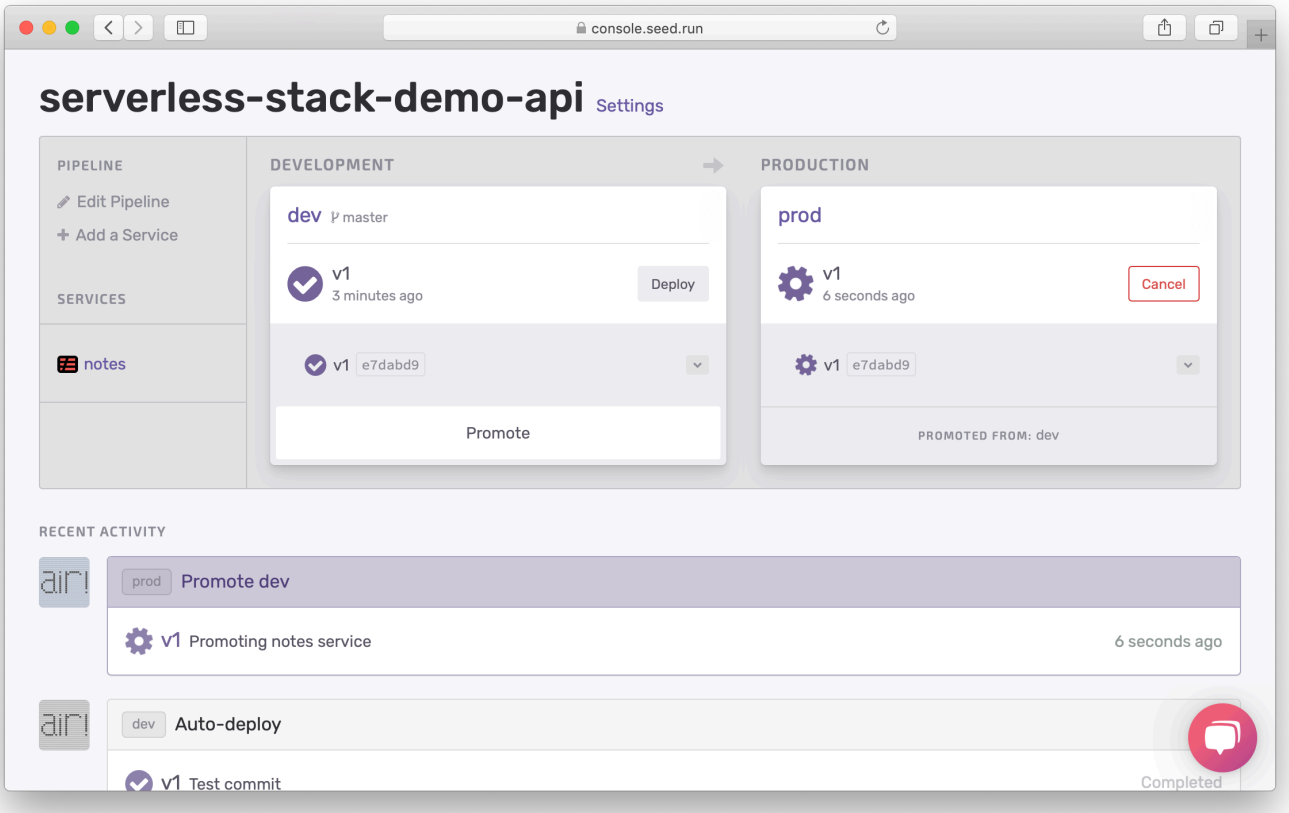
This brings up a dialog that will generate a Change Set. It compares the resources that are being updated with respect to what you have in production. It's a great way to compare the infrastructure changes that are being promoted.



Scroll down and hit **Promote to Production**.

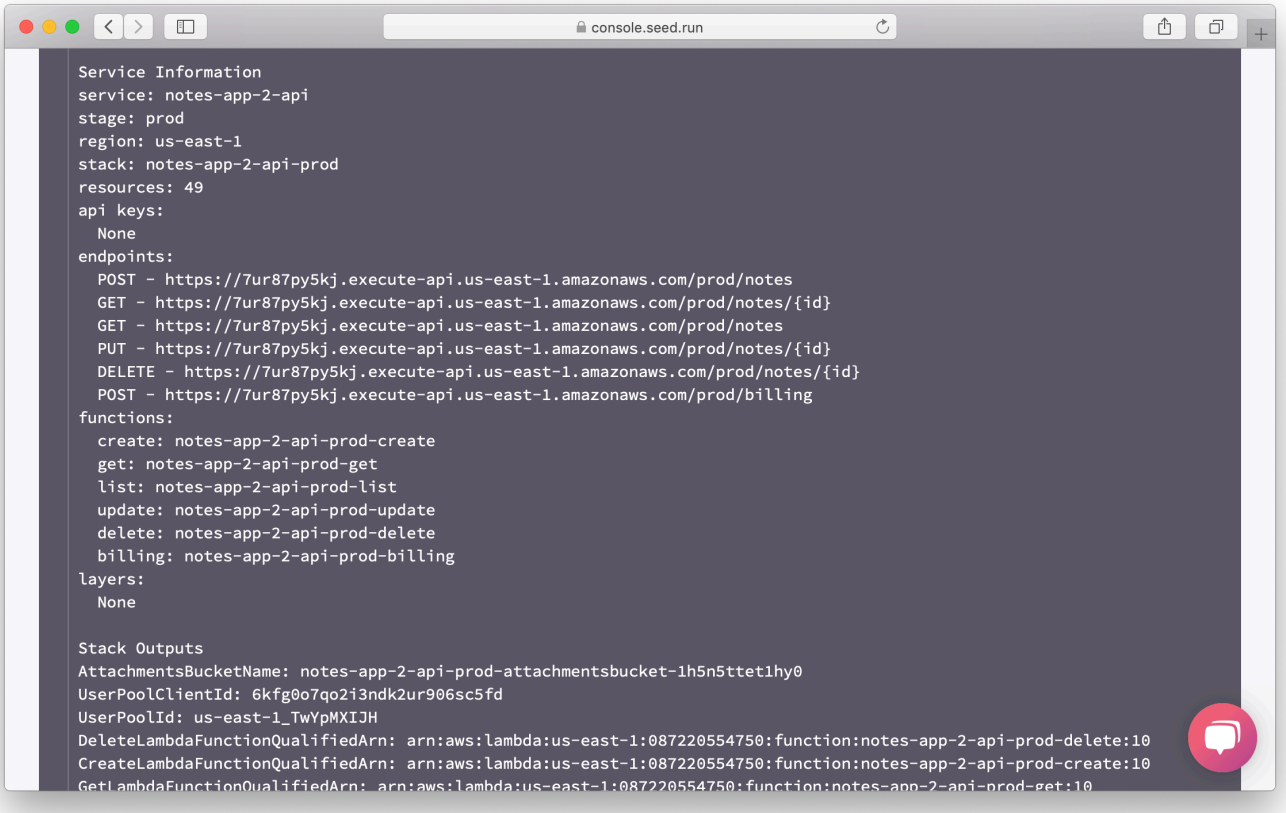


You'll notice that the build is being promoted to the **prod** stage.



And if you head over to the **prod** stage, you should see your prod deployment in action. It should take a second to deploy to production. And just like before, make a note of the following.

- Region: `region`
- Cognito User Pool Id: `UserPoolId`
- Cognito App Client Id: `UserPoolClientId`
- Cognito Identity Pool Id: `IdentityPoolId`
- S3 File Uploads Bucket: `AttachmentsBucketName`
- API Gateway URL: `ServiceEndpoint`



Next let's configure our serverless API with a custom domain.

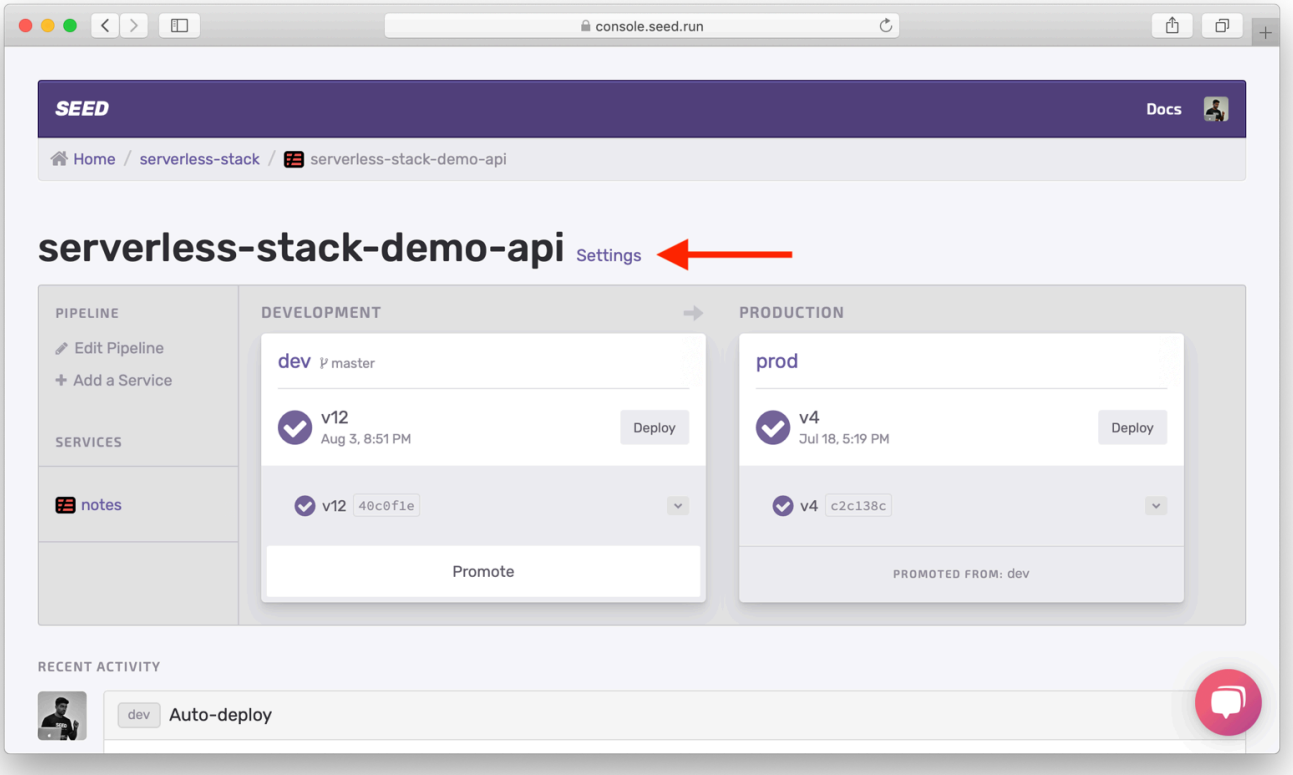
For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploying-through-seed/177)

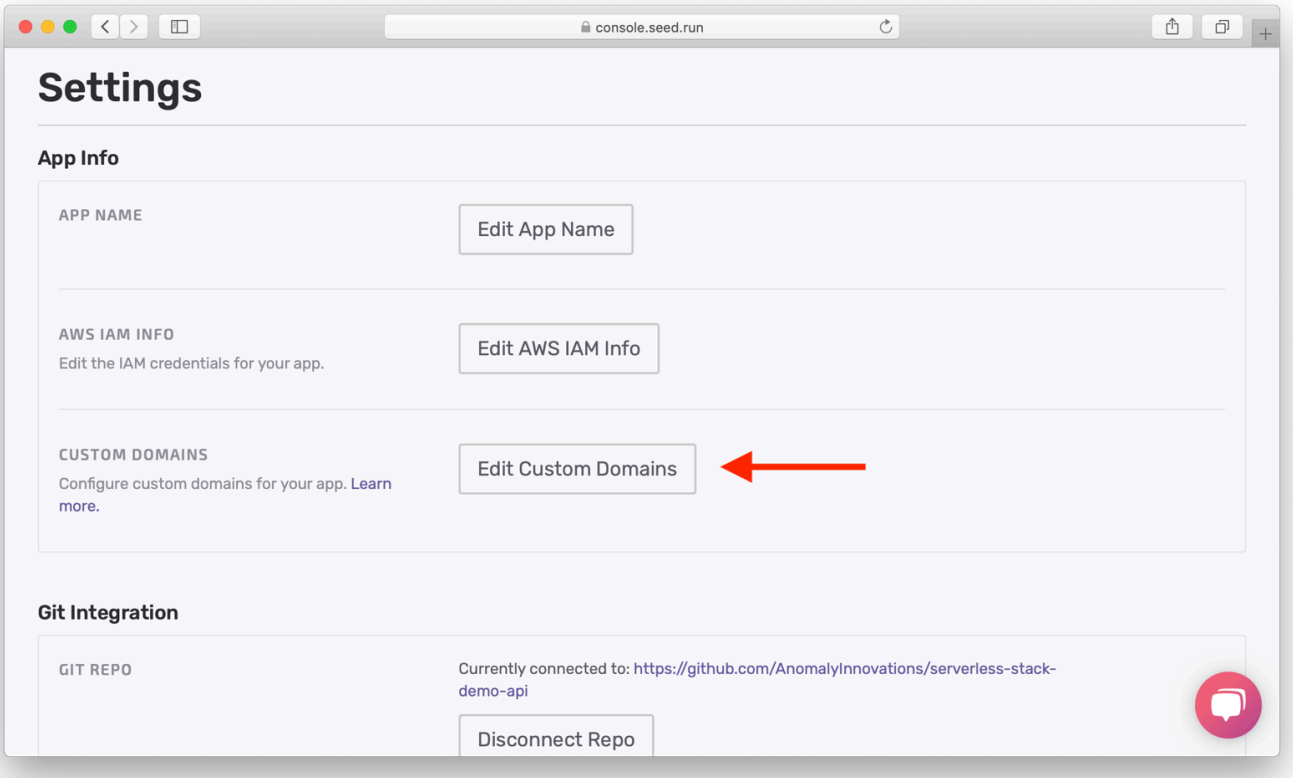
Set Custom Domains Through Seed

Our serverless API uses API Gateway and it gives us some auto-generated endpoints. We would like to configure them to use a scheme like `api.my-domain.com` or something similar. This can take a few different steps through the AWS Console, but it is pretty straightforward to configure through Seed (https://seed.run).

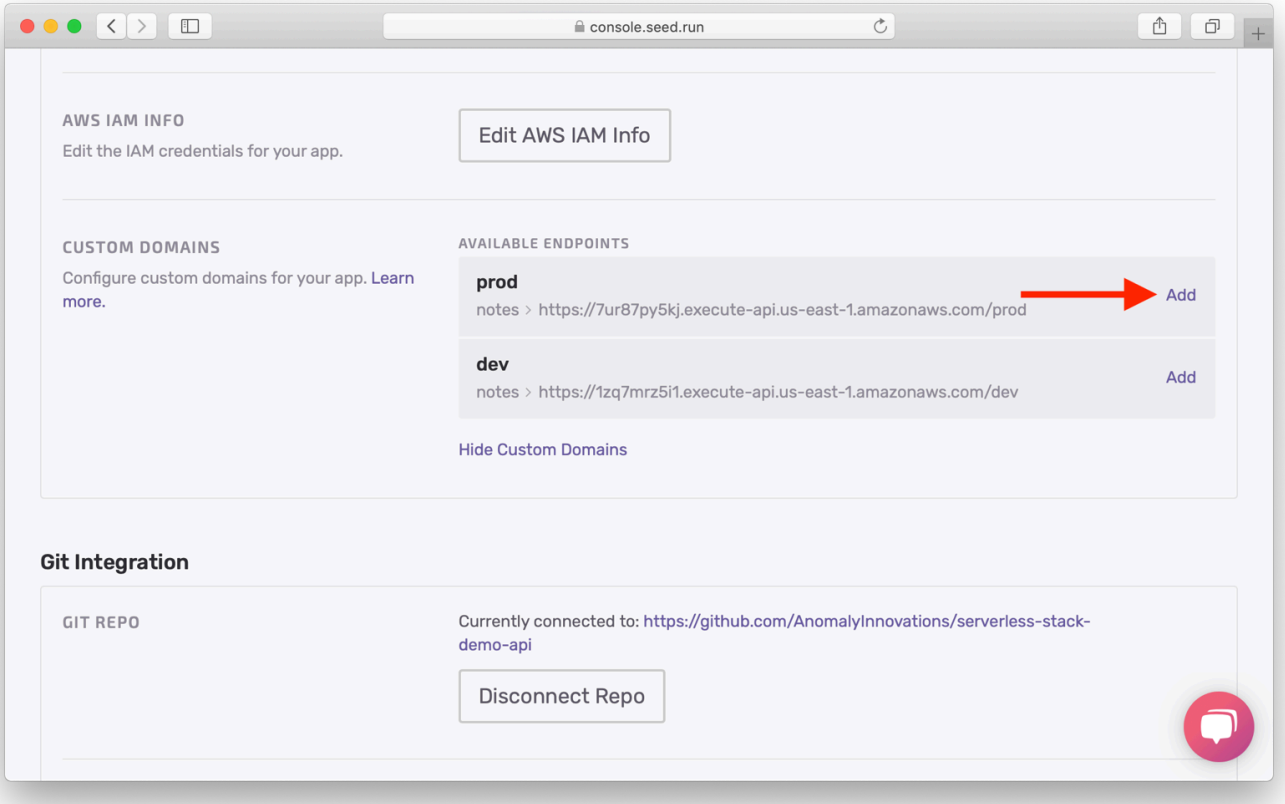
Head over to our app settings.



Here click on **Edit Custom Domains**.



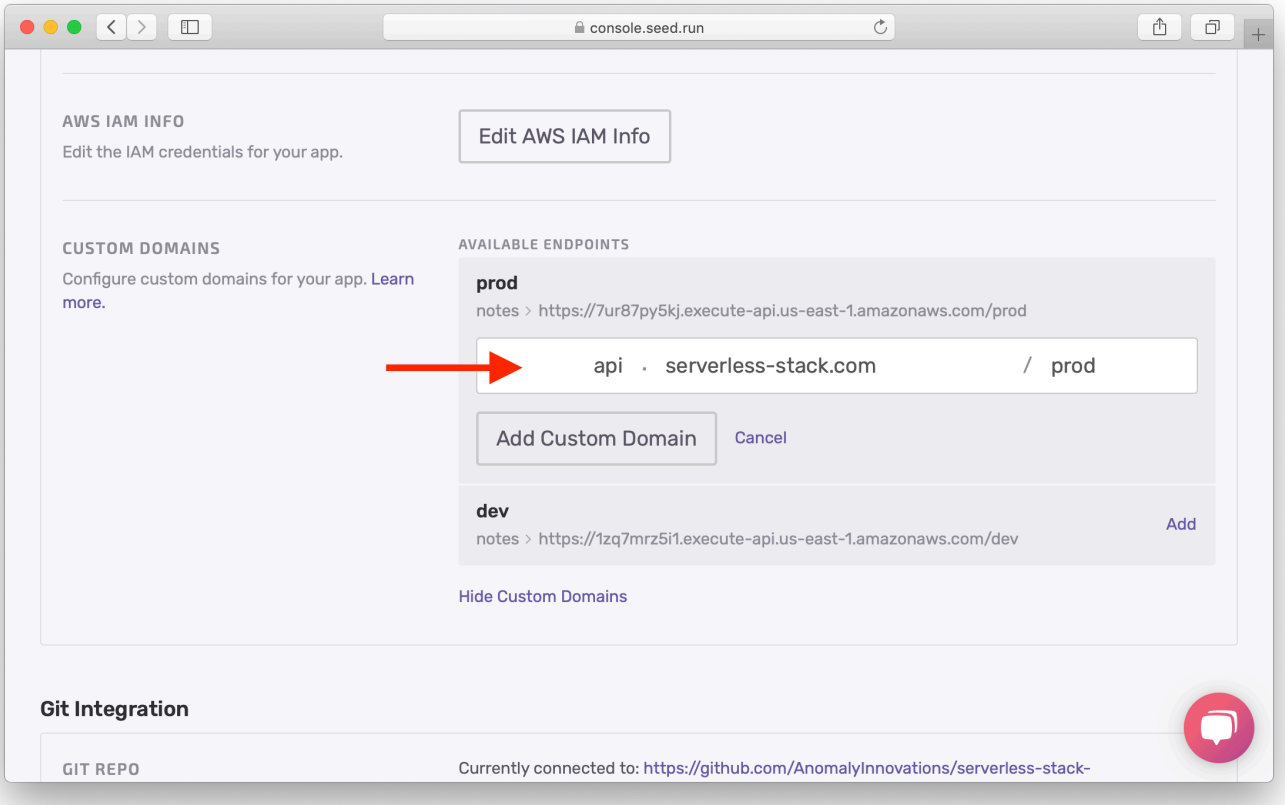
And click **Add** for our production endpoint.



In the first part of the tutorial we had added our domain to Route 53. If

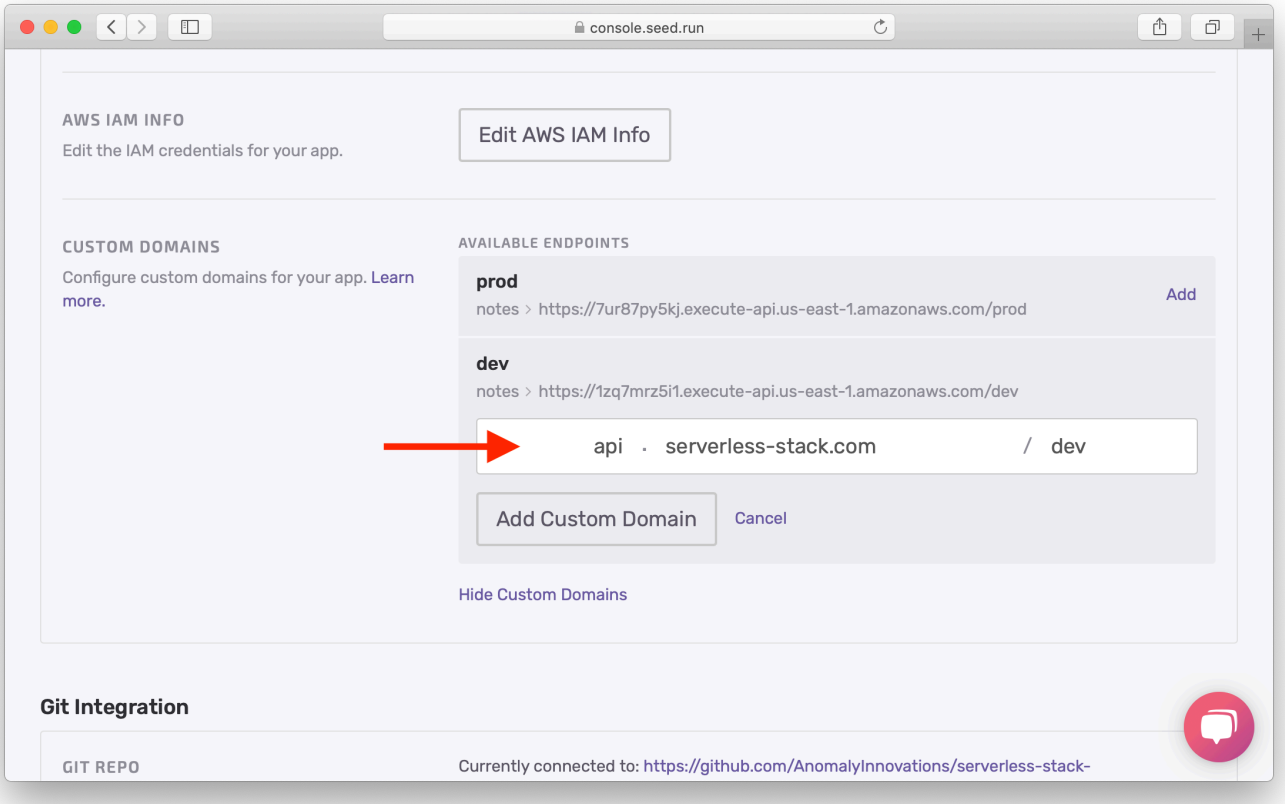
you haven't done so you can read more about it here (https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/MigratingDNS.html). Hit **Select a domain** and you should see a list of all your Route 53 domains. Select the one you intend to use. And fill in the sub-domain and base path. For example, you could use `api.my-domain.com/prod`; where `api` is the sub-domain and `prod` is the base path.

And hit **Add Custom Domain**.



Seed will now go through and configure the domain for this API Gateway endpoint, create the SSL certificate and attach it to the domain. This process can take up to 40 mins.

While we wait, we can do the same for our `dev` endpoint. Select the domain, sub-domain, and base path. In our case we'll use something like `api.my-domain.com/dev`.



Hit **Add Custom Domain** and wait for the changes to take place.

Now that we've automated our deployments, let's do a quick test to see what will happen if we make a mistake and push some faulty code to production.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/set-custom-domains-through-seed/178)

Test the Configured APIs

Now we have two sets of APIs (prod and dev), let's quickly test them to make sure they are working fine before we plug our frontend into them. Back in the Test the APIs (/chapters/test-the-apis.html) chapter, we used a simple utility called AWS API Gateway Test CLI (https://github.com/AnomalyInnovations/aws-api-gateway-cli-test).

Before we do the test let's create a test user for both the environments. We'll be following the exact same steps as the Create a Cognito test user (/chapters/create-a-cognito-test-user.html) chapter.

Create Test Users

We are going to use the AWS CLI for this.

CHANGE In your terminal, run.

```
$ aws cognito-idp sign-up \
  --region YOUR_DEV_COGNITO_REGION \
  --client-id YOUR_DEV_COGNITO_APP_CLIENT_ID \
  --username admin@example.com \
  --password Passw0rd!
```

Refer back to the Deploying through Seed (/chapters/deploying-through-seed.html) chapter to look up the **dev** version of your Cognito App Client Id. And replace `YOUR_DEV_COGNITO_REGION` with the region that you deployed to.

✦ CHANGE Next we'll confirm the user through the Cognito Admin CLI.

```
$ aws cognito-idp admin-confirm-sign-up \
  --region YOUR_DEV_COGNITO_REGION \
  --user-pool-id YOUR_DEV_COGNITO_USER_POOL_ID \
  --username admin@example.com
```

Again, replace `YOUR_DEV_COGNITO_USER_POOL_ID` with the **dev** version of your Cognito User Pool Id from the Deploying through Seed (/chapters/deploying-through-seed.html) chapter and the region from the previous command.

Let's quickly do the same with **prod** versions as well.

✦ CHANGE In your terminal, run.

```
$ aws cognito-idp sign-up \
  --region YOUR_PROD_COGNITO_REGION \
  --client-id YOUR_PROD_COGNITO_APP_CLIENT_ID \
  --username admin@example.com \
  --password Passw0rd!
```

Here use your prod version of your Cognito details.

✦ CHANGE And confirm the user.

```
$ aws cognito-idp admin-confirm-sign-up \
  --region YOUR_PROD_COGNITO_REGION \
  --user-pool-id YOUR_PROD_COGNITO_USER_POOL_ID \
  --username admin@example.com
```

Make sure to use the prod versions here as well.

Now we are ready to test our APIs.

Test the API

Let's test our dev endpoint. Run the following command:

```
$ npx aws-api-gateway-cli-test \
--username='admin@example.com' \
--password='Passw0rd!' \
--user-pool-id='YOUR_DEV_COGNITO_USER_POOL_ID' \
--app-client-id='YOUR_DEV_COGNITO_APP_CLIENT_ID' \
--cognito-region='YOUR_DEV_COGNITO_REGION' \
--identity-pool-id='YOUR_DEV_IDENTITY_POOL_ID' \
--invoke-url='YOUR_DEV_API_GATEWAY_URL' \
--api-gateway-region='YOUR_DEV_API_GATEWAY_REGION' \
--path-template='/notes' \
--method='POST' \
--body='{"content":"hello
world","attachment":"hello.jpg"}'
```

Refer back to the Deploying through Seed (/chapters/deploying-through-seed.html) chapter for these:

- `YOUR_DEV_COGNITO_USER_POOL_ID` and `YOUR_DEV_COGNITO_APP_CLIENT_ID` are all related to your Cognito User Pool.
- `YOUR_DEV_IDENTITY_POOL_ID` is for your Cognito Identity Pool.
- And `YOUR_DEV_API_GATEWAY_URL` is your API Gateway endpoint. It looks something likes this `https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/dev`. But if you have configured it with a custom domain use the one from the Set custom domains through

Seed (/chapters/set-custom-domains-through-seed.html) chapter.

- Finally, the `YOUR_DEV_API_GATEWAY_REGION` and `YOUR_DEV_COGNITO_REGION` is the region you deployed to. In our case it is `us-east-1`.

If the command is successful, it'll look something like this.

```
Authenticating with User Pool
Getting temporary credentials
Making API request
{ status: 200,
  statusText: 'OK',
  data:
   { userId: 'us-east-1:9bdc031d-ee9e-4ffa-9a2d-
123456789',
     noteId: '8f7da030-650b-11e7-a661-123456789',
     content: 'hello world',
     attachment: 'hello.jpg',
     createdAt: 1499648598452 } }
```

Also run the same command for prod. Make sure to use the prod versions.

```
$ npx aws-api-gateway-cli-test \
--username='admin@example.com' \
--password='Passw0rd!' \
--user-pool-id='YOUR_PROD_COGNITO_USER_POOL_ID' \
--app-client-id='YOUR_PROD_COGNITO_APP_CLIENT_ID' \
--cognito-region='YOUR_PROD_COGNITO_REGION' \
--identity-pool-id='YOUR_PROD_IDENTITY_POOL_ID' \
--invoke-url='YOUR_PROD_API_GATEWAY_URL' \
--api-gateway-region='YOUR_PROD_API_GATEWAY_REGION' \
```

```
--path-template='/notes' \
--method='POST' \
--body='{"content":"hello
world","attachment":"hello.jpg"}'
```

And you should see it give a similar output as dev.

Now that our APIs our tested we are ready to plug these into our frontend. Before we do that, let's quickly go over what happens if we push some faulty code to our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/test-the-configured-apis/179)

Monitoring Deployments in Seed

Despite our best intentions we might run into cases where some faulty code ends up in production. We want to make sure we have a plan for that. Let's go through what this would look like in Seed (https://seed.run).

Push Some Faulty Code

Let's start by pushing an obvious mistake.

✦ CHANGE Add the following to `functions/create.js` as the first line of our `main` function.

```
uuid.abc.gibberish;
```

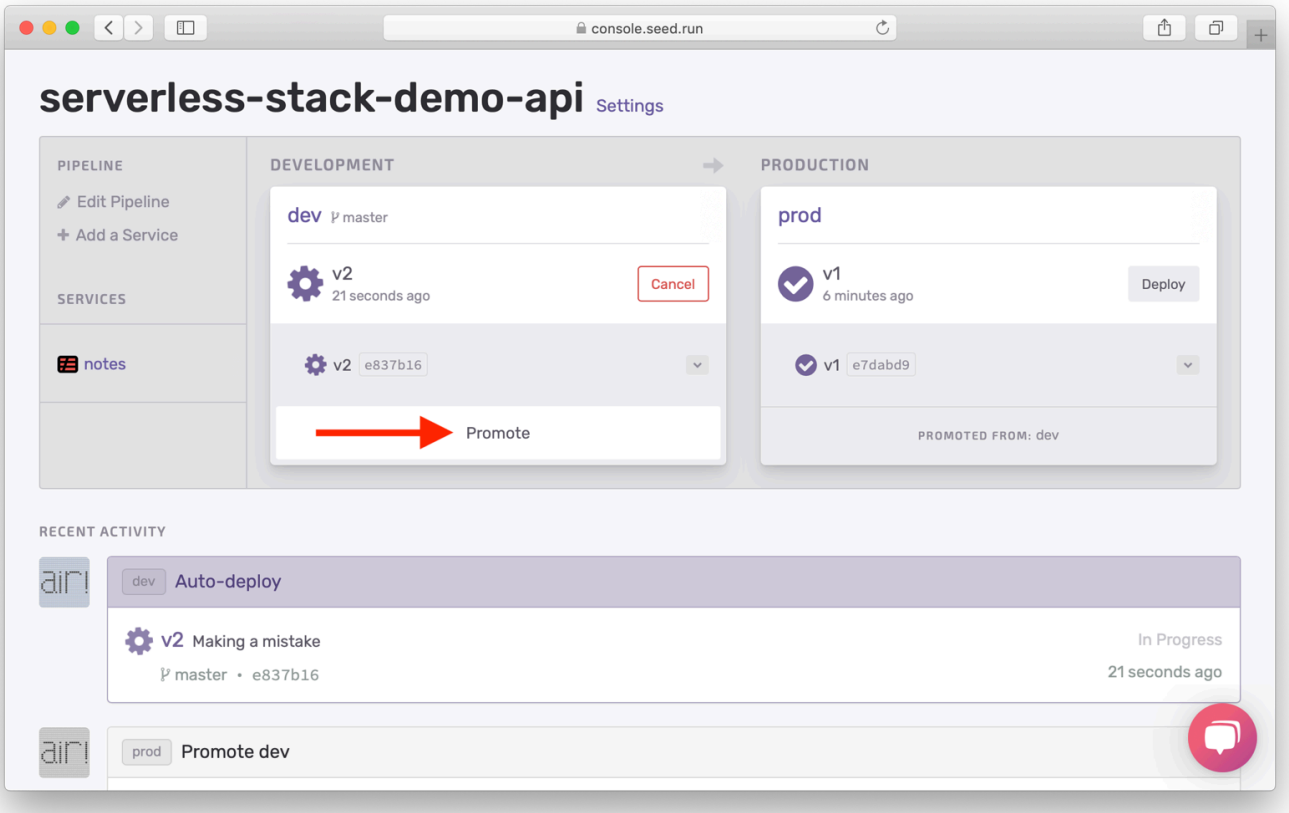
Since there is no property `abc.gibberish` in `uuid`, this code should fail.

✦ CHANGE Let's commit and push this to dev.

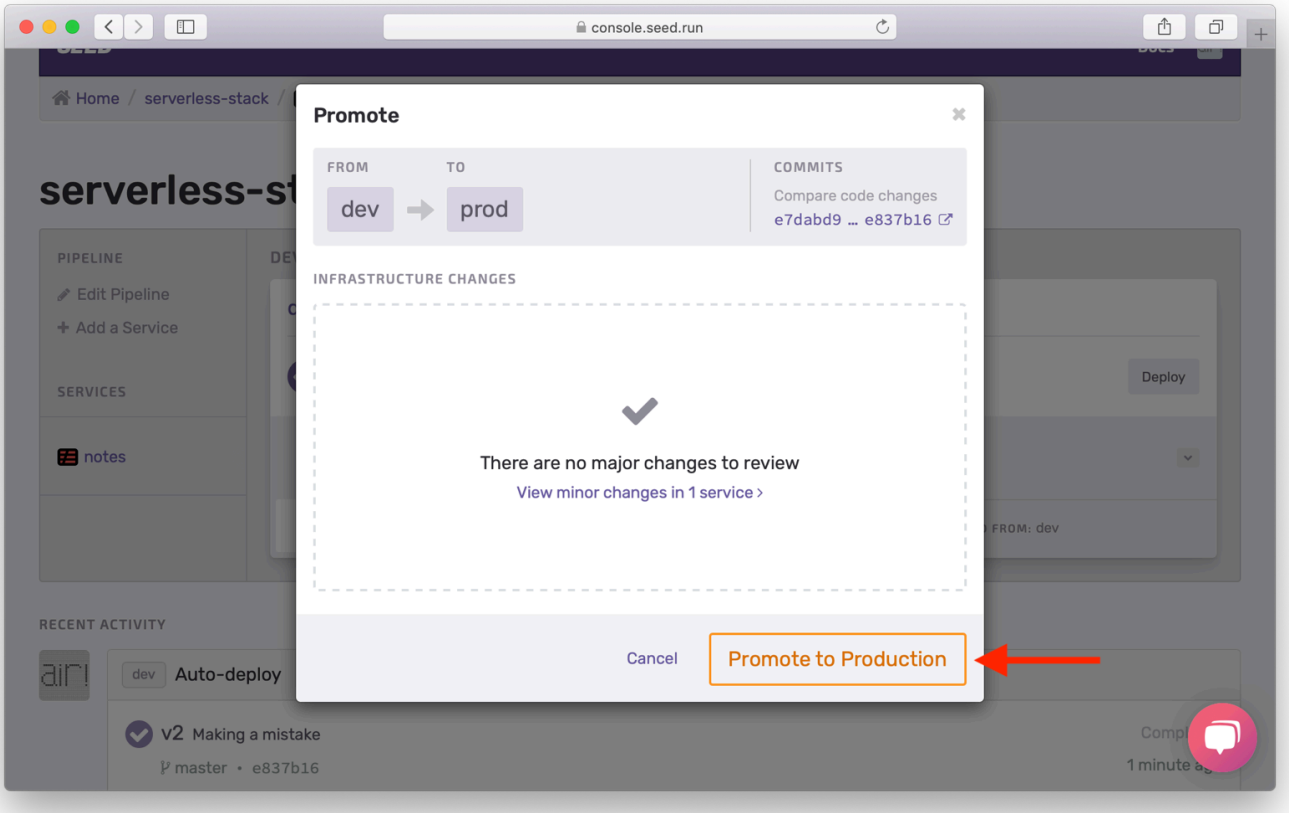
```
$ git add .
$ git commit -m "Making a mistake"
$ git push
```

Now you can see a build in progress. Wait for it to complete and hit

Promote.

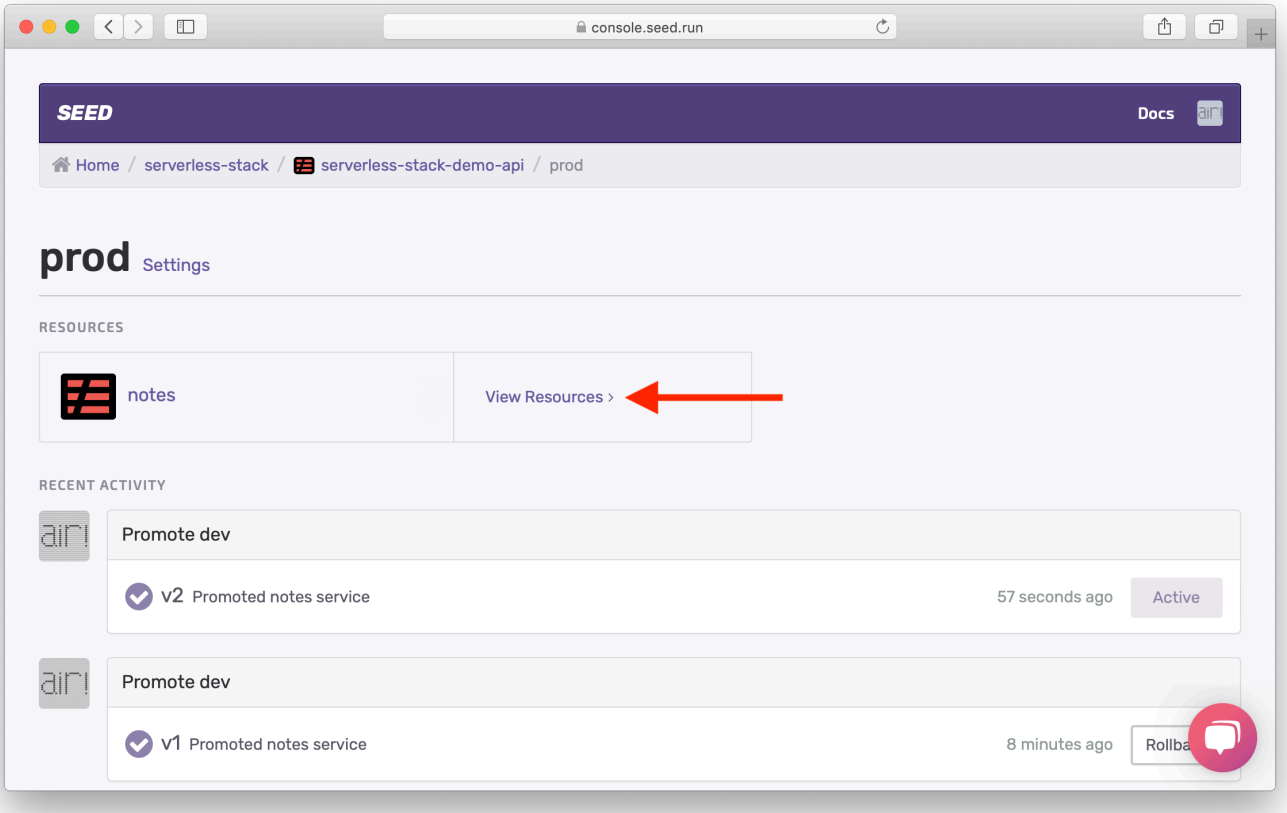


Confirm the Change Set by hitting **Promote to Production**.

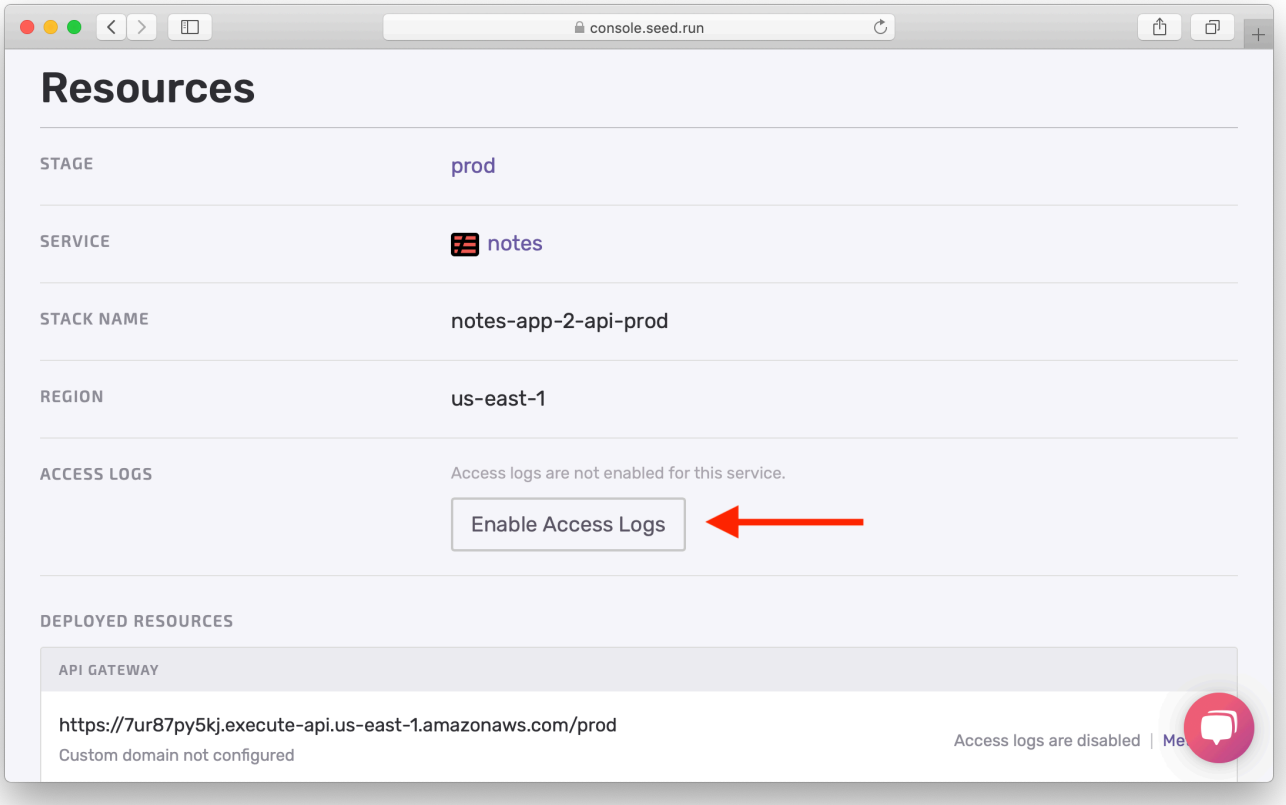


Enable Access Logs

Now before we test our faulty code, we'll turn on API Gateway access logs so we can see the error. Click on the **prod** stage **View Resources**.



Hit **Enable Access Logs**.



This will take a couple of minutes but Seed will automatically configure the IAM roles necessary for this and enable API Gateway access logs for your prod environment.

Test the Faulty Code

Now to test our code, run the same command from the last chapter (/chapters/test-the-configured-apis.html) to test our API.

```
$ npx aws-api-gateway-cli-test \
--username='admin@example.com' \
--password='Passw0rd!' \
--user-pool-id='YOUR_PROD_COGNITO_USER_POOL_ID' \
--app-client-id='YOUR_PROD_COGNITO_APP_CLIENT_ID' \
--cognito-region='YOUR_PROD_COGNITO_REGION' \
--identity-pool-id='YOUR_PROD_IDENTITY_POOL_ID' \
--invoke-url='YOUR_PROD_API_GATEWAY_URL' \
--api-gateway-region='YOUR_PROD_API_GATEWAY_REGION' \
--path-template='/notes' \
--method='POST' \
--body='{"content":"hello
world","attachment":"hello.jpg"}'
```

Make sure to use the prod version of your resources.

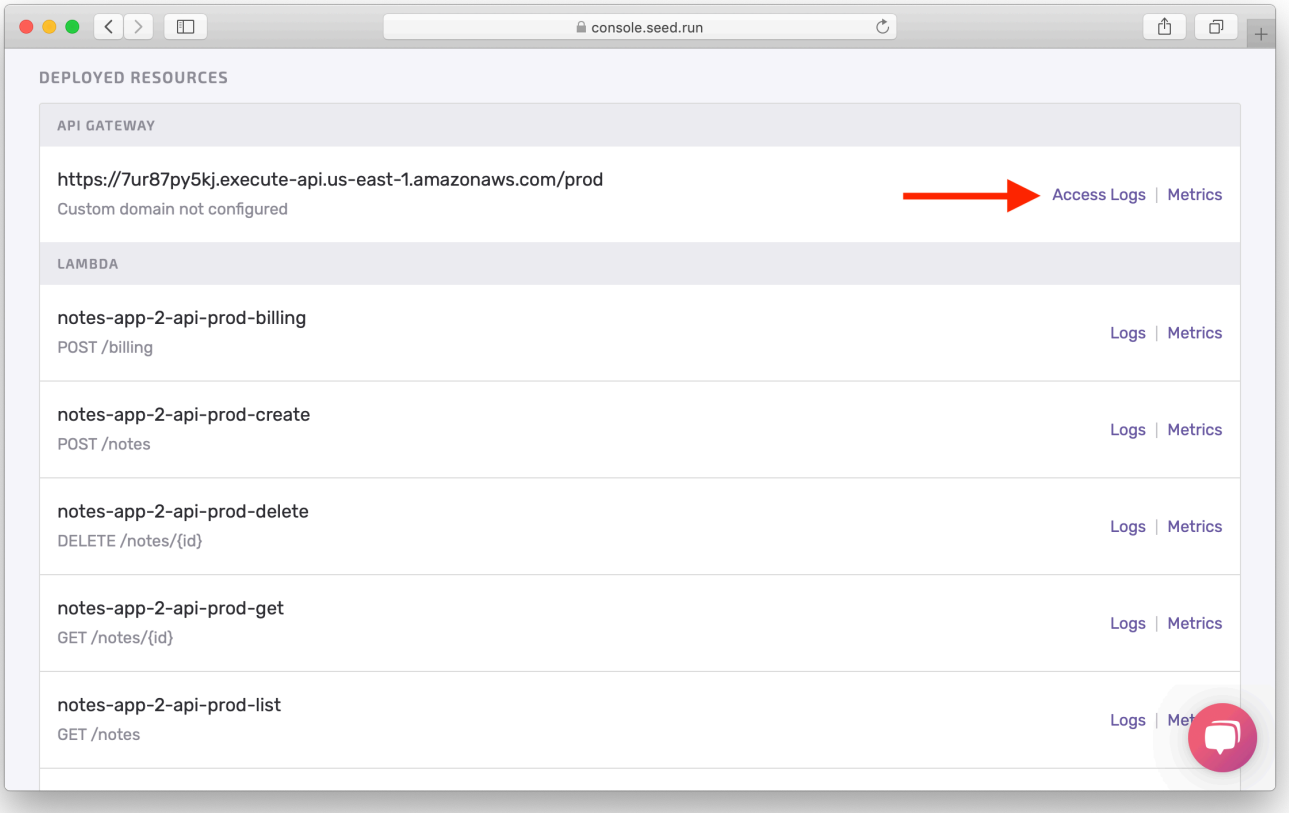
You should see an error that looks something like this.

```
Authenticating with User Pool
Getting temporary credentials
Making API request
{ status: 502,
  statusText: 'Bad Gateway',
```

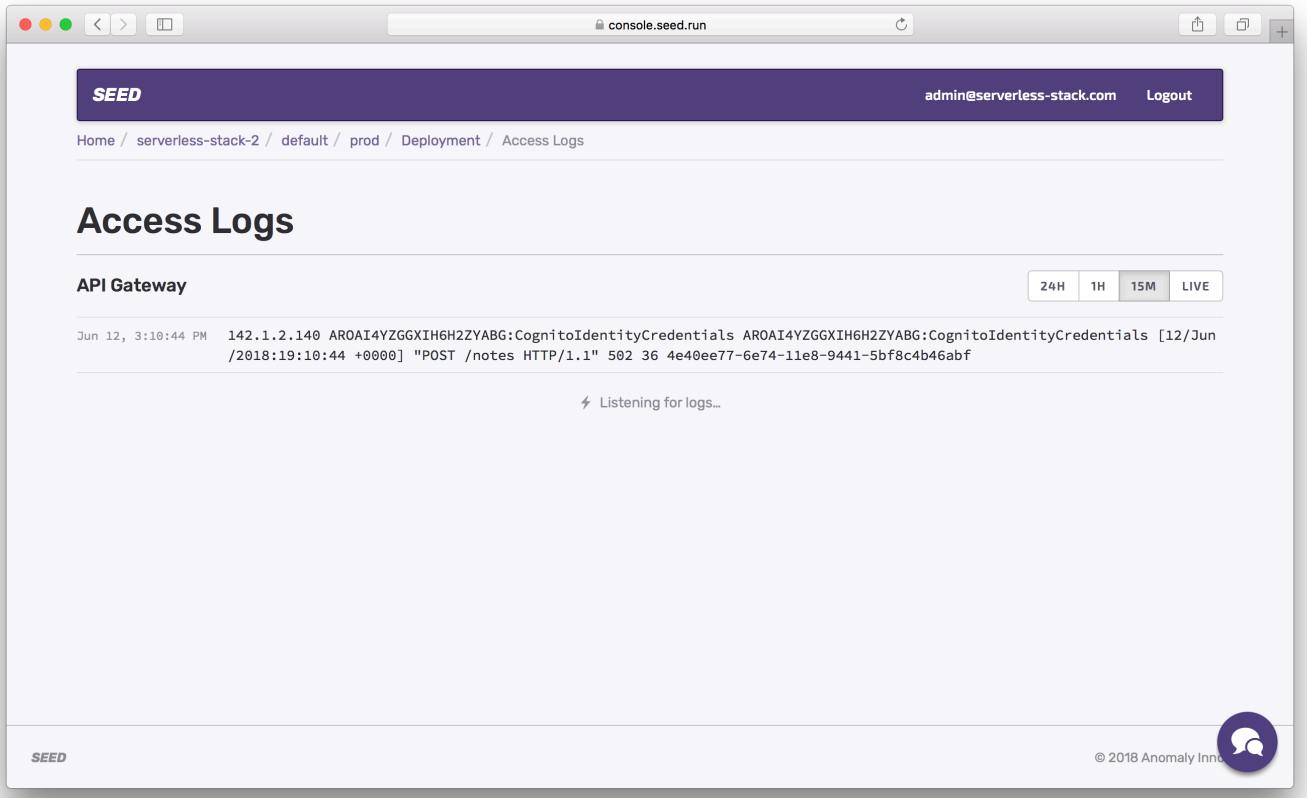
```
  data: { message: 'Internal server error' } }
```

View Logs and Metrics

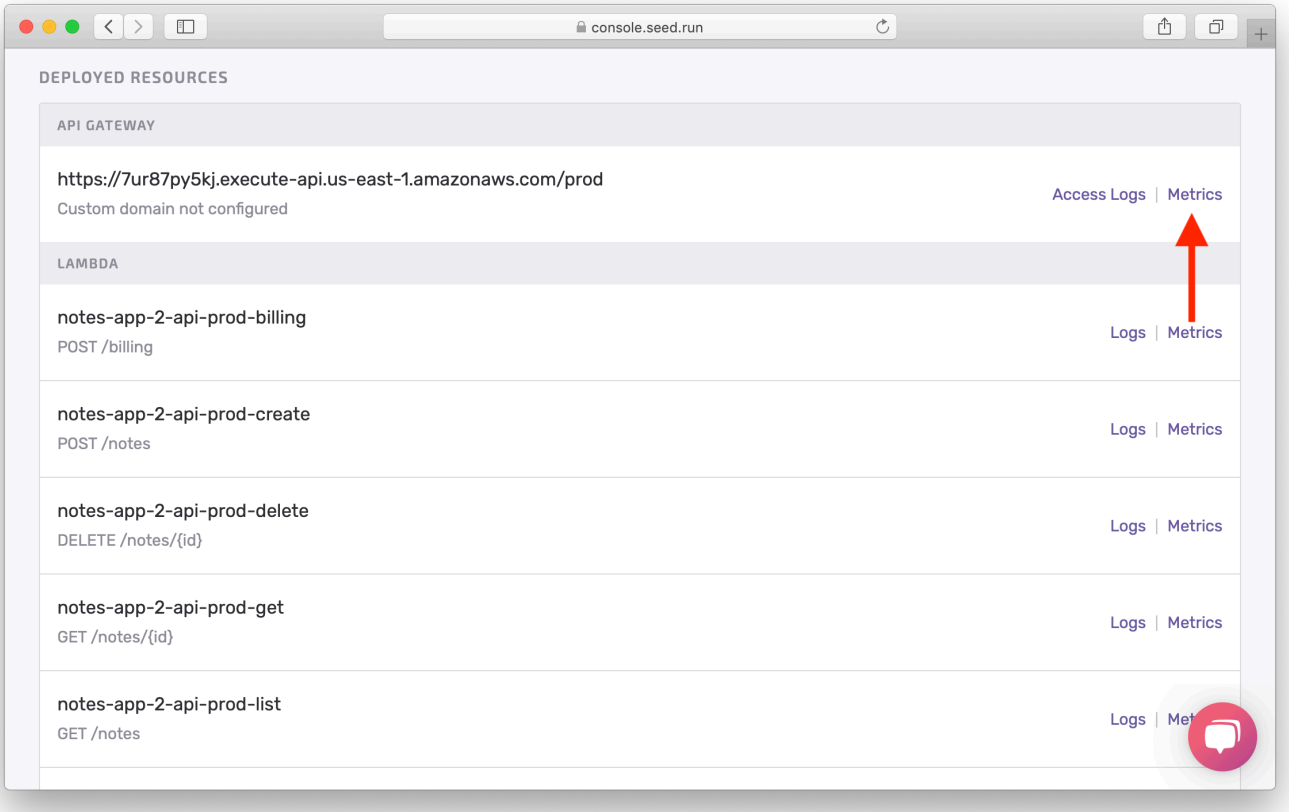
Back in the Seed console, you should be able to click on **Access Logs**.



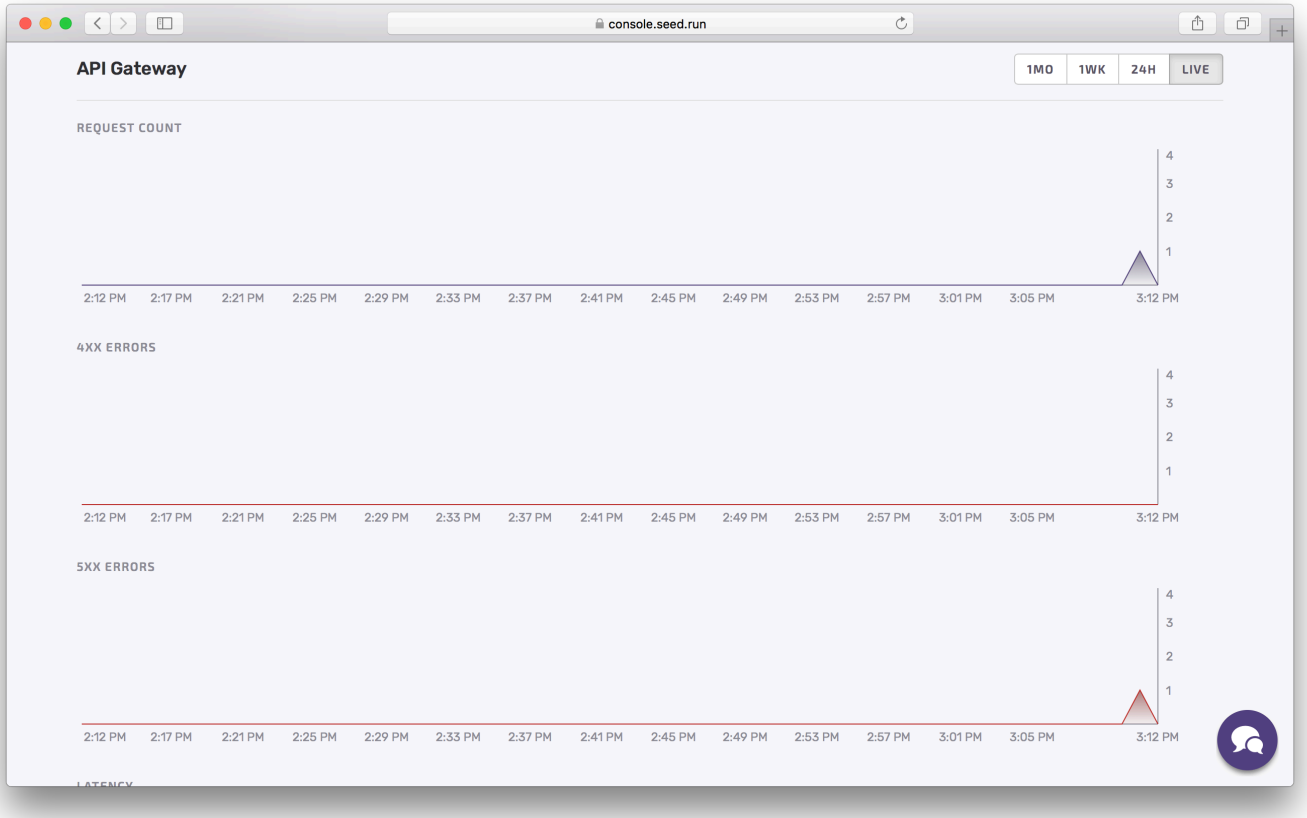
This should show you that there was a `502` error on a recent request.



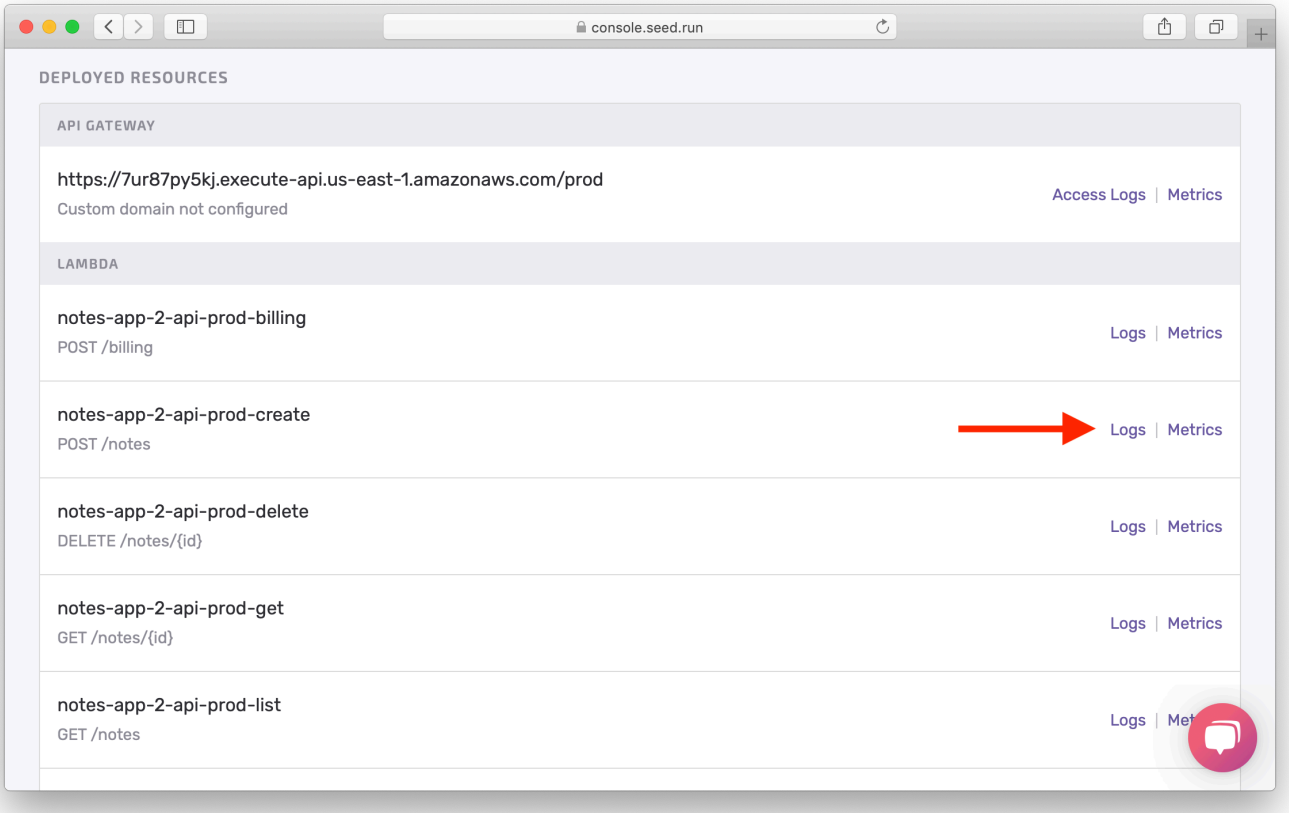
If you go back, you can click on **Metrics** to get a good overview of our requests.



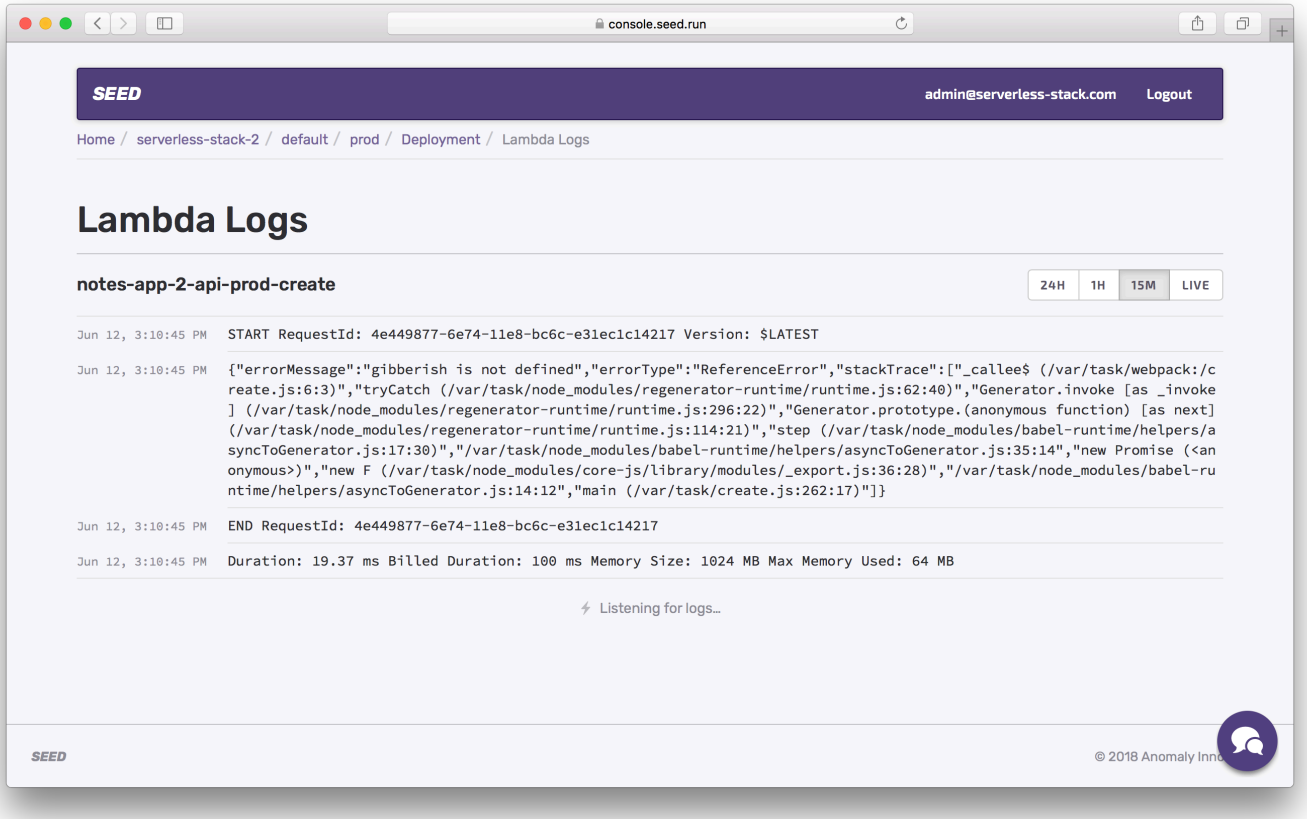
You'll notice the number of requests that were made, 4xx errors, 5xx error, and latency for those requests.



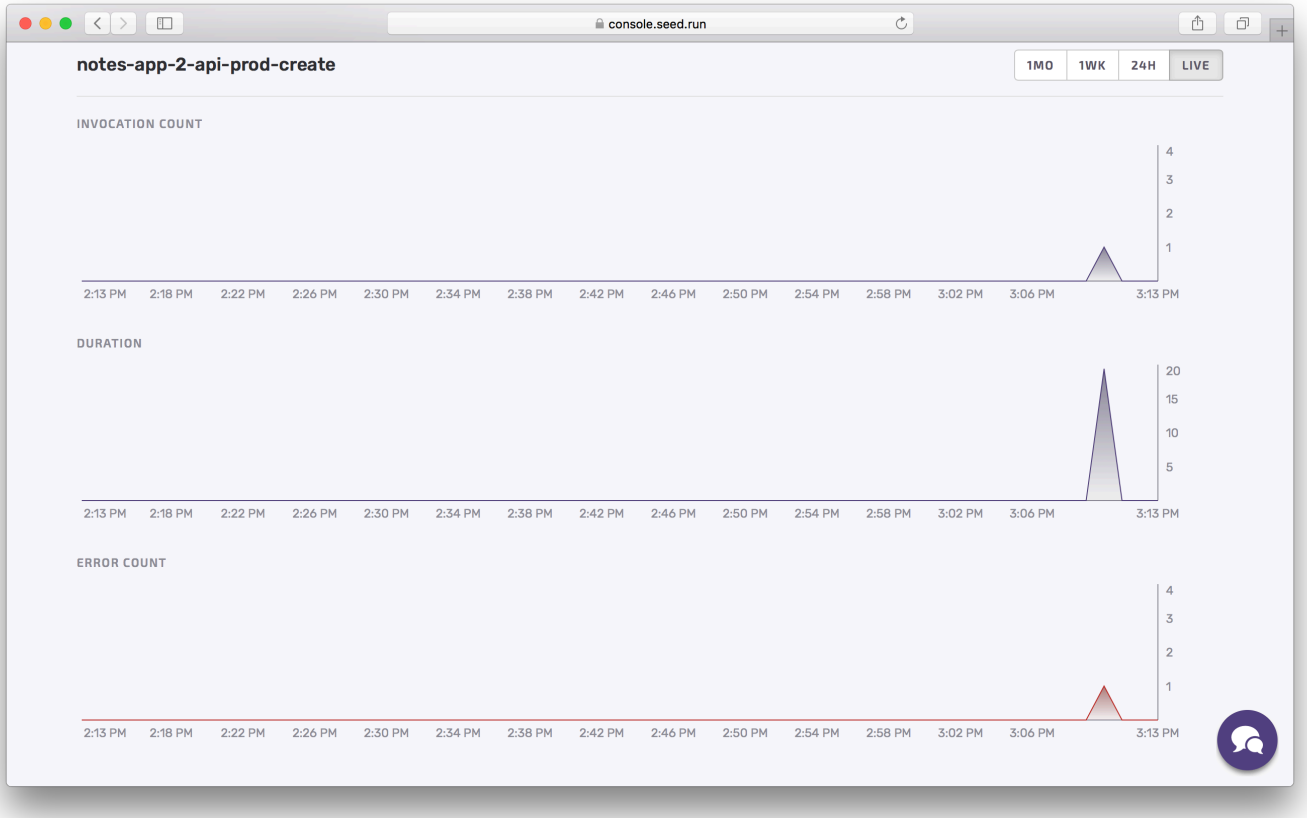
Now if we go back and click on the **Logs** for the **create** Lambda function.



This should show you clearly that there was an error in our code. Notice, that it is complaining that `gibberish` is not defined.



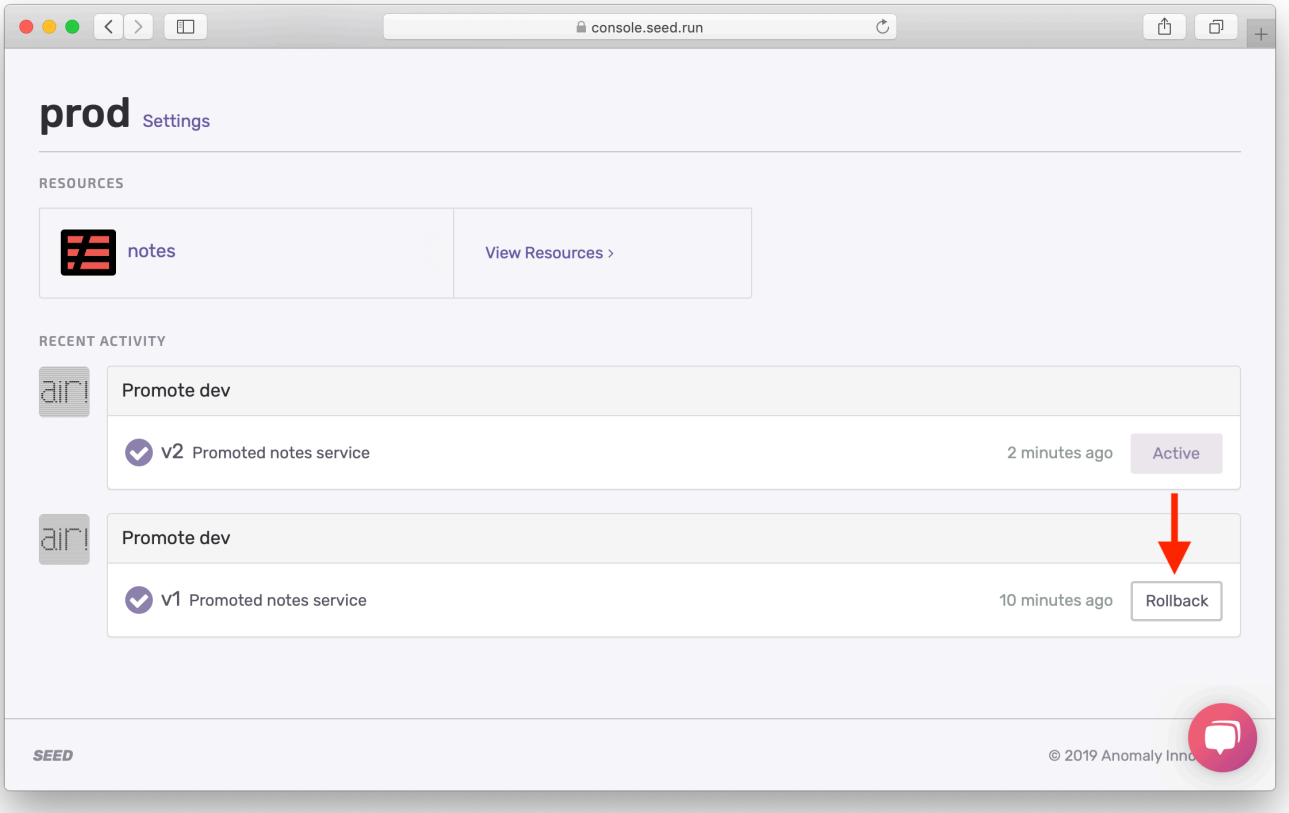
And just like the API metrics, the Lambda metrics will show you an overview of what is going on at a function level.



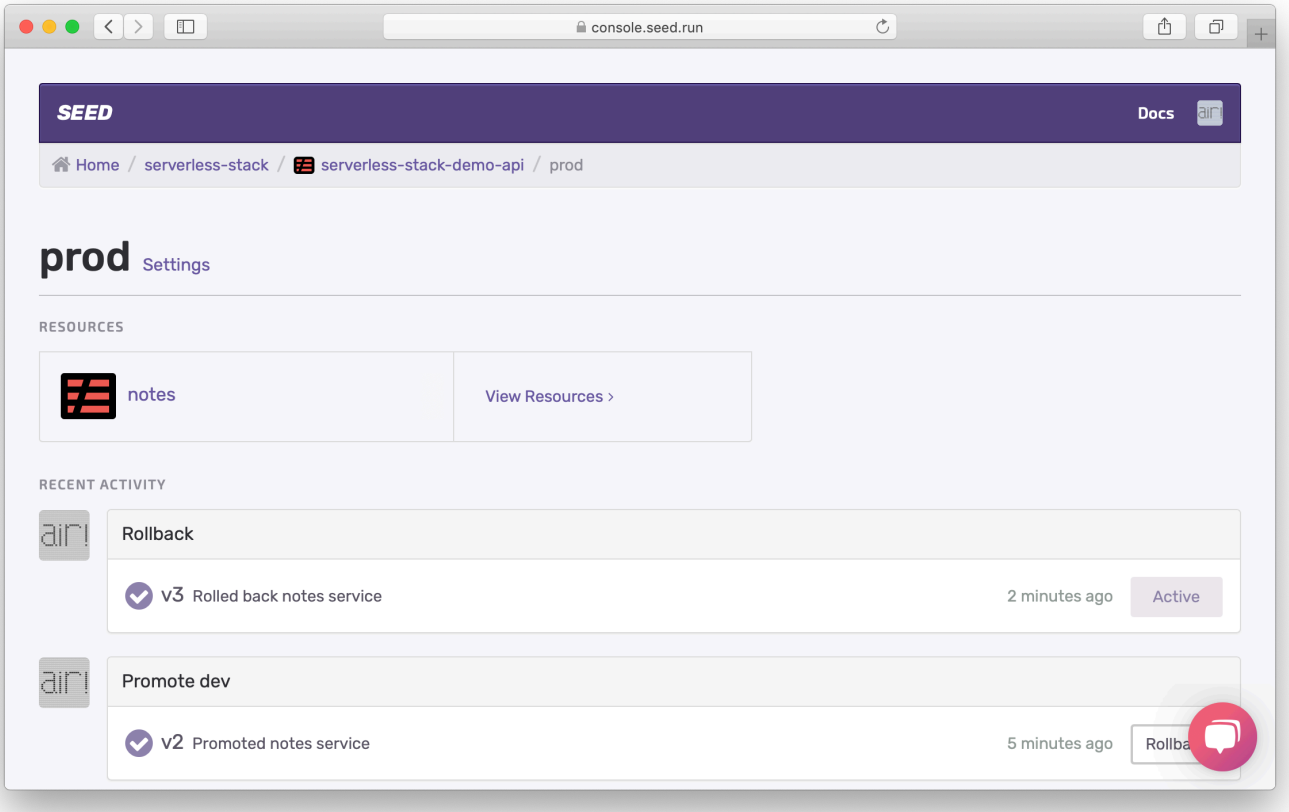
Rollback in Production

Now obviously, we have a problem. Usually you might be tempted to fix the code and push and promote the change. But since our users might be affected by faulty promotions to prod, we want to rollback our changes immediately.

To do this, head back to the **prod** stage. And hit the **Rollback** button on the previous build we had in production.



Seed keeps track of your past builds and simply uses the previously built package to deploy it again.



And now if you run your test command from before.

```
$ npx aws-api-gateway-cli-test \
--username='admin@example.com' \
--password='Passw0rd!' \
--user-pool-id='YOUR_PROD_COGNITO_USER_POOL_ID' \
--app-client-id='YOUR_PROD_COGNITO_APP_CLIENT_ID' \
--cognito-region='YOUR_PROD_COGNITO_REGION' \
--identity-pool-id='YOUR_PROD_IDENTITY_POOL_ID' \
--invoke-url='YOUR_PROD_API_GATEWAY_URL' \
--api-gateway-region='YOUR_PROD_API_GATEWAY_REGION' \
--path-template='/notes' \
--method='POST' \
--body='{"content":"hello
world","attachment":"hello.jpg"}'
```

You should see it succeed this time.

```
Authenticating with User Pool
Getting temporary credentials
Making API request
{ status: 200,
  statusText: 'OK',
  data:
   { userId: 'us-east-1:9bdc031d-ee9e-4ffa-9a2d-
123456789',
     noteId: '8f7da030-650b-11e7-a661-123456789',
     content: 'hello world',
     attachment: 'hello.jpg',
     createdAt: 1499648598452 } }
```

Revert the Code

CHANGE Finally, don't forget to revert your code in `functions/create.js`. Remove the faulty code:

```
uuid.abc.gibberish;
```

CHANGE And commit and push the changes.

```
$ git add .
$ git commit -m "Fixing the mistake"
$ git push
```

And that's it! Now we are ready to plug our new backend into our React app.

CHECKPOINT

For reference, here is the complete code for the backend.

Backend Source (https://github.com/AnomalyInnovations/serverless-stack-demo-api)

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/monitoring-deployments-in-

```
seed/180)
```

Automating React Deployments

Now that we have our backend deployed to production, we are ready to deploy our frotend to production as well! We'll be using a service called Netlify (https://www.netlify.com) to do this. Netlify will not only host our React app, it'll also help automate our deployments. It's a little like what we did for our serverless API backend. We'll configure it so that it'll deploy our React app when we push our changes to Git. However, there are a couple of subtle differences between the way we configure our backend and frontend deployments.

1. Netlify hosts the React app on their infrastructure. In the case of our serverless API backend, it was hosted on our AWS account.

2. Any changes that are pushed to our `master` branch will update the production version of our React app. This means that we'll need to use a slightly different workflow than our backend. We'll use a separate branch where we will do most of our development and only push to master once we are ready to update production.

We have an alternative version of this where we deploy our React app to S3 and we use CloudFront as a CDN in front of it. Then we used Route 53 to configure our domain with it. We also had to configure the www version of our domain and this needed another S3 and CloudFront distribution. This process can be a bit cumbersome. But if you are looking for a way to

deploy and host the React app in your AWS account, we have an Extra Credit chapter on this here — Deploying a React app on AWS (/chapters/deploying-a-react-app-to-aws.html).

Just as in the case with our backend, we could use Travis CI (https://travis-ci.org) or Circle CI (https://circleci.com) for this but it can take a bit more configuration and we'll cover that in a different chapter.

Now before we can automate our deployments, we'll need to configure environments in our React app. This'll allow the production version of our React app to connect to our production backend.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/automating-react-deployments/188)

Manage Environments in Create React App

Recall from our backend section that we created two environments (dev and prod) for our serverless backend API. In this chapter we'll configure our frontend Create React App to connect to it.

Let's start by looking at how our app is configured currently. Our `src/config.js` stores the info for all of our backend resources.

```
export default {
  MAX_ATTACHMENT_SIZE: 5000000,
  STRIPE_KEY: "pk_test_1234567890",
  s3: {
    REGION: "us-east-1",
    BUCKET: "notes-app-uploads"
  },
  apiGateway: {
    REGION: "us-east-1",
    URL: "https://5by75p4gn3.execute-api.us-east-
1.amazonaws.com/prod"
  },
  cognito: {
    REGION: "us-east-1",
    USER_POOL_ID: "us-east-1_udmFFSb92",
    APP_CLIENT_ID: "4hmari2sqvskrup67crkqa4rmo",
```

```
    IDENTITY_POOL_ID: "us-east-1:ceef8ccc-0a19-4616-
9067-854dc69c2d82"
  }
};
```

We need to change this so that when we *push* our app to **dev** it connectes to the dev environment of our backend and for **prod** it connects to the prod environment. Of course you can add many more environments, but let's just stick to these for now.

Environment Variables in Create React App

Our React app is a static single page app. This means that once a *build* is created for a certain environment it persists for that environment.

Create React App (https://github.com/facebookincubator/create-react-app/blob/master/packages/react-scripts/template/README.md#adding-custom-environment-variables) has support for custom environment variables baked into the build system. To set a custom environment variable, simply set it while starting the Create React App build process.

```
$ REACT_APP_TEST_VAR=123 npm start
```

Here `REACT_APP_TEST_VAR` is the custom environment variable and we are setting it to the value `123`. In our app we can access this variable as `process.env.REACT_APP_TEST_VAR`. So the following line in our app:

```
console.log(process.env.REACT_APP_TEST_VAR);
```

Will print out `123` in our console.

Note that, these variables are embedded during build time. Also, only the variables that start with `REACT_APP_` are embedded in our app. All the other environment variables are ignored.

Stage Environment Variable

For our purpose let's use an environment variable called `REACT_APP_STAGE`. This variable will take the values `dev` and `prod`. And by default it is set to `dev`. Now we can rewrite our config with this.

✦ CHANGE Replace `src/config.js` with this.

```
const dev = {
  STRIPE_KEY: "YOUR_STRIPE_DEV_PUBLIC_KEY",
  s3: {
    REGION: "YOUR_DEV_S3_UPLOADS_BUCKET_REGION",
    BUCKET: "YOUR_DEV_S3_UPLOADS_BUCKET_NAME"
  },
  apiGateway: {
    REGION: "YOUR_DEV_API_GATEWAY_REGION",
    URL: "YOUR_DEV_API_GATEWAY_URL"
  },
  cognito: {
    REGION: "YOUR_DEV_COGNITO_REGION",
    USER_POOL_ID: "YOUR_DEV_COGNITO_USER_POOL_ID",
    APP_CLIENT_ID: "YOUR_DEV_COGNITO_APP_CLIENT_ID",
    IDENTITY_POOL_ID: "YOUR_DEV_IDENTITY_POOL_ID"
  }
};

const prod = {
```

```
  STRIPE_KEY: "YOUR_STRIPE_PROD_PUBLIC_KEY",
  s3: {
    REGION: "YOUR_PROD_S3_UPLOADS_BUCKET_REGION",
    BUCKET: "YOUR_PROD_S3_UPLOADS_BUCKET_NAME"
  },
  apiGateway: {
    REGION: "YOUR_PROD_API_GATEWAY_REGION",
    URL: "YOUR_PROD_API_GATEWAY_URL"
  },
  cognito: {
    REGION: "YOUR_PROD_COGNITO_REGION",
    USER_POOL_ID: "YOUR_PROD_COGNITO_USER_POOL_ID",
    APP_CLIENT_ID: "YOUR_PROD_COGNITO_APP_CLIENT_ID",
    IDENTITY_POOL_ID: "YOUR_PROD_IDENTITY_POOL_ID"
  }
};

// Default to dev if not set
const config = process.env.REACT_APP_STAGE === 'prod'
  ? prod
  : dev;

export default {
  // Add common config values here
  MAX_ATTACHMENT_SIZE: 5000000,
  ...config
};
```

Make sure to replace the different version of the resources with the ones from the Deploying through Seed (/chapters/deploying-through-seed.html) chapter.

We did not complete our Stripe account setup back then, so we don't have the production version of this key. For now we'll just assume that we have two versions of the same key. So `YOUR_STRIPE_DEV_PUBLIC_KEY` and `YOUR_STRIPE_PROD_PUBLIC_KEY` are the same ones for now.

Note that we are defaulting our environment to dev if the `REACT_APP_STAGE` is not set. This means that our current build process (`npm start` and `npm run build`) will default to the `dev` environment. Also note that we've moved config values that are common to both environments (like `MAX_ATTACHMENT_SIZE`) to a different section.

If we switch over to our app, we should see it in development mode and it'll be connected to the dev version of our backend. We haven't changed the deployment process yet but in the coming chapters we'll change this when we automate our frontend deployments.

We don't need to worry about the prod version just yet. But as an example, if we wanted to build the prod version of our app we'd have to run the following:

```
$ REACT_APP_STAGE=prod npm run build
```

OR for Windows

```
set "REACT_APP_STAGE=prod" && npm start
```

Next, we'll set up automatic deployments for our React app using a service called Netlify (https://www.netlify.com). This will be fairly similar to what we did for our serverless backend API.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/manage-environments-in-create-react-app/182)

Create a Build Script

Before we can add our project to Netlify (https://www.netlify.com) we just need to set up a build script. If you recall, we had configured our app to use the `REACT_APP_STAGE` build environment variable. We are going to create a build script to tell Netlify to set this variable up for the different deployment cases.

Add the Netlify Build Script

✦ CHANGE Start by adding the following to a file called `netlify.toml` to your project root.

```
# Global settings applied to the whole site.
# “base” is directory to change to before starting
build, and
# “publish” is the directory to publish (relative to
root of your repo).
# “command” is your build command.

[build]
  base    = ""
  publish = "build"
  command = "REACT_APP_STAGE=dev npm run build"

# Production context: All deploys to the main
# repository branch will inherit these settings.
```

```
[context.production]
  command = "REACT_APP_STAGE=prod npm run build"

# Deploy Preview context: All Deploy Previews
# will inherit these settings.
[context.deploy-preview]
  command = "REACT_APP_STAGE=dev npm run build"

# Branch Deploy context: All deploys that are not in
# an active Deploy Preview will inherit these settings.
[context.branch-deploy]
  command = "REACT_APP_STAGE=dev npm run build"
```

The build script is configured based on contexts. There is a default one right up top. There are three parts to this:

1. The `base` is the directory where Netlify will run our build commands. In our case it is in the project root. So this is left empty.
2. The `publish` option points to where our build is generated. In the case of Create React App it is the `build` directory in our project root.
3. The `command` option is the build command that Netlify will use. If you recall the Manage environments in Create React App (/chapters/manage-environments-in-create-react-app.html) chapter, this will seem familiar. In the default context the command is `REACT_APP_STAGE=dev npm run build`.

The production context labelled, `context.production` is the only one where we set the `REACT_APP_STAGE` variable to `prod`. This is when we push to `master`. The `branch-deploy` is what we will be using when we push to any other non-production branch. The `deploy-preview` is

for pull requests.

Handle HTTP Status Codes

Just as in the first part of the tutorial, we'll need to handle requests to any non-root paths of our app. Our frontend is a single-page app and the routing is handled on the client side. We need to tell Netlify to always redirect any request to our `index.html` and return the 200 status code for it.

✦ CHANGE To do this, add a redirects rule at the bottom of `netlify.toml`:

```
# Always redirect any request to our index.html
# and return the status code 200.
[[redirects]]
    from    = "/*"
    to      = "/index.html"
    status  = 200
```

Commit the Changes

✦ CHANGE Let's quickly commit these to Git.

```
$ git add .
$ git commit -m "Adding a Netlify build script"
```

Push the Changes

✦ CHANGE We are pretty much done making changes to our project. So

let's go ahead and push them to GitHub.

```
$ git push
```

Now we are ready to add our project to Netlify.

CHECKPOINT

For reference, here is the code we have so far.

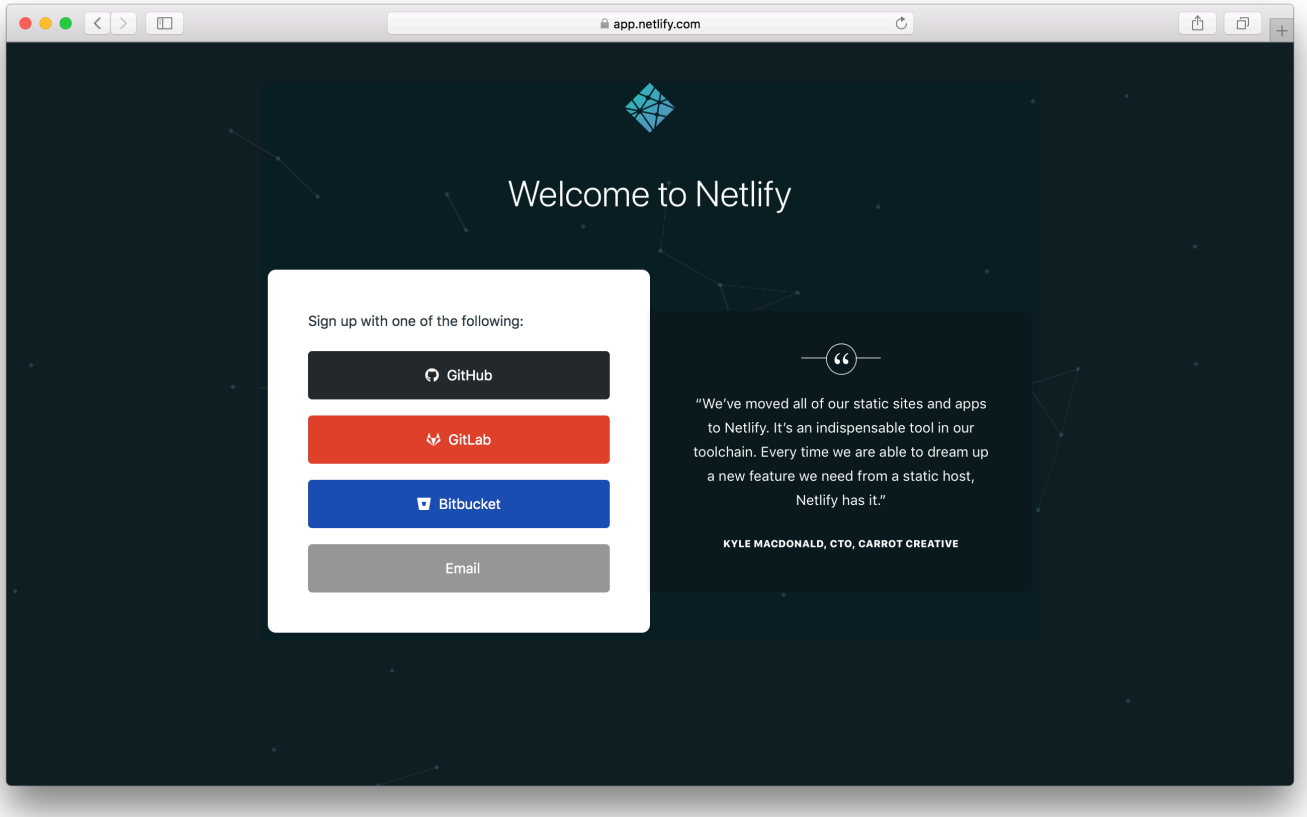
Frontend Source :create-a-build-script (https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/create-a-build-script)

For help and discussion

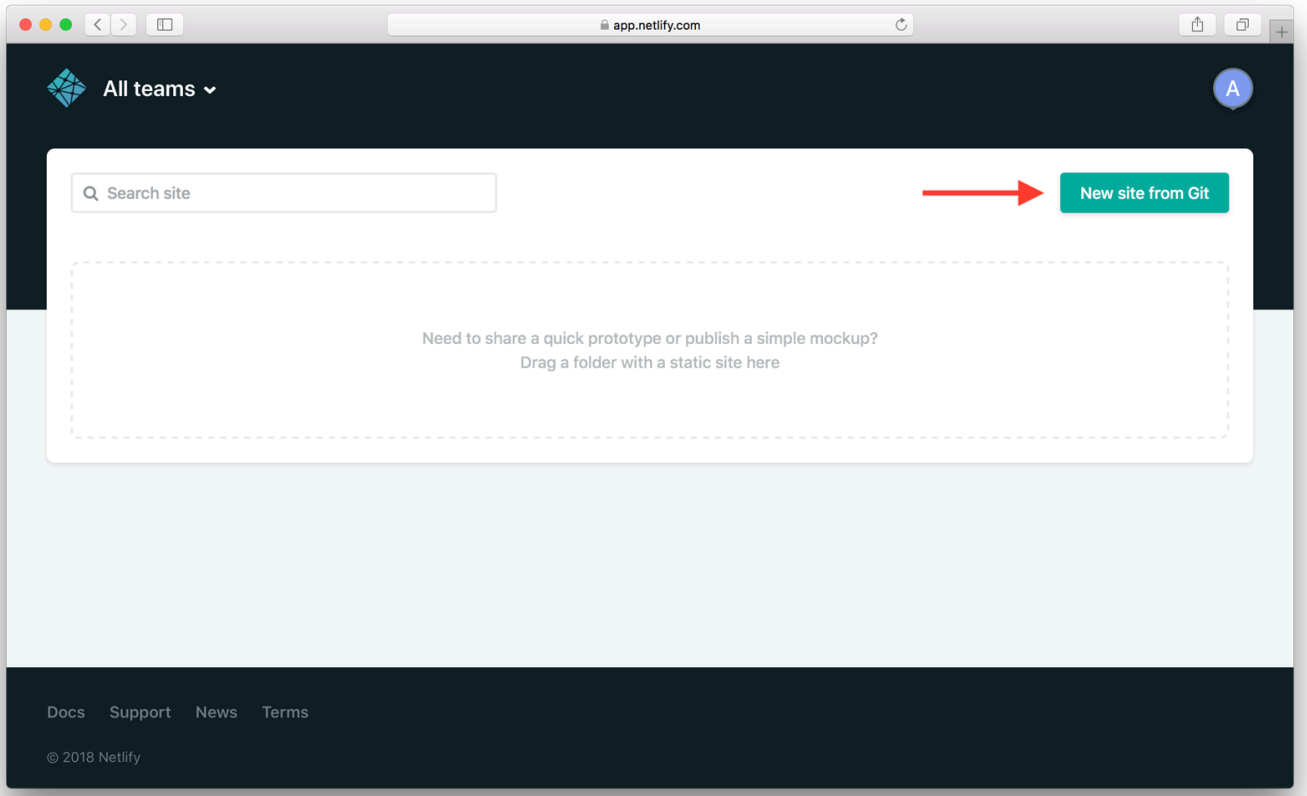
Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-build-script/189)

Setting up Your Project on Netlify

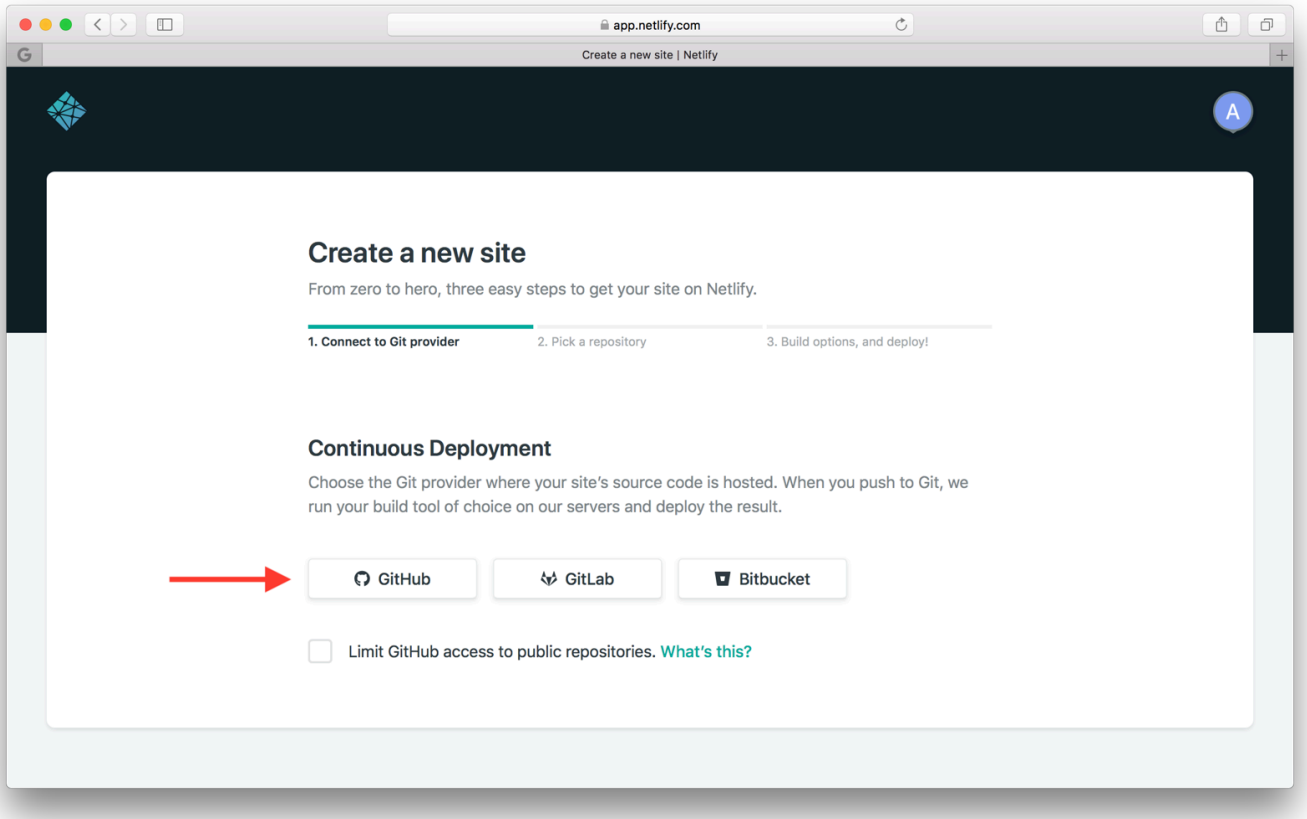
Now we are going to set our React app on Netlify (https://www.netlify.com). Start by creating a free account (https://app.netlify.com/signup).



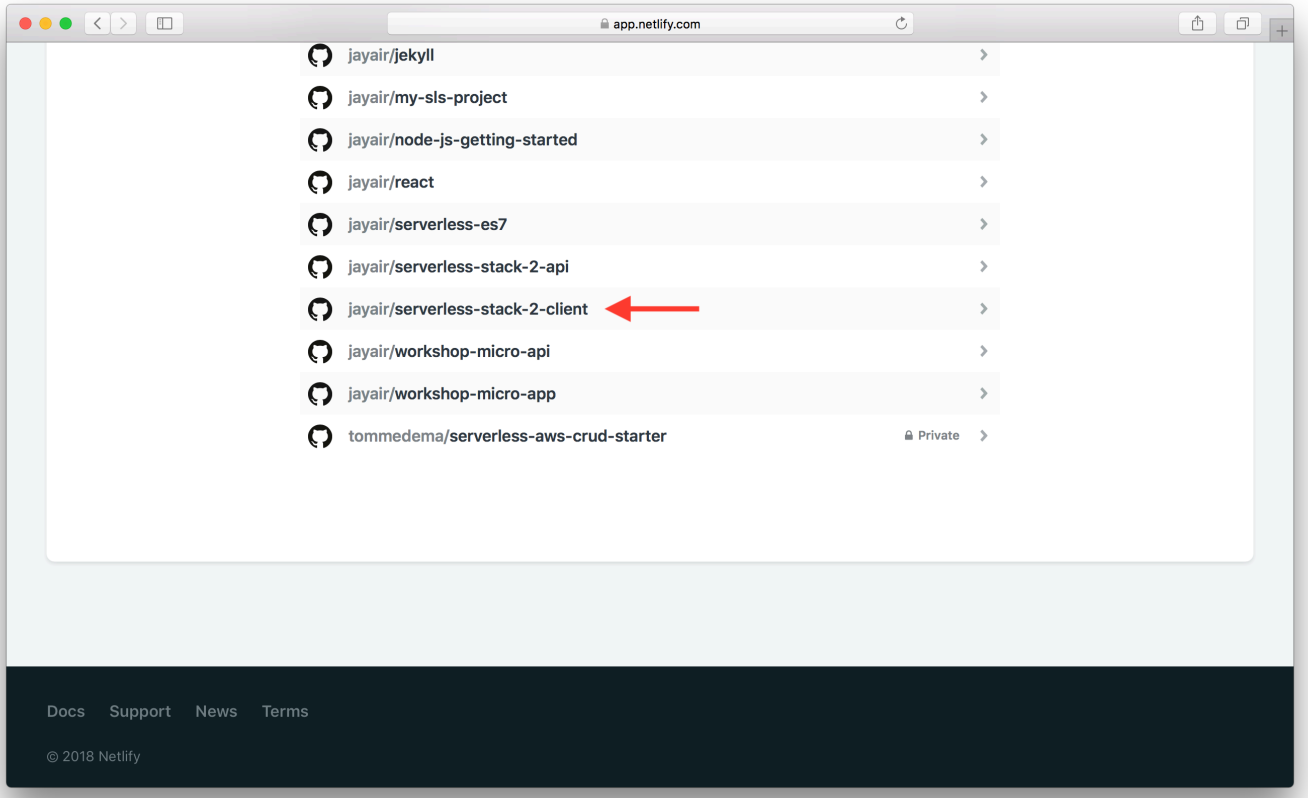
Next, create a new site by hitting the **New site from Git** button.



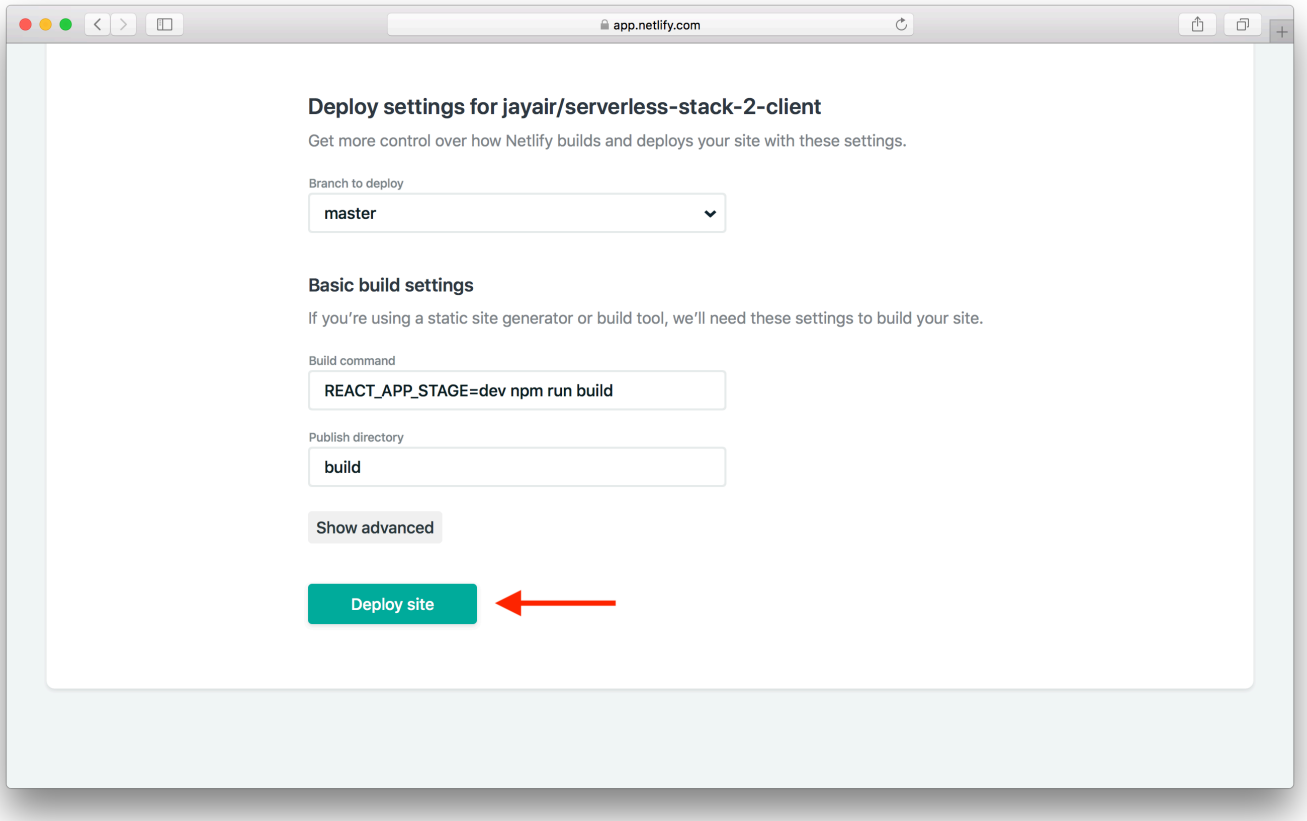
Pick **GitHub** as your provider.



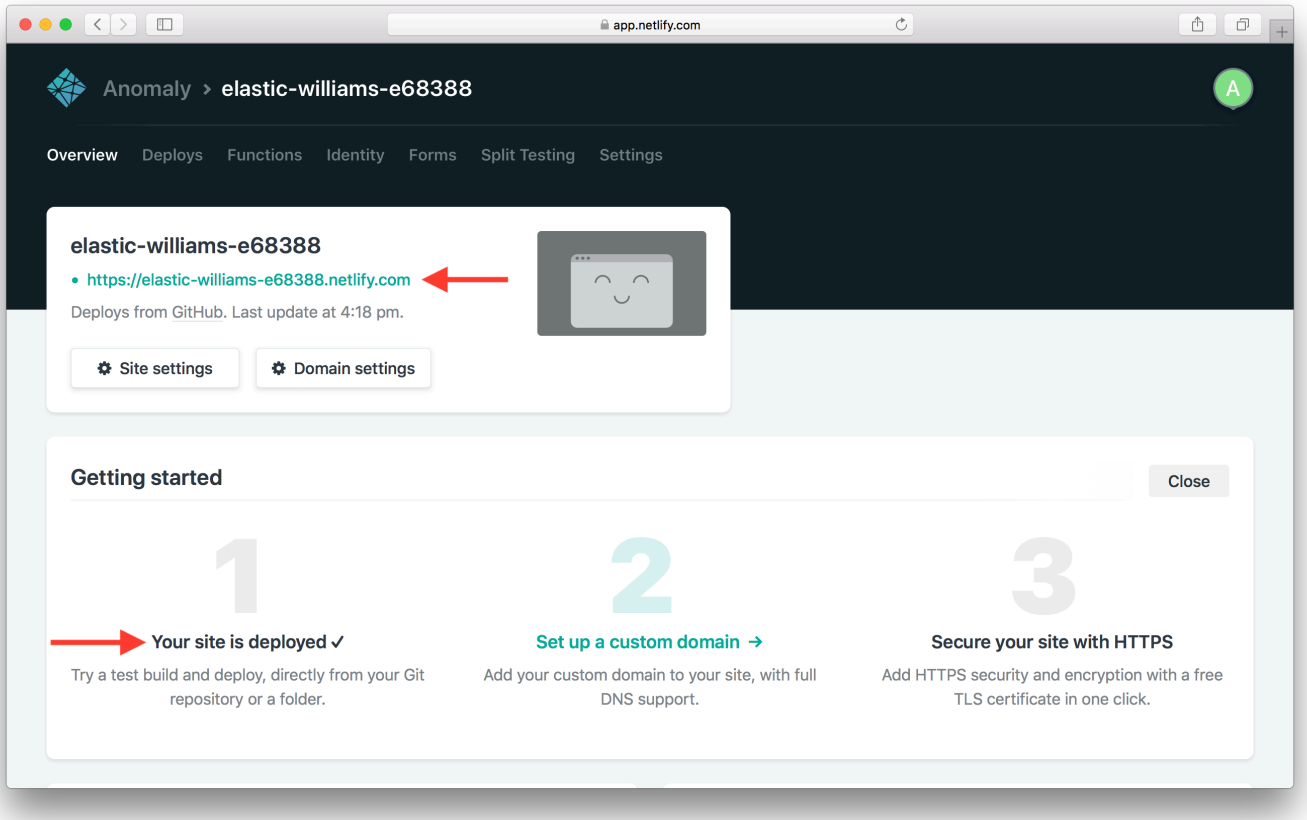
Then pick your project from the list.



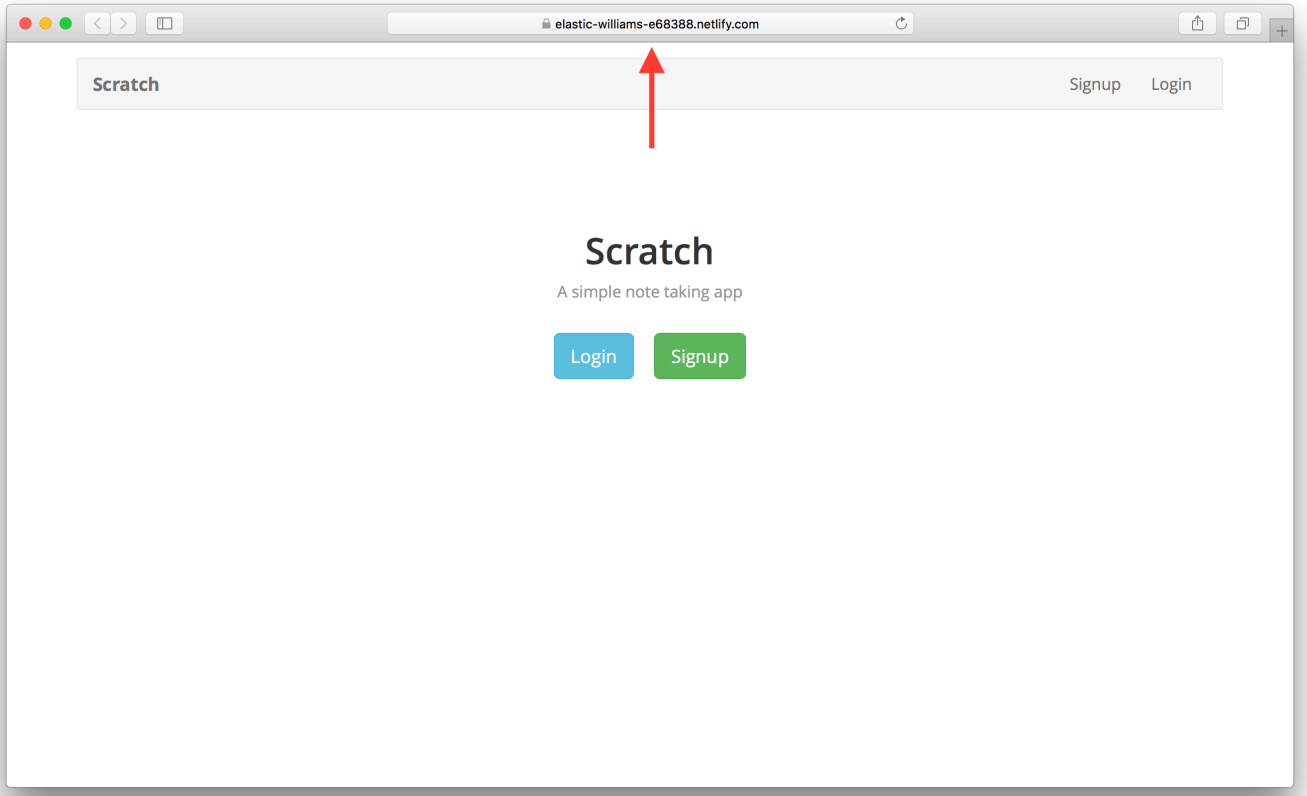
It'll default the branch to `master`. We can now deploy our app! Hit **Deploy site**.



This should be deploying our app. Once it is done, click on the deployment.



And you should see your app in action!



Of course, it is hosted on a Netlify URL. We'll change that by configuring custom domains next.

For help and discussion

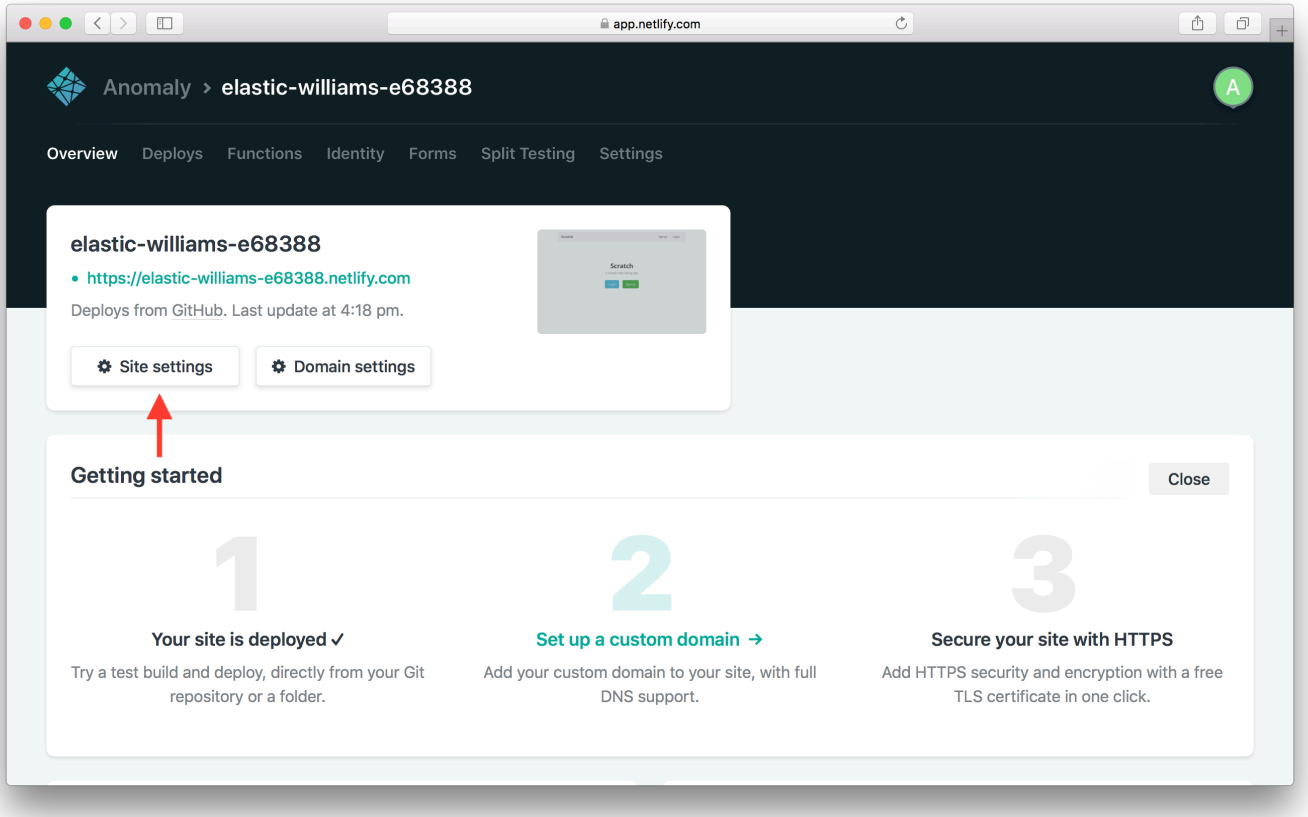
Comments on this chapter (https://discourse.serverless-stack.com/t/setting-up-your-project-on-netlify/190)

Custom Domains in Netlify

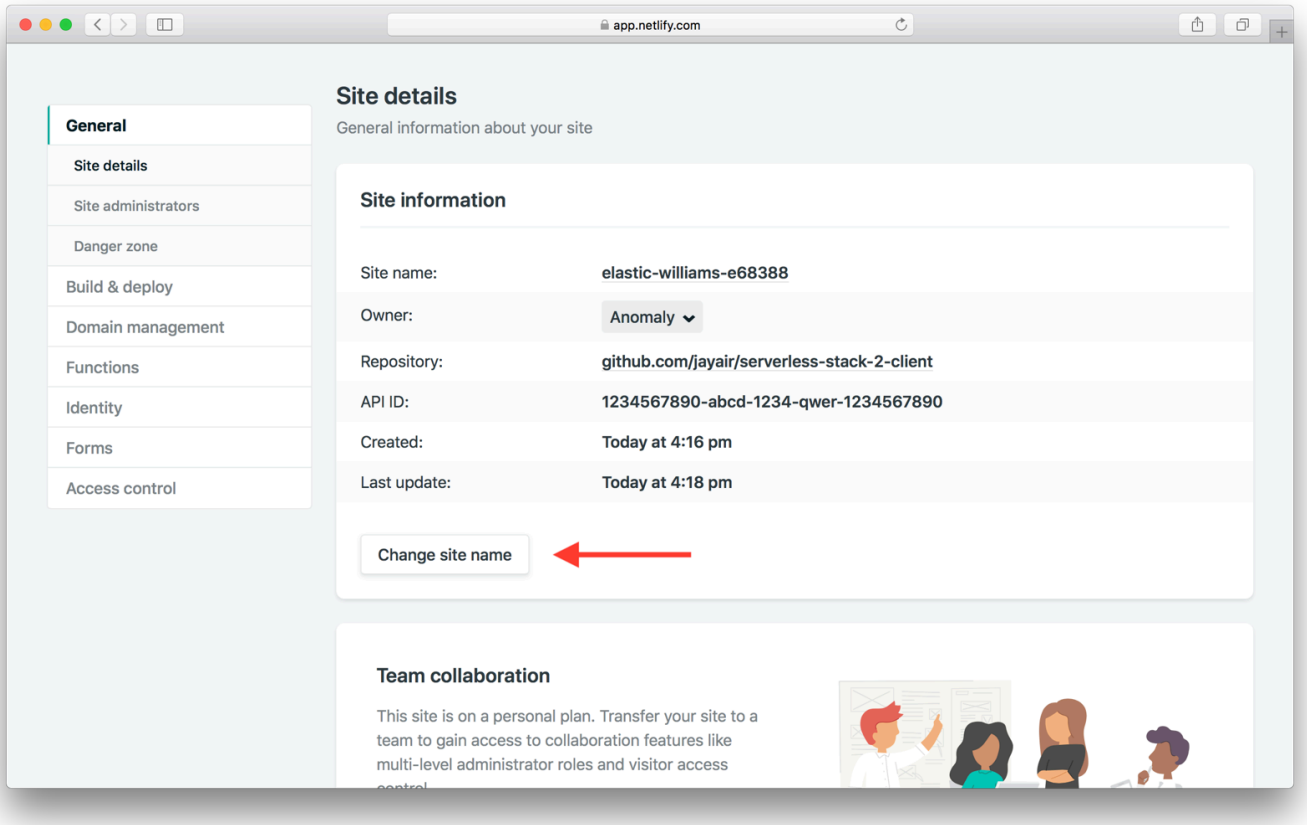
Now that we have our first deployment, let's configure a custom domain for our app through Netlify.

Pick a Netlify Site Name

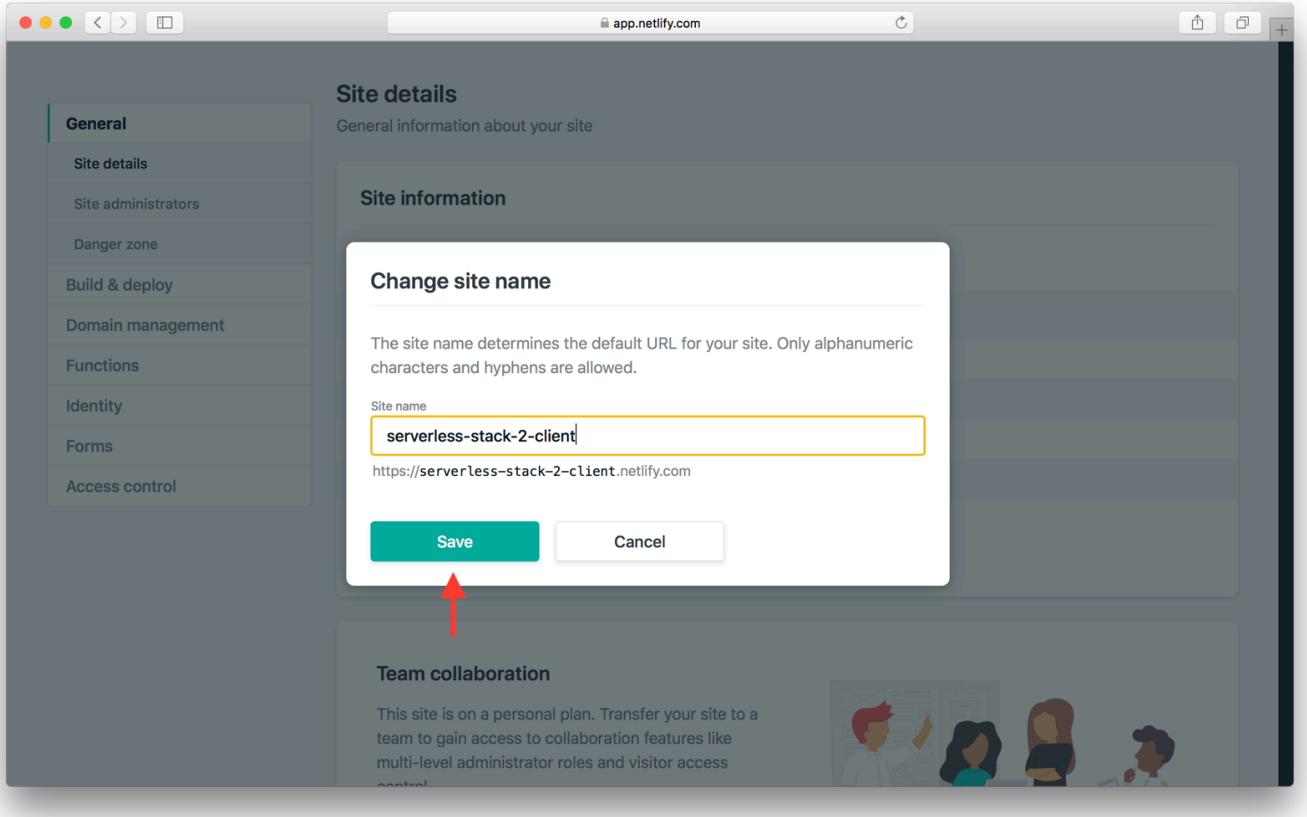
From the project page in Netlify, hit **Site settings**.



Under **Site information** hit **Change site name**.



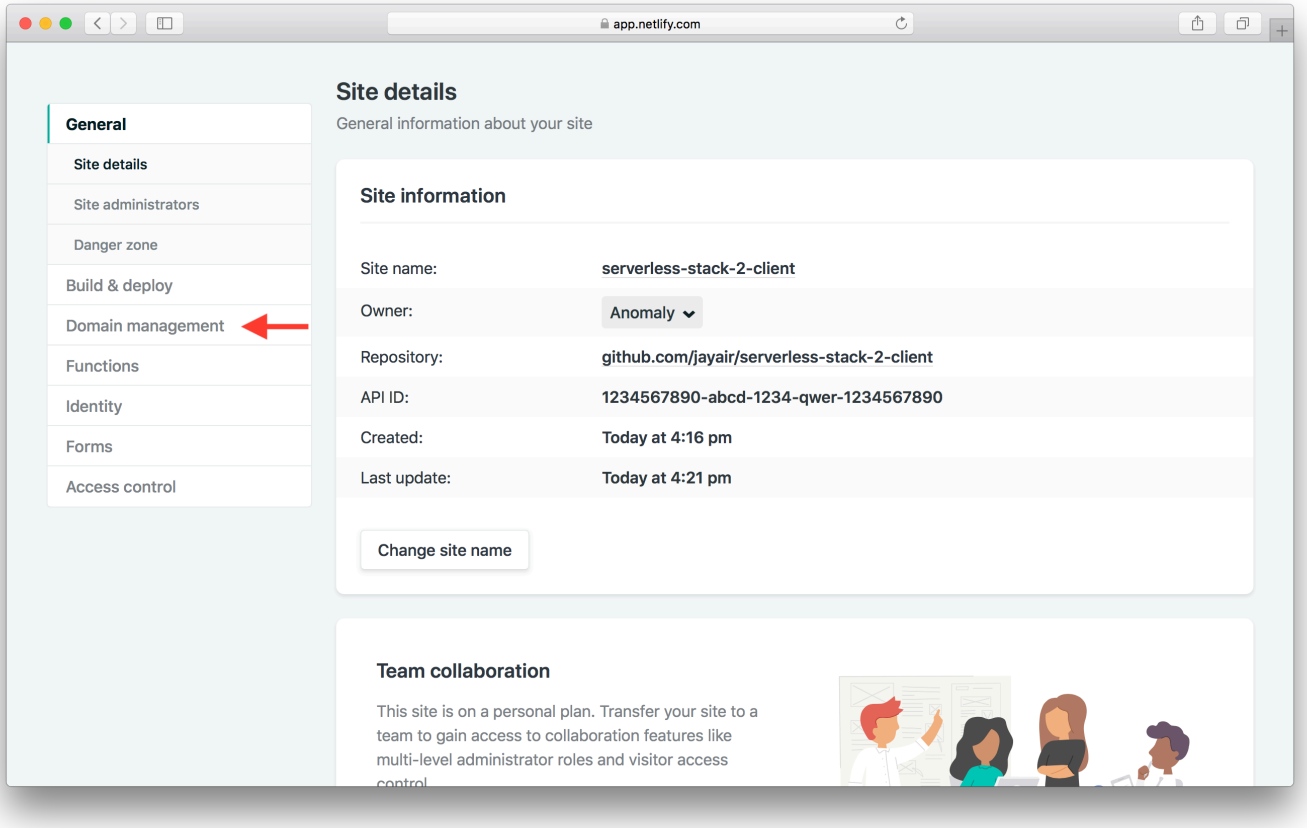
The site names are global, so pick a unique one. In our case we are using `serverless-stack-2-client`. And hit **Save**.



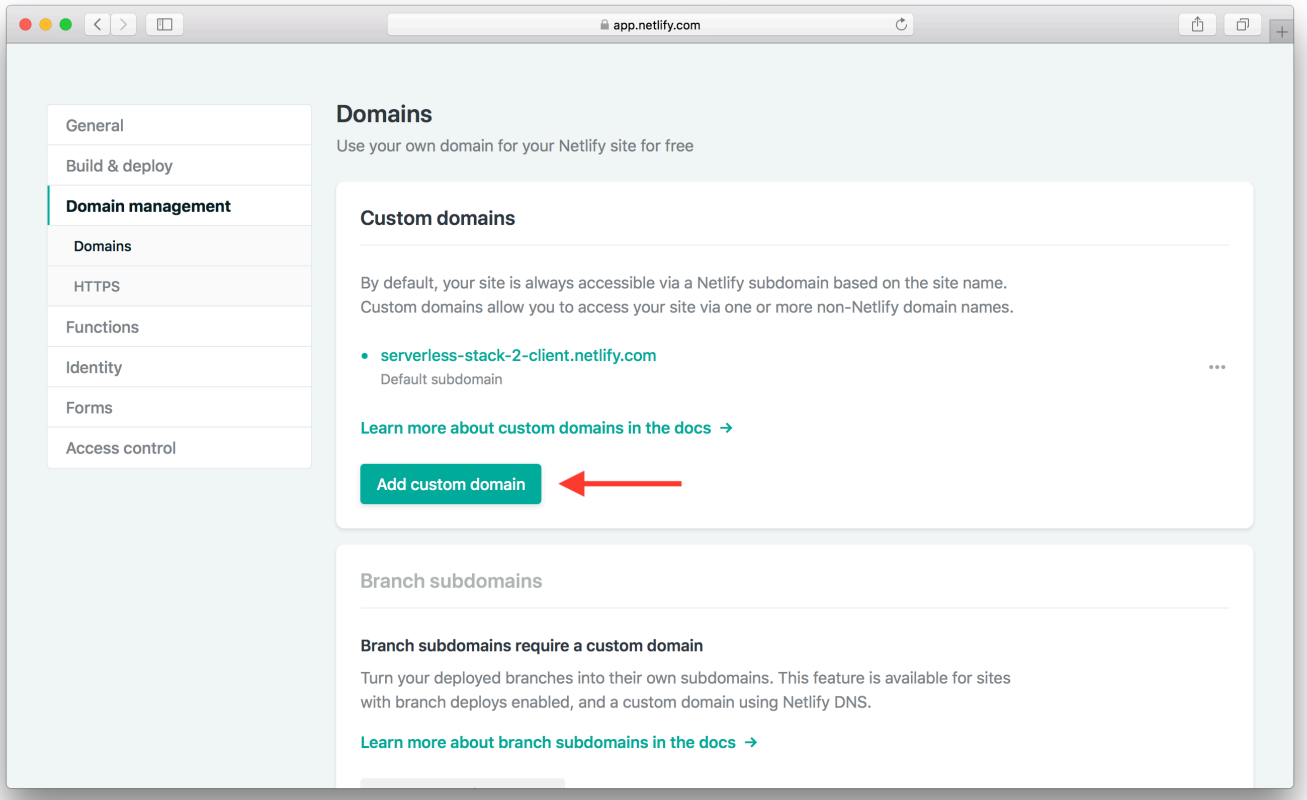
This means that our Netlify site URL is now going to be `https://serverless-stack-2-client.netlify.com`. Make a note of this as we will use this later in this chapter.

Domain Settings in Netlify

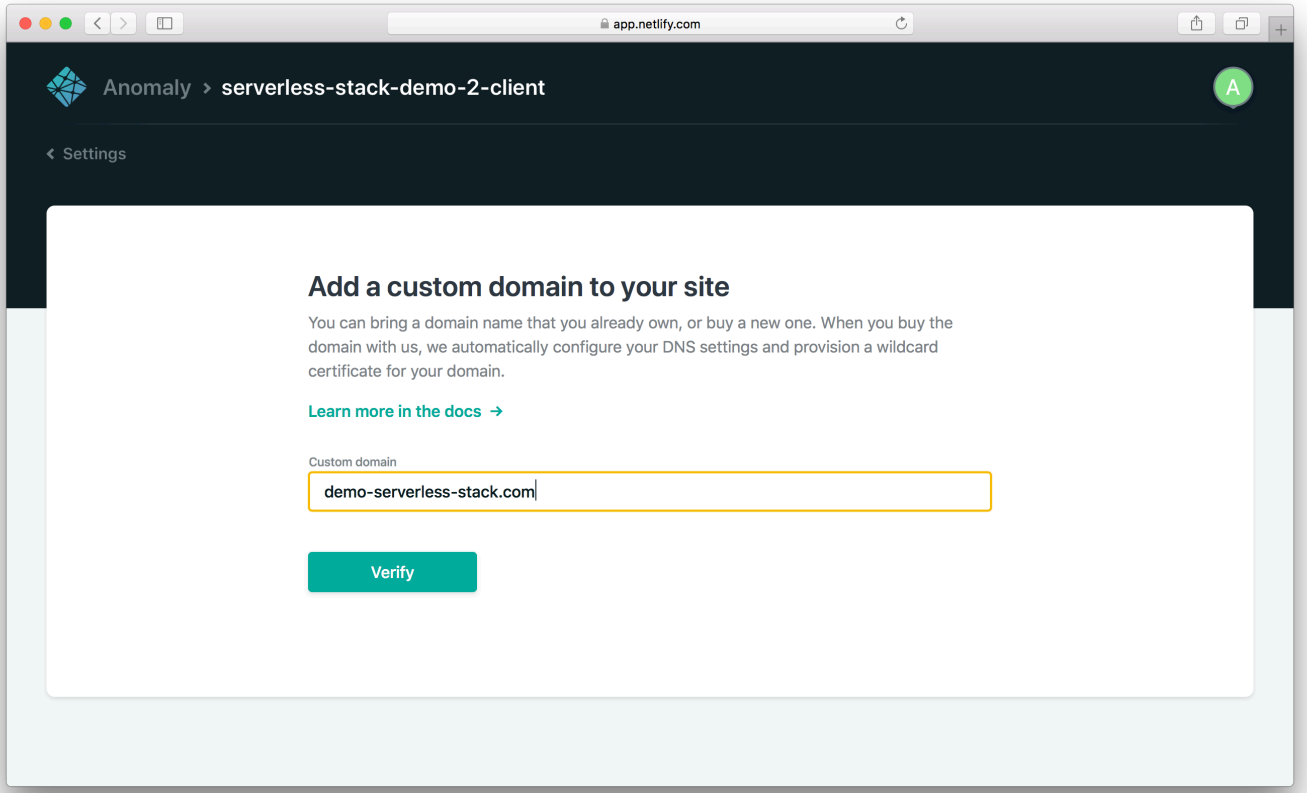
Next hit **Domain management** from the side panel.



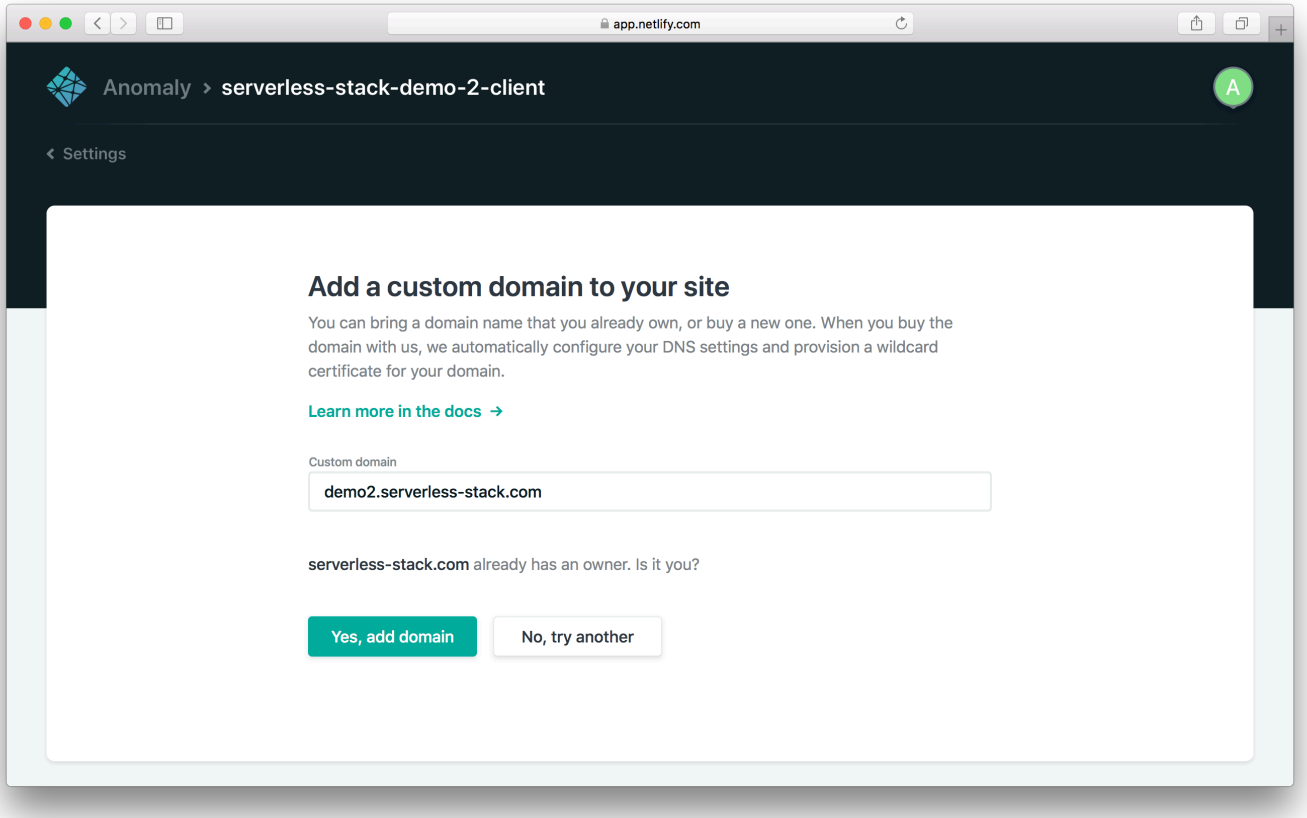
And hit **Add custom domain**.



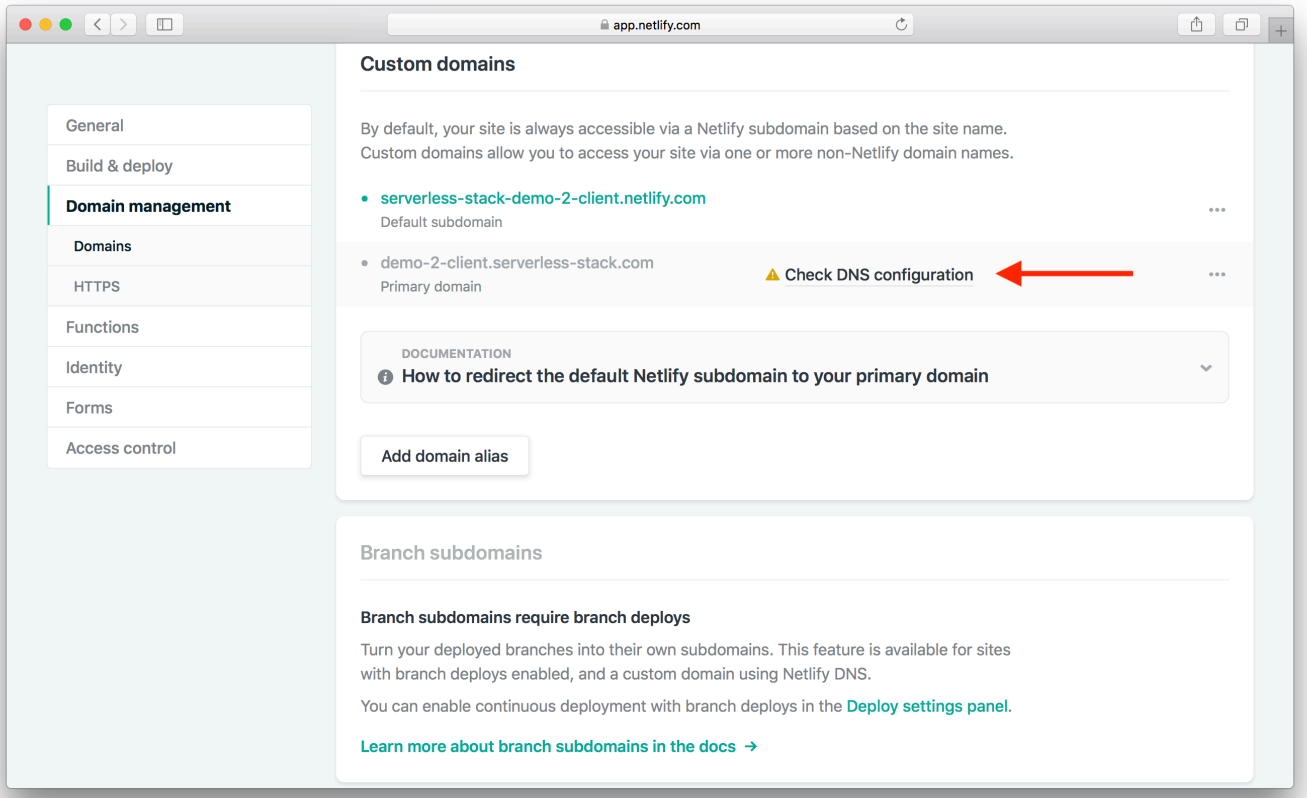
Type in the name of our domain, for example it might be `demo-serverless-stack.com`. And hit **Save**.



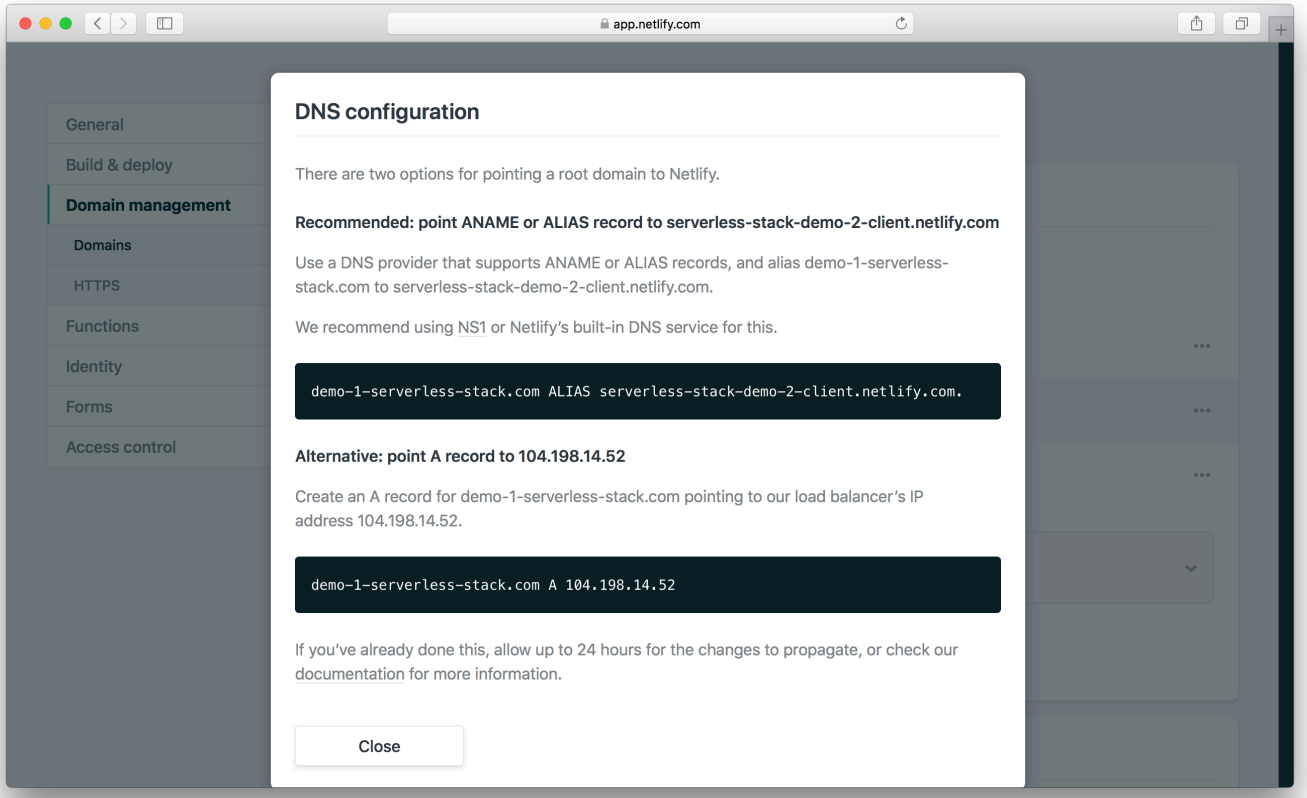
This will ask you to verify that you are the owner of this domain and to add it. Click **Yes, add domain**.



Next hit **Check DNS configuration**.

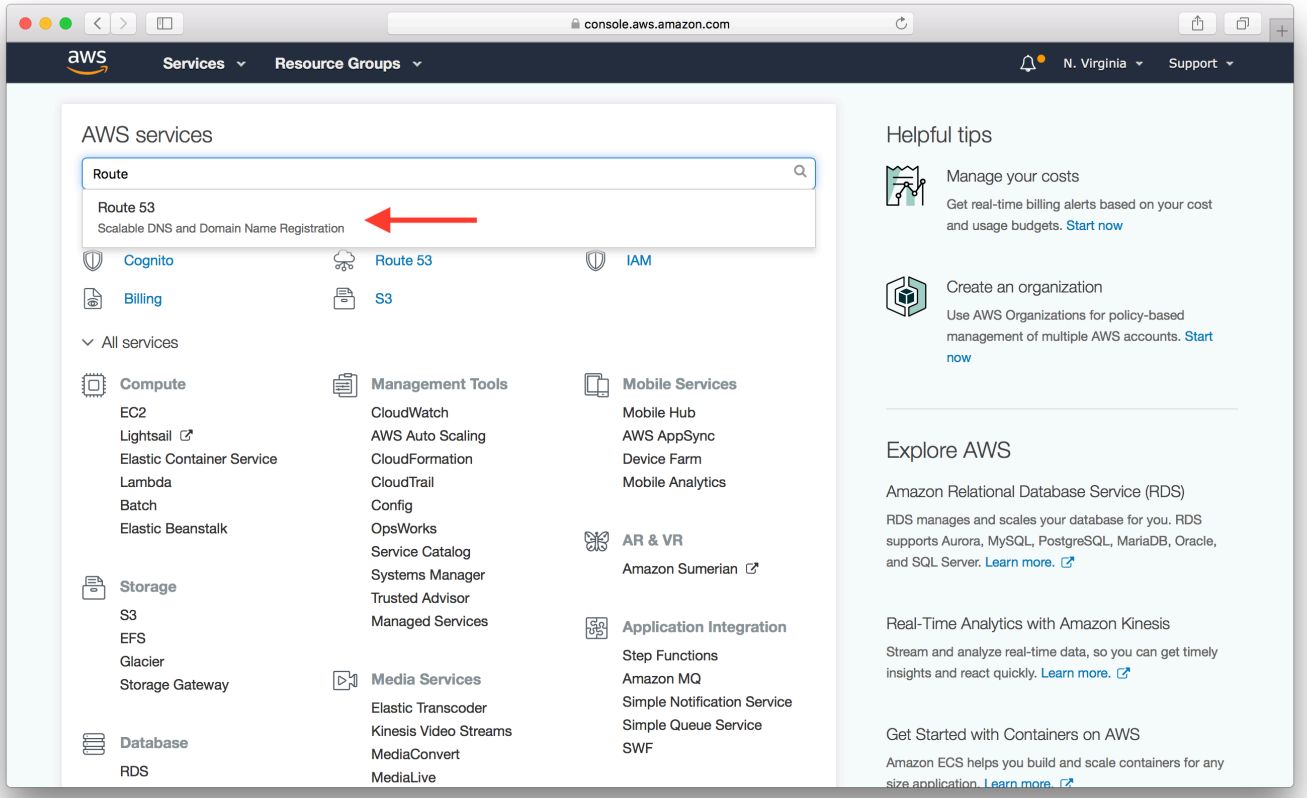


This will show you the instructions for setting up your domain through Route 53.

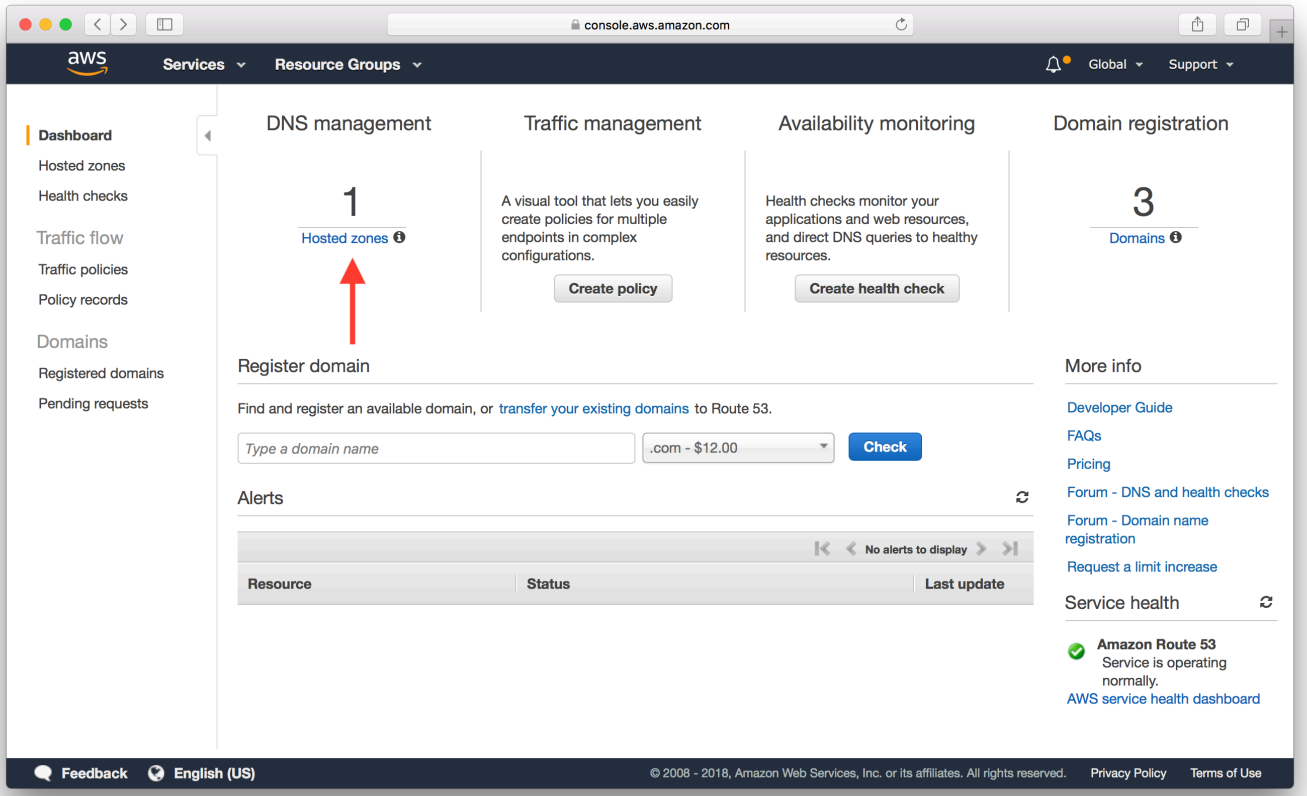


DNS Settings in Route 53

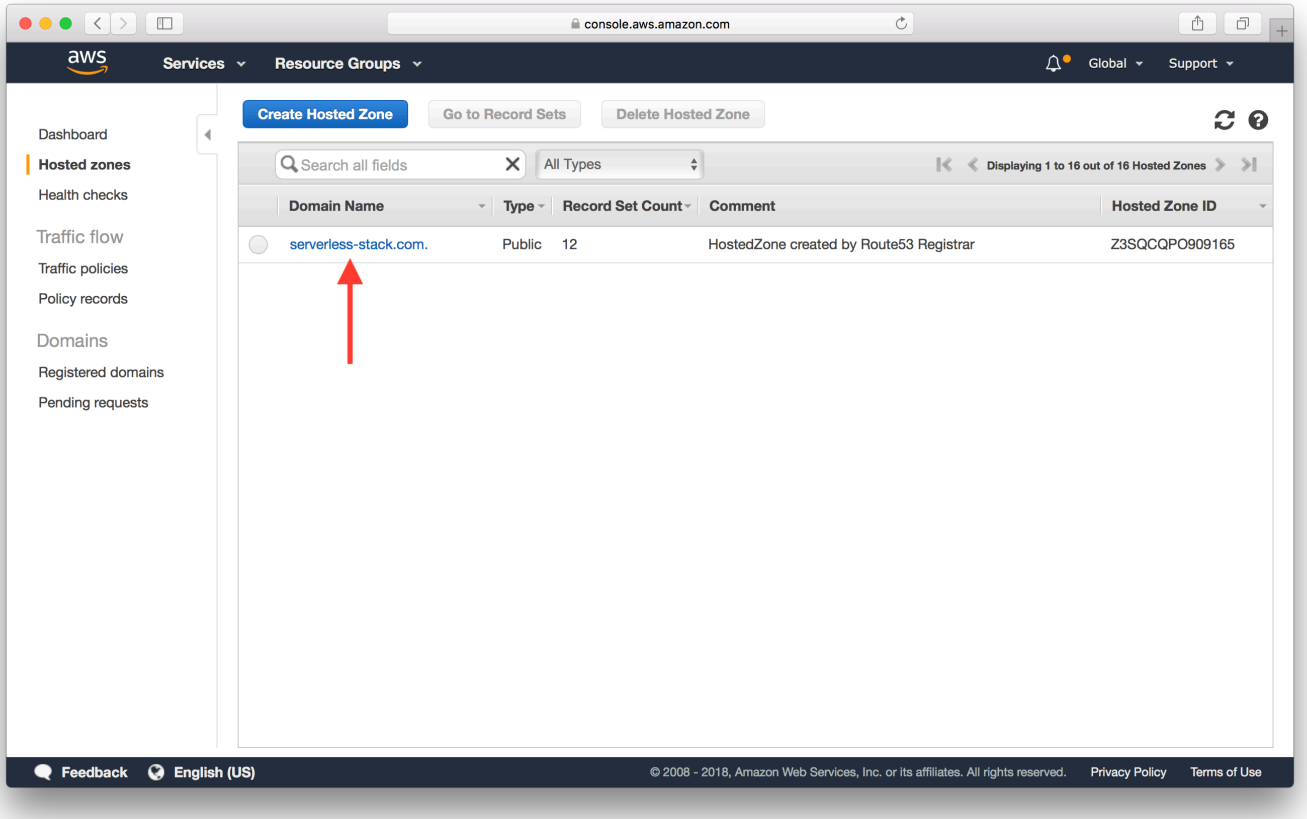
To do this we need to head back to the AWS Console (https://console.aws.amazon.com/). and search for Route 53 as the service.



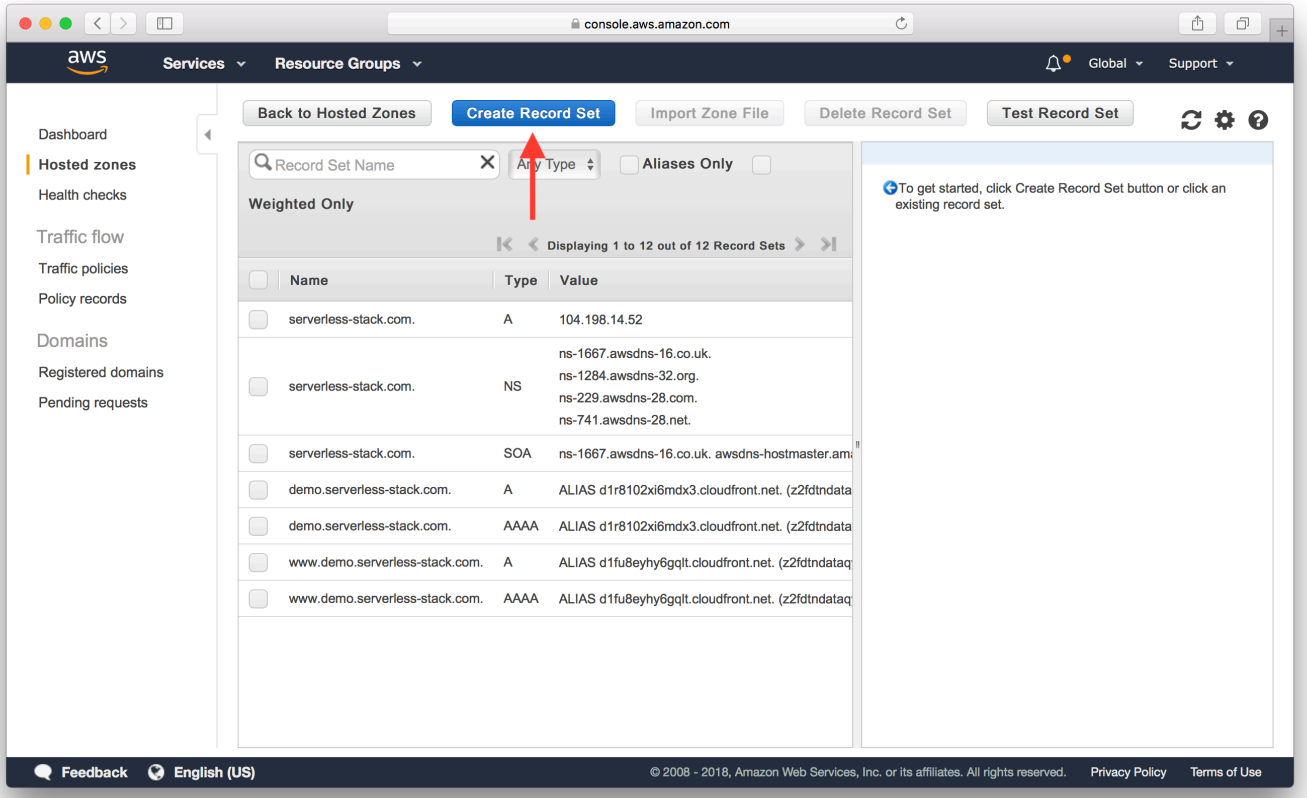
Click on **Hosted zones.**



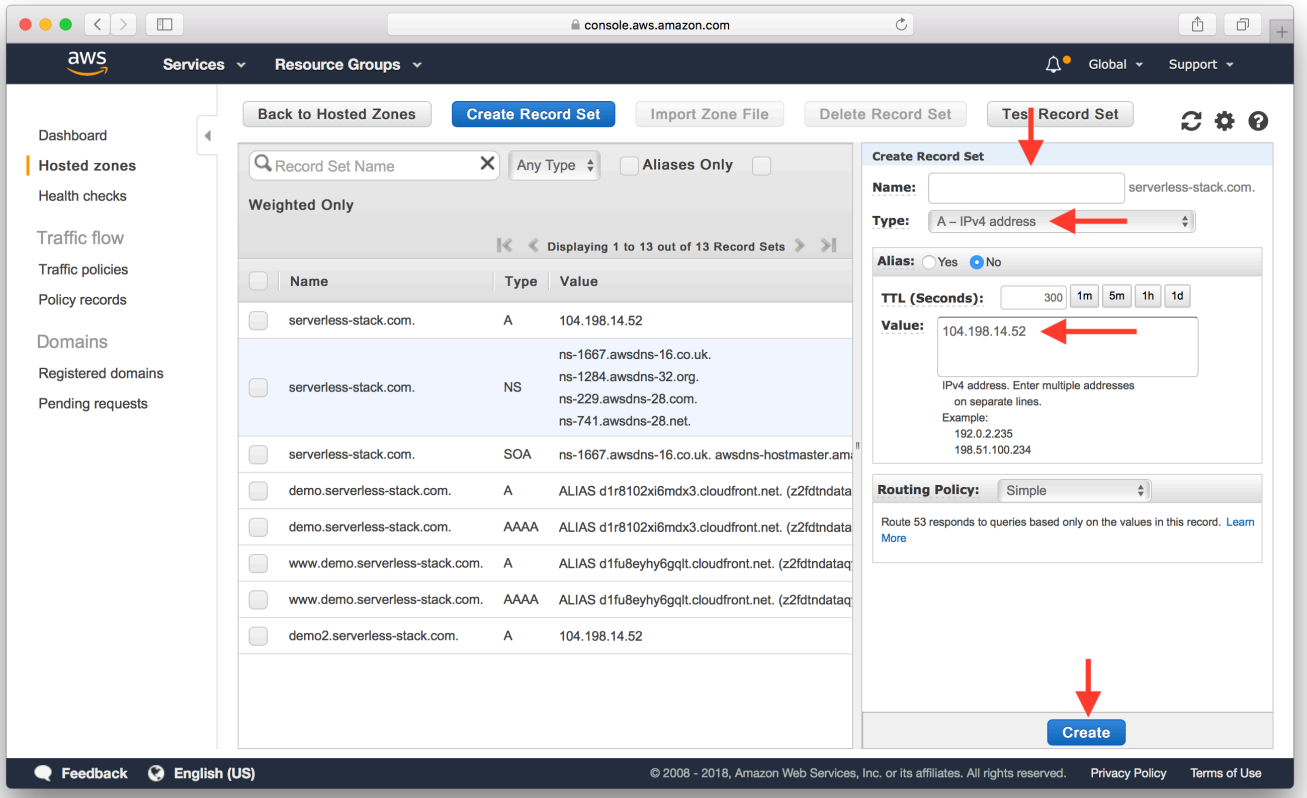
And select the domain we want to configure.



Here click on **Create Record Set**.

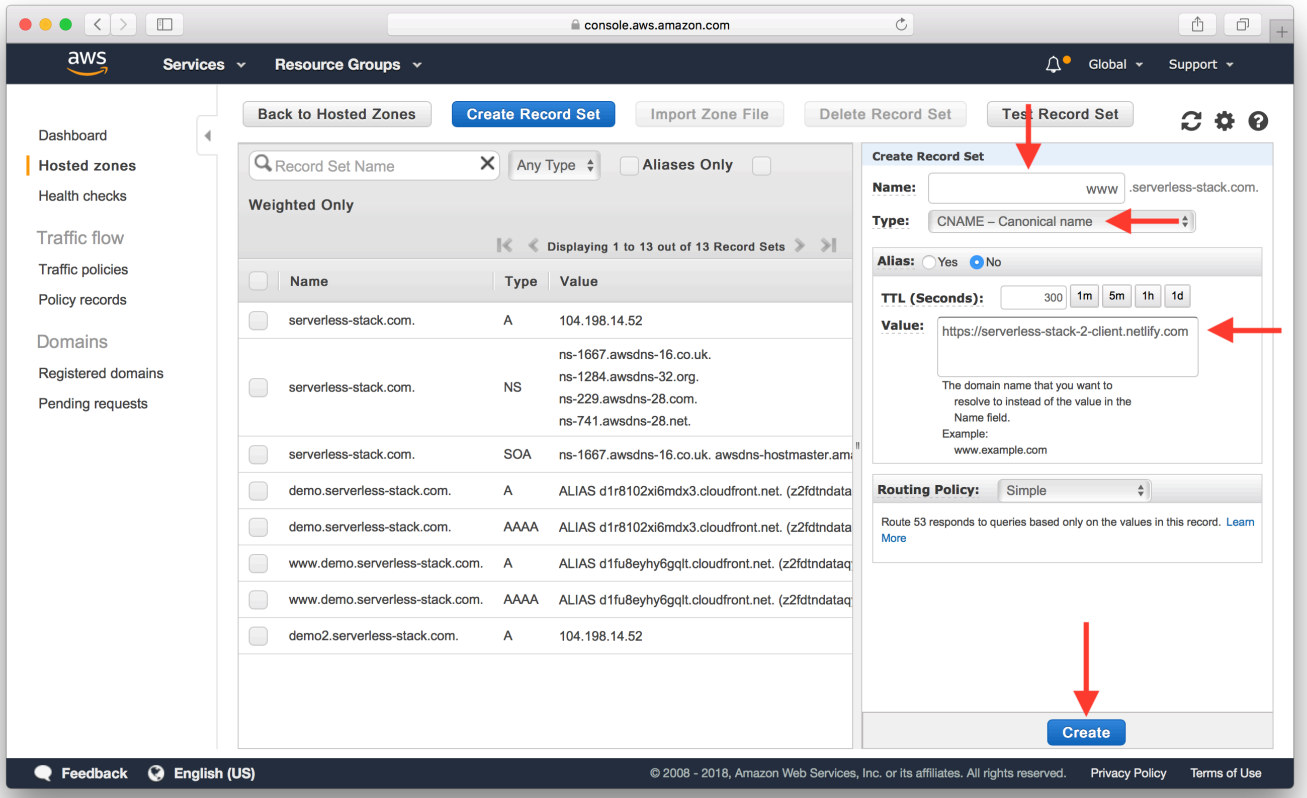


Select **Type** as **A - IPv4 address** and set the **Value** to **104.198.14.52**. And hit **Create**. We get this IP from the Netlify docs on adding custom domains (https://www.netlify.com/docs/custom-domains/).



Next hit **Create Record Set** again.

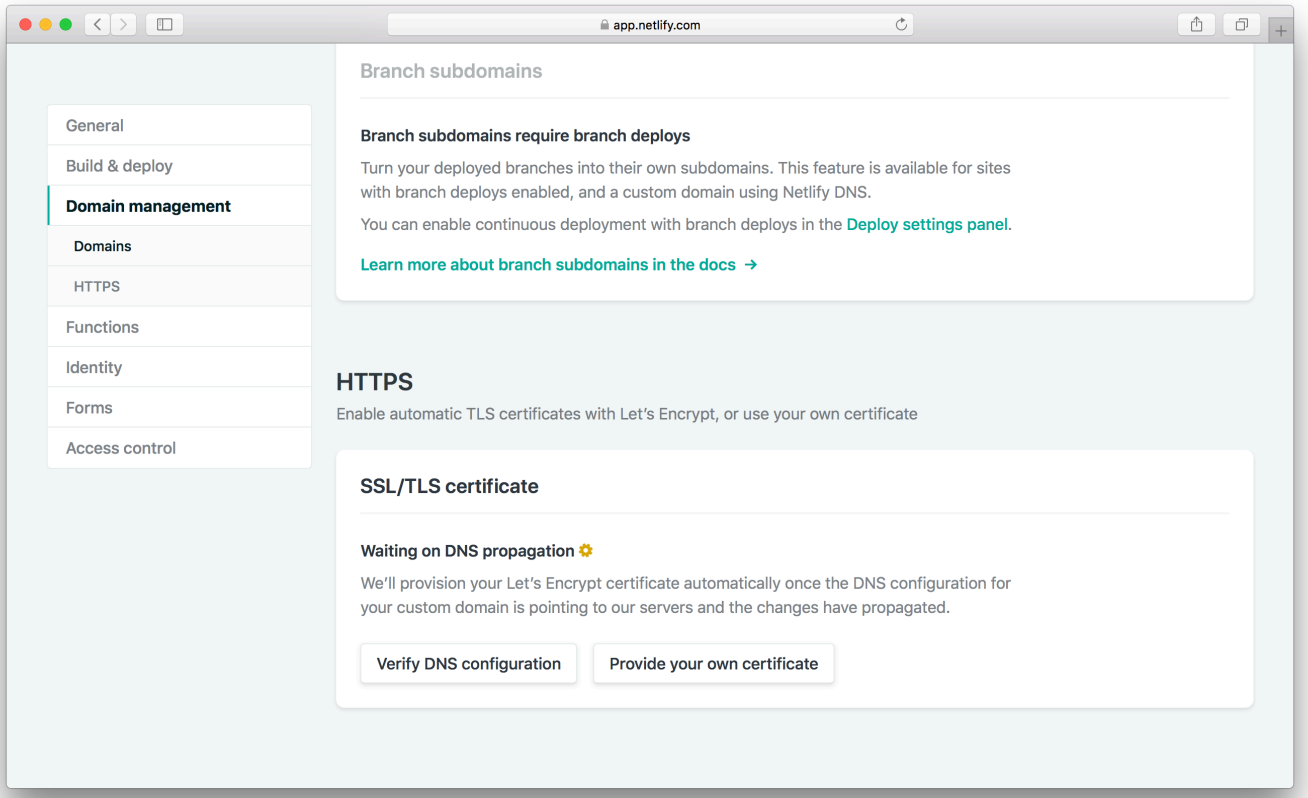
Set **Name** to `www`, **Type** to **CNAME - Canonical name**, and the value to the Netlify site name as we noted above. In our case it is `https://serverless-stack-2-client.netlify.com`. Hit **Create**.



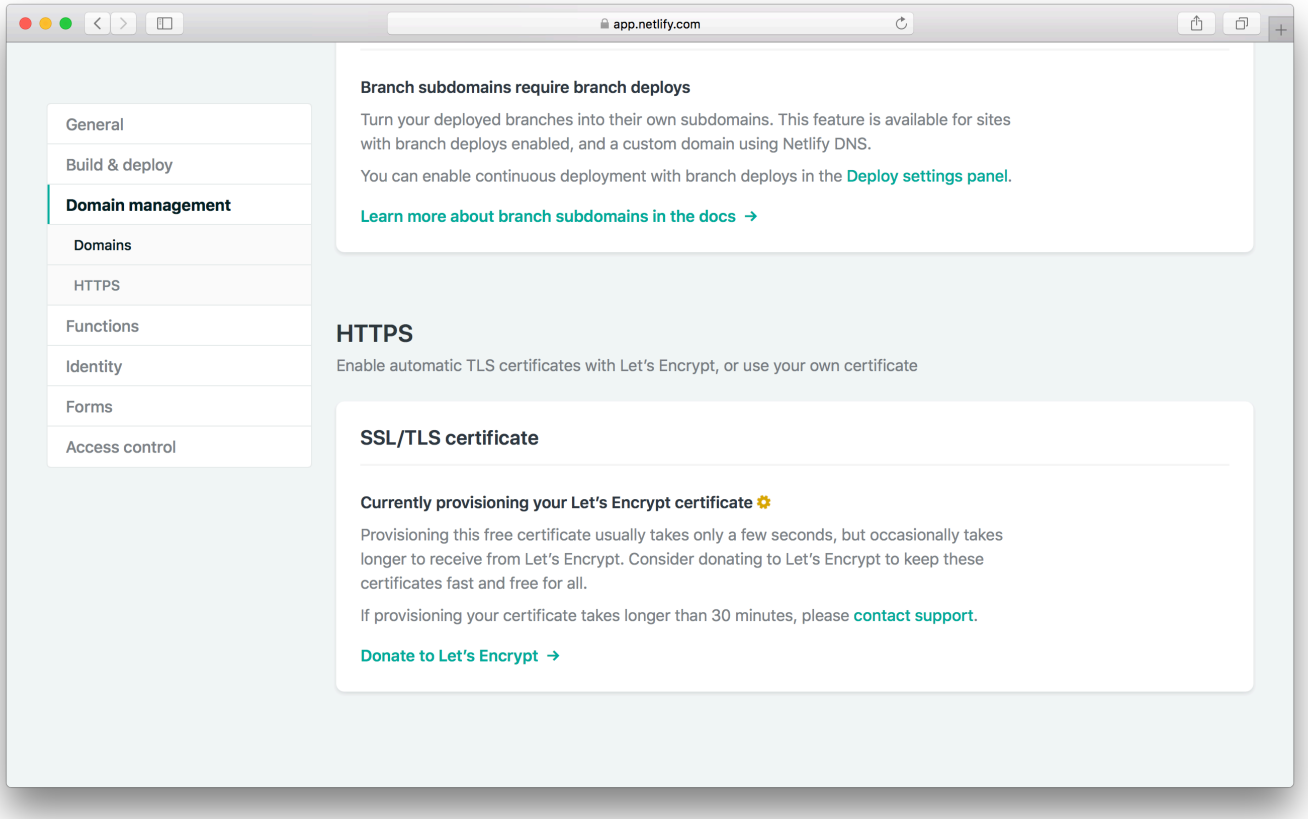
And give the DNS around 30 minutes to update.

Configure SSL

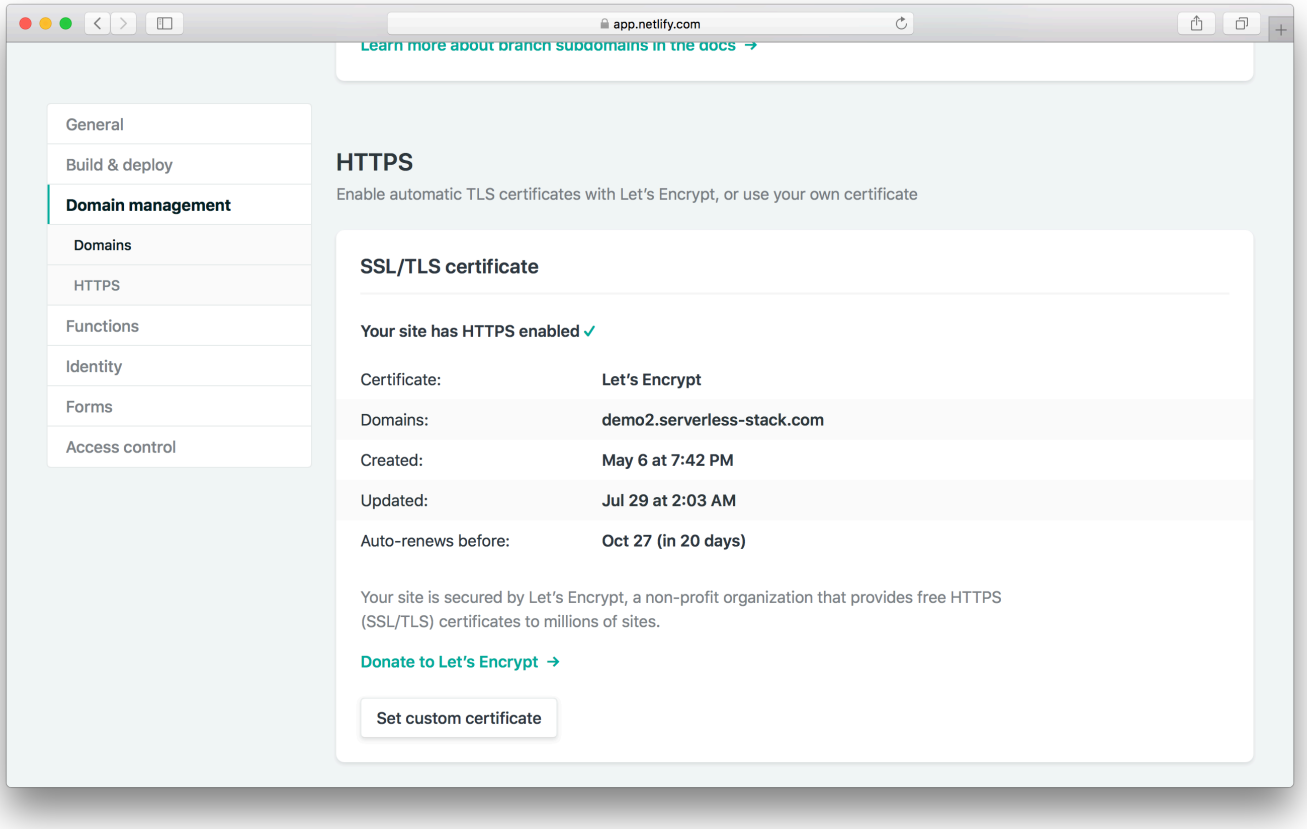
Back in Netlify, hit **HTTPS** in the side panel. And it should say that it is waiting for the DNS to propagate.



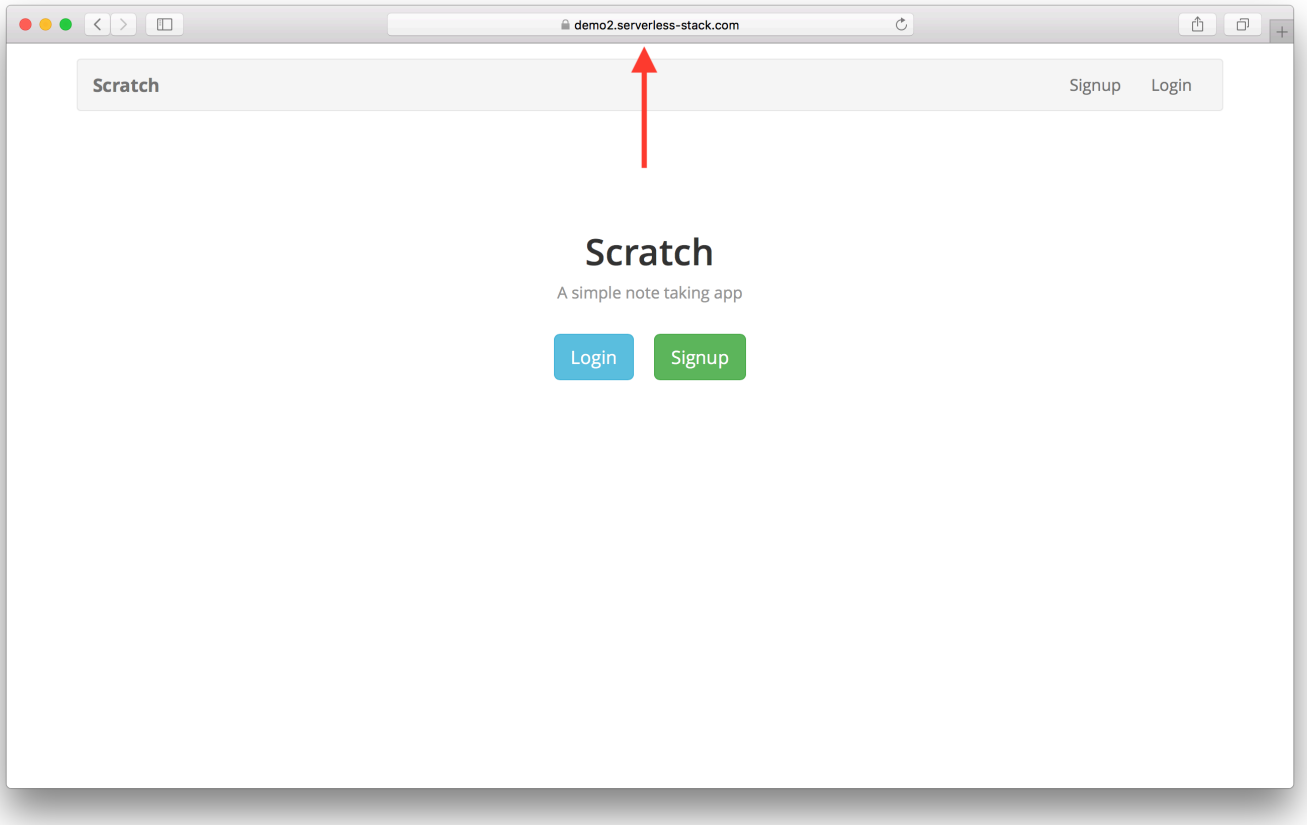
Once that is complete, Netlify will automatically provision your SSL ceritificate using Let's Encrypt.



Wait a few seconds for the ceritificate to be provisioned.



Now if you head over to your browser and go to your custom domain, your notes app should be up and running!



We have our app in production but we haven't had a chance to go through our workflow just yet. Let's take a look at that next.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/custom-domains-in-netlify/191)

Frontend Workflow

Now that we have our frontend deployed and configured, let's go over what our development workflow will look like.

Working in a Dev Branch

A good practise is to create a branch when we are working on something new.

✦ CHANGE Run the following in the root of your project.

```
$ git checkout -b "new-feature"
```

This creates a new branch for us called `new-feature`.

Let's make a couple of quick changes to test the process of deploying updates to our app.

We are going to add a Login and Signup button to our lander to give users a clear call to action.

✦ CHANGE To do this update our `renderLander` function in `src/containers/Home.js`.

```
function renderLander() {
  return (
    <div className="lander">
```

```
      <h1>Scratch</h1>
      <p>A simple note taking app</p>
      <div>
        <Link to="/login" className="btn btn-info btn-
lg">
          Login
        </Link>
        <Link to="/signup" className="btn btn-success
btn-lg">
          Signup
        </Link>
      </div>
    </div>
  );
}
```

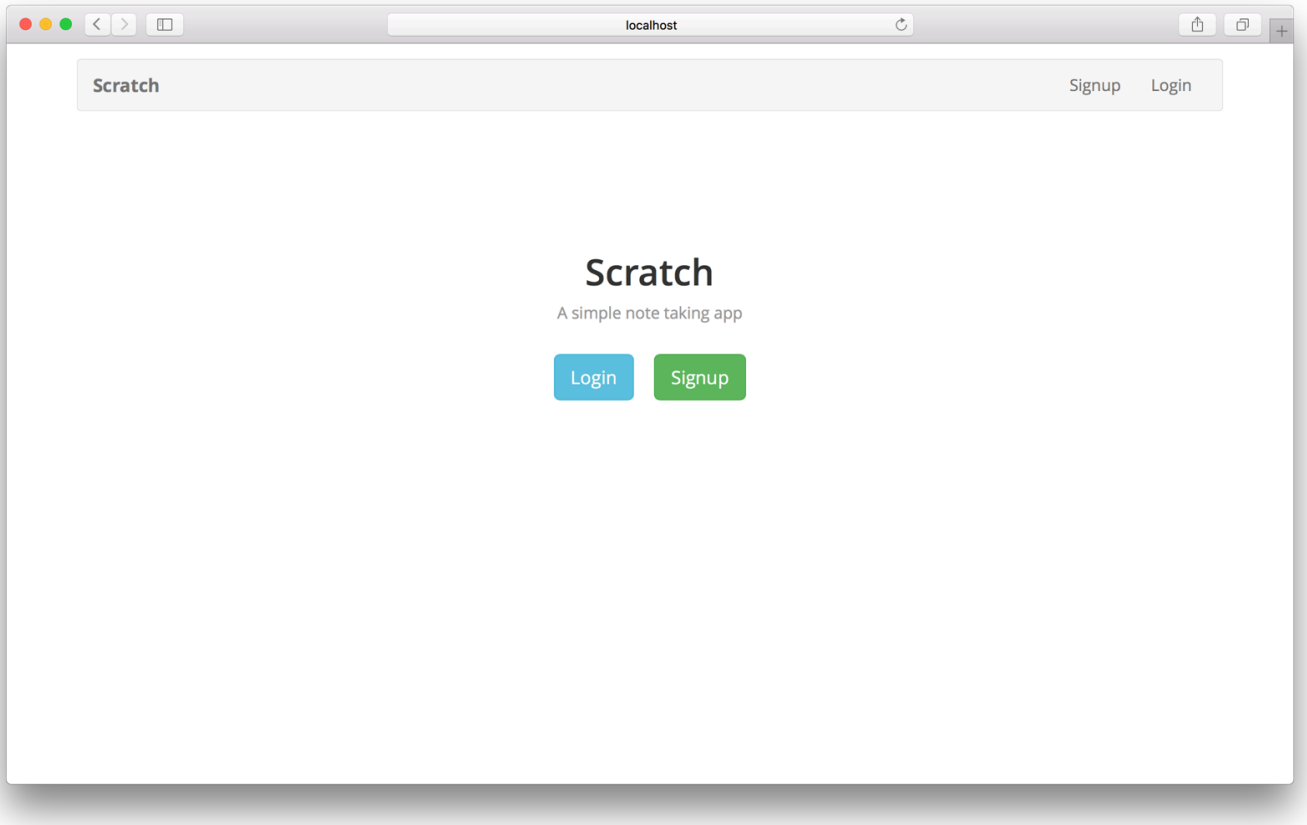
CHANGE And import the `Link` component from React-Router in the header.

```
import { Link } from "react-router-dom";
```

CHANGE Also, add a couple of styles to `src/containers/Home.css`.

```
.Home .lander div {
  padding-top: 20px;
}
.Home .lander div a:first-child {
  margin-right: 20px;
}
```

And our lander should look something like this.

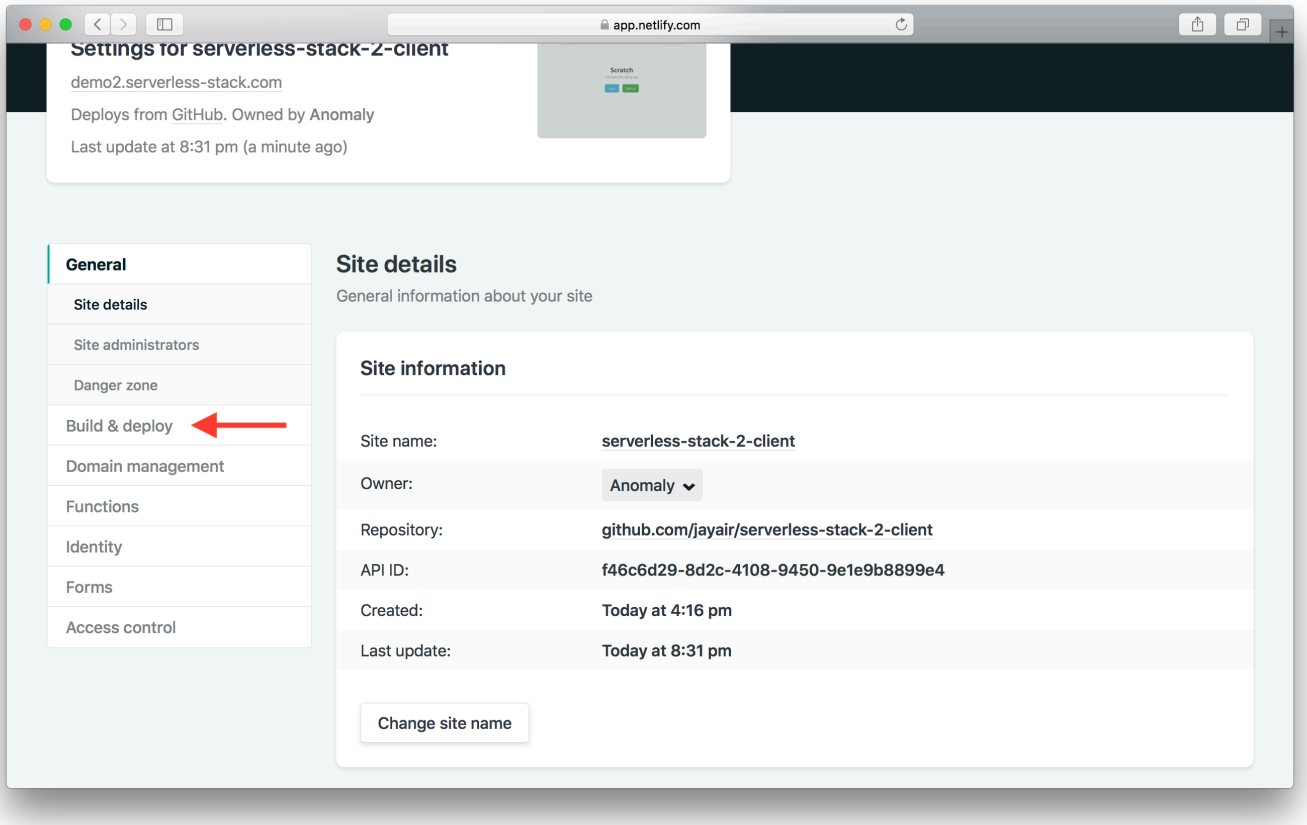


CHANGE Let's commit these changes to Git.

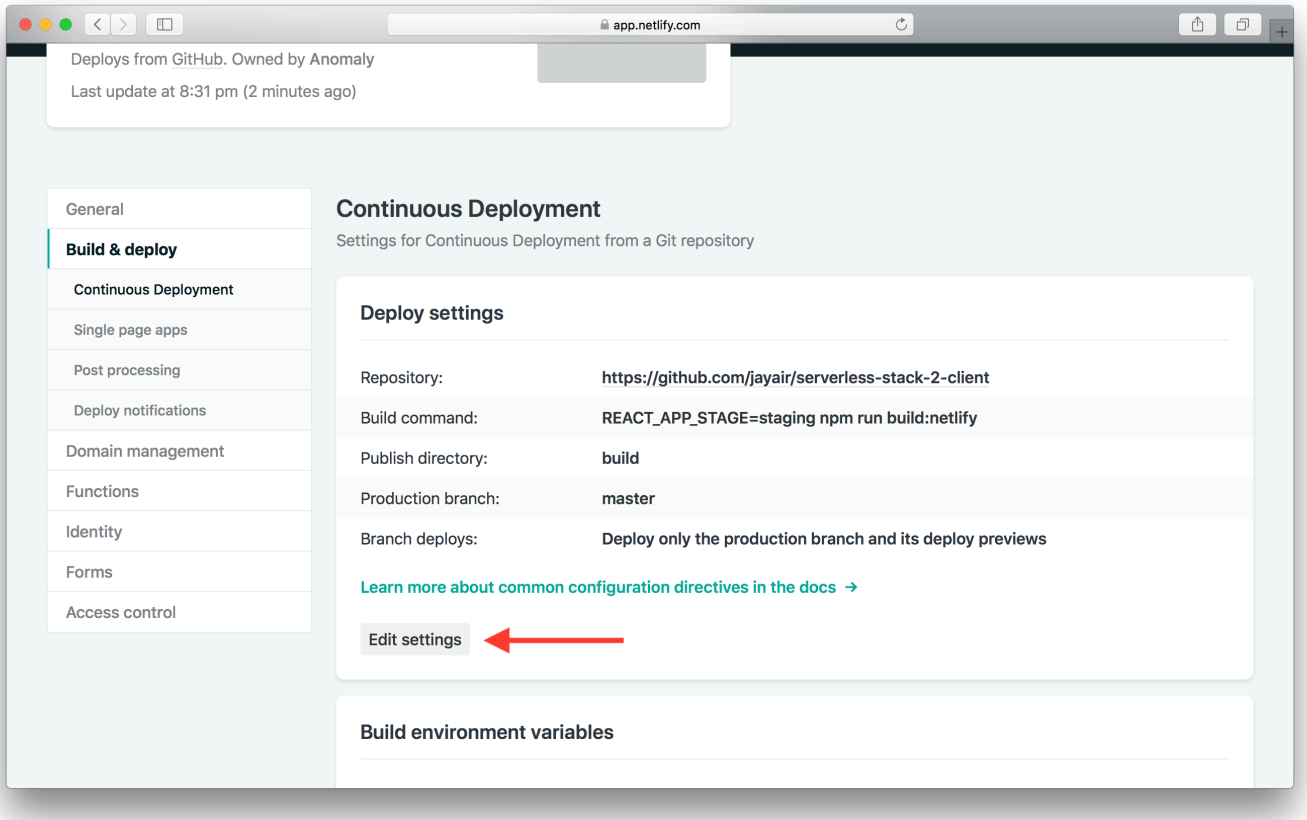
```
$ git add .
$ git commit -m "Updating the lander"
```

Create a Branch Deployment

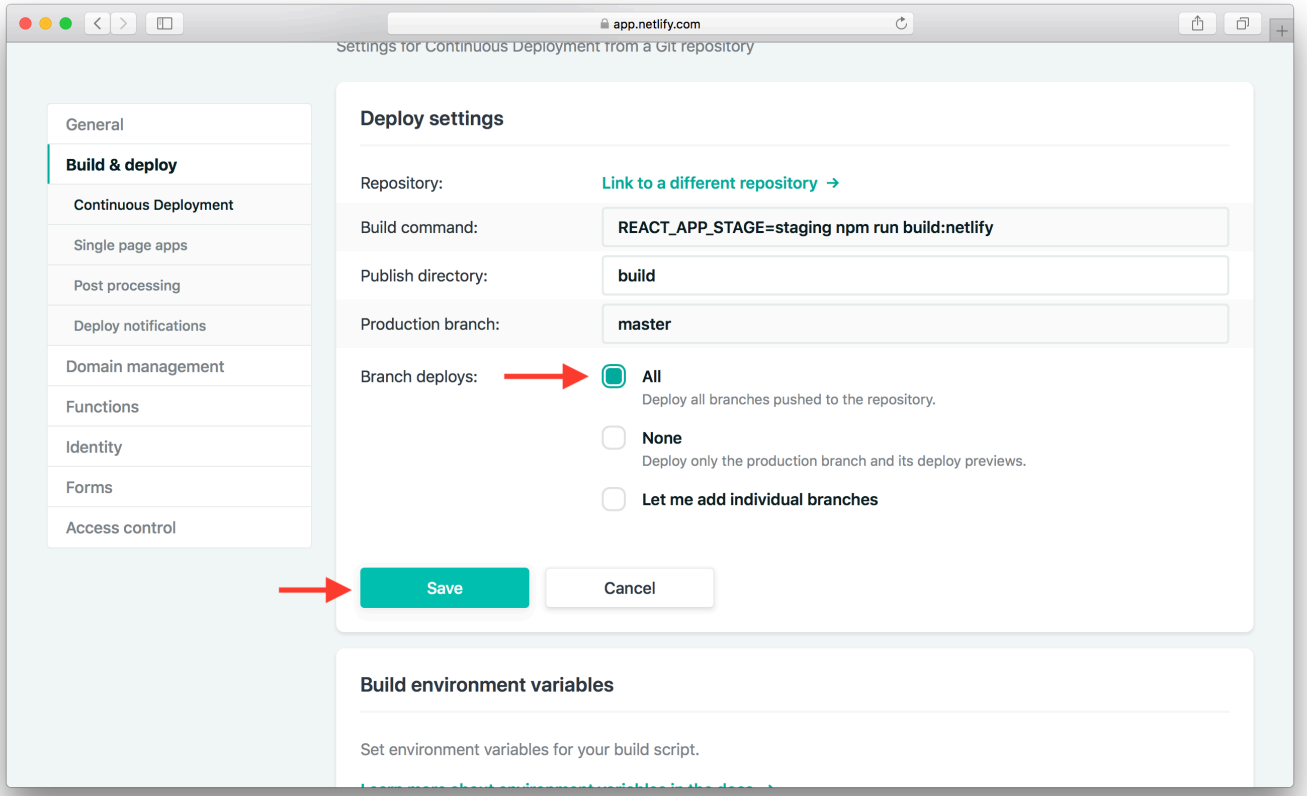
To be able to preview this change in its own environment we need to turn on branch deployments in Netlify. From the **Site settings** sidebar select **Build & deploy**.



And hit **Edit settings**.



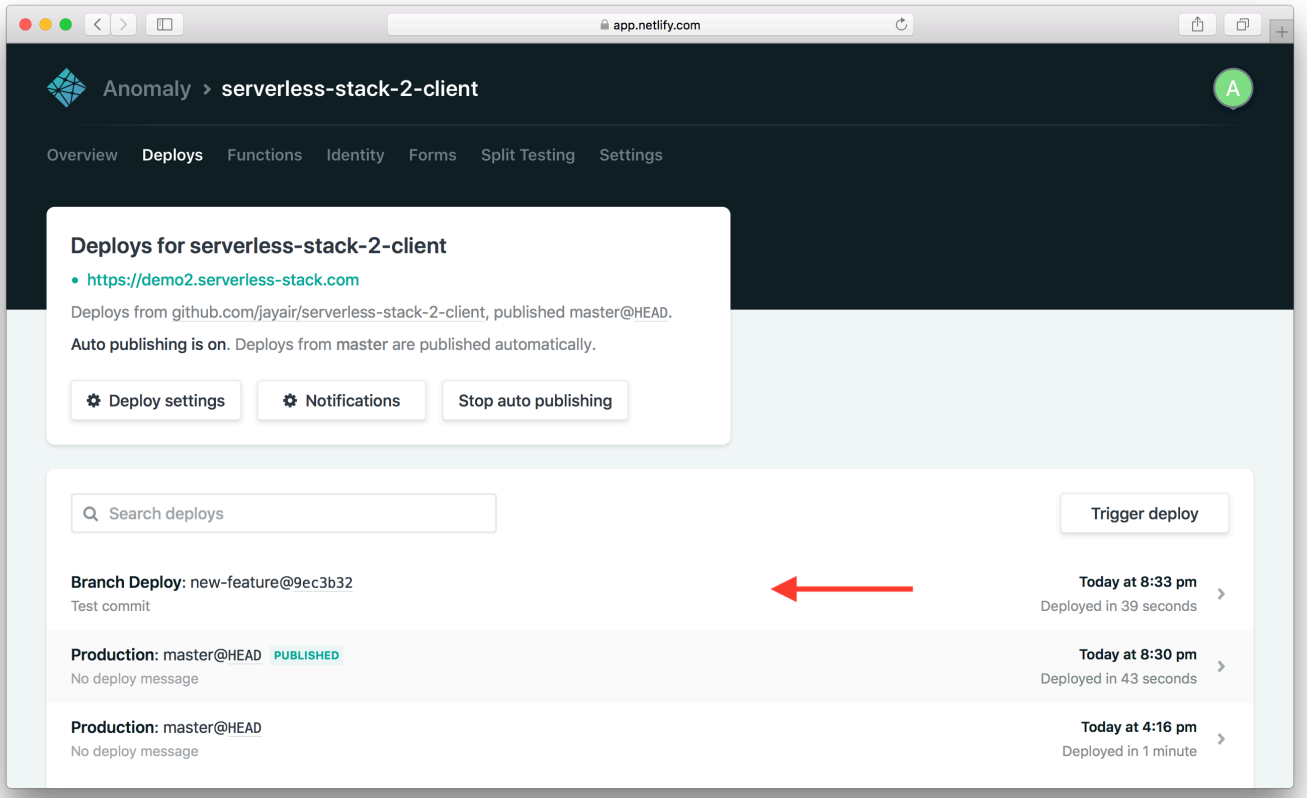
Set **Branch deploys** to **All** and hit **Save**.



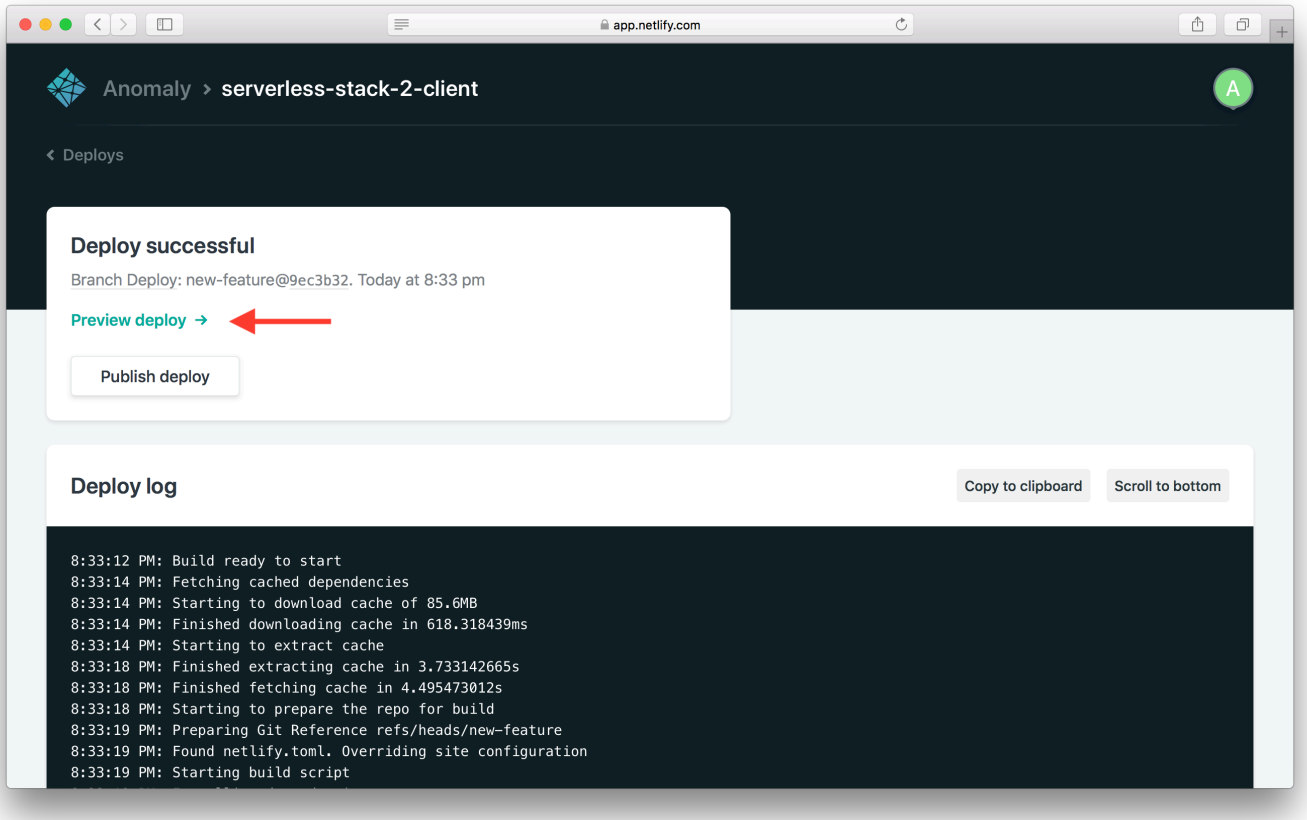
✦ CHANGE Now comes the fun part, we can deploy this to dev so we can test it right away. All we need to do is push it to Git.

```
$ git push -u origin new-feature
```

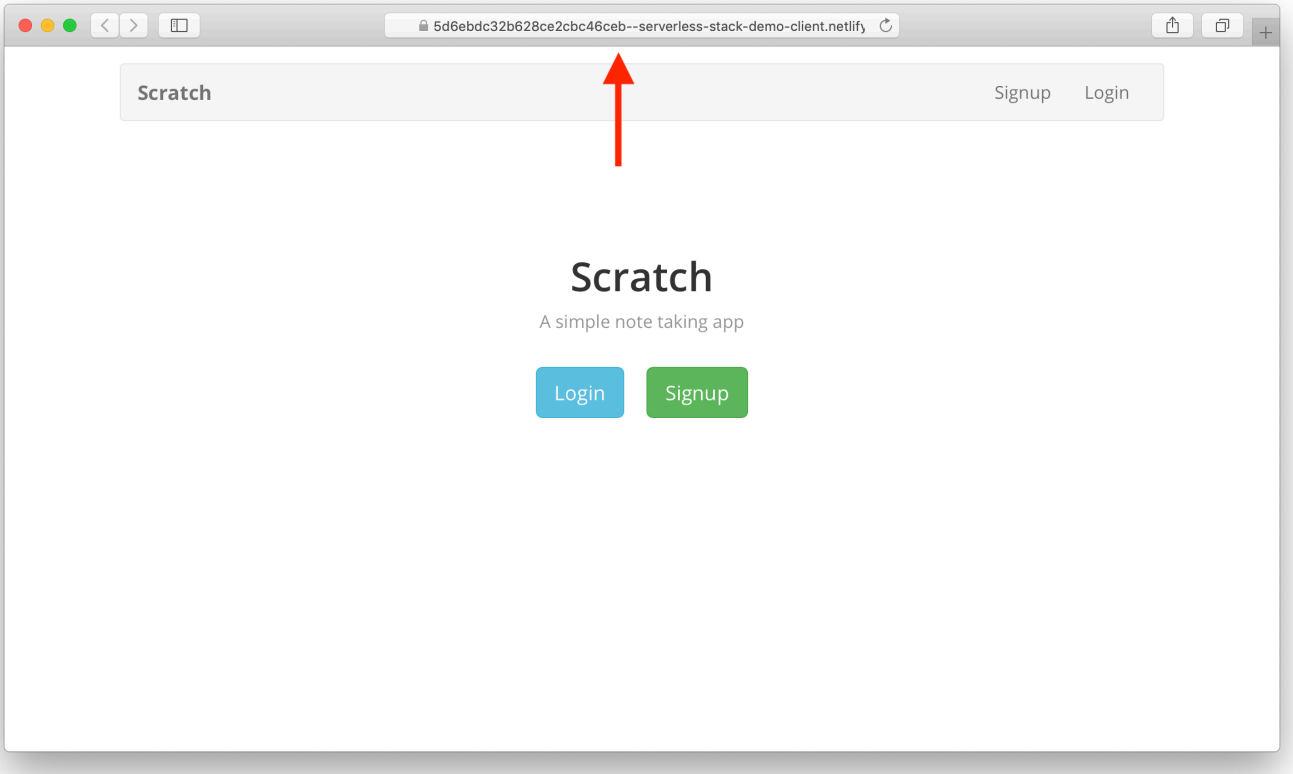
Now if you hop over to your Netlify project page; you'll see a new branch deploy in action. Wait for it to complete and click on it.



Hit **Preview deploy**.



And you can see a new version of your app in action!



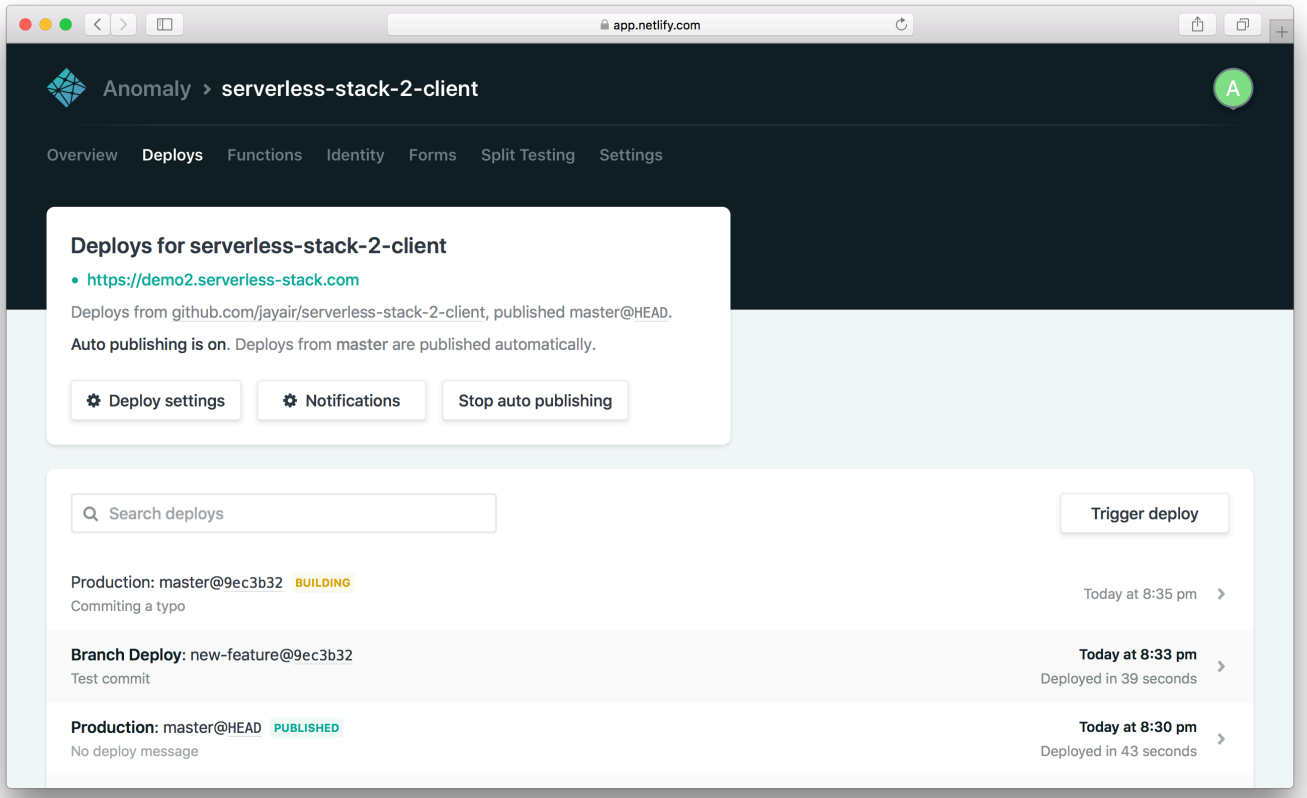
You can test around this version of our frontend app. It is connected to the dev version of our backend API. The idea is that we can test and play around with the changes here without affecting our production users.

Push to Production

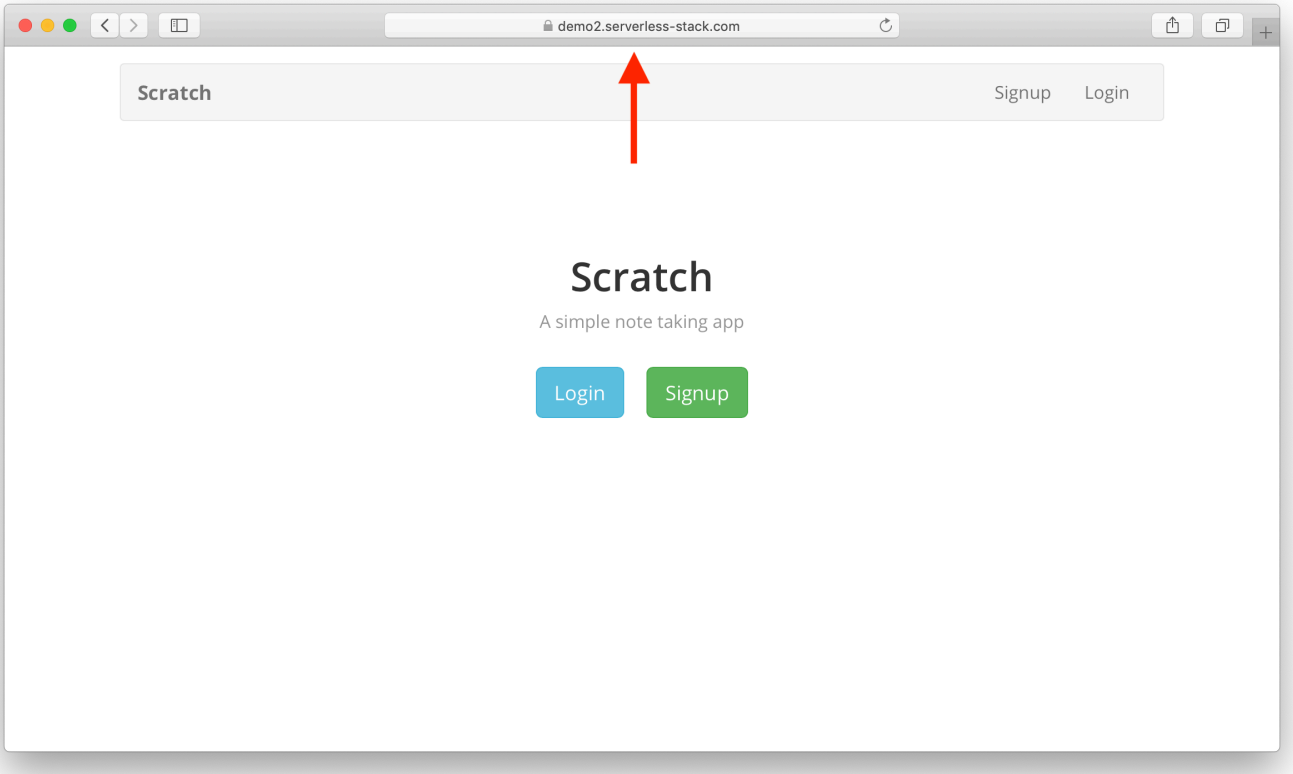
✦ CHANGE Now if we feel happy with the changes we can push this to production just by merging to master.

```
$ git checkout master
$ git merge new-feature
$ git push
```

You should see this deployment in action in Netlify.



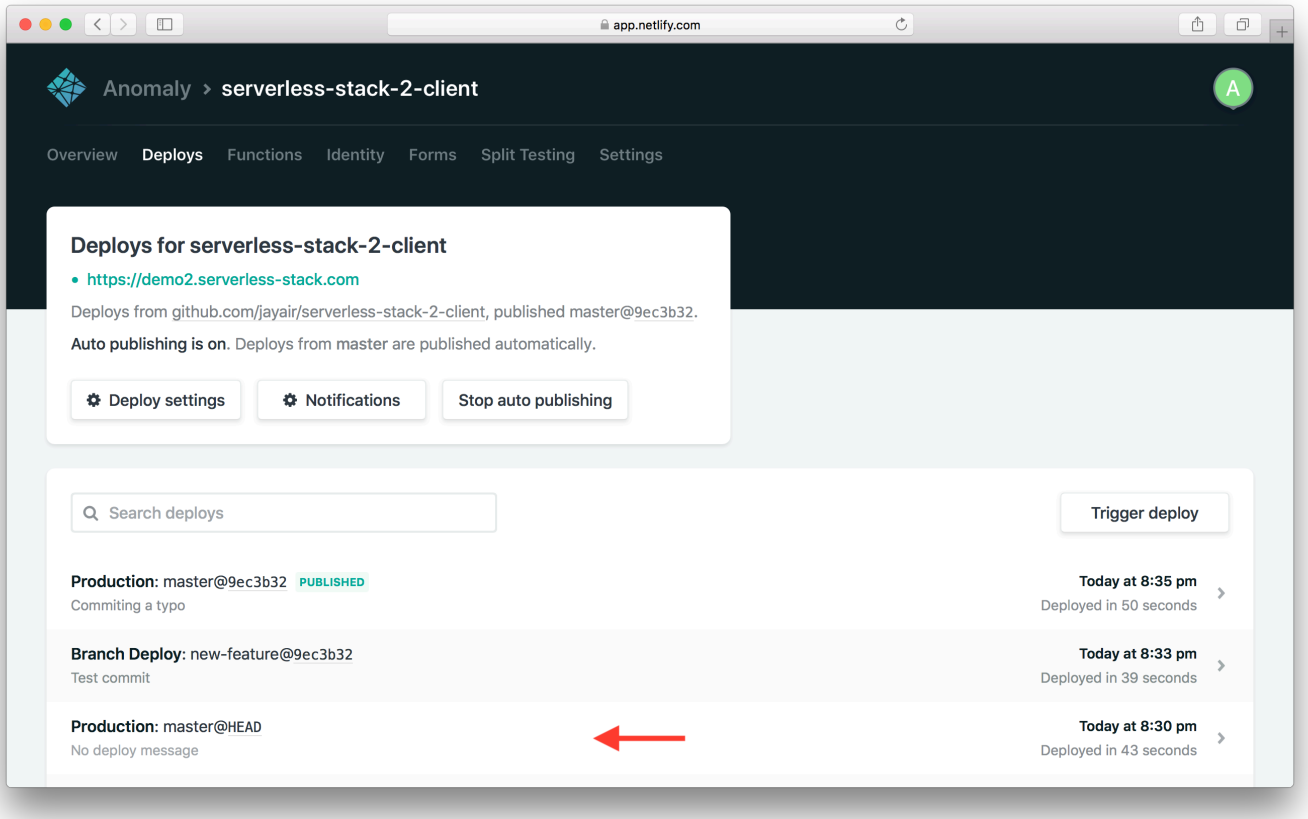
And once it is done, your changes should be live!



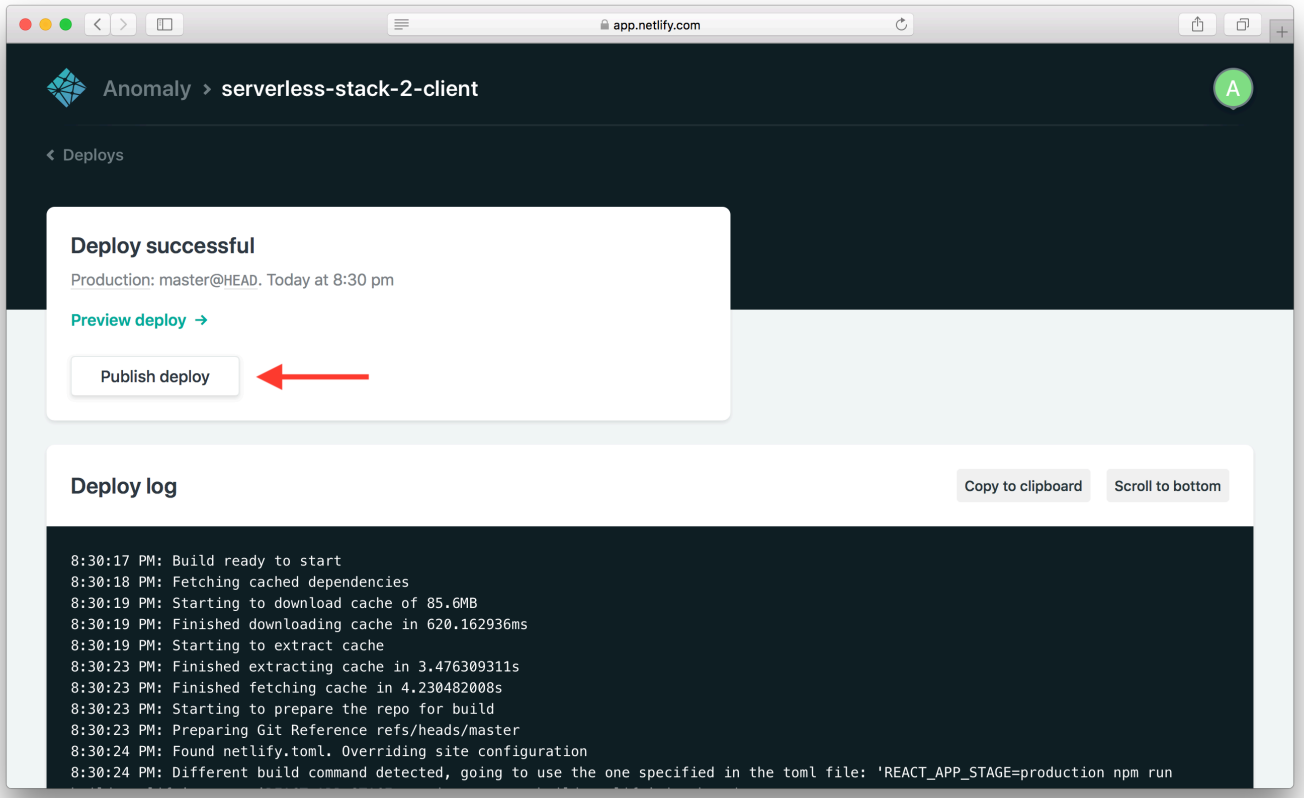
Rolling Back in Production

Now for some reason if we aren't happy with the build in production, we can rollback.

Click on the older production deployment.



And hit **Publish deploy**. This will publish our previous version again.



And that's it! Now you have an automated workflow for building and deploying your Create React App with serverless.

Let's wrap up the guide next.

CHECKPOINT

For reference, here is the complete code for the frontend.

`Frontend Source (https://github.com/AnomalyInnovations/serverless-stack-demo-client)`

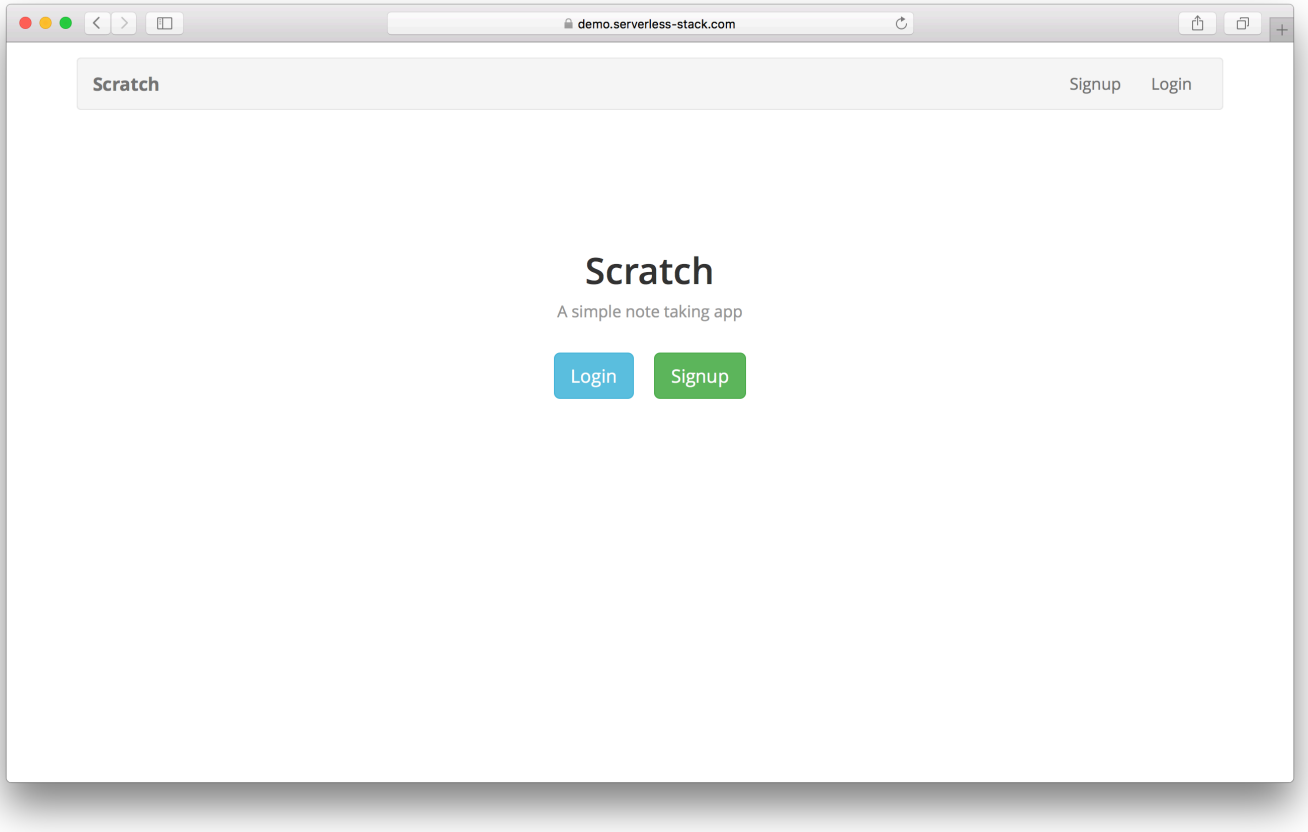
For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/frontend-workflow/192)

Wrapping Up

Congratulations on completing the guide!

We've covered how to build and deploy our backend serverless API and our frontend serverless app. And not only does it work well on the desktop.



It's mobile optimized as well!



We hope what you've learned here can be adapted to fit the use case you

have in mind. We are going to be covering a few other topics in the future while we keep this guide up to date.

If you are looking to use Serverless at your company and are wondering how best to structure a large Serverless project, make sure to check out our best practices section (/chapters/best-practices-for-building-serverless-apps.html). The entire workflow and the ideas covered are in production at a number of companies.

We'd love to hear from you about your experience following this guide. Please send us any comments or feedback you might have, via email (mailto:contact@anoma.ly). We'd love to feature your comments here. Also, if you'd like us to cover any of the chapters or concepts in a bit more detail, feel free to let us know (mailto:contact@anoma.ly).

Thank you and we hope you found this guide helpful!

For help and discussion

💬 Comments on this chapter (https://discourse.serverless-stack.com/t/comments-for-wrapping-up/100)

Further Reading

Once you've completed the guide, you are probably going to use the Serverless Stack for your next project. To help you along the way we try to compile a list of docs that you can use as a reference. The following can be used to drill down in detail for some of the technologies and services used in this guide.

- Serverless Framework Documentation (https://serverless.com/framework/docs/): Documentation for the Serverless Framework
- DynamoDB, explained (https://www.dynamodbguide.com): A Primer on the DynamoDB NoSQL database
- React JS Docs (https://reactjs.org/docs/hello-world.html): The official React docs
- JSX In Depth (https://reactjs.org/docs/jsx-in-depth.html): Learn JSX in a bit more detail
- Create React App User Guide (https://github.com/facebook/create-react-app/blob/master/packages/react-scripts/template/README.md): The really comprehensive Create React App user guide
- React-Bootstrap Docs (https://react-bootstrap.github.io/getting-started/introduction): The official React-Bootstrap docs

- Bootstrap v3 Docs (http://getbootstrap.com/docs/3.3/getting-started/): The Bootstrap v3 docs that React-Bootstrap is based on
- React Router Docs (https://reacttraining.com/react-router/web/guides/philosophy): The official React Router v4 docs
- AWS Amplify Developer Guide (https://aws.github.io/aws-amplify/media/developer_guide): The AWS Amplify developer guide
- AWS Amplify API Reference (https://aws.github.io/aws-amplify/api/): The AWS Amplify API reference
- AWS CloudFormation Docs (https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/GettingStarted.Walkthrough.html): The AWS user guide for CloudFormation
- Jest Unit Test Docs (https://facebook.github.io/jest/docs/en/getting-started.html): The official Jest docs
- Seed Docs (https://seed.run/docs/): The official Seed docs
- Netlify Docs (https://www.netlify.com/docs/): The official Netlify docs

If you have found any other guides or tutorials helpful in building your serverless app, feel free to edit this page and submit a PR. Or you can let us know via the comments.

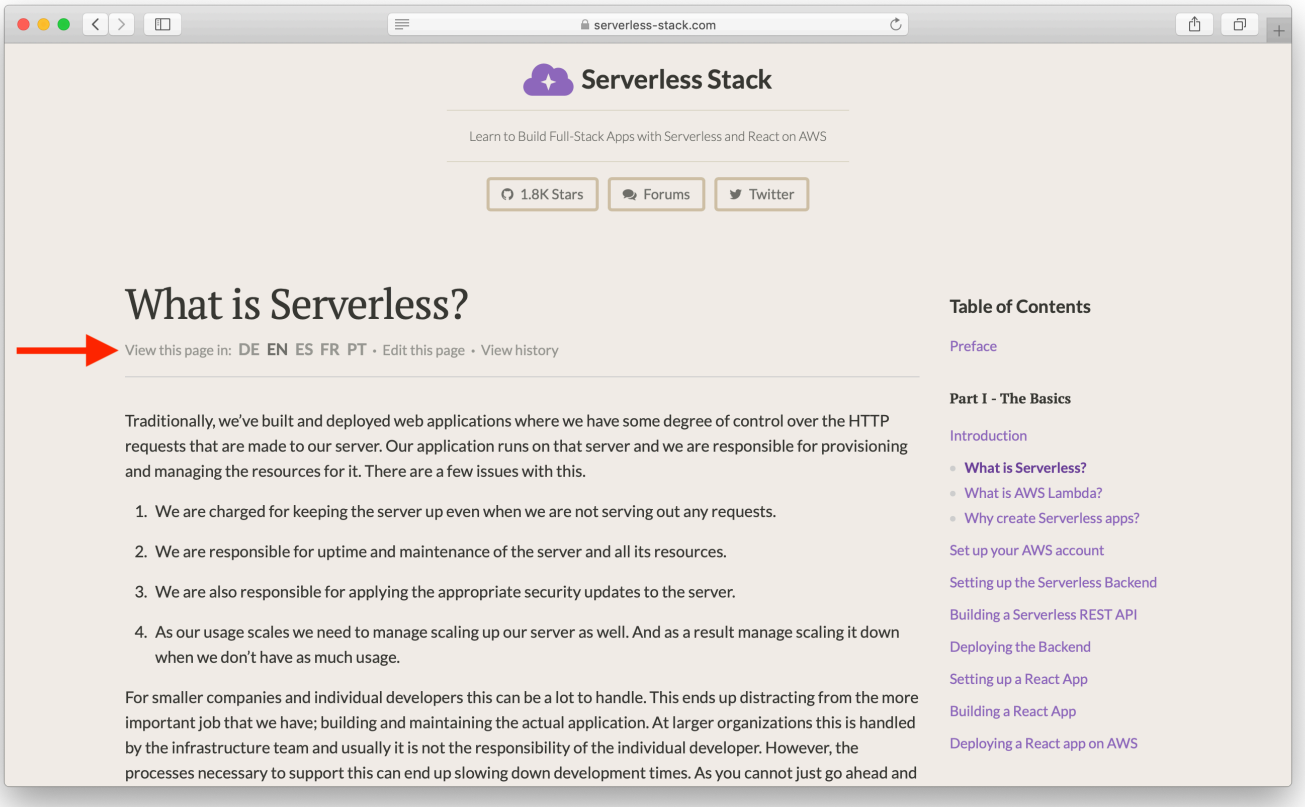
For help and discussion

Comments on this chapter (https://discourse.serverless-

```
stack.com/t/further-reading/209)
```

Translations

Our guide is available in several languages thanks to contributions by our incredible readers. You can view the translated versions of a chapter by clicking on the links below the chapter title.



Below is a list of all the chapters that are available in multiple languages. If you are interested in helping with our translation efforts, leave us a comment here (https://discourse.serverless-stack.com/t/help-us-translate-serverless-stack/596/15).

Who Is This Guide For? (/chapters/who-is-this-guide-for.html)

- es: ¿Para quién es esta guía? (/chapters/es/who-is-this-guide-for.html)
- fr: Pour qui est ce guide? (/chapters/fr/who-is-this-guide-for.html)
- ko: Who Is This Guide For? (/chapters/ko/who-is-this-guide-for.html)
- pt: Para quem é este guia? (/chapters/pt/who-is-this-guide-for.html)

What Does This Guide Cover? (/chapters/what-does-this-guide-cover.html)

- es: ¿Qué cubre esta guía? (/chapters/es/what-does-this-guide-cover.html)
- fr: Que couvre ce guide? (/chapters/fr/what-does-this-guide-cover.html)
- ko: What Does This Guide Cover? (/chapters/ko/what-does-this-guide-cover.html)
- pt: O que esse guia cobre? (/chapters/pt/what-does-this-guide-cover.html)

How to Get Help? (/chapters/how-to-get-help.html)

- es: ¿Cómo conseguir ayuda? (/chapters/es/how-to-get-help.html)
- fr: Comment obtenir de l'aide? (/chapters/fr/how-to-get-help.html)
- ko: How to Get Help? (/chapters/ko/how-to-get-help.html)
- pt: Como conseguir ajuda? (/chapters/pt/how-to-get-help.html)

What is Serverless? (/chapters/what-is-serverless.html)

- de: Was ist Serverless? (/chapters/de/what-is-serverless.html)
- es: ¿Qué es Serverless? (/chapters/es/what-is-serverless.html)
- fr: Serverless, c'est quoi ? (/chapters/fr/what-is-serverless.html)
- id: Apa itu Serverless? (/chapters/id/what-is-serverless.html)
- ko: What is Serverless? (/chapters/ko/what-is-serverless.html)
- pt: O que é Serverless? (/chapters/pt/what-is-serverless.html)

What is AWS Lambda? (/chapters/what-is-aws-lambda.html)

- es: ¿Qué es AWS Lambda? (/chapters/es/what-is-aws-lambda.html)
- fr: Qu'est-ce qu'AWS Lambda ? (/chapters/fr/what-is-aws-lambda.html)
- ko: What is AWS Lambda? (/chapters/ko/what-is-aws-lambda.html)
- pt: O que é AWS Lambda? (/chapters/pt/what-is-aws-lambda.html)

Why Create Serverless Apps? (/chapters/why-create-serverless-apps.html)
- es: ¿Por qué crear aplicaciones serverless? (/chapters/es/why-create-serverless-apps.html)
- ko: Why Create Serverless Apps? (/chapters/ko/why-create-serverless-apps.html)
- pt: Por que criar aplicações Serverless? (/chapters/pt/why-create-serverless-apps.html)

Create an AWS Account (/chapters/create-an-aws-account.html)
- es: Crear una cuenta de AWS (/chapters/es/create-an-aws-account.html)
- ko: Create an AWS Account (/chapters/ko/create-an-aws-account.html)
- pt: Crie uma conta na AWS (/chapters/pt/create-an-aws-account.html)

Create an IAM User (/chapters/create-an-iam-user.html)
- es: Crear un usuario de IAM (/chapters/es/create-an-iam-user.html)
- ko: Create an IAM User (/chapters/ko/create-an-iam-user.html)

What is IAM (/chapters/what-is-iam.html)
- es: ¿Qué es IAM? (/chapters/es/what-is-iam.html)
- ko: What is IAM (/chapters/ko/what-is-iam.html)

What is an ARN (/chapters/what-is-an-arn.html)
- es: ¿Qué es un ARN? (/chapters/es/what-is-an-arn.html)
- ko: What is an ARN (/chapters/ko/what-is-an-arn.html)

Configure the AWS CLI (/chapters/configure-the-aws-cli.html)

- es: Configurar AWS CLI (/chapters/es/configure-the-aws-cli.html)
- ko: Configure the AWS CLI (/chapters/ko/configure-the-aws-cli.html)

Create a DynamoDB Table (/chapters/create-a-dynamodb-table.html)

- es: Crear una tabla de DynamoDB (/chapters/es/create-a-dynamodb-table.html)
- ko: Create a DynamoDB Table (/chapters/ko/create-a-dynamodb-table.html)

Create an S3 Bucket for File Uploads (/chapters/create-an-s3-bucket-for-file-uploads.html)

- ko: Create an S3 Bucket for File Uploads (/chapters/ko/create-an-s3-bucket-for-file-uploads.html)

Create a Cognito User Pool (/chapters/create-a-cognito-user-pool.html)

- ko: Create a Cognito User Pool (/chapters/ko/create-a-cognito-user-pool.html)

Create a Cognito Test User (/chapters/create-a-cognito-test-user.html)

- ko: Create a Cognito Test User (/chapters/ko/create-a-cognito-test-user.html)

Set up the Serverless Framework (/chapters/setup-the-serverless-framework.html)

- ko: Set up the Serverless Framework (/chapters/ko/setup-the-serverless-framework.html)

Add Support for ES6/ES7 JavaScript (/chapters/add-support-for-es6-es7-javascript.html)

- ko: Add Support for ES6/ES7 JavaScript (/chapters/ko/add-support-for-es6-es7-javascript.html)

Initialize the Backend Repo (/chapters/initialize-the-backend-repo.html)

- ko: Initialize the Backend Repo (/chapters/ko/initialize-the-backend-repo.html)

Add a Create Note API (/chapters/add-a-create-note-api.html)

- ko: Add a Create Note API (/chapters/ko/add-a-create-note-api.html)

Add a Get Note API (/chapters/add-a-get-note-api.html)

- ko: Add a Get Note API (/chapters/ko/add-a-get-note-api.html)

Add a List All the Notes API (/chapters/add-a-list-all-the-notes-api.html)

- ko: Add a List All the Notes API (/chapters/ko/add-a-list-all-the-notes-api.html)

Add an Update Note API (/chapters/add-an-update-note-api.html)

- ko: Add an Update Note API (/chapters/ko/add-an-update-note-api.html)

Add a Delete Note API (/chapters/add-a-delete-note-api.html)

- ko: Add a Delete Note API (/chapters/ko/add-a-delete-note-api.html)

Handle API Gateway CORS Errors (/chapters/handle-api-gateway-cors-errors.html)

- ko: Handle API Gateway CORS Errors (/chapters/ko/handle-api-gateway-cors-errors.html)

Deploy the APIs (/chapters/deploy-the-apis.html)

- ko: Deploy the APIs (/chapters/ko/deploy-the-apis.html)

Create a Cognito Identity Pool (/chapters/create-a-cognito-identity-pool.html)

- ko: Create a Cognito Identity Pool (/chapters/ko/create-a-cognito-identity-pool.html)

Cognito User Pool vs Identity Pool (/chapters/cognito-user-pool-vs-identity-pool.html)

- ko: Cognito User Pool vs Identity Pool (/chapters/ko/cognito-user-pool-vs-identity-pool.html)

Test the APIs (/chapters/test-the-apis.html)

- ko: Test the APIs (/chapters/ko/test-the-apis.html)

Create a New React.js App (/chapters/create-a-new-reactjs-app.html)

- ko: Create a New React.js App (/chapters/ko/create-a-new-reactjs-app.html)

Add App Favicons (/chapters/add-app-favicons.html)

- ko: Add App Favicons (/chapters/ko/add-app-favicons.html)

Set up Custom Fonts (/chapters/setup-custom-fonts.html)

- ko: Set up Custom Fonts (/chapters/ko/setup-custom-fonts.html)

Set up Bootstrap (/chapters/setup-bootstrap.html)

- ko: Set up Bootstrap (/chapters/ko/setup-bootstrap.html)

Handle Routes with React Router (/chapters/handle-routes-with-react-router.html)

- ko: Handle Routes with React Router (/chapters/ko/handle-routes-with-react-router.html)

Create Containers (/chapters/create-containers.html)

- ko: Create Containers (/chapters/ko/create-containers.html)

Adding Links in the Navbar (/chapters/adding-links-in-the-navbar.html)

- ko: Adding Links in the Navbar (/chapters/ko/adding-links-in-the-navbar.html)

Handle 404s (/chapters/handle-404s.html)

- ko: Handle 404s (/chapters/ko/handle-404s.html)

Configure AWS Amplify (/chapters/configure-aws-amplify.html)
- ko: Configure AWS Amplify (/chapters/ko/configure-aws-amplify.html)

Login with AWS Cognito (/chapters/login-with-aws-cognito.html)
- ko: Login with AWS Cognito (/chapters/ko/login-with-aws-cognito.html)

Load the State from the Session (/chapters/load-the-state-from-the-session.html)
- ko: Load the State from the Session (/chapters/ko/load-the-state-from-the-session.html)

Clear the Session on Logout (/chapters/clear-the-session-on-logout.html)
- ko: Clear the Session on Logout (/chapters/ko/clear-the-session-on-logout.html)

Redirect on Login and Logout (/chapters/redirect-on-login-and-logout.html)
- ko: Redirect on Login and Logout (/chapters/ko/redirect-on-login-and-logout.html)

Give Feedback While Logging In (/chapters/give-feedback-while-logging-in.html)
- ko: Give Feedback While Logging In (/chapters/ko/give-feedback-while-logging-in.html)

Create a Signup Page (/chapters/create-a-signup-page.html)
- ko: Create a Signup Page (/chapters/ko/create-a-signup-page.html)

Create the Signup Form (/chapters/create-the-signup-form.html)
- ko: Create the Signup Form (/chapters/ko/create-the-signup-

form.html)

Signup with AWS Cognito (/chapters/signup-with-aws-cognito.html)
- ko: Signup with AWS Cognito (/chapters/ko/signup-with-aws-cognito.html)

Add the Create Note Page (/chapters/add-the-create-note-page.html)
- ko: Add the Create Note Page (/chapters/ko/add-the-create-note-page.html)

Call the Create API (/chapters/call-the-create-api.html)
- ko: Call the Create API (/chapters/ko/call-the-create-api.html)

Upload a File to S3 (/chapters/upload-a-file-to-s3.html)
- ko: Upload a File to S3 (/chapters/ko/upload-a-file-to-s3.html)

List All the Notes (/chapters/list-all-the-notes.html)
- ko: List All the Notes (/chapters/ko/list-all-the-notes.html)

Call the List API (/chapters/call-the-list-api.html)
- ko: Call the List API (/chapters/ko/call-the-list-api.html)

Display a Note (/chapters/display-a-note.html)
- ko: Display a Note (/chapters/ko/display-a-note.html)

Render the Note Form (/chapters/render-the-note-form.html)
- ko: Render the Note Form (/chapters/ko/render-the-note-form.html)

Save Changes to a Note (/chapters/save-changes-to-a-note.html)
- ko: Save Changes to a Note (/chapters/ko/save-changes-to-a-note.html)

Delete a Note (/chapters/delete-a-note.html)
- ko: Delete a Note (/chapters/ko/delete-a-note.html)

Create a Settings Page (/chapters/create-a-settings-page.html)

- ko: Create a Settings Page (/chapters/ko/create-a-settings-page.html)

Add Stripe Keys to Config (/chapters/add-stripe-keys-to-config.html)

- ko: Add Stripe Keys to Config (/chapters/ko/add-stripe-keys-to-config.html)

Create a Billing Form (/chapters/create-a-billing-form.html)

- ko: Create a Billing Form (/chapters/ko/create-a-billing-form.html)

Connect the Billing Form (/chapters/connect-the-billing-form.html)

- ko: Connect the Billing Form (/chapters/ko/connect-the-billing-form.html)

Set up Secure Pages (/chapters/setup-secure-pages.html)

- ko: Set up Secure Pages (/chapters/ko/setup-secure-pages.html)

Create a Route That Redirects (/chapters/create-a-route-that-redirects.html)

- ko: Create a Route That Redirects (/chapters/ko/create-a-route-that-redirects.html)

Use the Redirect Routes (/chapters/use-the-redirect-routes.html)

- ko: Use the Redirect Routes (/chapters/ko/use-the-redirect-routes.html)

Redirect on Login (/chapters/redirect-on-login.html)

- ko: Redirect on Login (/chapters/ko/redirect-on-login.html)

Deploy a React App to AWS (/chapters/deploying-a-react-app-to-aws.html)

- ko: Deploy the Frontend (/chapters/ko/deploy-the-frontend.html)

Create an S3 Bucket (/chapters/create-an-s3-bucket.html)

- ko: Create an S3 Bucket (/chapters/ko/create-an-s3-bucket.html)

Deploy to S3 (/chapters/deploy-to-s3.html)
- ko: Deploy to S3 (/chapters/ko/deploy-to-s3.html)

Create a CloudFront Distribution (/chapters/create-a-cloudfront-distribution.html)
- ko: Create a CloudFront Distribution (/chapters/ko/create-a-cloudfront-distribution.html)

Set up Your Domain with CloudFront (/chapters/setup-your-domain-with-cloudfront.html)
- ko: Set up Your Domain with CloudFront (/chapters/ko/setup-your-domain-with-cloudfront.html)

Set up WWW Domain Redirect (/chapters/setup-www-domain-redirect.html)
- ko: Set up WWW Domain Redirect (/chapters/ko/setup-www-domain-redirect.html)

Set up SSL (/chapters/setup-ssl.html)
- ko: Set up SSL (/chapters/ko/setup-ssl.html)

Deploy Updates (/chapters/deploy-updates.html)
- ko: Deploy Updates (/chapters/ko/deploy-updates.html)

Getting Production Ready (/chapters/getting-production-ready.html)
- ko: Getting Production Ready (/chapters/ko/getting-production-ready.html)

What Is Infrastructure as Code (/chapters/what-is-infrastructure-as-code.html)
- ko: What Is Infrastructure as Code (/chapters/ko/what-is-

infrastructure-as-code.html)

Configure DynamoDB in Serverless (/chapters/configure-dynamodb-in-serverless.html)
- ko: Configure DynamoDB in Serverless (/chapters/ko/configure-dynamodb-in-serverless.html)

Configure S3 in Serverless (/chapters/configure-s3-in-serverless.html)
- ko: Configure S3 in Serverless (/chapters/ko/configure-s3-in-serverless.html)

Configure Cognito User Pool in Serverless (/chapters/configure-cognito-user-pool-in-serverless.html)
- ko: Configure Cognito User Pool in Serverless (/chapters/ko/configure-cognito-user-pool-in-serverless.html)

Configure Cognito Identity Pool in Serverless (/chapters/configure-cognito-identity-pool-in-serverless.html)
- ko: Configure Cognito Identity Pool in Serverless (/chapters/ko/configure-cognito-identity-pool-in-serverless.html)

Deploy Your Serverless Infrastructure (/chapters/deploy-your-serverless-infrastructure.html)
- ko: Deploy Your Serverless Infrastructure (/chapters/ko/deploy-your-serverless-infrastructure.html)

Working with 3rd Party APIs (/chapters/working-with-3rd-party-apis.html)
- ko: Working with 3rd Party APIs (/chapters/ko/working-with-3rd-party-apis.html)

Setup a Stripe Account (/chapters/setup-a-stripe-account.html)
- ko: Setup a Stripe Account (/chapters/ko/setup-a-stripe-account.html)

Add a Billing API (/chapters/add-a-billing-api.html)

- ko: Add a Billing API (/chapters/ko/add-a-billing-api.html)

Load Secrets from .env (/chapters/load-secrets-from-env.html)

- ko: Load Secrets from env.yml (/chapters/ko/load-secrets-from-env-yml.html)

Test the Billing API (/chapters/test-the-billing-api.html)

- ko: Test the Billing API (/chapters/ko/test-the-billing-api.html)

Unit Tests in Serverless (/chapters/unit-tests-in-serverless.html)

- ko: Unit Tests in Serverless (/chapters/ko/unit-tests-in-serverless.html)

Automating Serverless Deployments (/chapters/automating-serverless-deployments.html)

- ko: Automating Serverless Deployments (/chapters/ko/automating-serverless-deployments.html)

Setting up Your Project on Seed (/chapters/setting-up-your-project-on-seed.html)

- ko: Setting up Your Project on Seed (/chapters/ko/setting-up-your-project-on-seed.html)

Configure Secrets in Seed (/chapters/configure-secrets-in-seed.html)

- ko: Configure Secrets in Seed (/chapters/ko/configure-secrets-in-seed.html)

Deploying Through Seed (/chapters/deploying-through-seed.html)

- ko: Deploying Through Seed (/chapters/ko/deploying-through-seed.html)

Set Custom Domains Through Seed (/chapters/set-custom-domains-through-seed.html)

- ko: Set Custom Domains Through Seed (/chapters/ko/set-custom-domains-through-seed.html)

Test the Configured APIs (/chapters/test-the-configured-apis.html)
- ko: Test the Configured APIs (/chapters/ko/test-the-configured-apis.html)

Monitoring Deployments in Seed (/chapters/monitoring-deployments-in-seed.html)
- ko: Monitoring Deployments in Seed (/chapters/ko/monitoring-deployments-in-seed.html)

Initialize the Frontend Repo (/chapters/initialize-the-frontend-repo.html)
- ko: Initialize the Frontend Repo (/chapters/ko/initialize-the-frontend-repo.html)

Manage Environments in Create React App (/chapters/manage-environments-in-create-react-app.html)
- ko: Manage Environments in Create React App (/chapters/ko/manage-environments-in-create-react-app.html)

Automating React Deployments (/chapters/automating-react-deployments.html)
- ko: Automating React Deployments (/chapters/ko/automating-react-deployments.html)

Create a Build Script (/chapters/create-a-build-script.html)
- ko: Create a Build Script (/chapters/ko/create-a-build-script.html)

Setting up Your Project on Netlify (/chapters/setting-up-your-project-on-netlify.html)
- ko: Setting up Your Project on Netlify (/chapters/ko/setting-up-your-project-on-netlify.html)

Custom Domains in Netlify (/chapters/custom-domain-in-netlify.html)

- ko: Netlify에서 사용자 정의 도메인 설정 (/chapters/ko/custom-domain-in-netlify.html)

Frontend Workflow (/chapters/frontend-workflow.html)

- ko: 프론트엔드 작업흐름 (/chapters/ko/frontend-workflow.html)

Wrapping Up (/chapters/wrapping-up.html)

- ko: 마무리하며 (/chapters/ko/wrapping-up.html)

Further Reading (/chapters/further-reading.html)

- ko: 더 읽어볼 것들 (/chapters/ko/further-reading.html)

Translations (/chapters/translations.html)

- ko: 번역 (/chapters/ko/translations.html)

Giving Back (/chapters/giving-back.html)

- ko: Giving Back (/chapters/ko/giving-back.html)

Changelog (/chapters/changelog.html)

- ko: 변경로그 (/chapters/ko/changelog.html)

Staying up to date (/chapters/staying-up-to-date.html)

- ko: 최신 상태 유지 (/chapters/ko/staying-up-to-date.html)

A big thanks to our contributors for helping make Serverless Stack more accessible!

- Bernardo Bugmann (https://github.com/bernardobugmann)
- Sebastian Gutierrez (https://github.com/pepas24)
- Vincent Oliveira (https://github.com/vincentoliveira)
- Leonardo Gonzalez (https://github.com/leogonzalez)

- Vieko Franetovic (https://github.com/vieko)
- Christian Kaindl (https://github.com/christiankaindl)
- Jae Chul Kim (https://github.com/bsg-bob)

For help and discussion

💬 Comments on this chapter (https://discourse.serverless-stack.com/t/comments-translations/788)

Giving Back

If you've found this guide helpful please consider helping us out by doing the following. You can also read about this in detail in our CONTRIBUTING.md (https://github.com/AnomalyInnovations/serverless-stack-com/blob/master/CONTRIBUTING.md).

- **Fixing typos and errors**

  The content on this site is kept up to date thanks in large part to our community and our readers. Submit a Pull Request (https://github.com/AnomalyInnovations/serverless-stack-com/compare) to fix any typos or errors you might find.

- **Helping others in the comments**

  If you've found yourself using the Discourse comments (https://discourse.serverless-stack.com) to get help, please consider helping anybody else with issues that you might have run into.

- **Keep the core guide updated**

  Serverless Stack is reliant on a large number of services and open source libraries and projects. The screenshots for the services and the dependencies need to be updated every once in a while. Here is a little more details on this (https://github.com/AnomalyInnovations/serverless-stack-com/blob/master/CONTRIBUTING.md#keep-the-core-guide-updated).

- **Help translate the guide**

  Our incredible readers are helping translate Serverless Stack into multiple languages. You can check out our progress here (/chapters/translations.html). If you would like to help with our translation efforts, leave us a comment here (https://discourse.serverless-stack.com/t/help-us-translate-serverless-stack/596/15).

- **Add an extra credit chapter**

  The core chapters are missing some extra details (for the sake of simplicity) that are necessary once you start customizing the Serverless Stack setup. Additionally, there are cases that we just don't handle in the core part of the guide. We are addressing these via *Extra Credit chapters*. If you have had a chance to extend Serverless Stack consider writing a chapter on it. Here are further details on how to add an extra credit chapter (https://github.com/AnomalyInnovations/serverless-stack-com/blob/master/CONTRIBUTING.md#add-an-extra-credit-chapter).

- **Improve tooling**

  Currently we do a lot of manual work to publish updates and maintain the tutorial. You can help by contributing to improve the process. Here are some more details on what we need help with (https://github.com/AnomalyInnovations/serverless-stack-com/blob/master/CONTRIBUTING.md#improve-tooling).

- **Give us a Star on GitHub** (https://github.com/AnomalyInnovations/serverless-stack-com)

  We rely on our GitHub repo for everything from hosting this site to

code samples and comments. Starring our repo (https://github.com/AnomalyInnovations/serverless-stack-com) helps us get the word out.

- **Sharing this guide**

  Share this guide via Twitter (https://twitter.com/intent/tweet?text=Serverless Stack&url=https://serverless-stack.com) or Facebook (https://www.facebook.com/sharer/sharer.php?u=https://serverless-stack.com&p[title]=Serverless Stack) with others that might find this helpful.

Also, if you have any other ideas on how to contribute; feel free to let us know via email (mailto:contact@anoma.ly).

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/comments-for-giving-back/193)

Changelog

As we continue to update Serverless Stack, we want to make sure that we give you a clear idea of all the changes that are being made. This is to ensure that you won't have to go through the entire tutorial again to get caught up on the updates. We also want to leave the older versions up in case you need a reference. This is also useful for readers who are working through the tutorial while it gets updated.

Below are the updates we've made to Serverless Stack, each with:

- Each update has a link to an **archived version of the tutorial**
- Updates to the tutorial **compared to the last version**
- Updates to the **API and Client repos**

While the hosted version of the tutorial and the code snippets are accurate, the sample project repo that is linked at the bottom of each chapter is unfortunately not. We do however maintain the past versions of the completed sample project repo. So you should be able to use those to figure things out. All this info is also available on the releases page (https://github.com/AnomalyInnovations/serverless-stack-com/releases) of our GitHub repo (https://github.com/AnomalyInnovations/serverless-stack-com).

You can get these updates emailed to you via our newsletter (https://emailoctopus.com/lists/1c11b9a8-1500-11e8-a3c9-06b79b628af2/forms/subscribe).

Changes

v4.0: New edition of Serverless Stack (https://branchv40--serverless-stack.netlify.com/) (Current)

Oct 8, 2019: Adding a new section for Serverless best practices. Updating to React Hooks. Reorganizing chapters. Updating backend to Node 10.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v3.4...v4.0)

v3.4: Updating to serverless-bundle and on-demand DynamoDB (https://branchv34--serverless-stack.netlify.com/)

Jul 18, 2019: Updating to serverless-bundle plugin and On-Demand Capacity for DynamoDB.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v3.3.3...v3.4)
- API (https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v3.3.3...v3.4)

v3.3.3: Handling API Gateway CORS errors (https://branchv333--serverless-stack.netlify.com/)

Jan 27, 2019: Adding CORS headers to API Gateway 4xx and 5xx errors.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v3.3.2...v3.3.3)
- API (https://github.com/AnomalyInnovations/serverless-stack-demo-

api/compare/v3.3.2...v3.3.3)

v3.3.2: Refactoring async Lambda functions (https://branchv332--serverless-stack.netlify.com)

Nov 1, 2018: Refactoring async Lambda functions to return instead of using the callback.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v3.3.1...v3.3.2)
- API (https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v3.1...v3.3.2)

v3.3.1: Updated to Create React App v2 (https://branchv331--serverless-stack.netlify.com)

Oct 5, 2018: Updated the frontend React app to use Create React App v2.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v3.3...v3.3.1)
- Client (https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v3.1...v3.3.1)

v3.3: Added new chapters (https://branchv33--serverless-stack.netlify.com)

Oct 5, 2018: Added new chapters on Facebook login with AWS Amplify and mapping Identity Id with User Pool Id. Also, added a new series of chapters on forgot password, change email and password.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v3.2...v3.3)

- Facebook Login Client (https://github.com/AnomalyInnovations/serverless-stack-demo-fb-login-client)
- User Management Client (https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client)

v3.2: Added section on Serverless architecture (https://branchv32--serverless-stack.netlify.com)

Aug 18, 2018: Adding a new section on organizing Serverless applications. Outlining how to use CloudFormation cross-stack references to link multiple Serverless services.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v3.1...v3.2)
- Monorepo API (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api)

v3.1: Update to use UsernameAttributes (https://branchv31--serverless-stack.netlify.com)

May 24, 2018: CloudFormation now supports UsernameAttributes. This means that we don't need the email as alias work around.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v3.0...v3.1)
- API (https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v3.0...v3.1)
- Client (https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v3.0...v3.1)

v3.0: Adding Part II (https://branchv30--serverless-stack.netlify.com)

May 10, 2018: Adding a new part to the guide to help create a production ready version of the note taking app. Discussion on the update (https://discourse.serverless-stack.com/t/serverless-stack-update-part-ii/194).

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v2.2...v3.0)
- API (https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v2.2...v3.0)
- Client (https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v2.0...v3.0)

v2.2: Updating to user Node.js starter and v8.10 (https://branchv22--serverless-stack.netlify.com)

Apr 11, 2018: Updating the backend to use Node.js starter and Lambda Node v8.10. Discussion on the update (https://github.com/AnomalyInnovations/serverless-stack-com/issues/223).

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v2.1...v2.2)
- API (https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v2.1...v2.2)

v2.1: Updating to Webpack 4 (https://branchv21--serverless-stack.netlify.com)

Mar 21, 2018: Updating the backend to use Webpack 4 and serverless-webpack 5.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v2.0...v2.1)
- API (https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v1.2.3...v2.1)

v2.0: AWS Amplify update (https://branchv20--serverless-stack.netlify.com)

Updating frontend to use AWS Amplify. Verifying SSL certificate now uses DNS validation. Discussion on the update (https://github.com/AnomalyInnovations/serverless-stack-com/issues/123).

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2.5...v2.0)
- Client (https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v1.2.5...v2.0)

v1.2.5: Using specific Bootstrap CSS version (https://branchv125--serverless-stack.netlify.com)

Feb 5, 2018: Using specific Bootstrap CSS version since `latest` now points to Bootstrap v4. But React-Bootstrap uses v3.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2.4...v1.2.5)
- Client (https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v1.2.4...v1.2.5)

v1.2.4: Updating to React 16 (https://5a4993f3a6188f5a88e0c777--serverless-stack.netlify.com/)

Dec 31, 2017: Updated to React 16 and fixed `sigv4Client.js` IE11 issue ({{ site.github_repo }}/issues/114#issuecomment-349938586).

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2.3...v1.2.4)
- Client (https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v1.2...v1.2.4)

v1.2.3: Updating to babel-preset-env (https://5a49938981987612a1279f--serverless-stack.netlify.com/)

Dec 30, 2017: Updated serverless backend to use babel-preset-env plugin and added a note to the Deploy to S3 chapter on reducing React app bundle size.

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2.2...v1.2.3)
- API (https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v1.2...v1.2.3)

v1.2.2: Adding new chapters (https://5a499324a6188f5a88e0c76d--serverless-stack.netlify.com/)

Dec 1, 2017: Added the following *Extra Credit* chapters.

1. Customize the Serverless IAM Policy
2. Environments in Create React App

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2.1...v1.2.2)

v1.2.1: Adding new chapters (https://5a4992e70b79b76fb0948300--serverless-stack.netlify.com/)

Oct 7, 2017: Added the following *Extra Credit* chapters.

1. API Gateway and Lambda Logs
2. Debugging Serverless API Issues
3. Serverless environment variables
4. Stages in Serverless Framework
5. Configure multiple AWS profiles

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2...v1.2.1)

v1.2: Upgrade to Serverless Webpack v3 (https://59caac9bcf321c5b78f2c3e2--serverless-stack.netlify.com/)

Sep 16, 2017: Upgrading serverless backend to using serverless-webpack plugin v3. The new version of the plugin changes some of the commands used to test the serverless backend. Discussion on the update (https://github.com/AnomalyInnovations/serverless-stack-com/issues/130).

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-

stack-com/compare/v1.1...v1.2)
- API (https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v1.1...v1.2)

v1.1: Improved Session Handling (https://59caae1e6f4c50416e86701d--serverless-stack.netlify.com/)

Aug 30, 2017: Fixing some issues with session handling in the React app. A few minor updates bundled together. Discussion on the update (https://github.com/AnomalyInnovations/serverless-stack-com/issues/123).

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.0...v1.1)
- Client (https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v1.0...v1.1)

v1.0: IAM as authorizer (https://59caae01424ef20727c342ce--serverless-stack.netlify.com/)

July 19, 2017: Switching to using IAM as an authorizer instead of the authenticating directly with User Pool. This was a major update to the tutorial. Discussion on the update (https://github.com/AnomalyInnovations/serverless-stack-com/issues/108).

- Tutorial changes (https://github.com/AnomalyInnovations/serverless-stack-com/compare/v0.9...v1.0)
- API (https://github.com/AnomalyInnovations/serverless-stack-demo-

api/compare/v0.9...v1.0)
- Client (https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v0.9...v1.0)

v0.9: Cognito User Pool as authorizer (https://59caadbd424ef20abdc342b4--serverless-stack.netlify.com/)

- API (https://github.com/AnomalyInnovations/serverless-stack-demo-api/releases/tag/v0.9)
- Client (https://github.com/AnomalyInnovations/serverless-stack-demo-client/releases/tag/v0.9)

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/comments-for-changelog/17)

Staying up to date

We made this guide open source to make sure that the content is kept up to date and accurate with the help of the community. We are also adding new chapters based on the needs of the community and the feedback we receive.

To help people stay up to date with the changes, we run the Serverless Stack newsletter (https://emailoctopus.com/lists/1c11b9a8-1500-11e8-a3c9-06b79b628af2/forms/subscribe). The newsletter is a:

- Short plain text email
- Outlines the recent updates to Serverless Stack
- Never sent out more than once a week
- One click unsubscribe
- And you get the entire guide as a 400 page PDF

You can also follow us on Twitter (https://twitter.com/anomaly_inv).

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/staying-up-to-date/28)

Best Practices for Building Serverless Apps

In this section of the guide we'll be covering the best practices for developing and maintaining large Serverless applications. It builds on what we've covered so far and it extends the demo notes app (https://demo2.serverless-stack.com) that we built in the first section. It's intended for teams as opposed to individual developers. It's meant to give you a foundation that scales as your app (and team) grows.

Background

Serverless Stack was launched back in March 2017. Since then thousands of folks have used the guide to build their first full-stack Serverless app. Many of you have used this as a starting point to build really large applications. Applications that are made up of scores of services worked on by a team of developers.

However, the challenges that teams face while developing large scale Serverless applications are very different from the one an individual faces while building his first app. You've to deal with architectural design decisions and questions that can be hard to answer if you haven't built and managed a large scale Serverless app before. Questions like:

- How should my project be structured when I have dozens of interdependent services?

- How should I manage my environments?
- What is the best practice for storing secrets?
- How do I make sure my production environments are completely secure?
- What does the workflow look like for the developers on my team?
- How do I debug large Serverless applications?

Some of these are not exclusive to folks working on large scale apps, but they are very common once your app grows to a certain size. We hear most of these through our readers, our users, and our Serverless Toronto Meetup (http://serverlesstoronto.org) members.

While there are tons of blog posts out there that answer some of these questions, it requires you to piece them together to figure out what the best practices are.

A new perspective

Now nearly 3 years into working on Serverless Stack and building large scale Serverless applications, there are some common design patterns that we can confidently share with our readers. Additionally, Serverless as a technology and community has also matured to the point where there are reasonable answers for the above questions.

This new addition to the guide is designed to lay out some of the best practices and give you a solid foundation to use Serverless at your company. You can be confident that as your application and team grows, you'll be on the right track for building something that scales.

Who is this for

While the topics covered in this section can be applied to any project you

are working on, they are far better suited for larger scale ones. For example, we talk about using multiple AWS accounts to configure your environments. This works well when you have multiple developers on your team but isn't worth the overhead when you are the only one working on the project.

What is covered in this section

Here is a rough rundown of the topics covered in this section of the guide.

We are covering primarily the backend Serverless portion. The frontend flow works relatively the same way as what we covered in the first section. We also found that there is a distinct lack of best practices for building Serverless backends as opposed to React apps.

- Organizing large Serverless apps (/chapters/organizing-serverless-projects.html)
  - Sharing resources using cross-stack references ({ link _chapters/cross-stack-references-in-serverless.md %})
  - Sharing code between services (/chapters/share-code-between-services.html)
  - Sharing API endpoints across services (/chapters/share-an-api-endpoint-between-services.html)
- Configuring environments (/chapters/environments-in-serverless-apps.html)
  - Using separate AWS accounts to manage environments (/chapters/structure-environments-across-aws-accounts.html)
  - Parameterizing resource names (/chapters/parameterize-serverless-resources-names.html)
  - Managing environment specific configs (/chapters/manage-environment-related-config.html)

    - Best practices for handling secrets (/chapters/storing-secrets-in-serverless-apps.html)
    - Sharing domains across environments (/chapters/share-route-53-domains-across-aws-accounts.html)
- Development lifecycle (/chapters/working-on-serverless-apps.html)
    - Working locally (/chapters/invoke-api-gateway-endpoints-locally.html)
    - Creating feature environments (/chapters/creating-feature-environments.html)
    - Creating pull request environments (/chapters/creating-pull-request-environments.html)
    - Promoting to production (/chapters/promoting-to-production.html)
    - Handling rollbacks (/chapters/rollback-changes.html)
- Using AWS X-Ray to trace Lambda functions (/chapters/tracing-serverless-apps-with-x-ray.html)

We think these concepts should be a good starting point for your projects and you should be able to adapt them to fit your use case!

How this new section is structured

This section of the guide has a fair bit of *theory* when compared to the first section. However, we try to take a similar approach. We'll slowly introduce these concepts as we work through the chapters.

The following repos will serve as the centerpiece of this section:

1. **Serverless Infrastructure** (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources)

A repo containing all the main infrastructure resources of our extended notes application. We are creating a DynamoDB table to store all the notes related info, an S3 bucket for uploading attachments, and a Cognito User Pool and Identity Pool to authenticate users.

2. **Serverless Services** (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api)

   A monorepo containing all the services in our extended notes application. We have three different services here. The `notes-api` service that powers the notes CRUD API endpoint, the `billing-api` service that processes payment information through Stripe and publishes a message on an SNS topic. Finally, we have a `notify-job` service that listens to the SNS topic and sends us a text message when somebody makes a purchase.

We'll start by forking these repos but unlike the first section we won't be directly working on the code. Instead as we work through the sections we'll point out the key aspects of the codebase.

We'll then go over step by step how to configure the environments through AWS. We'll use Seed (https://seed.run) to illustrate how to deploy our application to our environments. Note that, you do not need Seed to configure your own setup. We'll only be using it as an example. Once you complete the guide you should be able to use your favorite CI/CD service to build a pipeline that follows the best practices. Finally, we'll go over the development workflow for you and your team.

The end result of this will be that you'll have a fully functioning Serverless backend, hosted in your own GitHub repo, and deployed to your AWS environments. We want to make sure that you'll have a working setup in

place, so you can always refer back to it when you need to!

Let's get started.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/best-practices-for-building-serverless-apps/1315)

Organizing Serverless Projects

Once your serverless projects start to grow, you are faced with some choices on how to organize your growing projects. In this chapter we'll examine some of the most common ways to structure your projects at a services and application (multiple services) level.

First let's start by quickly looking at the common terms used when talking about Serverless Framework projects.

- **Service**

  A service is what you might call a Serverless project. It has a single `serverless.yml` file driving it.

- **Application**

  An application or app is a collection of multiple services.

Now let's look at the most common pattern for organizing serverless projects with our example repos.

An example

Our extended notes app has two API services, each has their own well defined business logic:

- **notes-api** service: Handles managing the notes.
- **billing-api** service: Handles making a purchase.

And your app also has a job service:

- **notify-job** service: Sends you a text message after a user successfully makes a purchase.

The infrastructure is created by the following services:

- **auth** service: Defines a Cognito User and Identity pool used to store user data.
- **database** service: Defines a DynamoDB table called `notes` used to store notes data.
- **uploads** service: Defines an S3 bucket used to store note images.

Microservices + Monorepo

Monorepo, as the term suggests is the idea of a single repository. This means that your entire application and all its services are in a single repository.

The microservice pattern on the other hand is a concept of keeping each of your services modular and lightweight. So for example; if your app allows users to create notes and make purchase; you could have a service that deals with notes and one that deals with buying.

The directory structure of your entire application under the microservice + monorepo pattern would look something like this.

```
|- services/
|--- auth/
|--- billing-api/
```

```
|--- database/
|--- notes-api/
|--- notify-job/
|--- uploads/
|- libs/
|- package.json
```

A couple of things to notice here:

1. We are going over a Node.js project here but this pattern applies to other languages as well.
2. The `services/` dir at the root is made up of a collection of services. Where a service contains a single `serverless.yml` file.
3. Each service deals with a relatively small and self-contained function. So for example, the `notes-api` service deals with everything from creating to deleting notes. Of course, the degree to which you want to separate your application is entirely up to you.
4. The `package.json` (and the `node_modules/` dir) are at the root of the repo. However, it is fairly common to have a separate `package.json` inside each service directory.
5. The `libs/` dir is just to illustrate that any common code that might be used across all services can be placed in here.
6. To deploy this application you are going to need to run `serverless deploy` separately in each of the services.
7. Environments (or stages) (/chapters/stages-in-serverless-framework.html) need to be co-ordinated across all the different services. So if your team is using a `dev`, `staging`, and `prod` environment, then you are going to need to define the specifics of this in each of the services.

Advantages of Monorepo

The microservice + monorepo pattern has grown in popularity for a couple of reasons:

1. Lambda functions are a natural fit for a microservice based architecture. This is due to a few of reasons. Firstly, the performance of Lambda functions is related to the size of the function. Secondly, debugging a Lambda function that deals with a specific event is much easier. Finally, it is just easier to conceptually relate a Lambda function with a single event.

2. The easiest way to share code between services is by having them all together in a single repository. Even though your services end up dealing with separate portions of your app, they still might need to share some code between them. Say for example; you have some code that formats your requests and responses in your Lambda functions. This would ideally be used across the board and it would not make sense to replicate this code in all the services.

Disadvantages of Monorepo

Before we go through alternative patterns, let's quickly look at the drawbacks of the microservice + monorepo pattern.

1. Microservices can grow out of control and each added service increases the complexity of your application.
2. This also means that you can end up with hundreds of Lambda functions.
3. Managing deployments for all these services and functions can get complicated.

Most of the issues described above start to appear when your application begins to grow. However, there are services that help you deal with some these issues. Services like IOpipe (https://www.iopipe.com), Epsagon (https://epsagon.com), and Dashbird (https://dashbird.io) help you with observability of your Lambda functions. And our own Seed (https://seed.run) helps you with managing deployments and environments of monorepo Serverless Framework applications.

Now let's look at some alternative approaches.

Multi-Repo

The obvious counterpart to the monorepo pattern is the multi-repo approach. In this pattern each of your repositories has a single Serverless Framework project.

A couple of things to watch out for with the multi-repo pattern.

1. Code sharing across repos can be tricky since your application is spread across multiple repos. There are a couple of ways to deal with this. In the case of Node you can use private NPM modules. Or you can find ways to link the common shared library of code to each of the repos. In both of these cases your deployment process needs to accommodate for the shared code.

2. Due to the friction involved in code sharing, we typically see each service (or repo) grow in the number of Lambda functions. This can cause you to hit the CloudFormation resource limit and get a deployment error that looks like:

```
Error ---------------------------------------------
----
```

```
The CloudFormation template is invalid: Template
format error: Number of resources, 201, is greater
than maximum allowed, 200
```

Even with the disadvantages the multi-repo pattern does have its place. We have come across cases where some infrastructure related pieces (setting up DynamoDB, Cognito, etc) is done in a service that is placed in a separate repo. And since this typically doesn't need a lot of code or even share anything with the rest of your application, it can live on its own. So in effect you can run a multi-repo setup where the standalone repos are for your *infrastructure* and your *API endpoints* live in a microservice + monorepo setup.

Finally, it's worth looking at the less common monolith pattern.

Monolith

The monolith pattern involves taking advantage of API Gateway's `{proxy+}` and `ANY` method to route all the requests to a single Lambda function. In this Lambda function you can potentially run an application server like Express (https://expressjs.com). So as an example, all the API requests below would be handled by the same Lambda function.

```
GET https://api.example.com/notes
GET https://api.example.com/notes/{id}
POST https://api.example.com/notes
PUT https://api.example.com/notes/{id}
DELETE https://api.example.com/notes/{id}

POST https://api.example.com/billing
```

And the specific section in your `serverless.yml` might look like the following:

```
handler: app.main
events:
  - http:
      method: any
      path: /{proxy+}
```

Where the `main` function in your `app.js` is responsible for parsing the routes and figuring out the HTTP methods to do the specific action necessary.

The biggest drawback here is that the size of your functions keeps growing. And this can affect the performance of your functions. It also makes it harder to debug your Lambda functions.

A practical approach

It's not the goal of this section to evaluate which setup is better. Instead, I want to layout what we think is a good setup and one that has worked out for most teams we work with. We are taking a middle ground approach and creating two repositories:

1. **serverless-stack-demo-ext-resources** (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources)
2. **serverless-stack-demo-ext-api** (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api)

In **serverless-stack-demo-ext-resources**, you have:

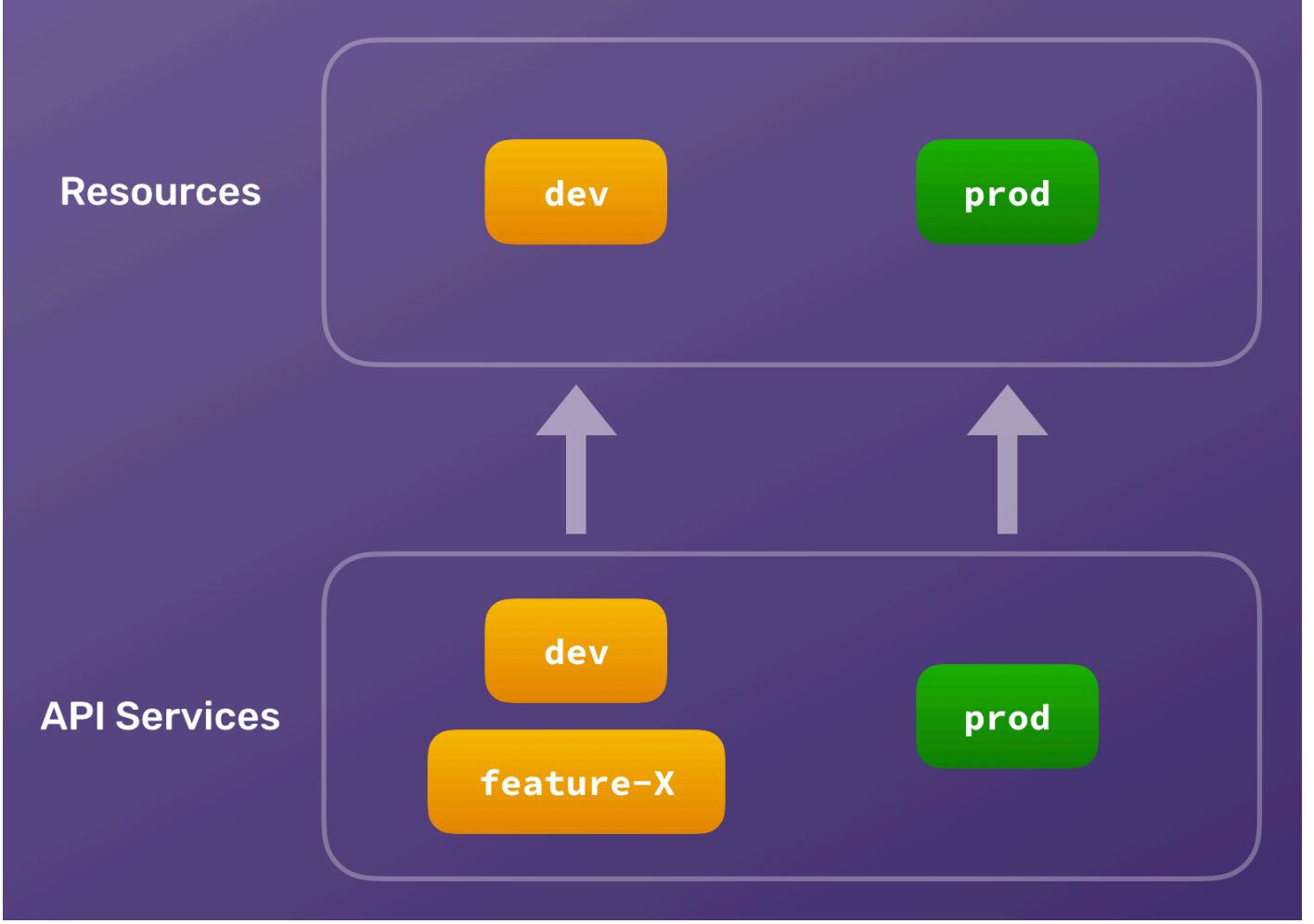
```
/
  services/
    auth/
    database/
    uploads/
```

And in **serverless-stack-demo-ext-api**, you have:

```
/
  libs/
  services/
    notes-api/
    billing-api/
    notify-job/
```

Why? Most of the code changes are going to happen in the **serverless-stack-demo-ext-api** repo. When your team is making rapid changes, you are likely to have many feature branches, bug fixes, and pull requests. A bonus with serverless is that you can spin up new environments at zero cost (you only pay for usage, not for provisioning resources). For example, a team can have dozens of ephemeral stages such as: prod, staging, dev, feature-x, feature-y, feature-z, bugfix-x, bugfix-y, pr-128, pr-132, etc. This ensures each change is tested on real infrastructure before being promoted to production.

On the other hand, changes are going to happen less frequently in the **serverless-stack-demo-ext-resources** repo. And most likely you don't need a complete set of standalone DynamoDB tables for each feature branch. In fact, a team can have three stages such as: prod, staging, and dev. And the feature/bugfix/pr stages of the **serverless-stack-demo-ext-api** can all connect to the dev stage of the **serverless-stack-demo-ext-resources**.



So if you have a service that doesn't make sense to replicate in an ephemeral environment, we would suggest moving it to the repo with all the infrastructure services. This is what we have seen most teams do. And this setup scales well as your project and team grows.

Now that we have figured out how to organize our application into repos, let's look at how we split our app into the various services. We'll start with creating a separate service for our DynamoDB tables.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/organizing-serverless-

```
projects/350)
```

Cross-Stack References in Serverless

In the previous chapter we looked at the most common patterns for organizing your Serverless applications (/chapters/organizing-serverless-projects.html). Now let's look at how to work with multiple services in your Serverless application.

You might recall that a Serverless service is where a single `serverless.yml` is used to define the project. And the `serverless.yml` file is converted into a CloudFormation template (https://aws.amazon.com/cloudformation/aws-cloudformation-templates/) using Serverless Framework. This means that in the case of multiple services you might need to reference a resource that is available in a different service. For example, you might have your DynamoDB tables created in one service and your APIs (which are in another service) need to refer to them. Of course you don't want to hard code this. And so over the next few chapters we will be breaking down the note taking application (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources) into multiple resources to illustrate how to do this.

However before we do, we need to cover the concept of cross-stack references. A cross-stack reference is a way for one CloudFormation template to refer to the resource in another CloudFormation template.

CloudFormation Cross-Stack References

To create a cross-stack reference, you need to:

1. Use the `Export:` flag in the `Outputs:` section in the `serverless.yml` of the service you would like to reference.

2. Then in the service where you want to use the reference; use the `Fn::ImportValue` CloudFormation function.

So as a quick example (we will go over this in detail shortly), say you wanted to refer to the DynamoDB table across services.

1. First export the table name in your DynamoDB service using the `Export:` flag:

```
resources:
  Resources:
    NotesTable:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: notes

# ...

 Outputs:
   Value:
     Ref: NotesTable
   Export:
     Name: NotesTableName
```

2. And in your API service, import it using the `Fn::ImportValue` function:

```
'Fn::ImportValue': NotesTableName
```

The `Fn::ImportValue` function takes the export name and returns the exported value. In this case the imported value is the DynamoDB table name.

Now before we dig into the details of cross-stack references in Serverless, let's quickly look at some of its details.

- Cross-stack references only apply within a single region. Meaning that an exported value can be referenced by any service in that region.
- Consequently, the `Export:` function needs to be unique within that region.
- If a service's export is being referenced in another stack, the service cannot be removed. So for the above example, you won't be able to remove the DynamoDB service if it is still being referenced in the API service.
- The services need to be deployed in a specific order. The service that is exporting a value needs to be deployed before the one doing the importing. Using the above example again, the DynamoDB service needs to be deployed before the API service.

Advantages of Cross-Stack References

As your application grows, it can become hard to track the dependencies between the services in the application. And cross-stack references can help with that. It creates a strong link between the services. As a comparison, if you were to refer to the linked resource by hard coding the value, it'll be difficult to keep track of it as your application grows.

The other advantage is that you can easily recreate the entire application (say for testing) with ease. This is because none of the services of your application are statically linked to each other.

In the next chapter let's look at setting up DynamoDB as a separate service.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/cross-stack-references-in-serverless/405)

DynamoDB as a Serverless Service

While creating a Serverless application with multiple services, you might want to split the DynamoDB portion out separately. This can be useful because you are probably not going to be making changes to this very frequently. Also, if you have multiple development environments, it is not likely that you are going to connect them to different database environments. For example, you might give the developers on your team their own environment but they might all connect to the same DynamoDB environment. So it would make sense to configure DynamoDB separately from the application API services.

In the example repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources), you'll notice that we have a `database` service in the `services/` directory. And the `serverless.yml` in this service helps us manage our DynamoDB table.

```
service: notes-app-ext-database

custom:
  # Our stage is based on what is passed in when
running serverless
  # commands. Or fallsback to what we have set in the
provider section.
  stage: ${opt:stage, self:provider.stage}
```

```yaml
  # Set the table name here so we can use it while
testing locally
  tableName: ${self:custom.stage}-ext-notes

provider:
  name: aws
  stage: dev
  region: us-east-1

resources:
  Resources:
    NotesTable:
      Type: AWS::DynamoDB::Table
      Properties:
        # Generate a name based on the stage
        TableName: ${self:custom.tableName}
        AttributeDefinitions:
          - AttributeName: userId
            AttributeType: S
          - AttributeName: noteId
            AttributeType: S
        KeySchema:
          - AttributeName: userId
            KeyType: HASH
          - AttributeName: noteId
            KeyType: RANGE
        # Set the capacity to auto-scale
        BillingMode: PAY_PER_REQUEST

  Outputs:
    NotesTableArn:
```

```
      Value:
        Fn::GetAtt:
          - NotesTable
          - Arn
      Export:
        Name: ${self:custom.stage}-ExtNotesTableArn
```

If you have followed along with the first part of our guide (/chapters/configure-dynamodb-in-serverless.html), the `Resources:` section should seem familiar. It is creating the Notes table that we use in our note taking application (https://github.com/AnomalyInnovations/serverless-stack-demo-api). The key addition here in regards to the cross-stack references is in the `Outputs:` section. Let's go over them quickly.

1. We are exporting one value here. The `NotesTableArn` is the ARN (/chapters/what-is-an-arn.html) of the DynamoDB table that we are creating. The ARN is necessary for any IAM roles that are going to reference the DynamoDB table.

2. The export name is based on the stage we are using to deploy this service - `${self:custom.stage}`. This is important because we want our entire application to be easily replicable across multiple stages. If we don't include the stage name the exports will thrash when we deploy to multiple stages.

3. The names of the exported values is `${self:custom.stage}-NotesTableArn`.

4. We get the table ARN by using the `Fn::GetAtt` CloudFormation function. This function takes a reference from the current service and

the attribute we need. The reference in this case is `NotesTable`. You'll notice that the table we created in the `Resources:` section is created using `NotesTable` as the name.

When we deploy this service we'll notice the exported values in the output and we can reference these cross-stack in our other services.

Next we'll do something similar for our S3 bucket.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/dynamodb-as-a-serverless-service/406)

S3 as a Serverless Service

Just as we did with DynamoDB in the last chapter (/chapters/dynamodb-as-a-serverless-service.html), we'll look at splitting S3 into a separate Serverless service. It should be noted that for our simple note taking application, it doesn't make too much sense to split S3 into its own service. But it is useful to go over this because when your app grows, you'll be looking to split your services up. Also it's good to understand cross-stack references in Serverless.

In the example repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources), you'll notice that we have a `uploads` service in the `services/` directory. And the `serverless.yml` in this service looks like the following.

```
service: notes-app-ext-uploads

custom:
  # Our stage is based on what is passed in when
running serverless
  # commands. Or fallsback to what we have set in the
provider section.
  stage: ${opt:stage, self:provider.stage}

provider:
  name: aws
  stage: dev
```

```
  region: us-east-1

resources:
  Resources:
    S3Bucket:
      Type: AWS::S3::Bucket
      Properties:
        # Set the CORS policy
        CorsConfiguration:
          CorsRules:
            -
              AllowedOrigins:
                - '*'
              AllowedHeaders:
                - '*'
              AllowedMethods:
                - GET
                - PUT
                - POST
                - DELETE
                - HEAD
              MaxAge: 3000

  # Print out the name of the bucket that is created
  Outputs:
    AttachmentsBucketArn:
      Value:
         Fn::GetAtt:
          - S3Bucket
          - Arn
      Export:
```

```
        Name: ${self:custom.stage}-
ExtAttachmentsBucketArn

    AttachmentsBucketName:
      Value:
        Ref: S3Bucket
      Export:
        Name: ${self:custom.stage}-ExtAttachmentsBucket
```

Most of the `Resources:` section should be fairly straightforward and is based on first part of this guide (/chapters/configure-s3-in-serverless.html). So let's go over the cross-stack exports in the `Outputs:` section.

1. Just as in the DynamoDB service (/chapters/dynamodb-as-a-serverless-service.html), we are exporting the ARN (/chapters/what-is-an-arn.html) (`AttachmentsBucketArn`). For this service, we are also exporting the name of the bucket (`AttachmentsBucketName`).

2. The names of the exported values are based on the stage: `${self:custom.stage}-ExtAttachmentsBucketArn` and `${self:custom.stage}-ExtAttachmentsBucket` .

3. We can get the ARN by using the `Fn::GetAtt` function by passing in the ref (`S3Bucket`) and the attribute we need (`Arn`).

4. And finally, we can get the bucket name by just using the ref (`S3Bucket`). Note that unlike the DynamoDB table name, the S3 bucket name is auto-generated. So while we could get away with not exporting the DynamoDB table name; in the case of S3, we need to export it.

Next, let's look at how to setup our user authentication services, Cognito User Pool and Identity Pool.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/s3-as-a-serverless-service/407)

Cognito as a Serverless Service

Now let's look at splittiing Cognito User Pool and Cognito Federated Identities into a separate Serverless service. In this chapter we are going to create a Serverless service that will use cross-stack references to tie all of our resources together.

In the example repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources), open the `auth` service in the `services/` directory.

```
service: notes-app-ext-auth

custom:
  # Our stage is based on what is passed in when
running serverless
  # commands. Or fallsback to what we have set in the
provider section.
  stage: ${opt:stage, self:provider.stage}

provider:
  name: aws
  stage: dev
  region: us-east-1
```

```
resources:
  Resources:
    CognitoUserPool:
      Type: AWS::Cognito::UserPool
      Properties:
        # Generate a name based on the stage
        UserPoolName: ${self:custom.stage}-ext-user-
pool
        # Set email as an alias
        UsernameAttributes:
          - email
        AutoVerifiedAttributes:
          - email

    CognitoUserPoolClient:
      Type: AWS::Cognito::UserPoolClient
      Properties:
        # Generate an app client name based on the
stage
        ClientName: ${self:custom.stage}-ext-user-pool-
client
        UserPoolId:
          Ref: CognitoUserPool
        ExplicitAuthFlows:
          - ADMIN_NO_SRP_AUTH
        GenerateSecret: false

    # The federated identity for our user pool to auth
with
    CognitoIdentityPool:
      Type: AWS::Cognito::IdentityPool
```

```
      Properties:
        # Generate a name based on the stage
        IdentityPoolName:
${self:custom.stage}ExtIdentityPool
        # Don't allow unathenticated users
        AllowUnauthenticatedIdentities: false
        # Link to our User Pool
        CognitoIdentityProviders:
          - ClientId:
              Ref: CognitoUserPoolClient
            ProviderName:
              Fn::GetAtt: [ "CognitoUserPool",
"ProviderName" ]

    # IAM roles
    CognitoIdentityPoolRoles:
      Type: AWS::Cognito::IdentityPoolRoleAttachment
      Properties:
        IdentityPoolId:
          Ref: CognitoIdentityPool
        Roles:
          authenticated:
            Fn::GetAtt: [CognitoAuthRole, Arn]

    # IAM role used for authenticated users
    CognitoAuthRole:
      Type: AWS::IAM::Role
      Properties:
        Path: /
        AssumeRolePolicyDocument:
          Version: '2012-10-17'
```

```
          Statement:
            - Effect: 'Allow'
              Principal:
                Federated: 'cognito-
identity.amazonaws.com'
              Action:
                - 'sts:AssumeRoleWithWebIdentity'
              Condition:
                StringEquals:
                  'cognito-identity.amazonaws.com:aud':
                    Ref: CognitoIdentityPool
                'ForAnyValue:StringLike':
                  'cognito-identity.amazonaws.com:amr':
authenticated
        Policies:
          - PolicyName: 'CognitoAuthorizedPolicy'
            PolicyDocument:
              Version: '2012-10-17'
              Statement:
                - Effect: 'Allow'
                  Action:
                    - 'mobileanalytics:PutEvents'
                    - 'cognito-sync:*'
                    - 'cognito-identity:*'
                  Resource: '*'
                # Allow users to upload attachments to
their
                # folder inside our S3 bucket
                - Effect: 'Allow'
                  Action:
                    - 's3:*'
```

```
                  Resource:
                    - Fn::Join:
                      - ''
                      -
                        - 'Fn::ImportValue':
${self:custom.stage}-ExtAttachmentsBucketArn
                        - '/private/'
                        - '$'
                        - '{cognito-
identity.amazonaws.com:sub}/*'

  # Print out the Id of the User Pool and Identity Pool
that are created
  Outputs:
    UserPoolId:
      Value:
        Ref: CognitoUserPool

    UserPoolClientId:
      Value:
        Ref: CognitoUserPoolClient

    IdentityPoolId:
      Value:
        Ref: CognitoIdentityPool

    CognitoAuthRole:
      Value:
        Ref: CognitoAuthRole
      Export:
        Name: ExtCognitoAuthRole-${self:custom.stage}
```

This can seem like a lot but both the `CognitoUserPool:` and the `CognitoUserPoolClient:` section are simply creating our Cognito User Pool. And you'll notice that both these sections are not using any cross-stack references. They are effectively standalone. If you are looking for more details on this, refer to the earlier part of this guide (/chapters/configure-cognito-user-pool-in-serverless.html).

Cognito Identity Pool

The Cognito Identity Pool on the other hand needs to reference all the resources created thus far. It can be a little intimidating to start but let's break it down into its various parts:

- `CognitoIdentityPool:` creates the role and states that the Cognito User Pool that we created above is going to be our auth provider.
- The Identity Pool has an IAM role attached to its authenticated and unauthenticated users. Since, we only allow authenticated users to our note taking app; we only have one role. The `CognitoIdentityPoolRoles:` section states that we have an authenticated user role that we are going to create below and we are referencing it here by doing `Fn::GetAtt: [CognitoAuthRole, Arn]`.
- Finally, the `CognitoAuthRole:` section creates the IAM role that will allow access to our S3 file uploads bucket.

Let's look at the Cognito auth IAM role in detail.

Cognito Auth IAM Role

The IAM role that our authenticated users are going to use needs to allow access to our S3 file uploads bucket.

This is the relevant section from the above `serverless.yml`.

```
# Allow users to upload attachments to their
# folder inside our S3 bucket
- Effect: 'Allow'
  Action:
    - 's3:*'
  Resource:
    - Fn::Join:
      - ''
      -
        - 'Fn::ImportValue': ${self:custom.stage}-
ExtAttachmentsBucketArn
        - '/private/'
        - '$'
        - '{cognito-identity.amazonaws.com:sub}/*'
```

The S3 bucket resource in our IAM role looks something like:

```
"arn:aws:s3:::my_s3_bucket/private/${cognito-
identity.amazonaws.com:sub}/*"
```

Where `my_s3_bucket` is the name of the bucket. We are going to use the generated name that we exported back in the S3 bucket chapter (/chapters/s3-as-a-serverless-service.html). And we can import it using:

```
'Fn::ImportValue': ${self:custom.stage}-
ExtAttachmentsBucketArn
```

Again, all of our references are based on the stage we are deploying to. You'll notice that we don't have the IAM role here that allows access to our APIs. We are going to be doing that along side our API services.

And finally, you'll notice that we are outputting:

- A couple of things in this service. We need the Ids of the Cognito resources created in our frontend. But we don't have to export any cross-stack values.
- The name of the CognitoAuthRole IAM role. We need it in our api service to grant permissions for the role to invoke the API Gateway endpoint that will be deployed.

Now that all of our resources are complete, let's look at the API services.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/cognito-as-a-serverless-service/409)

Share Code Between Services

In the previous few chapters, we looked at how to organize all our infrastructure resources in one repo. In these next couple of chapters we'll look at how to organize all our business logic services (APIs) in the same repo. We'll start by attempting to answer the following questions:

1. Do I have just one or multiple `package.json` files?
2. How do I share common code and config between services?
3. How do I share common config between the various `serverless.yml`?

We are using an extended version of the notes app for this section. You can find the **sample repo here** (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api). Let's take a quick look at how the repo is organized.

```
/
  package.json
  config.js
  serverless.common.yml
  libs/
  services/
    notes-api/
      package.json
      serverless.yml
      handler.js
    billing-api/
```

```
      package.json
      serverless.yml
      handler.js
    notify-job/
      serverless.yml
      handler.js
```

1. Structuring the package.json

The first question you typically have is about the `package.json`. Do I just have one `package.json` or do I have one for each service? We recommend having multiple `package.json` files.

We use the `package.json` at the project root to install the dependencies that will be shared across all the services. For example, the serverless-bundle (https://github.com/AnomalyInnovations/serverless-bundle) plugin that we are using to optimally package our Lambda functions is installed at the root level. It doesn't make sense to install it in each and every service.

On the other hand, dependencies that are specific to a single service are installed in the `package.json` for that service. In our example, the `billing-api` service uses the `stripe` NPM package. So it's added just to that `package.json`. Similarly, the `notes-api` service uses the `uuid` NPM package, and it's added just to that `package.json`.

This setup implies that when you are deploying your app through a CI; you'll need to do an `npm install` twice. Once in the root level and once in a specific service. Seed (https://seed.run/) does this automatically for you.

Usually, you might have to manually pick and choose the modules that need to be packaged with your Lambda function. Simply packaging all the dependencies will increase the code size of your Lambda function and this leads to longer cold start times. However, in our example we are using the `serverless-bundle` plugin that internally uses Webpack (https://webpack.js.org/)'s tree shaking algorithm to only package the code that our Lambda function needs.

2. Sharing common code and config

The biggest reason you are using a monorepo setup is because your services need to share some common code, and this is the most convenient way to do so.

Alternatively, you could use a multi-repo approach where all your common code is published as private NPM packages. However, this adds an extra layer of complexity and it doesn't make sense if you are a small team just wanting to share some common code.

In our example, we want to share some common code. We'll be placing these in a `libs/` directory. Our services need to make calls to various AWS services using the AWS SDK. And we have the common SDK configuration code in the `libs/aws-sdk.js` file.

```
import aws from "aws-sdk";
import xray from "aws-xray-sdk";

// Do not enable tracing for 'invoke local'
const awsWrapped = process.env.IS_LOCAL ? aws :
xray.captureAWS(aws);
```

```
export default awsWrapped;
```

Our Lambda functions will now import this instead of the standard AWS SDK.

```
import AWS from '../../libs/aws-sdk';
```

The great thing about this is that we can easily change any AWS related config and it'll apply across all of our services. In this case, we are using AWS X-Ray (https://aws.amazon.com/xray/) to enabled tracing across our entire application. You don't need to do this but we are going to be talking about this in one of the later chapters. And this is a good example of how to share the same AWS config across all our services.

3. Share common serverless.yml config

We have separate `serverless.yml` configs for our services. However, we end up needing to share some config across all of our `serverless.yml` files. To do that:

1. Place the shared config values in a common yaml file at the root level.
2. And reference them in your individual `serverless.yml` files.

For example, we want to define the current stage and the resources stage we want to connect to across all of our services. Also, to be able to use X-Ray, we need to grant the necessary X-Ray permissions in the Lambda IAM role. So we added a `serverless.common.yml` at the repo root.

```
custom:
  # Our stage is based on what is passed in when
running serverless
```

```
  # commands. Or fallsback to what we have set in the
provider section.
  stage: ${opt:stage, self:provider.stage}
  resourcesStages:
    prod: prod
    dev: dev
  resourcesStage:
${self:custom.resourcesStages.${self:custom.stage},
self:custom.resourcesStages.dev}

lambdaPolicyXRay:
  Effect: Allow
  Action:
    - xray:PutTraceSegments
    - xray:PutTelemetryRecords
  Resource: "*"
```

And in each of our service, we include the **custom** definition in their `serverless.yml`:

```
custom: ${file(../../serverless.common.yml):custom}
```

And we include the **lambdaPolicyXRay** IAM policy:

```
  iamRoleStatements:
    -
${file(../../serverless.common.yml):lambdaPolicyXRay}
```

You can do something similar for any other `serverless.yml` config that needs to be shared.

Next, let's look at what happens when multiple API services need to share the same API endpoint.

For help and discussion

💬 Comments on this chapter (https://discourse.serverless-stack.com/t/share-code-between-services/1333)

Share an API Endpoint Between Services

In this chapter we will look at how to work with API Gateway across multiple services. A challenge that you run into when splitting your APIs into multiple services is sharing the same domain for them. You might recall that APIs that are created as a part of the Serverless service get their own unique URL that looks something like:

```
https://z6pv80ao4l.execute-api.us-east-1.amazonaws.com/dev
```

When you attach a custom domain for your API, it is attached to a specific endpoint like the one above. This means that if you create multiple API services, they will all have unique endpoints.

You can assign different base paths for your custom domains. For example, `api.example.com/notes` can point to one service while `api.example.com/billing` can point to another. But if you try to split your `notes` service up, you'll face the challenge of sharing the custom domain across them.

In our notes app, we have two services with API endpoints, `notes-api` and `billing-api`. In this chapter, we are going to look at how to configure API Gateway such that both services are served out via a single API endpoint.

The API path we want to setup is:

- `notes-api` list all notes ⇒ GET `https://api.example.com/notes`
- `notes-api` get one note ⇒ GET `https://api.example.com/notes/{noteId}`
- `notes-api` create one note ⇒ POST `https://api.example.com/notes`
- `notes-api` update one note ⇒ PUT `https://api.example.com/notes/{noteId}`
- `notes-api` delete one note ⇒ DELETE `https://api.example.com/notes/{noteId}`
- `billing-api` checkout ⇒ POST `https://api.example.com/billing`

How paths work in API Gateway

API Gateway is structured in a slightly tricky way. Let's look at this in detail.

- Each path part is a separate API Gateway resource object.
- And a path part is a child resource of the preceding part.

So the part path `/notes`, is a child resource of `/`. And `/notes/{noteId}` is a child resource of `/notes`.

Based on our setup, we want the `billing-api` to have the `/billing` path. And this would be a child resource of `/`. However, `/` is created in the `notes-api` service. So we'll need to find a way to share the resource across services.

Notes Service

To do this, the `notes-api` needs to share the API Gateway project and the root path `/`.

In our serverless-stack-demo-ext-api (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api) repo, go into the `services/notes-api/` directory. In the `serverless.yml`, near the end, you will notice:

```
...

- Outputs:
    ApiGatewayRestApiId:
      Value:
        Ref: ApiGatewayRestApi
      Export:
        Name: ${self:custom.stage}-
ExtApiGatewayRestApiId

    ApiGatewayRestApiRootResourceId:
      Value:
         Fn::GetAtt:
          - ApiGatewayRestApi
          - RootResourceId
      Export:
        Name: ${self:custom.stage}-
ExtApiGatewayRestApiRootResourceId
```

Let's look at what we are doing here.

1. The first cross-stack reference that needs to be shared is the API Gateway Id that is created as a part of this service. We are going to export it with the name `${self:custom.stage}-ExtApiGatewayRestApiId`. Again, we want the exports to work across all our environments/stages and so we include the stage name as a part of it. The value of this export is available as a reference in our current stack called `ApiGatewayRestApi`.
2. We also need to export the `RootResourceId`. This is a reference to the `/` path of this API Gateway project. To retrieve this Id we use the `Fn::GetAtt` CloudFormation function and pass in the current `ApiGatewayRestApi` and look up the attribute `RootResourceId`. We export this using the name `${self:custom.stage}-ExtApiGatewayRestApiRootResourceId`.

Billing Service

Let's look at how we are importing the above. Open the `billing-api` service in the `services/` directory.

```
...

provider:
  apiGateway:
    restApiId:
      'Fn::ImportValue': ${self:custom.stage}-
ExtApiGatewayRestApiId
    restApiRootResourceId:
      'Fn::ImportValue': ${self:custom.stage}-
ExtApiGatewayRestApiRootResourceId
...
```

```
functions:
  billing:
    handler: billing.main
    events:
      - http:
          path: billing
          method: post
          cors: true
          authorizer: aws_iam
```

To share the same API Gateway domain as our `notes-api` service, we are adding an `apiGateway:` section to the `provider:` block.

1. Here we state that we want to use the `restApiId` of our notes service. We do this by using the cross-stack reference `'Fn::ImportValue': ${self:custom.stage}-ExtApiGatewayRestApiId` that we had exported above.

2. We also state that we want all the APIs in our service to be linked under the root path of our notes service. We do this by setting the `restApiRootResourceId` to the cross-stack reference `'Fn::ImportValue': ${self:custom.stage}-ExtApiGatewayRestApiRootResourceId` from above.

Now when you deploy the `billing-api` service, instead of creating a new API Gateway project, Serverless Framework is going to reuse the project you imported.

The key thing to note in this setup is that API Gateway needs to know where to attach the routes that are created in this service. We want the

`/billing` path to be attached to the root of our API Gateway project. Hence the `restApiRootResourceId` points to the root resource of our `notes-api` service. Of course we don't have to do it this way. We can organize our service such that the `/billing` path is created in our main API service and we link to it here.

Dependency

By sharing API Gateway project, we are making the `billing-api` depend on the `notes-api`. When deploying, you need to ensure the `notes-api` is deployed first.

Limitations

Note that, a path part can only be created **ONCE**. Let's look at an example to understand how this works. Say you need to add another API service that uses the following endpoint.

```
https://api.example.com/billing/xyz
```

This new service **CANNOT** import `/` from the `notes-api`.

This is because, Serverless Framework tries to create the following two path parts:

1. `/billing`
2. `/billing/xyz`

But `/billing` has already been created in the `billing-api` service. So if you were to deploy this new service, CloudFormation will fail and complain that the resource already exists.

You **HAVE TO** import `/billing` from the `billing-api`, so the new service will only need to create the `/billing/xyz` part.

Now we are done organizing our services and we are ready to deploy them. To recap, we have a couple of dependencies in our resources repo and a couple in our API repo.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/api-gateway-domains-across-services/408)

Deploy a Serverless App with Dependencies

So now that we have a couple of downstream services that are referencing a resource deployed in an upstream service; let's look at how this dependency affects the way we deploy our app.

The short version is that:

- When you introduce a new dependency in your app you cannot deploy all the services concurrently.
- However, once these services have been deployed, you can do so.

First deployment

In our resources repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources) we need to:

- Deploy the `database` and `uploads` service. These can be deployed concurrently.
- Then we can deploy the `auth` service.

Next for the API repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api) for the first time, you have to:

- Deploy the `notes-api` first. This will export the value `dev-ExtApiGatewayRestApiId` and `dev-`

`ExtApiGatewayRestApiRootResourceId`.

- Then deploy the `billing-api` next. This will export the value `ExtNotePurchasedTopicArn-dev`.
- Then deploy the `notify-job`.

Assuming that you are deploying to the `dev` stage.

If you were to deploy `billing-api` and `notify-job` concurrently, the `notify-job` will fail with the following CloudFormation error:

```
notify-job - No export named ExtNotePurchasedTopicArn-dev found.
```

This error is basically saying that the ARN referenced in its `serverless.yml` does not exist. This makes sense because we haven't created it yet!

Subsequent deployments

Once all the services have been successfully deployed, you can deploy them all concurrently. This is because the referenced ARN has already been created.

Adding new dependencies

Say you add a new SNS topic in `billing-api` service and you want the `notify-job` service to subscribe to that topic. The first deployment after the change, will again fail if all the services are deployed concurrently. You need to deploy the `billing-api` service first, and then deploy the `notify-job` service.

We are almost ready to deploy our extended notes app. But before we can

do that, let's configure our environments.

Deploying through a CI

If you are using a CI, you'll need to deploy the above in phases. With Seed (https://seed.run), we handle this using a concept of Deploy Phases (https://seed.run/docs/configuring-deploy-phases).

Managing deployment in phases for api

For our api repo, the dependencies look like:

```
notes-api > billing-api > notify-job
```

To break it down in detail:

- The `billing-api` service relies on the `notes-api` service for the API Gateway export.
- The `notify-job` service relies on the `billing-api` service for the SNS Topic export.

That covers our section on organizing your Serverless app. Next, let's configure our environments.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploying-multiple-services-in-serverless/410)

Environments in Serverless Apps

In this section, we are going to be looking at the best practices of configuring environments for your Serverless app. But before we do that, let's quickly go over some of the concepts involved.

Pay per use = multiple dev environments

Serverless apps and their associated services (Lambda, API Gateway, DynamoDB, etc.) all have a pay per use model. And it seems very likely that more AWS services are moving towards that model. Also, thanks to the infrastructure as code (/chapters/what-is-infrastructure-as-code.html) idea that Serverless uses, it's very easy to replicate environments. So creating multiple dev/staging environments is something that is highly recommended.

Long lived environments

Serverless doesn't change how you setup long lived stages. You still have the usual `dev` stage, `prod` stage. And the intermediate stages in between such as `staging`, `qa`, `preprod`, etc. The larger your team, the more intermediate stages you tend to have.

Ephemeral environments

During development, you usually have a number of development Git branches like feature branches or hotfix branches. In the traditional server world, many teams don't setup an environment for each of their branches. This is due to the infrastructure cost of setting up a new environment. And the manual work involved in doing so. In addition, once a branch is ready to be merged, a pull request is usually created. Ideally you want to deploy the temporarily merged version of code to a pull request environment. If you've used Heroku in the past, this is the idea behind their Review Apps (https://devcenter.heroku.com/articles/github-integration-review-apps).

However, since creating new environments in Serverless is both convenient and cheap, it's considered best practice to create these ephemeral stages. You want to configure your CI/CD pipeline to automatically bring these stages up, test against them, and tear them down once you are done.

Seed can automatically create stages for a feature branch on branch creation (https://seed.run/docs/working-with-branches) and for PRs on PR creation (https://seed.run/docs/working-with-pull-requests). It'll also automatically remove the stage and all the associated resources once the branch is removed or the PR is merged.

Next, let's configure our environments!

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/environments-in-serverless-apps/1323)

Structure Environments Across AWS Accounts

The typical recommendation for teams is to deploy each of their environments to a separate AWS account. We find that this ends up being excessive for most teams. In our experience, what seems to work for many teams is:

- One account for the Production environment. You want to apply very strict IAM access permissions to this account.
- One account for **EACH** Staging environment. If you have multiple staging environment ie. `preprod`, `qa`, `uat`, etc, use a separate AWS account for each. You don't want to have multiple Staging environments share the same account because each Staging environment needs to mirror Production as closely as possible.
- One account for **ALL** Development environments. All feature and hotfix environments share the same AWS account. You will have many Development environments, many will be very short-lived. Creating a temporary AWS account for each environment and tearing it down after the change is merged into master is far too excessive. Especially when you need to push a quick hotfix. Also, as we mentioned in the How to organize your services (/chapters/organizing-serverless-projects.html) chapter, you have one repo for the infrastructure and one repo for the code. Each Development environment most likely does not need their own version of the infrastructure. Consider when the scenario where you push a hotfix. You can have multiple API

environments all talk to the one Infrastructure environment. And having them all sit inside the same AWS account makes it easy for them to talk to each other without configuring cross-account IAM permissions. Of course, this is not a hard rule. If you have a major feature release, the release can have its own Infrastructure environment and the entire setup can be deployed to a separate AWS account.
- One account for **EACH** Developer environment. Each developer on your team has their own playground account.

At first glance, this might just seem like a whole lot of extra work and you might wonder if the added complexity is worth while. However, for teams this really helps protect your production environment. And we'll go over the various ways it does so.

Environment separation

Imagine that you (or somebody on your team) removes a DynamoDB table or Lambda function in your `serverless.yml` definition. Now, instead of deploying it to your *dev* environment, you accidentally deploy it to *production*. This happens more often than you think!

To avoid mishaps like these, not every developer on your team should have access or direct access from their terminal to the *production* environment. However, if all your environments are in the same AWS account, you need to carefully craft a detailed IAM policy to restrict/grant access to specific resources. This can be hard to do and you are likely to make mistakes.

By keeping each environment in a separate account, you can manage user access on a per account basis. And for *dev* environments you could potentially grant your developers *AdministratorAccess* without worrying

about the specific resources.

Resource limits

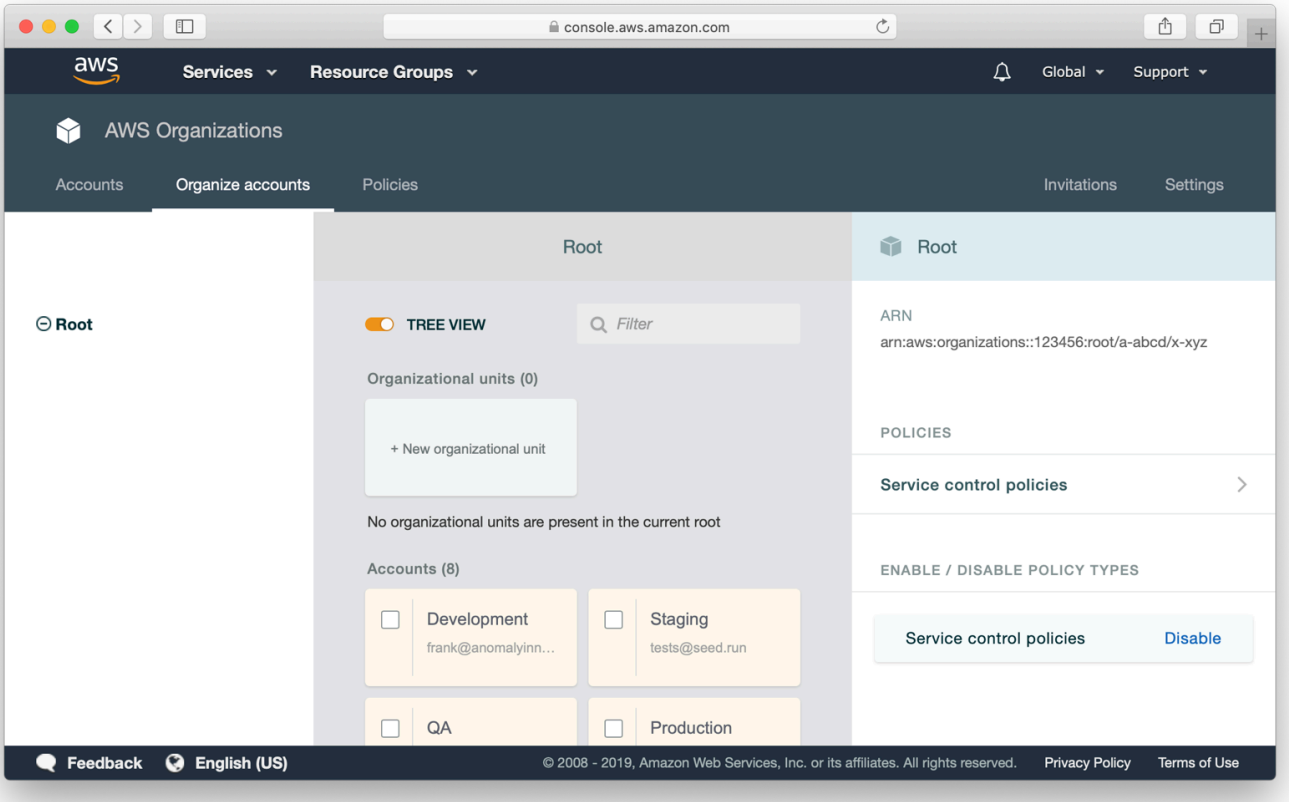
Another benefit of separating environments across AWS accounts is helping you work with AWS' service limits. You might be familiar with the various hard and soft limits of the AWS services you use. Like the Lambda's 75 GB code storage limit (https://docs.aws.amazon.com/lambda/latest/dg/limits.html) or the total number of S3 buckets per account limit (https://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html). These limits are applicable on a per account basis. Therefore, an issue with having a single AWS account for all your environments is that, hitting these limits can affect your production environment. Hence, affecting your users!

For example, you are likely to deploy to your *dev* environment an order of magnitude more often than to your *production* environment. Meaning that you'll hit the Lambda code storage limit on your *dev* environment quicker than on *production*. If you only have one account for all your environments, hitting this limit would critically affect your production builds to fail.

By using multiple AWS accounts, you can be sure that the service limits will not interfere across environments.

Consolidated billing

Finally, having separate accounts for your environments is recommended by AWS. And AWS has great support for it as well. In the AWS Organizations console, you can view and manage all the AWS accounts in your master account.



You don't have to setup the billing details for each account. Billing is consolidated to the master account. You can also view a breakdown of usage and cost for each service in each account.

In the next chapter, we are going to look at how to setup these AWS accounts using AWS Organizations.

For help and discussion

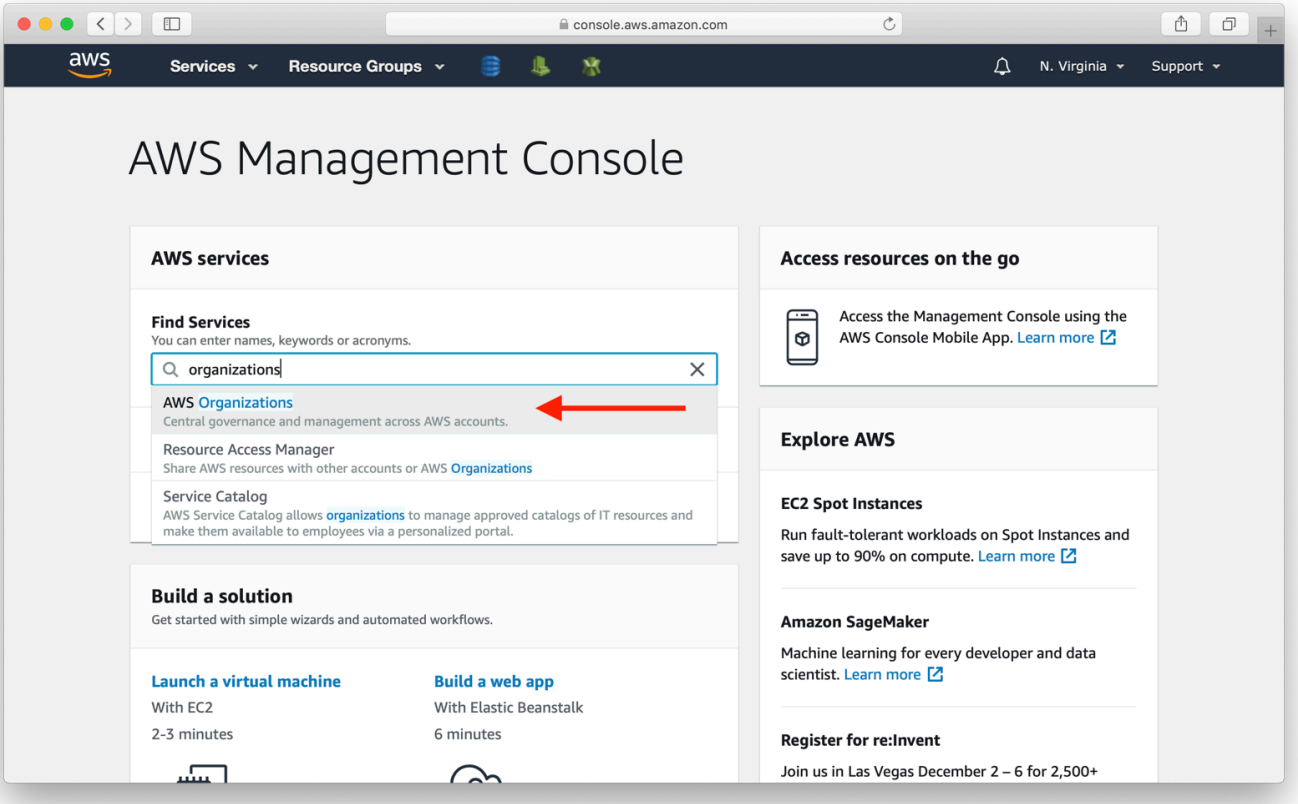
Comments on this chapter (https://discourse.serverless-stack.com/t/structure-environments-across-aws-accounts/1336)

Manage AWS Accounts Using AWS Organizations

With AWS Organizations, you can create and manage all the AWS accounts in your master account. In this chapter we'll look at how to create multiple AWS accounts for the environments in our serverless app.

Create AWS accounts

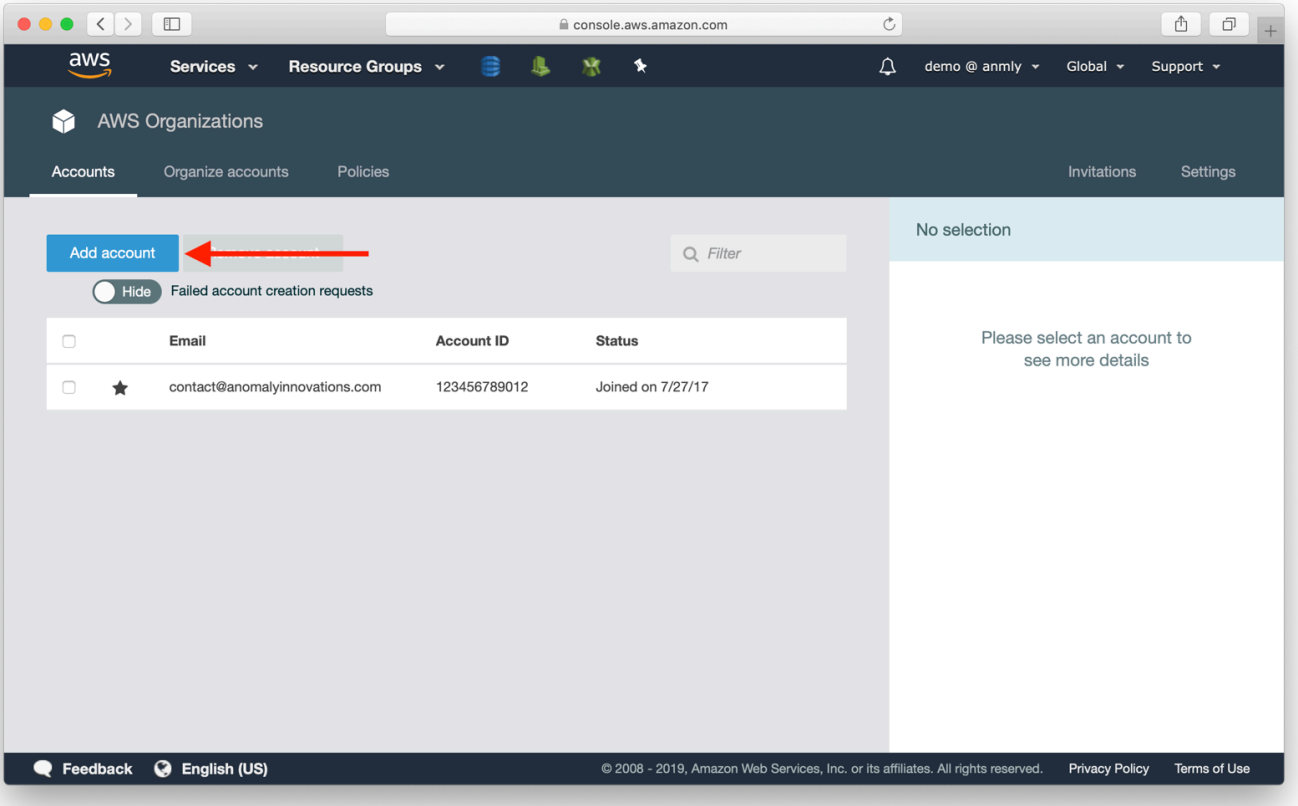
Go to the AWS Organizations console.



The account labeled with the star is your **master** AWS account. This

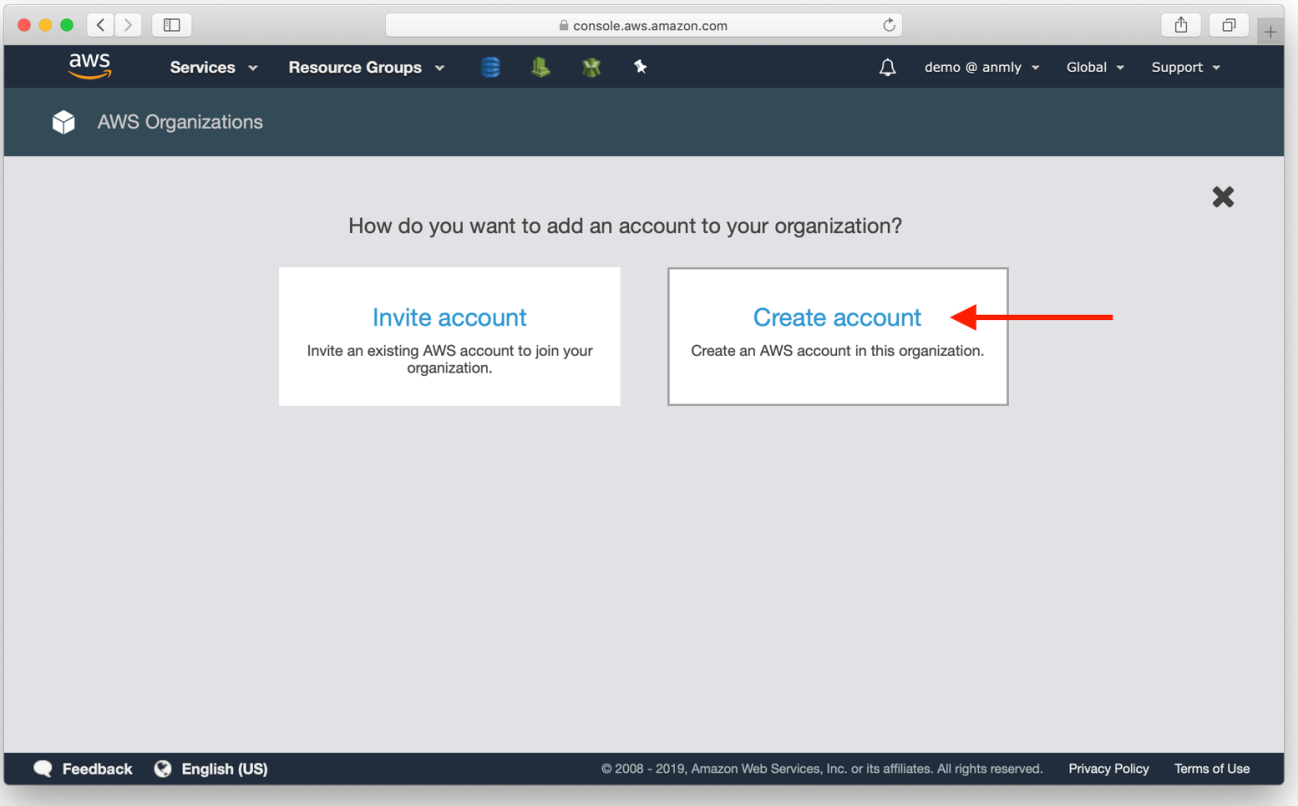
account cannot be removed from the organization.

Select **Add account**.



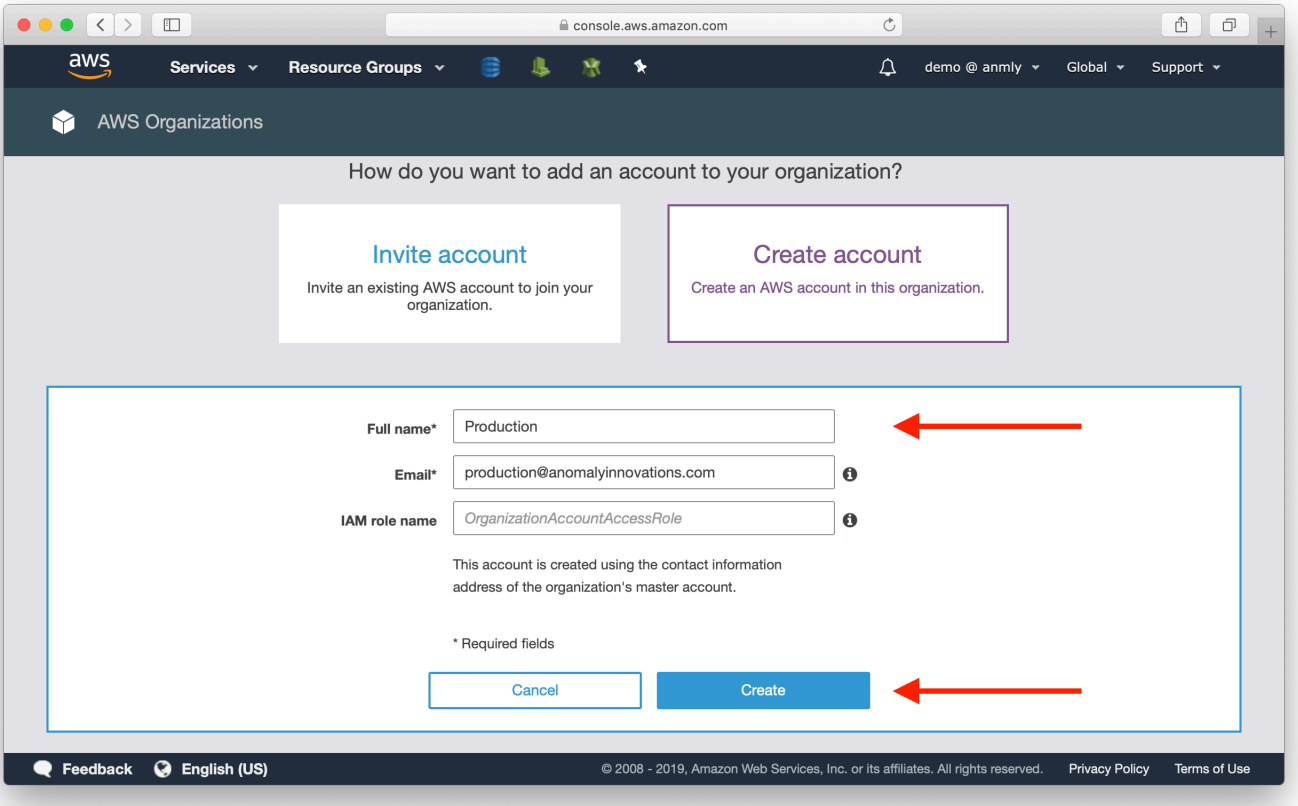
You can either create a new AWS account or if you already have multiple standalone AWS accounts, you can add them into your organization.

Select **Create account**.

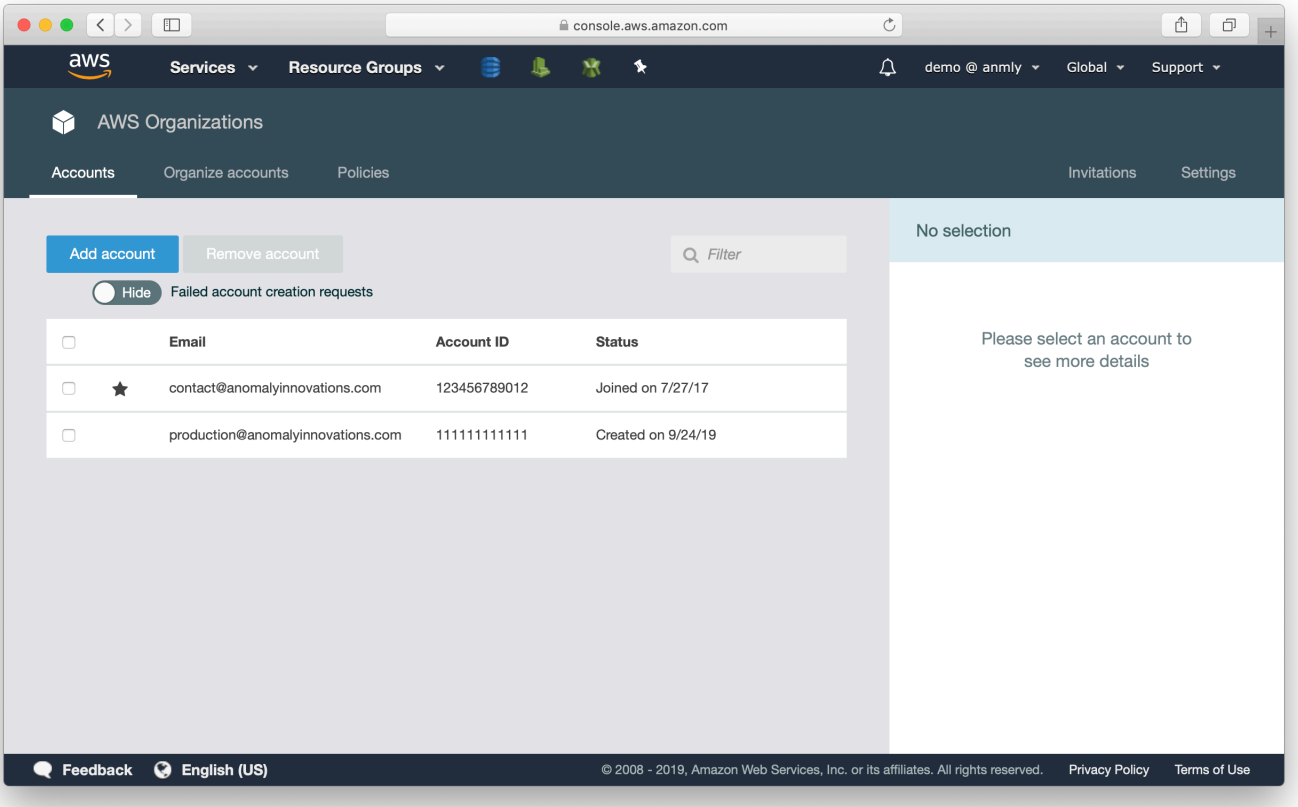


Let's create our Production account first. Fill out the following:

- **Full name**: Enter Prod, Production or what you would like to call this account. It is used for display purposes only.
- **Email**: Each account requires a unique email address. Emails with the '+' sign are allowed.
- **IAM role name**: Leave this empty. When creating a new account, AWS Organizations automatically creates an IAM role in the new account that allows the master account to be able to assume into it. Actually, it's the only way to access a newly created account. By default, the IAM role is named **OrganizationAccountAccessRole**, you can give it another name.



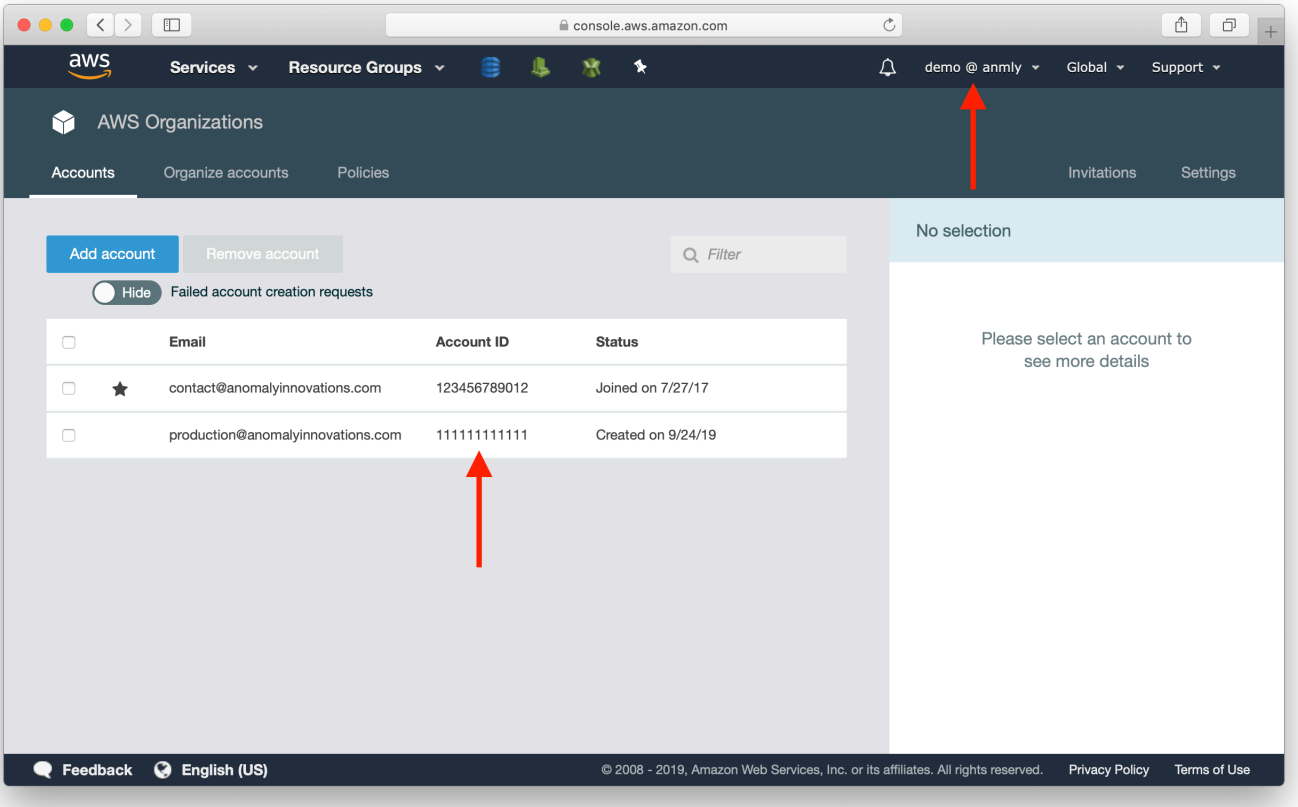
Now, you have 2 AWS accounts in your organization.



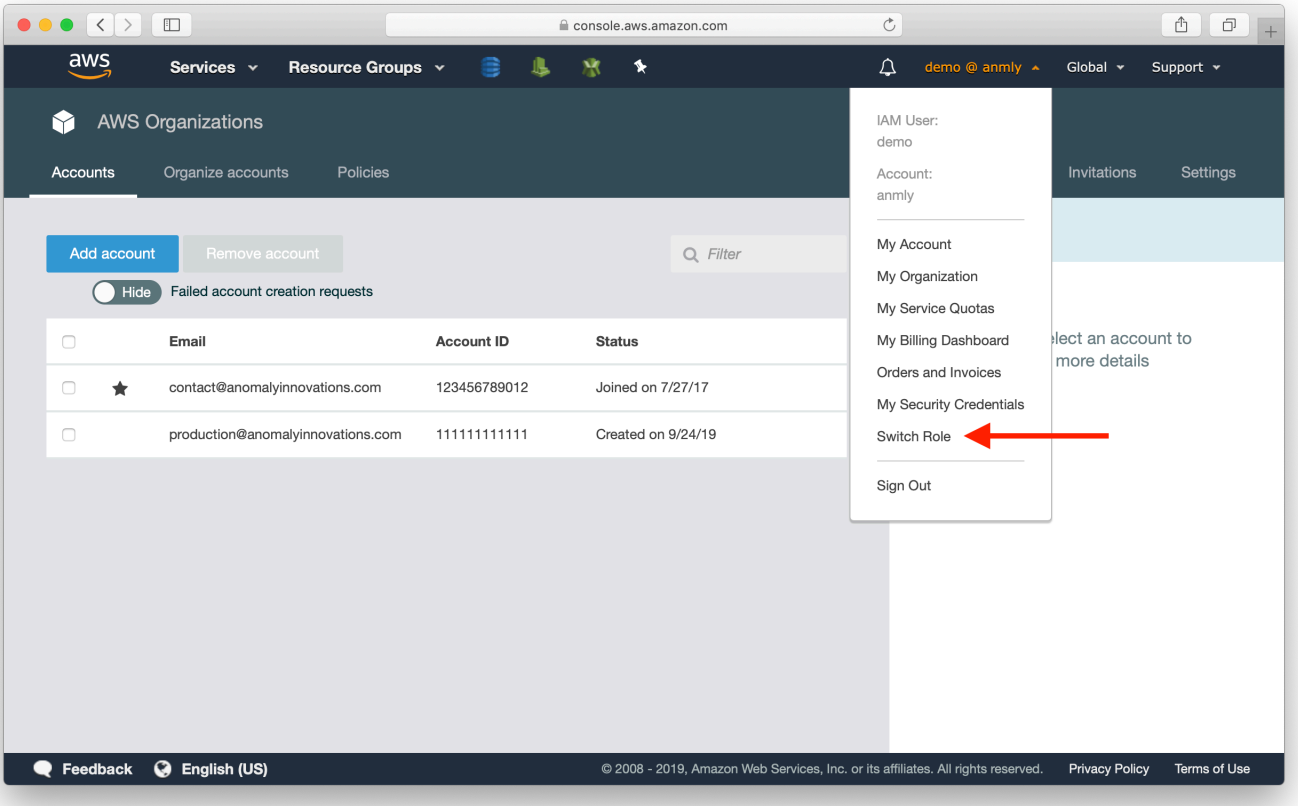
Access AWS accounts

Next, let's try switch into the Production account. First, take a note of the newly created **Account ID**. We need this number in the next step.

Then, select the account picker at the top.



Select **Switch Role**.

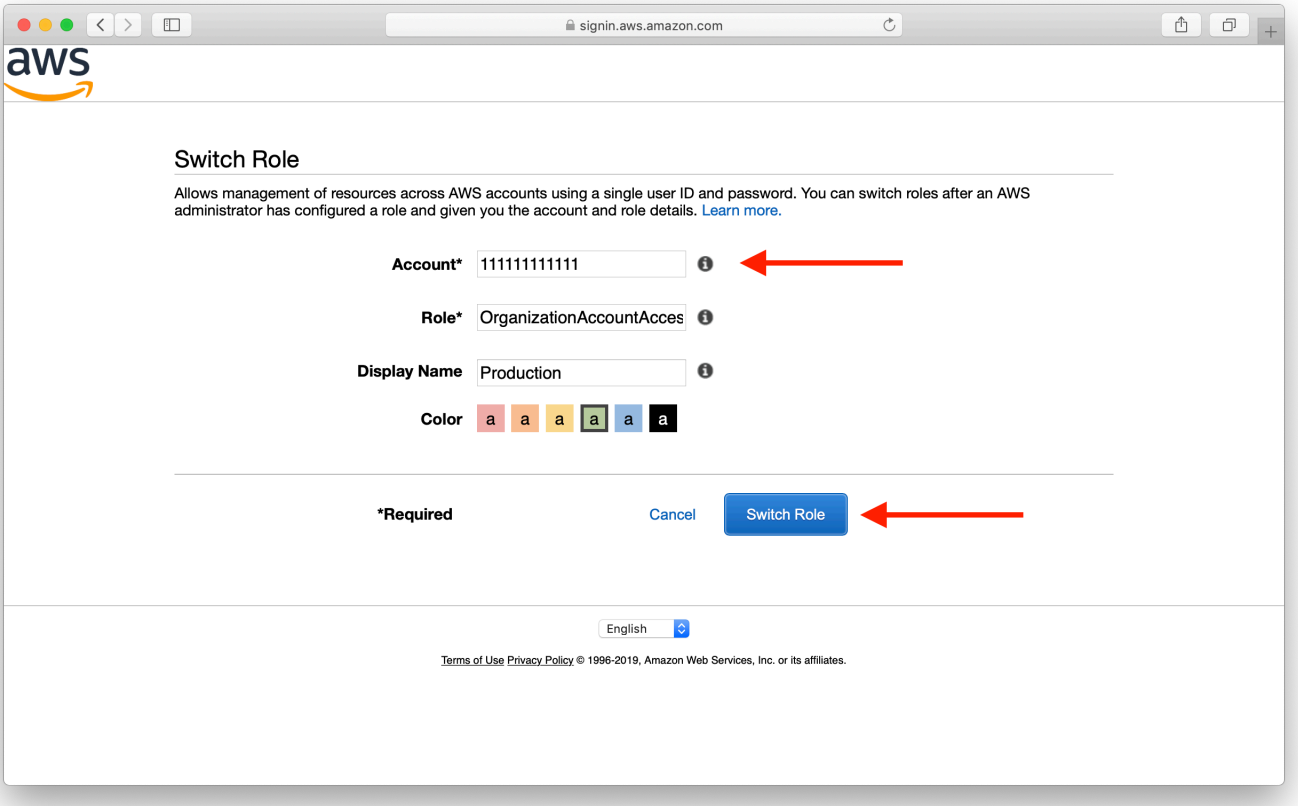


Fill in the following:

- **Account**: Account ID of the newly created Prod account from the previous step.
- **Role**: Name of the IAM role from the previous step. If you left it blank, use **OrganizationAccountAccessRole**.
- **Display Name**: It's good to use the name (Full name) from when we created the account. It'll help keep things recognizable.
- **Color**: Pick a color that represents **Production** for you.

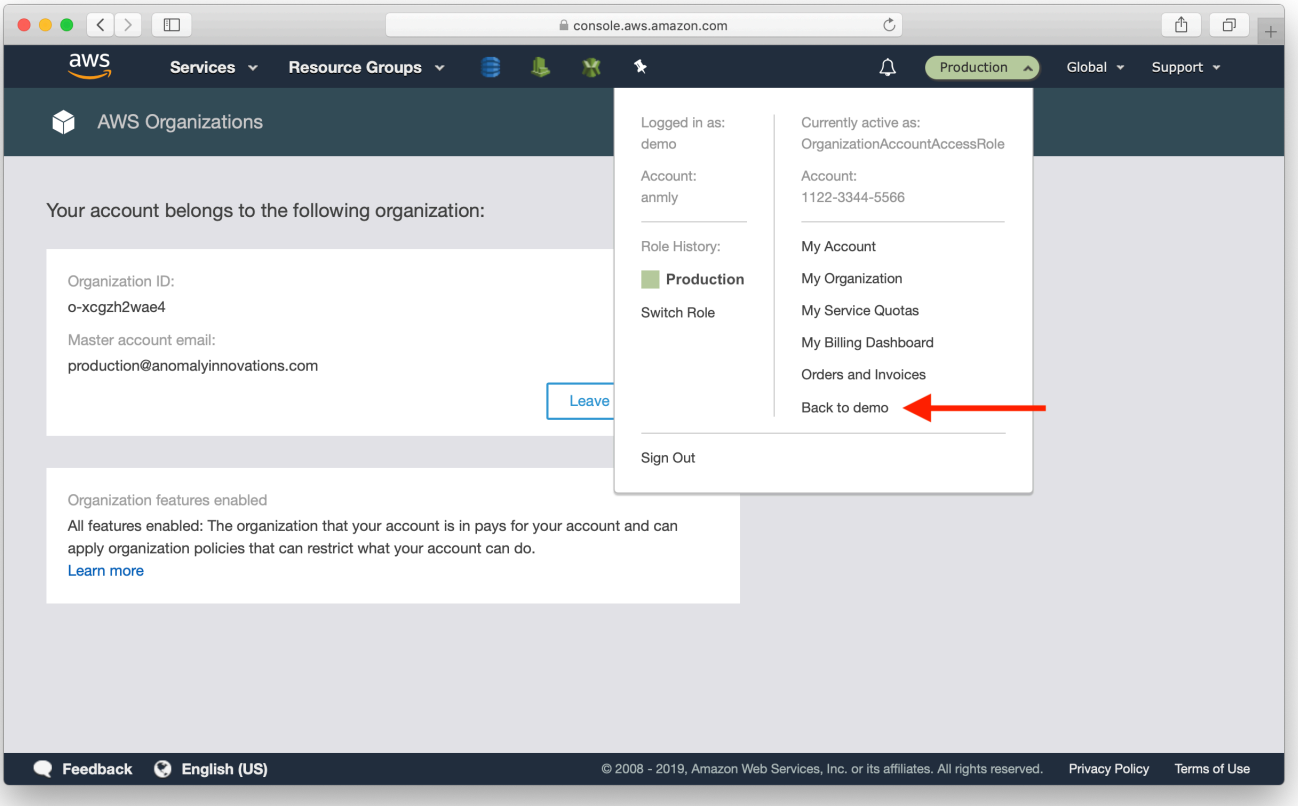
Note that the Display Name and Color fields are personal to you. Your team members will need to set this up again on their own.

Then select **Switch Role**.

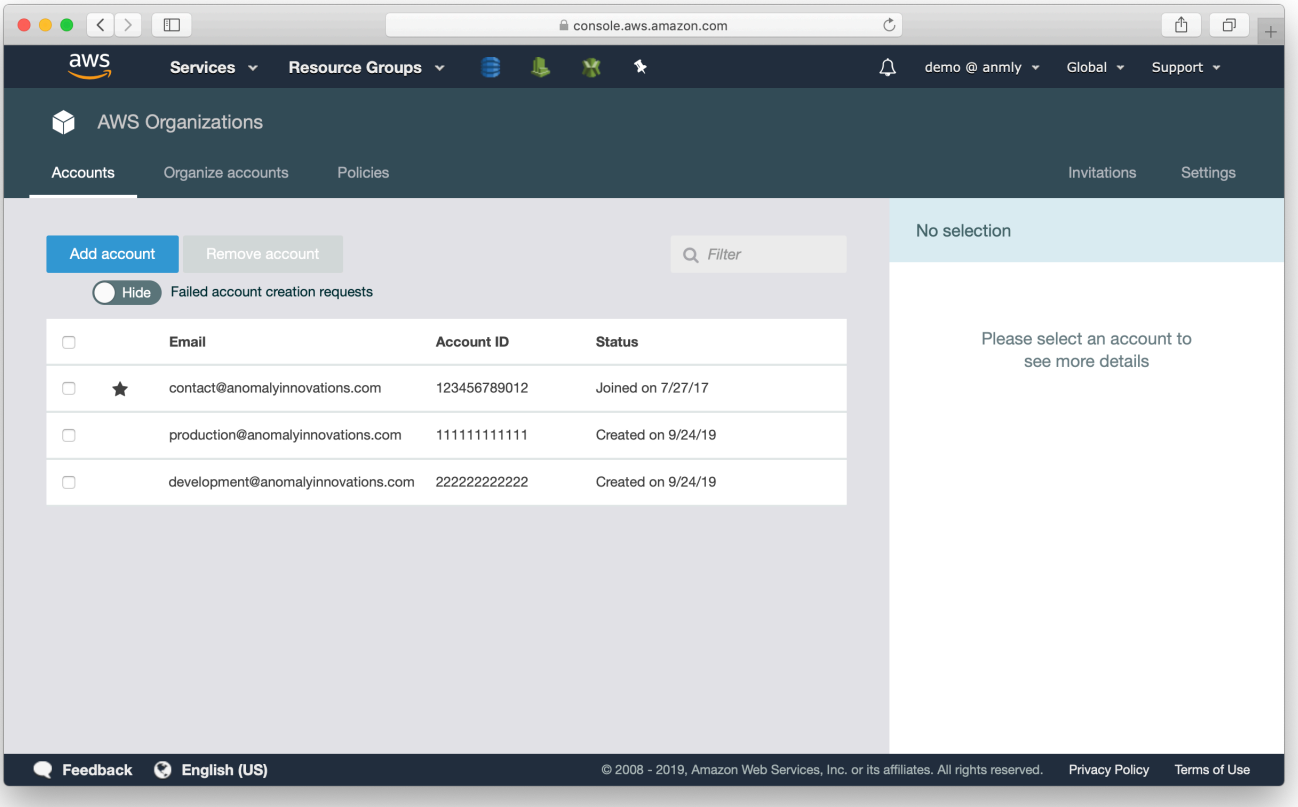


Now, you are in the **Prod** account. You can check which account you are currently assumed into by looking at the top bar.

You can switch back to the master account by clicking on the account picker and selecting **Back to master** .



Next, repeat the above steps to create the Development account.



Now we have our AWS accounts created. Let's make sure we are using these environments correctly in the configuration of our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/manage-aws-accounts-using-aws-organizations/1326)

Parameterize Serverless Resources Names

When deploying multiple environments, some into the same AWS account, some across multiple AWS accounts, we need to ensure the resource names do not thrash across environments. For example, in our `checkout-api` service, we have a Lambda function called `checkout`. Now, if two developers are working on two different features, one deploys to the `featureA` environment and one deploys to the `featureB` environment, and both environments reside in the `Dev` AWS account, only one environment can be successfully deployed. The second environment will get an error indicating that a Lambda function with the name `checkout` already exists.

AWS resources need to be uniquely named within a scope, and the scope is different for different resource types. Here are the rules for some of the commonly used serverless resources:

- Unique per account per region: Lambda functions, API Gateway projects, SNS Topic, etc.
- Unique per account (across all regions): IAM users/roles
- Unique globally: S3 buckets

The best practice to ensure uniqueness is by parameterizing resource names with the name of the stage. In our example, we can name the Lambda function `checkout-featureA` for the `featureA` stage;

`checkout-featureB` for the `featureB` stage; and `checkout-dev` for the `dev` stage.

Luckily, Serverless Framework already parameterizes a few of the default resources:

Resource	Scheme	Example
Lambda functions	`$serviceName-$stage-$functionName`	notes-app-ext-notes-api-dev-get
API Gateway project	`$stage-$serviceName`	dev-notes-app-ext-notes-api
CloudWatch log groups	`/aws/lambda/$serviceName-$stage-$functionName`	/aws/lambda/notes-app-ext-notes-api-dev-get
IAM roles	`$serviceName-$stage-$region-lambdaRole`	notes-app-ext-notes-api-dev-us-east-1-lambdaRole
S3 bucket	`$stackName-$resourceName-$hash`	notes-app-ext-notes-api-serverlessdeploymentbuck-19fhidl3prw0m

A couple of things to note here:

- Resource names are parameterized with `$serviceName` to ensure resource names do not thrash when deploying multiple services

- The IAM role is the one used by the Lambda functions. IAM role names are also parameterized with `$region` since the name needs to be unique across regions in an account.
- S3 bucket is the one used by Serverless Framework to store deployment artifacts. It is not given a name. In these cases, CloudFormation will automatically assign a unique name for it based on the name of the current stack — `$stackName`.

For all the other resources we define in our `serverless.yml`, we are responsible for parameterizing them.

Here are a couple of examples where we need to be aware of resource names being parameterized.

SNS topic names in `billing-api` service

```
resources:
  Resources:
    NotePurchasedTopic:
      Type: AWS::SNS::Topic
      Properties:
        TopicName: note-purchased-${self:custom.stage}
```

DynamoDB table names in `database` service

```
...
custom:
  stage: ${opt:stage, self:provider.stage}
  tableName: ${self:custom.stage}-ext-notes
```

```
...
resources:
  Resources:
    NotesTable:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: ${self:custom.tableName}
        AttributeDefinitions:
          - AttributeName: userId
            AttributeType: S
          - AttributeName: noteId
            AttributeType: S
        KeySchema:
          - AttributeName: userId
            KeyType: HASH
          - AttributeName: noteId
            KeyType: RANGE
        BillingMode: PAY_PER_REQUEST
```

S3 bucket names in `uploads` service

Since, S3 bucket names need to be globally unique; leave the bucket name empty and let CloudFormation auto generate it.

```
...
resources:
  Resources:
    S3Bucket:
      Type: AWS::S3::Bucket
      Properties:
        # Set the CORS policy
```

```
      CorsConfiguration:
        CorsRules:
          -
            AllowedOrigins:
              - '*'
            AllowedHeaders:
              - '*'
            AllowedMethods:
              - GET
              - PUT
              - POST
              - DELETE
              - HEAD
            MaxAge: 3000
```

Parameterizing your resources allows your app to be deployed to multiple environments without naming conflicts. Next, let's deploy our app!

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/parameterize-serverless-resources-names/1329)

Deploying to Multiple AWS Accounts

Now that you have a couple of AWS accounts created and your resources have been parameterized, let's look at how to deploy them. In this chapter, we'll deploy the following:

1. The resources repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources) will be deployed in phases to the `dev` and `prod` stage. These two stages are configured in our `Development` and `Production` AWS accounts respectively.

2. Then we'll do the same with the APIs repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api).

Configure AWS Profiles

Follow the setup up IAM users (/chapters/create-an-iam-user.html) chapter to create an IAM user in your `Development` AWS account. And take a note of the **Access key ID** and **Secret access key** for the user.

Next, set these credentials in your local machine using the AWS CLI:

```
$ aws configure --profile default
```

This sets the default IAM credentials to those of the Development account. Meaning when you run `serverless deploy`, a service will get deployed into the Development account.

Repeat the step to create an IAM user in your `Production` account. And make a note of the credentials. We will not add the IAM credentials for the Production account on our local machine. This is because we do not want to be able to deploy code to the Production environment EVER from our local machine.

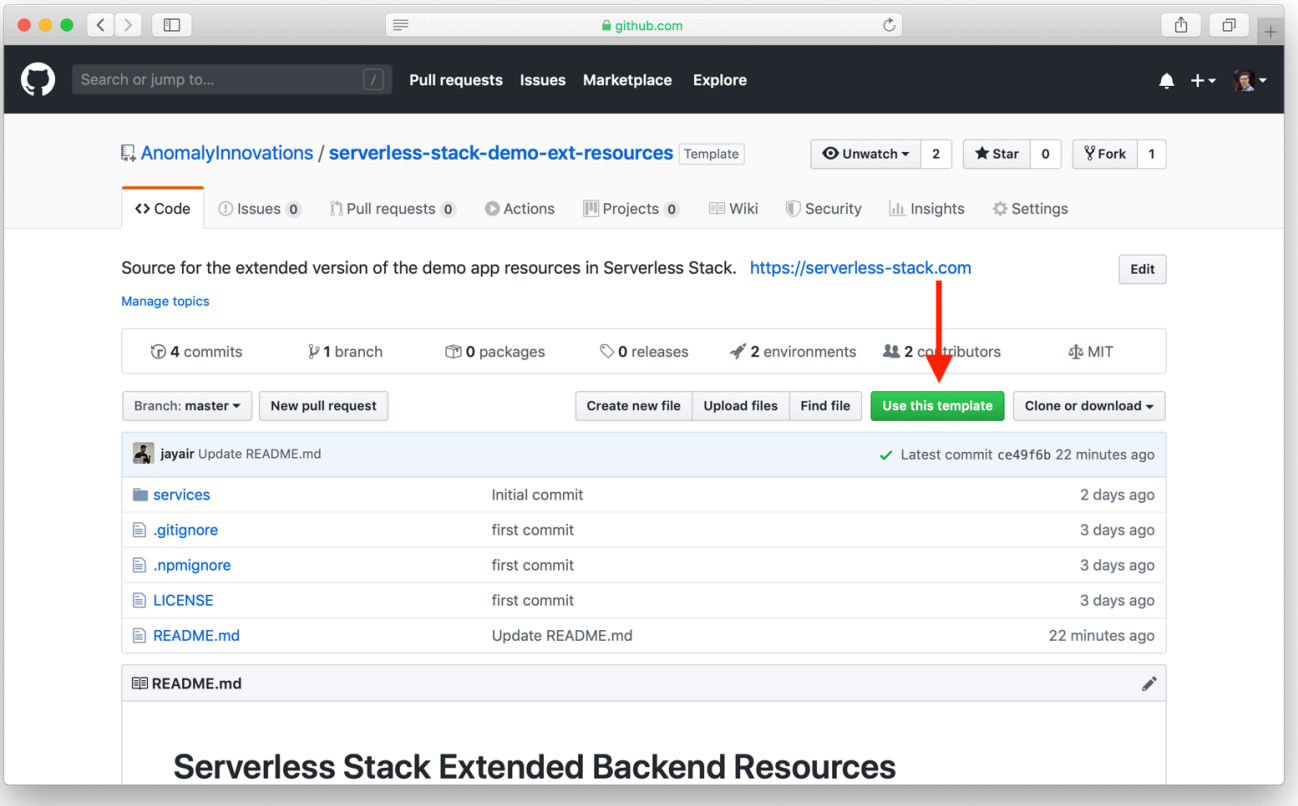
Production deployments should always go through our CI/CD pipeline.

Next we are going to deploy our two repos to our environments. We want you to follow along so you can get a really good sense of what the workflow is like.

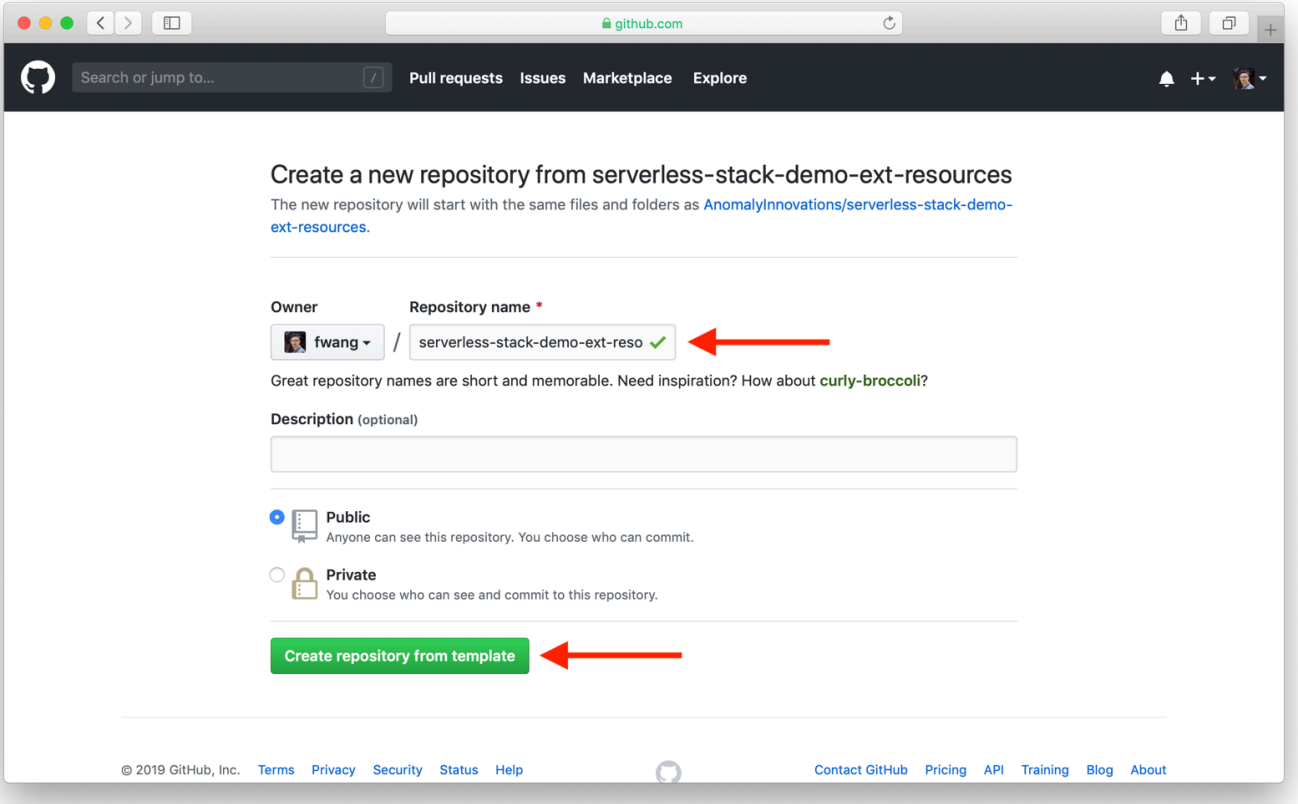
So let's start by using the demo repo templates from GitHub.

Create demo repos

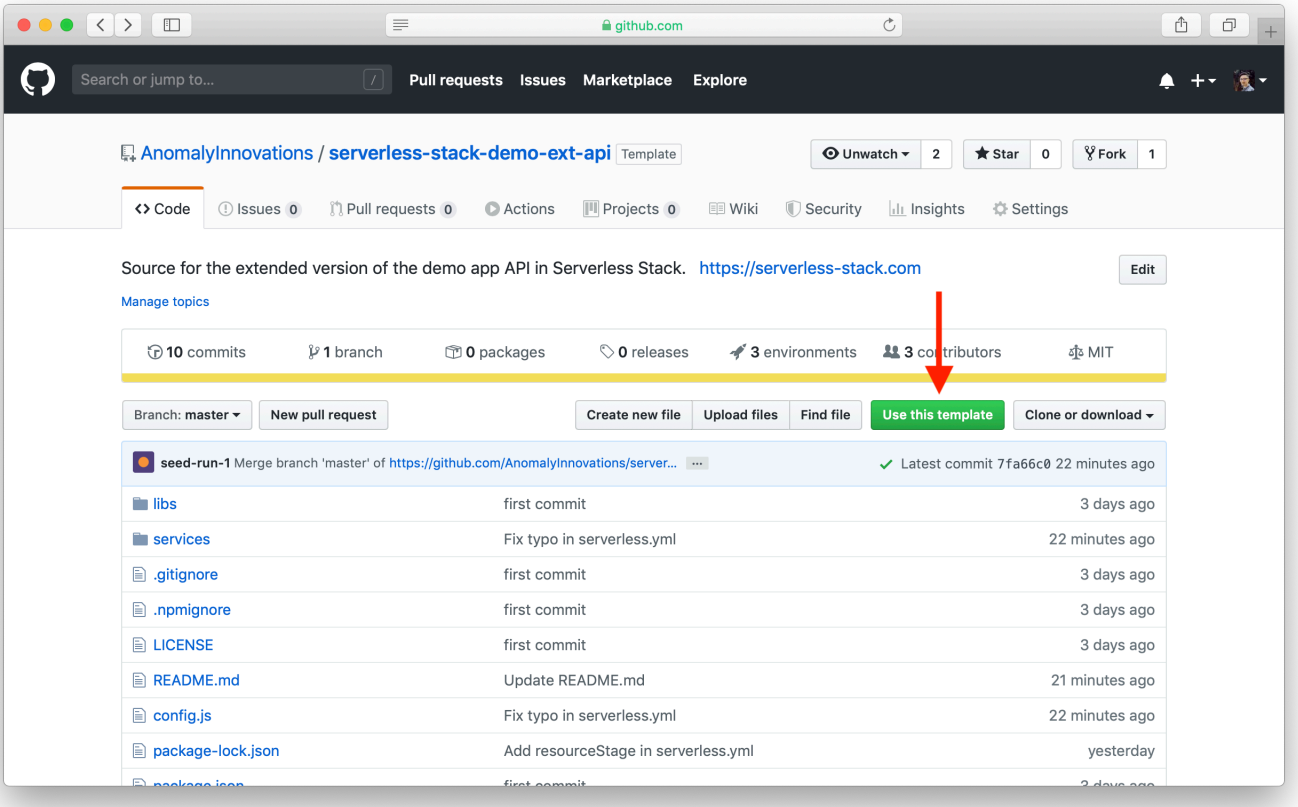
Let's first create the resources repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources). Click **Use this template**.



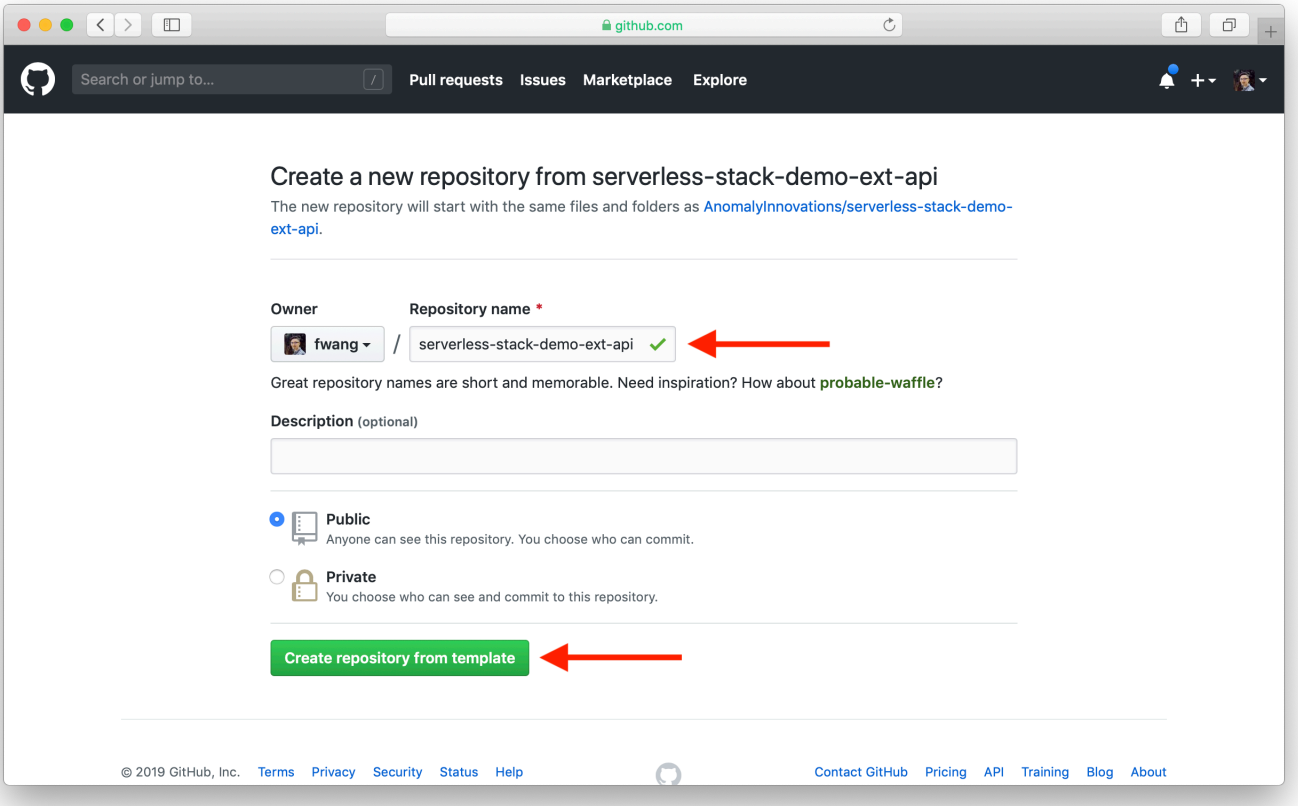
Enter Repository name **serverless-stack-demo-ext-resources** and click **Create repository from template**.



And do the same for the API services repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api).



Enter Repository name **serverless-stack-demo-ext-api** and click **Create repository from template**.



Now that we've forked these repos, let's deploy them to our environments. We are going to use Seed (https://seed.run) to do this but you can set this up later with your favorite CI/CD tool.

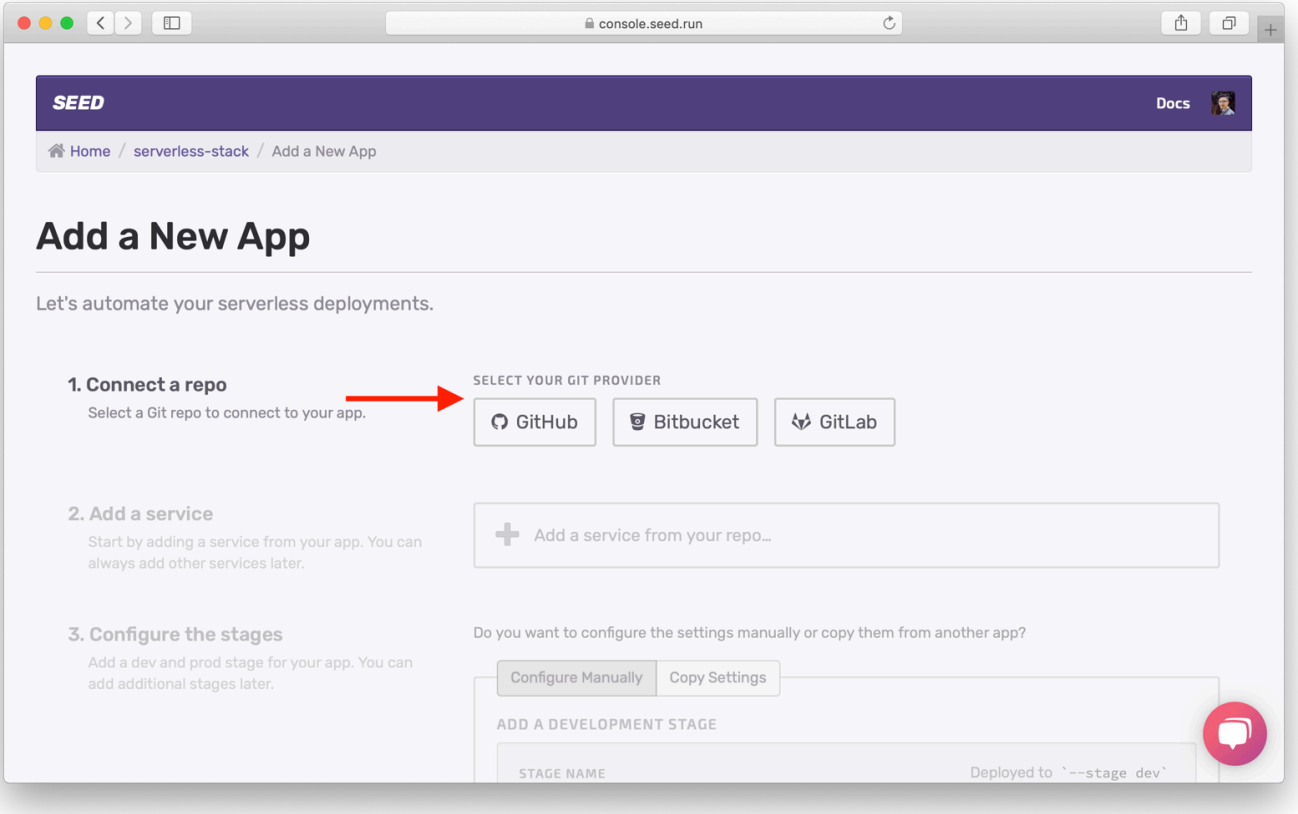
For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploying-to-multiple-aws-accounts/1322)

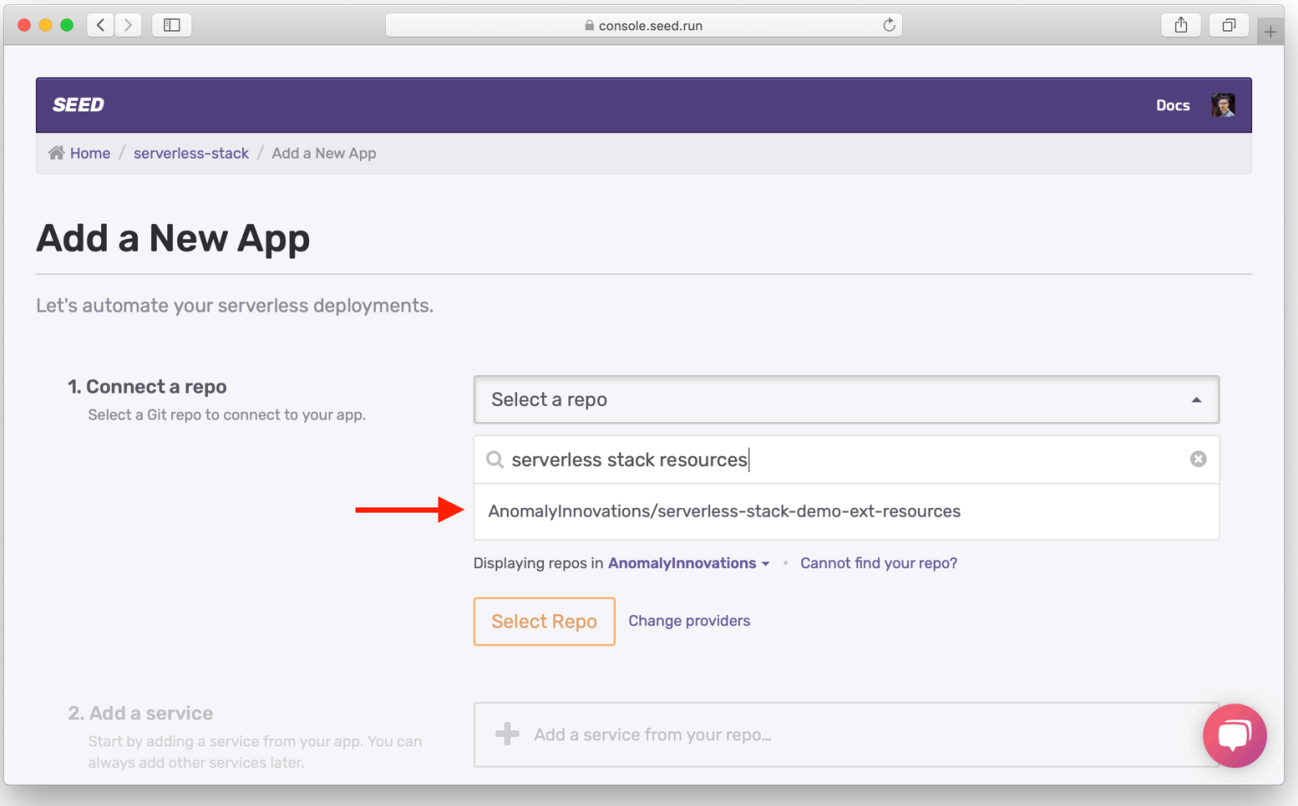
Deploy the Resources Repo

First, add the resources repo on Seed. If you haven't yet, you can create a free account here (https://console.seed.run/signup).

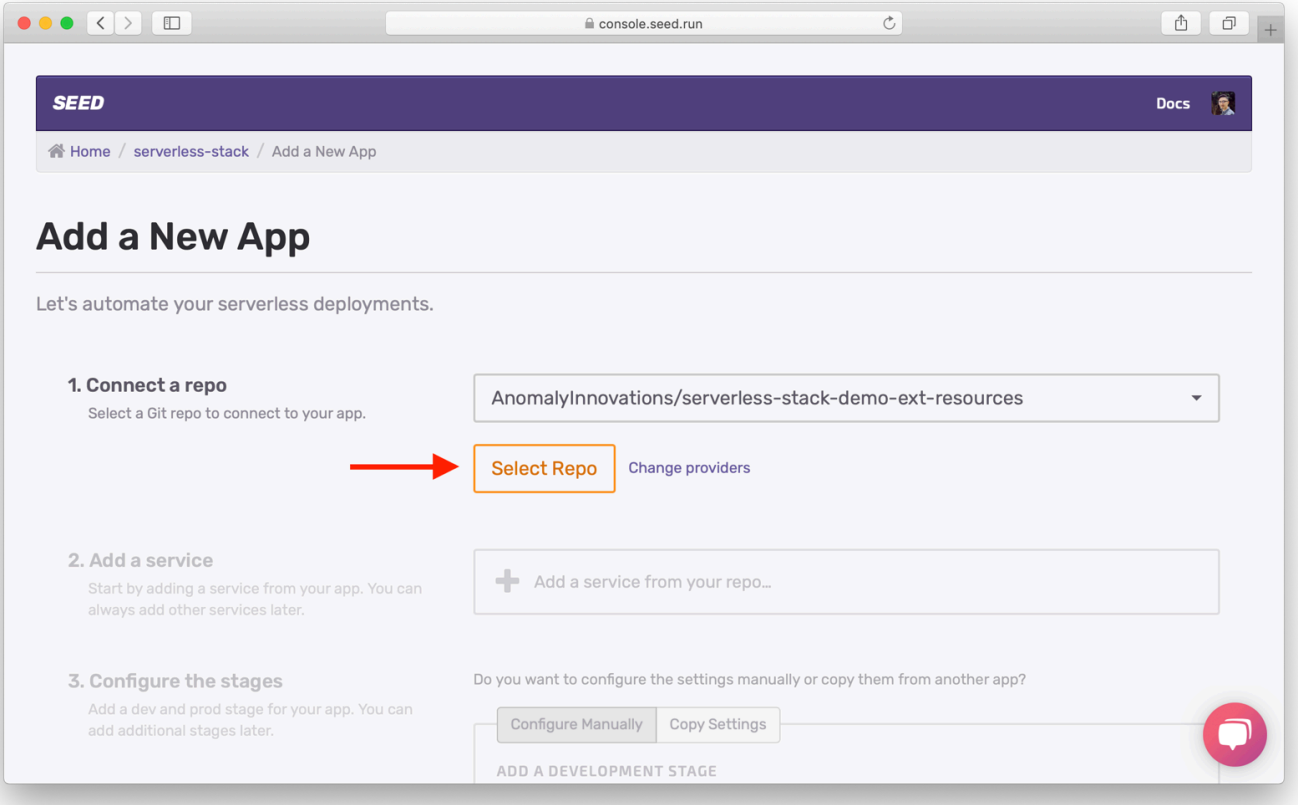
Go in to your Seed account (https://console.seed.run) and click **Add an App**, and select your Git provider.



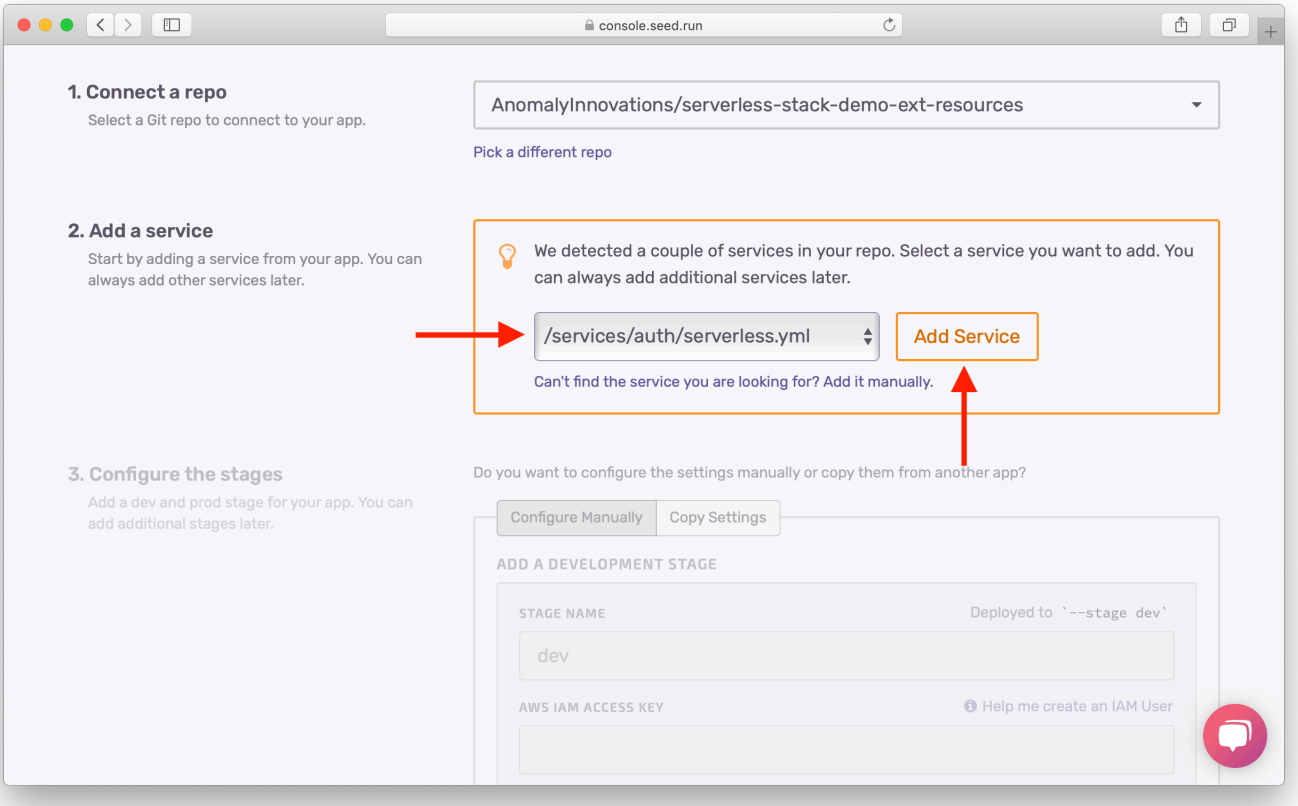
After authenticating GitHub, search for the resources repo, and select it.



Click **Select Repo**.



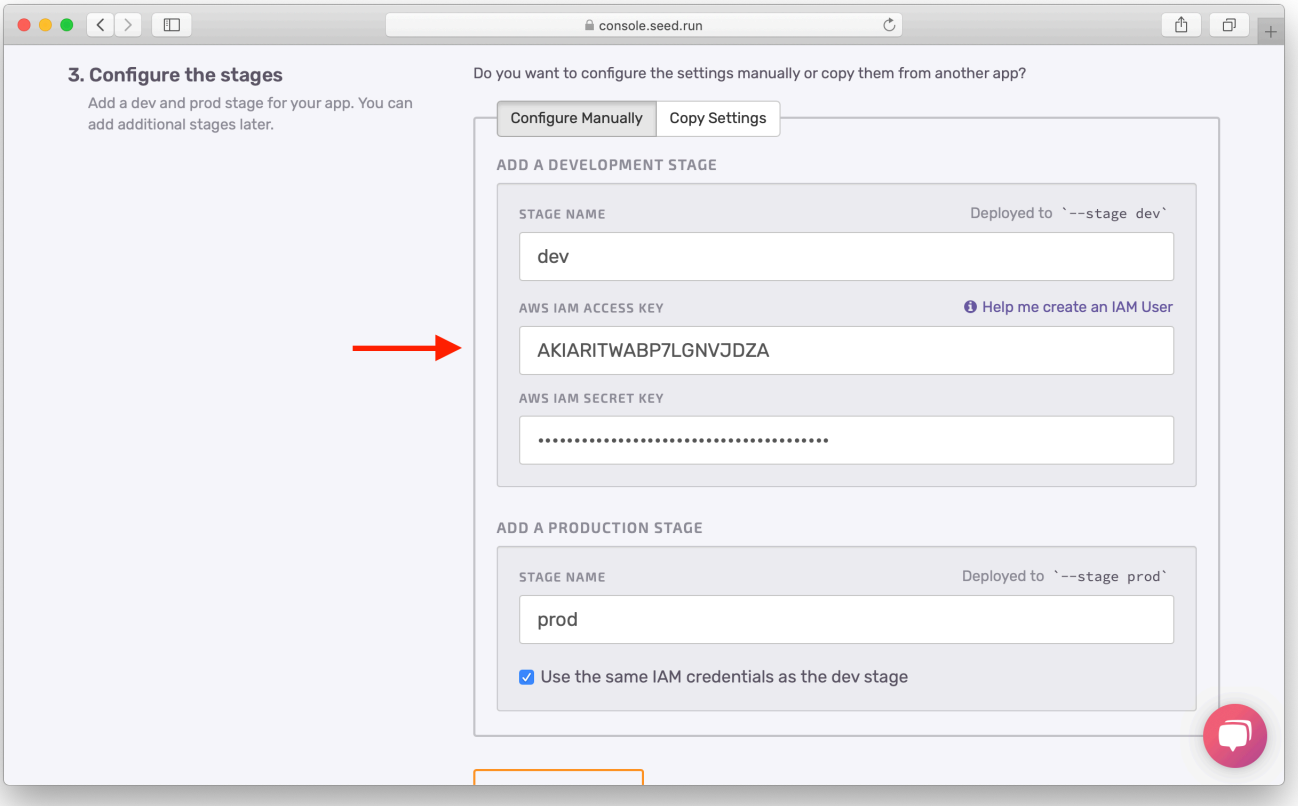
Seed will now automatically detect the Serverless services in the repo. After detection, select a service. Let's select the **auth** service. Then click **Add Service**.



By default, Seed lets you configure two stages out of the box, a **Development** and a **Production** stage. Serverless Framework has a concept of stages. They are synonymous with environments. Recall that in the previous chapter we used this stage name to parameterize our resource names.

Let's first configure the **Development** stage. Enter:

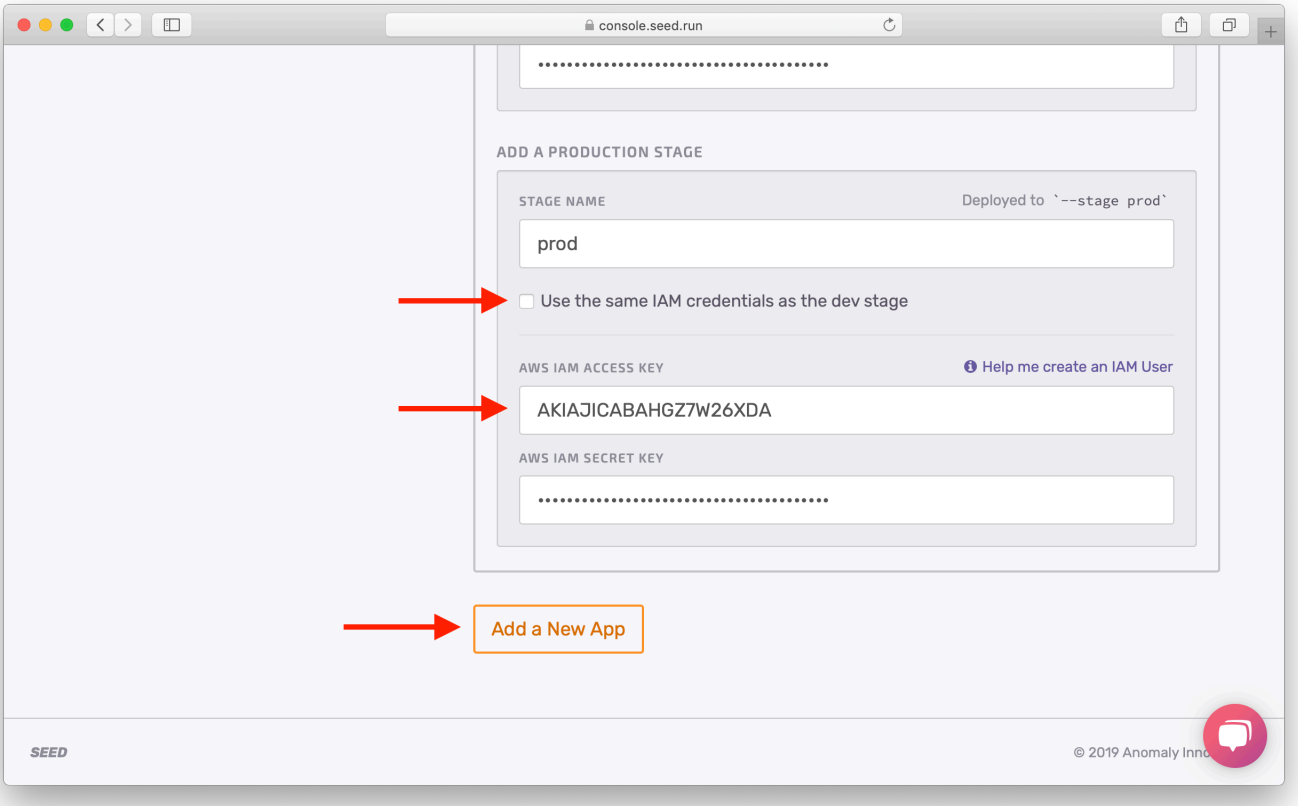
- **Stage Name**: dev
- **AWS IAM Access Key** and **AWS IAM Secret Key**: the IAM credentials of the IAM user you created in your **Development** AWS account above.



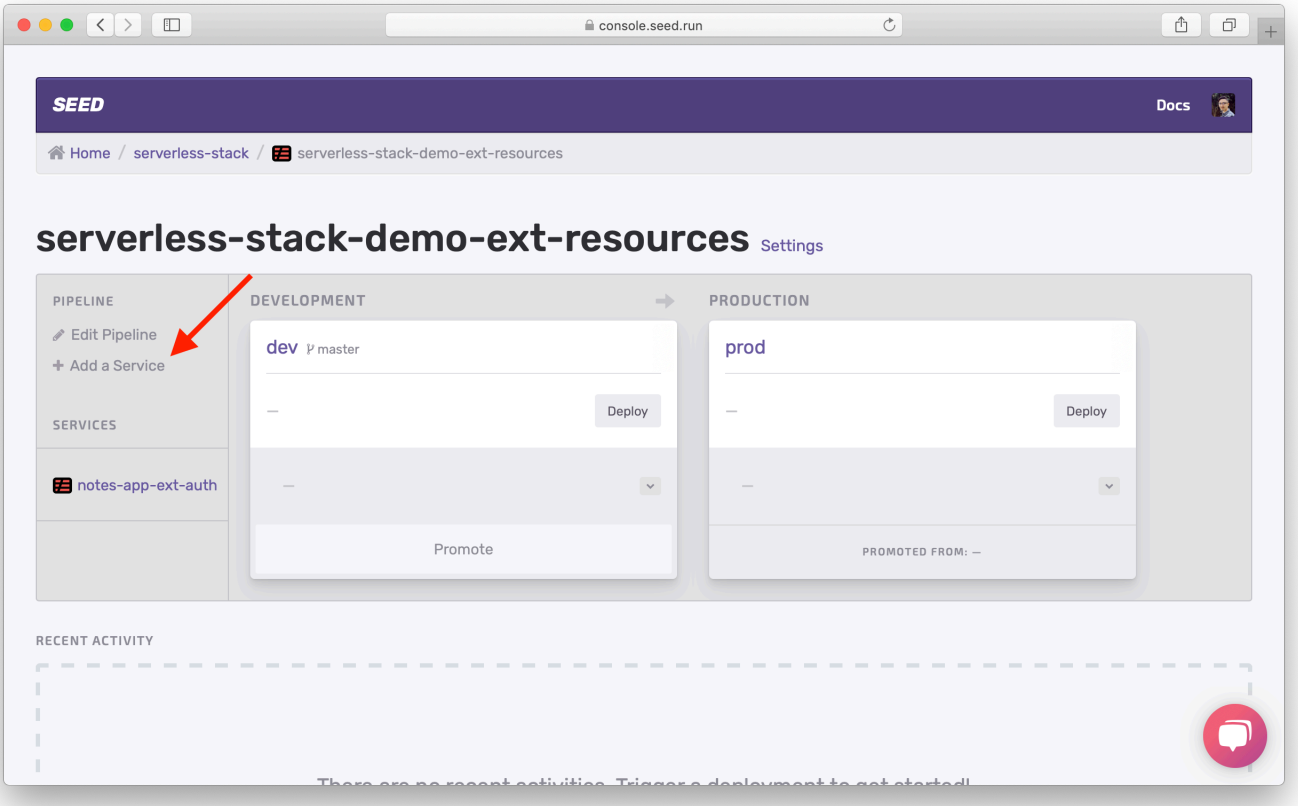
Next, let's configure the **Production** stage. Uncheck **Use the same IAM credentials as the dev stage** checkbox since we want to use a different AWS account for **Production**. Then enter:

- **Stage Name**: prod
- **AWS IAM Access Key** and **AWS IAM Secret Key**: the IAM credentials of the IAM user you created in your **Production** AWS account above.

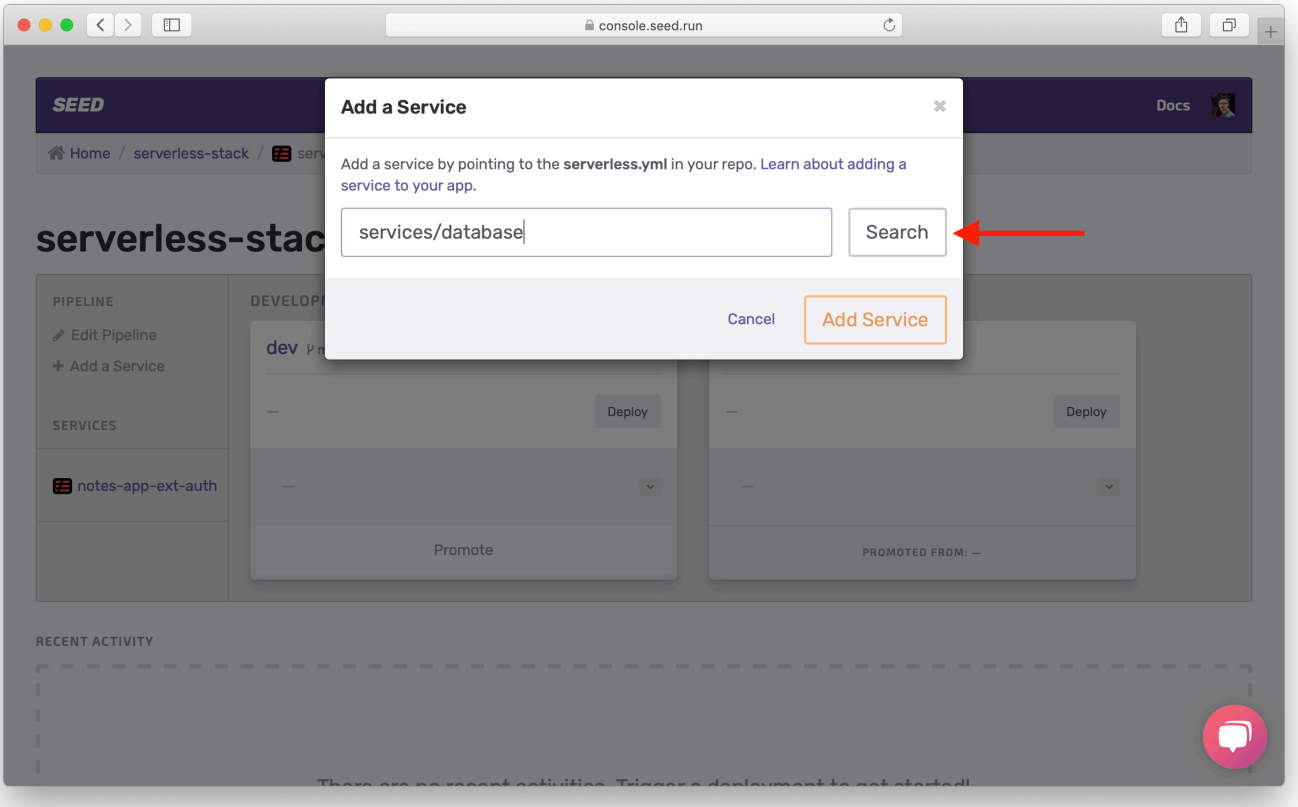
Finally hit **Add a New App**.



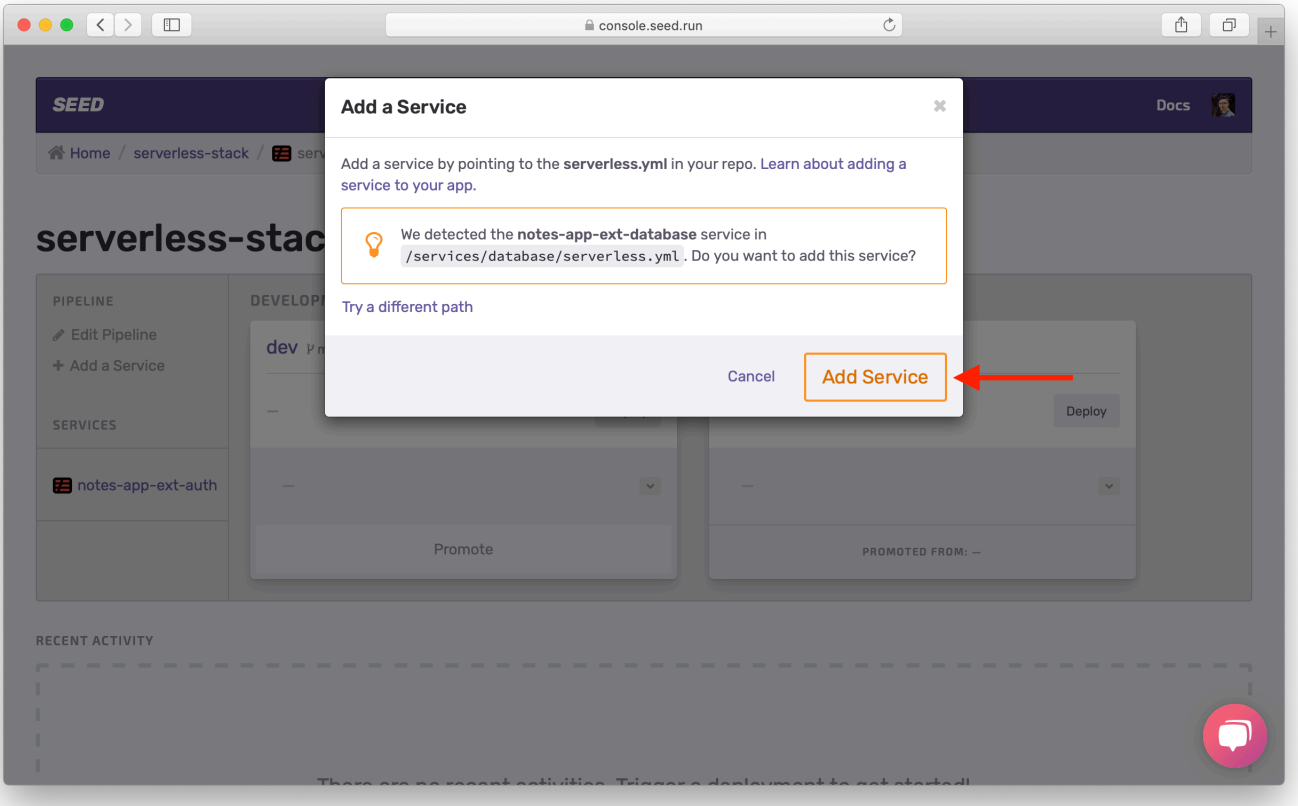
Now, let's add the other services in the resources repo. Click **Add a Service**.



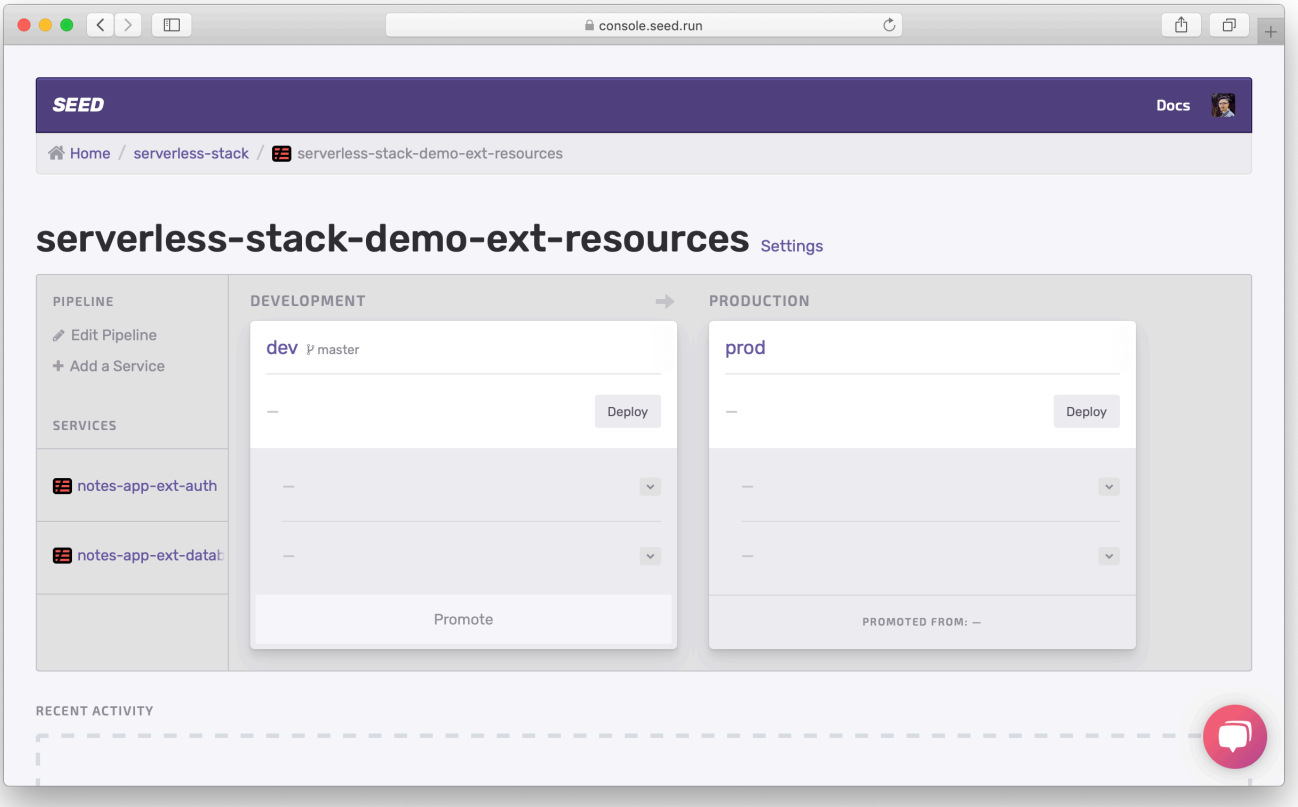
Enter the path to the **database** service `services/database`. Then hit **Search**.



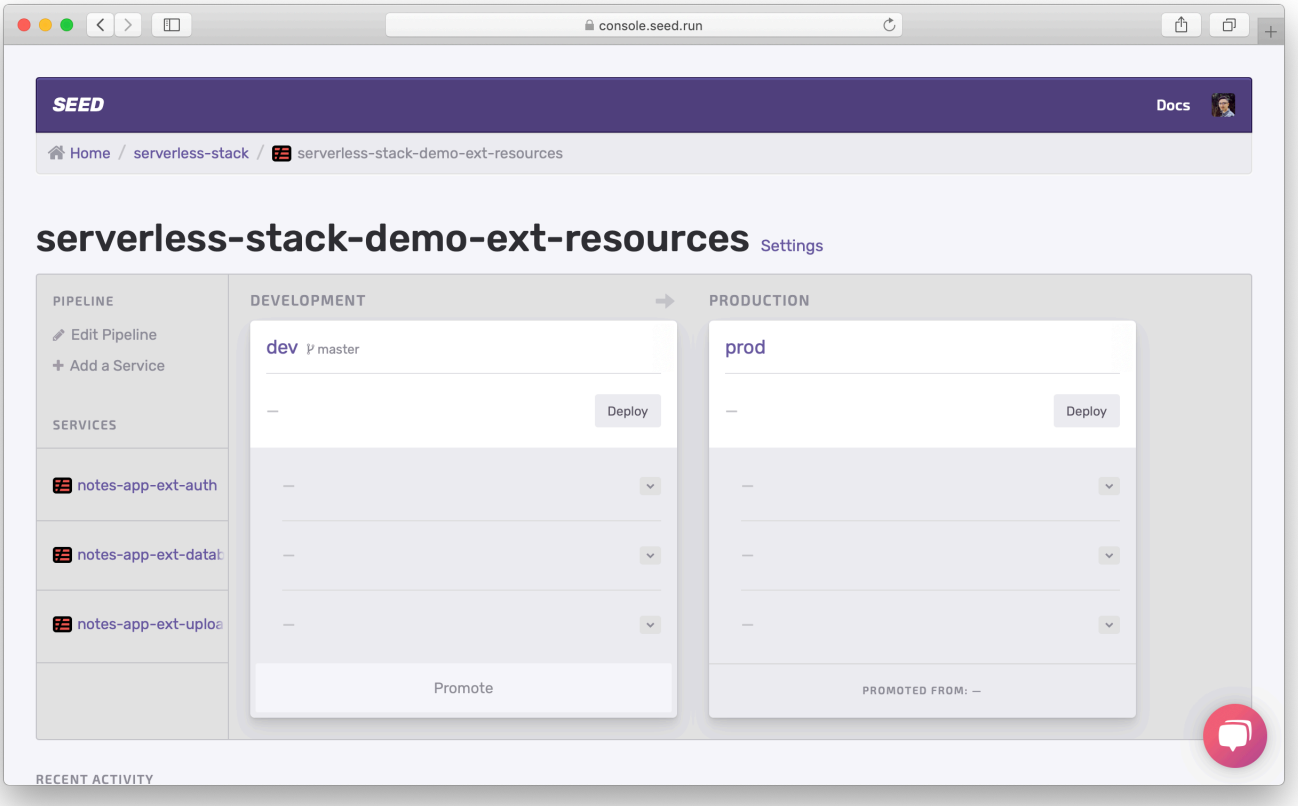
Seed will search for the `serverless.yml` file in the path, to ensure you entered the right path. Hit **Add Service**.



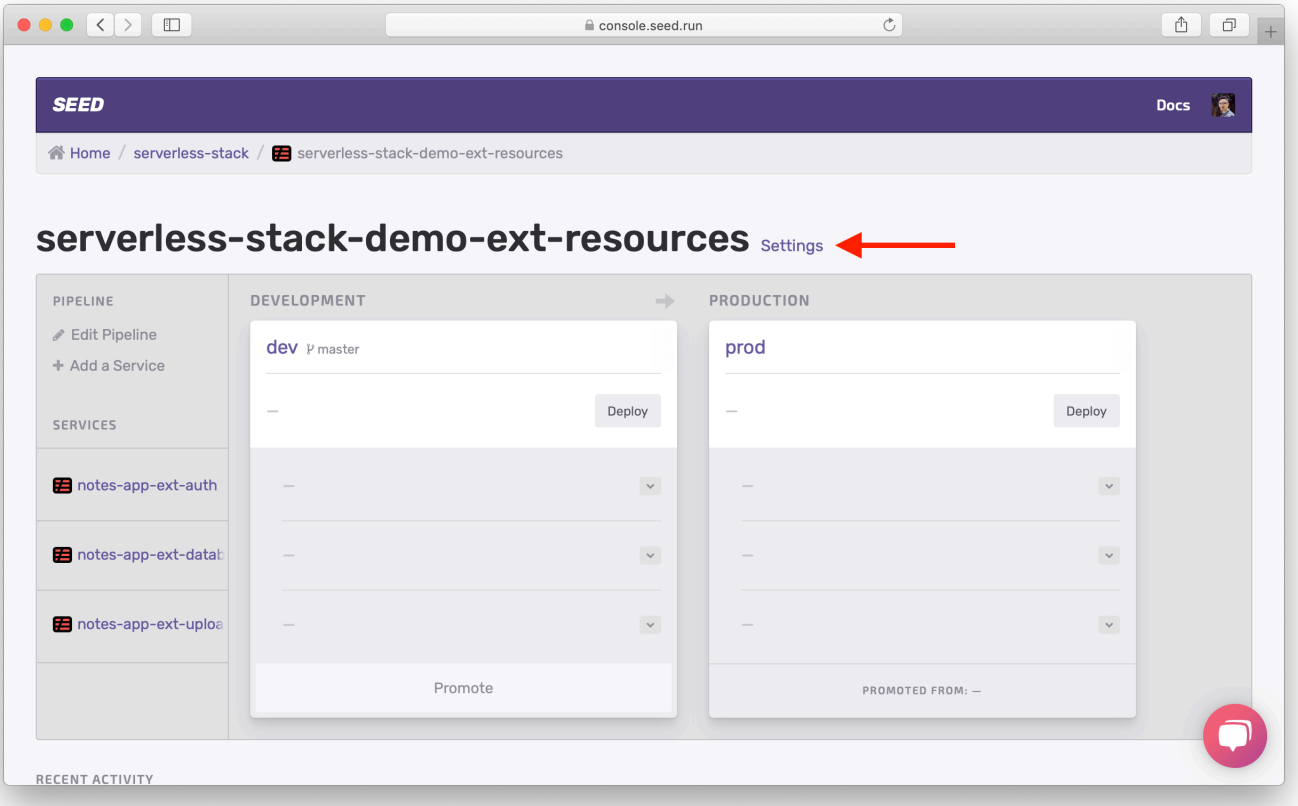
Now you have 2 services.



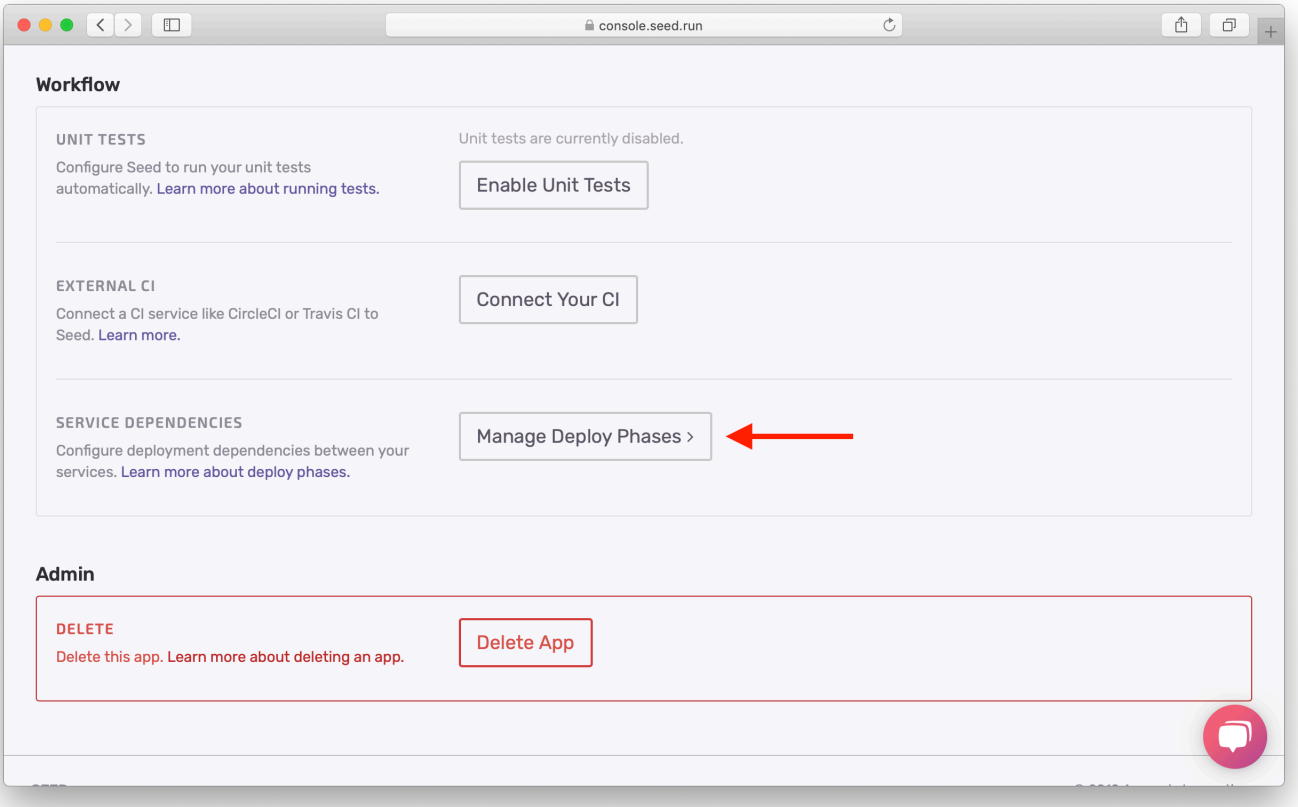
Repeat the process and add the **uploads** service in `services/uploads`.



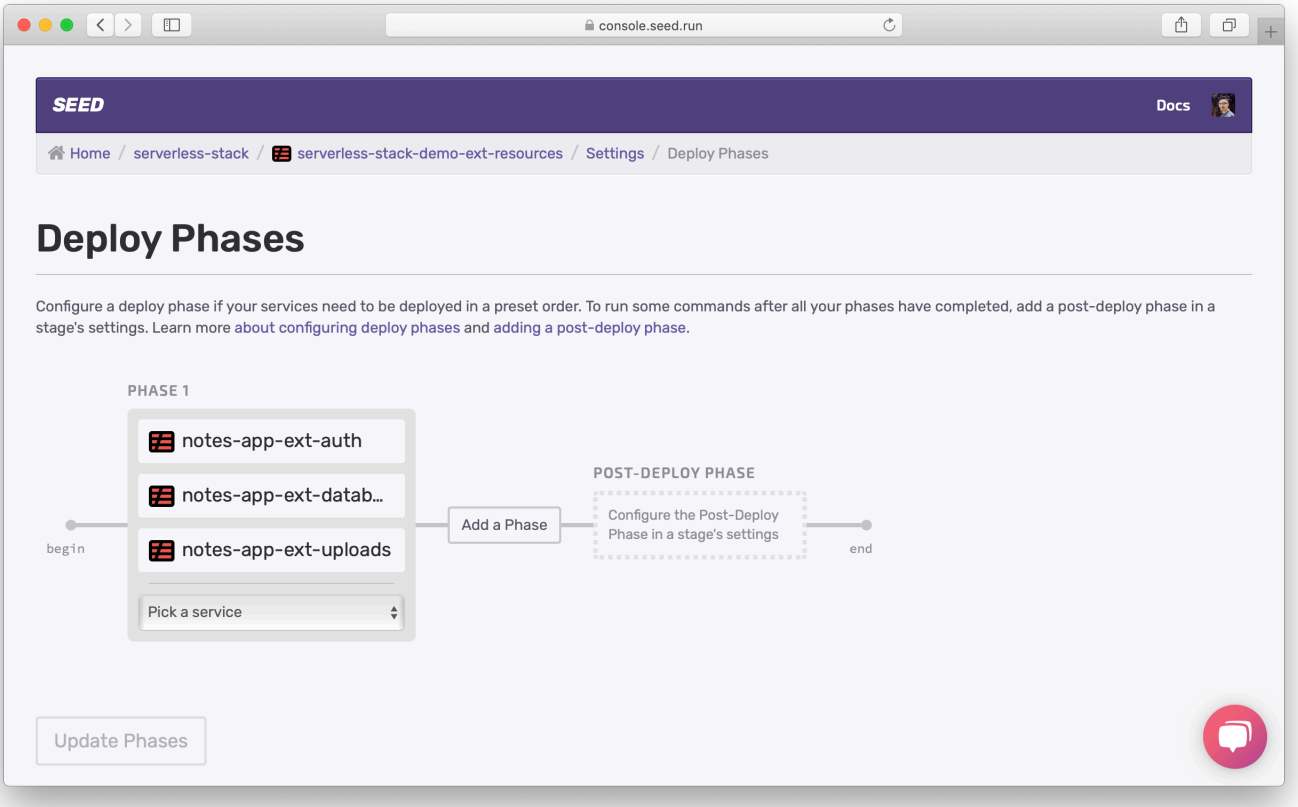
Before we deploy, let's make sure the services will deploy in the desired order. Recall from the Deploy a Serverless app with dependencies (/chapters/deploy-a-serverless-app-with-dependencies.html) chapter that you can configure the phases by heading to the app settings.



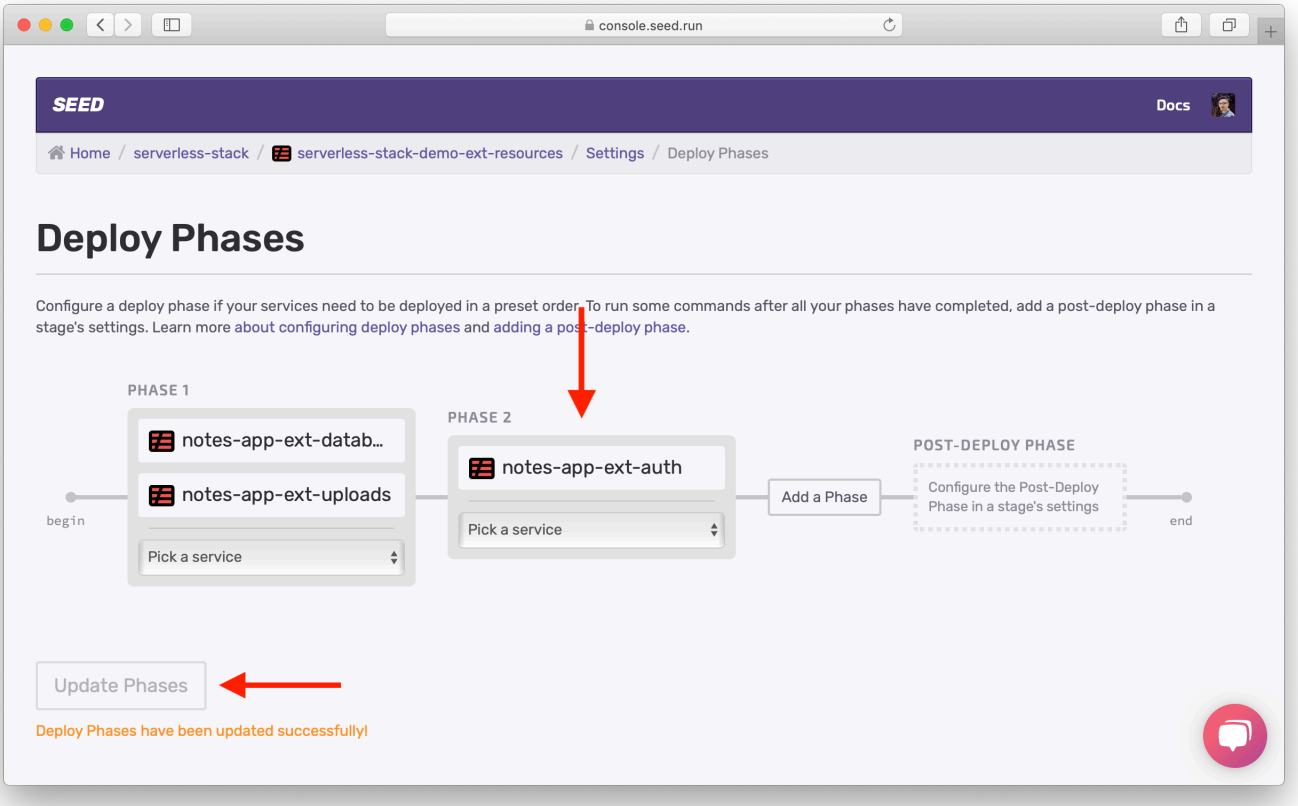
Scroll down and select **Manage Deploy Phases**.



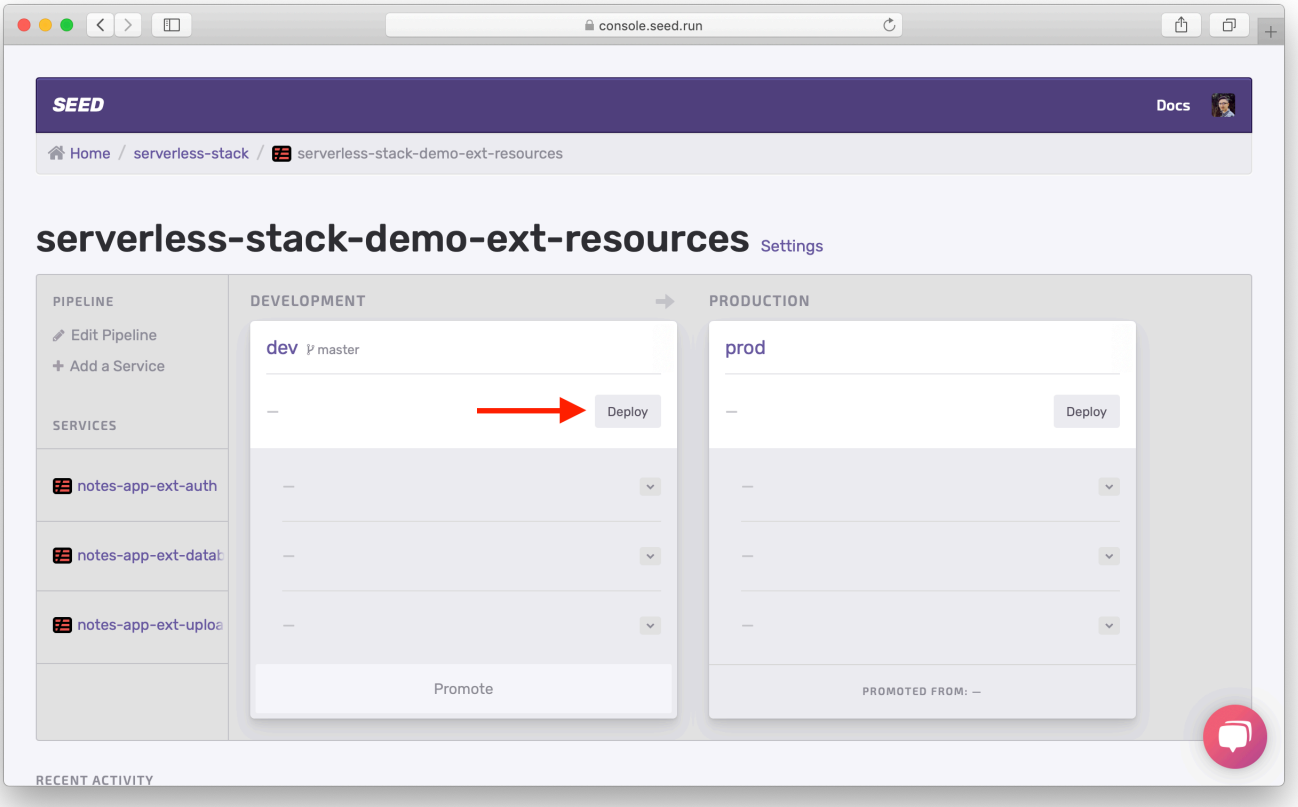
Here you'll notice that by default all the services are deployed concurrently.



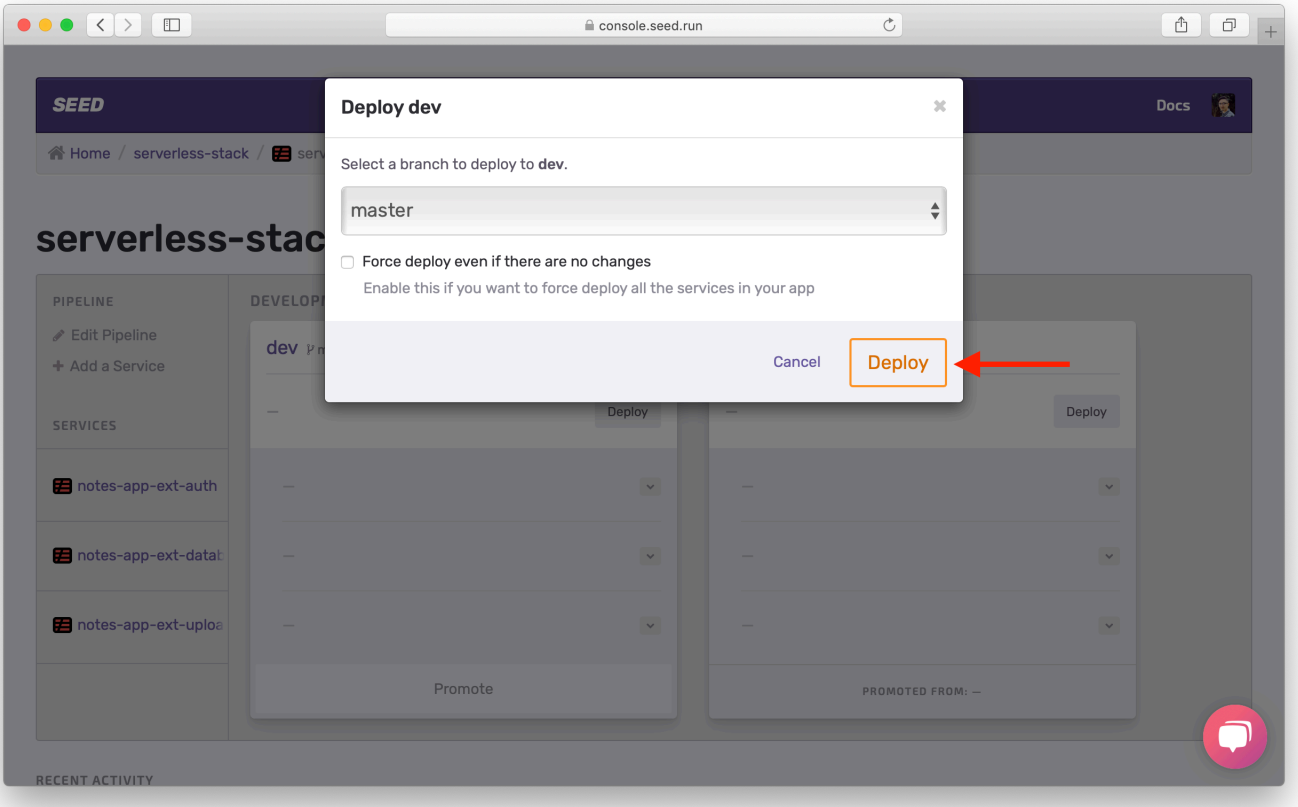
Select **Add a phase** and move the **auth** service to **Phase 2**. And hit **Update Phases**.



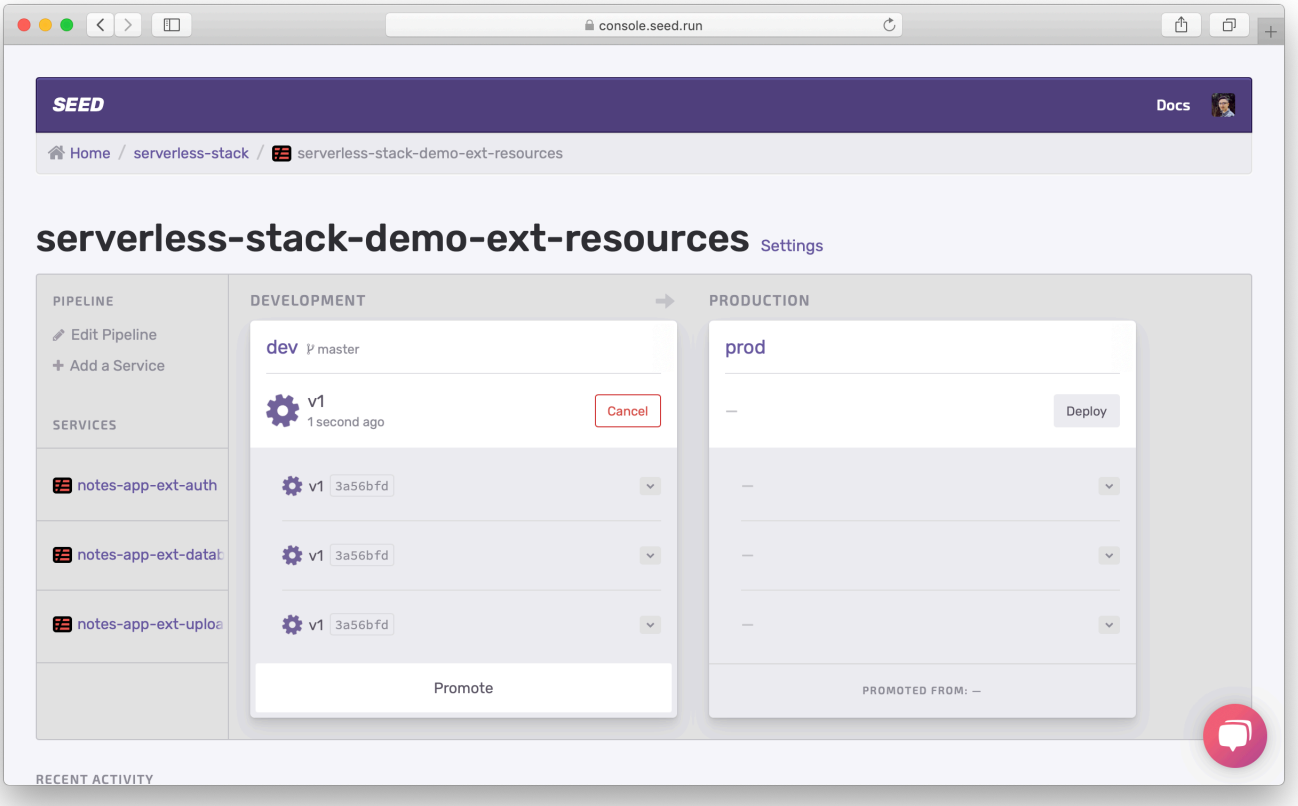
Now let's make our first deployment. Click **Deploy** under the **dev** stage.



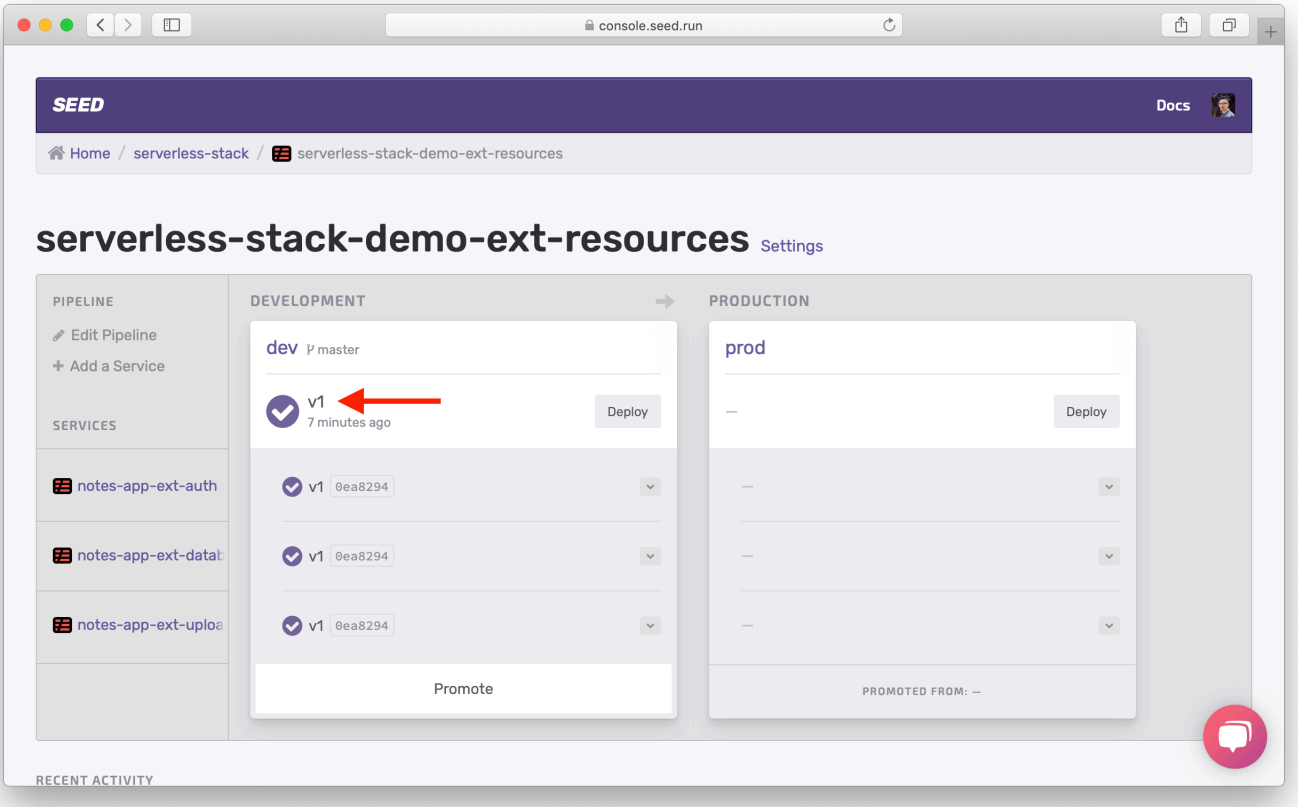
We are deploying the `master` branch here. Confirm this by clicking **Deploy**.



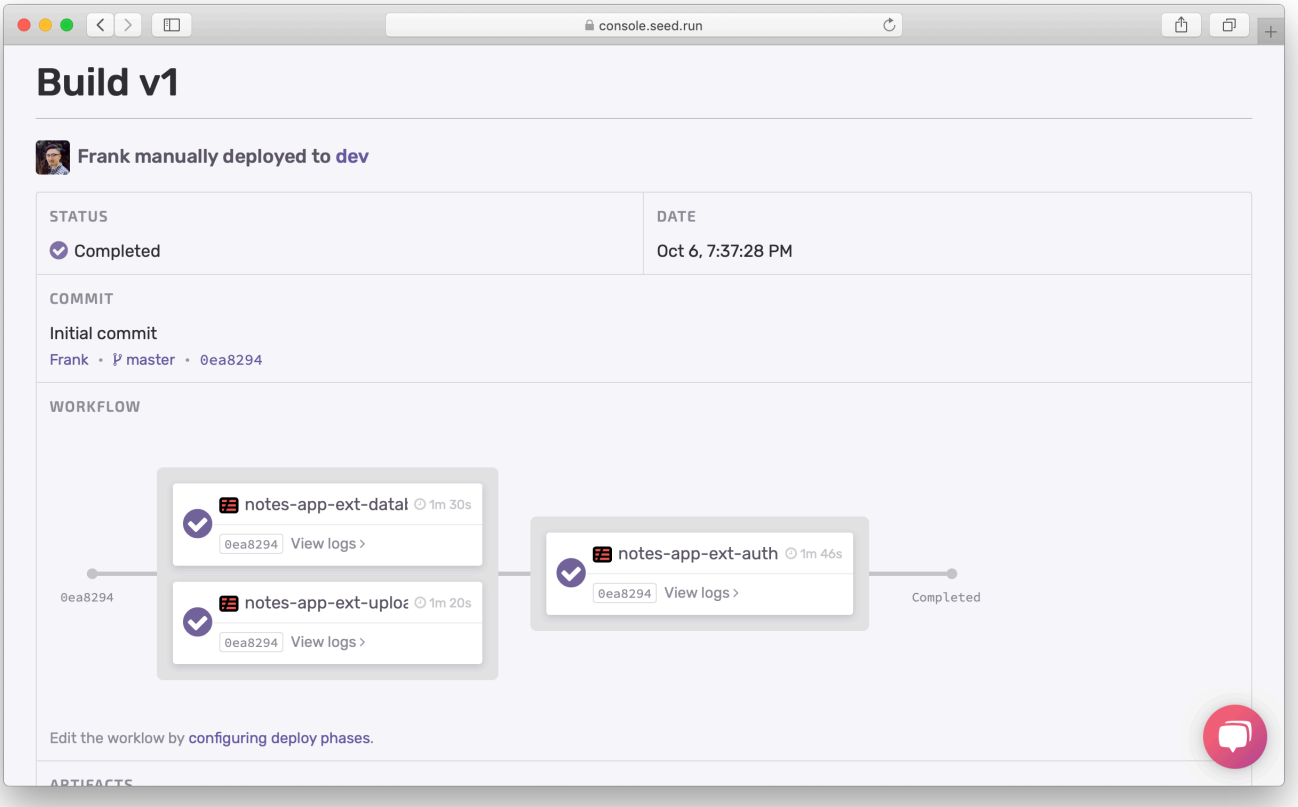
You'll notice that all the services are being deployed.



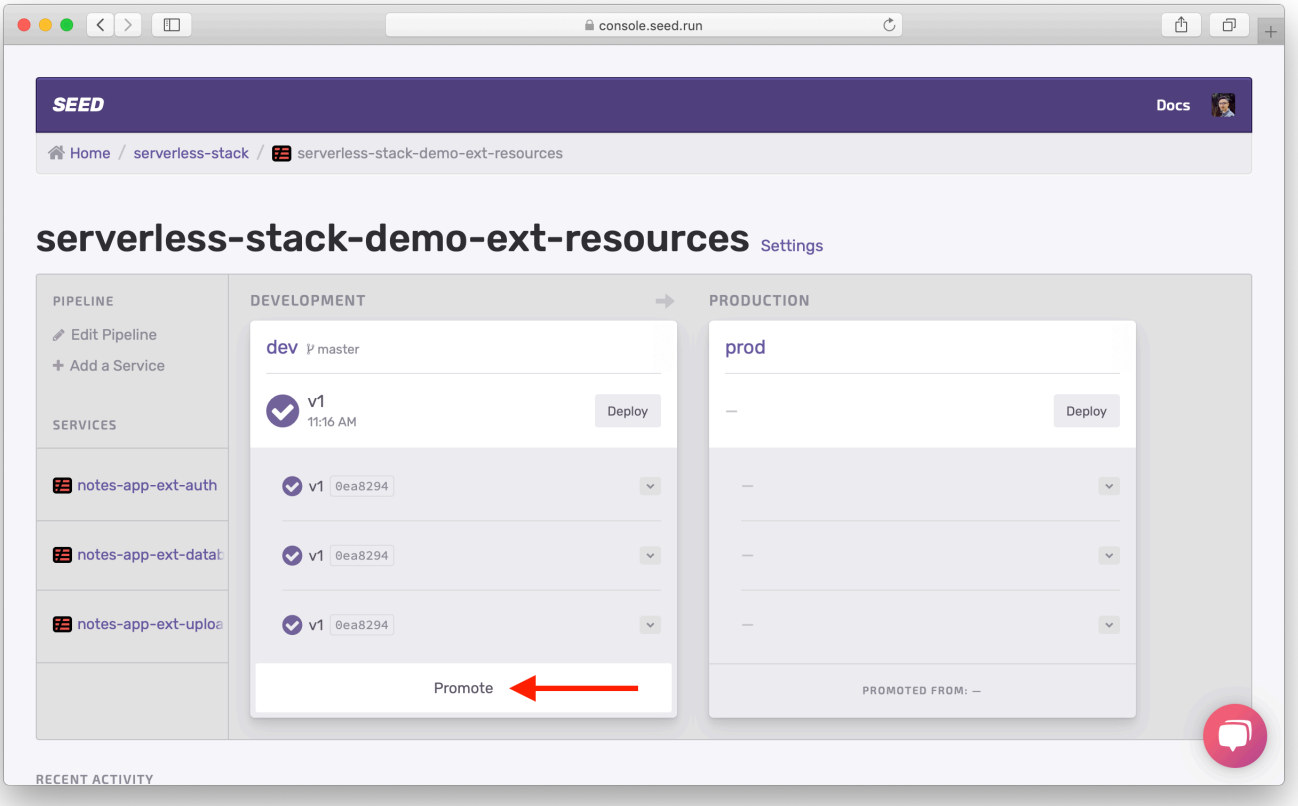
After all services are successfully deployed. Click the build **v1**.



You can see that the deployments are carried out in the order specified by the deploy phases.

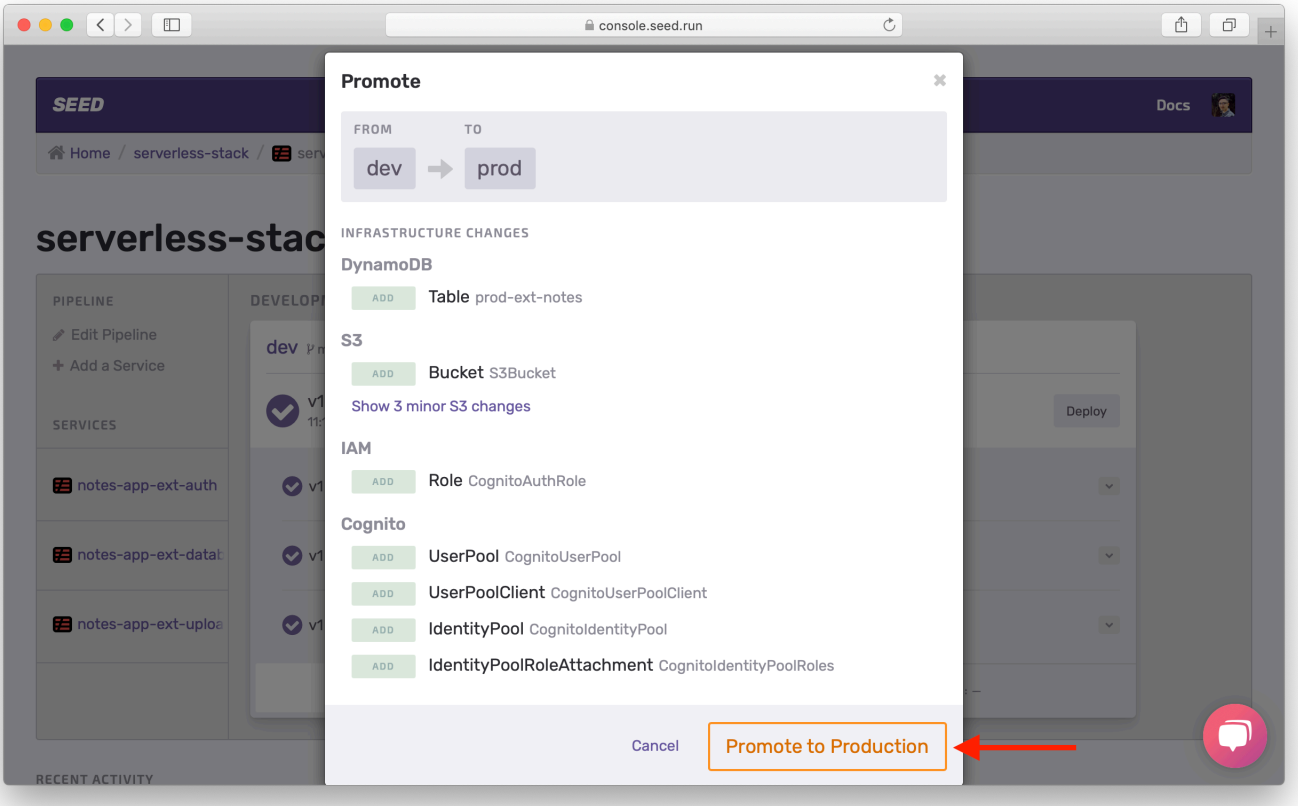


Go back to the app dashboard, and hit **Promote** to deploy this to the **prod** stage.

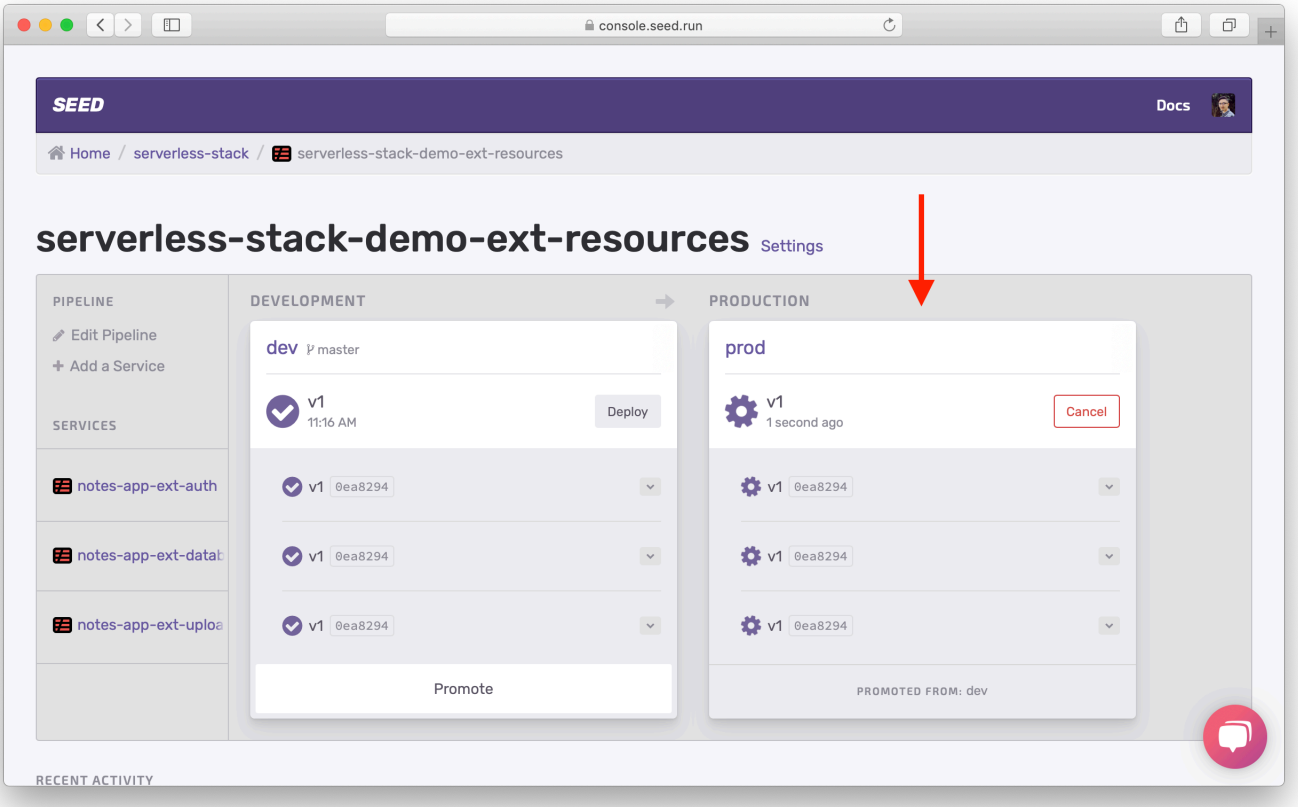


You will see a list of changes in resources. Since this is the first time we are deploying to the `prod` stage, the change list shows all the resources that will be created. We'll take a look at this in detail later in the Promoting to production (/chapters/promoting-to-production.html) chapter.

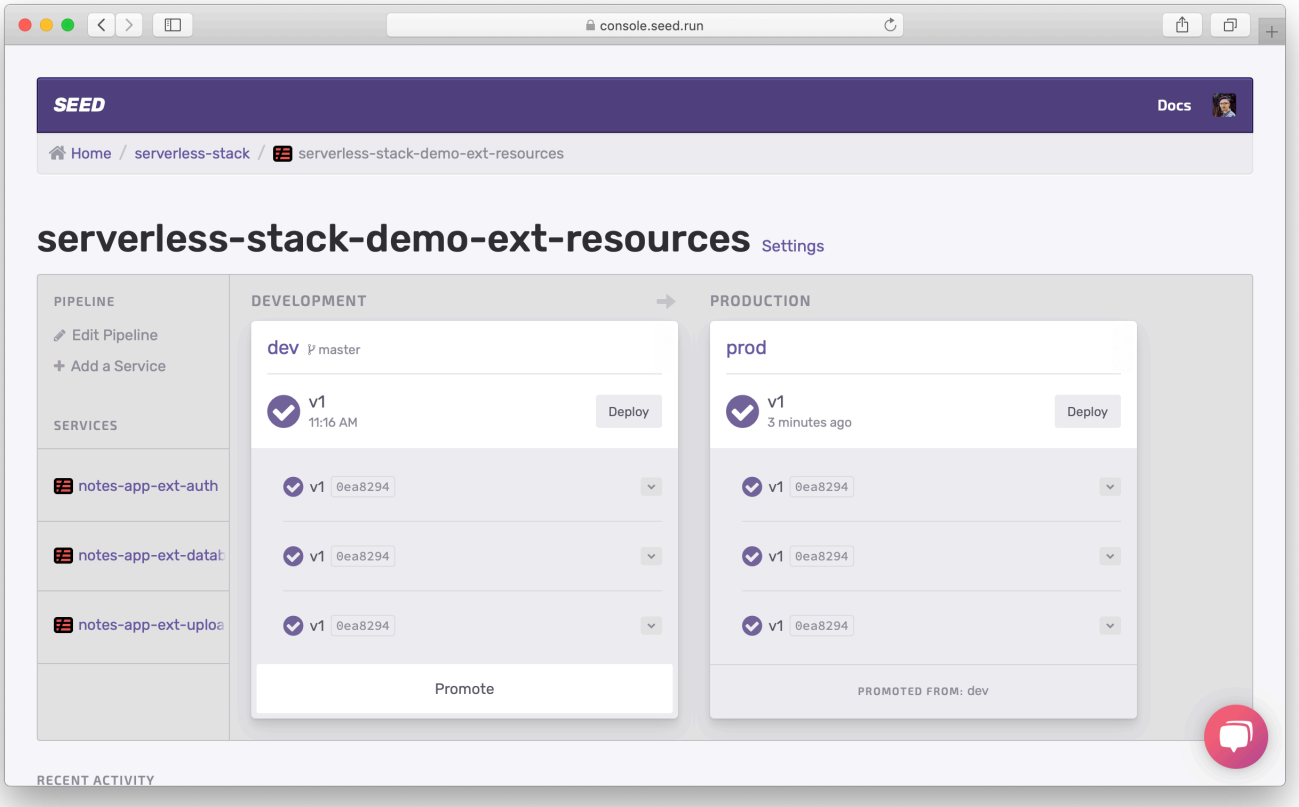
Click **Promote to Production**.



This will trigger the services to deploy in the same order we specified.



Now our resources have been deployed to both **dev** and **prod**.



Next, let's deploy our API services repo.

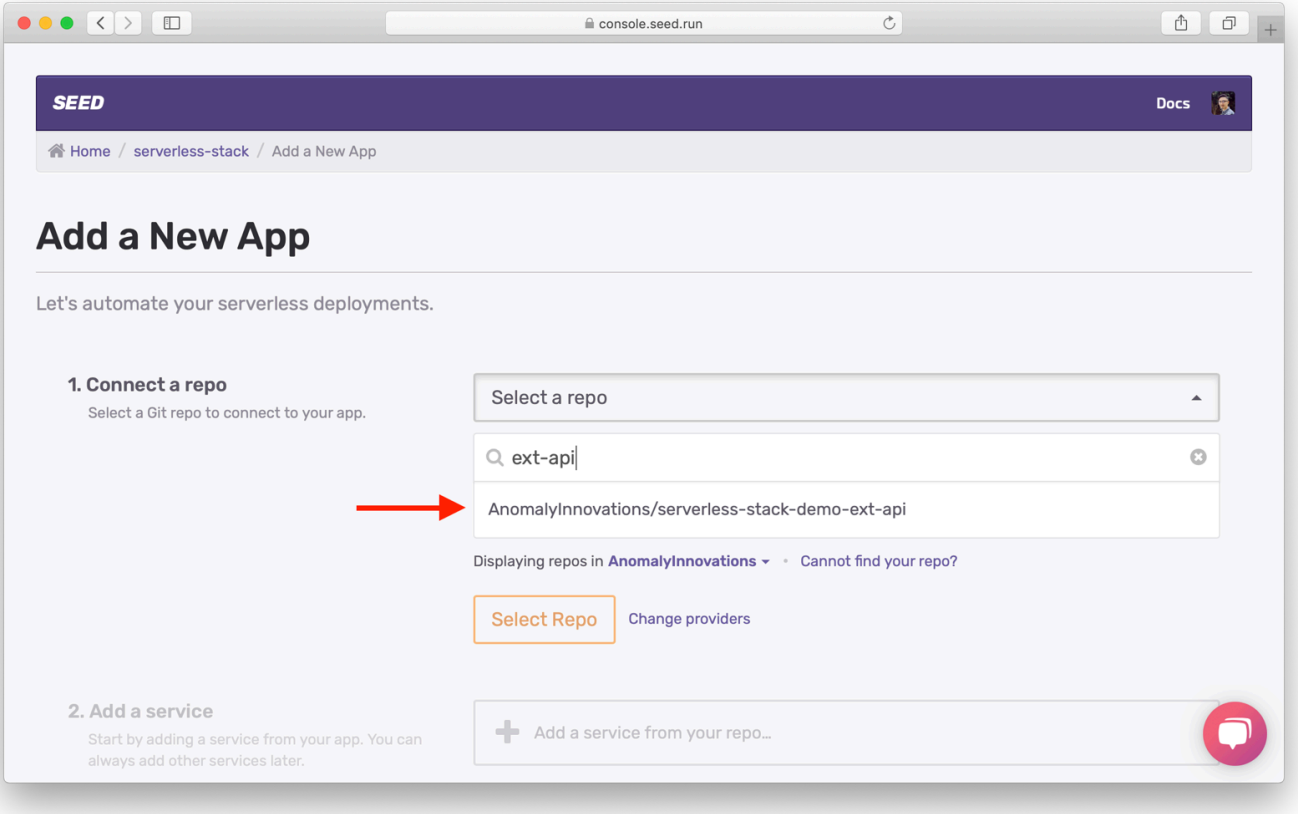
For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploy-the-resources-repo/1320)

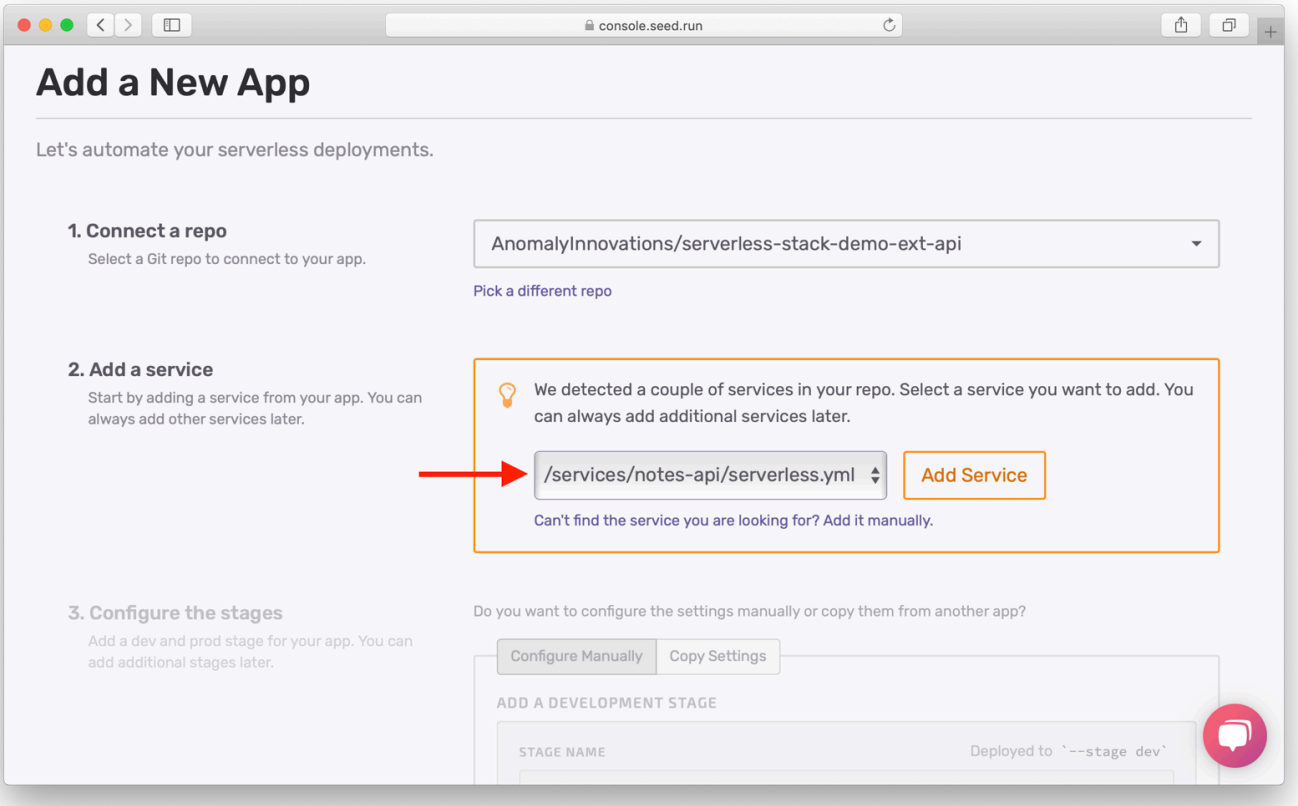
Deploy the API Services Repo

Just as the previous chapter we'll add the API repo on Seed and deploy it to our environments.

Click **Add an App** again, and select your Git provider. This time, select the API repo.

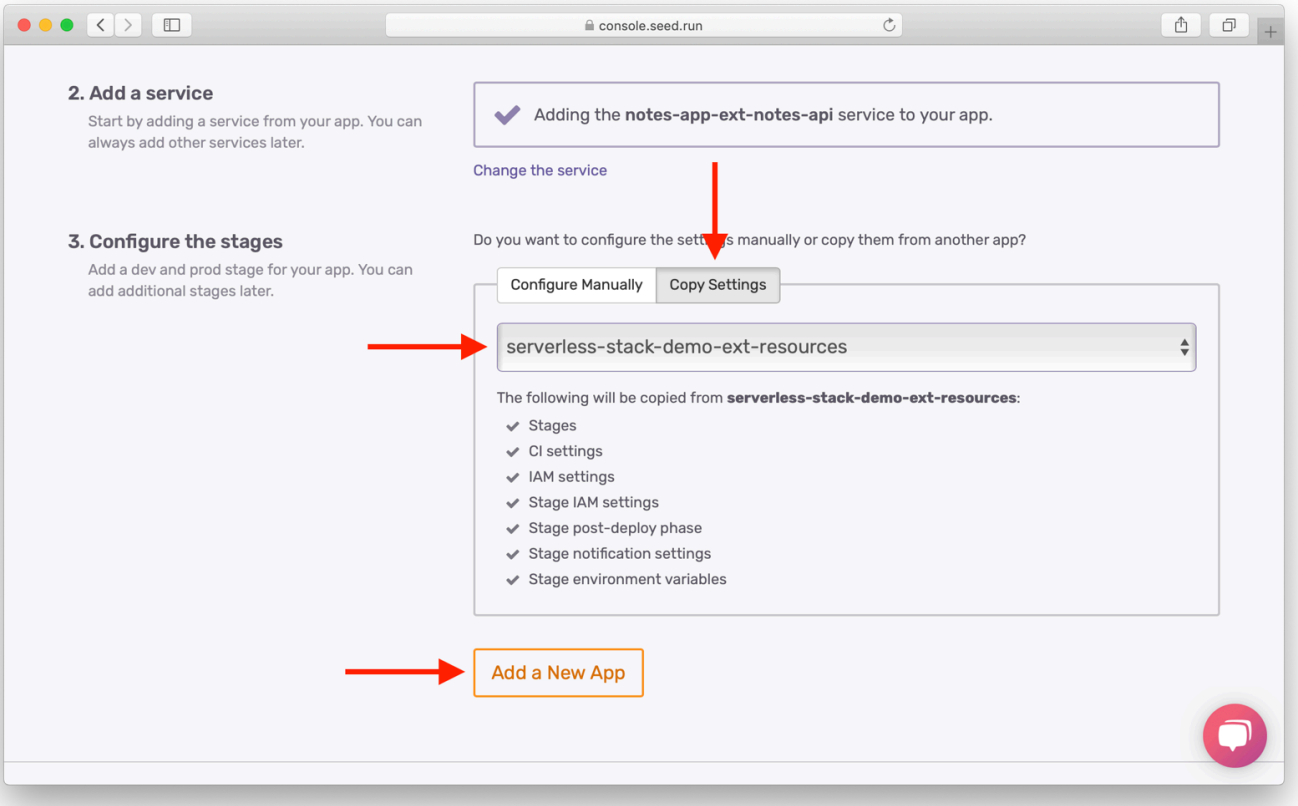


After detection, let's select the **notes-api** service.

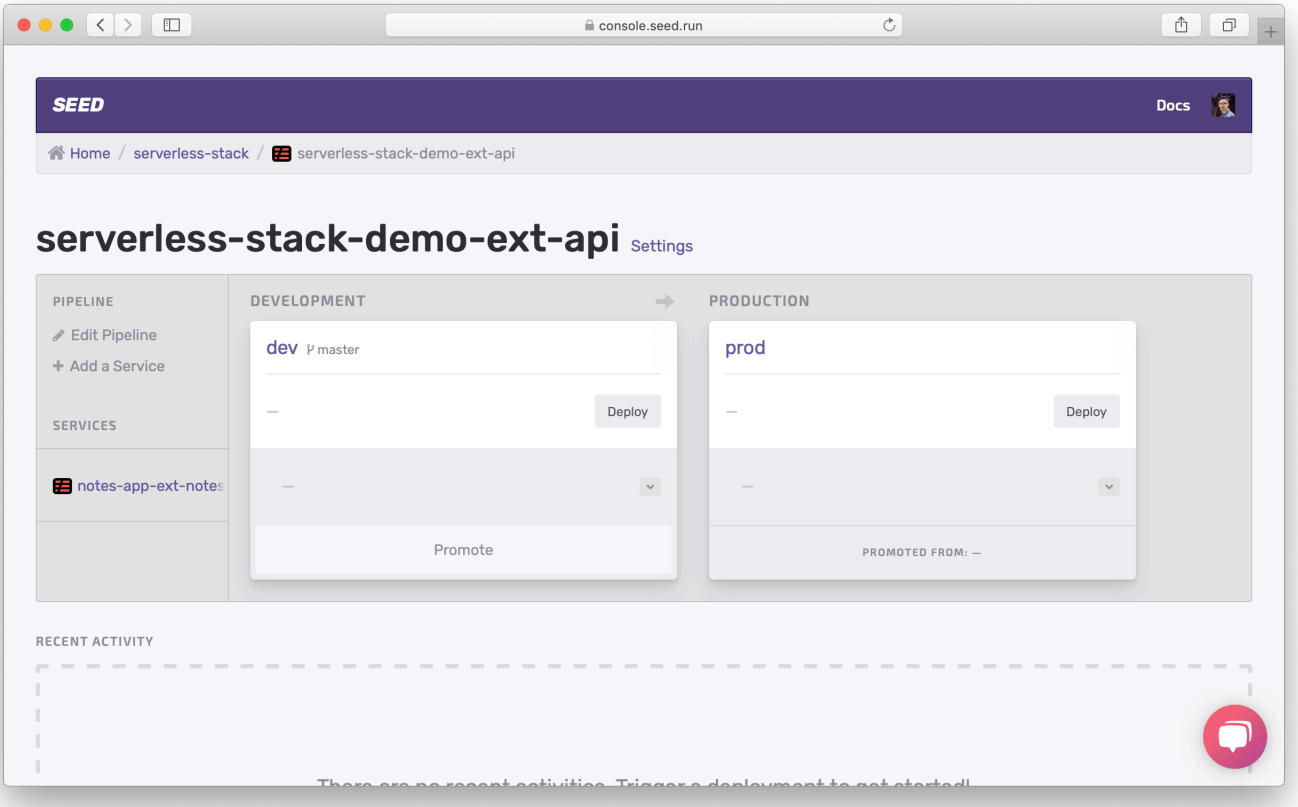


The environments for our API repo are identical to our resources repo. So instead of manually configuring them, we'll copy the settings.

Select **Copy Settings** tab, and select the resources app. Then hit **Add a New App**.



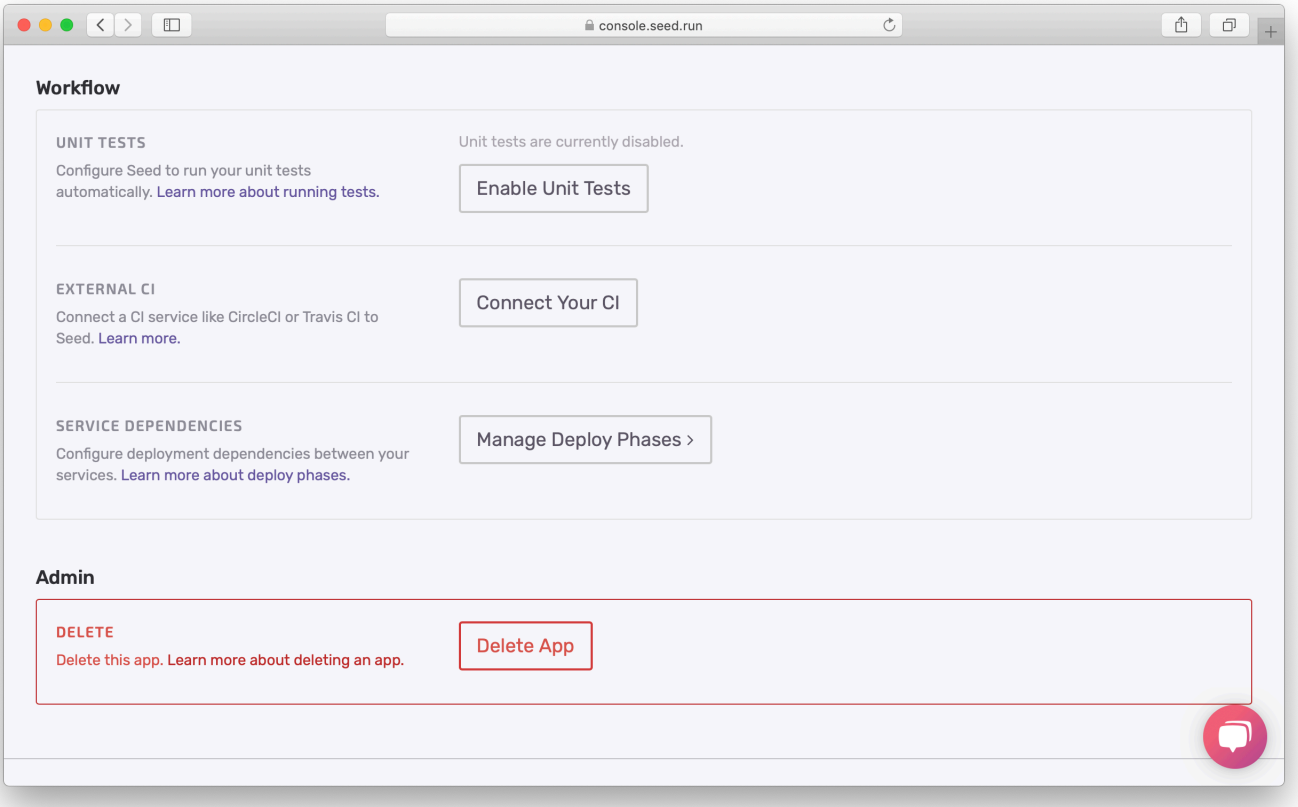
The API app has been created.



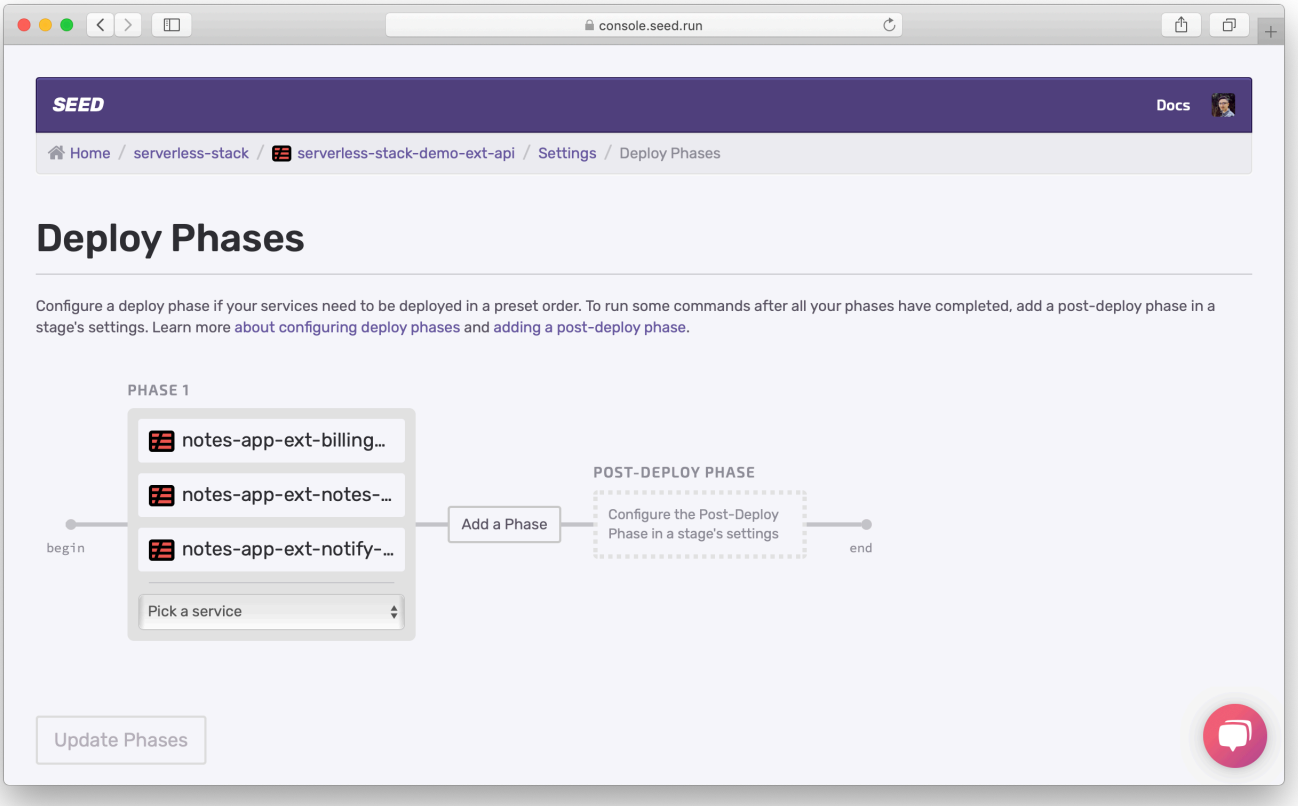
Click **Add a service** to add the **billing-api** service at the `services/billing-api` path. And then repeat the step to add the **notify-job** service at the `services/notify-job` path.

![Added all services in Seed (/assets/best-practices/deploy-api-services-repo-to-seed/[added-all-services-in-seed.png)

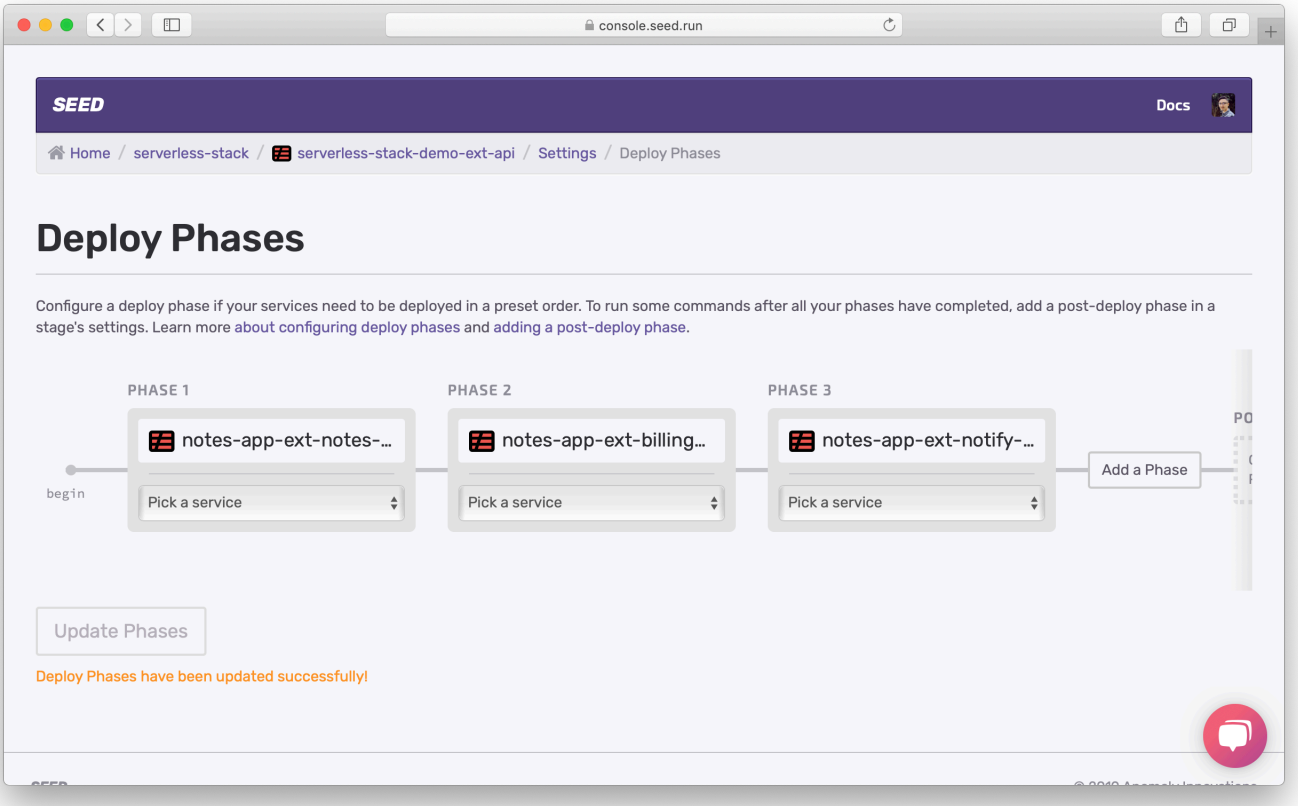
Head over to the app settings and click on **Manage Deploy Phases**.



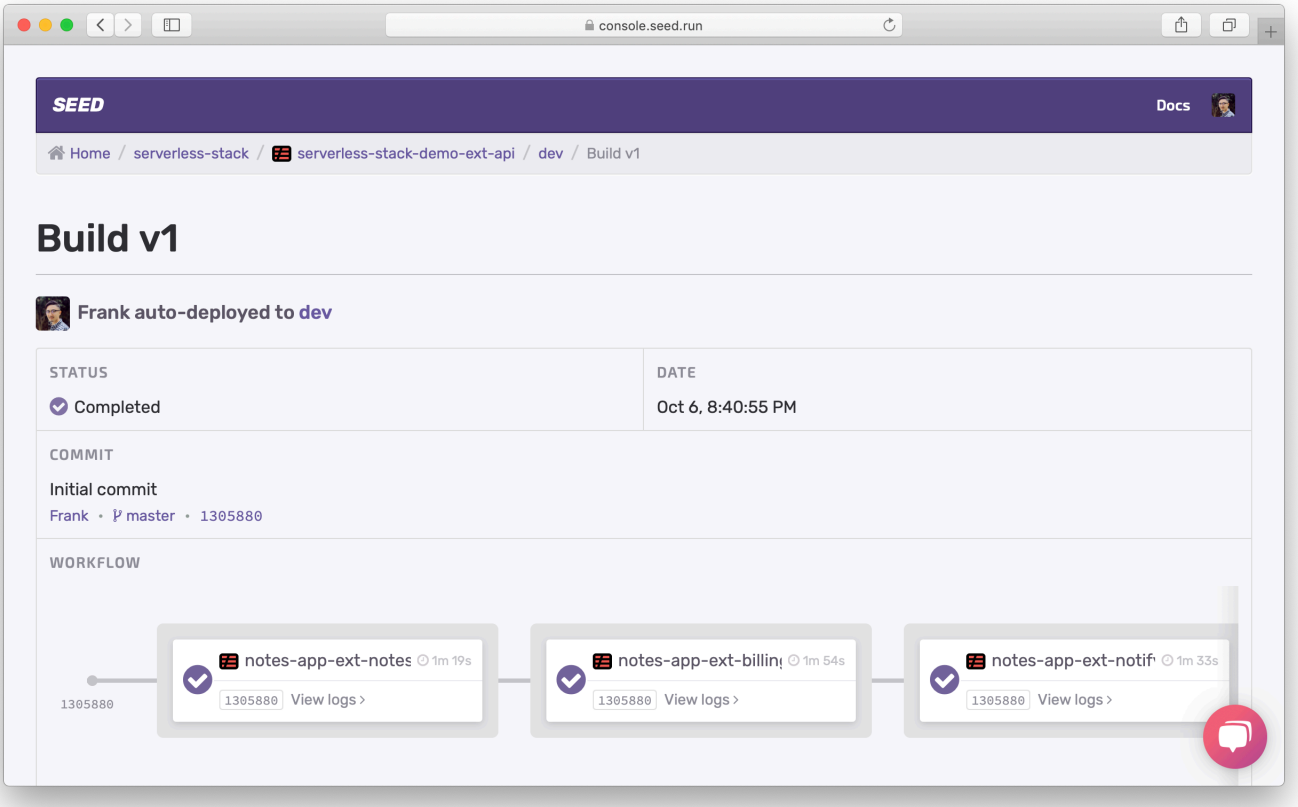
Again you'll notice that by default all the services are deployed concurrently.



Since the **billing-api** service depends on the **notes-api** service, and in turn the **notify-job** service depends on the **billing-api** service, we are going too add 2 phases. And move the **billing-api** service to **Phase 2**, and the **notify-job** service to **Phase 3**. Finally, click **Update Phases**.

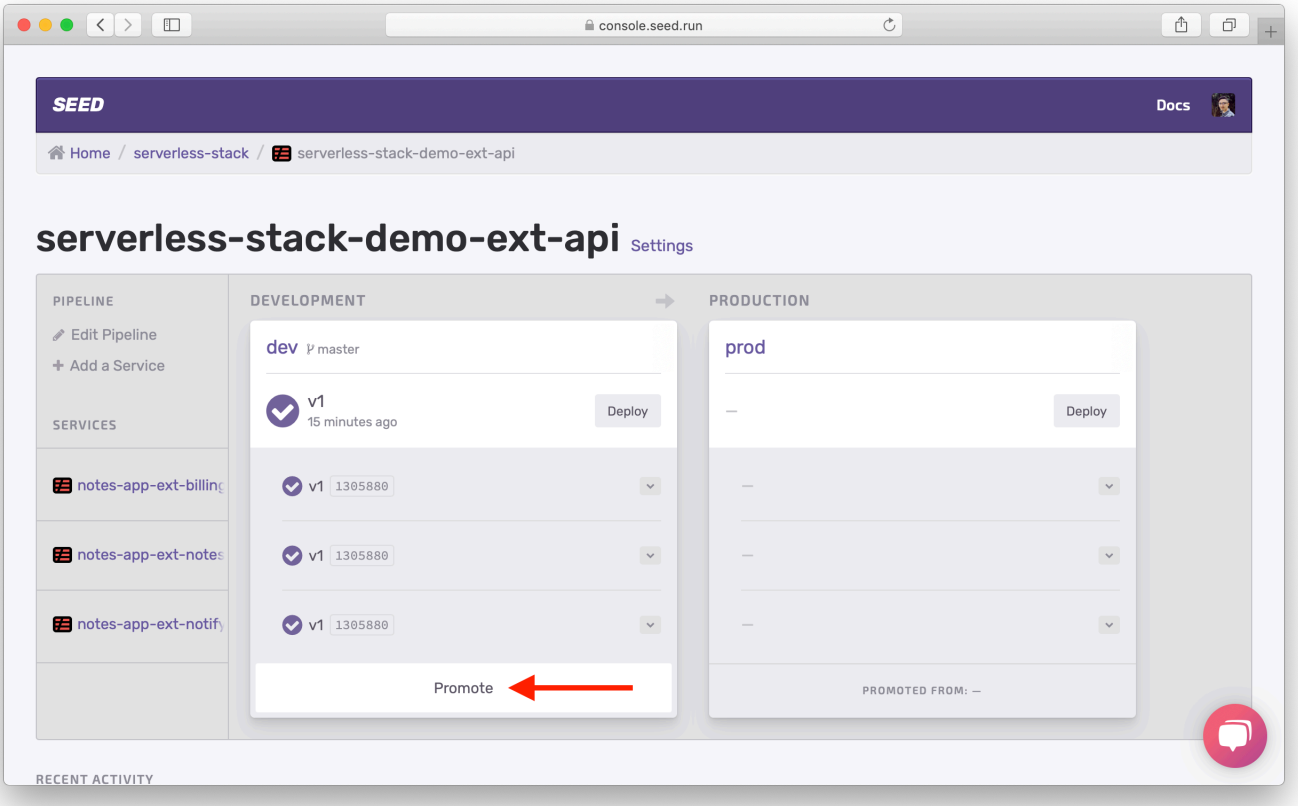


Now let's make our first deployment.

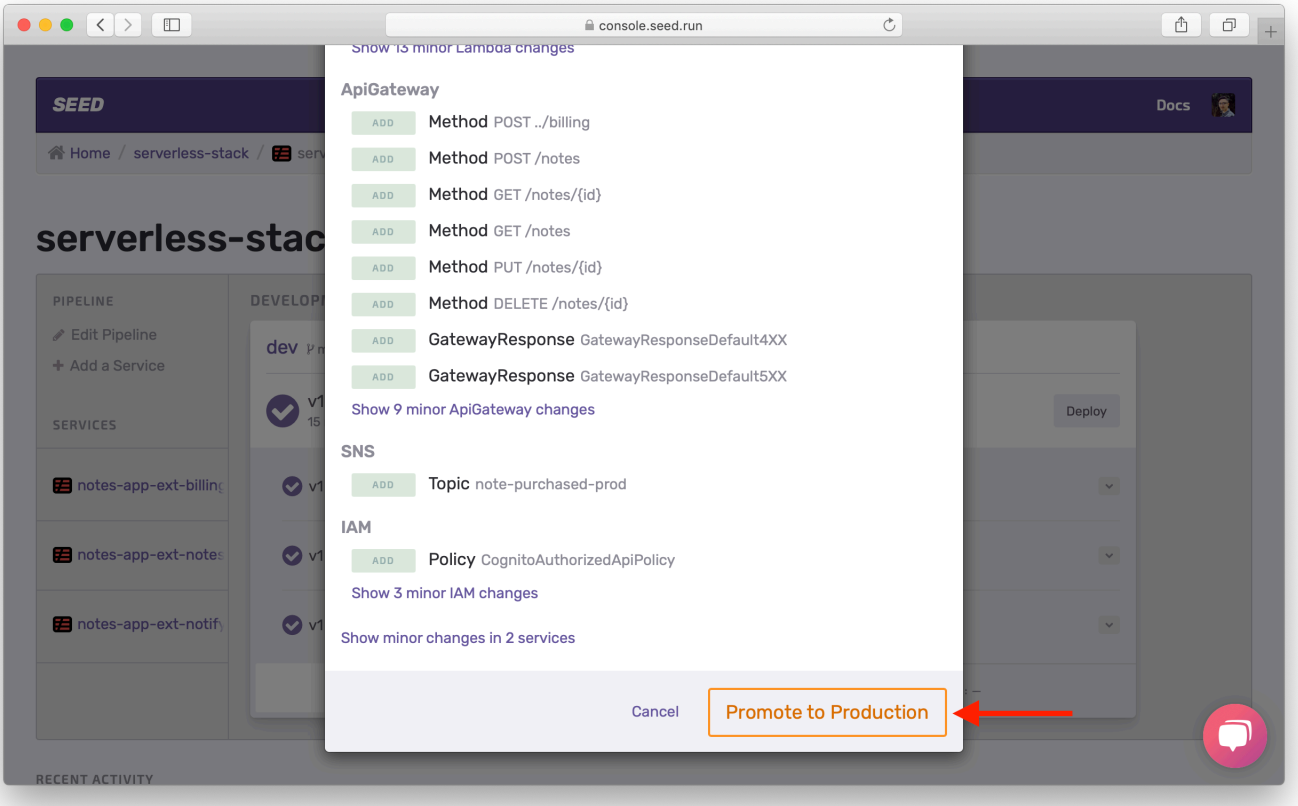


You can see the deployments are carried out according to the deploy phases specified.

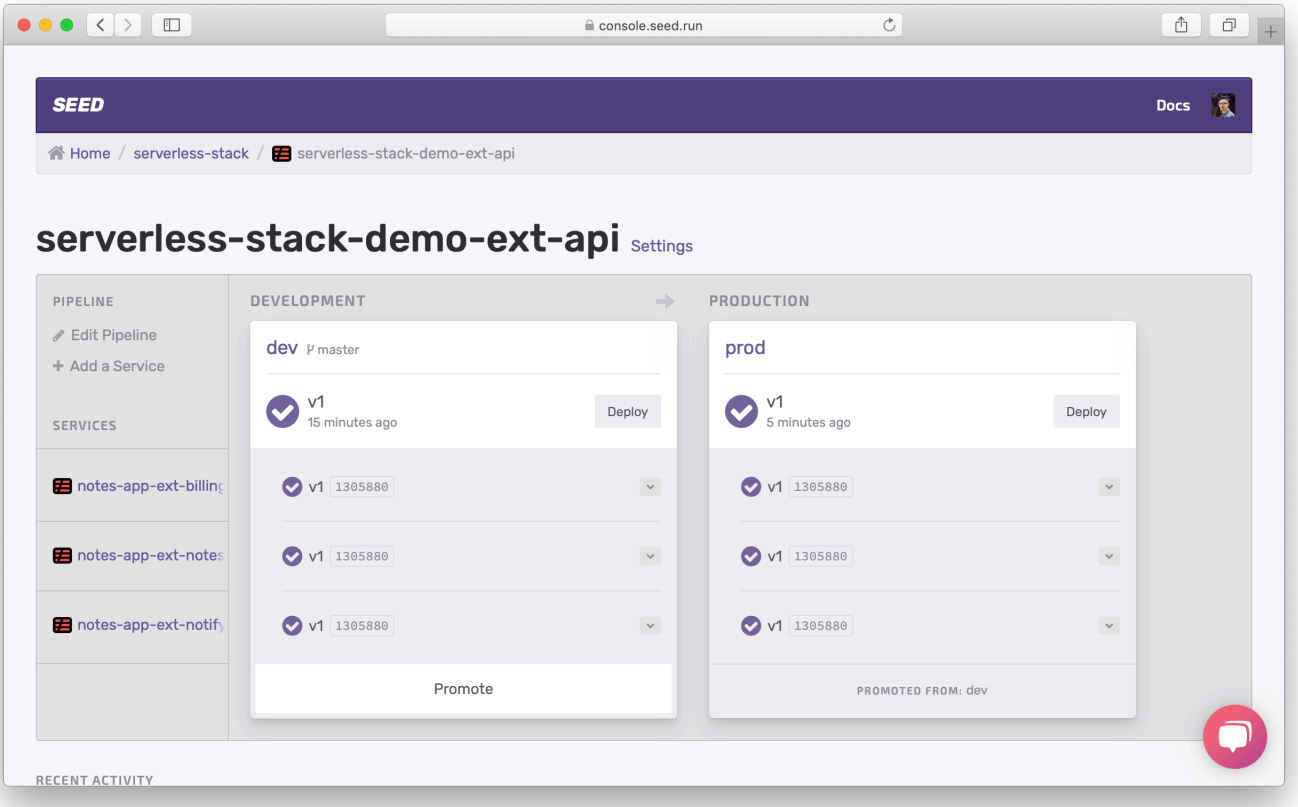
Just as before, promote **dev** to **prod**.



Hit **Promote to Production**.



Now we have the API deployed to both **dev** and **prod**.



Now that our entire app has been deployed, let's look at how we are sharing environment specific configs across our services.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploy-the-api-services-repo/1319)

Manage Environment Related Config

In this chapter we'll look at how our services will connect to each other while they are deployed across multiple environments.

Let's quickly review the setup that we've created back in the Organizing services chapter (/chapters/organizing-serverless-projects.html).

1. We have two repos — serverless-stack-demo-ext-resources (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources) and serverless-stack-demo-ext-api (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api). One has our infrastructure specific resources, while the other has all our Lambda functions.
2. The `serverless-stack-demo-ext-resources` repo is deployed a couple of long lived environments; like `dev` and `prod`.
3. While, the `serverless-stack-demo-ext-api` will be deployed to a few ephemeral environments (like `featureX` that is connected to the `dev` environment), in addition to the long lived environments above.

But before we can deploy to an ephemeral environment like `featureX`, we need to figure out a way to let our services know which infrastructure environment they need to talk to.

Let's look at how to do that.

Set a stage environment variable

In the `serverless.common.yml` file, we defined:

```
custom:
  # Our stage is based on what is passed in when
running serverless
  # commands. Or fallsback to what we have set in the
provider section.
  stage: ${opt:stage, self:provider.stage}
  resourcesStages:
    prod: prod
    dev: dev
  resourcesStage:
${self:custom.resourcesStages.${self:custom.stage},
self:custom.resourcesStages.dev}
```

The above code reads the current stage from the `serverless` commands, and selects the corresponding `resourcesStage` config.

- If the stage is `prod`, it uses the `prod` infrastructure.
- If the stage is `dev`, it uses the `dev` infrastructure.
- And if stage is `featureX`, it falls back to the dev config and uses the `dev` infrastructure.

And then in each service, we are going to pass the `resourcesStage` to the Lambda functions as an environment variable. Open up the `serverless.yml` file in a service.

```
...

custom: ${file(../../serverless.common.yml):custom}

provider:
  environment:
    stage: ${self:custom.stage}
    resourcesStage: ${self:custom.resourcesStage}
...
```

This adds a `resourcesStage` environment variable to all the Lambda functions in the service. Recall that we can access this via the `process.env.resourcesStage` variable at runtime.

Create a stage based config

Now in our `config.js`, we'll read the `resourcesStage` from the environment variable `process.env.resourcesStage`.

```
const stage = process.env.stage;
const resourcesStage = process.env.resourcesStage;
const adminPhoneNumber = "+14151234567";

const stageConfigs = {
  dev: {
    stripeKeyName: "/stripeSecretKey/test"
  },
  prod: {
    stripeKeyName: "/stripeSecretKey/live"
  }
};
```

```
const config = stageConfigs[stage] || stageConfigs.dev;

export default {
  stage,
  resourcesStage,
  adminPhoneNumber,
  ...config
};
```

Finally, while calling DynamoDB we can use the config to get the DynamoDB table we want to use. In `libs/dynamodb-lib.js`:

```
import AWS from "./aws-sdk";
import config from "../config";

const dynamoDb = new AWS.DynamoDB.DocumentClient();

export function call(action, params) {
  // Parameterize table names with stage name
  return dynamoDb[action]({
    ...params,
    TableName: `${config.resourcesStage}-
${params.TableName}`
  }).promise();
}
```

The above setup ensures that even when we create numerous ephemeral environments for our API services, they'll always connect back to the `dev` environment of our resources.

Next, let's look at how to store secrets across our environments.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/manage-environment-related-config/1327)

Storing Secrets in Serverless Apps

The general idea behind secrets is to store them outside of your codebase — don't commit them to Git! And to make them available at runtime. There are many ways people tend to do this, some are less secure than others. This chapter is going to layout the best practice for storing secrets and managing them across multiple environments.

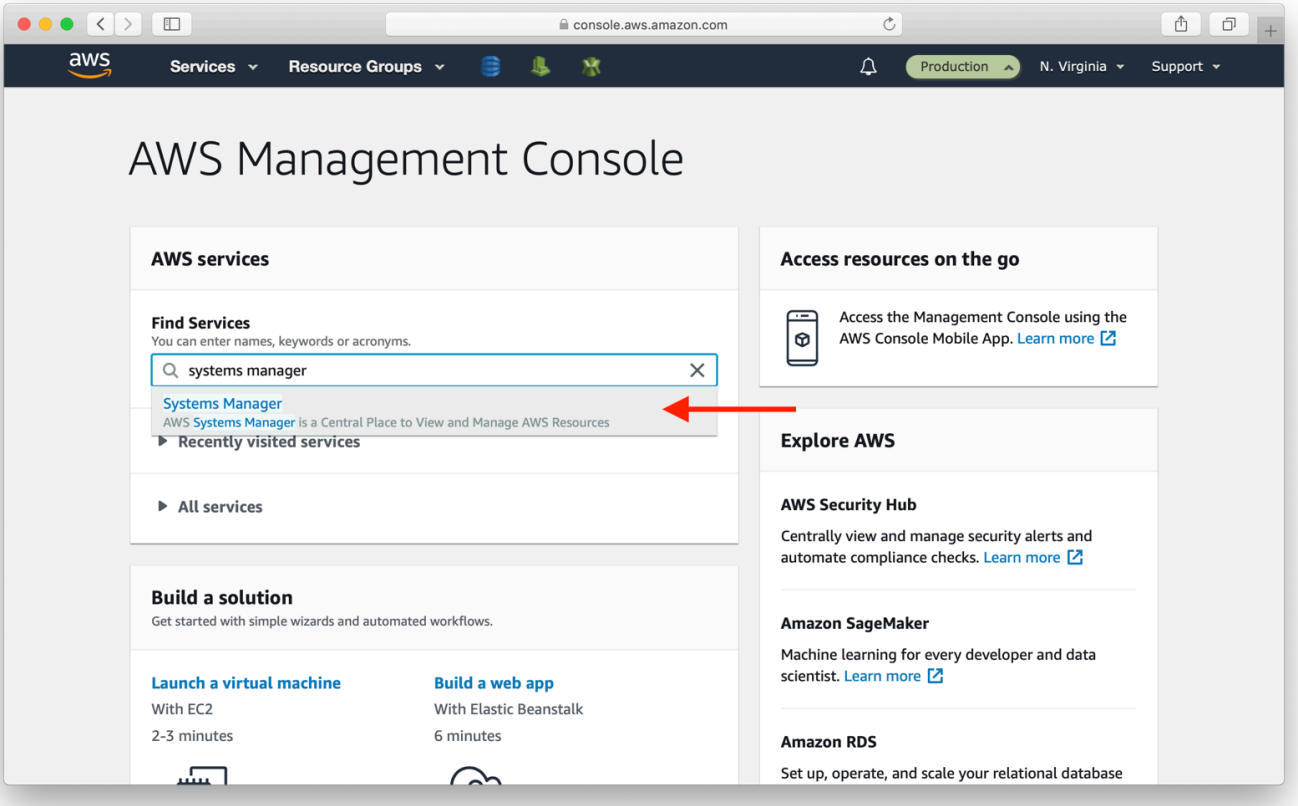
Store Secrets in AWS Parameter Store

AWS Systems Manager Parameter Store (https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-parameter-store.html) (SSM) is an AWS service that lets you store configuration data and secrets as key-value pairs in a central place. The values can be stored as plain text or as encrypted data. When stored as encrypted data, the value is encrypted on write using your AWS KMS key (https://aws.amazon.com/kms/), and decrypted on read.

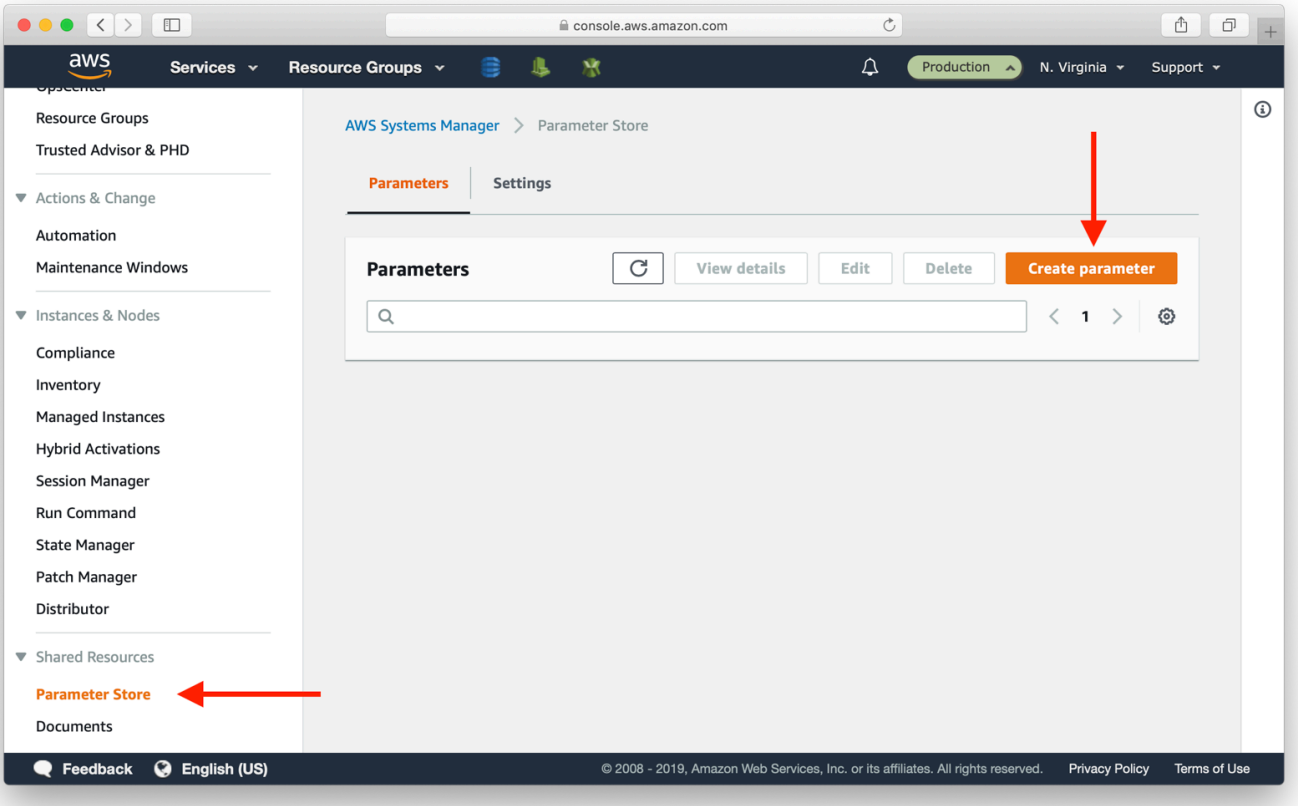
As an example, we are going to use SSM to store our Stripe secret key. Note that, Stripe gives us 2 keys: a **live** key and a **test** key. We are going to store:

- The **live** key in the Production account's SSM console.
- The **test** key in the Development account's SSM console.

First go in to your **Production** account, and go to your Systems Manager console.

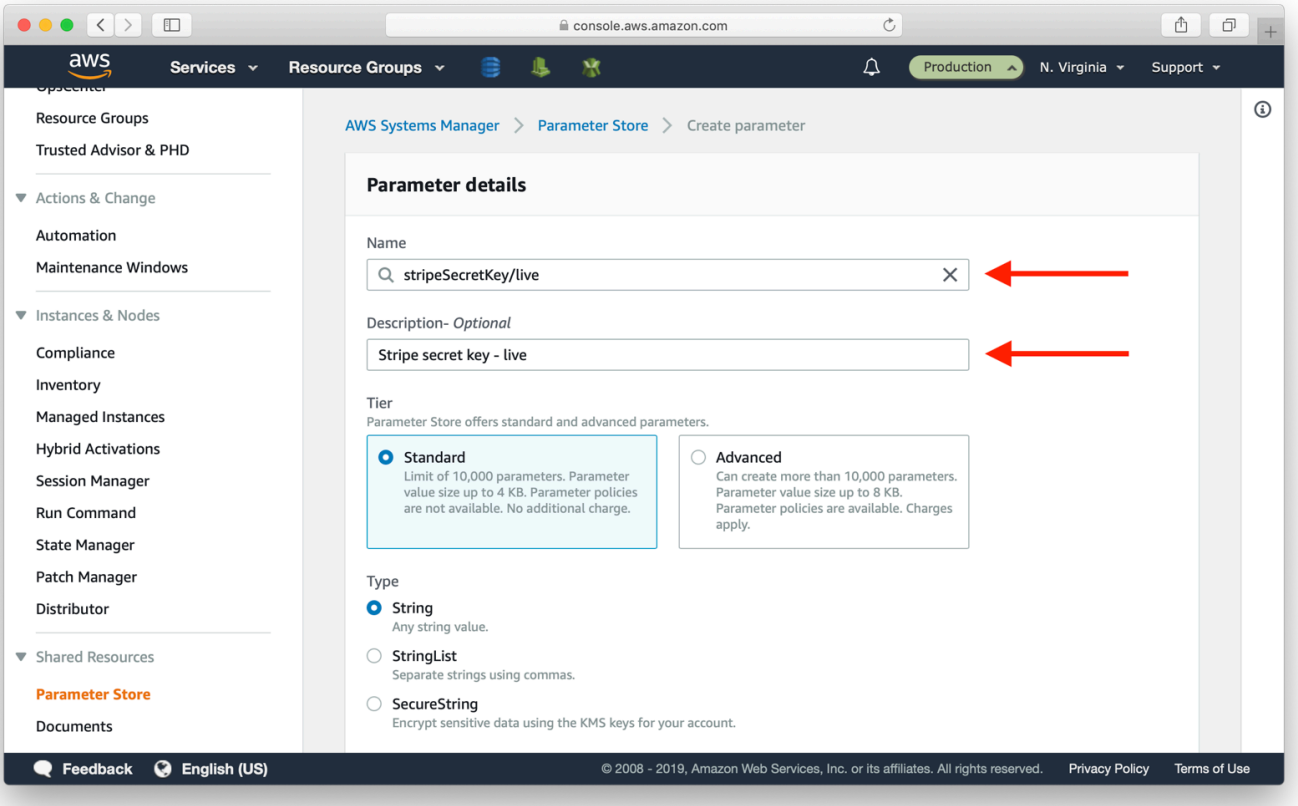


Select **Parameter Store** from the left menu, and select **Create parameter**.

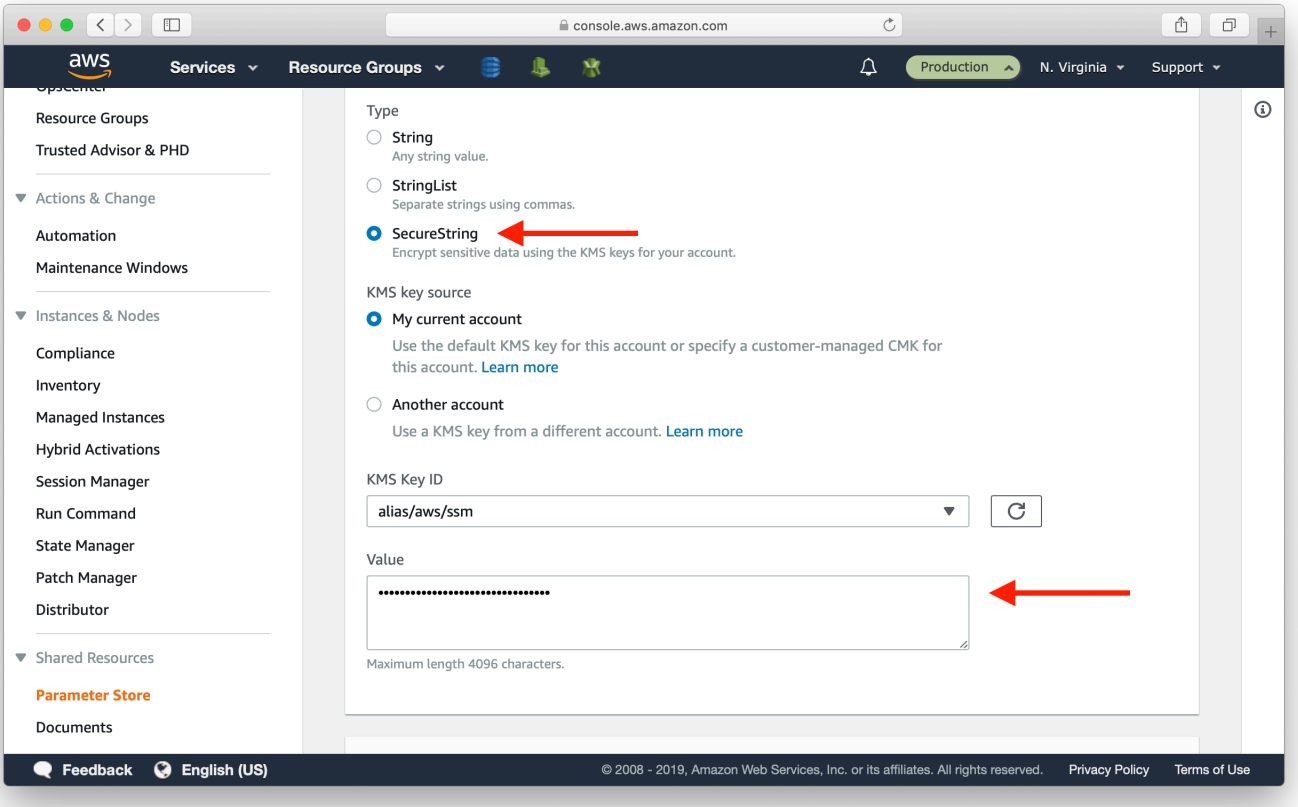


Fill in:

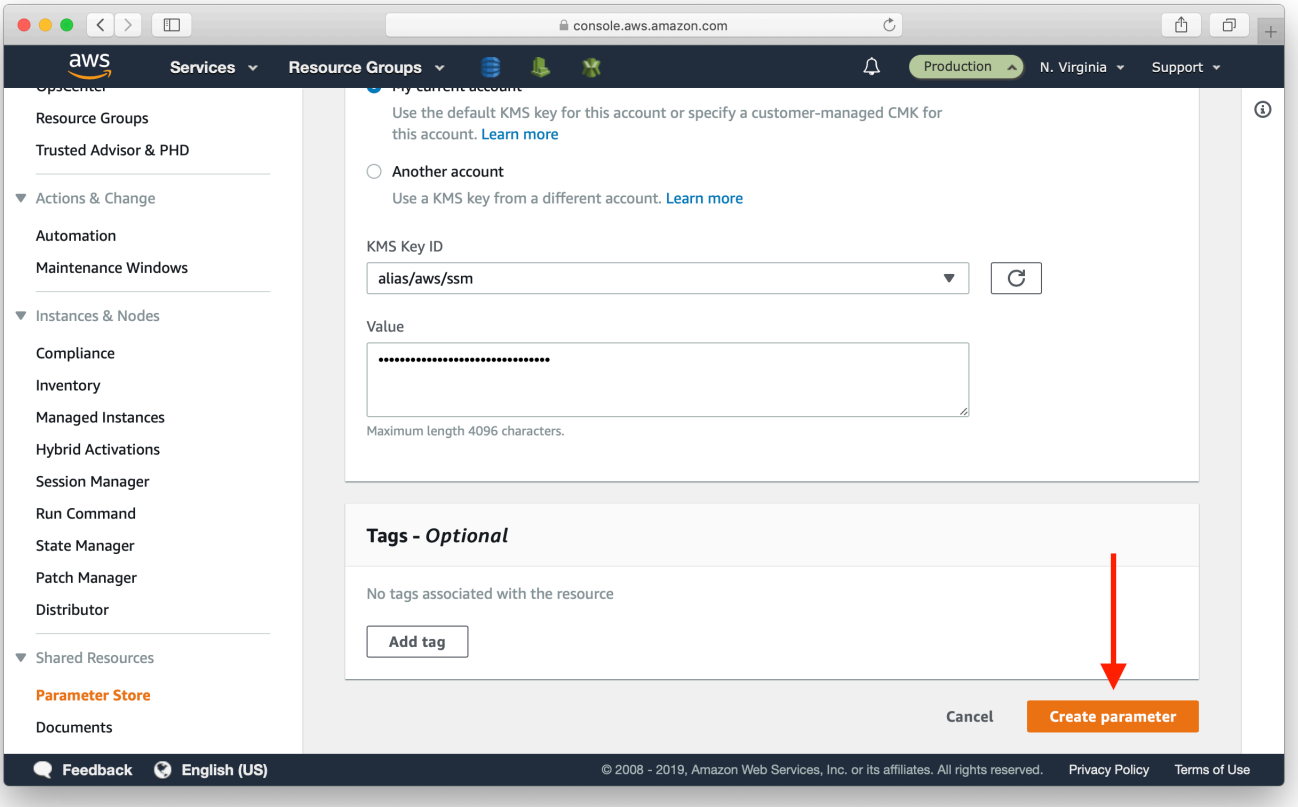
- **Name**: /stripeSecretKey/live
- **Description**: Stripe secret key - live



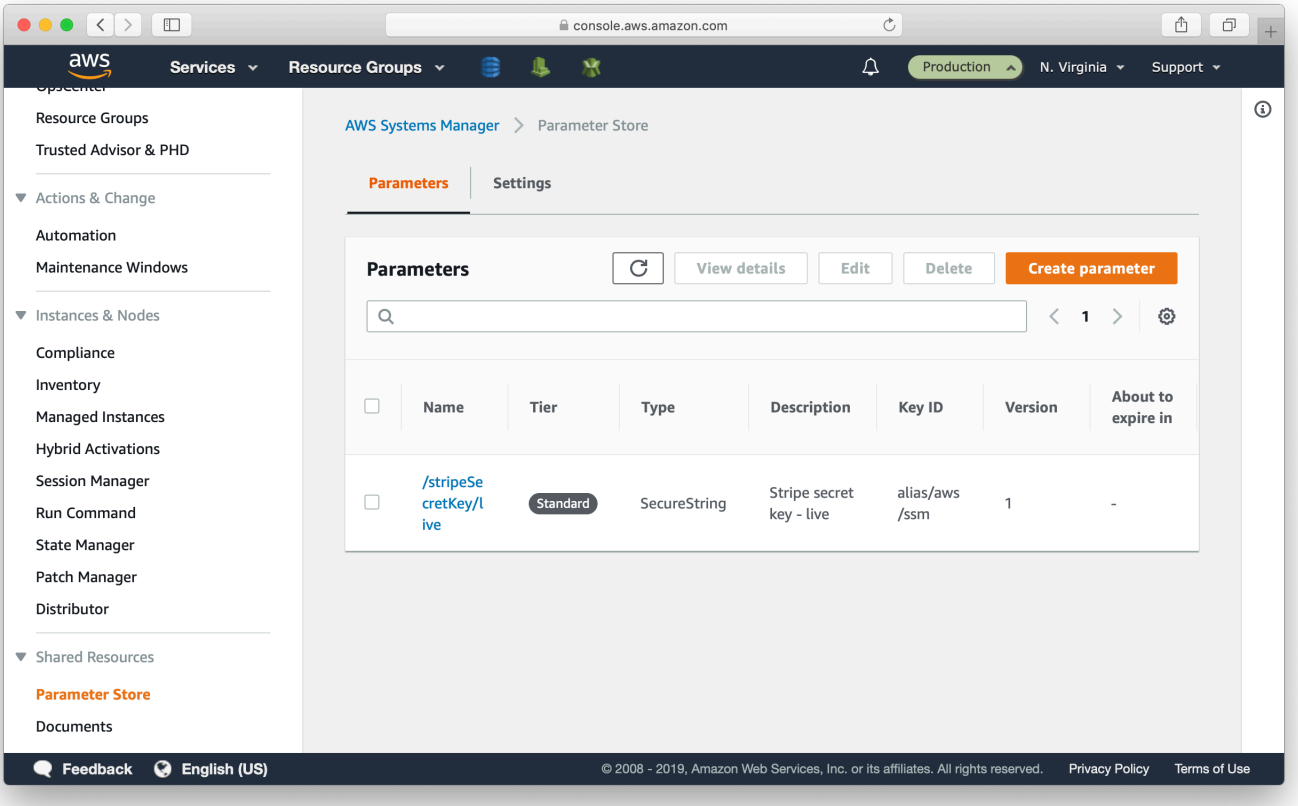
Select **SecureString**, and paste your live Stripe key in **Value**.



Scroll to the bottom and hit **Create parameter**.

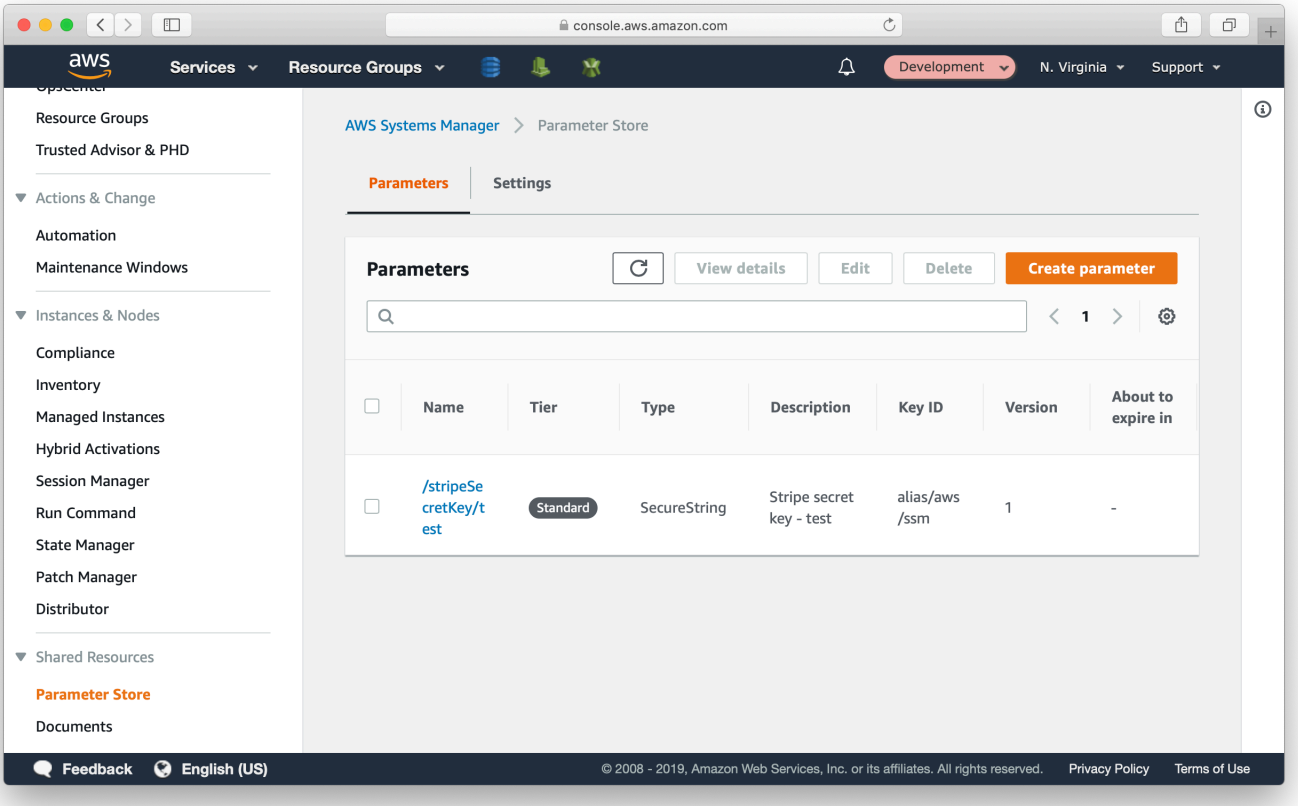


The key is added.



Then, switch to your **Development** account, and repeat the steps to add the **test** Stripe key with:

- **Name**: /stripeSecretKey/test
- **Description**: Stripe secret key - test



Access SSM Parameter in Lambda

Now to use our SSM parameters, we need to let Lambda function know which environment it is running in. We are going to pass the name of the stage to Lambda functions as environment variables.

Update our `serverless.yml`.

```
...


custom: ${file(../../serverless.common.yml):custom}

provider:
  environment:
    stage: ${self:custom.stage}
    resourcesStage: ${self:custom.resourcesStage}
```

```
    notePurchasedTopicArn:
      Ref: NotePurchasedTopic

  iamRoleStatements:
    -
${file(../../serverless.common.yml):lambdaPolicyXRay}
    - Effect: Allow
      Action:
        - ssm:GetParameter
      Resource:
        Fn::Join:
          - ''
          -
            - 'arn:aws:ssm:'
            - Ref: AWS::Region
            - ':'
            - Ref: AWS::AccountId
            - ':parameter/stripeSecretKey/*'
...
```

We are granting Lambda functions permission to fetch and decrypt the SSM parameters.

Next we'll add the parameter names in our `config.js`.

```
const stage = process.env.stage;
const resourcesStage = process.env.resourcesStage;
const adminPhoneNumber = "+14151234567";

const stageConfigs = {
  dev: {
    stripeKeyName: "/stripeSecretKey/test"
```

```
  },
  prod: {
    stripeKeyName: "/stripeSecretKey/live"
  }
};

const config = stageConfigs[stage] || stageConfigs.dev;

export default {
  stage,
  resourcesStage,
  adminPhoneNumber,
  ...config
};
```

The above code reads the current stage from the environment variable `process.env.stage`, and selects the corresponding config.

- If the stage is `prod`, it exports `stageConfigs.prod`.
- If the stage is `dev`, it exports `stageConfigs.dev`.
- And if stage is `featureX`, it falls back to the dev config and exports `stageConfigs.dev`.

If you need a refresher on the structure of our config, refer to the Manage environment related config (/chapters/manage-environment-related-config.html).

Now we can access the SSM value in our Lambda function.

```
import AWS from '../../libs/aws-sdk';
import config from "../../config";
```

```
// Load our secret key from SSM
const ssm = new AWS.SSM();
const stripeSecretKeyPromise = ssm
  .getParameter({
    Name: config.stripeKeyName,
    WithDecryption: true
  })
  .promise();

export const handler = (event, context) => {
  ...

  // Charge via stripe
  const stripeSecretKey = await stripeSecretKeyPromise;
  const stripe =
stripePackage(stripeSecretKey.Parameter.Value);
  ...
};
```

By calling `ssm.getParameter` with `WithDecryption: true`, the value returned to you is decrypted and ready to be used.

Note that, you want to decrypt your secrets **outside** your Lambda function handler. This is because you only need to do this once per Lambda function invocation. Your secrets will get decrypted on the first invocation and you'll simply use the same value in subsequent invocations.

So in summary, you want to store your sensitive data in SSM. Store the SSM parameter name in your config. When the Lambda function runs, use the environment it's running in to figure out which SSM parameter to use. Finally, decrypt the secret outside your Lambda handler function.

Next, we'll look at how we can setup our API Gateway custom domains to work across environments.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/storing-secrets-in-serverless-apps/1335)

Share Route 53 Domains Across AWS Accounts

Our notes app has an API Gateway endpoint. In this chapter, we are going to look at how to set up custom domains for each of our environments. Recall that our environments are split across multiple AWS accounts.

We are going to setup the following custom domain scheme:

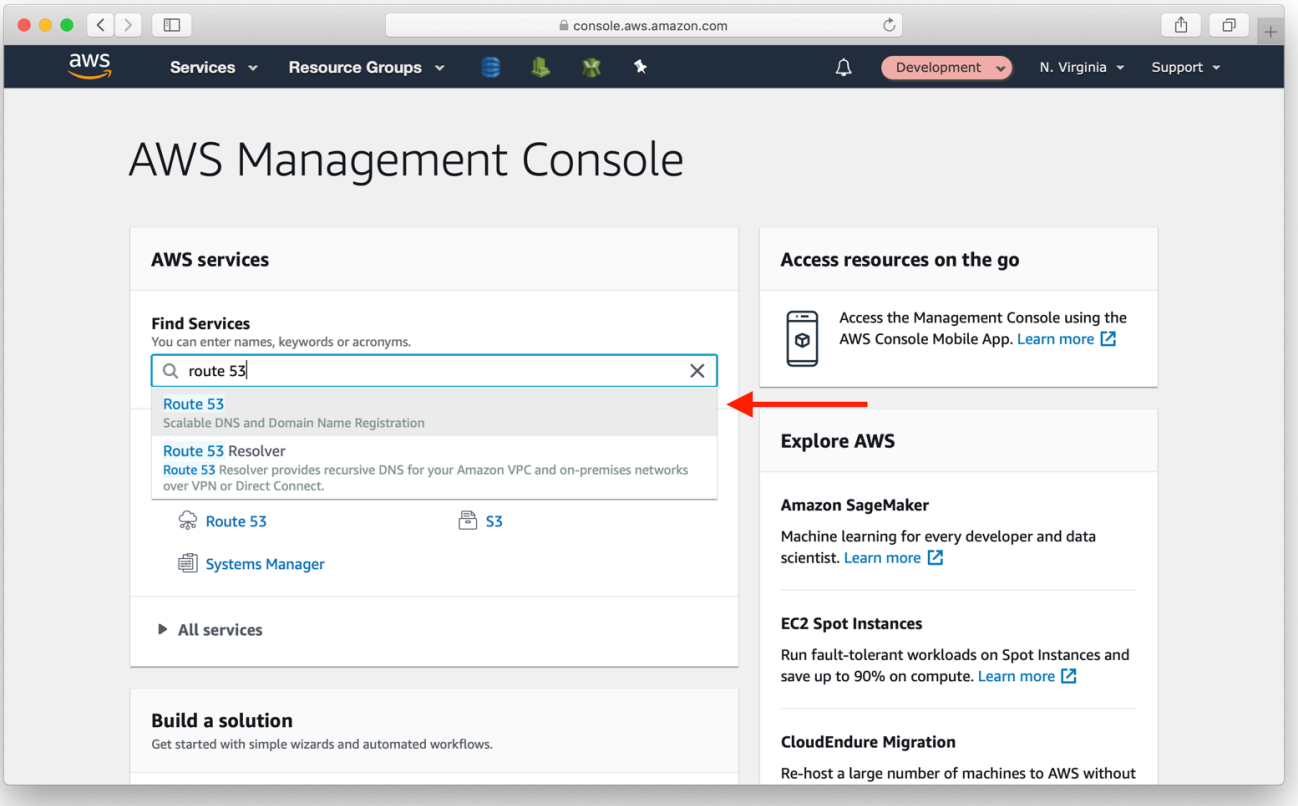
- `prod` ⇒ ext-api.serverless-stack.com
- `dev` ⇒ dev.ext-api.serverless-stack.com

Assuming that our domain is hosted in our `Production` AWS account. We want to set it up so that our `Development` AWS account can use the above subdomain. This takes an extra setup.

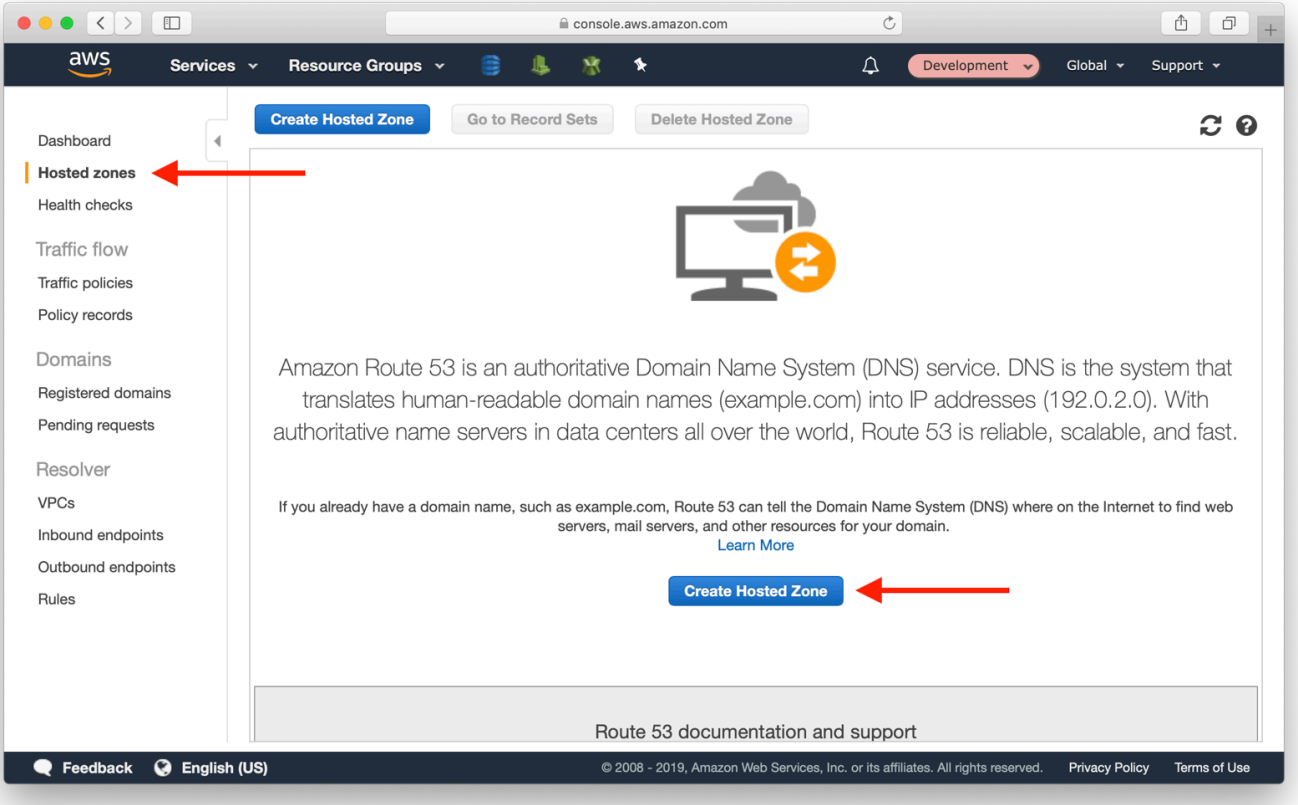
Delegate domains across AWS accounts

We are going to have to delegate the subdomain `dev.ext-api.serverless-stack.com` to be hosted in the `Development` AWS account. Just a quick note, as you follow these steps, pay attention to the account name shown at the top right corner of the screenshot. It'll tell you which account we are working with.

First, go into your Route 53 console in your `Development` account.



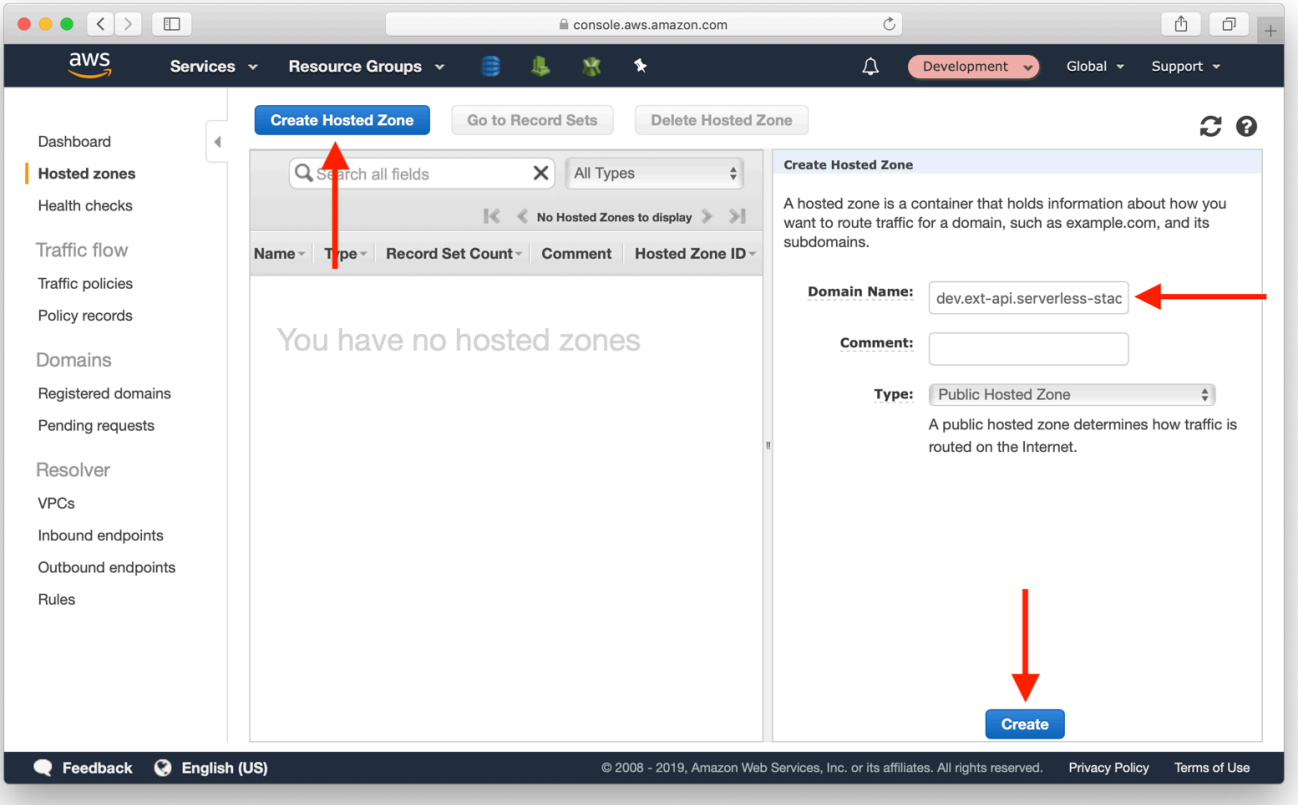
Click **Hosted zones** in the left menu. Then select **Create Hosted Zone**.



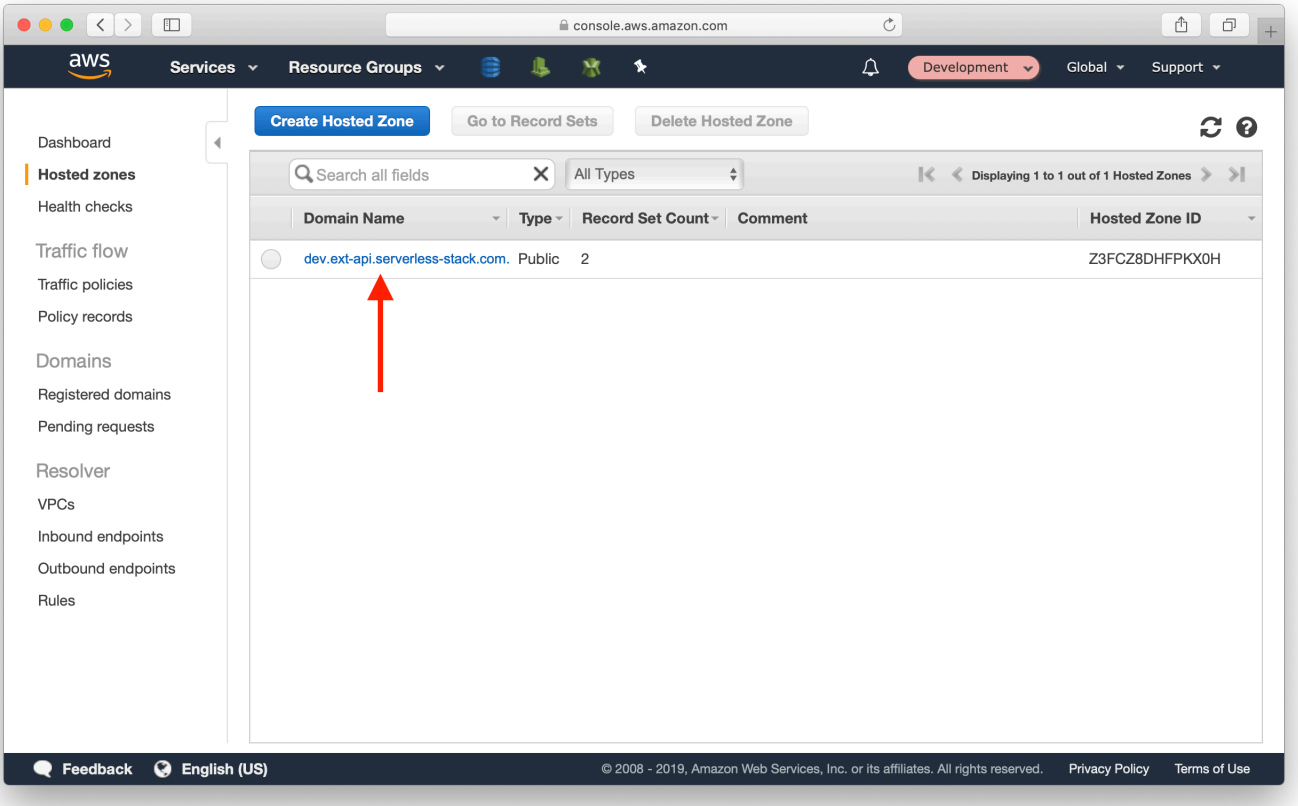
Select **Create Hosted Zone** at the top. Enter:

- **Domain Name**: dev.ext-api.serverless-stack.com

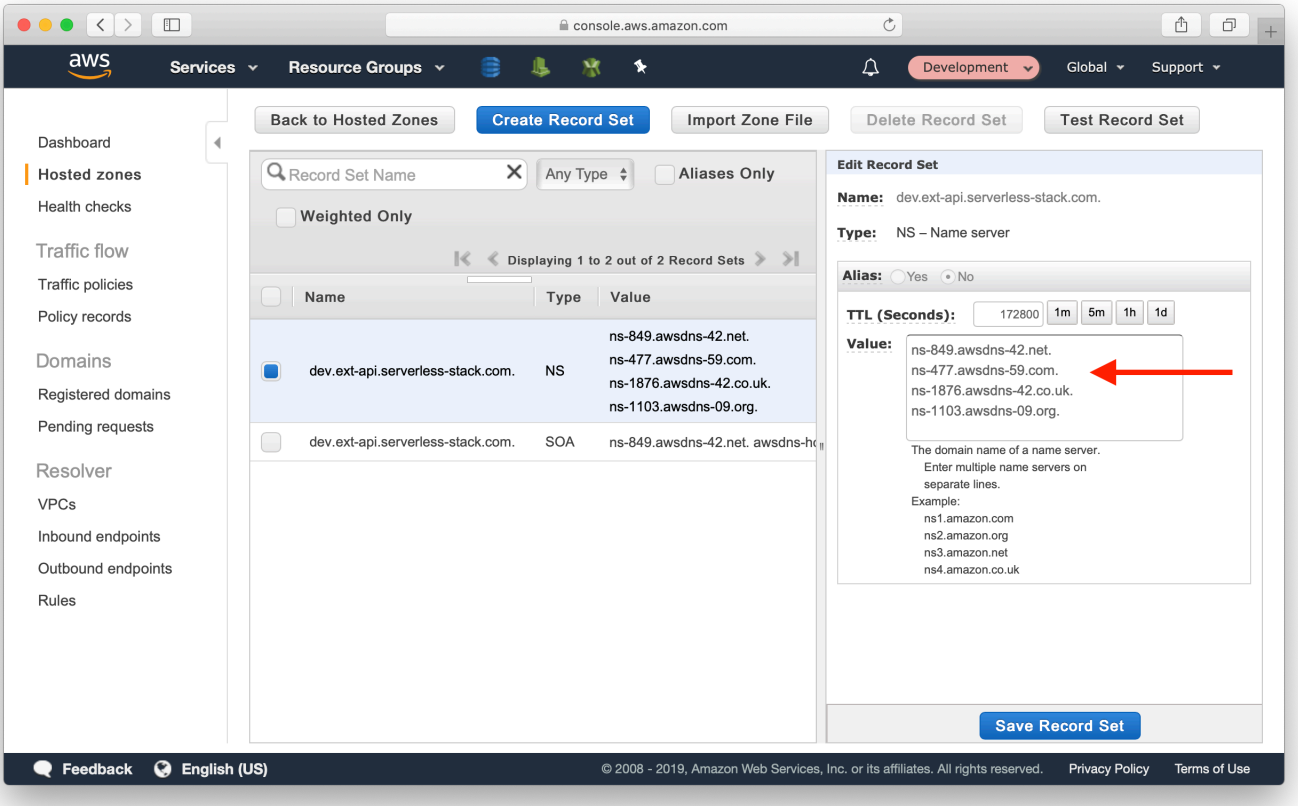
Then click **Create**.



Select the zone you just created.

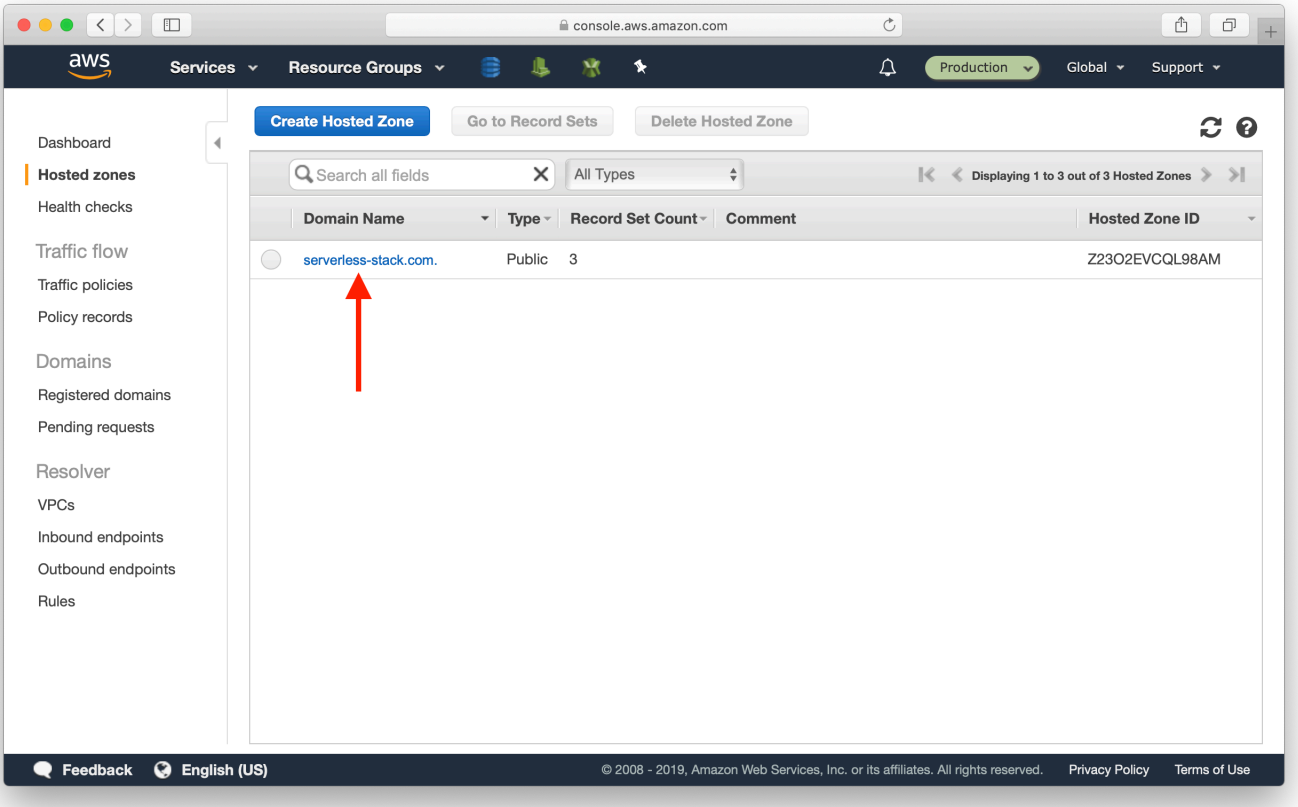


Click on the row with **NS** type. And copy the 4 lines in the **Value** field. We need this in the steps after.

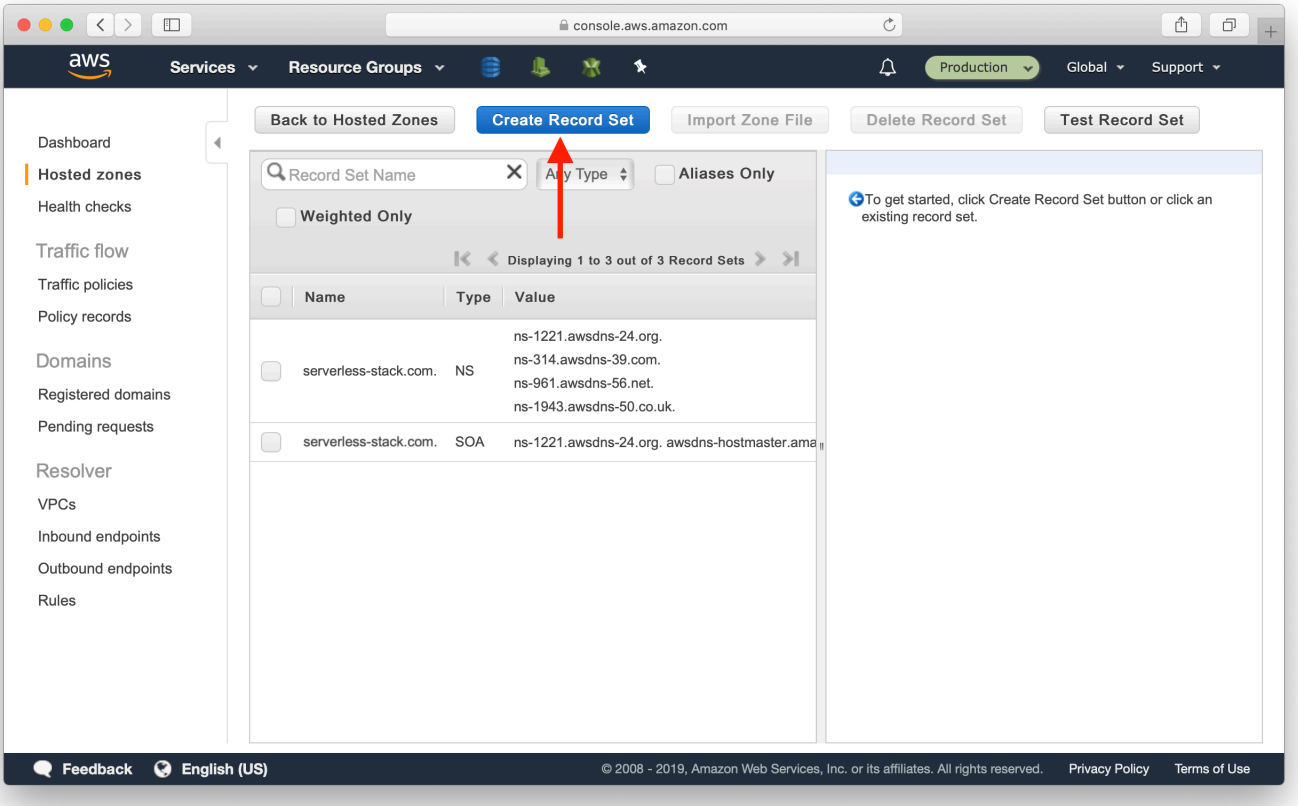


Now, switch to the `Production` account where the domain is hosted. And go into Route 53 console.

Select the domain.



Click **Create Record Set**.

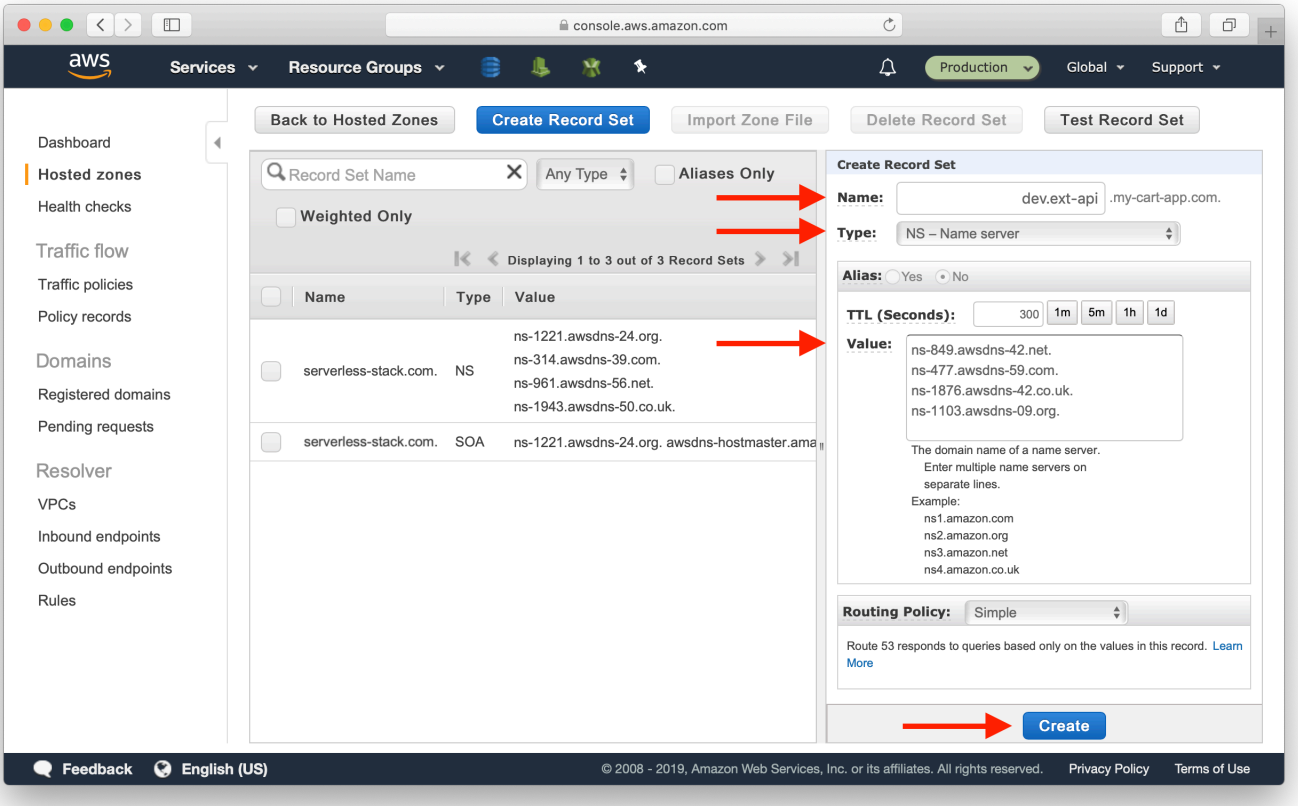


Fill in:

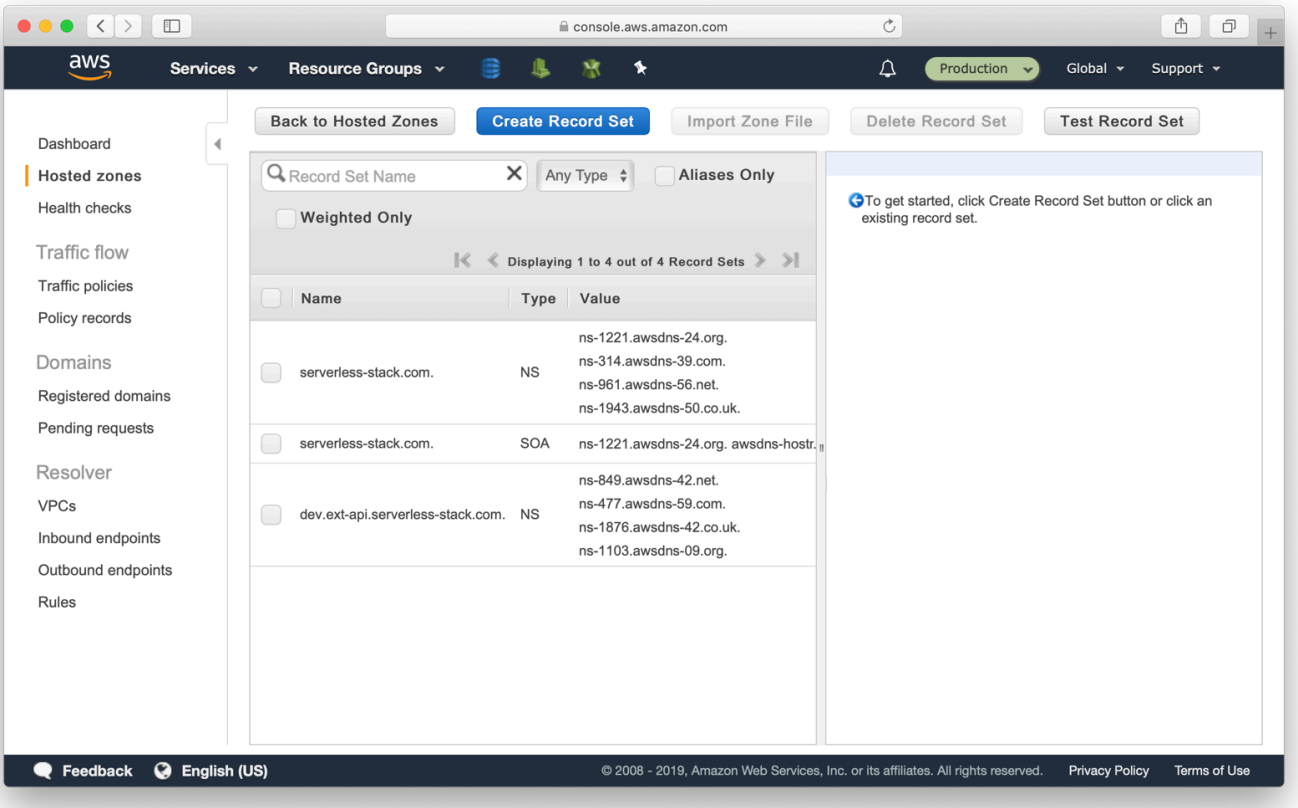
- **Name**: dev.ext-api
- **Type**: NS - Name server

And paste the 4 lines from above in the **Value** field.

Click **Create**.

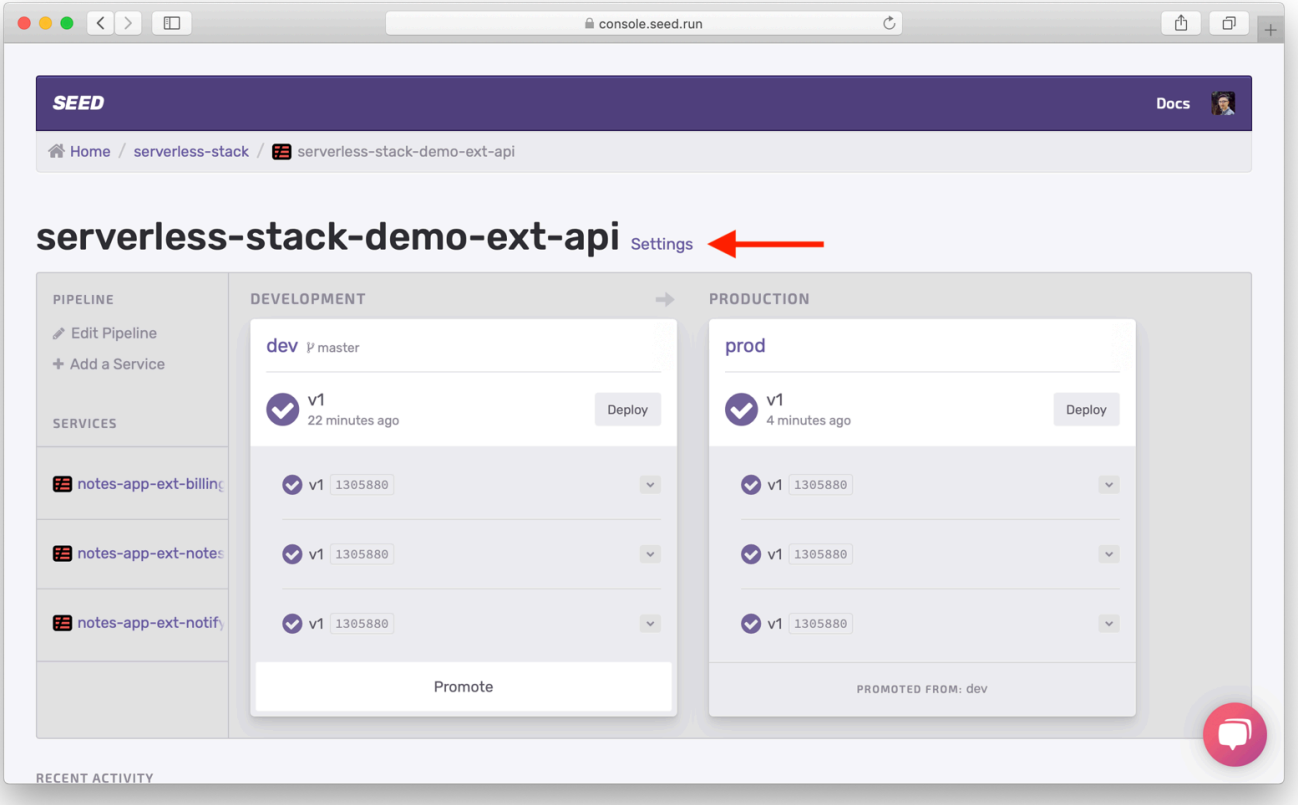


You should see a new `dev.ext-api.serverless-stack.com` row in the table.

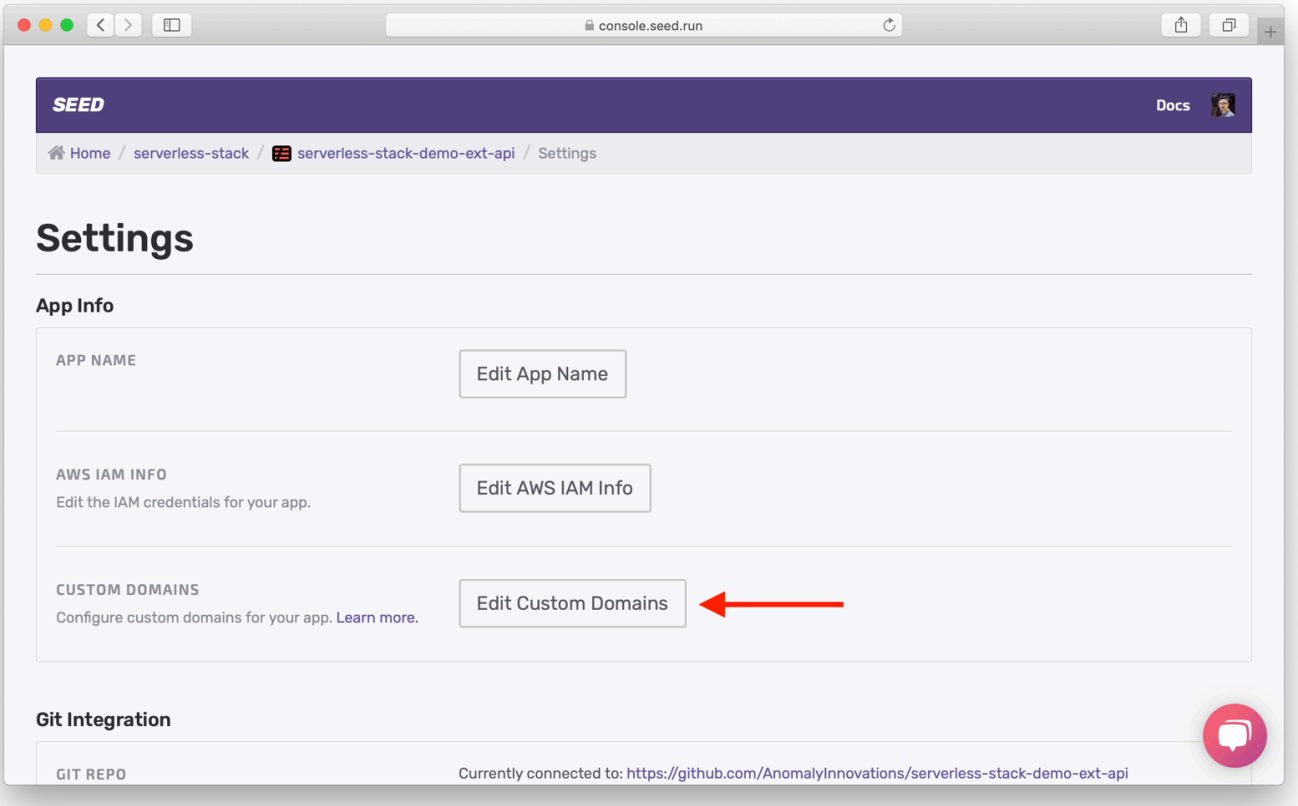


Now we've delegated the `dev.ext-api` subdomain of `serverless-stack.com` to our `Development` AWS account. You can now head over to your app or to Seed and add this as a custom domain (https://seed.run/docs/configuring-custom-domains) for the `dev` stage.

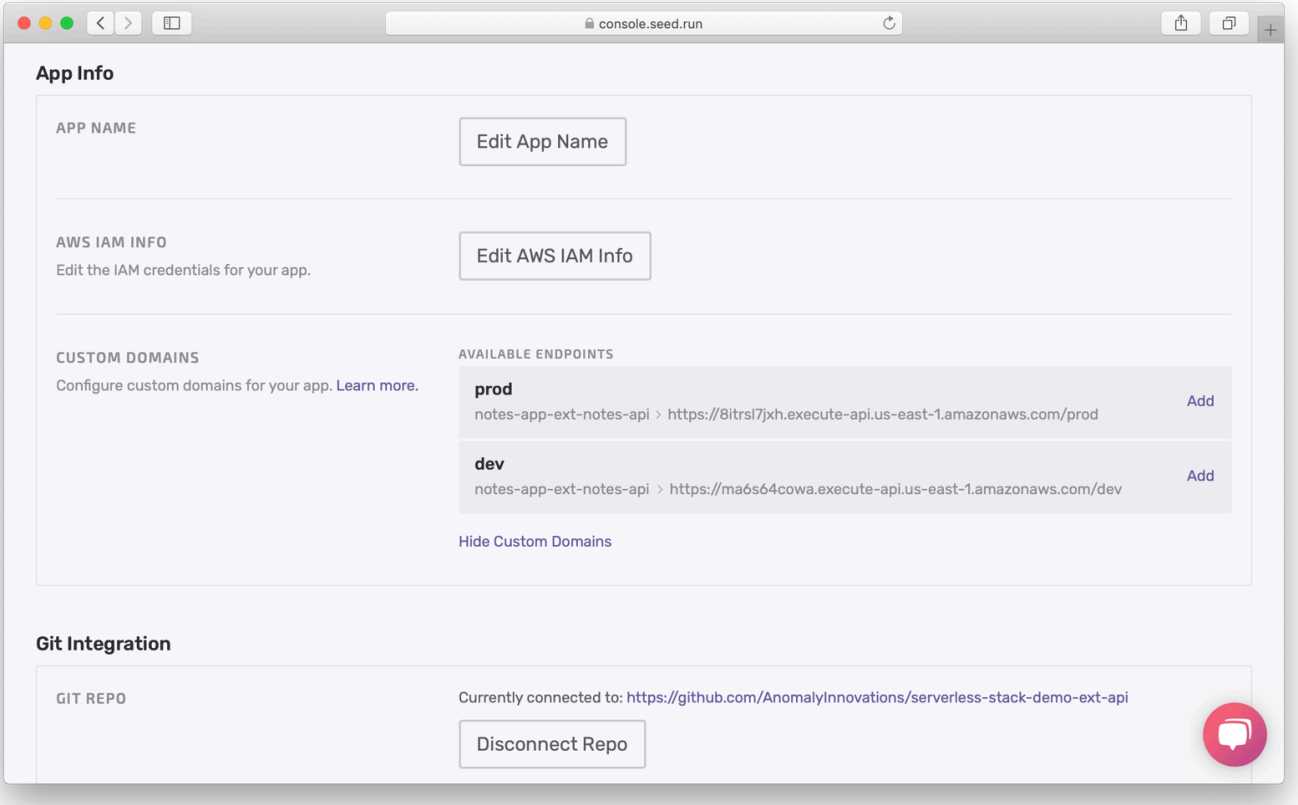
Go to the API app, and head into app settings.



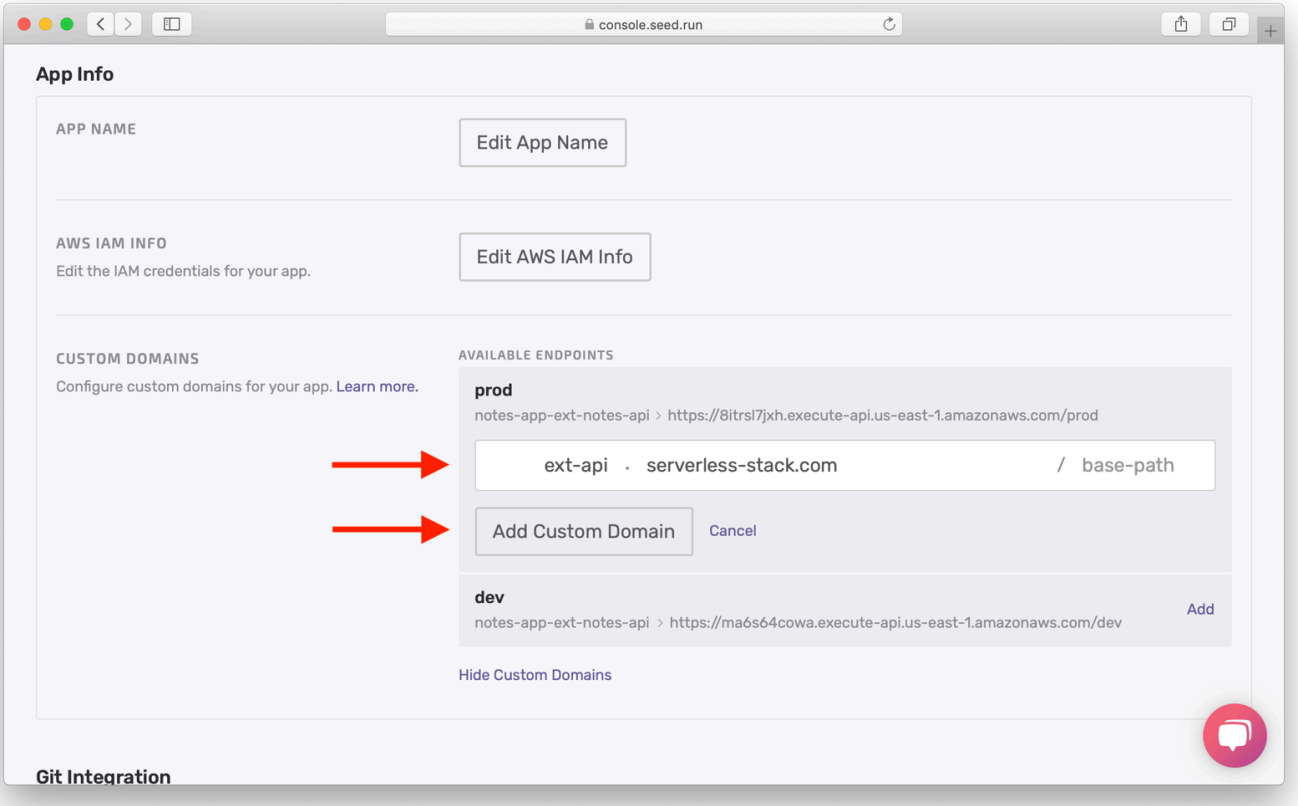
Select **Edit Custom Domains**.



Both **dev** and **prod** endpoints are listed. Select **Add** on the **prod** endpoint.

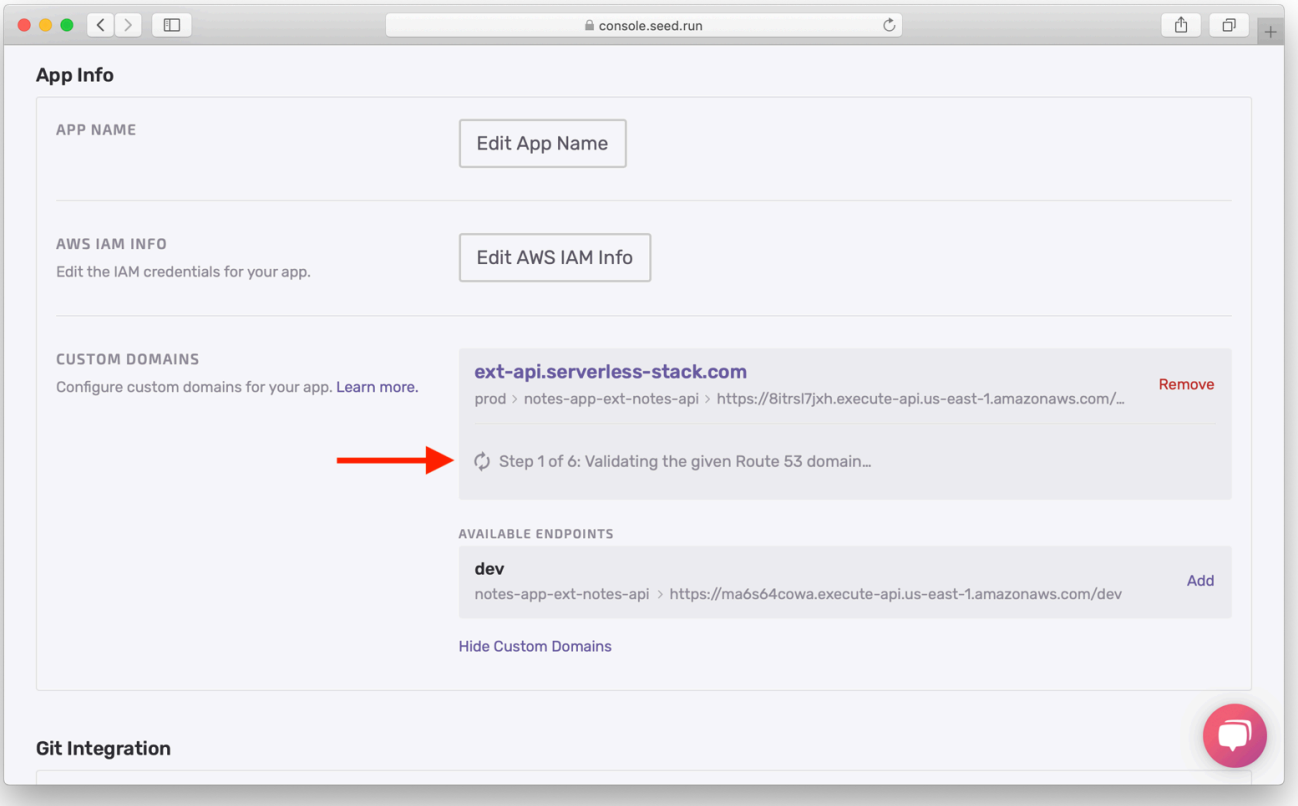


Select the domain **serverless-stack.com** and enter the subdomain **ext-api**. Then select **Add Custom Domain**.

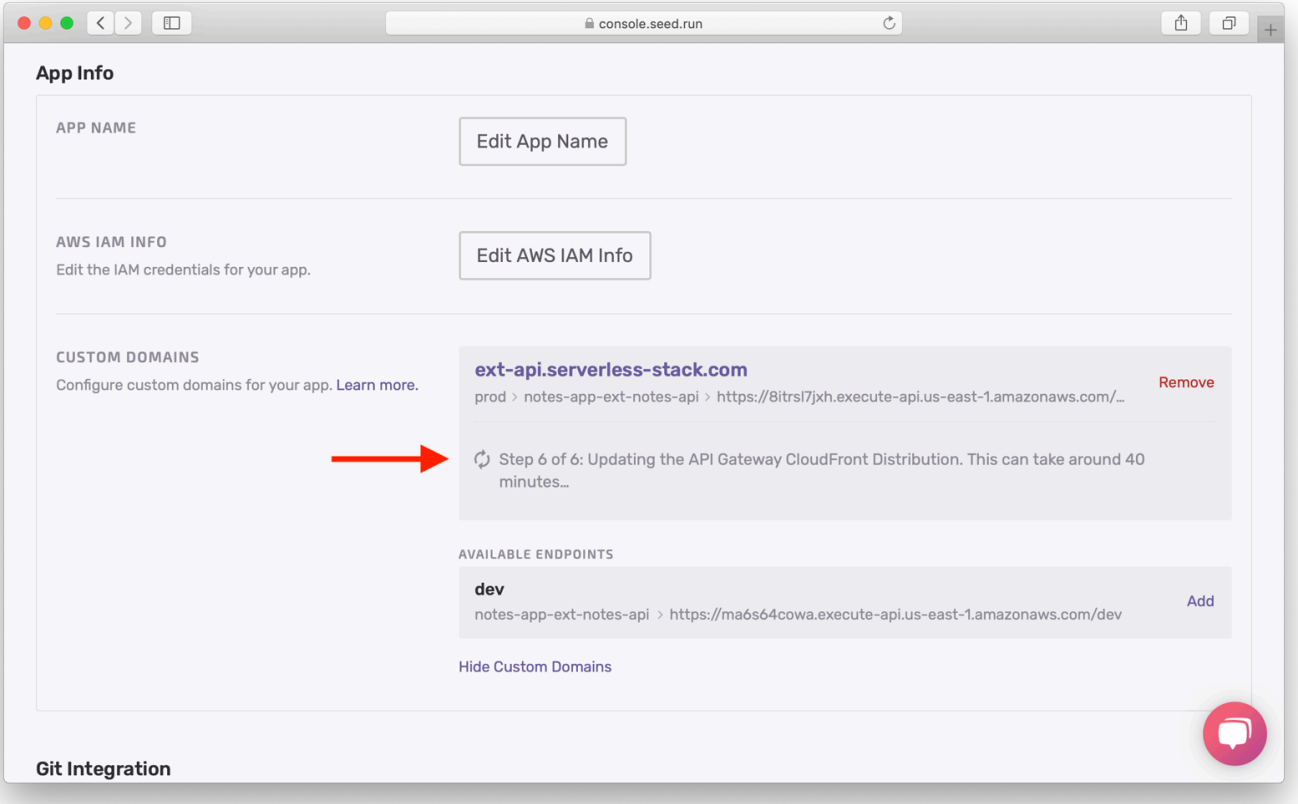


The creation process will go through a couple of phases of

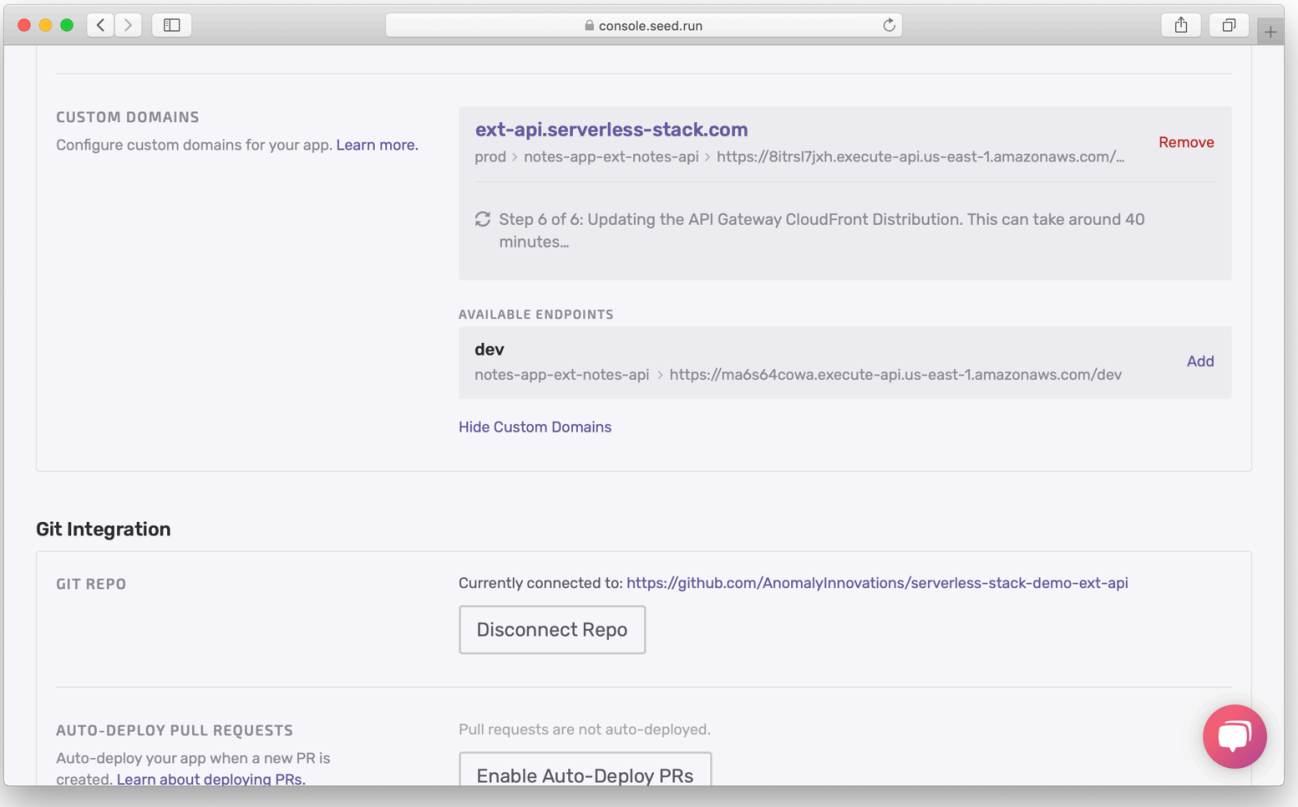
- validating the domain is hosted on Route 53;
- creating the SSL certificate; and
- creating the API Gateway custom domain. Some of the steps are short-lived so you might not see all of them.



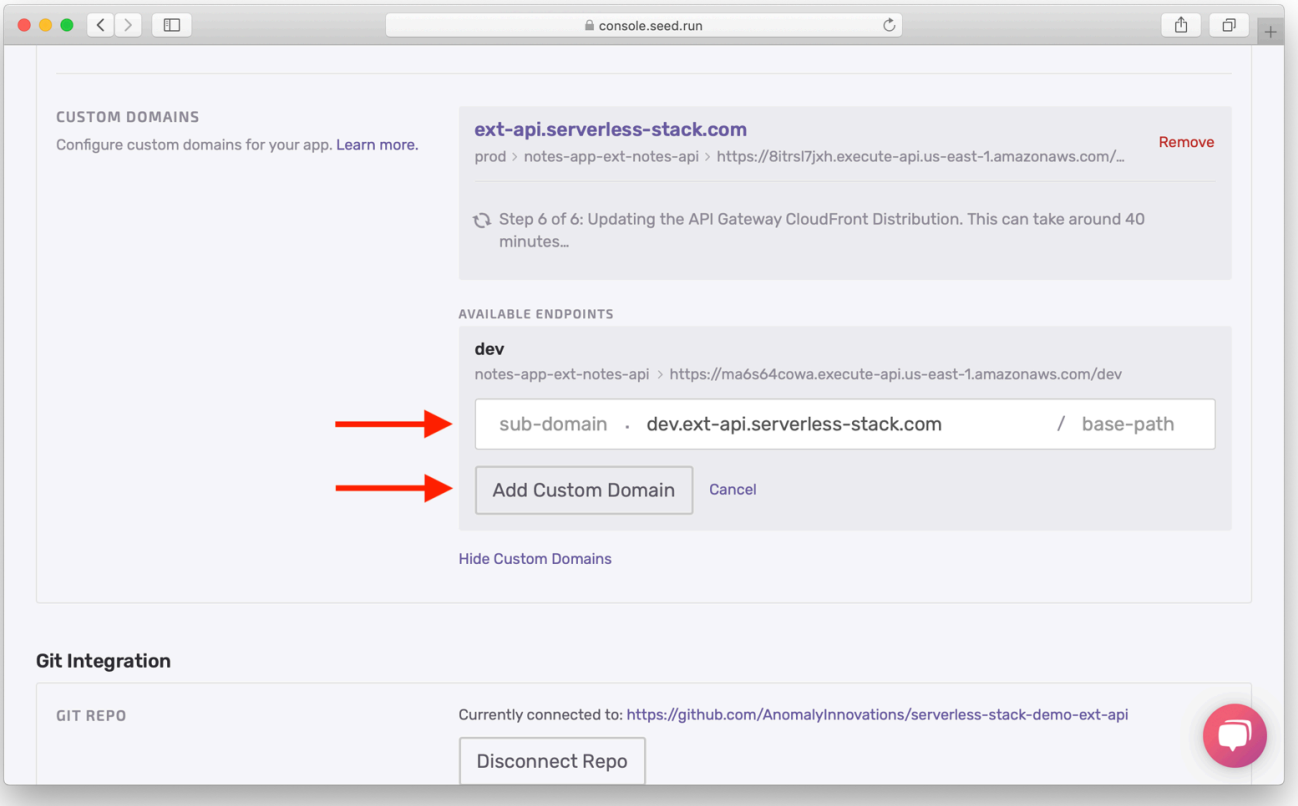
The last step is update the CloudFront distribution, which can take up to 40 minutes. You will be waiting on this step.



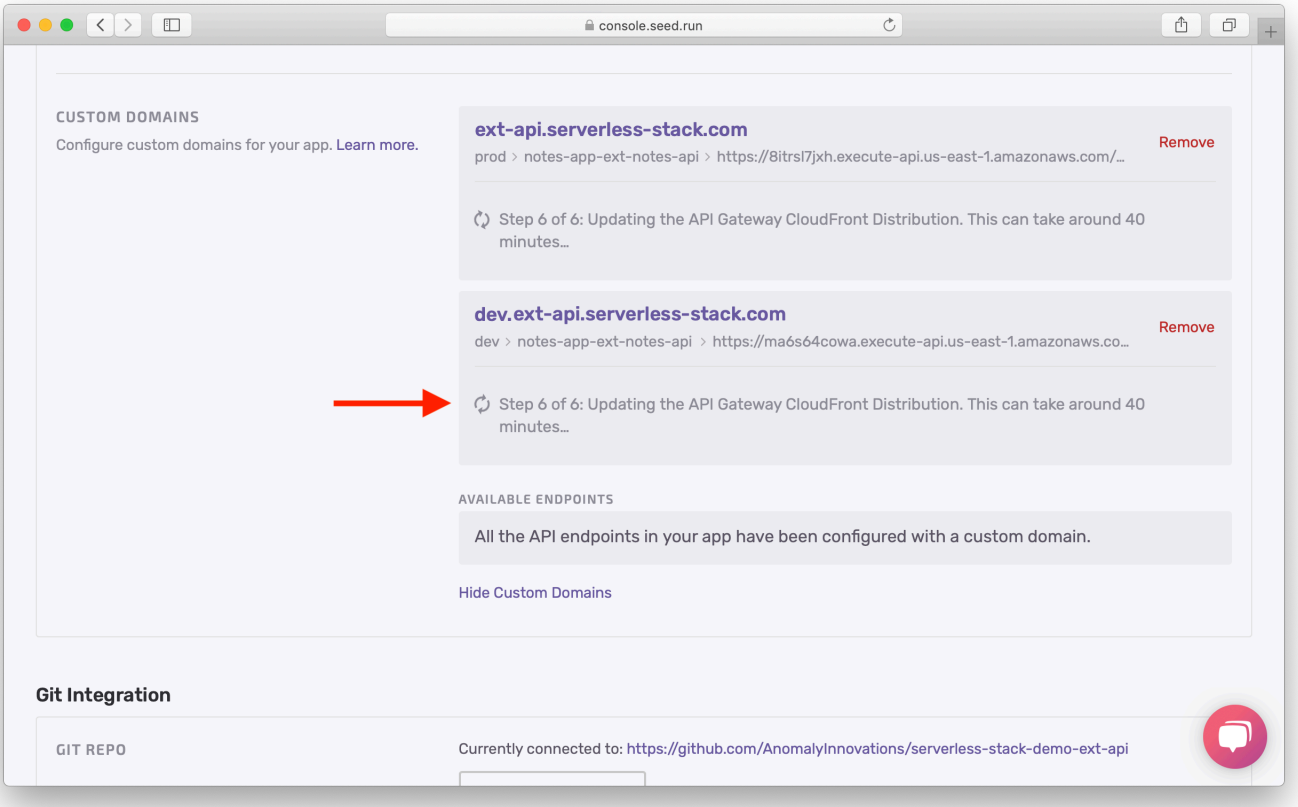
While we wait, let's setup the domain for our **dev** api. Select **Add**.



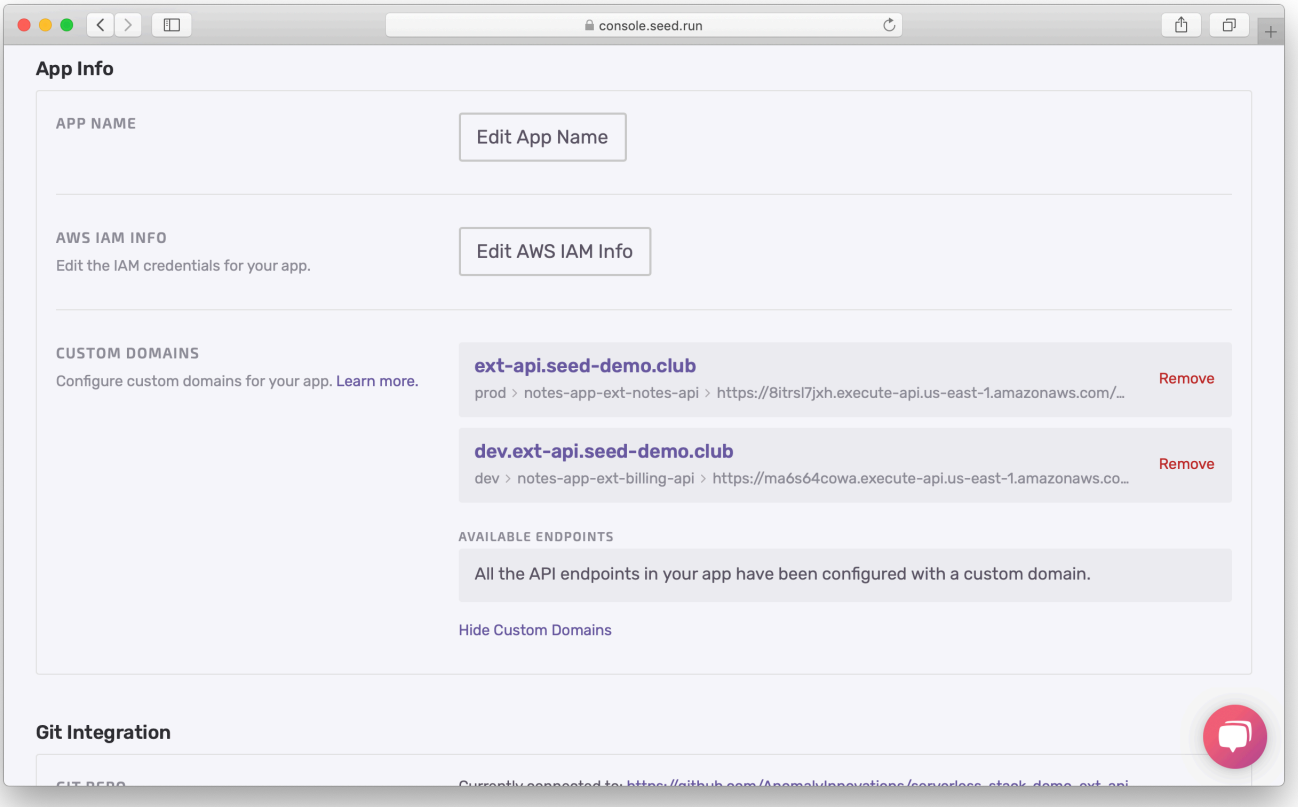
Select the domain **dev.ext-api.serverless-stack.com** and leave the subdomain empty. Then select **Add Custom Domain**.



Similarly, you will wait for up to 40 minutes.



After 40 minutes, the domains will be ready.



Now we've delegated the `dev.api` subdomain of `notes-app.com` to our `Development` AWS account. We'll be configuring our app to use these domains in a later chapter (/chapters/share-route-53-domains-across-aws-accounts.html).

Next, let's quickly look at how you'll be managing the cost and usage for your two AWS accounts.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/share-route-53-domains-across-aws-accounts/1334)

Monitor Usage for Environments

So far we've split the environments for our Serverless app across two AWS accounts. But before we go ahead and look at the development workflow, let's look at how to manage the cost and usage for them.

Our accounts are organized under AWS Organizations. So you don't have to setup the billing details for each account. Billing is consolidated to the master account. You can also see a breakdown of usage and cost for each service in each account.

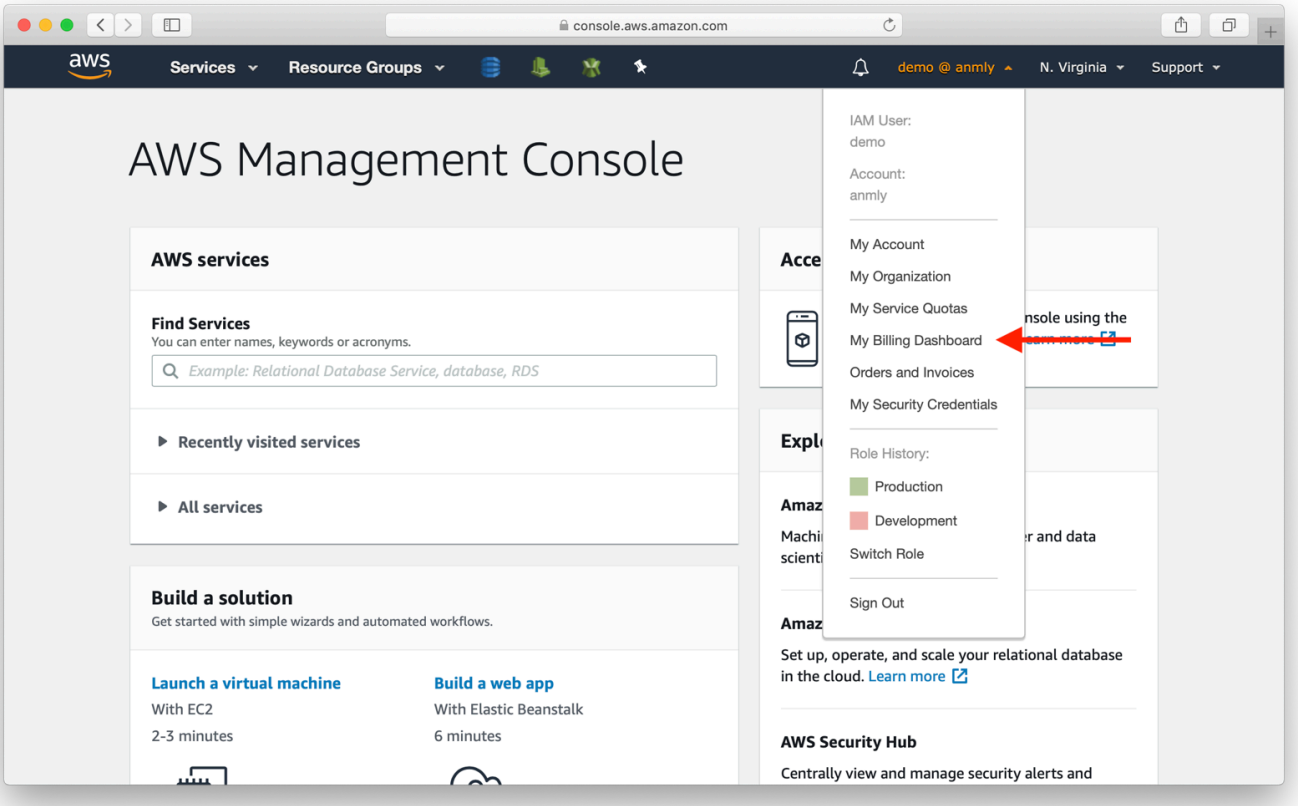
Free Tier

An added bonus of splitting up environments by AWS accounts is that each account in your AWS Organization benefits from the free tier.

For example, Lambda's free tier includes 400 000 seconds per month for 1GB memory Lambda function. That is 400 000 seconds for each of your AWS accounts! If the usage in your `Development` account ends up being low, you'll likely not be paying for it.

Cost/Usage Breakdown by Account

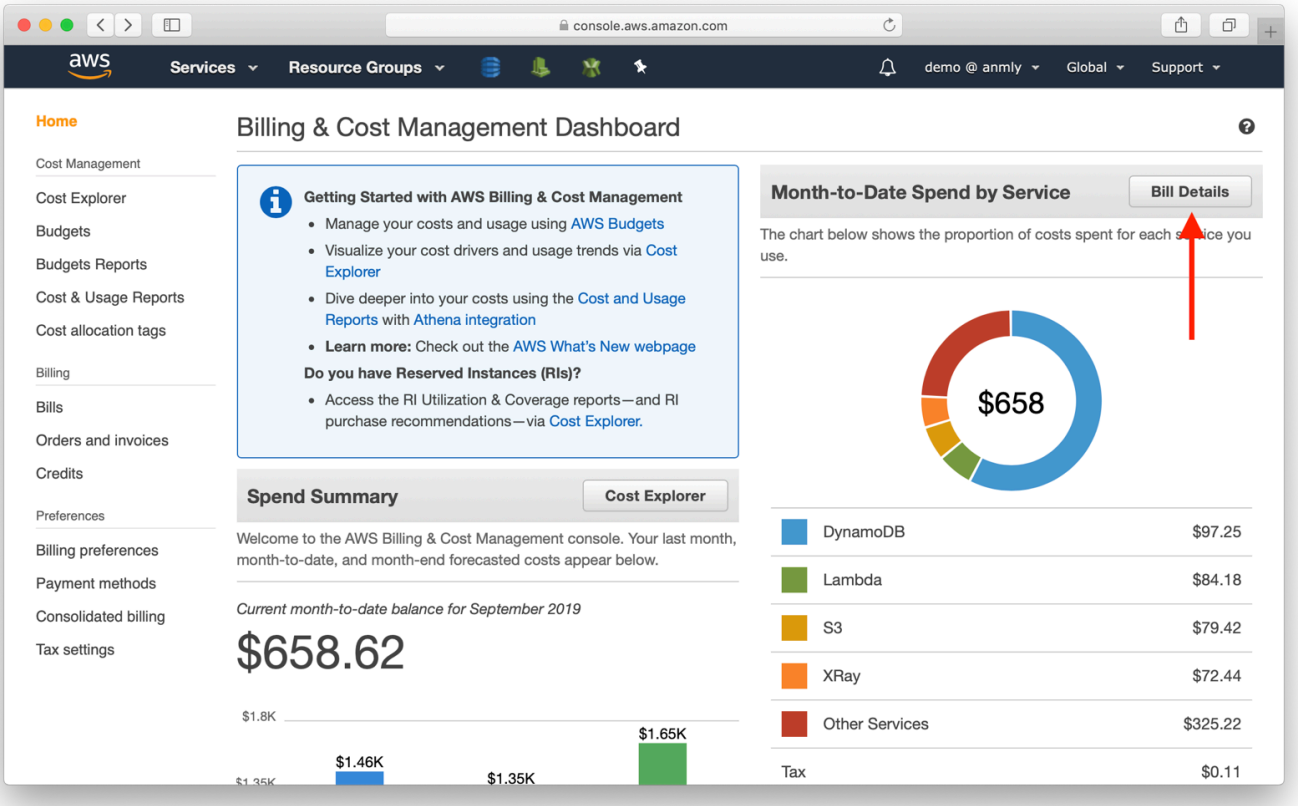
Go into your master account. Select the account picker at the top. Then click **My Billing Dashboard**.



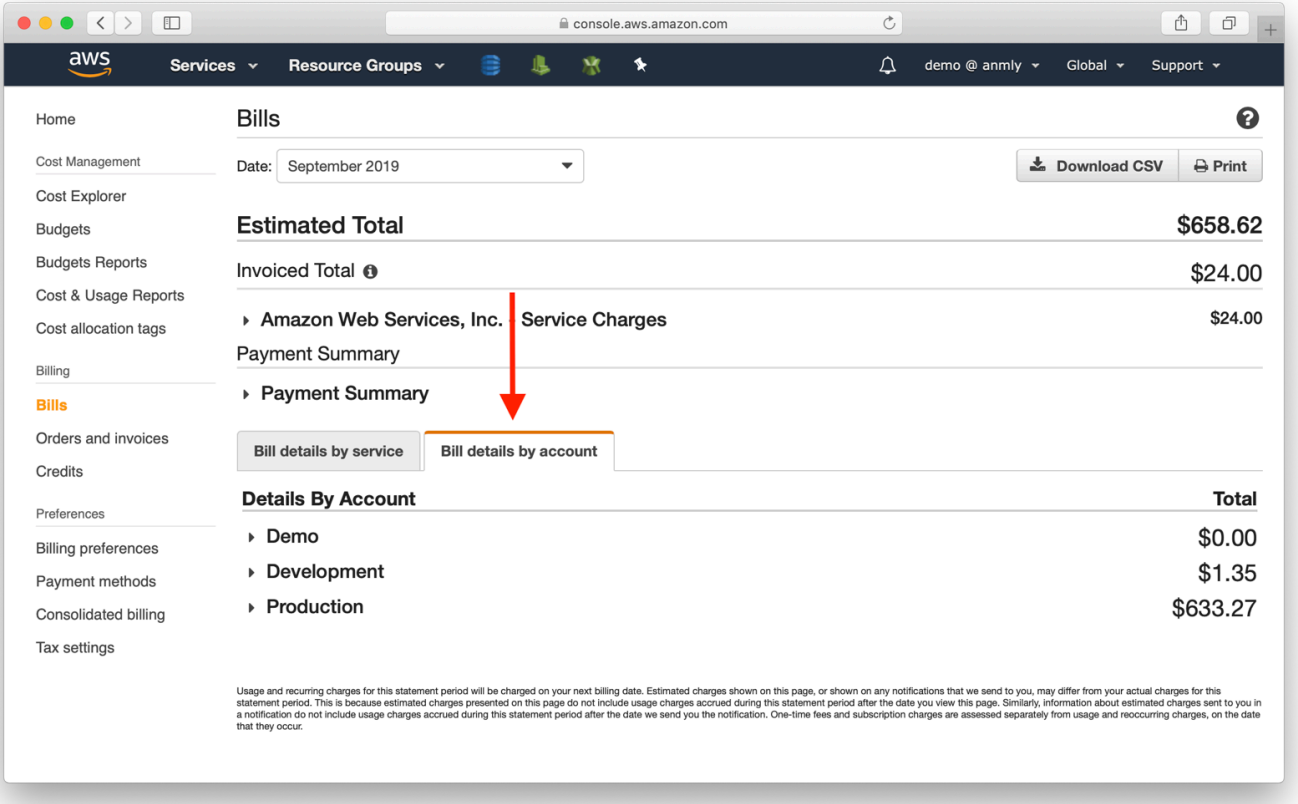
The Billing Dashboard homepage shows you the cost to date for the current calendar month. A couple of very useful features on this page are:

1. **Cost Explorer**: See the cost break down by day/week/month; by account; by resource tag; by service; etc.
2. **Budgets**: Set alert based on usage limits and cost limits.

Click on **Bill Details**.



And click **Bill details by account**. Here you can see the cost allocation for each account.



This should give you a really good idea of the usage and cost for each of your environments.

Now we are ready to look at the development workflow for our app!

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/monitor-usage-for-environments/1328)

Working on Serverless Apps

So to quickly recap, we've split our real world Serverless app into two repos, one creates our infrastructure resources (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-resources) and the second creates our API services (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api).

We've also split our environments across two AWS accounts; `Development` and `Production`. In this section, we are going to look at the development workflow for a real world Serverless app.

Here is roughly what we are going to be covering:

- Developing your Lambda functions locally (/chapters/invoke-lambda-functions-locally.html)
- Invoking API Gateway endpoints locally (/chapters/invoke-api-gateway-endpoints-locally.html)
- Creating and working on feature environments (/chapters/creating-feature-environments.html)
- Creating a pull request environment (/chapters/creating-pull-request-environments.html)
- Promoting dev to production (/chapters/promoting-to-production.html)
- Rolling back (/chapters/rollback-changes.html)

Let's start with how you work locally on your Lambda functions.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/)

Invoke Lambda Functions Locally

After you finish creating a Lambda function, you want to first run it locally.

Invoking Lambda locally

Let's take the **get** function defined in the `serverless.yml` file in the `notes-api` service .

```
functions:
  get:
    handler: get.main
    events:
      - http:
          path: notes/{id}
          method: get
          cors: true
          authorizer: aws_iam
```

And `get.js` looks like:

```
import * as dynamoDbLib from "../../libs/dynamodb-lib";
import { success, failure } from "../../libs/response-lib";
```

```
export async function main(event, context) {
  const params = {
    TableName: 'mono-notes',
    // 'Key' defines the partition key and sort key of
the item to be retrieved
    // - 'userId': Identity Pool identity id of the
authenticated user
    // - 'noteId': path parameter
    Key: {
      userId:
event.requestContext.identity.cognitoIdentityId,
      noteId: event.pathParameters.id
    }
  };

  try {
    const result = await dynamoDbLib.call("get",
params);
    if (result.Item) {
      // Return the retrieved item
      return success(result.Item);
    } else {
      return failure({ status: false, error: "Item not
found." });
    }
  } catch (e) {
    return failure({ status: false });
  }
}
```

The Lambda function is invoked by an API Gateway GET HTTP request, we

need to mock the request parameters. In the `events` directory inside `services/notes-api/`, there is a mock event file called `get-event.json`:

```
{
  "pathParameters": {
    "id": "578eb840-f70f-11e6-9d1a-1359b3b22944"
  },
  "requestContext": {
    "identity": {
      "cognitoIdentityId": "USER-SUB-1234"
    }
  }
}
```

To invoke this function, run the following inside `services/notes-api`:

```
$ serverless invoke local -f get --path events/get-event.json
```

Let's look at a couple of example HTTP event objects.

Query string parameters

To pass in a query string parameter:

```
{
  "queryStringParameters": {
    "key": "value"
  }
}
```

Post data

To pass in a HTTP body for a POST request:

```
{
  "body": "{\"key\":\"value\"}"
}
```

You can also mock the event as if the Lambda function is invoked by other events like SNS, SQS, etc. The content in the mock event file is passed into the Lambda function's event object directly.

Distinguish locally invoked Lambda

You might want to distinguish if the Lambda function was triggered by `serverless invoke local` during testing. For example, you don't want to send analytical events to your analytics server; or you don't want to send emails. You can simply add a runtime environment variable by doing the following:

```
$ IS_LOCAL=true serverless invoke local -f get --path
events/get-event.json
```

And in your code, you can check the environment variable. We use this in our `libs/aws-sdk.js` to disable X-Ray tracing when invoking a function locally:

```
import aws from "aws-sdk";
import xray from "aws-xray-sdk";

// Do not enable tracing for 'invoke local'
```

```
const awsWrapped = process.env.IS_LOCAL ? aws :
xray.captureAWS(aws);

export default awsWrapped;
```

Next, let's look at how we can work with API Gateway endpoints locally.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/invoke-lambda-functions-locally/1325)

Invoke API Gateway Endpoints Locally

Our notes app backend has an API Gateway endpoint. We want to be able to develop against this endpoint locally. To do this we'll use the **serverless-offline plugin** (https://github.com/dherault/serverless-offline) to start a local web server.

Invoke API locally

We installed the above plugin at the repo root, because all API services require the plugin. Open `serverless.yml` in our `notes-api`. You'll notice `serverless-offline` is listed under plugins.

```
service: notes-api

plugins:
  - serverless-offline

...
```

Let's start our local web server.

```
$ cd notes-api
$ serverless offline
```

By default, the server starts on `http://localhost` and on port `3000`. Let's try making a request to the endpoint:

```
$ curl http://localhost:3000/notes
```

Mocking Cognito Identity Pool authentication

Our API endpoint is secured using Cognito Identity Pool. The serverless-offline plugin allows you to pass in Cognito authentication information through the request headers. This allows you to invoke the Lambdas as if they were authenticated by Cognito Identity pool.

To mock a User Pool user id:

```
$ curl --header "cognito-identity-id: 13179724-6380-41c4-8936-64bca3f3a25b" \
  http://localhost:3000/notes
```

You can access the id via `event.requestContext.identity.cognitoIdentityId` in your Lambda function.

To mock the Identity Pool user id:

```
$ curl --header "cognito-authentication-provider: cognito-idp.us-east-1.amazonaws.com/us-east-1_Jw6lUuyG2,cognito-idp.us-east-1.amazonaws.com/us-east-1_Jw6lUuyG2:CognitoSignIn:5f24dbc9-d3ab-4bce-8d5f-eafaeced67ff" \
```

```
  http://localhost:3000/notes
```

And you can access this id via `event.requestContext.identity.cognitoAuthenticationProvider` in your Lambda function.

Working with multiple services

Our app is made up of multiple API services; `notes-api` and `billing-api`. They are two separate Serverless Framework services. They respond to `/notes` and `/billing` path respectively.

The serverless-offline plugin cannot emulate an overall API endpoint. It cannot handle requests and route them to the corresponding service that is responsible for it. This is because the plugin works on the service level and not at the app level.

That said, here is a quick script that lets you run a server on port `8080` while routing `/notes` and `/billing` to their separate services.

```
#!/usr/bin/env node

const { spawn } = require('child_process');
const http = require('http');
const httpProxy = require('http-proxy');
const services = [
  {route:'/billing/*', path:'services/billing-api',
port:3001},
  {route:'/notes/*', path:'services/notes-api',
port:3002},
];
```

```
// Start `serverless offline` for each service
services.forEach(service => {
  const child = spawn('serverless', ['offline',
'start', '--stage', 'dev', '--port', service.port],
{cwd: service.path});
  child.stdout.setEncoding('utf8');
  child.stdout.on('data', chunk => console.log(chunk));
  child.stderr.on('data', chunk => console.log(chunk));
  child.on('close', code => console.log(`child exited
with code ${code}`));
});

// Start a proxy server on port 8080 forwarding based
on url path
const proxy = httpProxy.createProxyServer({});
const server = http.createServer(function(req, res) {
  const service = services.find(per =>
urlMatchRoute(req.url, per.route));
  // Case 1: matching service FOUND => forward request
to the service
  if (service) {
    proxy.web(req, res,
{target:`http://localhost:${service.port}`});
  }
  // Case 2: matching service NOT found => display
available routes
  else {
    res.writeHead(200, { 'Content-Type': 'text/plain'
});
    res.write(`Url path "${req.url}" does not match
```

```
routes defined in services\n\n`);
    res.write(`Available routes are:\n`);
    services.map(service => res.write(`-
${service.route}\n`));
    res.end();
  }
});
server.listen(8080);

// Check match route
// - ie. url is '/notes/123'
// - ie. route is '/notes/*'
function urlMatchRoute(url, route) {
  const urlParts = url.split('/');
  const routeParts = route.split('/');
  for (let i = 0, l = routeParts.length; i < l; i++) {
    const urlPart = urlParts[i];
    const routePart = routeParts[i];

    // Case 1: If either part is undefined => not match
    if (urlPart === undefined || routePart ===
undefined) { return false; }

    // Case 2: If route part is match all => match
    if (routePart === '*') { return true; }

    // Case 3: Exact match => keep checking
    if (urlPart === routePart) { continue; }

    // Case 4: route part is variable => keep checking
    if (routePart.startsWith('{')) { continue; }
```

```
  }

  return true;
}
```

This script is in included as `startServer` in the sample repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api). But let's quickly look at how it works. It has 4 sections:

1. At the very top, we define the services we are going to start. Tweak this to include any new services that you add.
2. We then start each service based on the port defined using the serverless-offline plugin.
3. We start an HTTP server on port 8080. In the request handling logic, we look for a service with a matching route. If one is found, the server proxies the request to the service.
4. At the bottom, we have a function that checks if a route matches a url.

You can run this server locally from the project root using:

```
$ ./startServer
```

Now that we have a good idea of how to develop our Lambda functions locally, Let's look at what happens when you want to create an environment for a new feature.

For help and discussion

Comments on this chapter (https://discourse.serverless-

```
stack.com/t/invoke-api-gateway-endpoints-
locally/1324)
```

Creating Feature Environments

Over the last couple of chapters we looked at how to work on Lambda and API Gateway locally. However, besides Lambda and API Gateway, your project will have other AWS services. To run your code locally, you have to simulate all the AWS services. Similar to serverless-offline (https://www.github.com/dherault/serverless-offline), there are plugins like serverless-dynamodb-local (https://www.github.com/99xt/serverless-dynamodb-local) and serverless-offline-sns (https://github.com/mj1618/serverless-offline-sns) that can simulate DynamoDB and SNS. However, mocking only takes you so far since they do not simulate IAM permissions and they are not always up to date with the services' latest changes. You want to test your code with the real resources.

Serverless is really good at creating ephemeral environments. Let's look at what the workflow looks like when you are trying to add a new feature to your app.

As an example we'll add a feature that lets you *like* a note. We will add a new API endpoint `/notes/{id}/like`. We are going to work on this in a new feature branch and then deploy this using Seed.

Create a feature branch

We will create a new feature branch called `like`.

```
$ git checkout -b like
```

Since we are going to be using `/notes/{id}/like` as our endpoint we need to first export the `/notes/{id}` API path. Open the `serverless.yml` in the `services/notes-api` service, and append to the resource outputs.

```
ApiGatewayResourceNotesIdVarId:
  Value:
    Ref: ApiGatewayResourceNotesIdVar
  Export:
    Name: ${self:custom.stage}-
ExtApiGatewayResourceNotesIdVarId
```

Our resource outputs should now look like:

```
...
  - Outputs:
      ApiGatewayRestApiId:
        Value:
          Ref: ApiGatewayRestApi
        Export:
          Name: ${self:custom.stage}-
ExtApiGatewayRestApiId

      ApiGatewayRestApiRootResourceId:
        Value:
           Fn::GetAtt:
            - ApiGatewayRestApi
```

```
            - RootResourceId
        Export:
          Name: ${self:custom.stage}-
ExtApiGatewayRestApiRootResourceId

      ApiGatewayResourceNotesIdVarId:
        Value:
          Ref: ApiGatewayResourceNotesIdVar
        Export:
          Name: ${self:custom.stage}-
ExtApiGatewayResourceNotesIdVarId
```

Let's create the `like-api` service.

```
$ cd services
$ mkdir like-api
$ cd like-api
```

Add a `serverless.yml`.

```
service: notes-app-ext-like-api

plugins:
  - serverless-bundle
  - serverless-offline

custom: ${file(../../serverless.common.yml):custom}

package:
  individually: true
```

```
provider:
  name: aws
  runtime: nodejs10.x
  stage: dev
  region: us-east-1
  tracing:
    lambda: true

  apiGateway:
    restApiId:
      'Fn::ImportValue': ${self:custom.stage}-
ExtApiGatewayRestApiId
    restApiRootResourceId:
      'Fn::ImportValue': ${self:custom.stage}-
ExtApiGatewayRestApiRootResourceId
    restApiResources:
      /notes/{id}:
        'Fn::ImportValue': ${self:custom.stage}-
ExtApiGatewayResourceNotesIdVarId

  environment:
    stage: ${self:custom.stage}
    resourcesStage: ${self:custom.resourcesStage}

  iamRoleStatements:
    -
${file(../../serverless.common.yml):lambdaPolicyXRay}

functions:
  like:
    handler: like.main
```

```
    events:
      - http:
          path: /notes/{id}/like
          method: post
          cors: true
          authorizer: aws_iam
```

Again, the `like-api` will share the same API endpoint as the `notes-api` service.

Add the handler file `like.js`.

```
import { success } from "../../libs/response-lib";

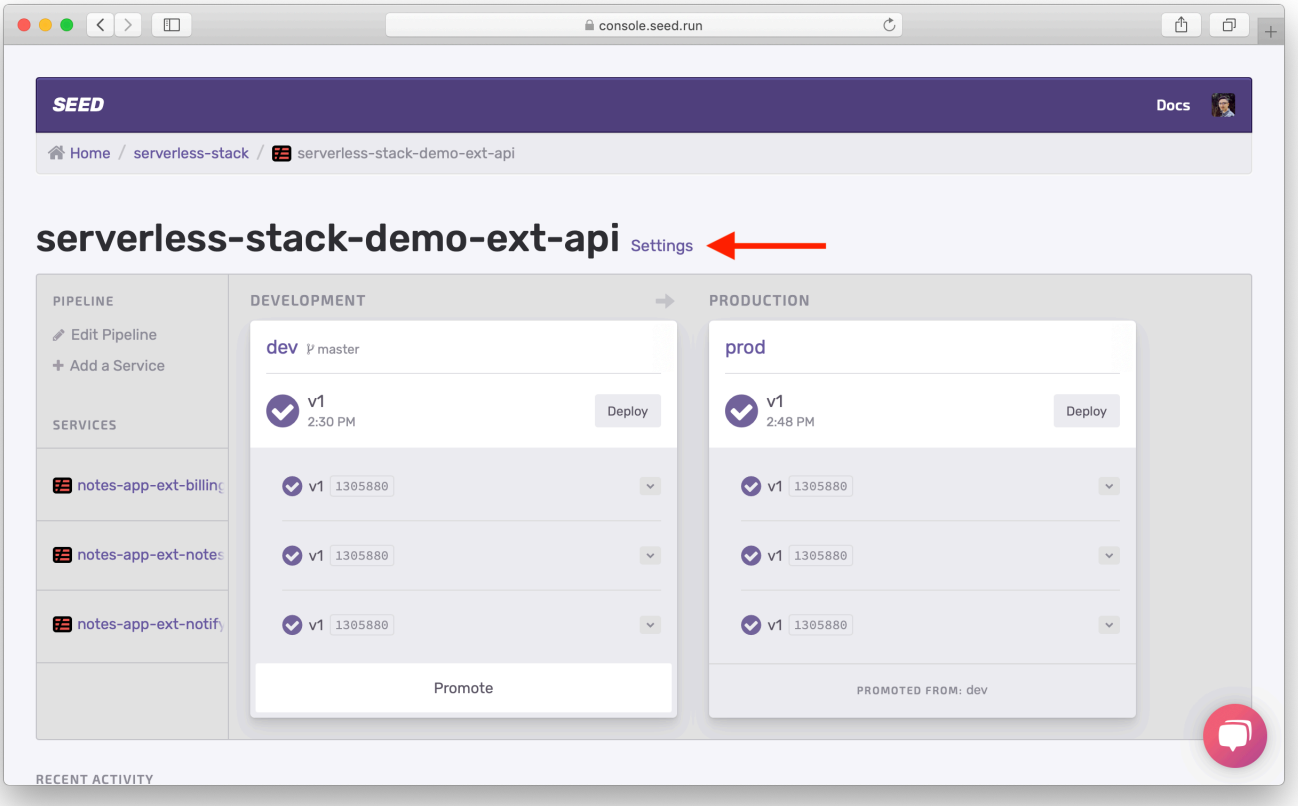
export async function main(event, context) {
  // business logic code for liking a post

  return success({ status: true });
}
```

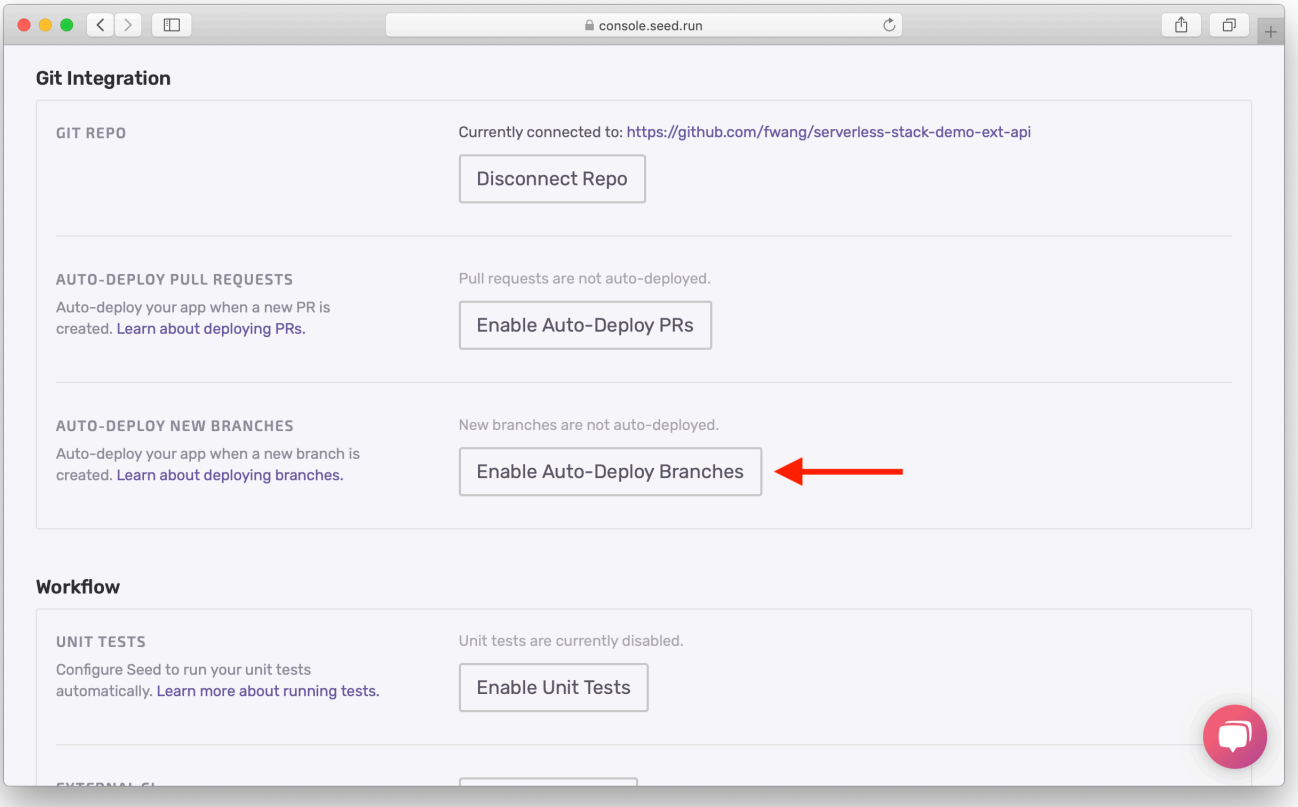
Now before we push our Git branch, let's enable the branch workflow in Seed.

Enable branch workflow in Seed

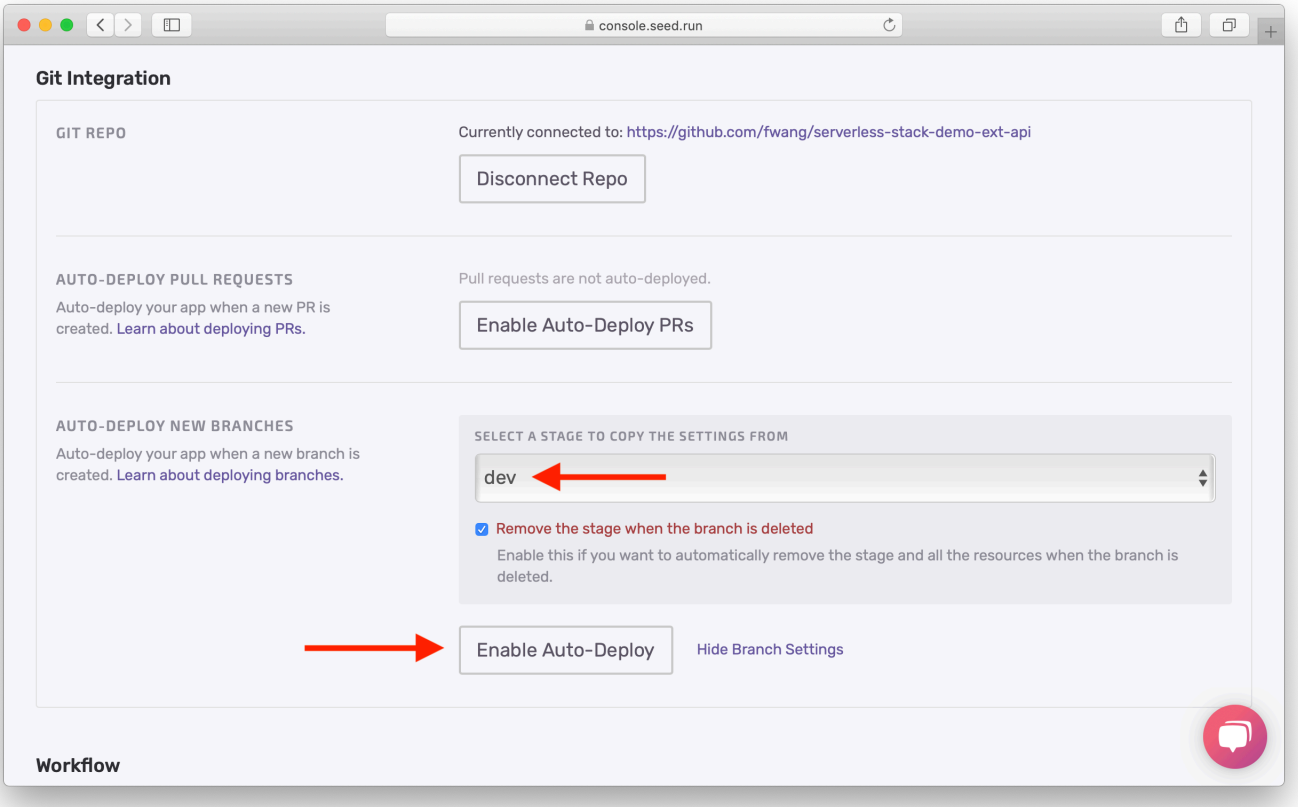
Go to your app on Seed. Click **Settings**.



Scroll down to **Git Integration**. Then select **Enable Auto-Deploy Branches**.

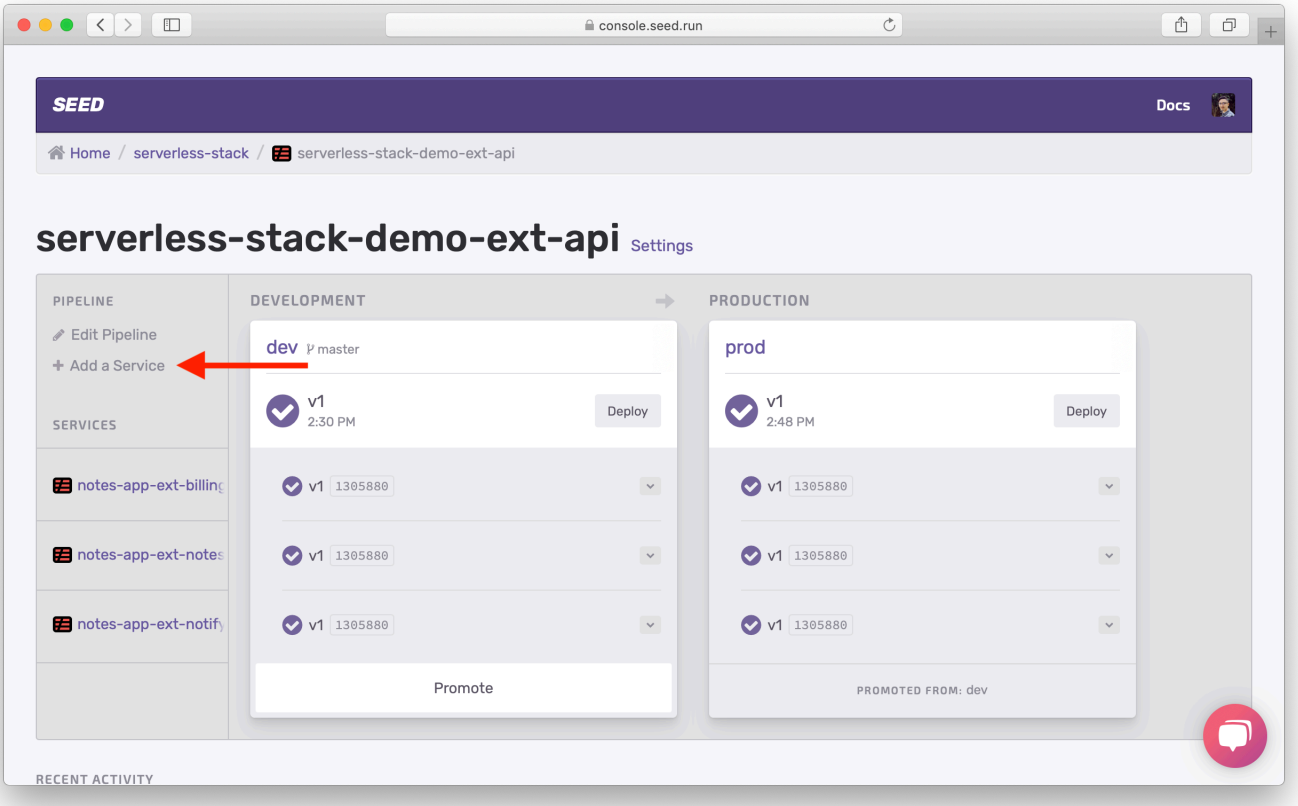


Select the **dev** stage, since we want the stage to be deployed into the **Development** AWS account. Click **Enable Auto-Deploy**.

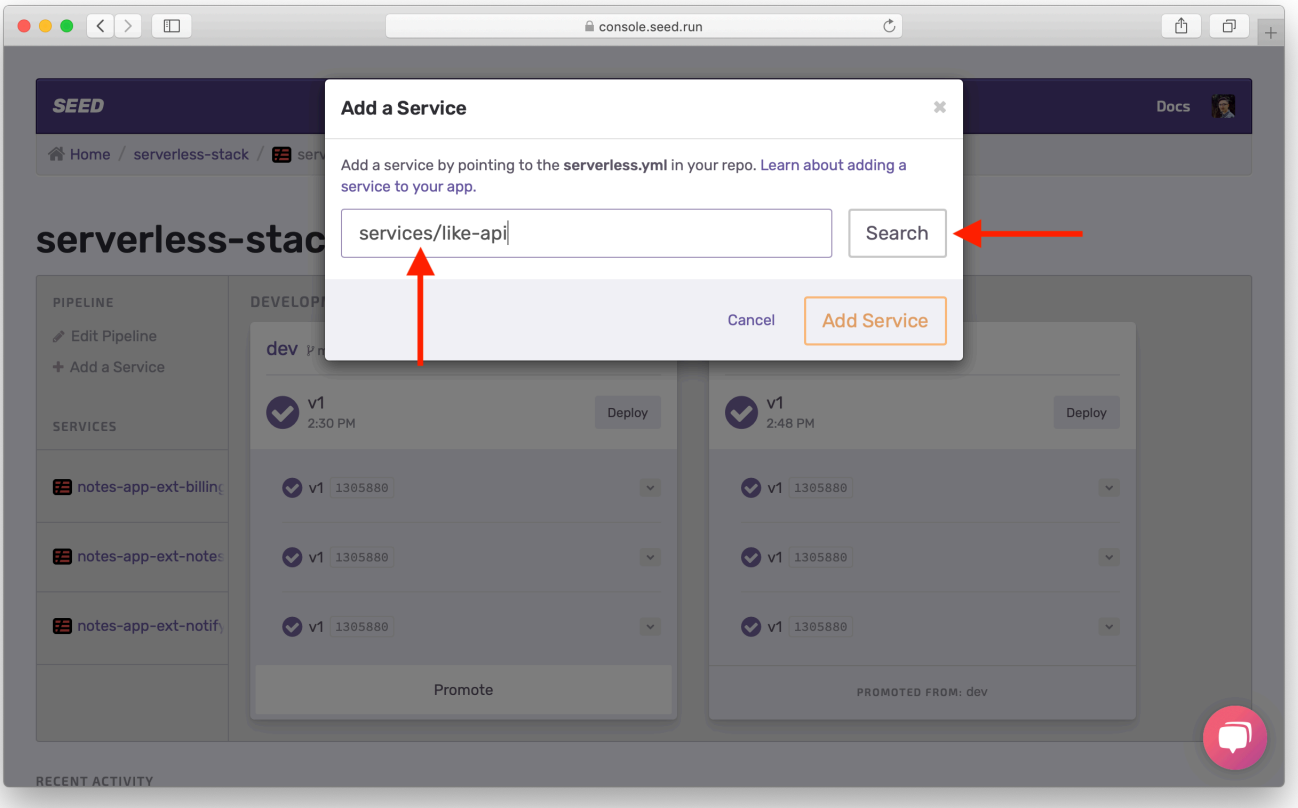


Add the new service to Seed

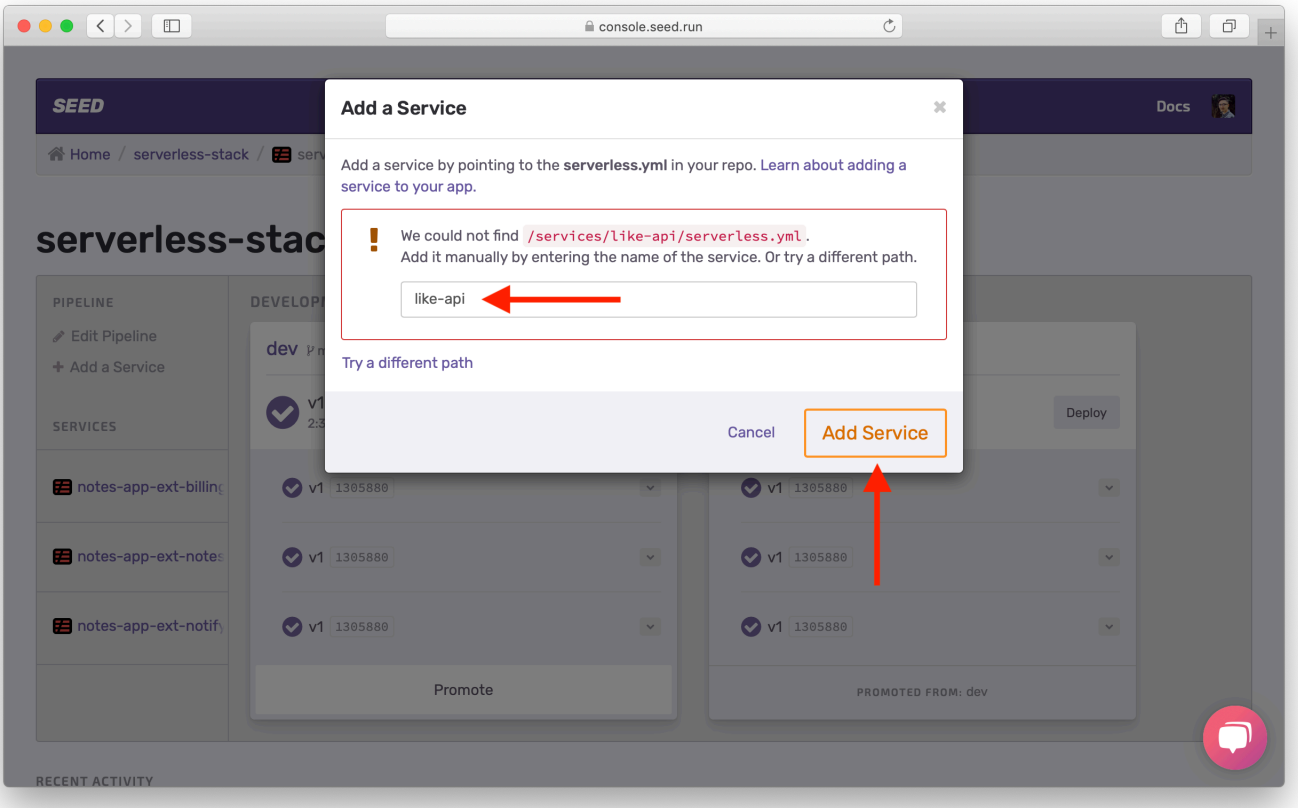
Back in our app, click on **Add a Service**.



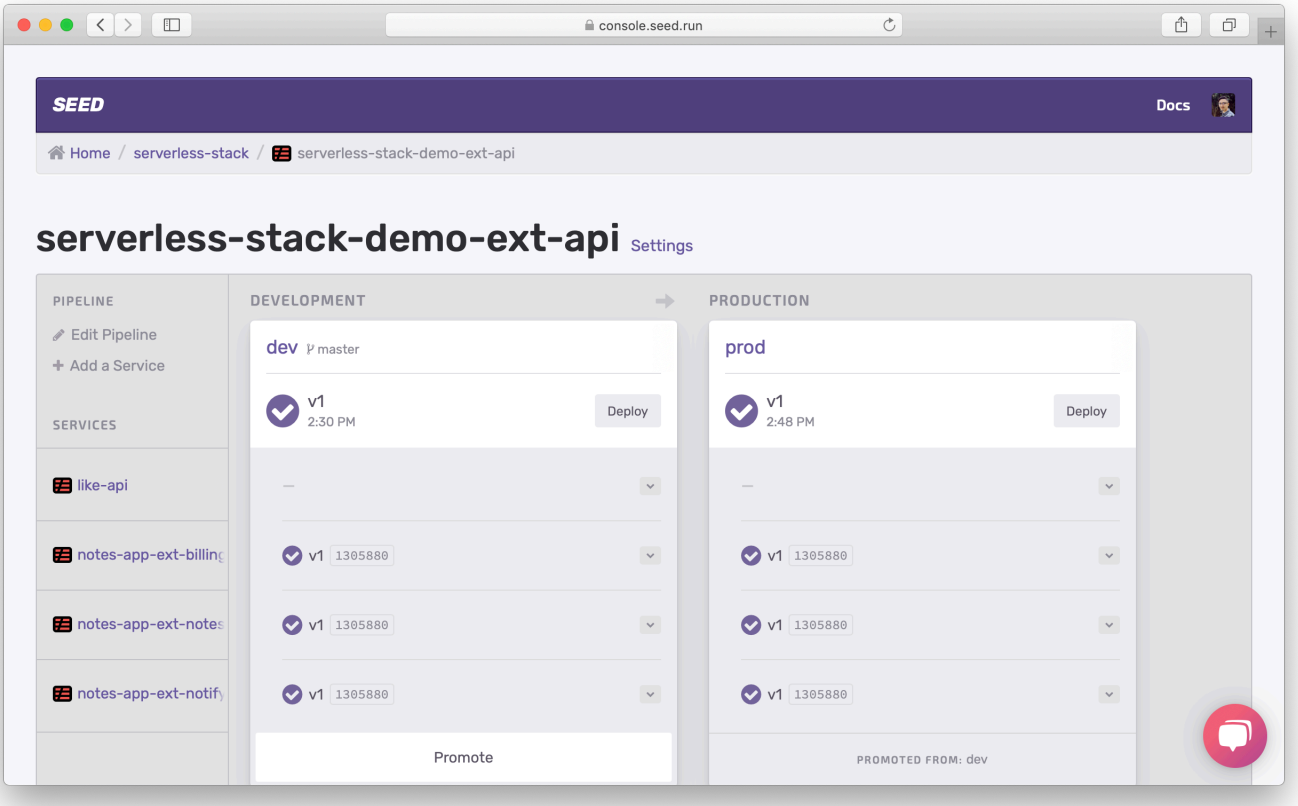
Enter the path to the service `services/like-api` and click **Search**.



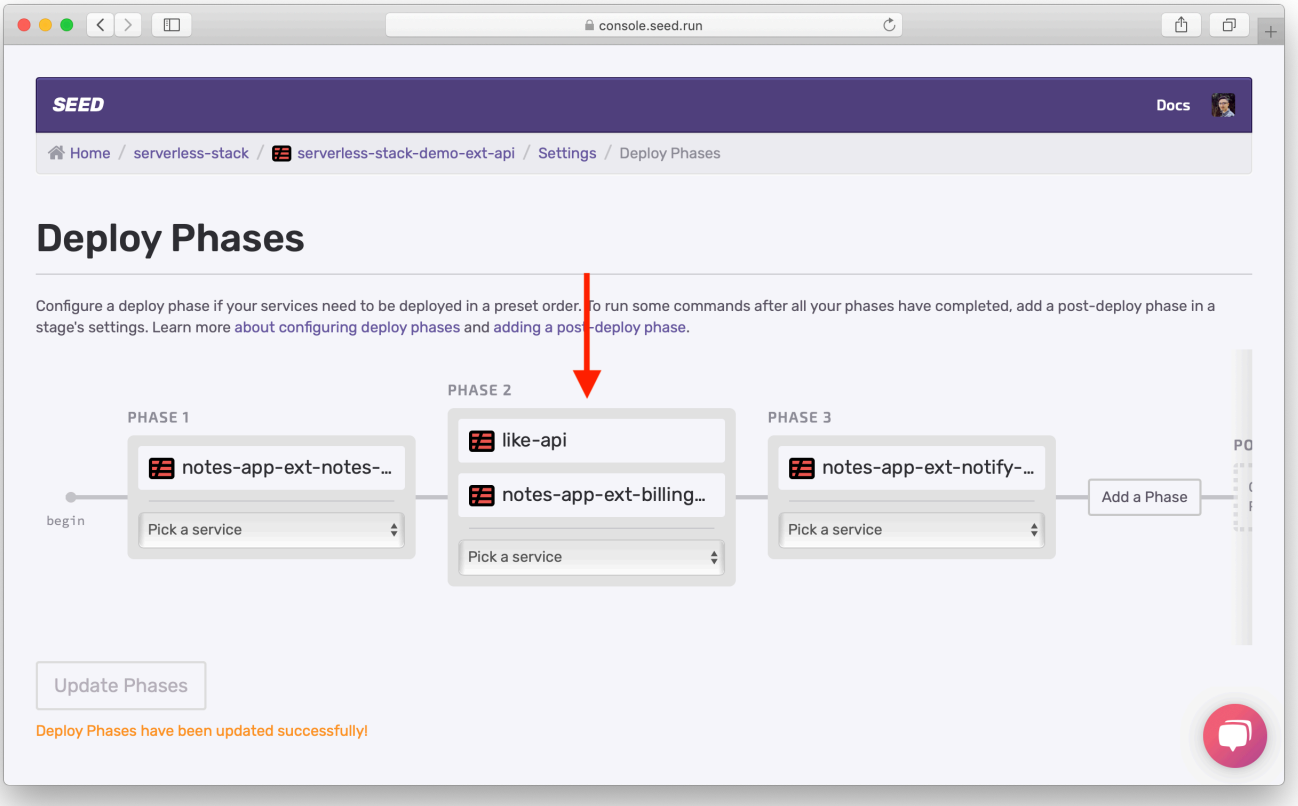
Since the code has not been committed to Git yet, Seed is not able to find the `serverless.yml` of the service. That's totally fine. We'll specify a name for the service `like-api`. Then hit **Add Service**.



Now, we have the service added.



By default, the new service is added to the latest deploy phase. Let's head into the **Manage Deploy Phases** in the app **Settings**, and move it to Phase 2. This is because it's dependent on the API Gateway resources exported by `notes-api`.

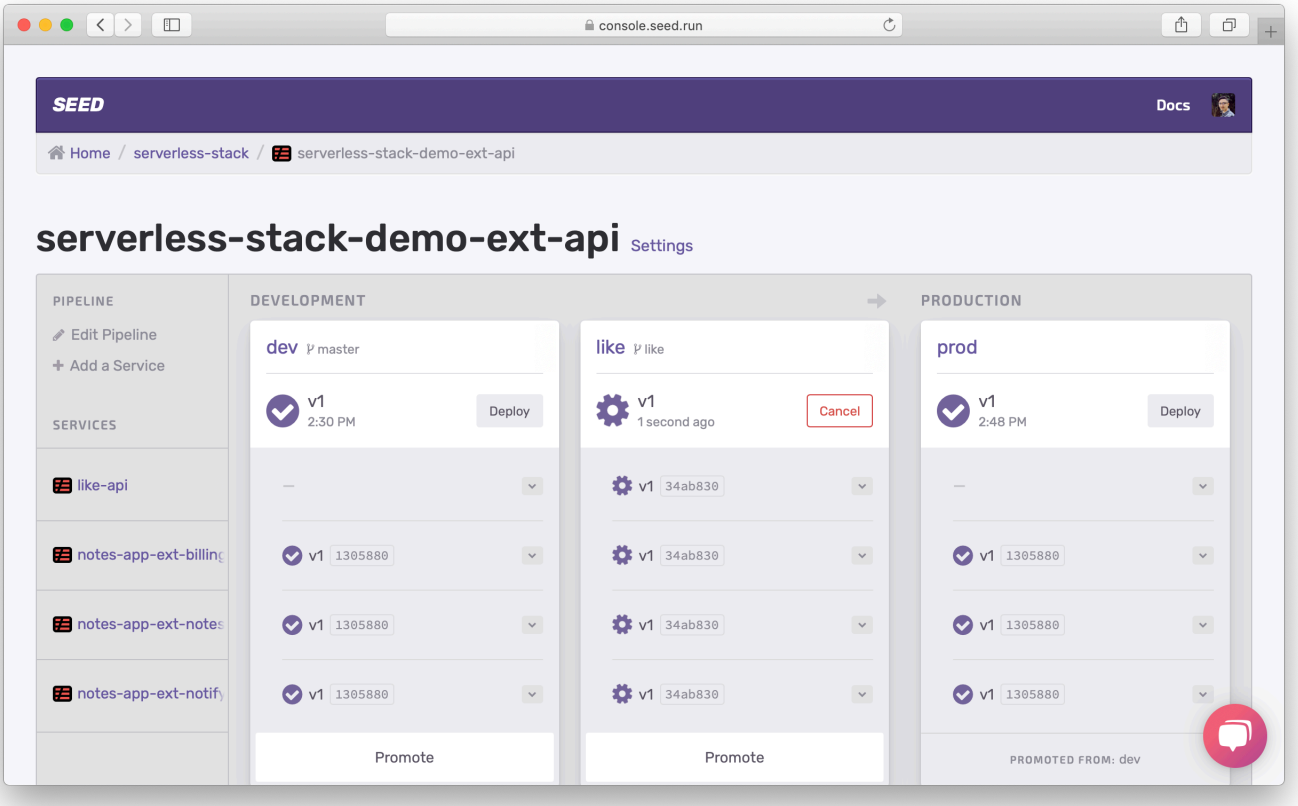


Git push to deploy new feature

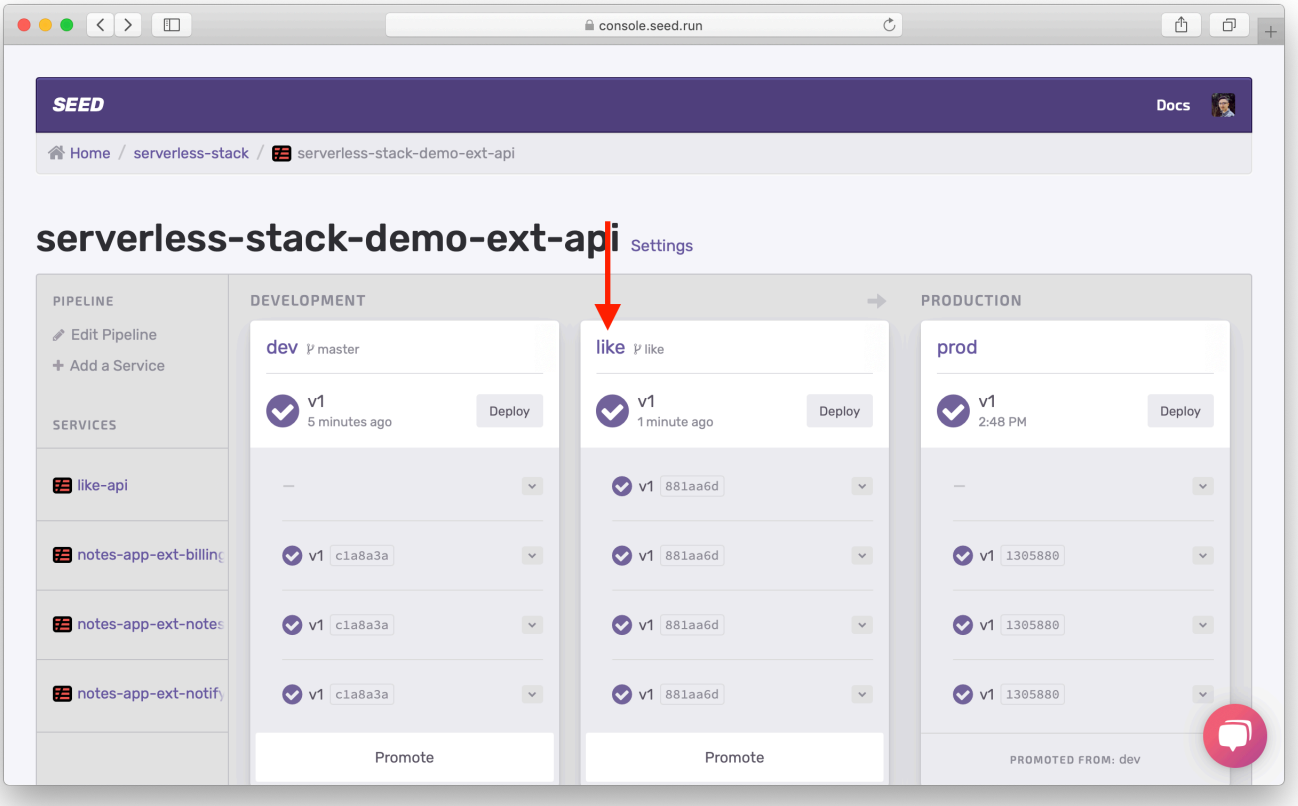
Now we are ready to create our new feature environment. Go back to our command line, and then push the code to the `like` branch.

```
$ git add .
$ git commit -m "Add like API"
$ git push --set-upstream origin like
```

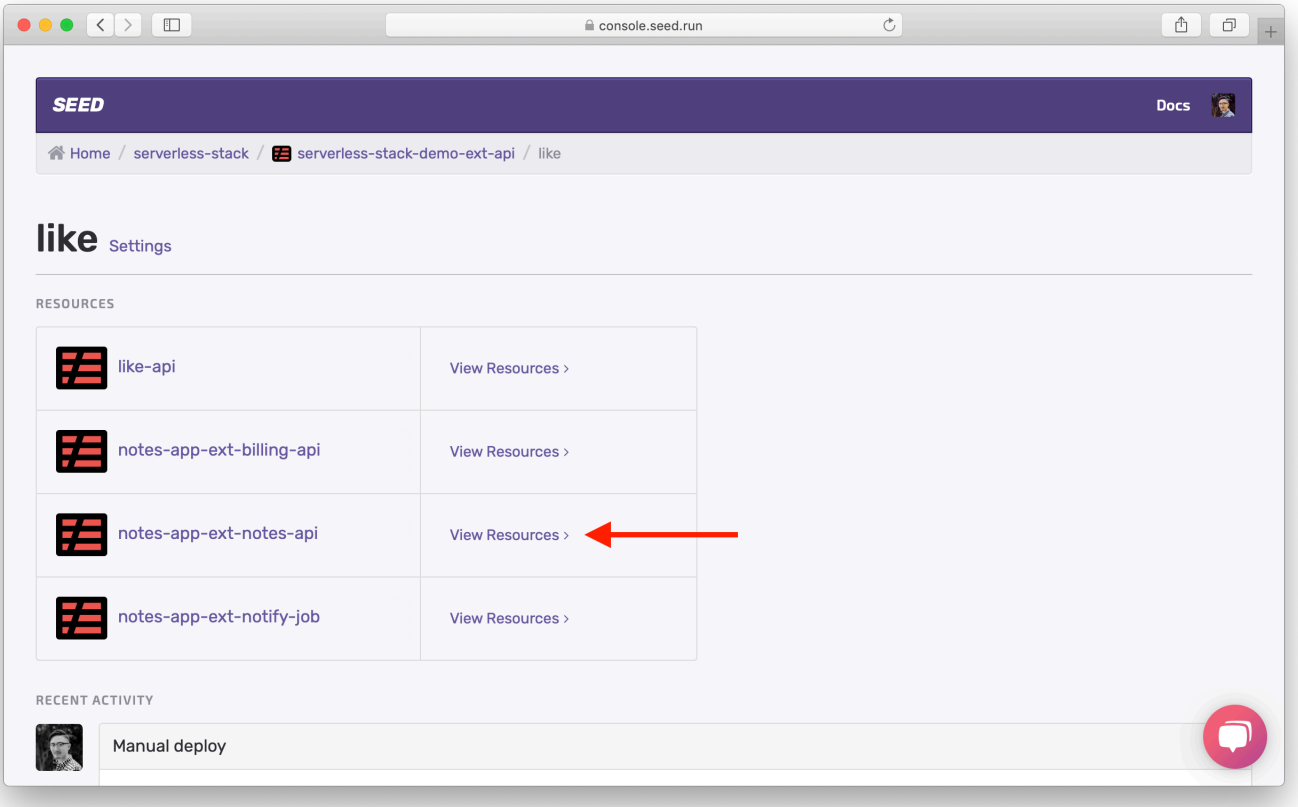
Back in Seed, a new stage called **like** is created and is being deployed automatically.



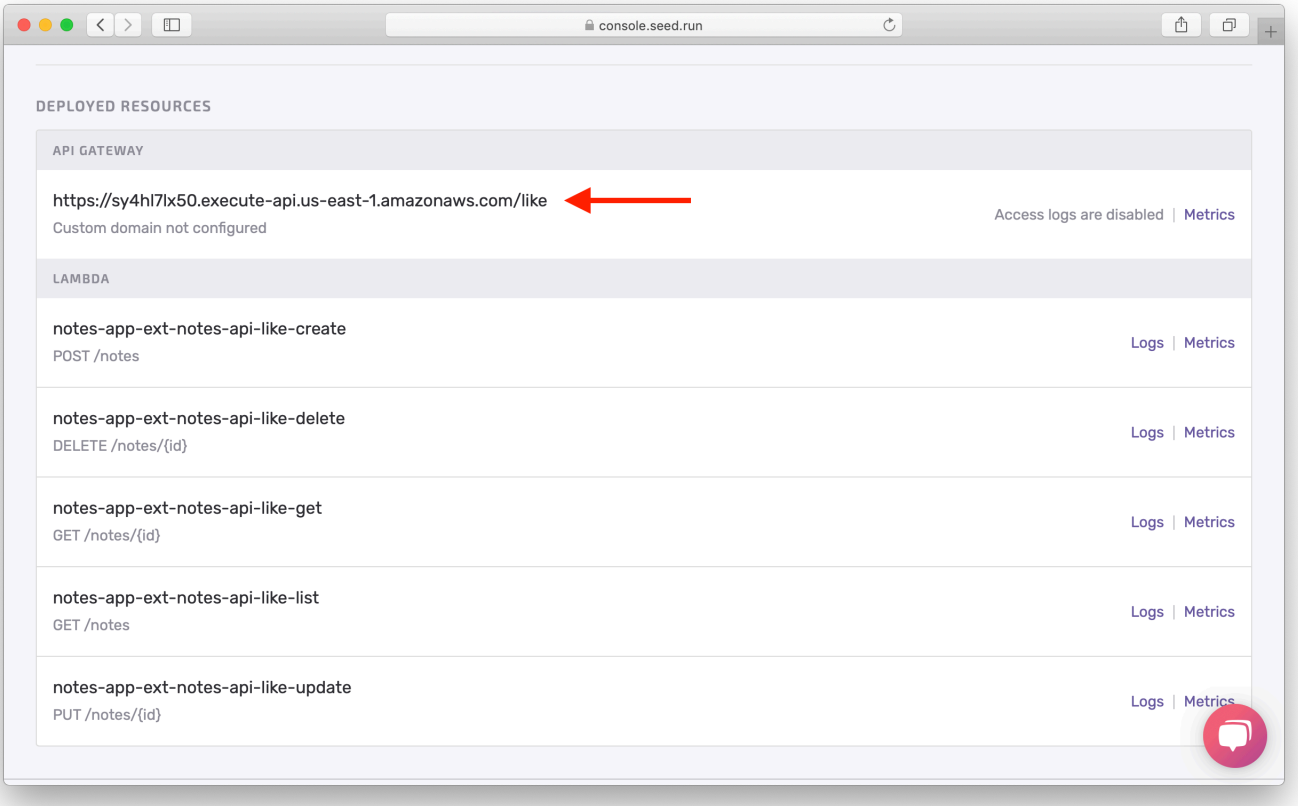
After the new stage successfully deploys, you can get the API endpoint in the stage's resources page. Click on the **like** stage.



Select **View Resources** for the **notes-api** service.



Scroll down and you will see the API Gateway endpoint for the **like** stage.



You can now use the endpoint in your frontend for further testing and development.

Now that our new feature environment has been created, let's quickly look at the flow for working on your new feature.

Working on new feature environments locally

Once the environment has been created, we want to continue working on the feature. A common problem people run into is that `serverless deploy` takes very long to execute. And running `serverless deploy` for every change just does not work.

Why is 'serverless deploy' slow?

When you run `serverless deploy`, Serverless Framework does two things:

1. Package the Lambda code into zip files.
2. Build a CloudFormation template with all the resources defined in `serverless.yml`.

The code is uploaded to S3 and the template is submitted to CloudFormation.

There are a couple of things that are causing the slowness here:

- When working on a feature, most of the changes are code changes. It is not necessary to rebuild and resubmit the CloudFormation template for every code change.
- When making a code change, a lot of the times you are only changing one Lambda function. In this case, it's not necessary to repackage the code for all Lambda functions in the service.

Deploying individual functions

Fortunately, there is a way to deploy individual functions using the `serverless deploy -f` command. Let's take a look at an example.

Say we change our new `like.js` code to:

```
import { success } from "../../libs/response-lib";

export async function main(event, context) {
  // business logic code for liking a post

  console.log("adding some debug code to test");
```

```
  return success({ status: true });
}
```

To deploy the code for this function, run:

```
$ cd services/like-api
$ serverless deploy -f like -s like
```

Deploying an individual function should be much quicker than deploying the entire stack.

Deploy multiple functions

Sometimes a code change can affect multiple functions at the same time. For example, if you changed a shared library, you have to redeploy all the services importing the library.

However, there isn't a convenient way to deploy multiple Lambda functions. If you can easily tell which Lambda functions are affected, deploy them individually. If there are many functions involved, run `serverless deploy -s like` to deploy all of them. Just to be on the safe side.

Now let's assume we are done working on our new feature and we want our team lead to review our code before we promote it to production. To do this we are going to create a pull request environment. Let's look at how to do that next.

For help and discussion

Comments on this chapter

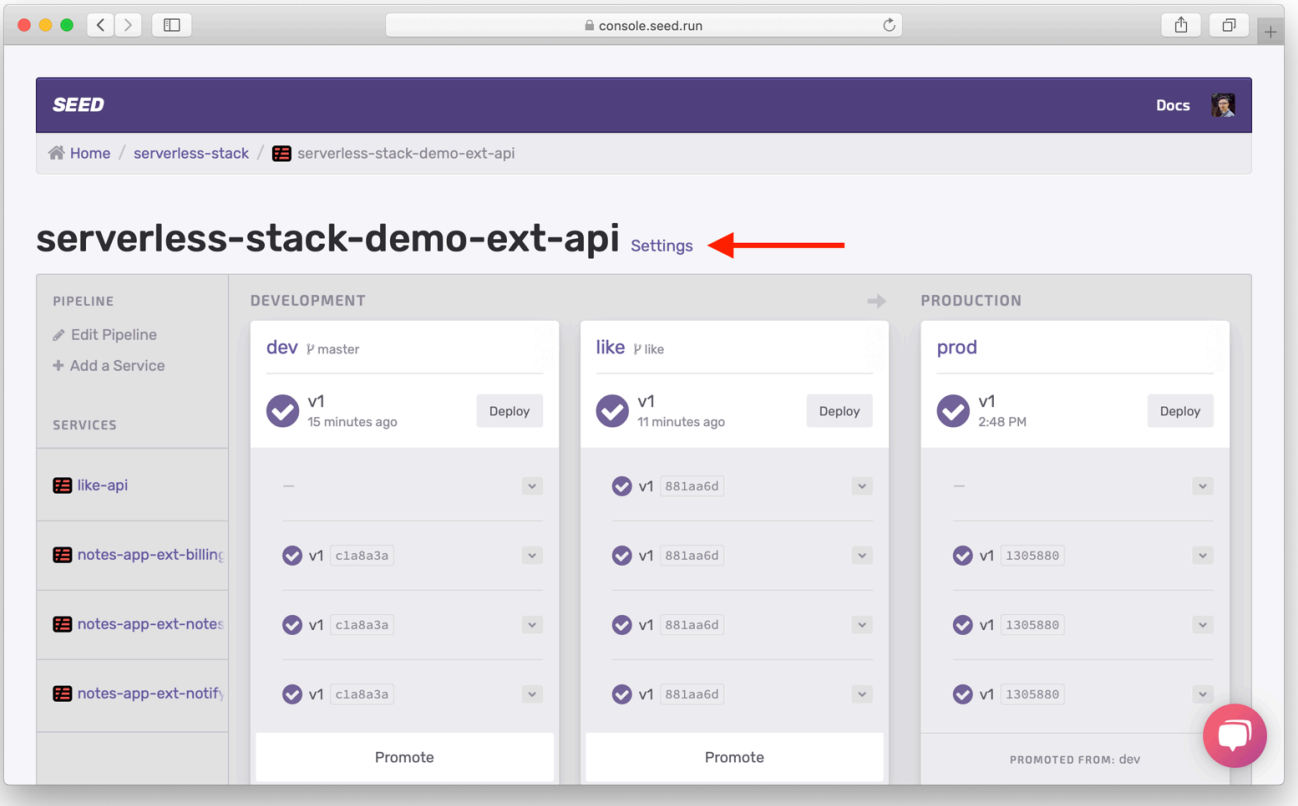
```
(https://discourse.serverless-
stack.com/t/creating-feature-
environments/1317)
```

Creating Pull Request Environments

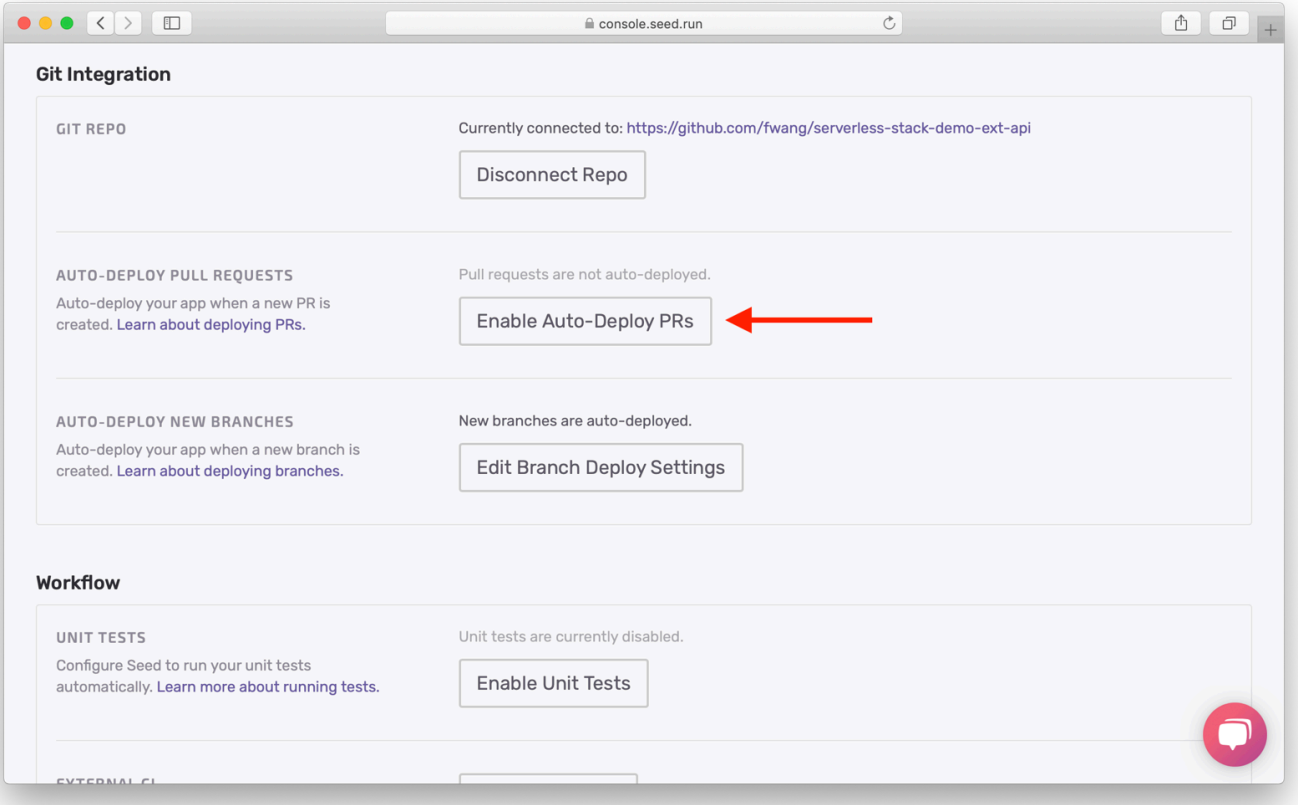
Now that we are done working on our new feature, we would like our team lead to review our work before promoting it to production. To do that we are going to create a pull request and Seed will automatically create an ephemeral environment for it.

Enable pull request workflow on Seed

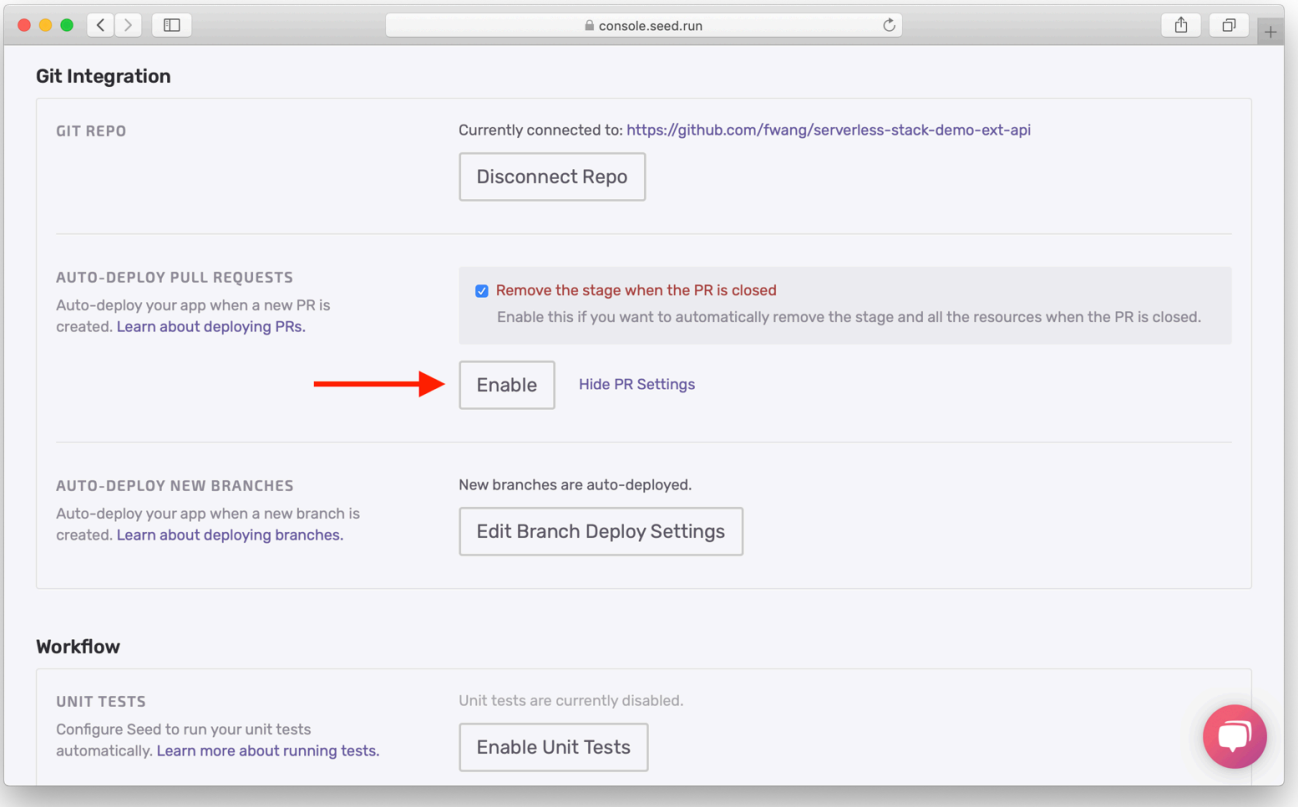
To enable auto-deploying pull requests, head over to your app on Seed. Click **Settings**.



Scroll down to **Git Integration**. Then click **Enable Auto-Deploy PRs**.

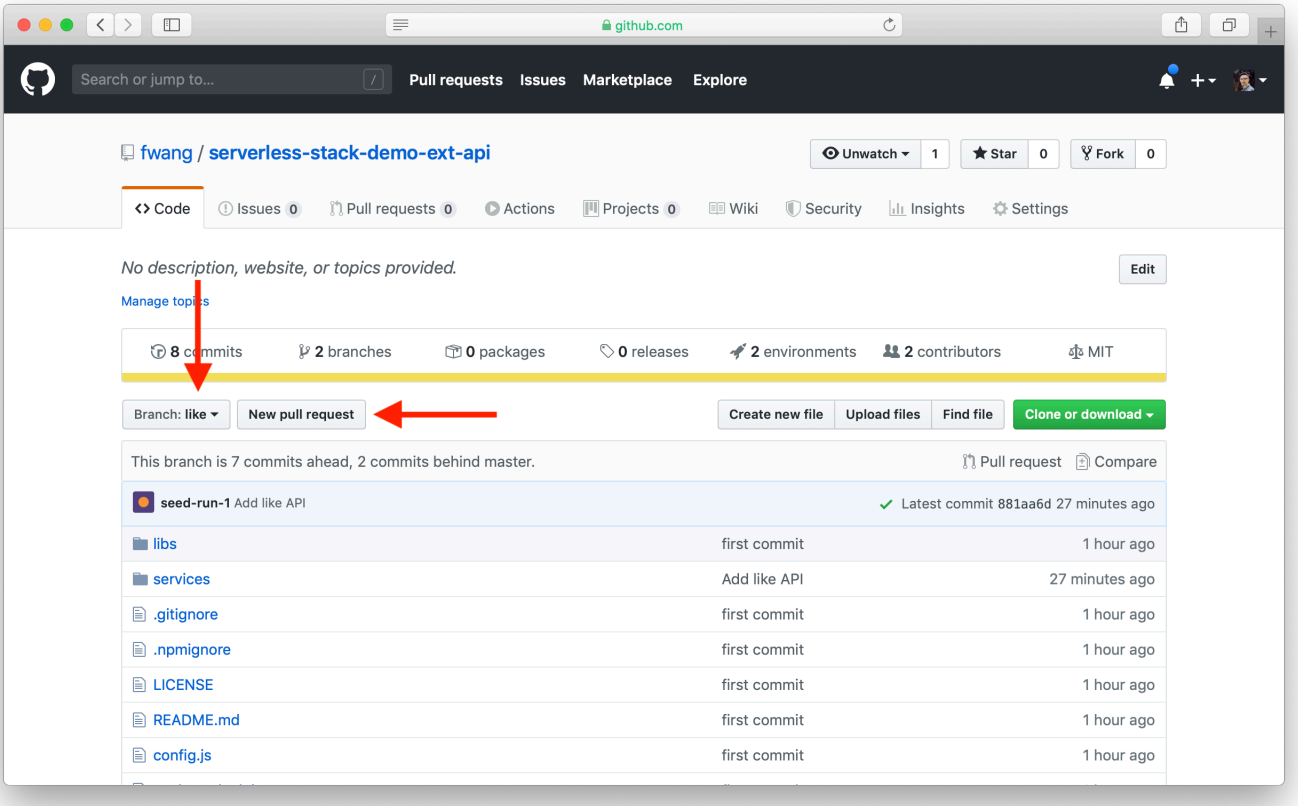


Hit **Enable**.

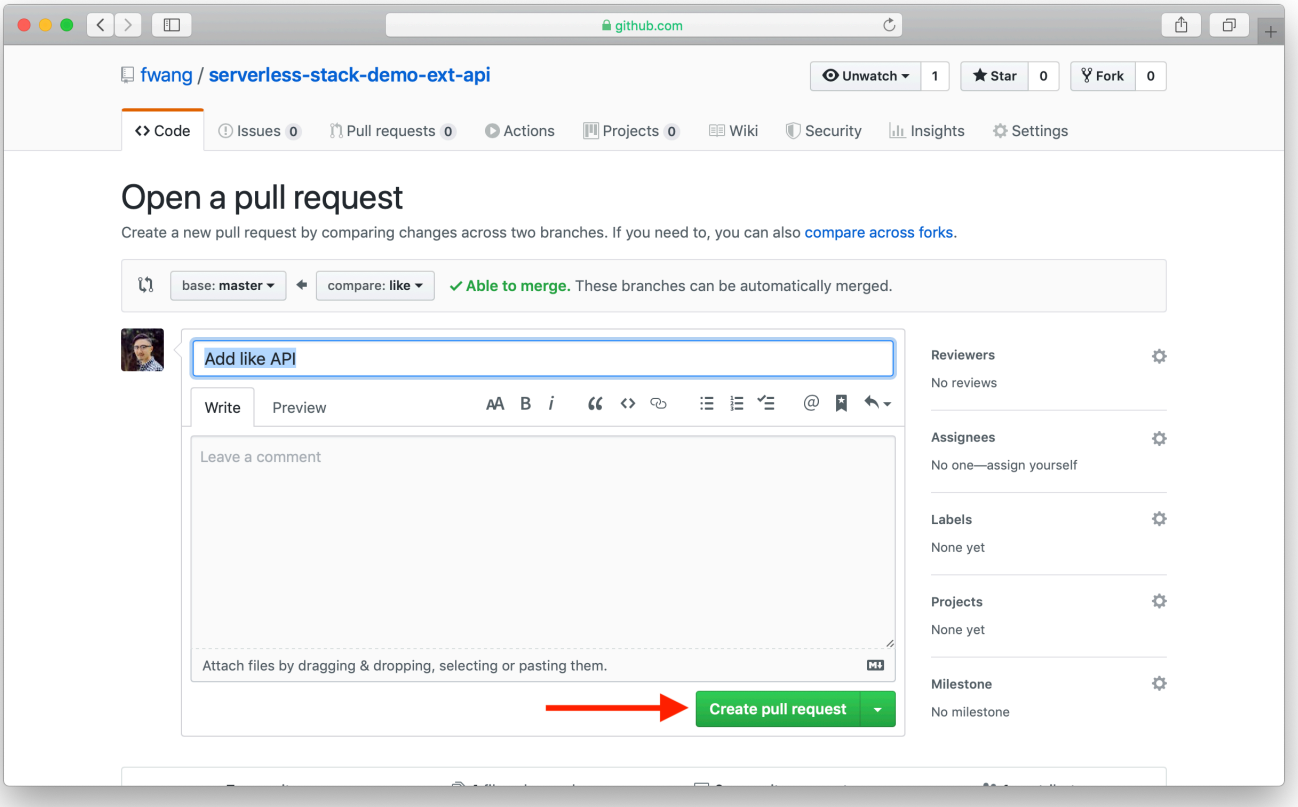


Create a pull request

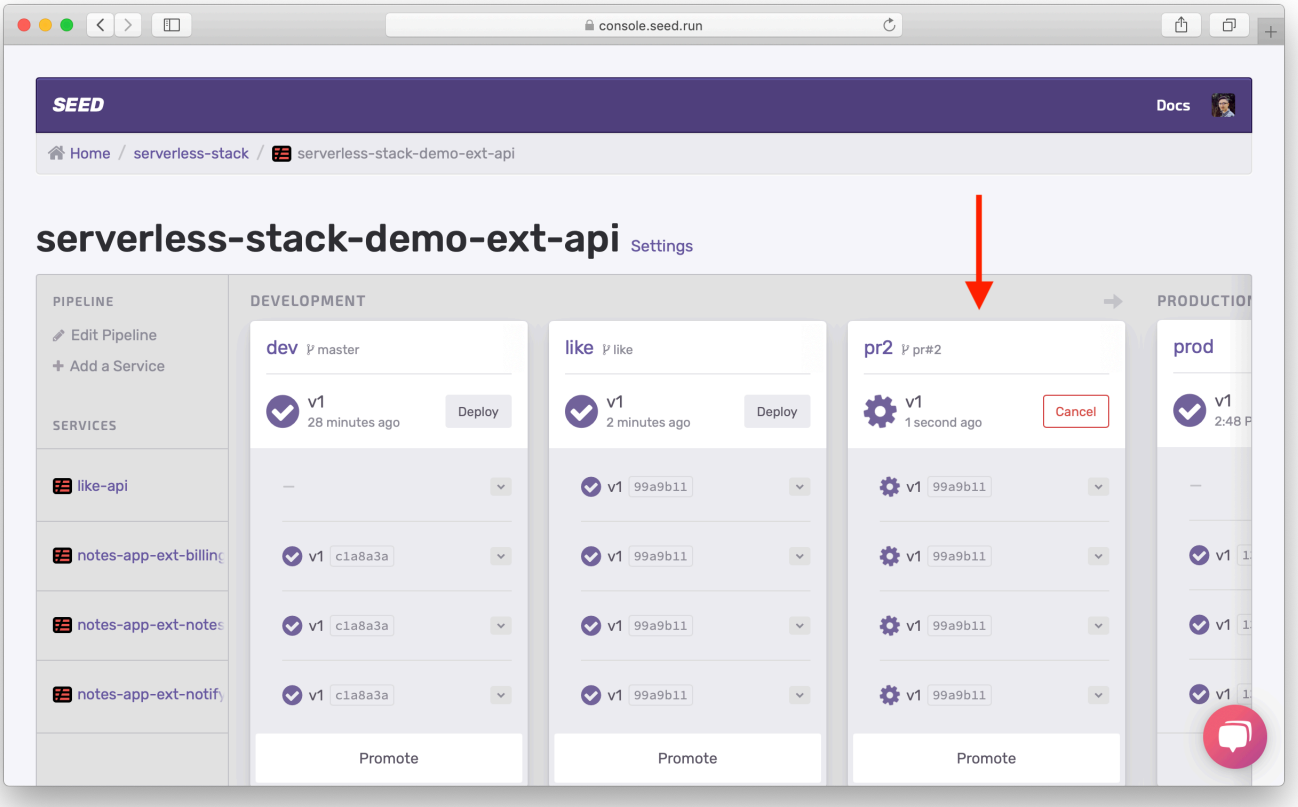
Go to GitHub, and select the **like** branch. Then hit **New pull request**.



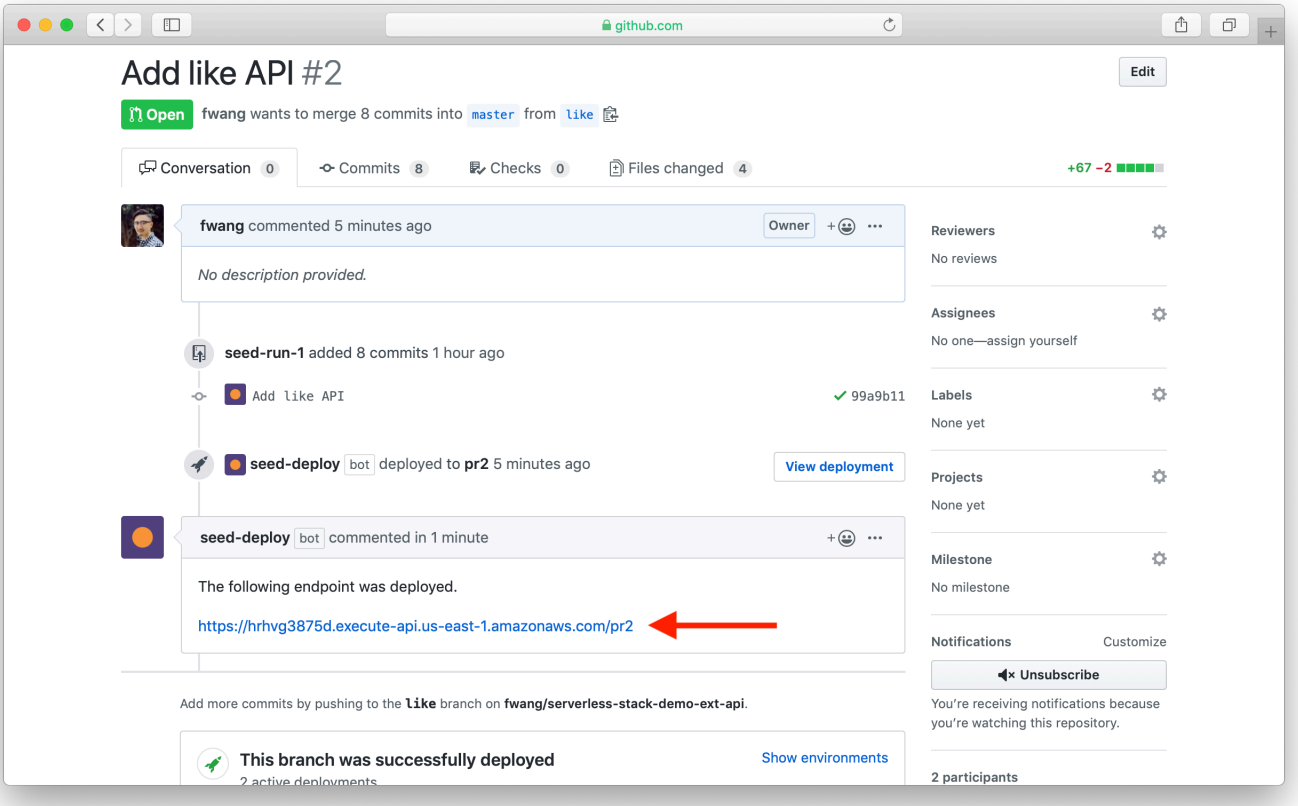
Click **Create pull request.**



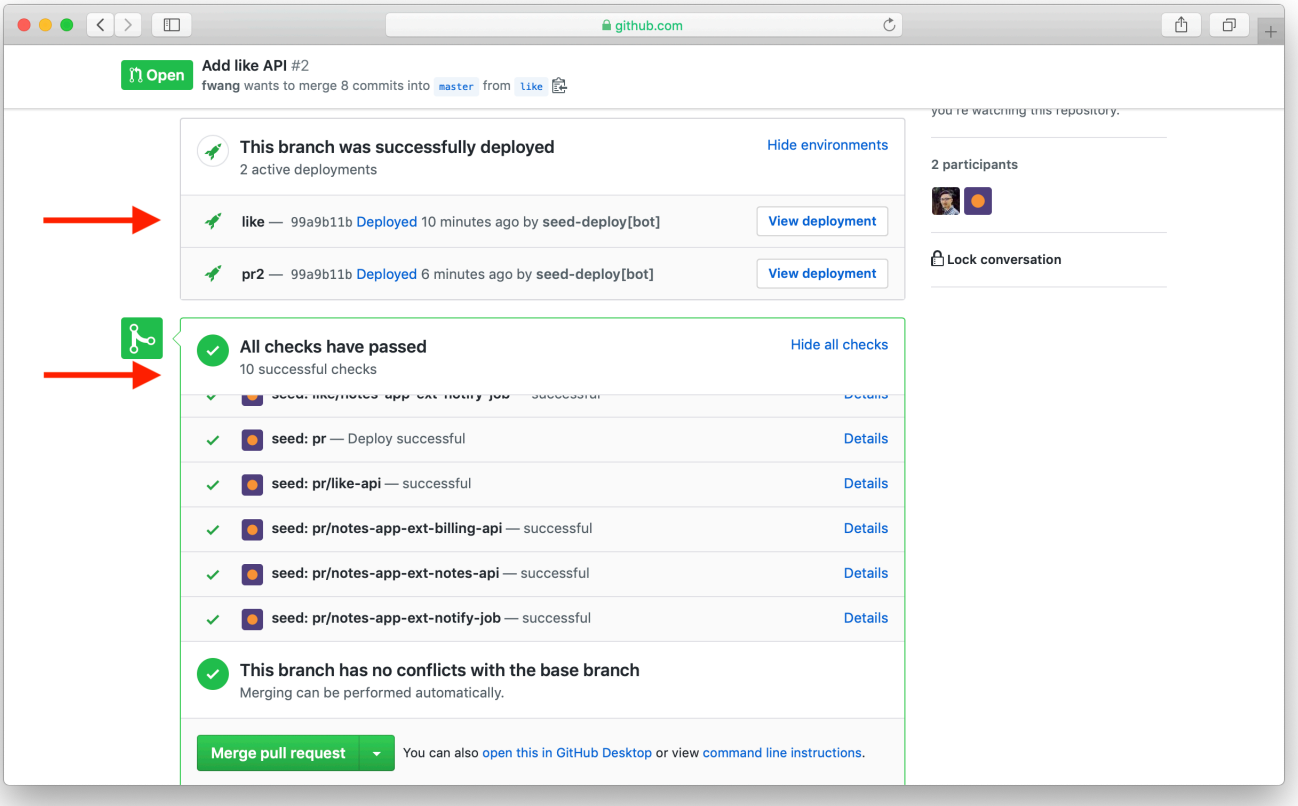
Now back in Seed, a new stage (in this case **pr2**) should be created and is being deployed automatically.



After the **pr2** stage successfully deploys, you can see the deployed API endpoint on the PR page. You can give the endpoint to your frontend team for testing.



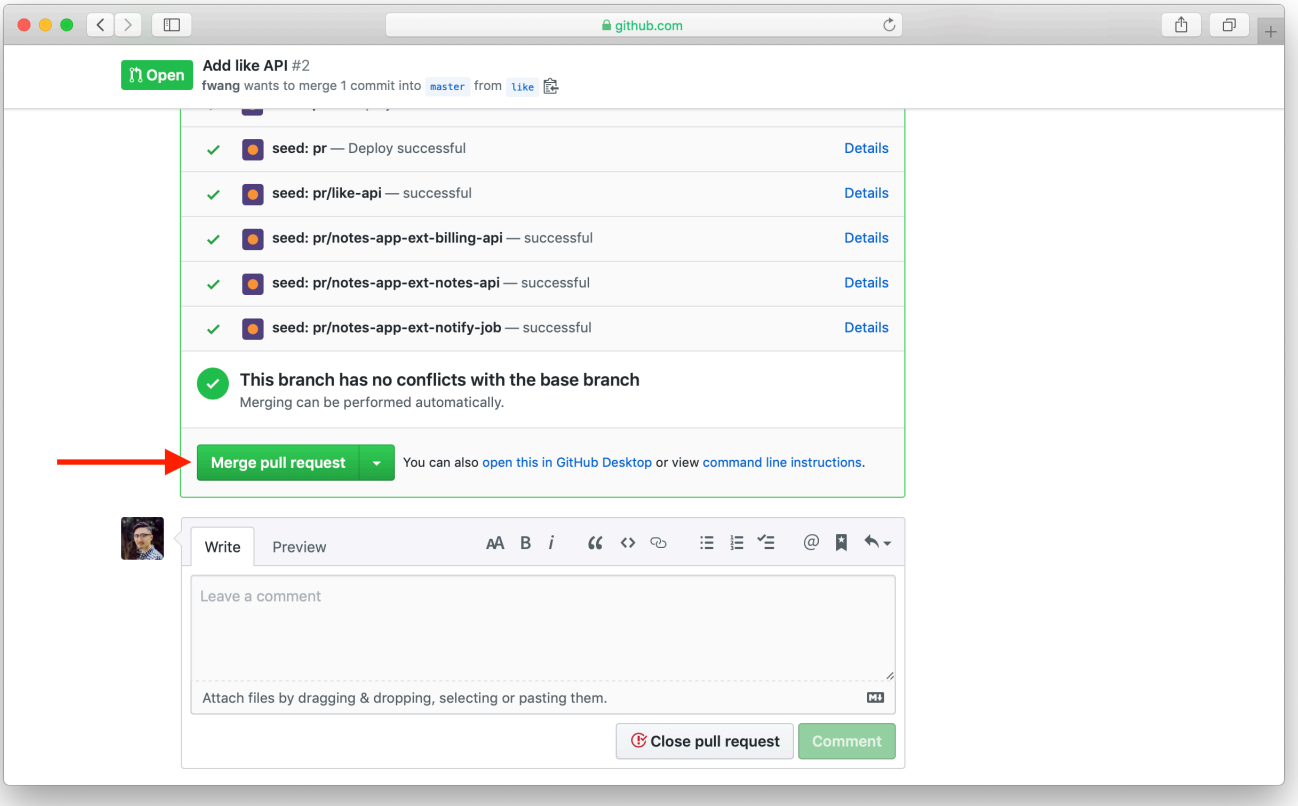
You can also access the **pr2** stage and the upstream **like** stage on Seed via the **View deployment** button. And you can see the deployment status for each service under the **checks** section.



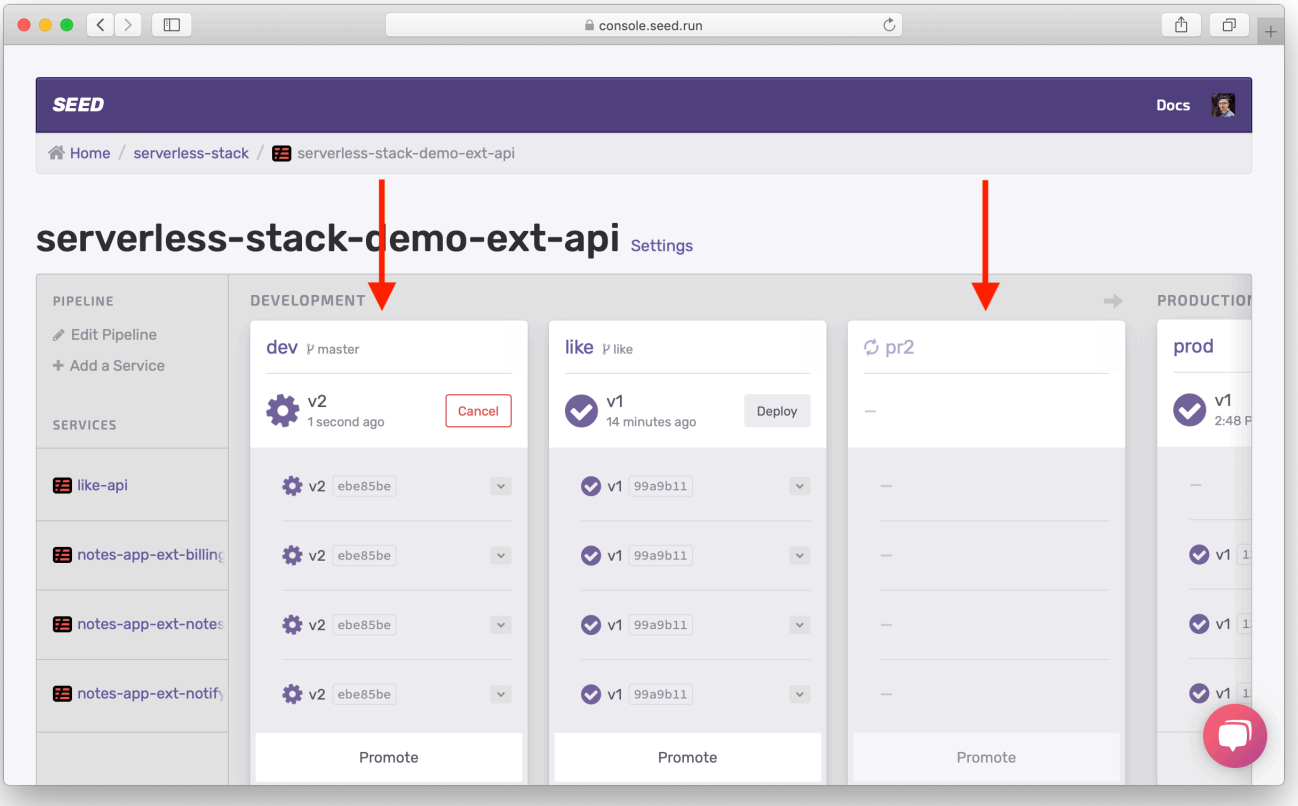
Now that our new feature has been reviewed, we are ready to merge it to master.

Merge to master

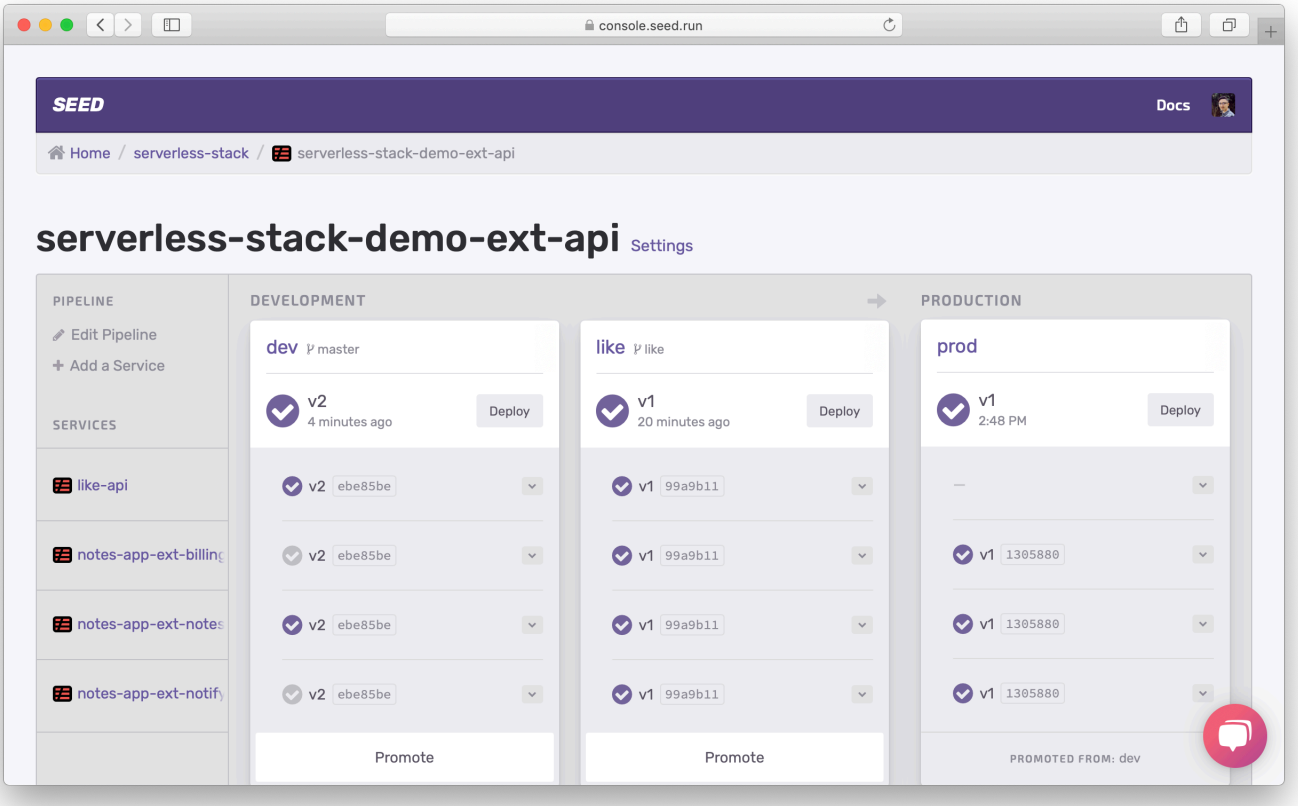
Once your final test looks good, you are ready to merge the pull request. Go to GitHub's pr page and click **Merge pull request**.



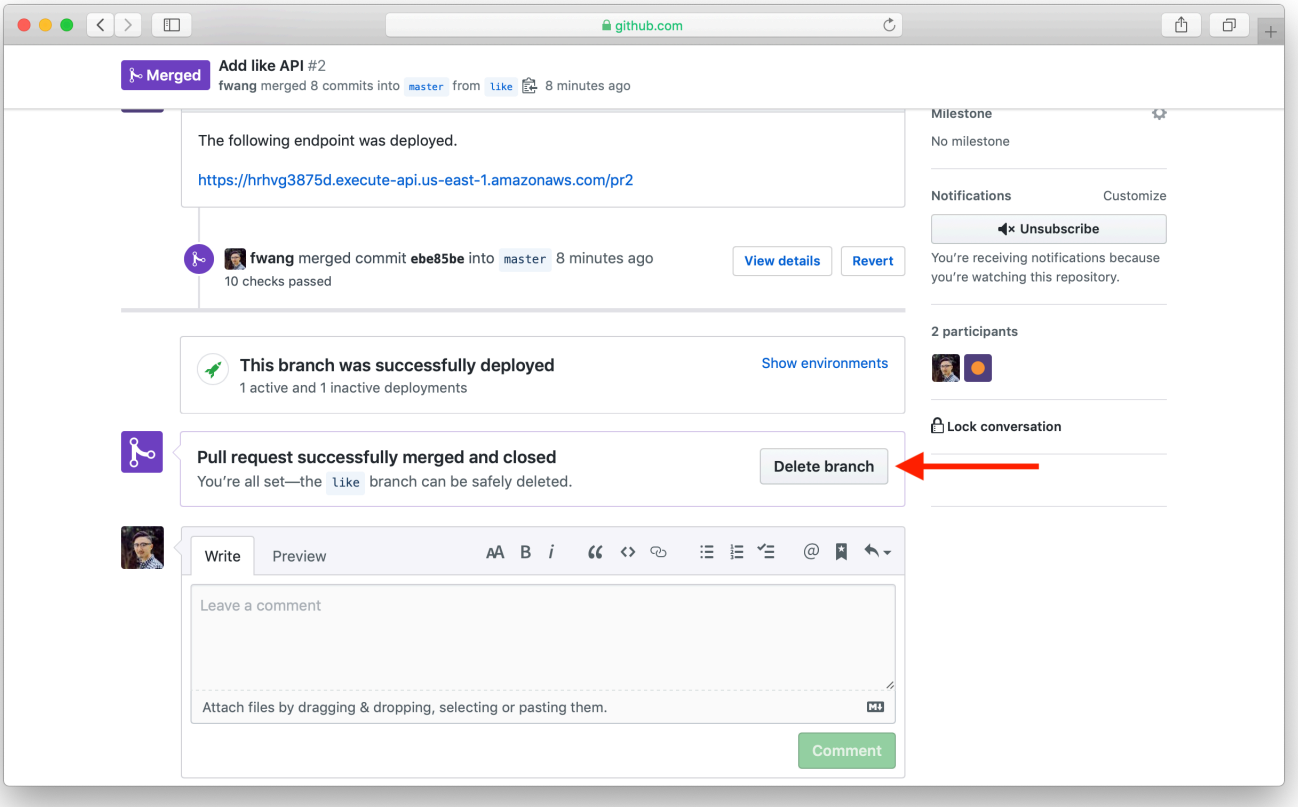
Back in Seed, this will trigger a deployment in the **dev** stage automatically, since the stage auto-deploys changes in the **master** branch. Also, since merging the pull request closes it, this will automatically remove the **pr2** stage.



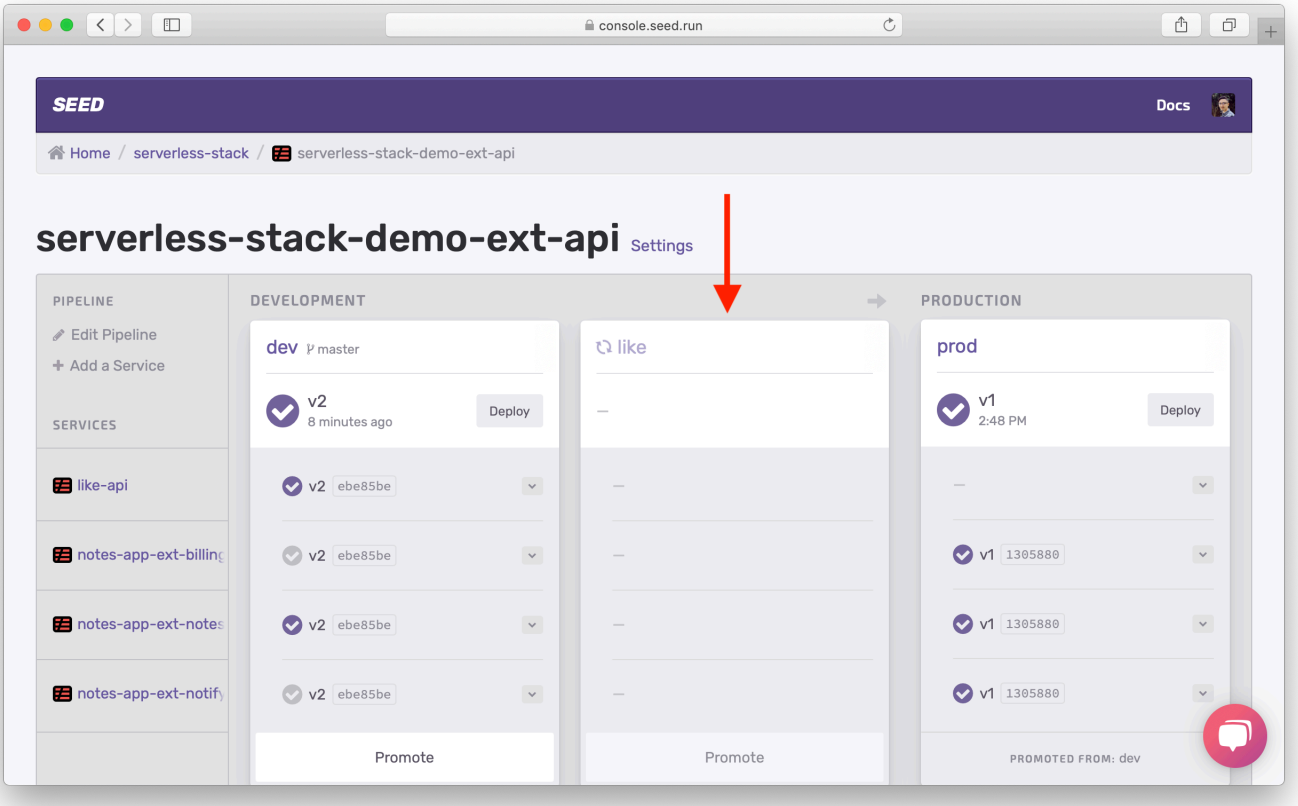
After the deployment completes and the **pr2** stage is removed, this is what your pipeline should look like:



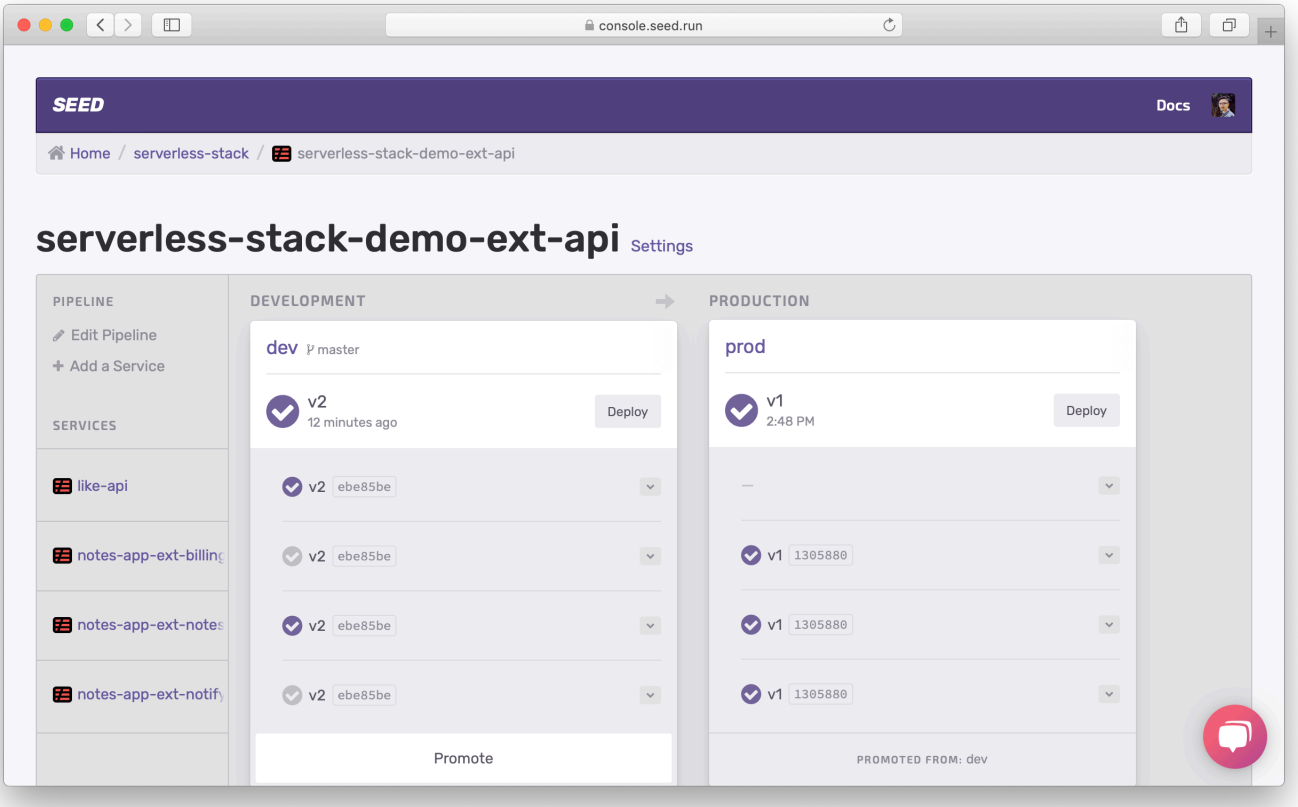
From GitHub's pull request screen, we can remove the **like** branch.



Back in Seed, this will trigger the **like** stage to be automatically removed.



After the removal is completed, your pipeline should now look like this.



Next, we are ready to promote our new feature to production.

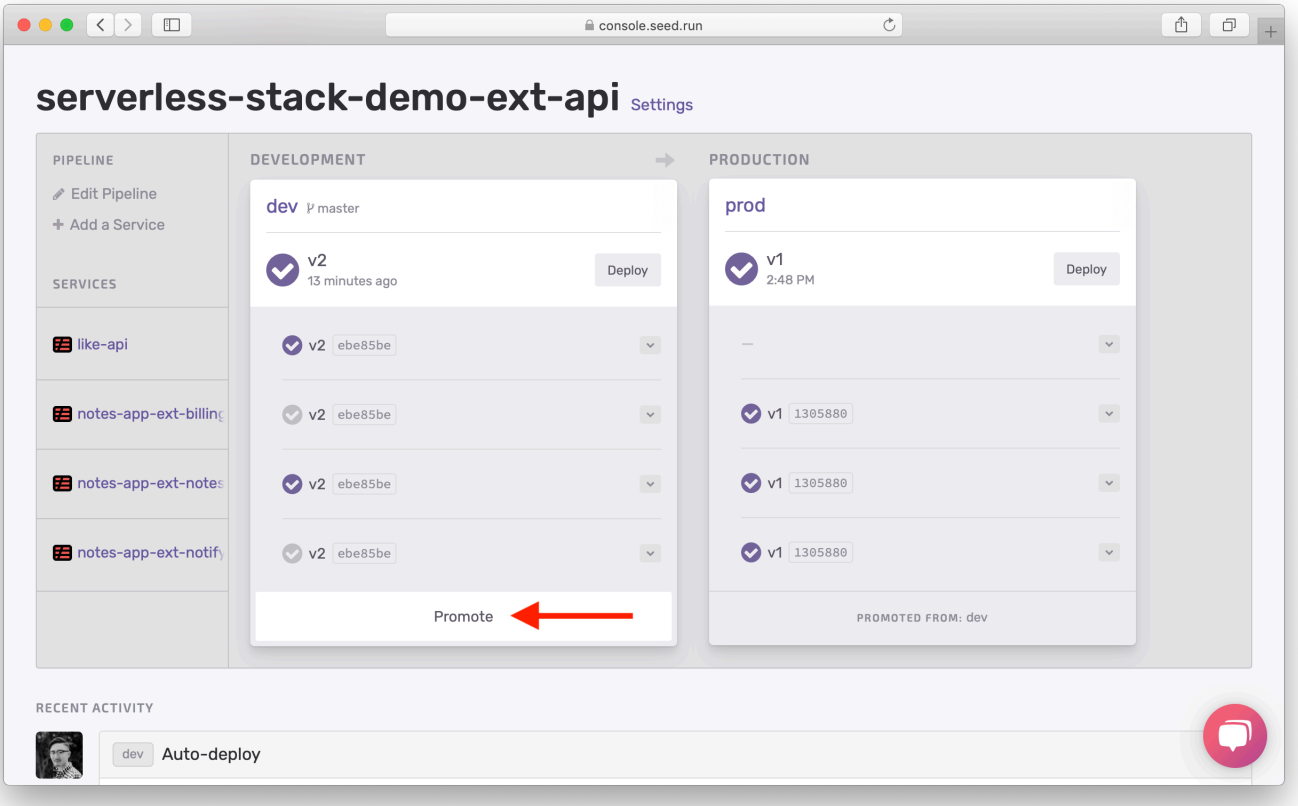
For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/creating-pull-request-environments/1318)

Promoting to Production

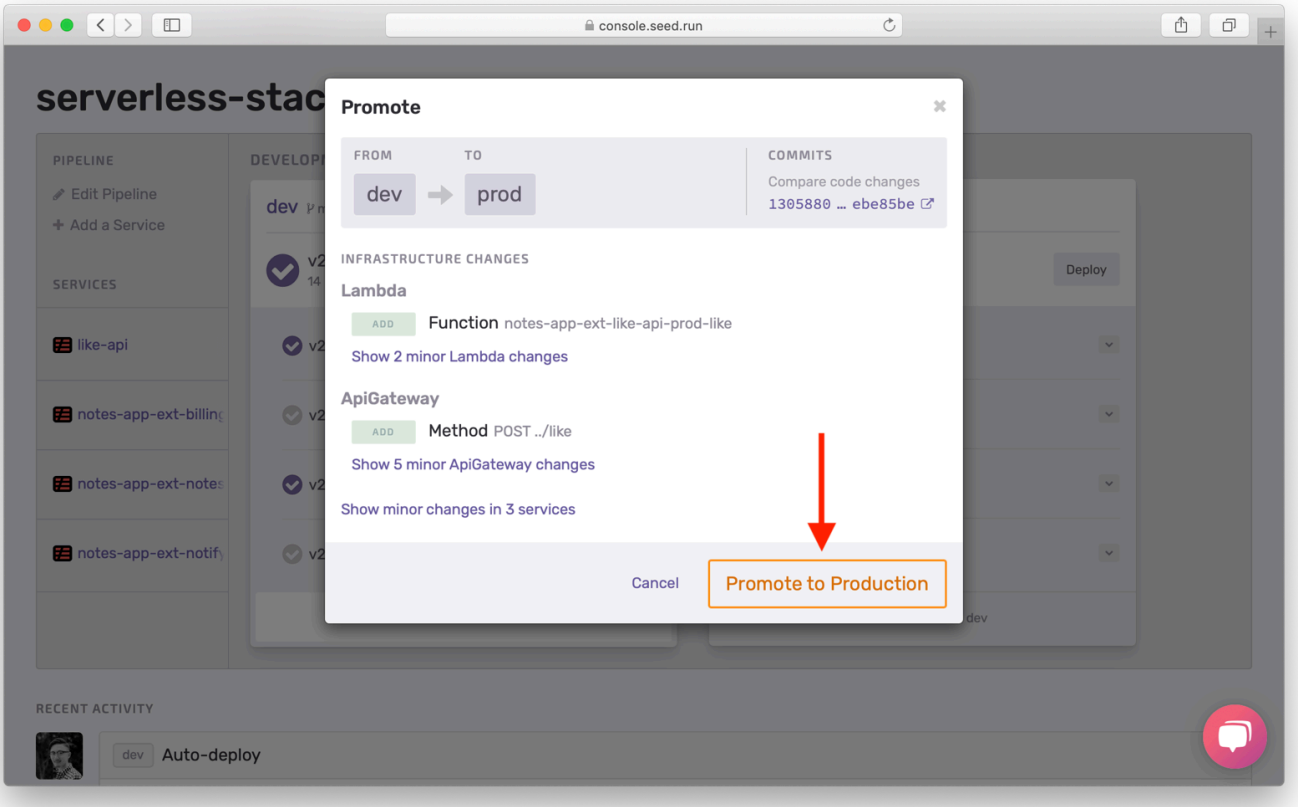
Now that our new feature has been tested and merged to master, we are ready to promote it to production. We are going to do so by promoting our `dev` stage to `prod`.

Head over to Seed. And then hit **Promote** at the bottom of the `dev` stage.

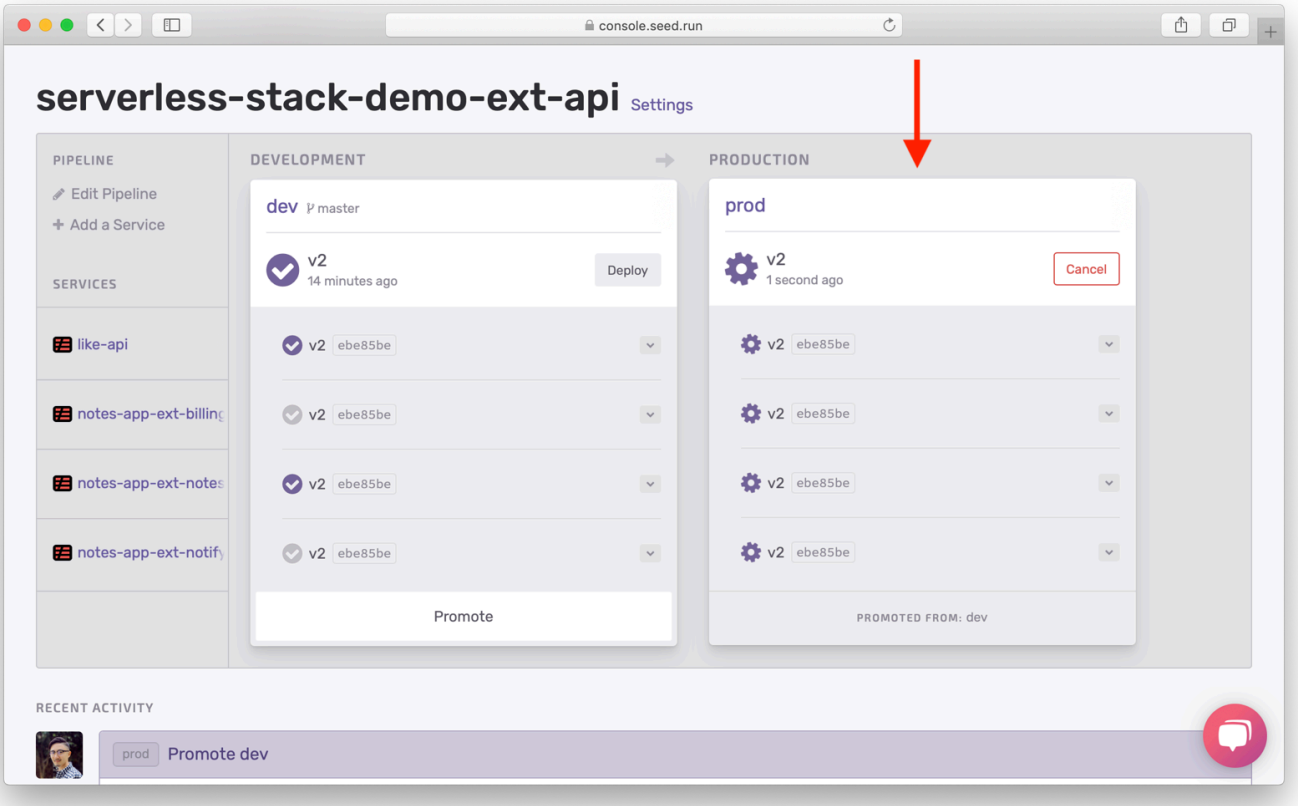


You will see a list of changes. Note, only the major changes are shown here. The change list shows that we added a Lambda functions and an API Gateway method. A couple of other minor resources like the Lambda execution IAM role and Lambda's CloudWatch log group were also added but those are hidden by default.

Hit **Promote to Production**.



This will trigger the `prod` stage to start building.



Why manual promote?

In a traditional monolithic application (non-Serverless) development, your code mostly contains application logic. Application logic can be rolled back relatively easily and is usually side-effect free.

Serverless apps adopt the infrastructure as code pattern (/chapters/what-is-infrastructure-as-code.html), and your infrastructure definition (`serverless.yml`) sits in your codebase. When your Serverless app is deployed, the code is updated, and the infrastructure changes are applied. A typo in your `serverless.yml` could remove your resources. And in the case of a database resource, this could result in permanent data loss.

To avoid these issues in the first place, it's recommended that you have a step to review your infrastructure changes before they get promoted to production.

What is a change set?

CloudFormation provides a feature called Change Sets (https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-updating-stacks-changesets.html). You give CloudFormation the new template you are going to deploy into a stack, and CloudFormation will show you the resources that are going to be added, modified, and removed. You can think of it as a dry run.

We recommend generating CloudFormation Change Sets as a part of the manual approval step in your CI/CD pipeline. This will let you review the exact infrastructure changes that are going to be applied. We think this extra step can really help prevent any irreversible infrastructure changes from being deployed to your production environment.

However, CloudFormation templates and Change Sets can be pretty hard to read. Here is where Serverless Framework does a really good job of allowing you to provision a Lambda and API resources in a simple and compact syntax. Behind the scene, a great number of resources are provisioned: Lambda roles, Lambda versions, Lambda log groups, API Gateway resource, API Gateway method, API Gateway deployment, just to name a few. However when CloudFormation shows you a list of changes with these resources (usually with cryptic names), it obscures the actual changes that you need to be paying attention to. This is why with Seed we've taken extra care to improve on the CloudFormation Change Set. We do this by showing changes that are relevant to us as developers and highlight the changes that need extra attention.

Next, let's look at the scenario where you might end up having to rollback your code.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/promoting-to-production/1330)

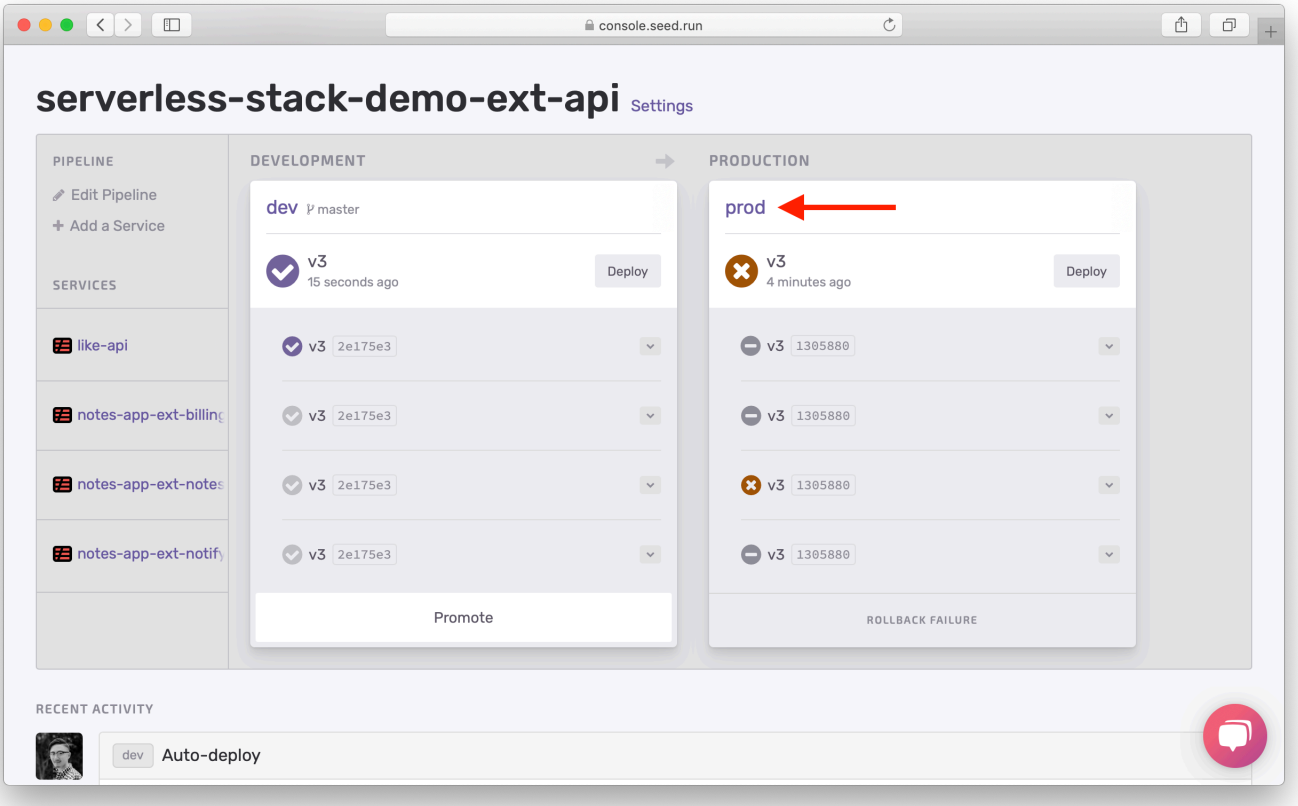
Rollback Changes

So we've worked on a new feature, deployed it to a feature branch, created a PR for it, merged it to master, and promoted it to production! We are almost done going over the workflow. But before we move on we want to make sure that you are able to rollback your Serverless deployments in case there is a problem. We think this is a critical aspect of your CI/CD pipeline. In this chapter we'll look at what the right rollback strategy is for your Serverless apps.

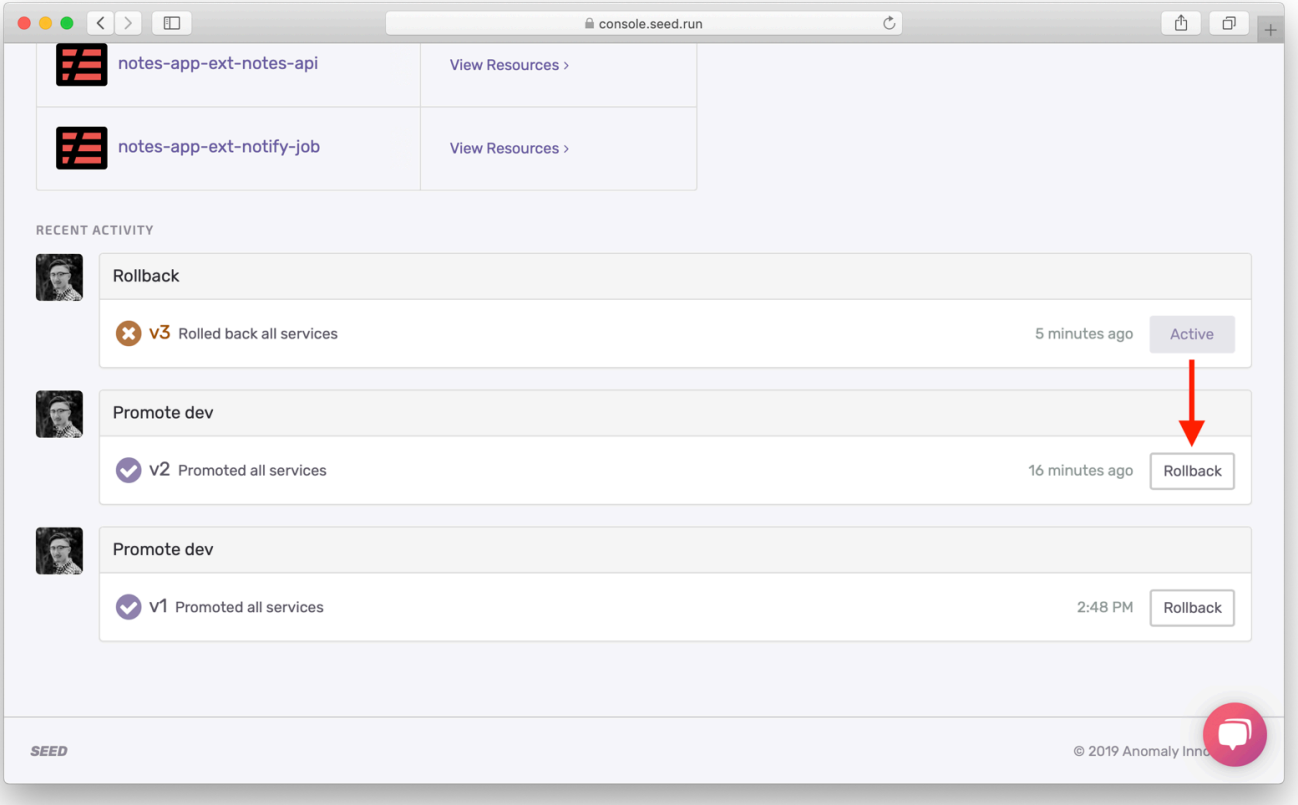
Let's quickly look at how to do that in Seed.

Rollback to previous build

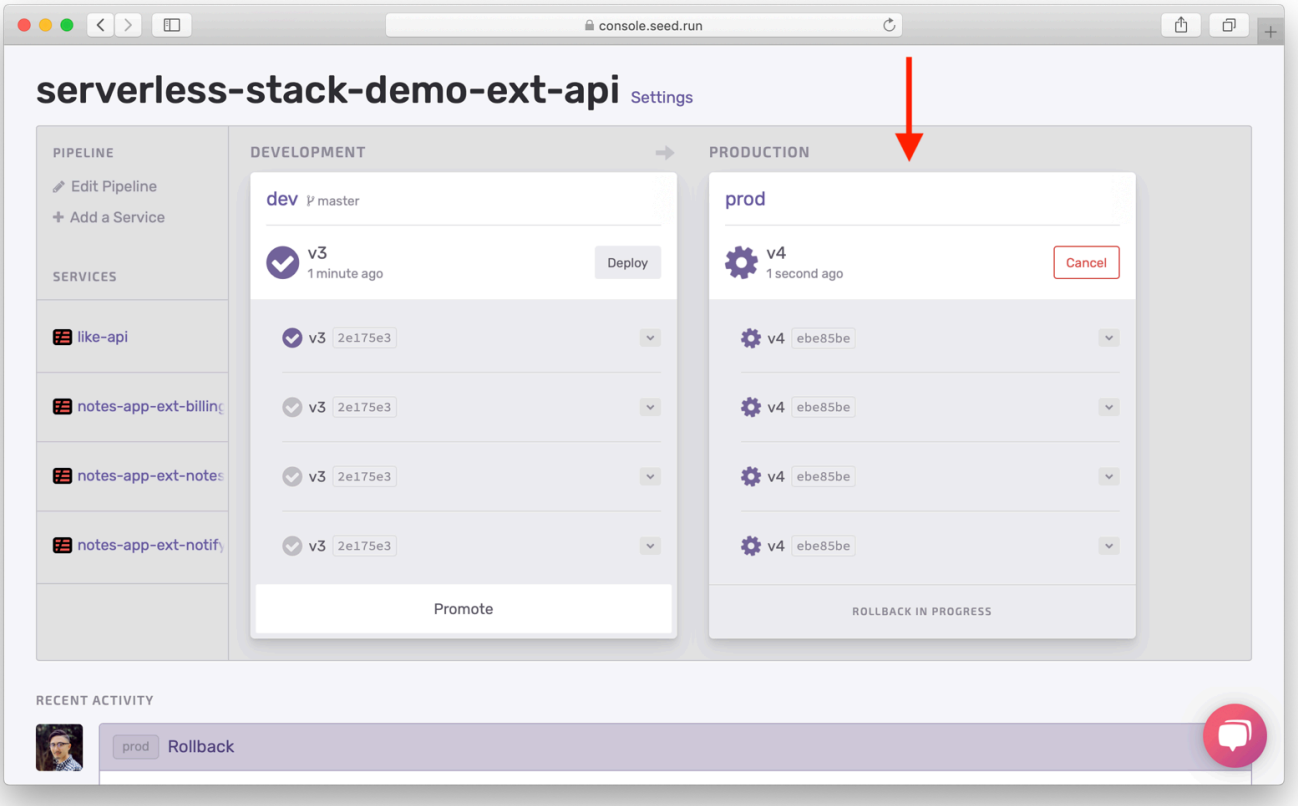
To rollback to a previous build, go to your app in Seed. Let's suppose we've pushed some faulty code to `prod` stage. Click on the `prod` stage to see a list of historical builds in the stage.



Pick an older successful build and hit **Rollback**.



Notice a new build is triggered for the `prod` stage.



Rollback infrastructure change

In our monorepo setup, our app is made up of multiple services, and some services are dependent on each other. These dependencies require the services to be deployed in a specific order. Previously, we talked about how to deploy services with dependencies (/chapters/deploy-a-serverless-app-with-dependencies.html). We also need to watch out for the deployment order when rolling back a change.

Let's consider a simple example with just two services, `billing-api` and `notify-job`. Where `billing-api` exports an SNS topic named `note-purchased`. Here is an example of `billing-api`'s `serverless.yml`:

```
Outputs:
  NotePurchasedTopicArn:
    Value:
      Ref: NotePurchasedTopic
    Export:
      Name: ExtNotePurchasedTopicArn-
${self:custom.stage}
```

And the `notify-job` service imports the topic and uses it to trigger the `notify` function:

```
functions:
  notify:
    handler: notify.main
    events:
      - sns:
        'Fn::ImportValue': ExtNotePurchasedTopicArn-
${self:custom.stage}
```

Note that the `billing-api` service had to be deployed first. This is to make sure that the export value `ExtNotePurchasedTopicArn` has first been created. Then we can deploy the `notify-job` service.

Assume that after the services have been deployed, you push a faulty commit and you have to rollback.

In this case, you need to: **rollback the services in the reverse order of the deployment**.

Meaning `notify-job` needs to be rolled back first, such that the exported value `ExtNotePurchasedTopicArn` is not used by other

services, and then rollback the `billing-api` service to remove the SNS topic along with the export.

Next we are going to look at an optimization that you can make to speed up your builds.

For help and discussion

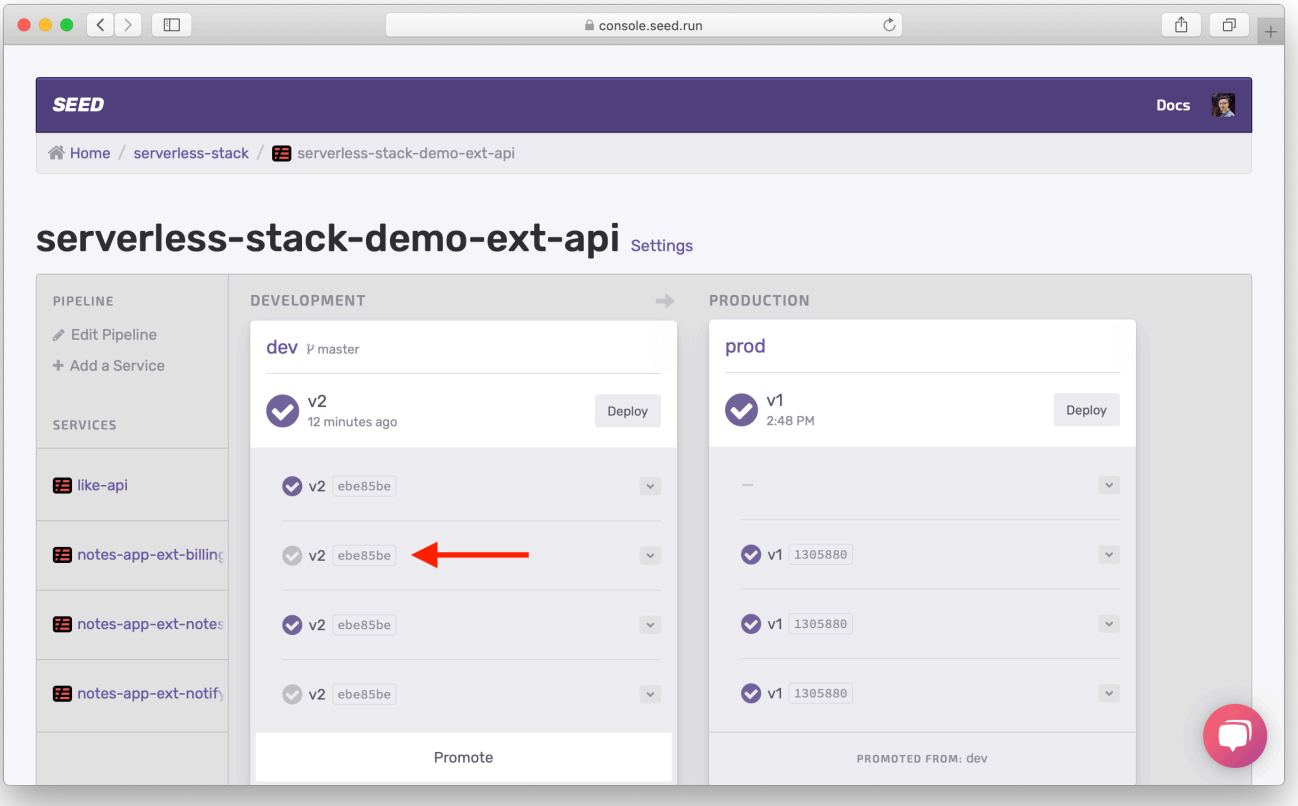
Comments on this chapter (https://discourse.serverless-stack.com/t/rollback-changes/1331)

Deploying Only Updated Services

Once you are repeatedly deploying your Serverless application, you might notice that the Serverless deployments are not very fast. This is especially true if your app has a ton of service. There are a couple of things you can do here to speed up your builds. One of them is to only deploy the services that've been updated.

In this chapter we'll look at how to do that.

Note that, we are doing this by default in Seed. Recall that when we merged the **like** branch to the **master** branch, only the `like-api` service and the `notes-api` showed a solid check. The other two services showed a greyed out check mark. This means that there were no changes to be deployed for this service.



In a Serverless app with a single service, the deployment strategy in your CI/CD pipeline is straight forward: deploy my app on every git push.

However in a monorepo setup, an app is made up of many Serverless Framework (https://serverless.com/) services. It's not uncommon for teams to have apps with over 40 services in a single repo on Seed (https://seed.run/). For these setups, it does not make sense to deploy all the services when all you are trying to do is fix a typo! You only want to deploy the services that have been updated because deploying all your services on every commit is:

1. Slow: deploying all services can take very long, especially when you are not deploying them concurrently.
2. Expensive: traditional CI services charge extra for concurrency. As of writing this chapter, it costs $50 for each added level of concurrency on CircleCI (https://circleci.com/), hence it can be very costly to deploy all your services concurrently.

There are a couple of ways to only deploy the services that have been updated in your Serverless CI/CD pipeline.

Strategy 1: Skip deployments in Serverless Framework

The `serverless deploy` command has built-in support to skip a deployment if the deployment package has not changed.

A bit of a background, when you run `serverless deploy`, two things are done behind the scenes. It first does a `serverless package` to generate a deployment package. This includes the CloudFormation template and the zipped Lambda code. Next, it does a `serverless deploy -p path/to/package` to deploy the package that was created. Before Serverless deploys the package in the second step, it first computes the hash of the package and compares it with that of the previous deployment. If the hash is the same, the deployment is skipped. We are simplifying the process here but that's the basic idea.

However, there are two downsides to this.

1. Serverless still has to generate the deployment package first. For a Node.js application, this could mean installing the dependencies, linting, running Webpack, and finally packaging the code. Meaning that the entire process can still be pretty slow even if you skip the deployment.
2. If your previous deployment had failed due to an external cause, after you fix the issue and re-run `serverless deploy`, the deployment will be skipped. For example, you tried to create an S3 bucket in your `serverless.yml`, but you hit the 100 S3 buckets per account

limit. You talked to AWS support and had the limit lifted. Now you re-run `serverless deploy`, but since neither the CloudFormation template or the Lambda code changed, the deployment will be skipped. To fix this, you need to use the `--force` flag to skip the check and force a deployment.

Strategy 2: Check the Git log for changes

A better approach here is to check if there are any commits in a service directory before deploying that service.

When some code is pushed, you can run the following command to get a list of updated files:

```
$ git diff --name-only ${prevCommitSHA}
${currentCommitSHA}
```

This will give you a list of files that have changed between the two commits. With the list of changed files, there are three scenarios from the perspective of a given service. We are going to use `notes-api` as an example:

1. A file was changed in my service directory (ie. `services/notes-api`) ⇒ we deploy the `notes-api` service
2. A file was changed in another service's directory (ie. `services/like-api`) ⇒ we do not deploy the `notes-api` service
3. Or, a file was changed in `libs/` ⇒ we deploy the `notes-api` service

Your repo setup can look different, but the general concept still holds true.

You have to figure out which file change affects an individual service, and which affects all the services. The advantage of this strategy is that you know upfront which services can be skipped, allowing you to skip a portion of the entire build process!

And this concludes our section on the development workflow! Next, we'll look at how to trace our Serverless applications using AWS X-Ray.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploying-only-updated-services/1321)

Tracing Serverless Apps with X-Ray

This chapter is based on a blog post over on the Seed blog (https://seed.run/blog/) — www.seed.run/blog/how-to-trace-serverless-apps-with-aws-x-ray (https://seed.run/blog/how-to-trace-serverless-apps-with-aws-x-ray).

Typically as a Serverless app grows, the number of AWS services involved also increases. This can make it tricky to debug them. AWS X-Ray (https://aws.amazon.com/xray/) is a service that records and visualizes requests made by your application. It provides an end-to-end view of requests as they travel through your Serverless application, and shows a map of your application's underlying components.

In this chapter we'll show you how to set up AWS X-Ray to trace API requests and Lambda invocations for your Serverless Framework application.

Enable X-Ray tracing for API Gateway and Lambda

First let's start by enabling X-Ray for your application.

Open your `serverless.yml` and add a `tracing` config inside the `provider` section:

```
provider:
  ...
  tracing:
    apiGateway: true
    lambda: true
```

Then add the IAM permissions required for Lambda to write to X-Ray under `iamRoleStatements` inside the `provider` section:

```
provider:
  ...
  iamRoleStatements:
    - Effect: Allow
      Action:
        ...
        - xray:PutTraceSegments
        - xray:PutTelemetryRecords
      Resource: "*"
```

Let's use the following Lambda function as an example.

```
const AWS = require('aws-sdk');
const dynamodb = new AWS.DynamoDB.DocumentClient();
const sns = new AWS.SNS();

exports.main = async function(event) {
  await dynamodb.get({
    TableName: 'notes',
```

```
    Key: { noteId: 'note1' },
  }).promise();

  await sns.publish({
    Message           : 'test',
    TopicArn          : 'arn:aws:sns:us-east-
1:113345762000:test-topic',
  }).promise();


  return { statusCode: 200, body: 'successful' };
}
```

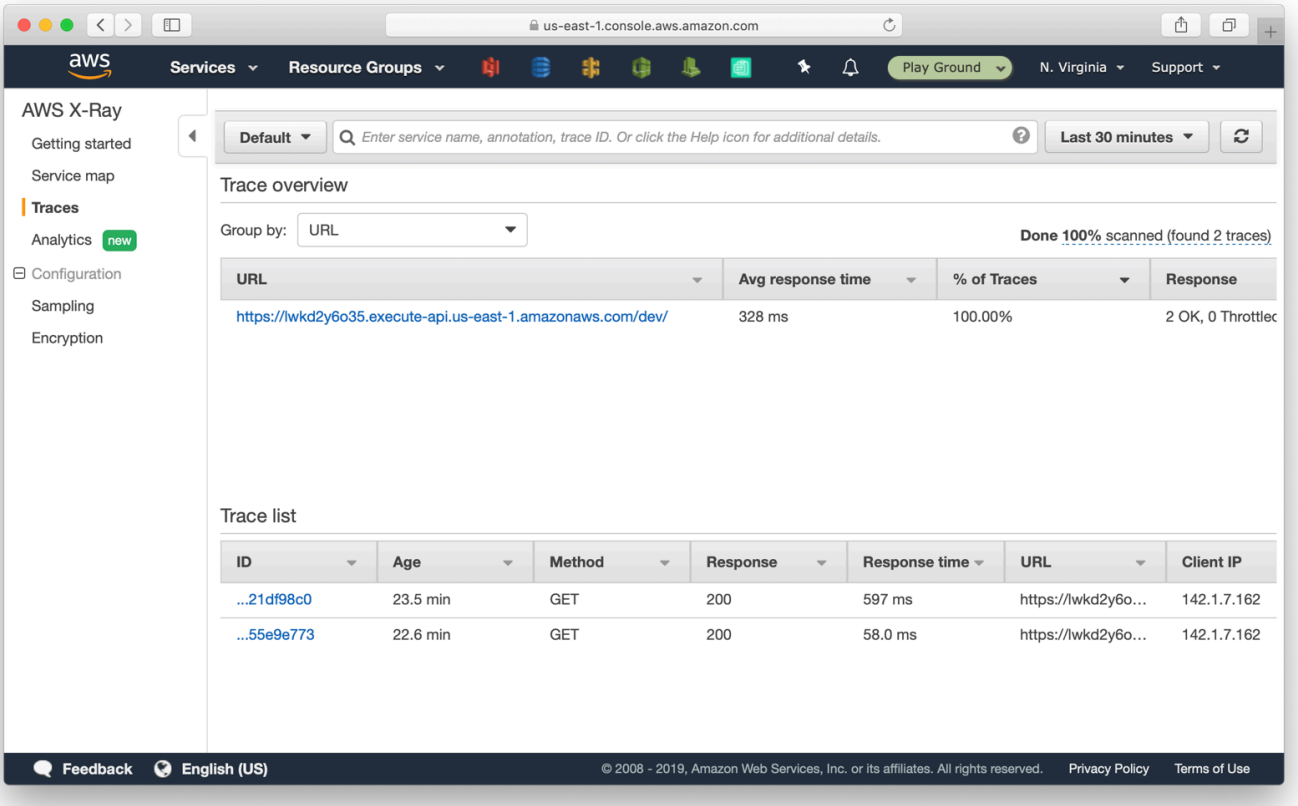
Now run `serverless deploy` to deploy your service. Make sure to deploy your entire application (not just an individual function), since you made changes to your `serverless.yml`.

Note that, if you are trying to enable AWS X-Ray Tracing on existing Serverless projects, make sure your Serverless CLI version is later than **1.44**.

After you deploy, invoke your API Gateway endpoint:

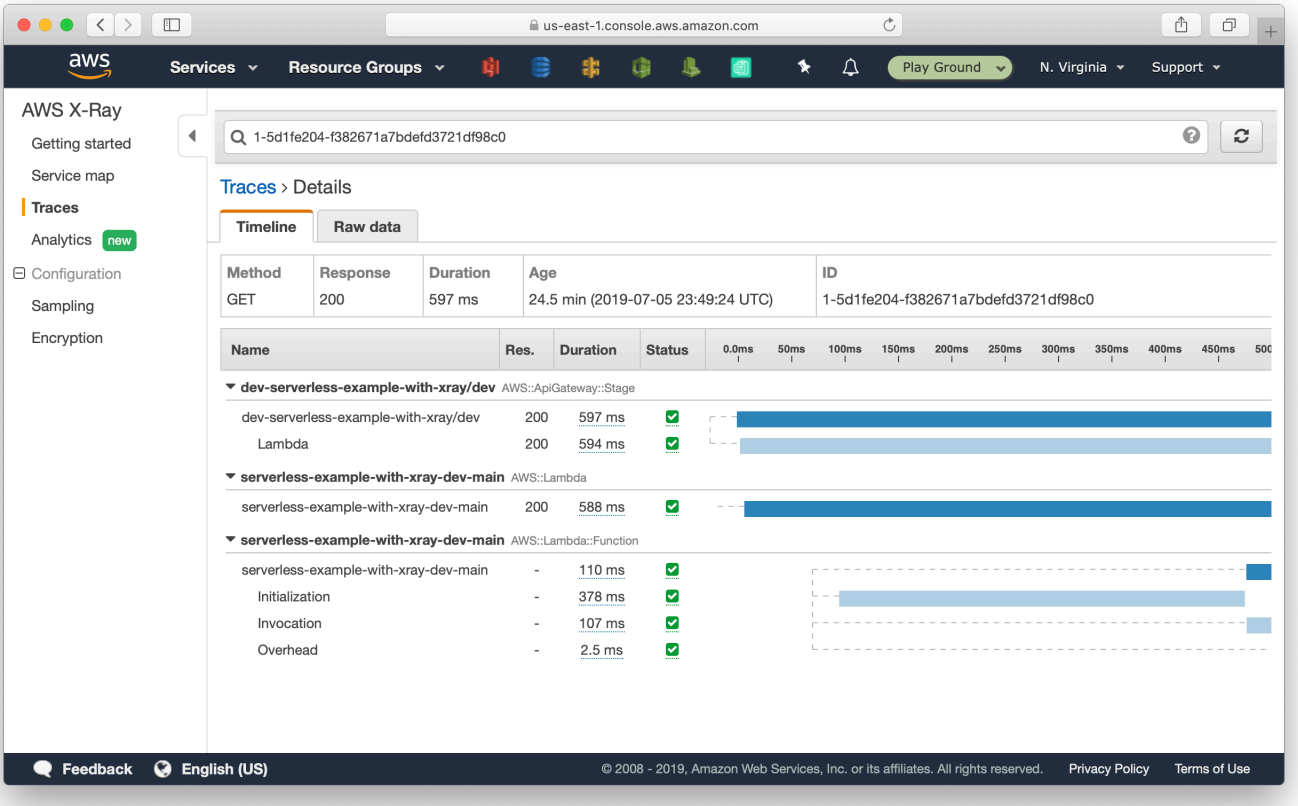
```
$ curl https://xxxxxxxxxx.execute-api.us-east-
1.amazonaws.com/xxx
```

Head over to your AWS X-Ray console, and select **Traces** from the left menu.



The **Trace overview** section at the top shows all the URLs that initiated the trace. And the **Trace list** section at the bottom shows each individual trace. By default, it shows all the traces within the last 5 minutes. However, you can pick a different time range.

Click on a trace in the **Trace list**. Note: it might take up to 30 seconds before a trace shows up after a request has been made.



Here are a couple of things you can see:

- The API request was a GET request and succeeded with a HTTP 200 status.
- The entire request took API Gateway 597ms to process.
- Out of 597ms, 594ms was spent by Lambda function. Meaning API Gateway added an overhead of 3ms to this request.
- And out of 594ms, it took 387ms for Lambda to initialize. That's the **Cold Start** time.
- The actual function took 107ms to run.

However I'm still left wondering about:

- How long did the DynamoDB query and the SNS call each take?
- If an API request fails, how do I know if it failed at the DynamoDB step or the SNS step?

To do this we need to enable X-Ray tracing for the services that were

invoked by Lambda.

Enable X-Ray tracing for other AWS services invoked by AWS Lambda

Install the AWS X-Ray SDK. In your project directory, run:

```
$ npm install -s aws-xray-sdk
```

Update your Lambda code and wrap AWS SDK with the X-Ray SDK. Change:

```
const AWS = require('aws-sdk');
```

To:

```
const AWSXRay = require('aws-xray-sdk-core');
const AWS = AWSXRay.captureAWS(require('aws-sdk'));
```

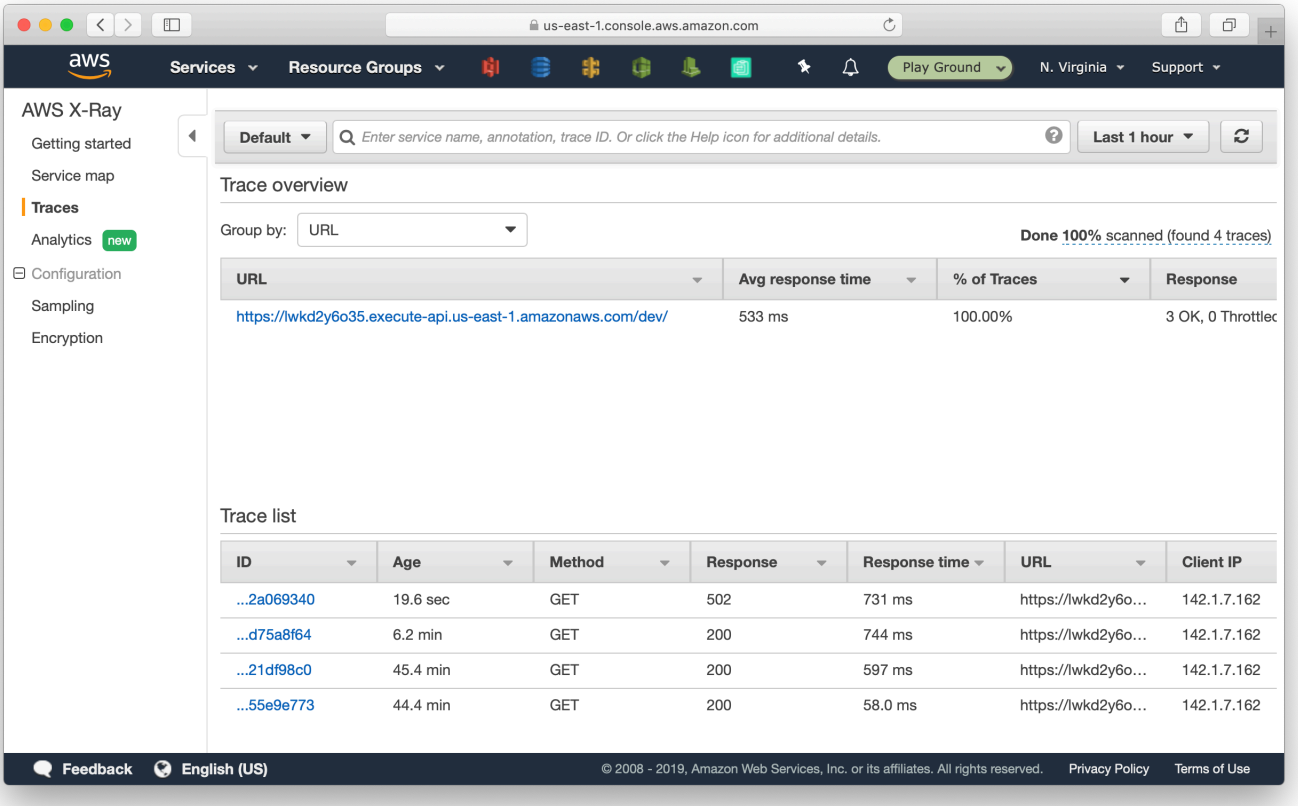
That's it!

Now run `serverless deploy` again to deploy the change. This time you can deploy a single function using `serverless deploy -f FUNCTION_NAME`, since we only changed the function code, not our `serverless.yml`.

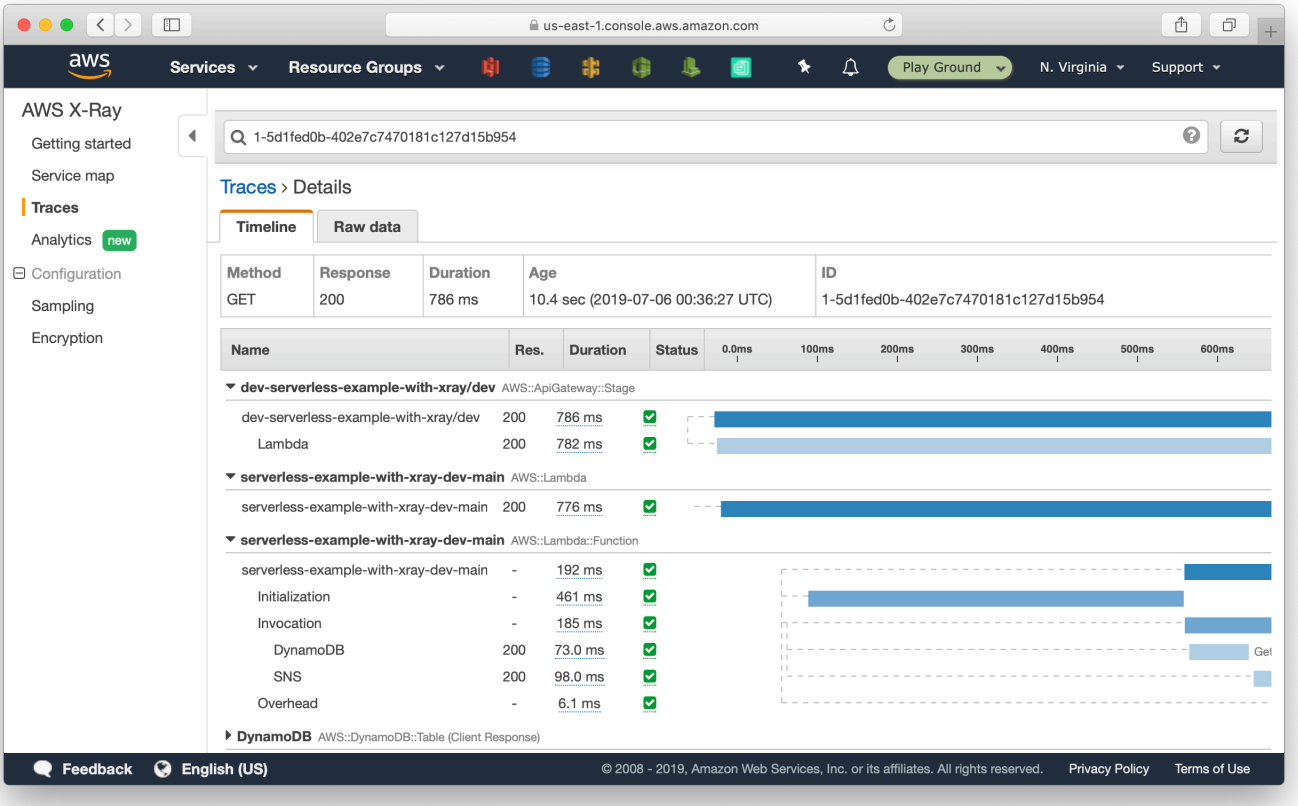
After you deploy, invoke your API Gateway endpoint again:

```
$ curl https://xxxxxxxxxx.execute-api.us-east-
1.amazonaws.com/xxx
```

Go back to your AWS X-Ray console, wait for the new trace to show up. It might take up to 30 seconds to do so. You can tell if a trace is recent by looking at its **Age**:



Select the new trace.



This time:

- The Lambda cold start took 461ms, and 185ms to process the request.
- Out of the 185ms, the DynamoDB query took 73ms and the SNS publish call took 98ms.

We can also see clearly the various steps that took place as a part of our Lambda function invocation. Our sample repo (https://github.com/AnomalyInnovations/serverless-stack-demo-ext-api) has AWS X-Ray enabled by default so you can play around with the concepts we talked about in this chapter.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/)

Wrapping up the Best Practices

Congratulations on completing this best practices section of the guide!

You should now have a working real-world Serverless app with:

1. Resources exported and shared between services
2. A single API endpoint shared between services
3. With multiple environments configured across AWS accounts
4. Resource names parameterized with the environment names
5. Deployed in phases to the environments
6. Secrets stored in SSM
7. Tracing enabled in AWS X-Ray

And you should've stepped through the development workflow that we recommended!

The above setup and workflow is exactly what we, and a number of other companies are using in production. We hope that this gives you a good starting point for your projects!

We'd love to hear from you about your experience following this guide. Please send us any comments or feedback you might have, via email (mailto:contact@anoma.ly). Also, if there are any other topics you'd like us to cover, please leave a comment in the discussion thread below!

Thank you and we hope you found this guide helpful!

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/wrapping-up-the-best-practices/1340)

API Gateway and Lambda Logs

Logging is an essential part of building backends and it is no different for a serverless API. It gives us visibility into how we are processing and responding to incoming requests.

Types of Logs

There are 2 types of logs we usually take for granted in a monolithic environment.

- Server logs

  Web server logs maintain a history of requests, in the order they took place. Each log entry contains the information about the request, including client IP address, request date/time, request path, HTTP code, bytes served, user agent, etc.

- Application logs

  Application logs are a file of events that are logged by the web application. It usually contains errors, warnings, and informational events. It could contain everything from unexpected function failures, to key events for understanding how users behave.

In the serverless environment, we have lesser control over the underlying

infrastructure, logging is the only way to acquire knowledge on how the application is performing. Amazon CloudWatch (https://aws.amazon.com/cloudwatch/) is a monitoring service to help you collect and track metrics for your resources. Using the analogy of server logs and application logs, you can roughly think of the API Gateway logs as your server logs and Lambda logs as your application logs.

In the chapter we are going to look to do the following:

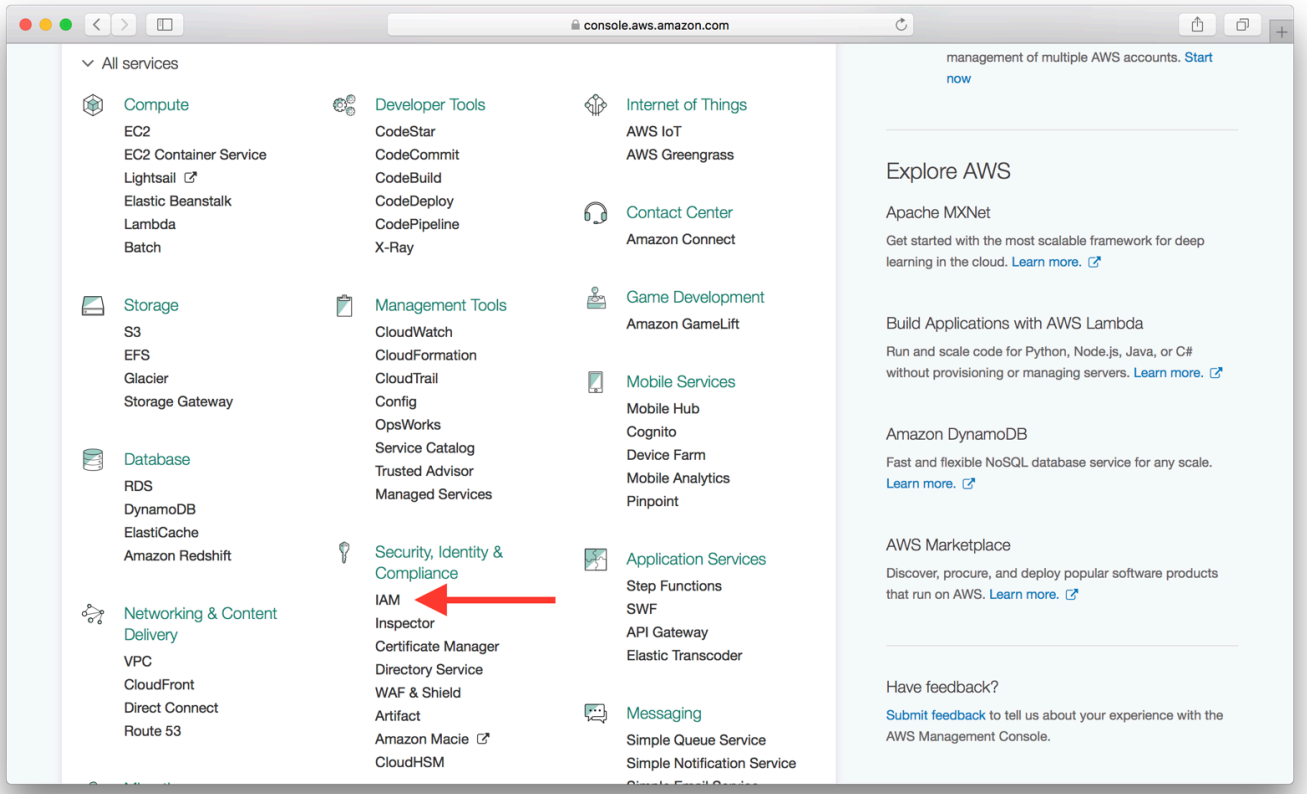
- Enable API Gateway CloudWatch Logs (#enable-api-gateway-cloudwatch-logs)
- Enable Lambda CloudWatch Logs (#enable-lambda-cloudwatch-logs)
- Viewing API Gateway CloudWatch Logs (#viewing-api-gateway-cloudwatch-logs)
- Viewing Lambda CloudWatch Logs (#viewing-lambda-cloudwatch-logs)

Let's get started.

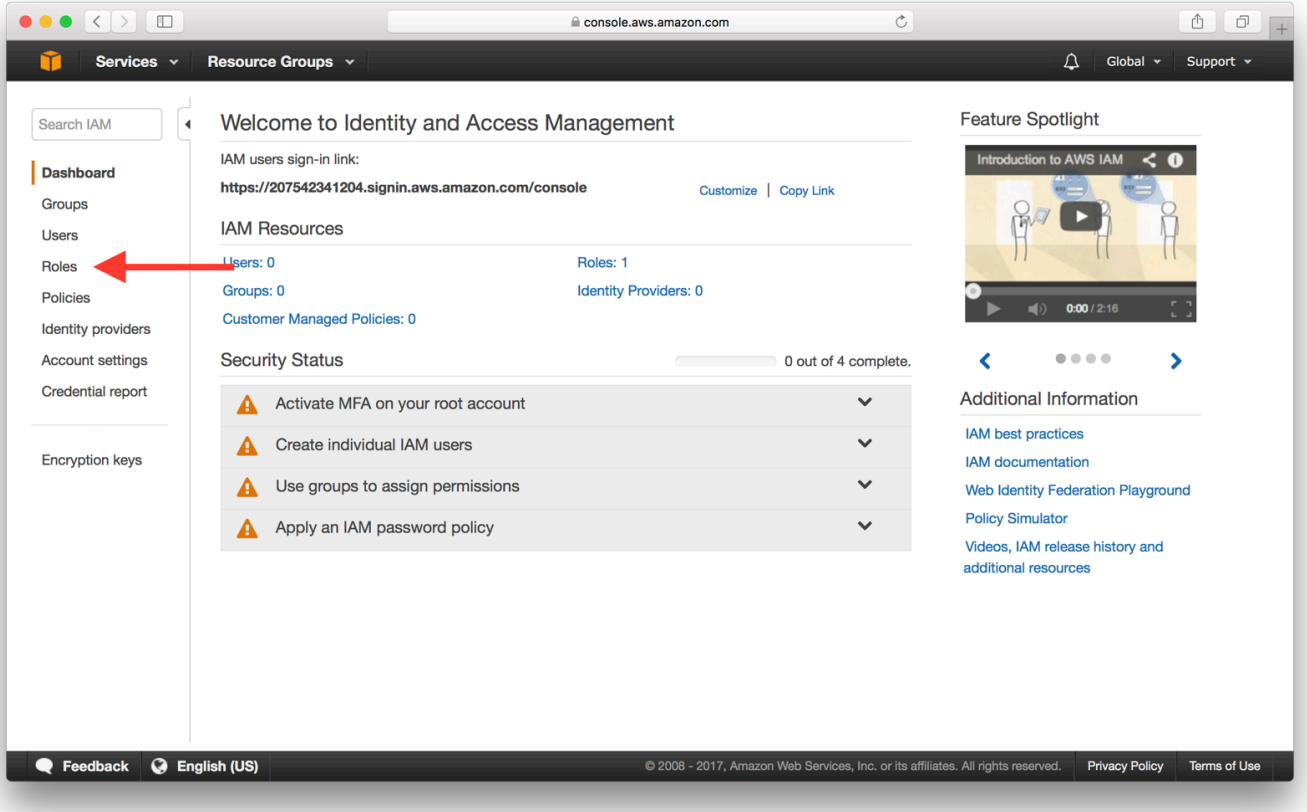
Enable API Gateway CloudWatch Logs

This is a two step process. First, we need to create an IAM role that allows API Gateway to write logs in CloudWatch. Then we need to turn on logging for our API project.

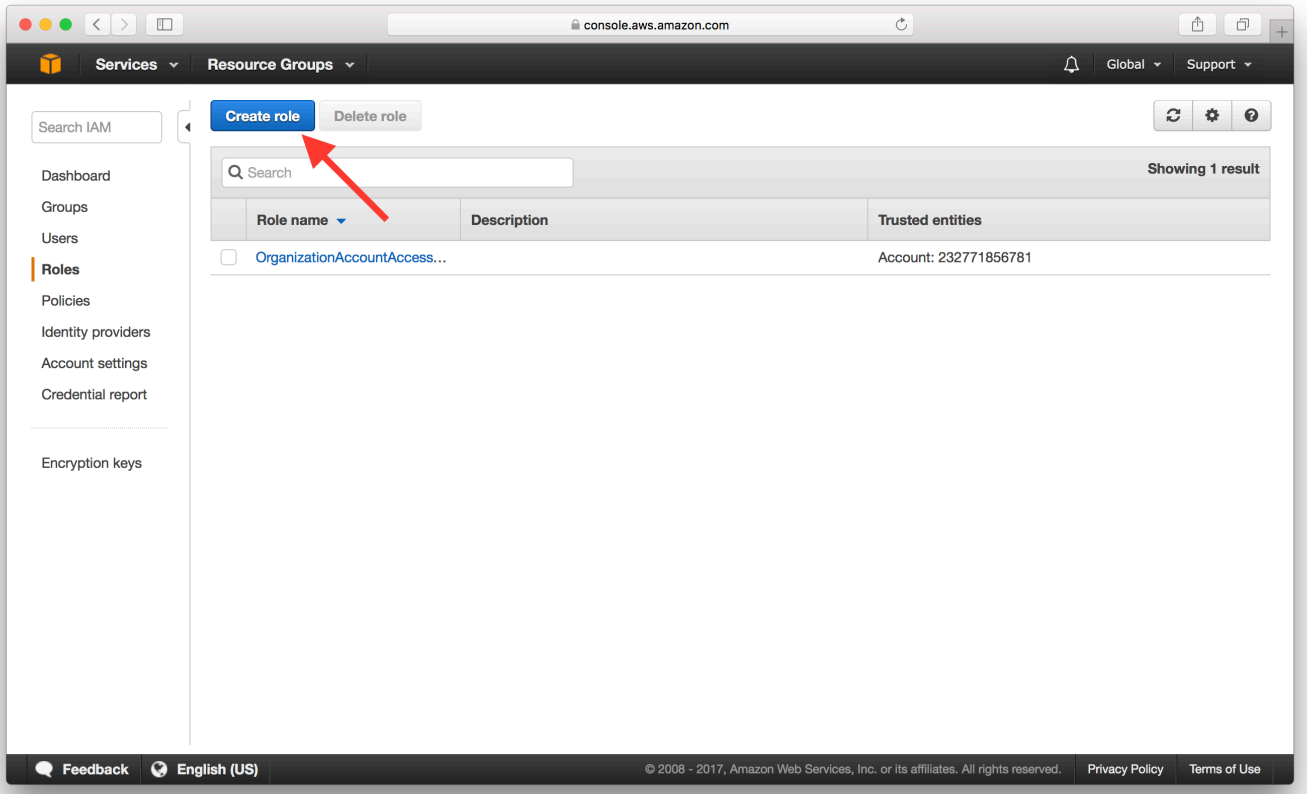
First, log in to your AWS Console (https://console.aws.amazon.com) and select IAM from the list of services.



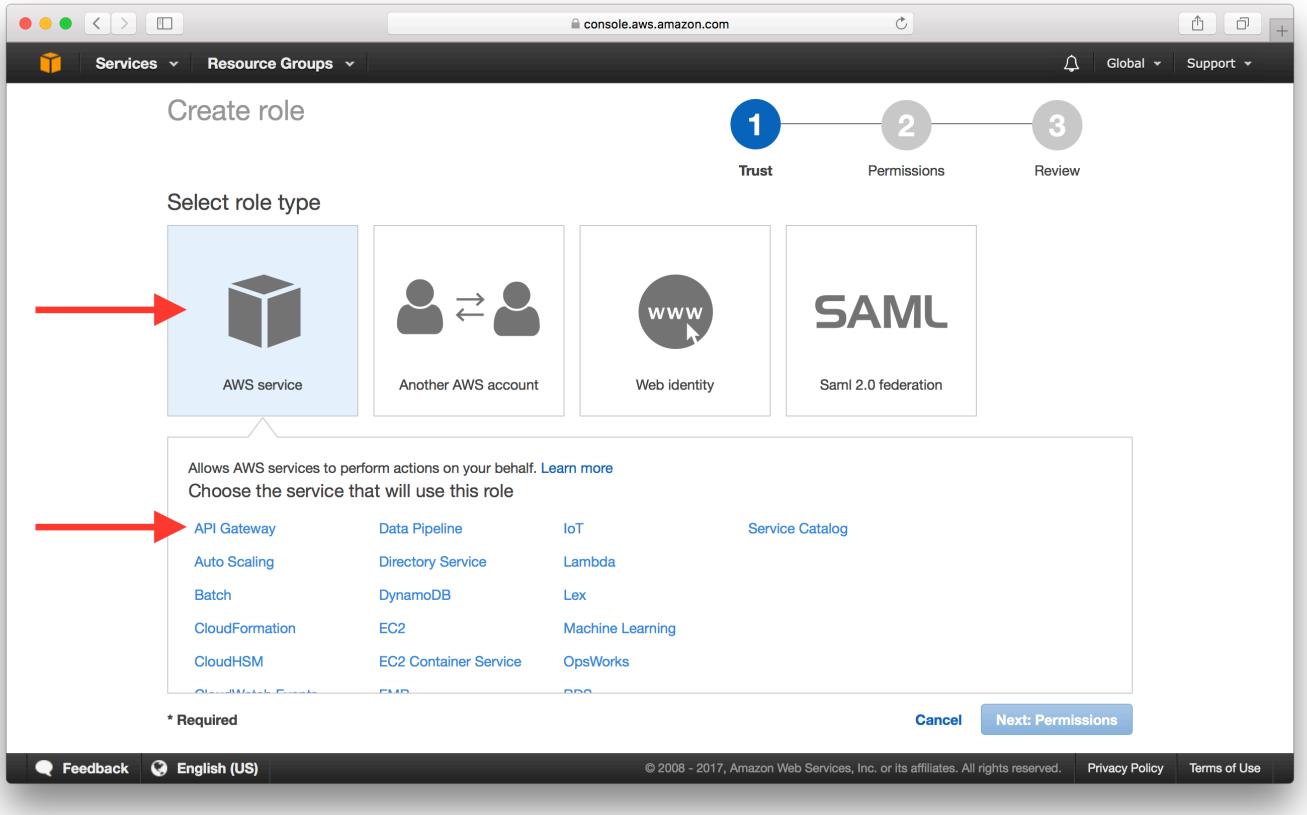
Select **Roles** on the left menu.



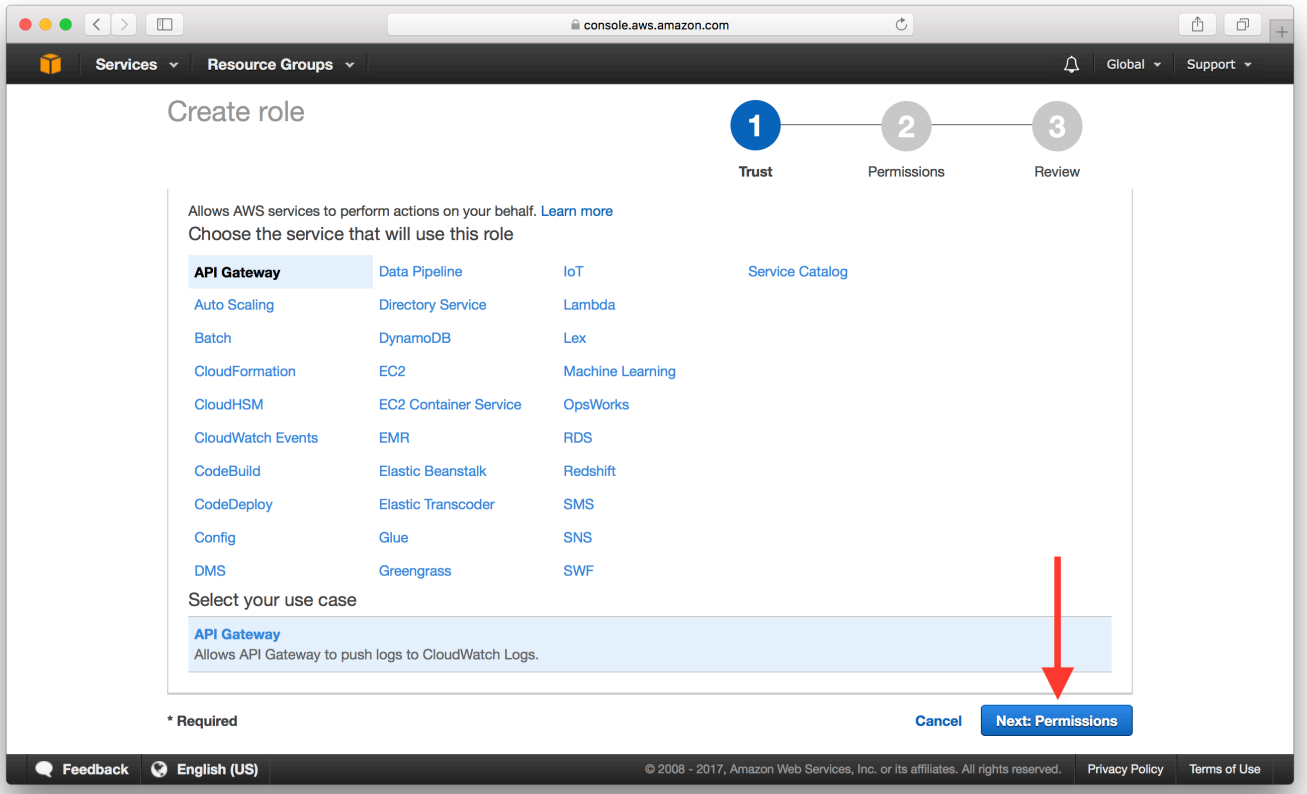
Select **Create Role**.



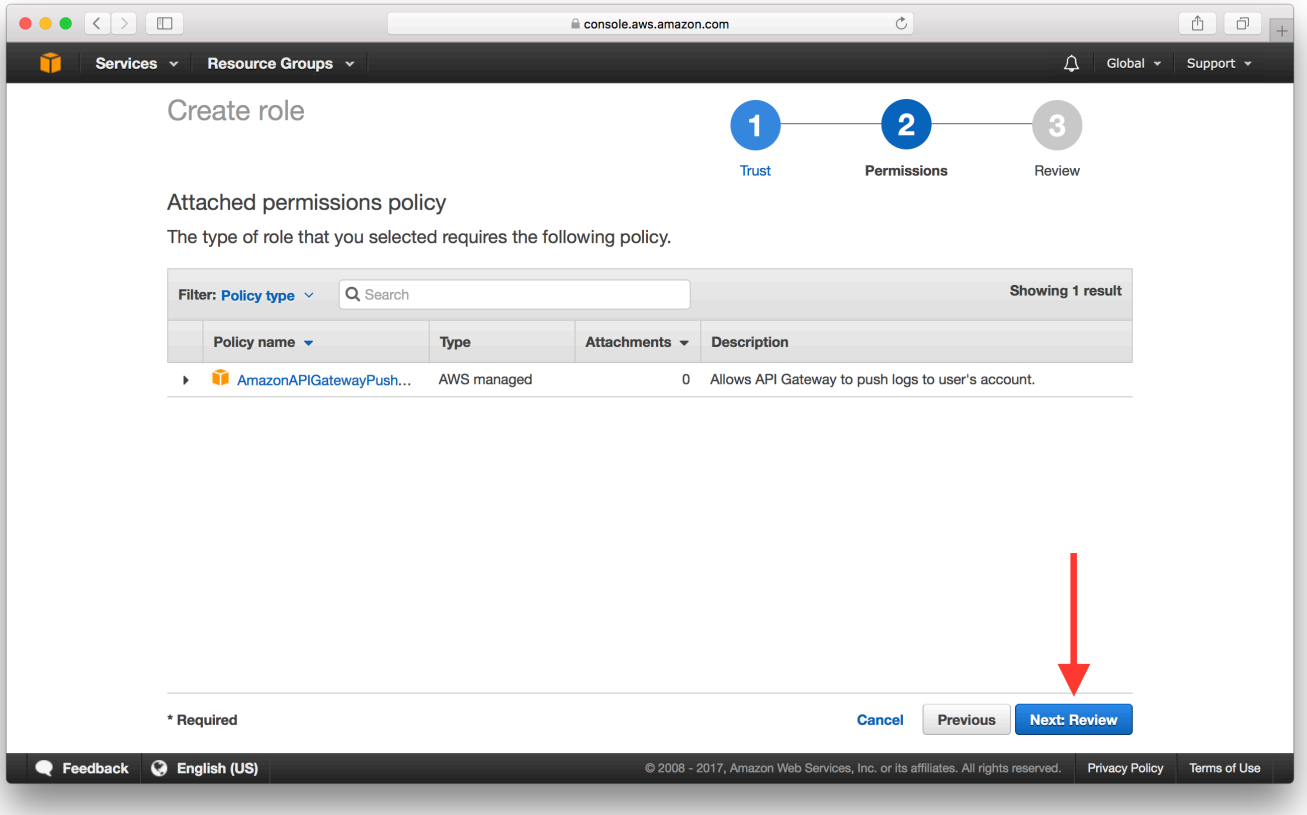
Under **AWS service**, select **API Gateway**.



Click **Next: Permissions**.

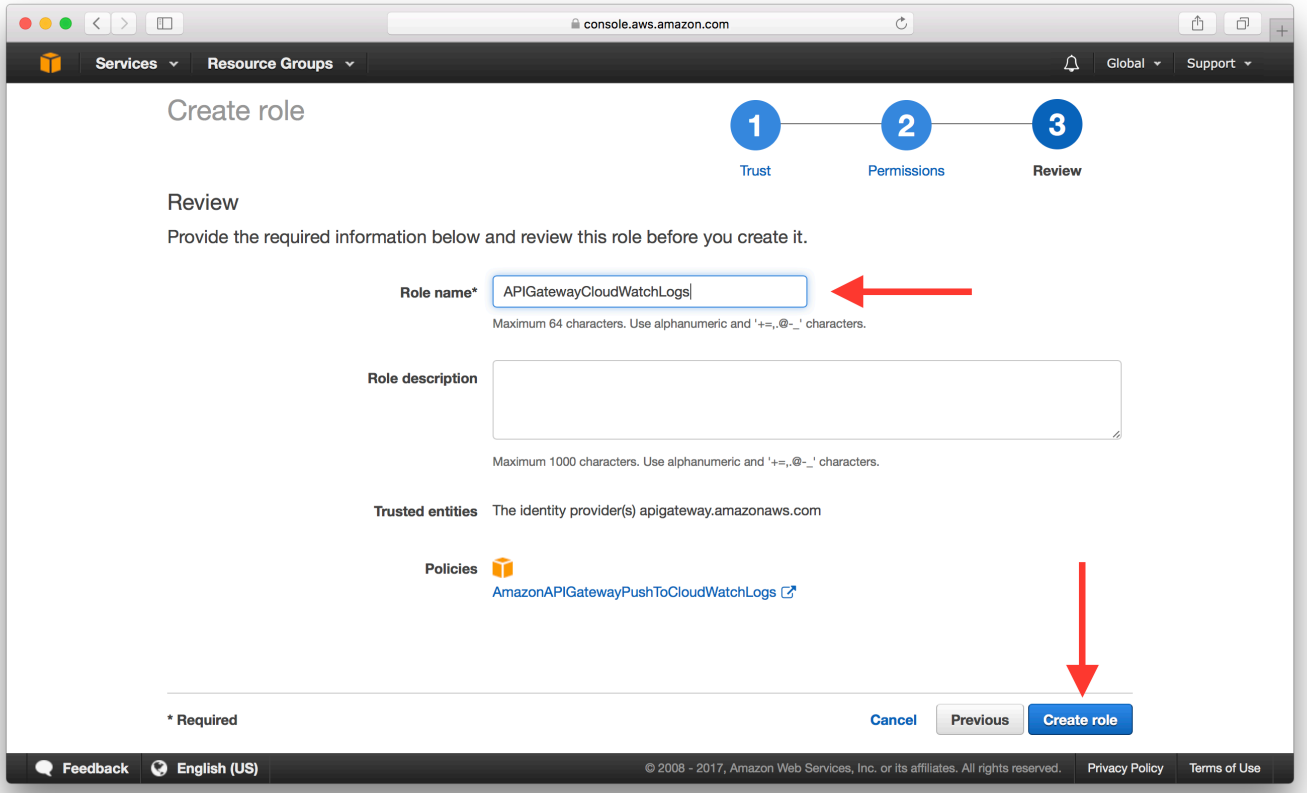


Click **Next: Review.**

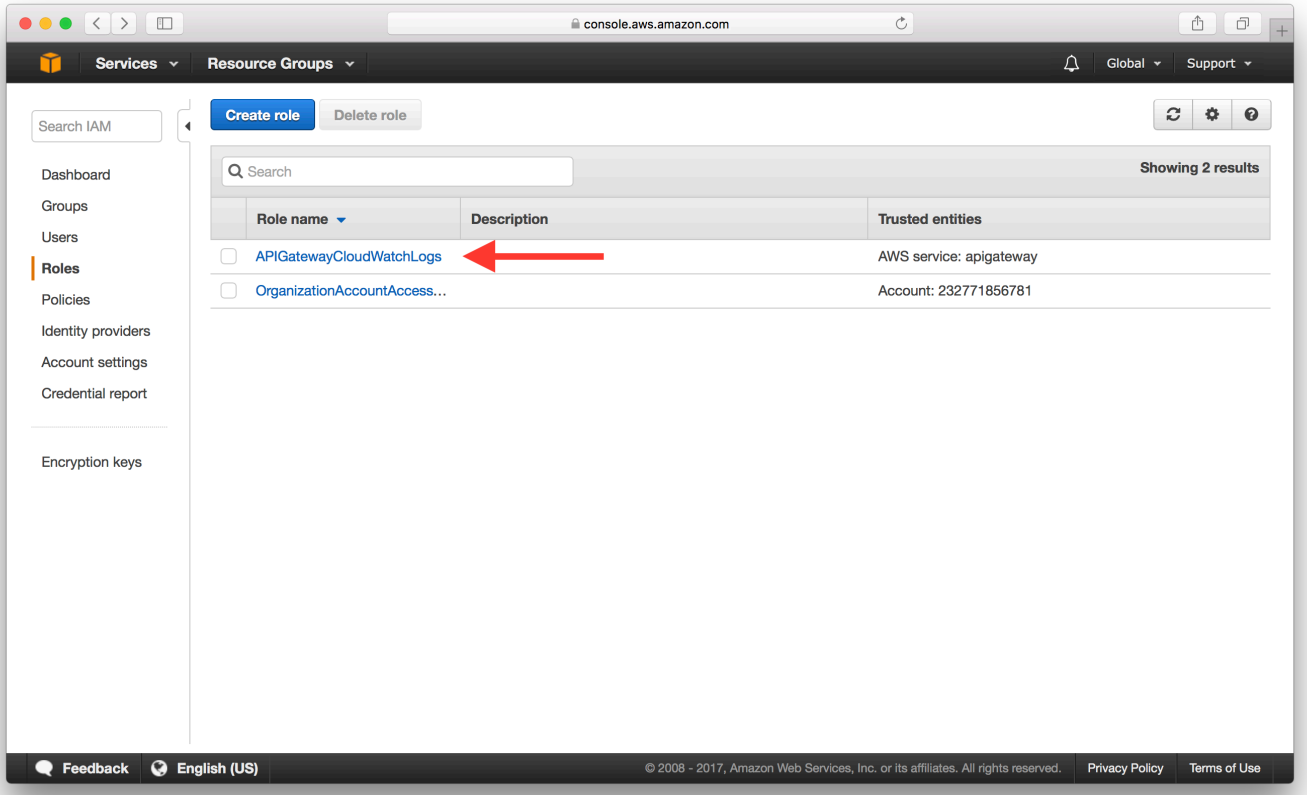


Enter a **Role name** and select **Create role**. In our case, we called our role

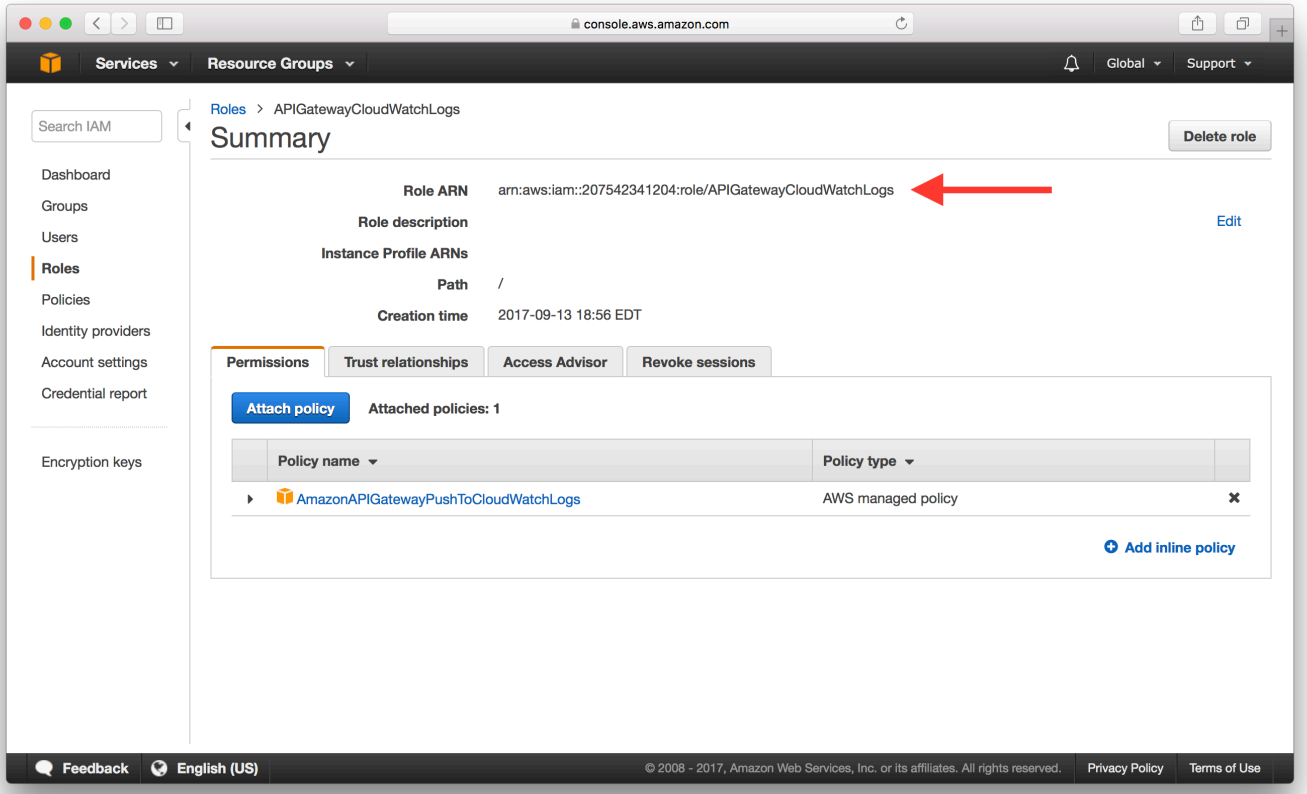
`APIGatewayCloudWatchLogs`.



Click on the role we just created.

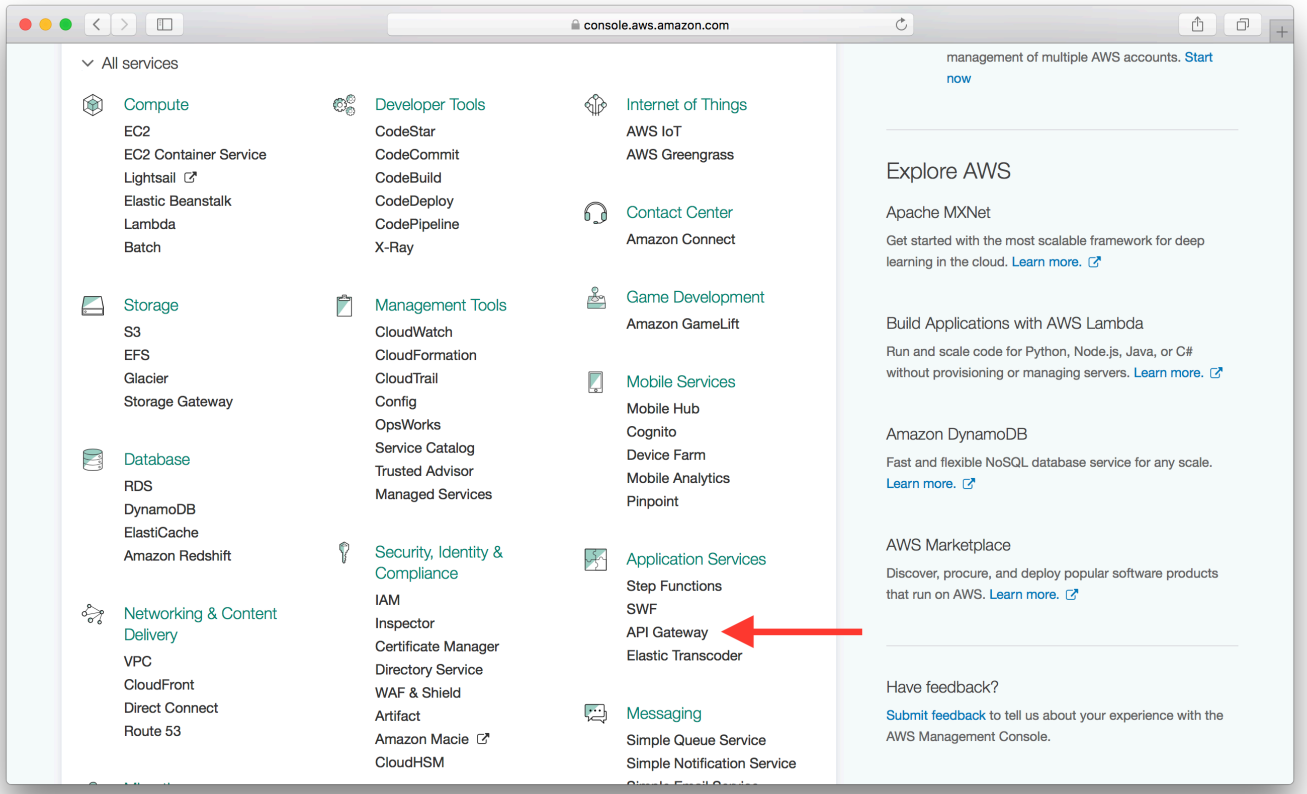


Take a note of the **Role ARN**. We will be needing this soon.

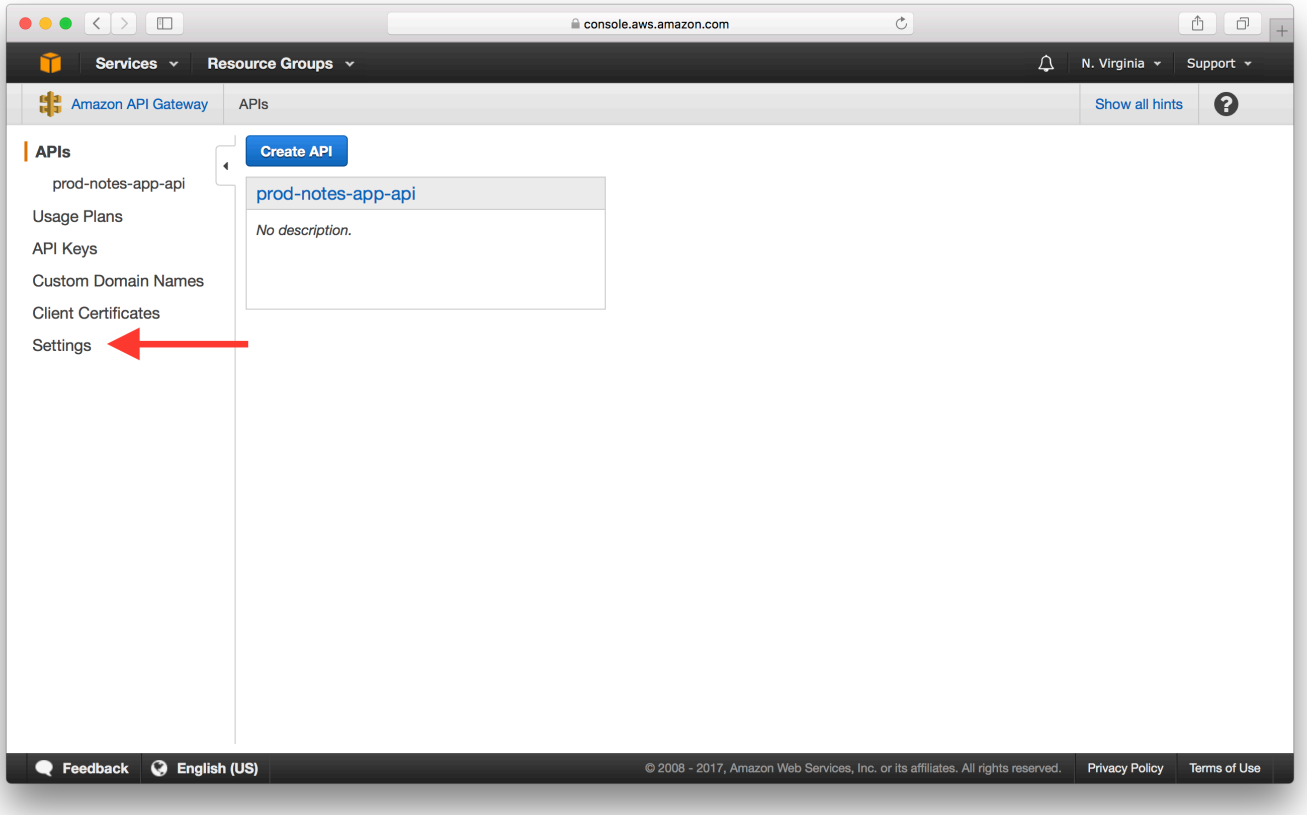


Now that we have created our IAM role, let's turn on logging for our API Gateway project.

Go back to your AWS Console (https://console.aws.amazon.com) and select API Gateway from the list of services.

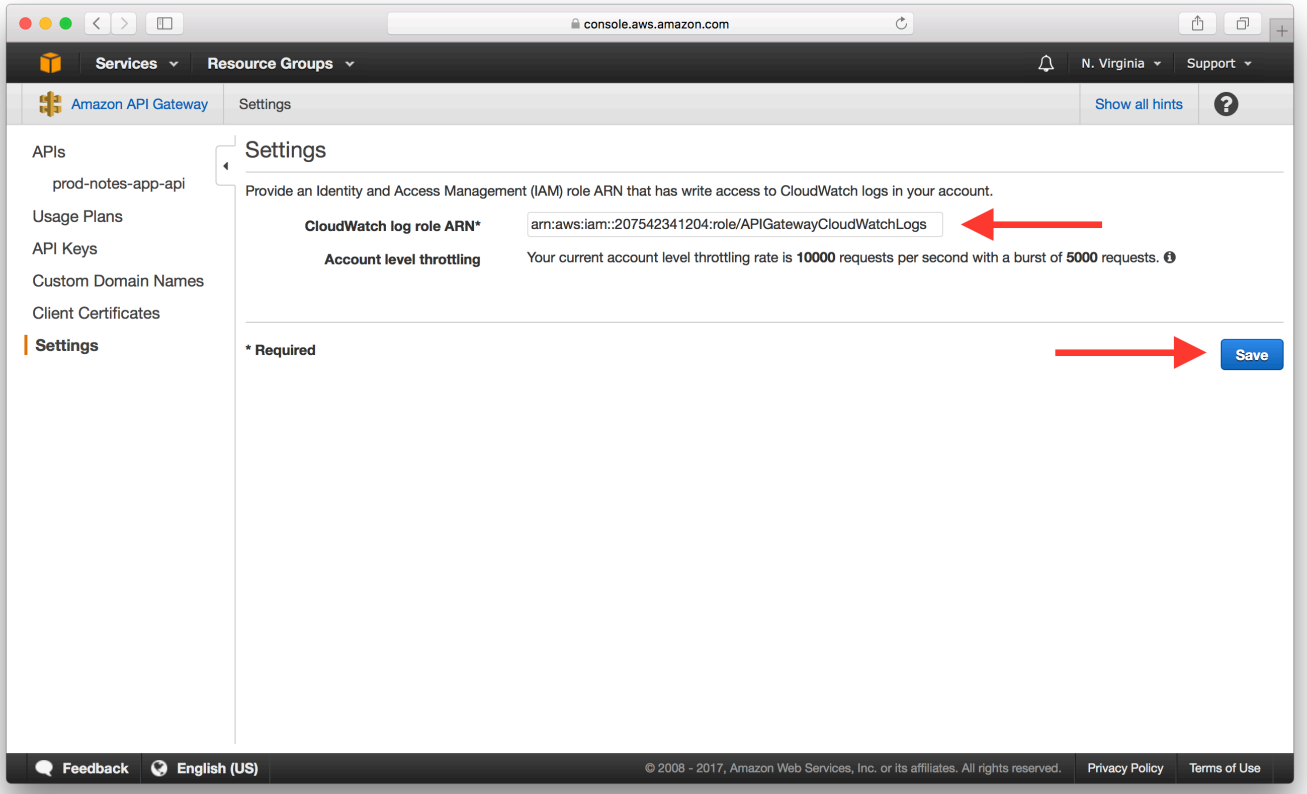


Select **Settings** from the left panel.

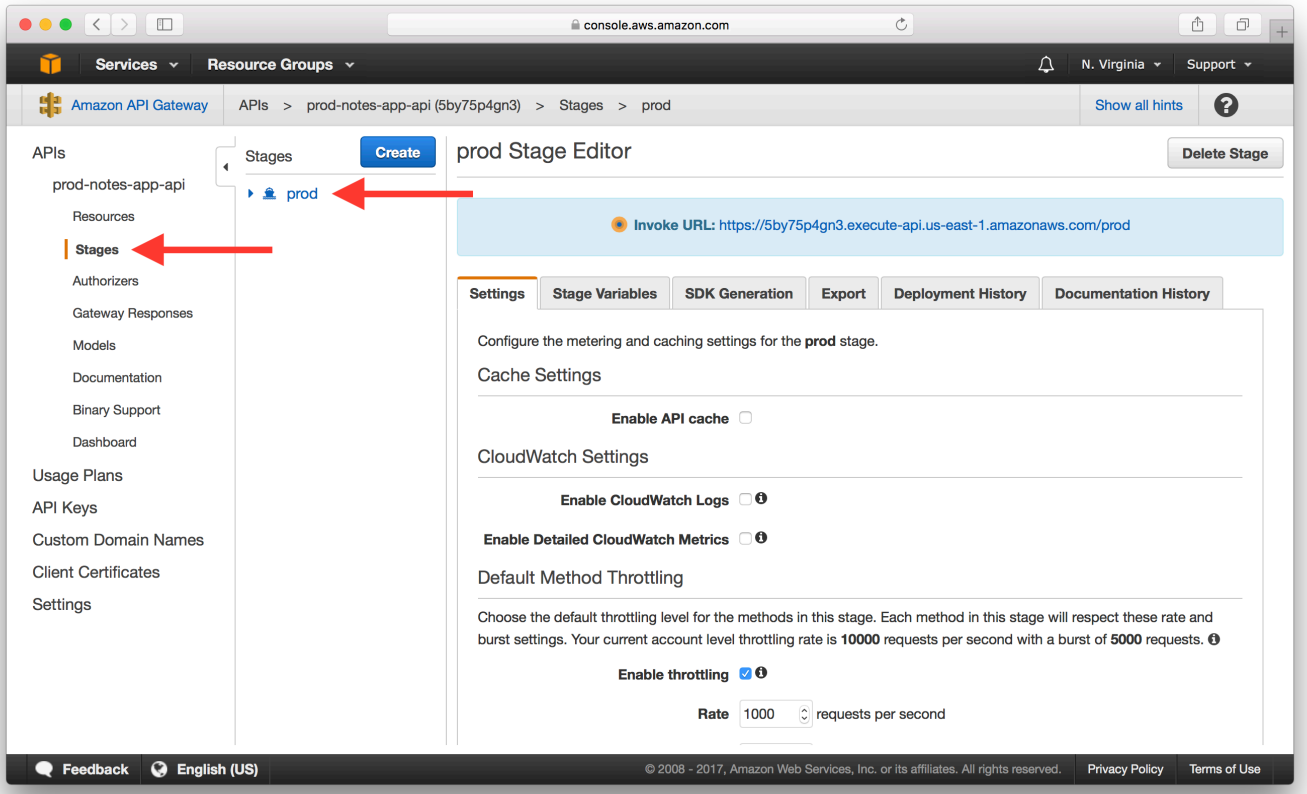


Enter the ARN of the IAM role we just created in the **CloudWatch log role**

ARN field and hit **Save**.

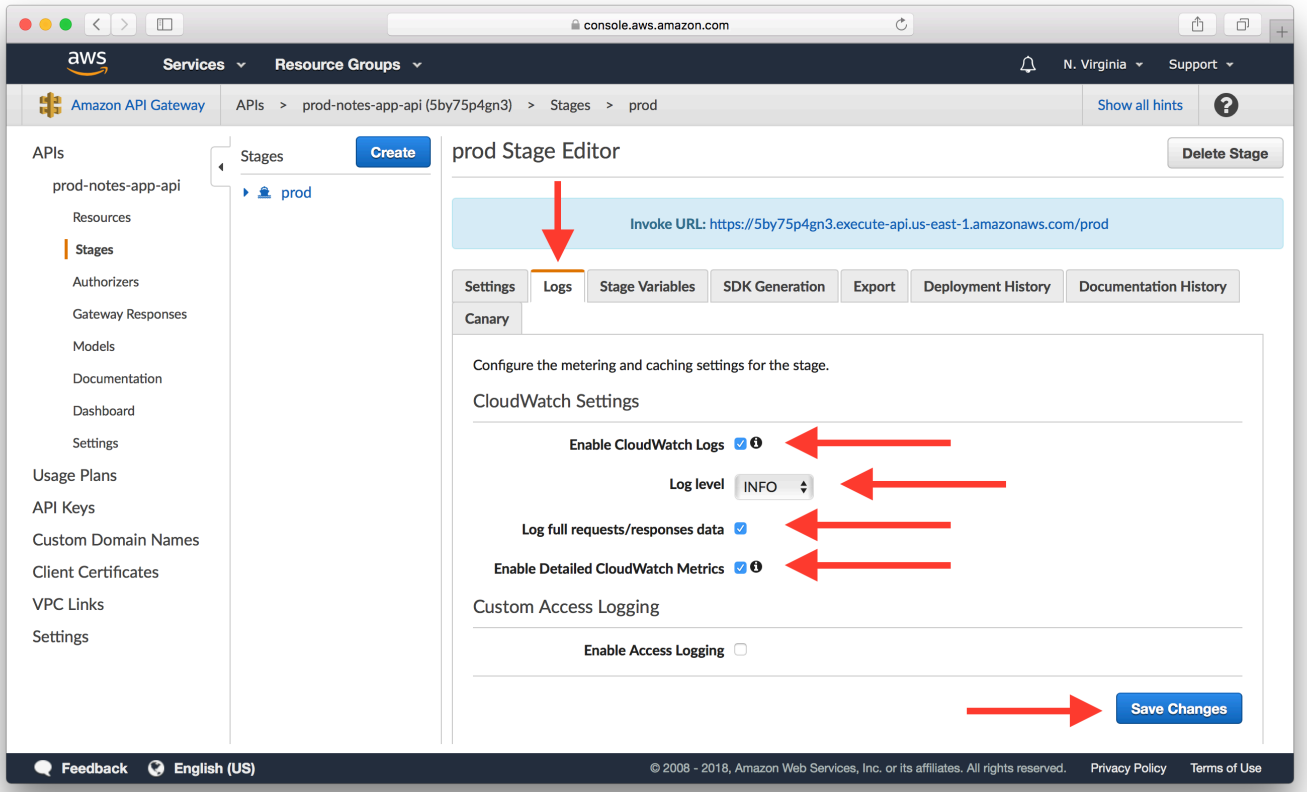


Select your API project from the left panel, select **Stages**, then pick the stage you want to enable logging for. For the case of our Notes App API (https://github.com/AnomalyInnovations/serverless-stack-demo-api), we deployed to the `prod` stage.



In the **Logs** tab:

- Check **Enable CloudWatch Logs**.
- Select **INFO** for **Log level** to log every request.
- Check **Log full requests/responses data** to include entire request and response body in the log.
- Check **Enable Detailed CloudWatch Metrics** to track latencies and errors in CloudWatch metrics.



Scroll to the bottom of the page and click **Save Changes**. Now our API Gateway requests should be logged via CloudWatch.

Note that, the execution logs can generate a ton of log data and it's not recommended to leave them on. They are much better for debugging. API Gateway does have support for access logs, which we recomend leaving on. Here is how to enable access logs for your API Gateway project (https://seed.run/blog/how-to-enable-access-logs-for-api-gateway).

Enable Lambda CloudWatch Logs

Lambda CloudWatch logs are enabled by default. It tracks the duration and max memory usage for each execution. You can write additional information to CloudWatch via `console.log`. For example:

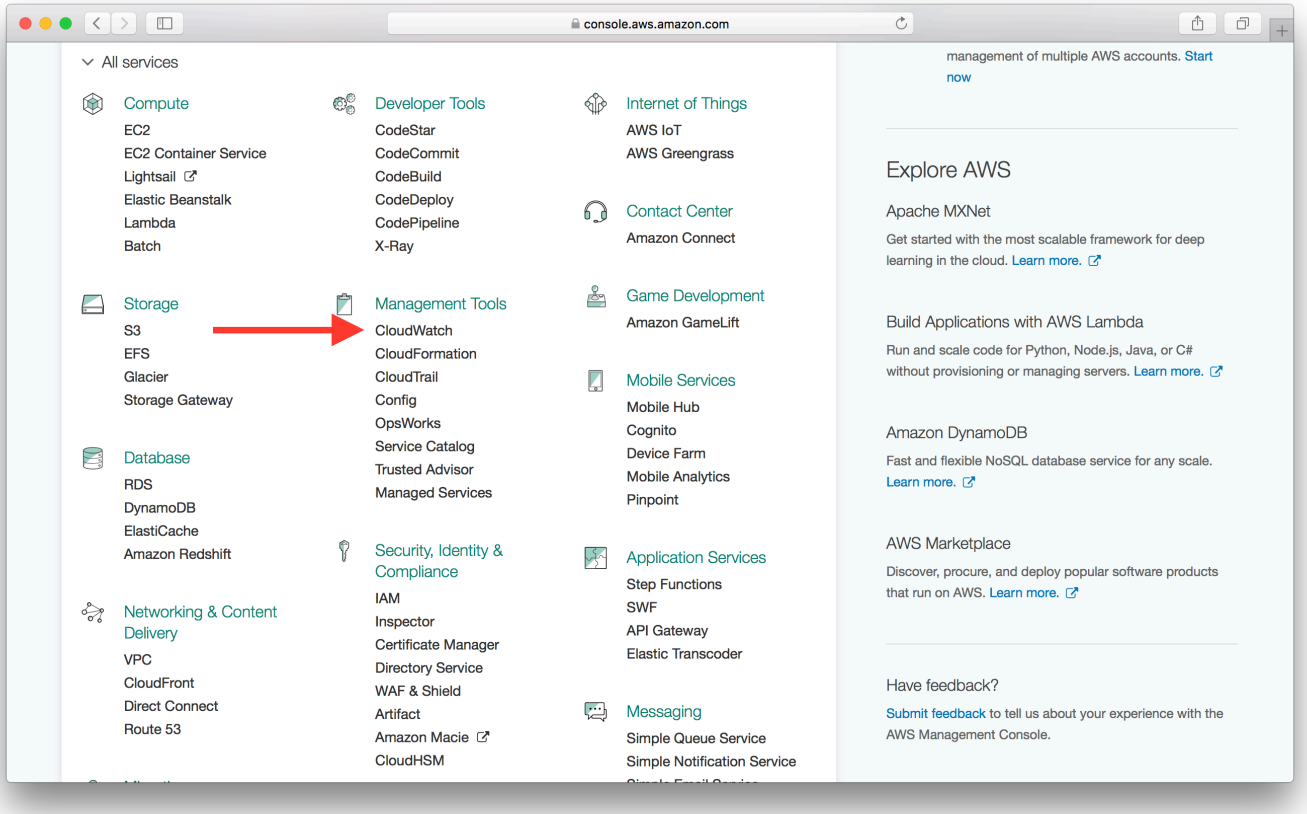
```
export function main(event, context, callback) {
  console.log('Hello world');
```

```
  callback(null, { body: '' });
}
```

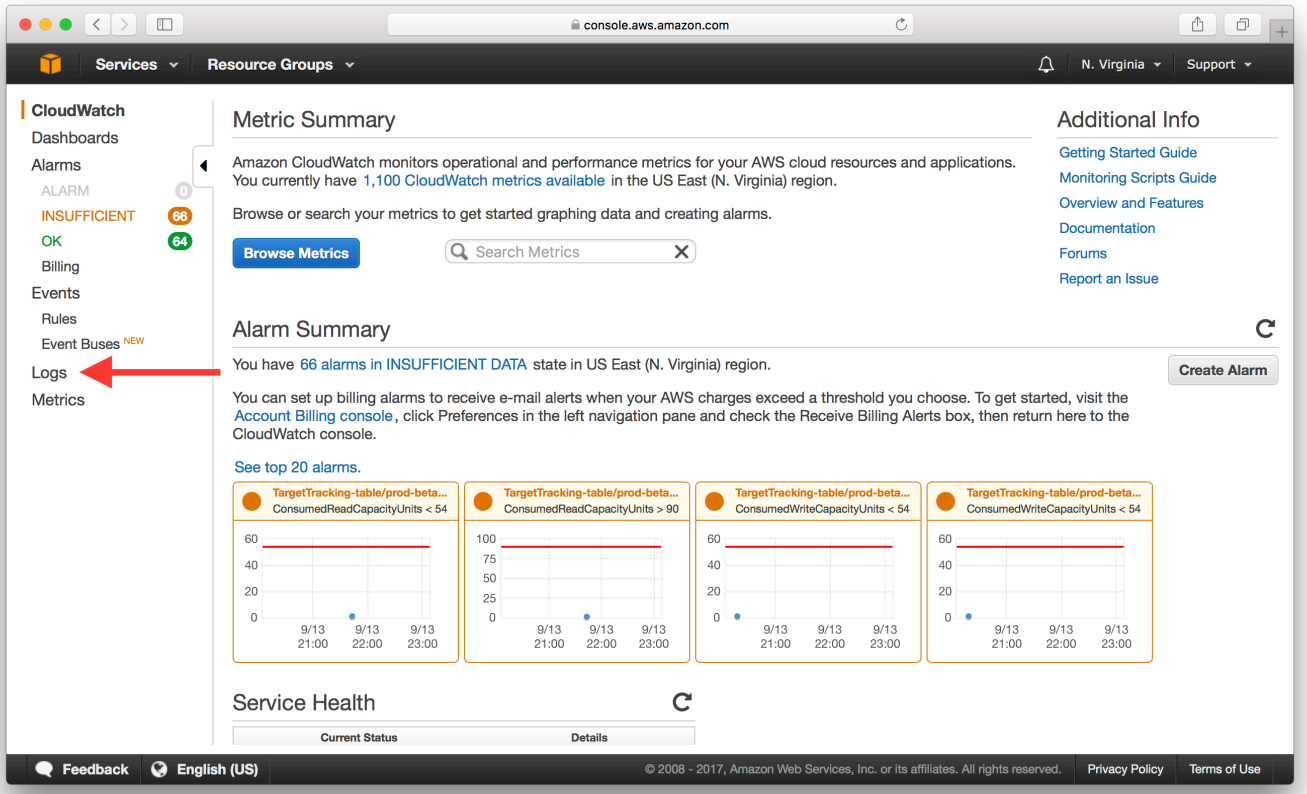
Viewing API Gateway CloudWatch Logs

CloudWatch groups log entries into **Log Groups** and then further into **Log Streams**. Log Groups and Log Streams can mean different things for different AWS services. For API Gateway, when logging is first enabled in an API project's stage, API Gateway creates 1 log group for the stage, and 300 log streams in the group ready to store log entries. API Gateway picks one of these streams when there is an incoming request.

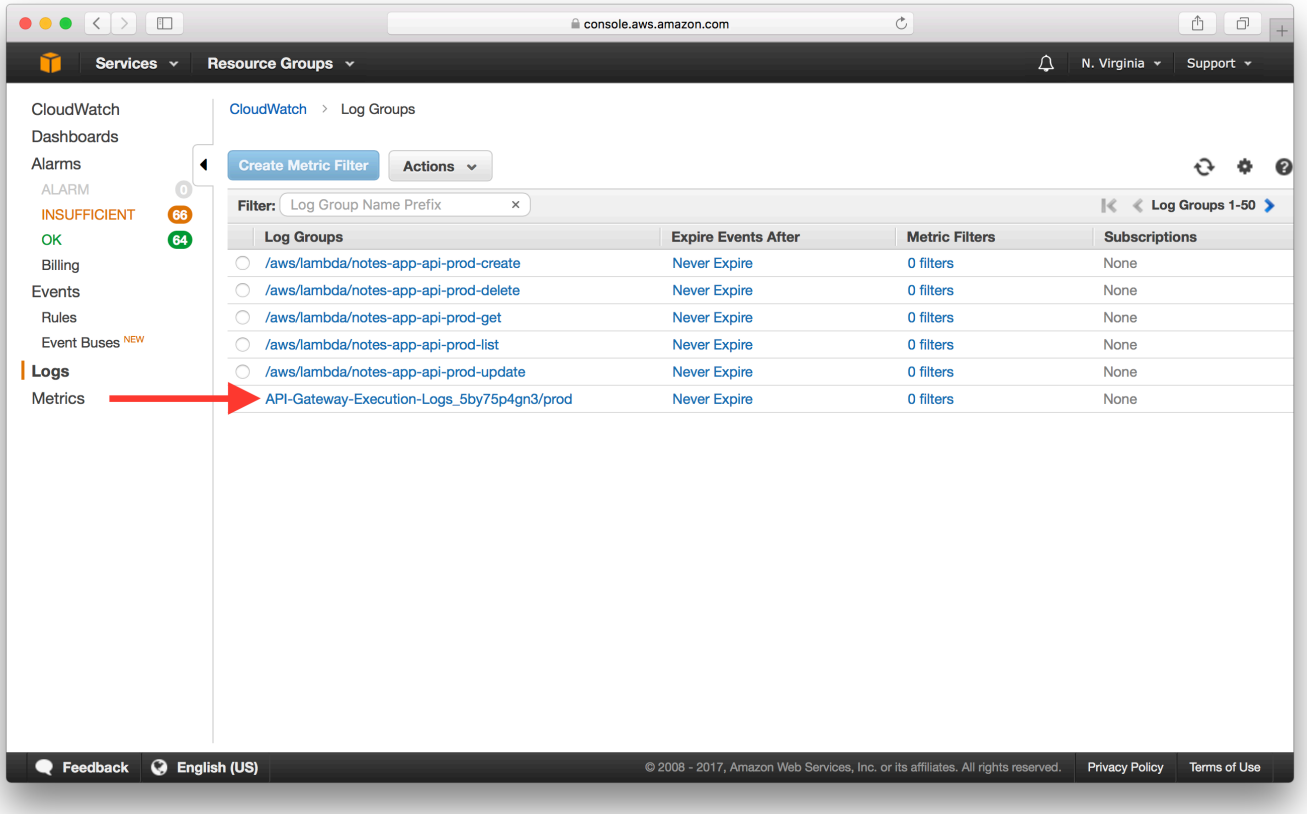
To view API Gateway logs, log in to your AWS Console (https://console.aws.amazon.com) and select CloudWatch from the list of services.



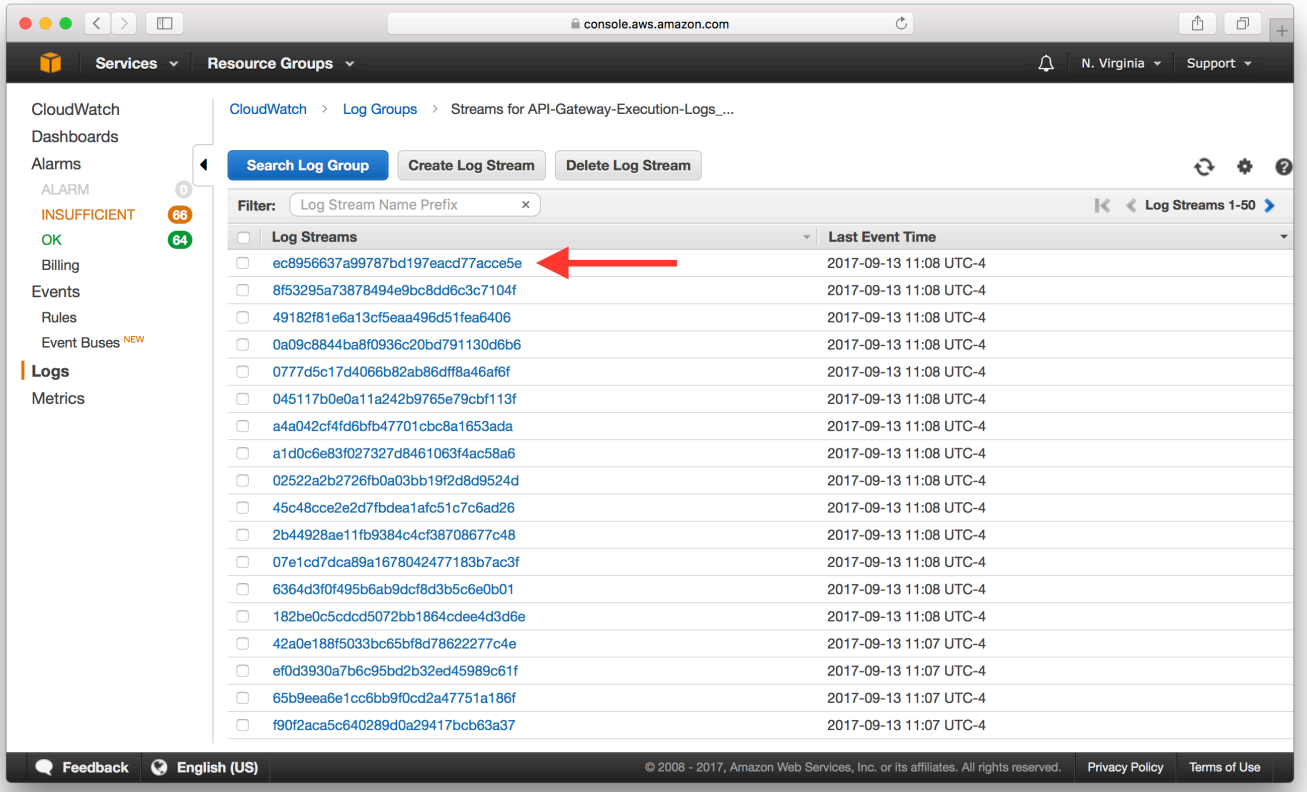
Select **Logs** from the left panel.



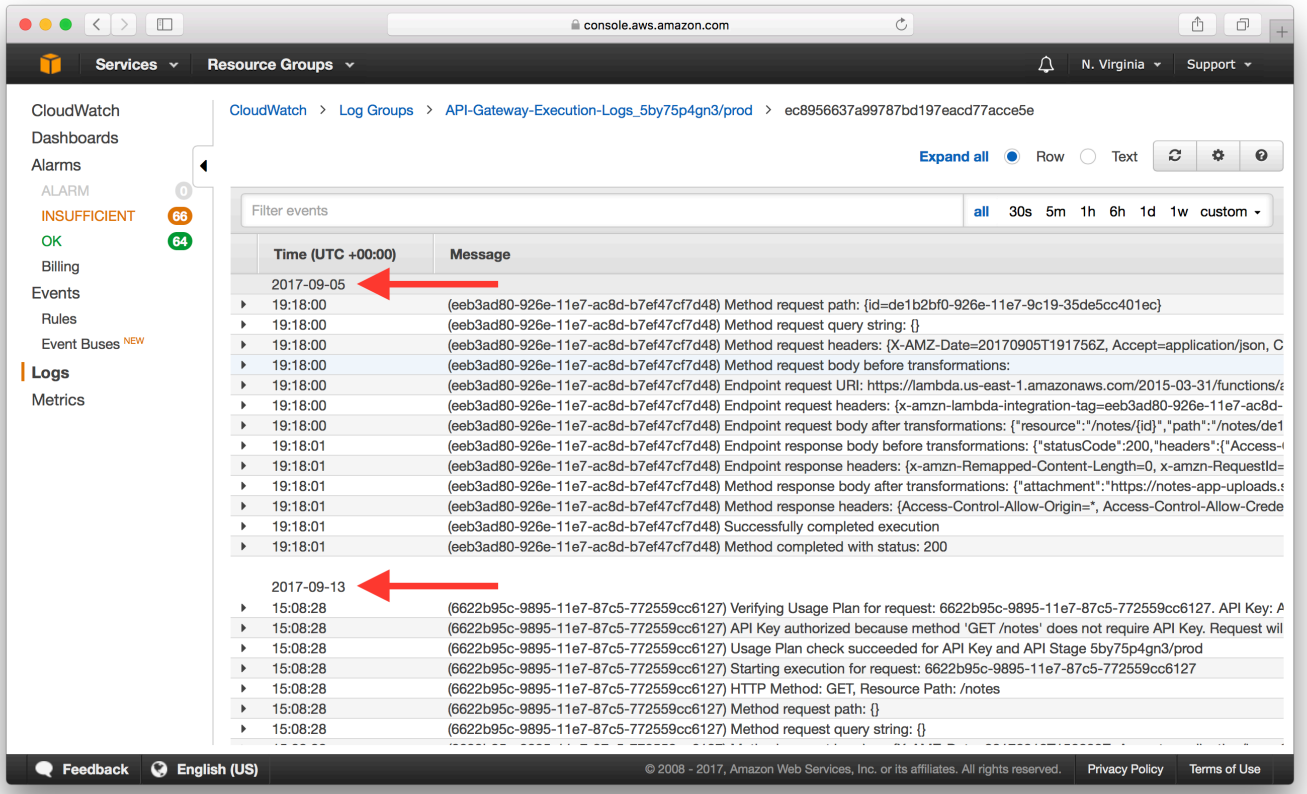
Select the log group prefixed with **API-Gateway-Execution-Logs_** followed by the API Gateway id.



You should see 300 log streams ordered by the last event time. This is the last time a request was recorded. Select the first stream.



This shows you the log entries grouped by request.

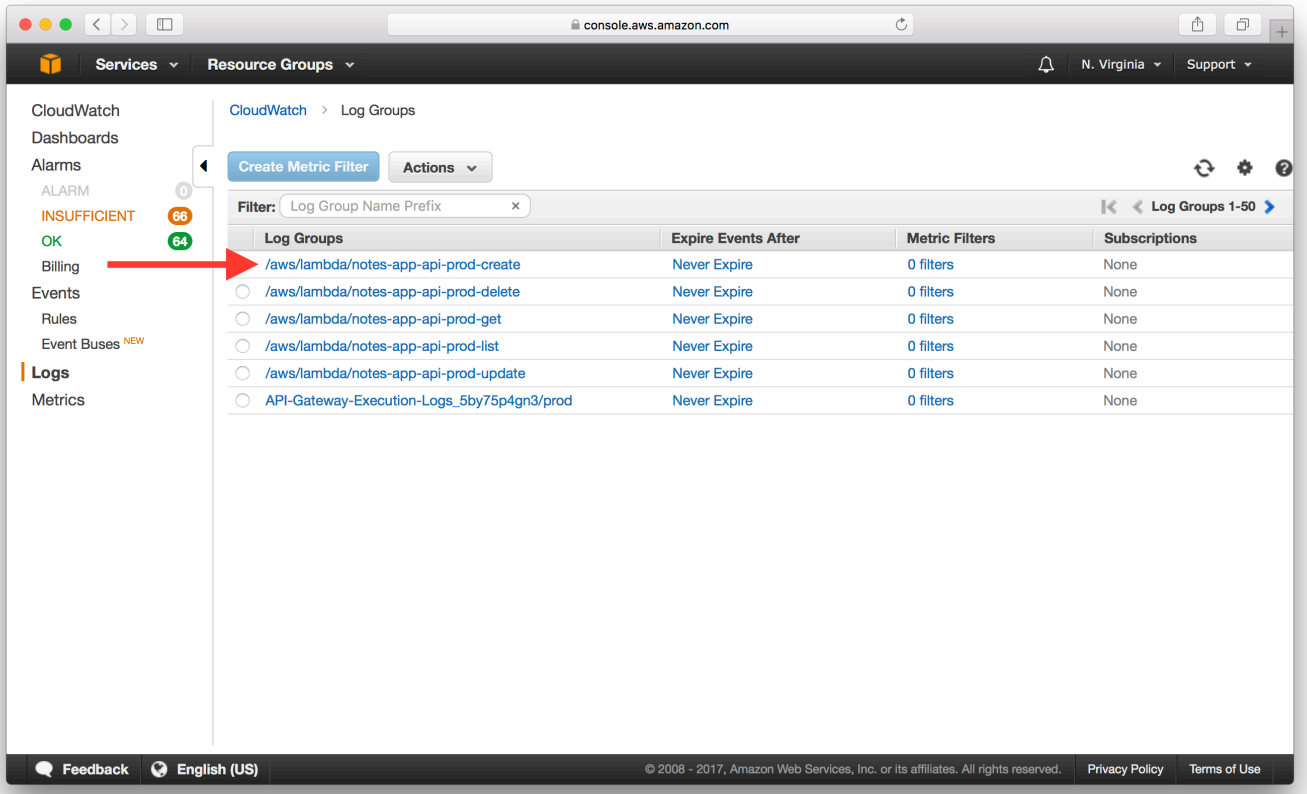


Note that two consecutive groups of logs are not necessarily two consecutive requests in real time. This is because there might be other requests that are processed in between these two that were picked up by one of the other log streams.

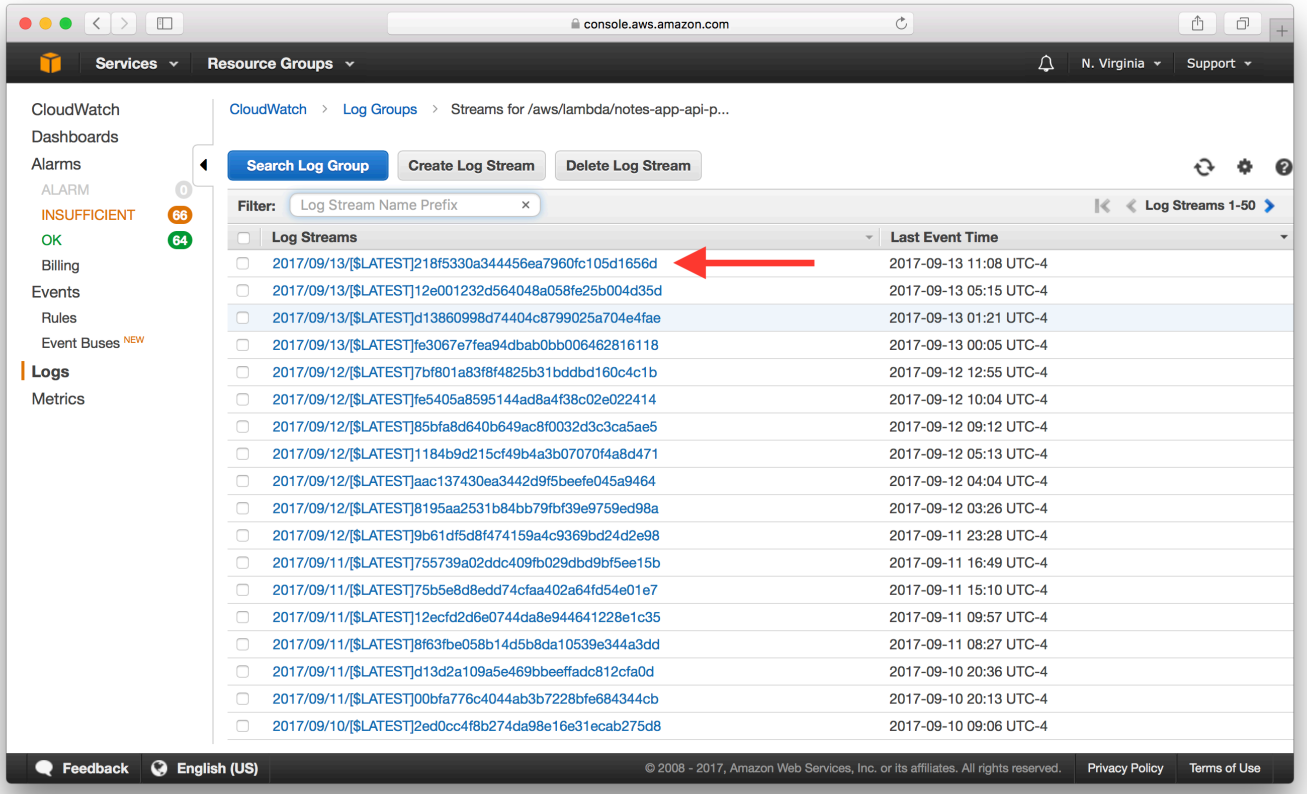
Viewing Lambda CloudWatch Logs

For Lambda, each function has its own log group. And the log stream rotates if a new version of the Lambda function has been deployed or if it has been idle for some time.

To view Lambda logs, select **Logs** again from the left panel. Then select the first log group prefixed with **/aws/lambda/** followed by the function name.

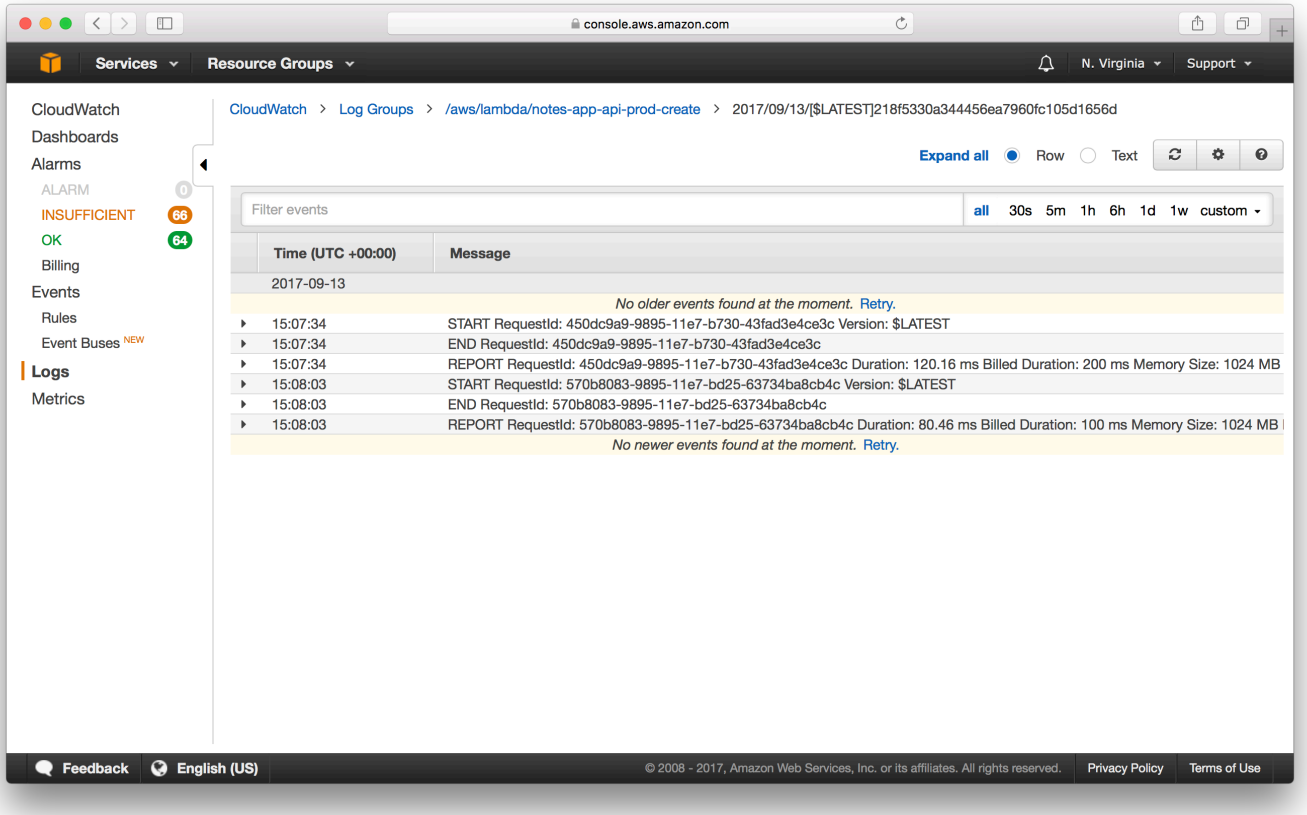


Select the first stream.



You should see **START**, **END** and **REPORT** with basic execution

information for each function invocation. You can also see content logged via `console.log` in your Lambda code.



You can also use the Serverless CLI to view CloudWatch logs for a Lambda function.

From your project root run the following.

```
$ serverless logs -f <func-name>
```

Where the `<func-name>` is the name of the Lambda function you are looking for. Additionally, you can use the `--tail` flag to stream the logs automatically to your console.

```
$ serverless logs -f <func-name> --tail
```

This can be very helpful during development when trying to debug your

functions using the `console.log` call.

Hopefully, this has helped you set up CloudWatch logging for your API Gateway and Lambda projects. And given you a quick idea of how to read your serverless logs using the AWS Console.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/api-gateway-and-lambda-logs/31)

Debugging Serverless API Issues

In this chapter we are going to take a brief look at some common API Gateway and Lambda issues we come across and how to debug them.

We've also compiled a list of some of the most common Serverless errors over on Seed (https://seed.run). Check out Common Serverless Errors (https://seed.run/docs/serverless-errors/) and do a quick search for your error message and see if it has a solution.

When a request is made to your serverless API, it starts by hitting API Gateway and makes its way through to Lambda and invokes your function. It takes quite a few hops along the way and each hop can be a point of failure. And since we don't have great visibility over each of the specific hops, pinpointing the issue can be a bit tricky. We are going to take a look at the following issues:

- Invalid API Endpoint (#invalid-api-endpoint)
- Missing IAM Policy (#missing-iam-policy)
- Lambda Function Error (#lambda-function-error)
- Lambda Function Timeout (#lambda-function-timeout)

This chapter assumes you have turned on CloudWatch logging for API Gateway and that you know how to read both the API Gateway and Lambda logs. If you have not done so, start by taking a look at the chapter on API Gateway and Lambda Logs (/chapters/api-gateway-and-lambda-

logs.html).

Invalid API Endpoint

The first and most basic issue we see is when the API Gateway endpoint that is requested is invalid. An API Gateway endpoint usually looks something like this:

```
https://API_ID.execute-
api.REGION.amazonaws.com/STAGE/PATH
```

- **API_ID** - a unique identifier per API Gateway project
- **REGION** - the AWS region in which the API Gateway project is deployed to
- **STAGE** - the stage of the project (defined in your **serverless.yml** or passed in through the **serverless deploy –stage** command)
- **PATH** - the path of an API endpoint (defined in your **serverless.yml** for each function)

An API request will fail if:

- The **API_ID** is not found in the specified **REGION**
- The API Gateway project does not have the specified **STAGE**
- API endpoint invoked does not match a pre-defined **PATH**

In all of these cases, the error does not get logged to CloudWatch since the request does not hit your API Gateway project.

Missing IAM Policy

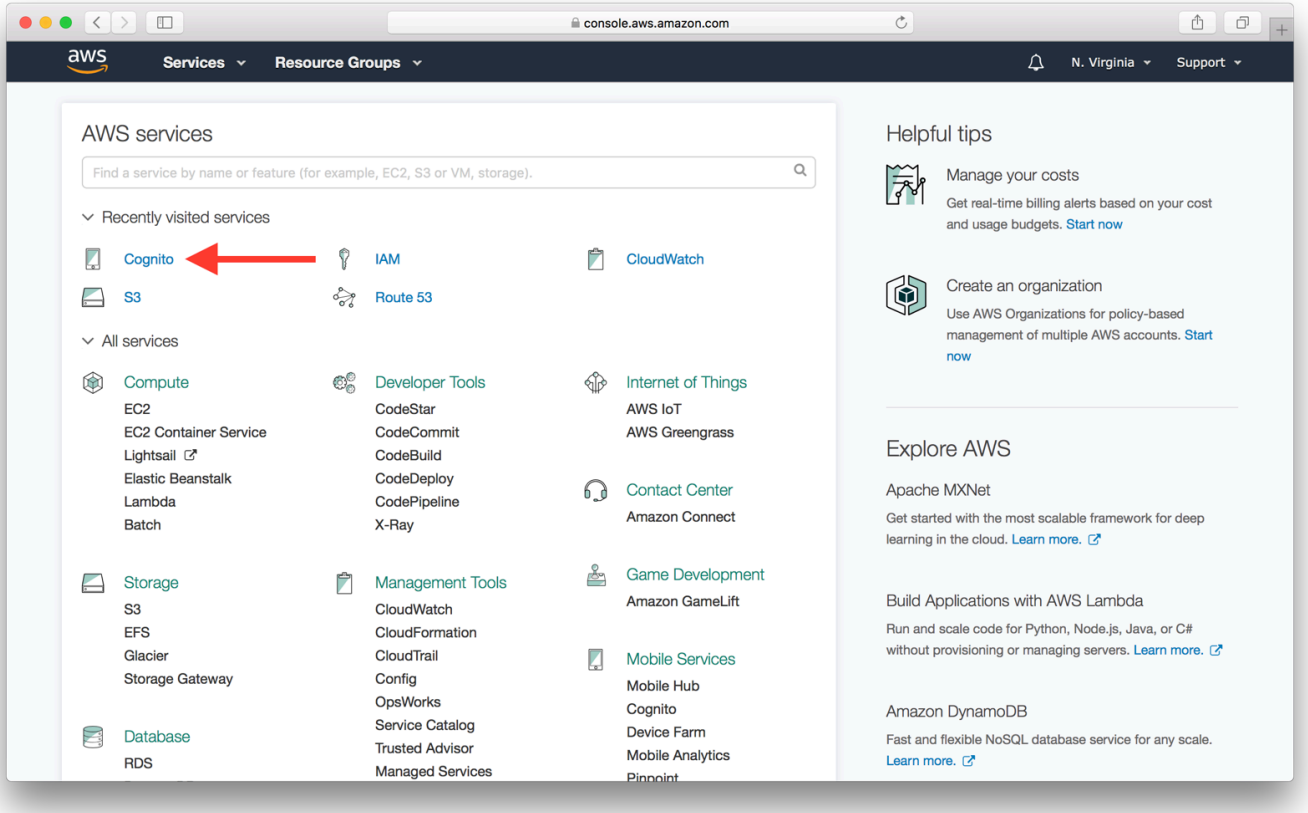
This happens when your API endpoint uses **aws_iam** as the authorizer, and the IAM role assigned to the Cognito Identity Pool has not been granted

the **execute-api:Invoke** permission for your API Gateway resource.

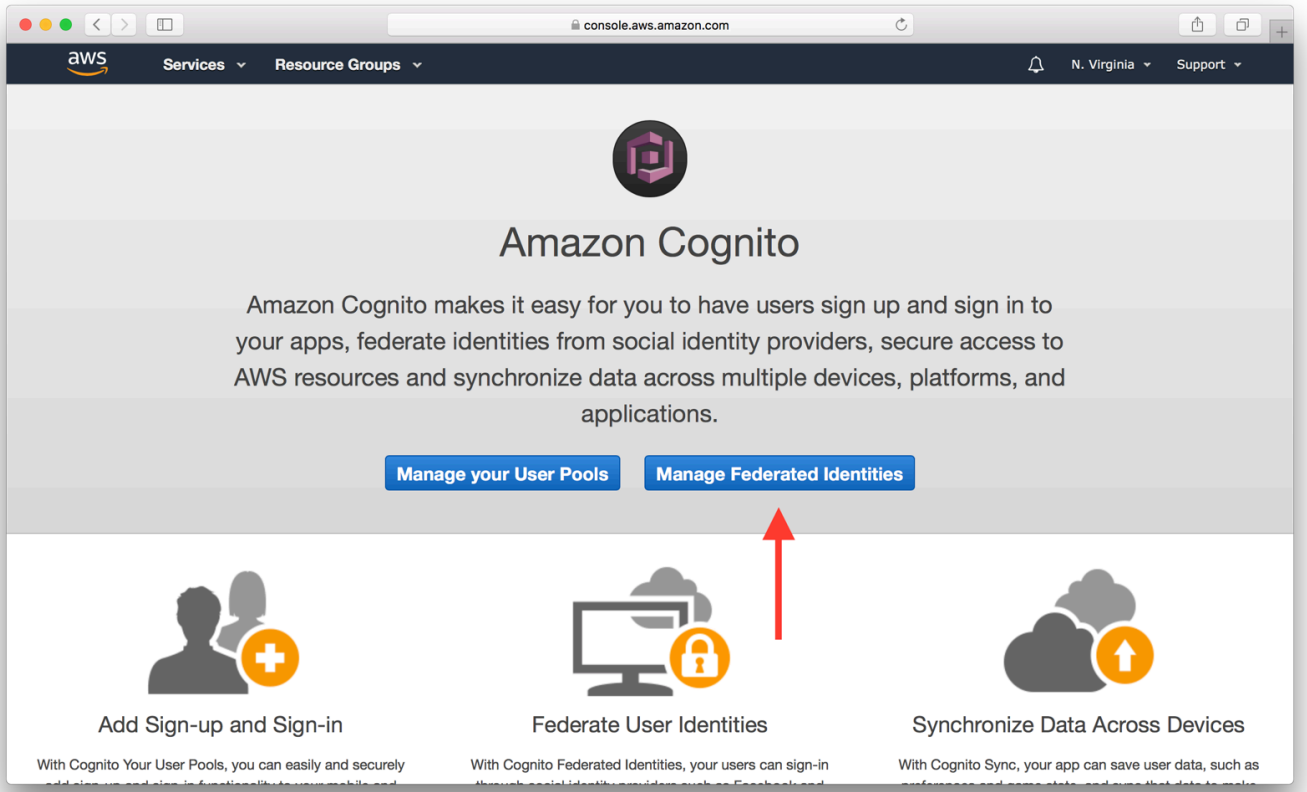
This is a tricky issue to debug because the request still has not reached API Gateway, and hence the error is not logged in the API Gateway CloudWatch logs. But we can perform a check to ensure that our Cognito Identity Pool users have the required permissions, using the IAM policy Simulator (https://policysim.aws.amazon.com).

Before we can use the simulator we first need to find out the name of the IAM role that we are using to connect to API Gateway. We had created this role back in the Create a Cognito identity pool (/chapters/create-a-cognito-identity-pool.html) chapter.

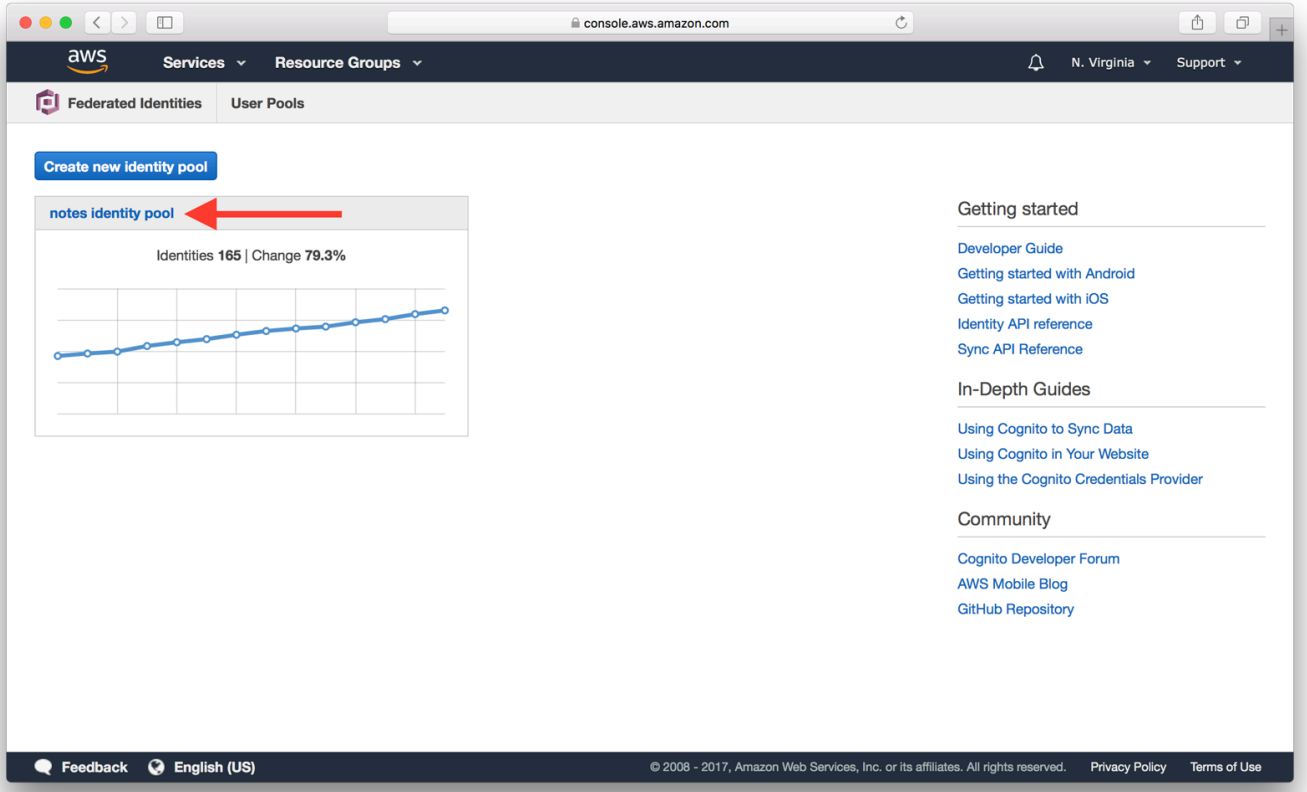
Select **Cognito** from your AWS Console (https://console.aws.amazon.com).



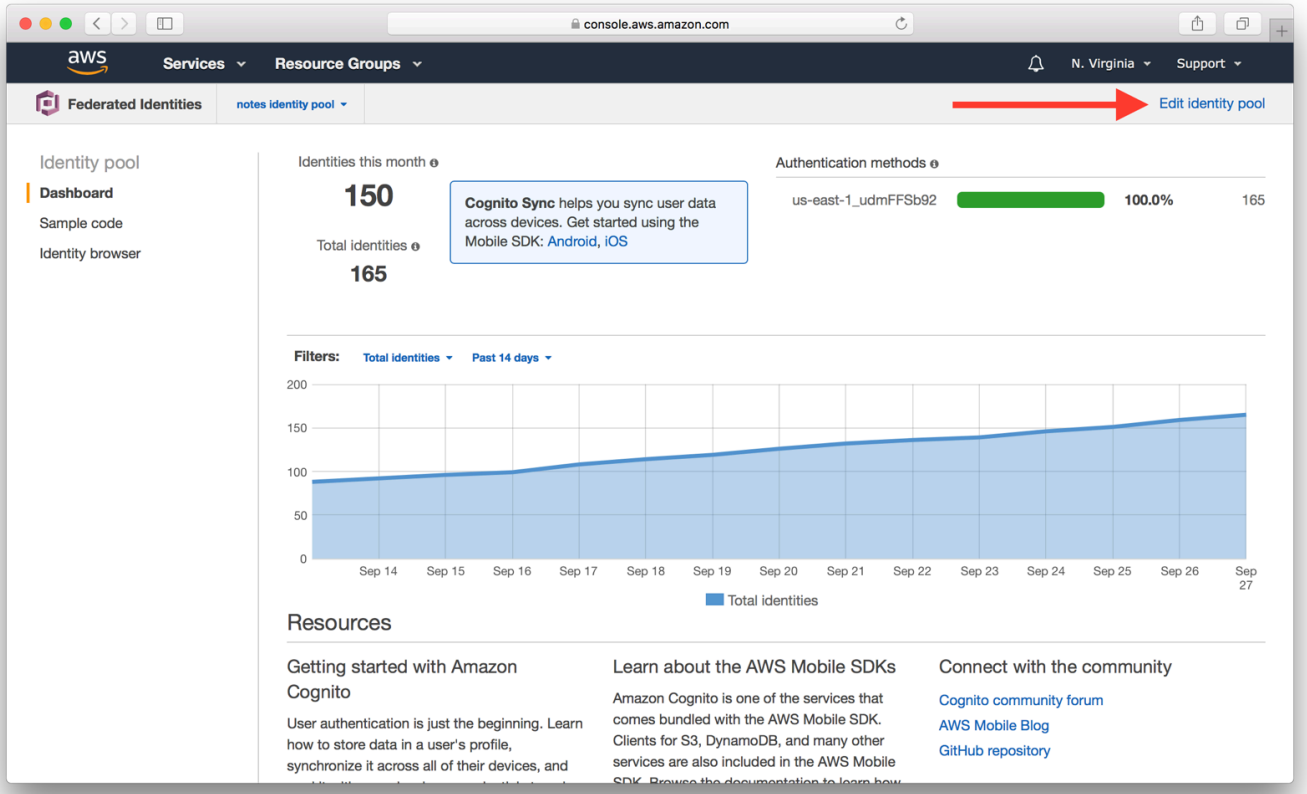
Next hit the **Manage Federated Identities** button.



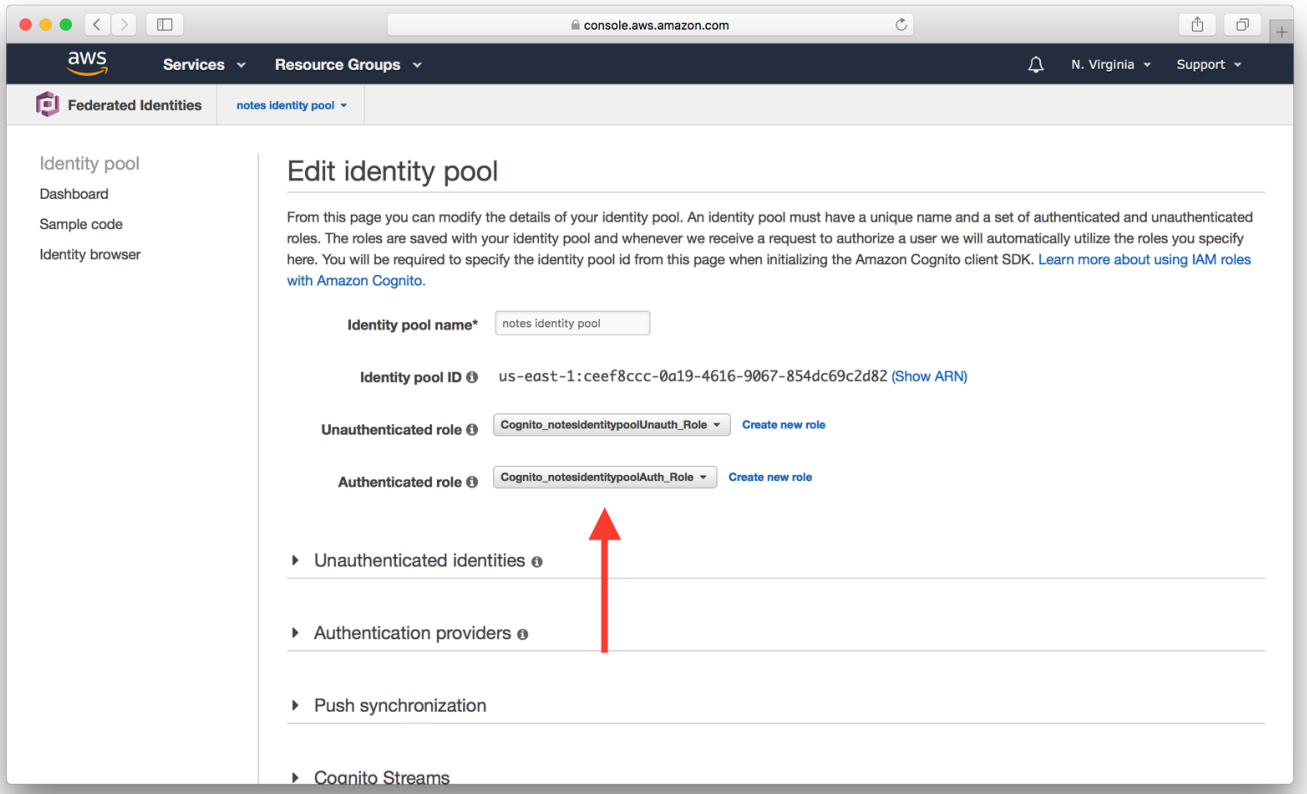
And select your Identity Pool. In our case it's called `notes identity pool`.



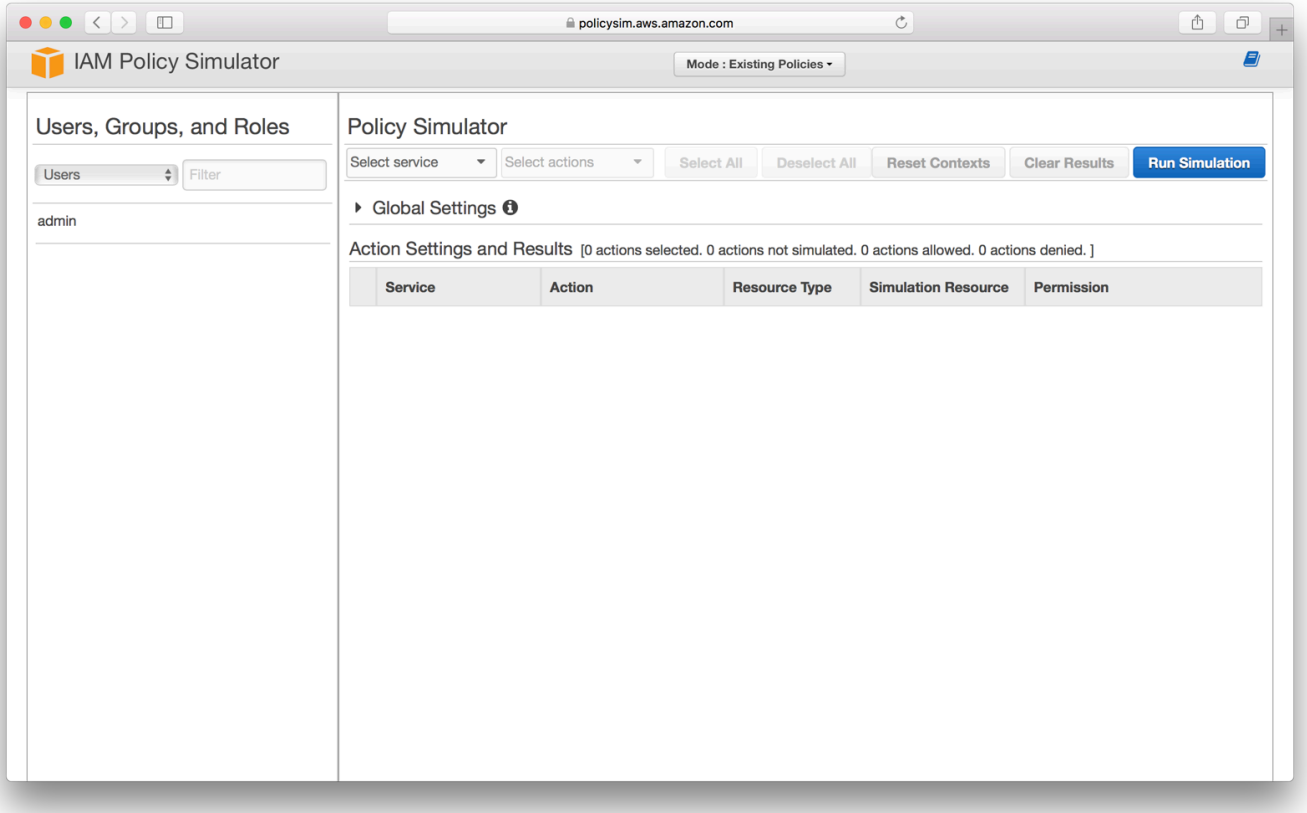
Click **Edit identity pool** at the top right.



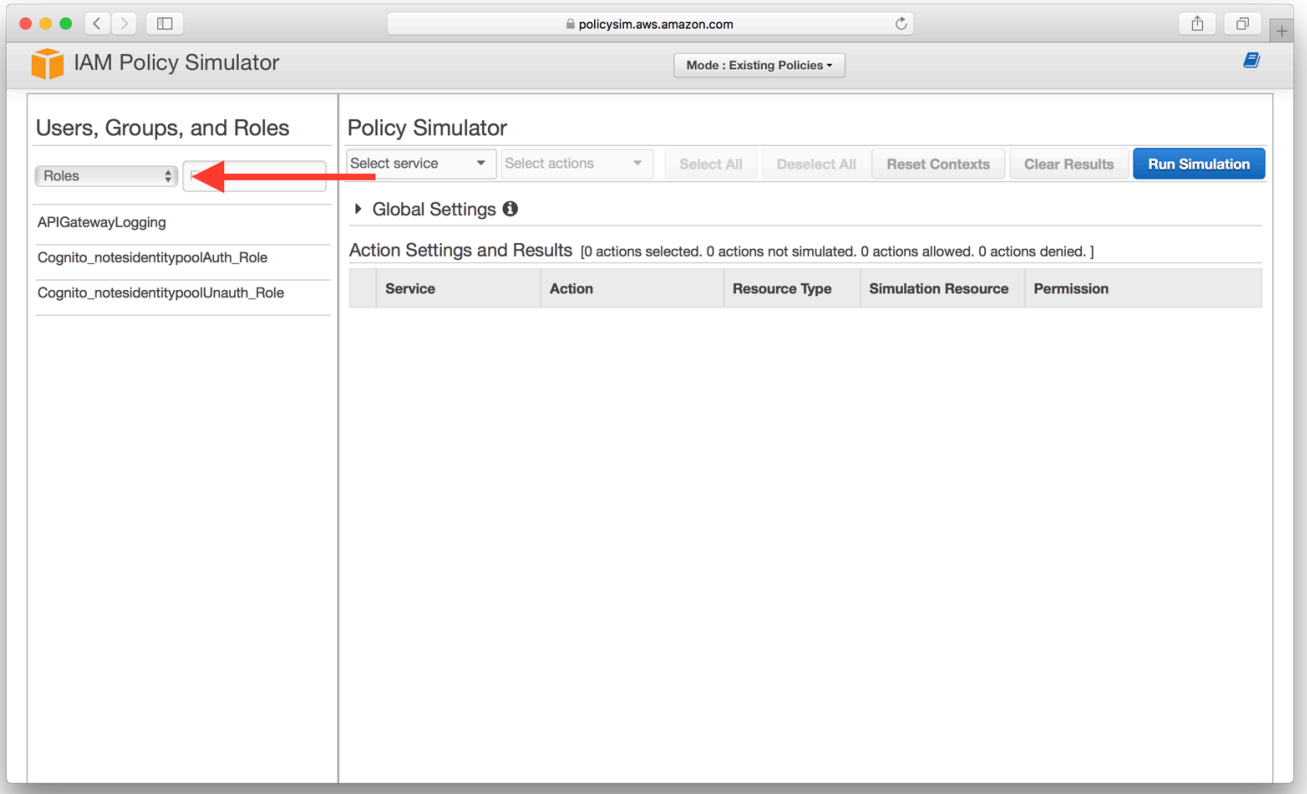
Here make a note of the name of the **Authenticated role**. In our case it is `Cognito_notesidentitypoolAuth_Role`.



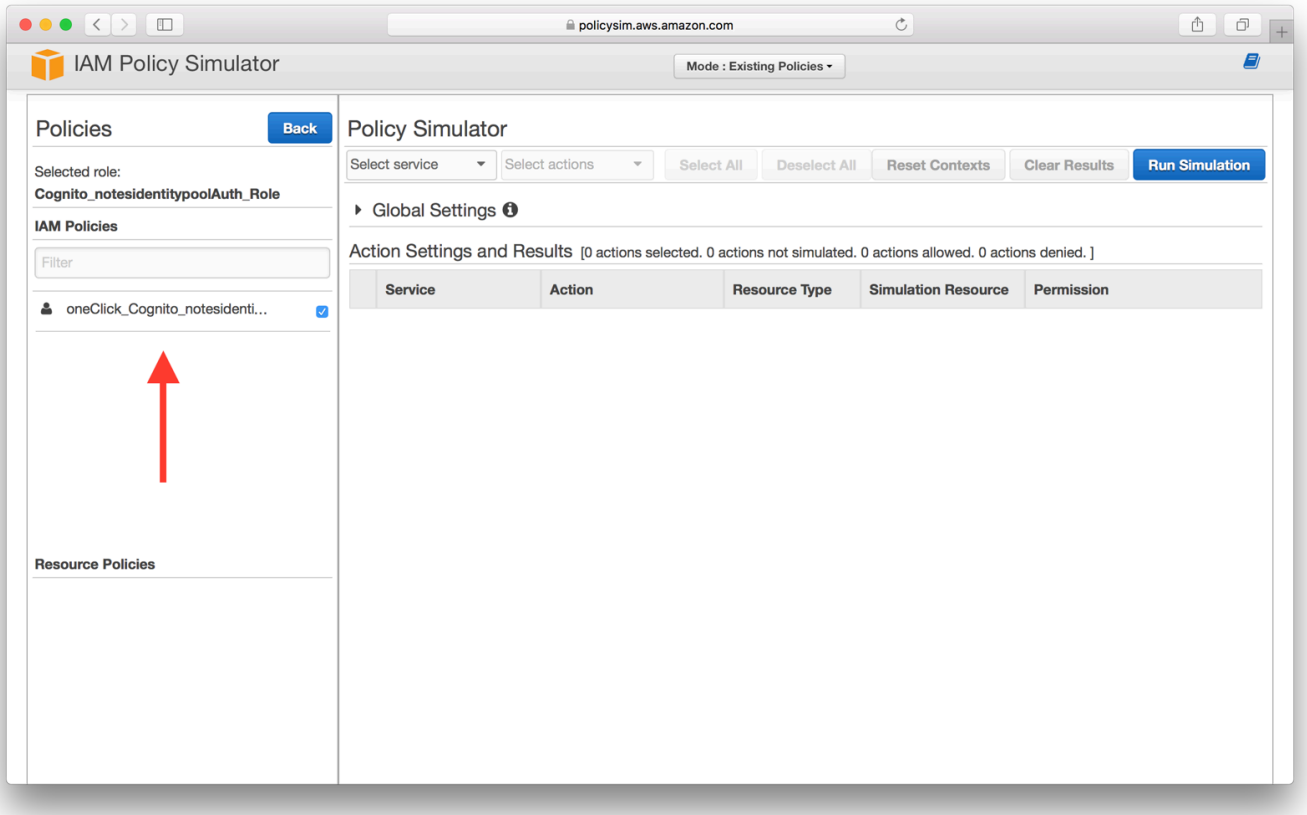
Now that we know the IAM role we are testing, let's open up the IAM Policy Simulator (https://policysim.aws.amazon.com).



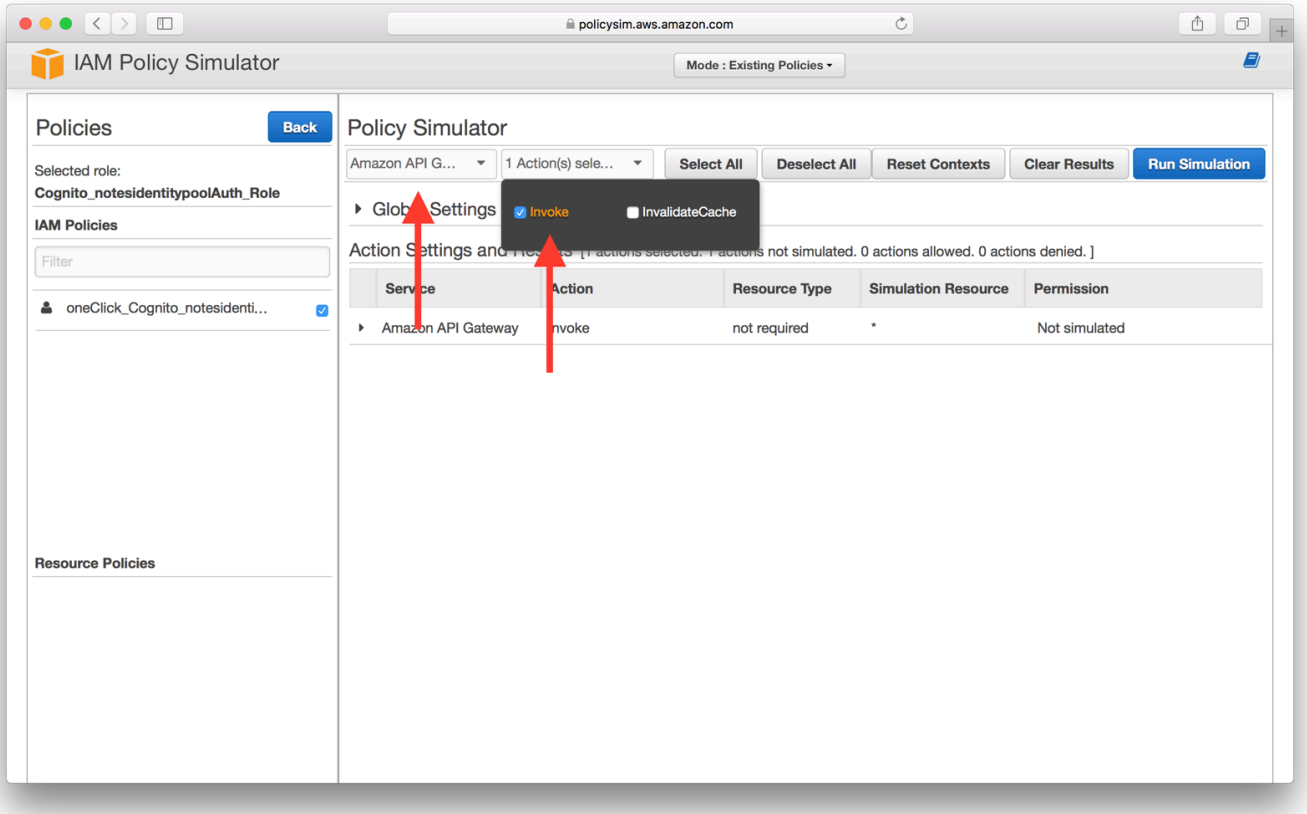
Select **Roles**.



Select the IAM role that we made a note of in the steps above. In our case it is `Cognito_notesidentitypoolAuth_Role`.

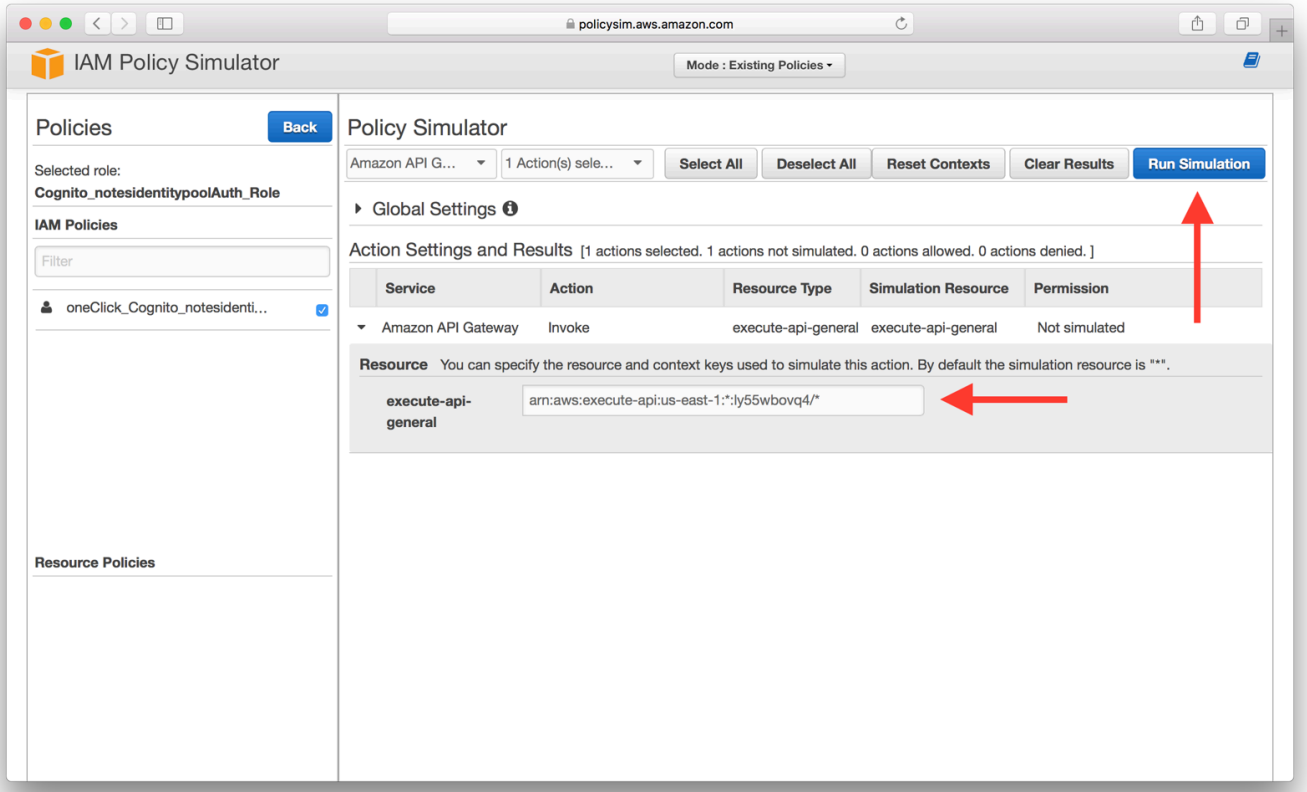


Select **API Gateway** as the service and select the **Invoke** action.

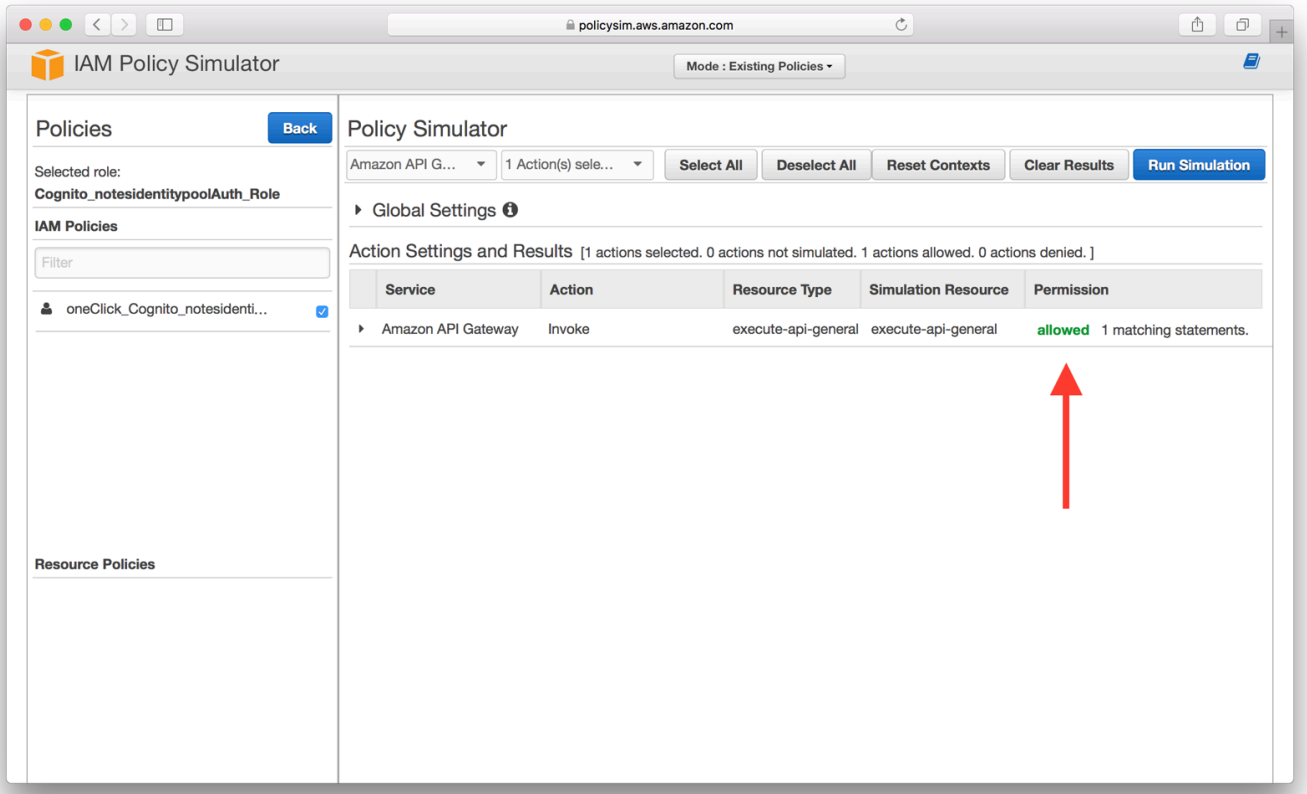


Expand the service and enter the API Gateway endpoint ARN, then select

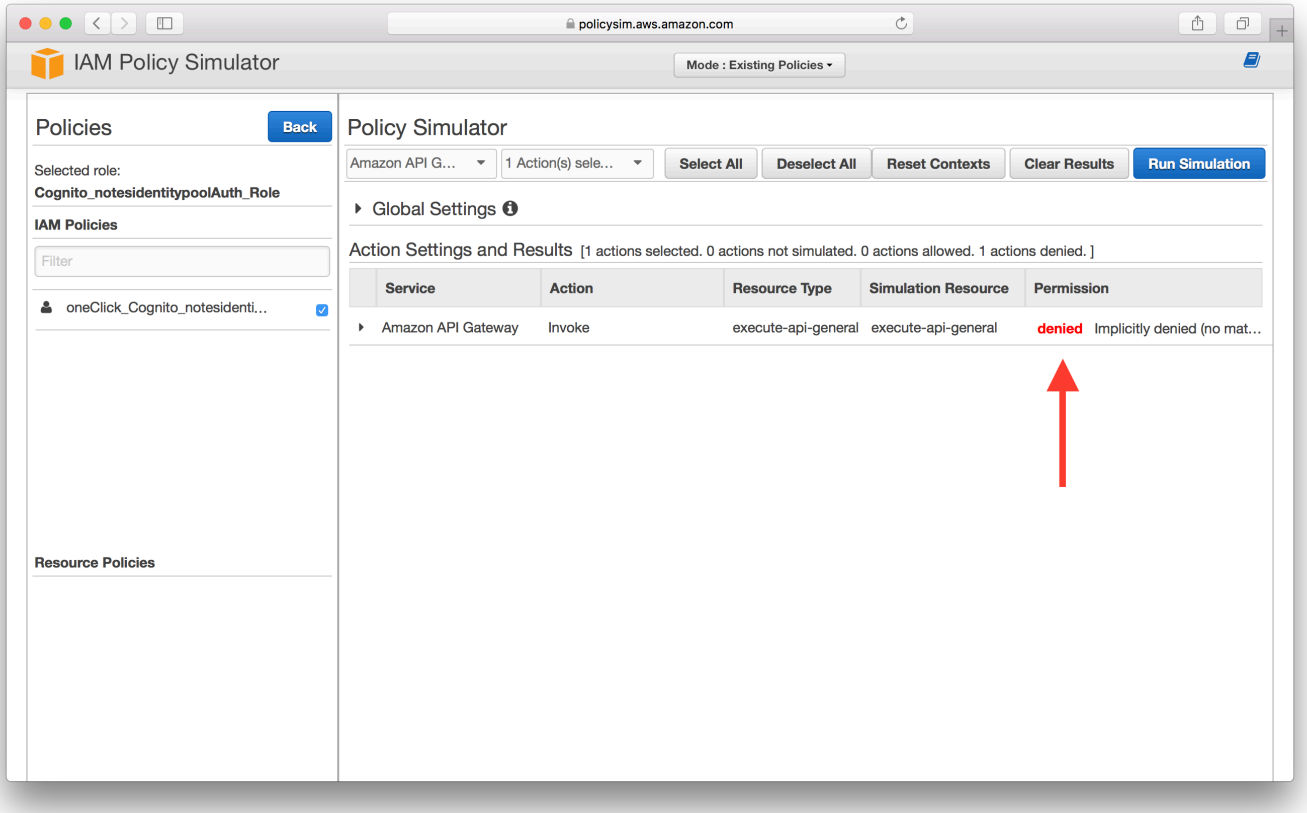
Run Simulation. The format here is the same one we used back in the Create a Cognito identity pool (/chapters/create-a-cognito-identity-pool.html) chapter; `arn:aws:execute-api:YOUR_API_GATEWAY_REGION:*:YOUR_API_GATEWAY_ID/*`. In our case this looks like `arn:aws:execute-api:us-east-1:*:ly55wbovq4/*`.



If your IAM role is configured properly you should see **allowed** under **Permission**.

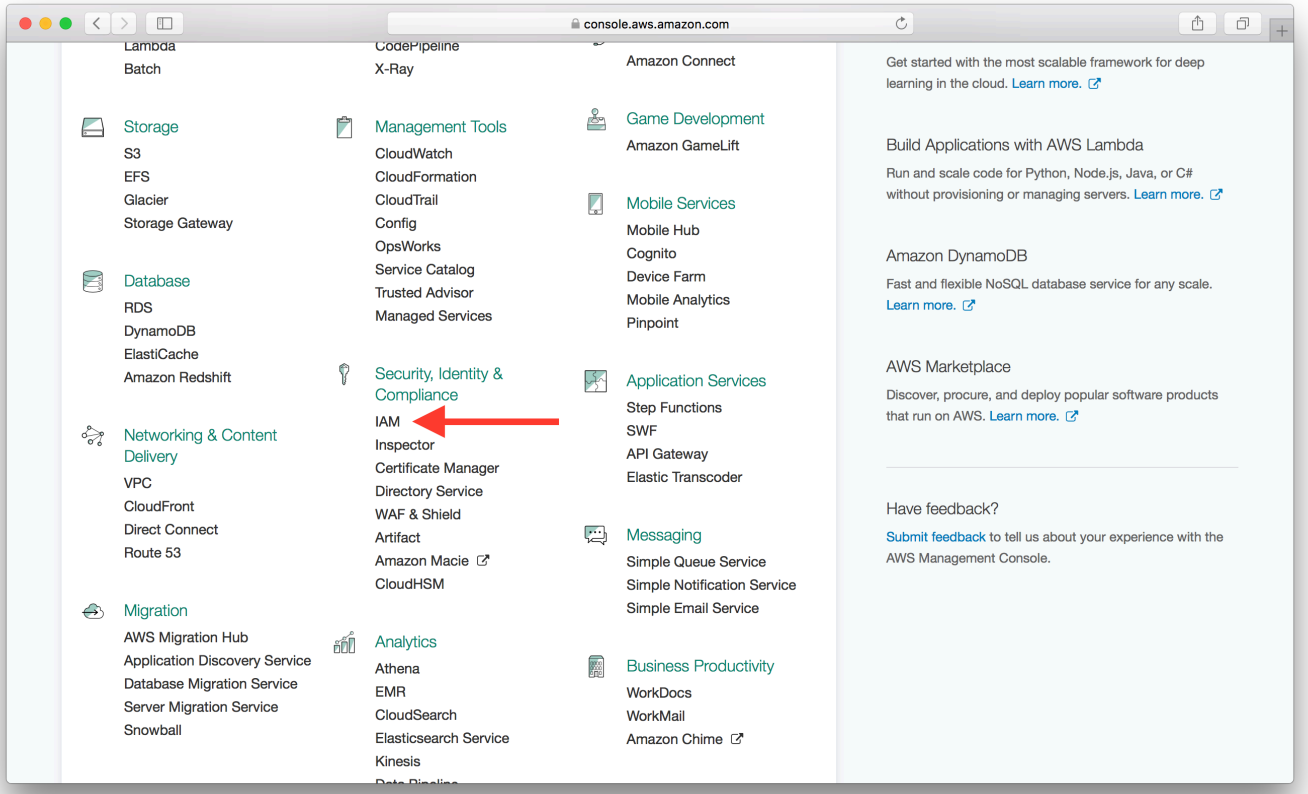


But if something is off, you'll see **denied**.

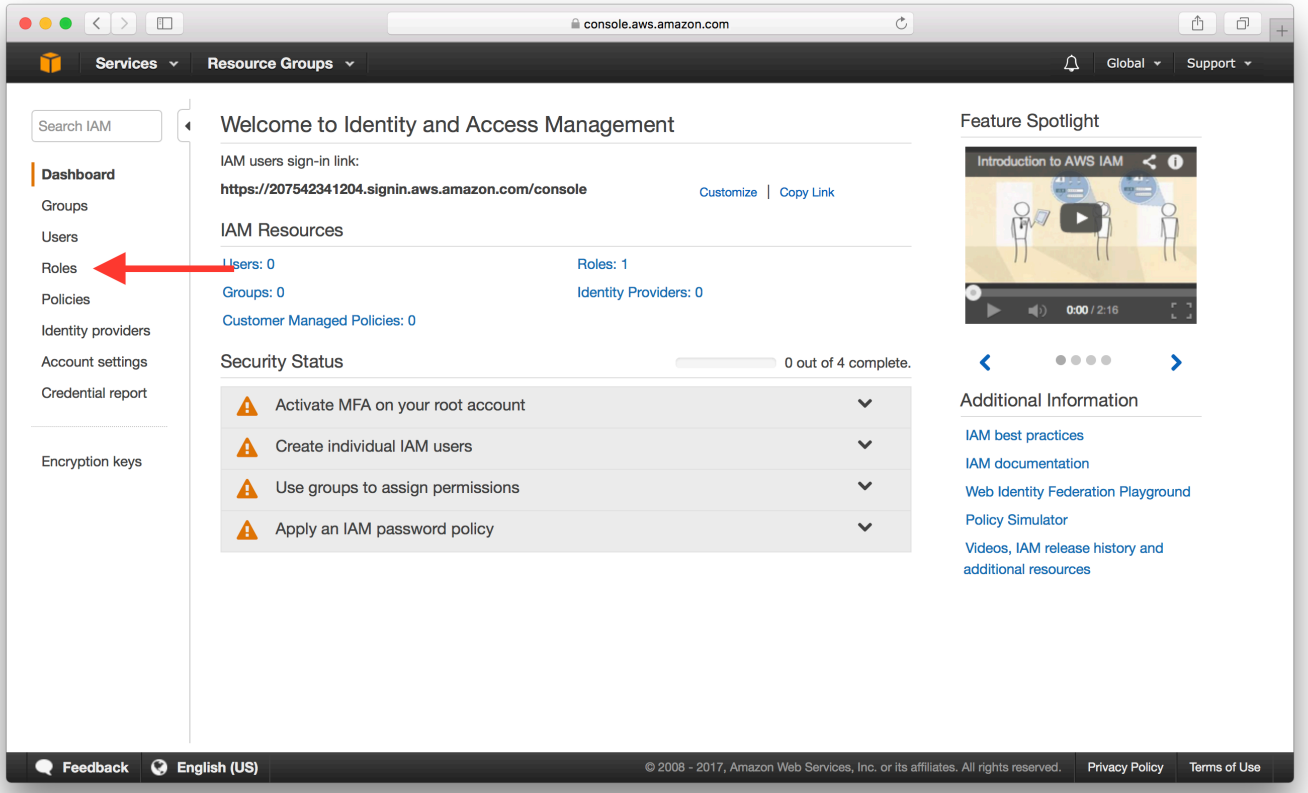


To fix this and edit the role we need to go back to the AWS Console

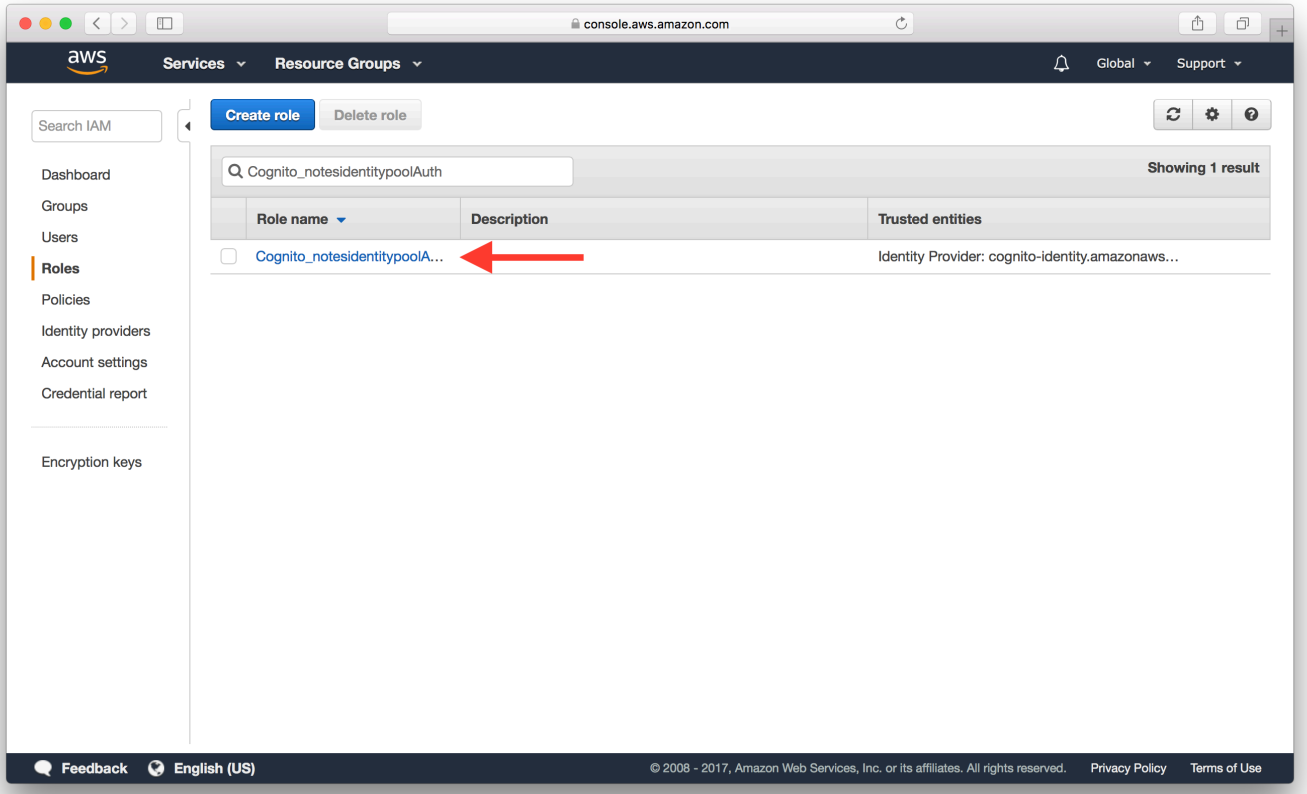
(https://console.aws.amazon.com) and select IAM from the list of services.



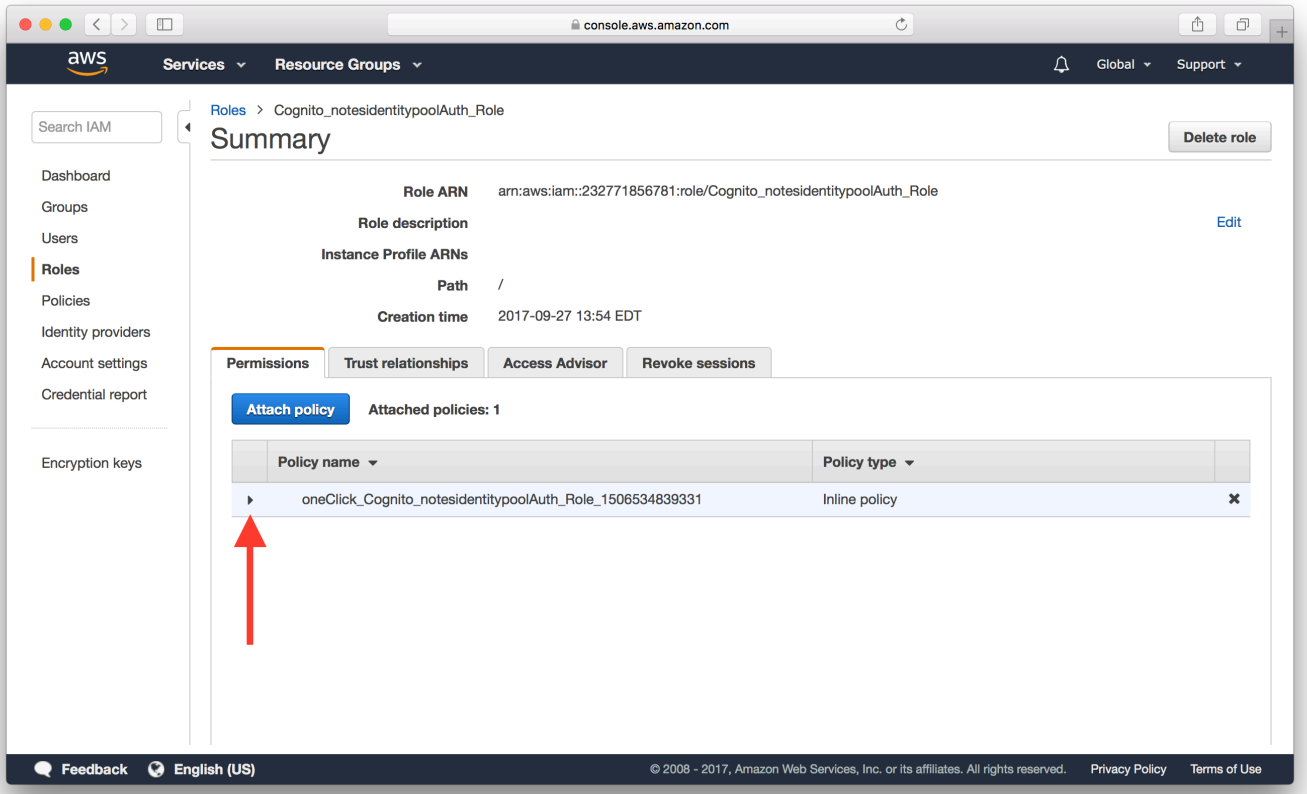
Select **Roles** on the left menu.



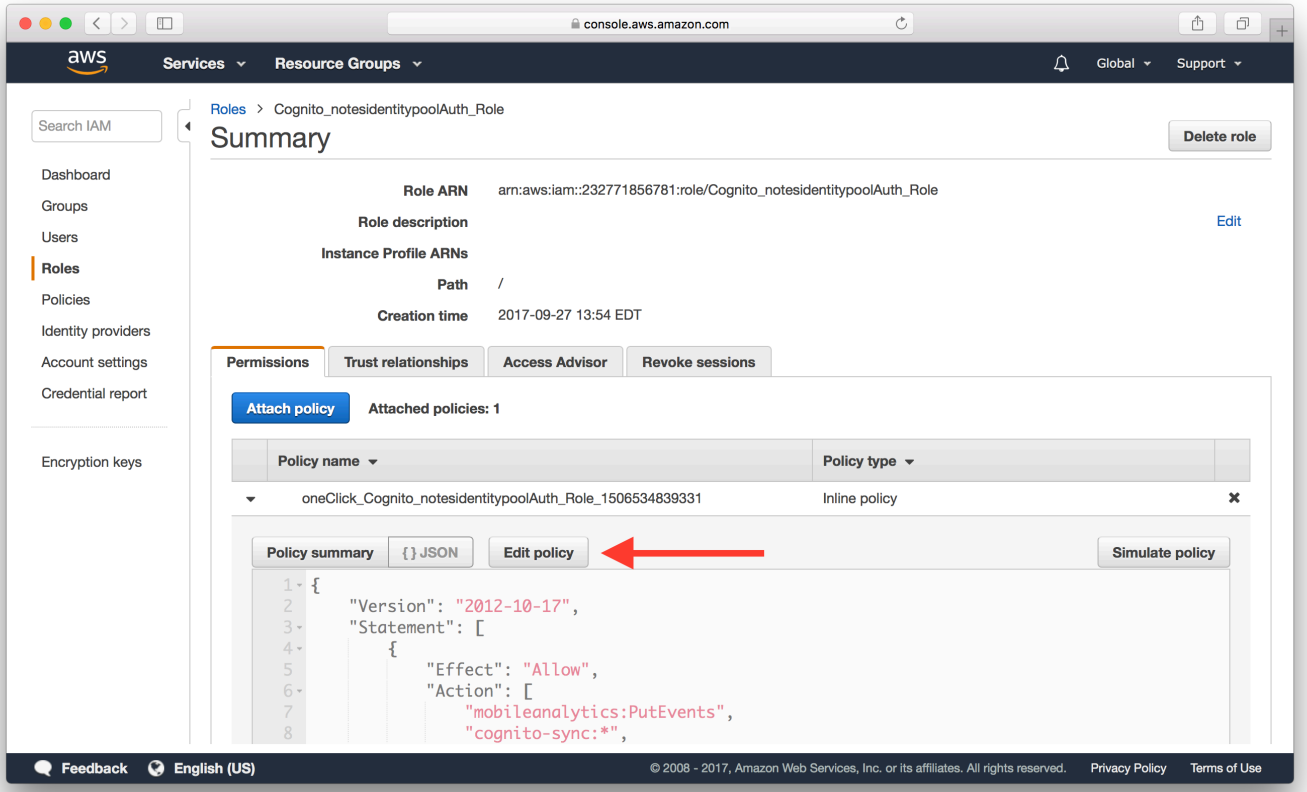
And select the IAM role that our Identity Pool is using. In our case it's called `Cognito_notesidentitypoolAuth_Role`.



Expand the policy under the list of policies.



Click **Edit policy**.

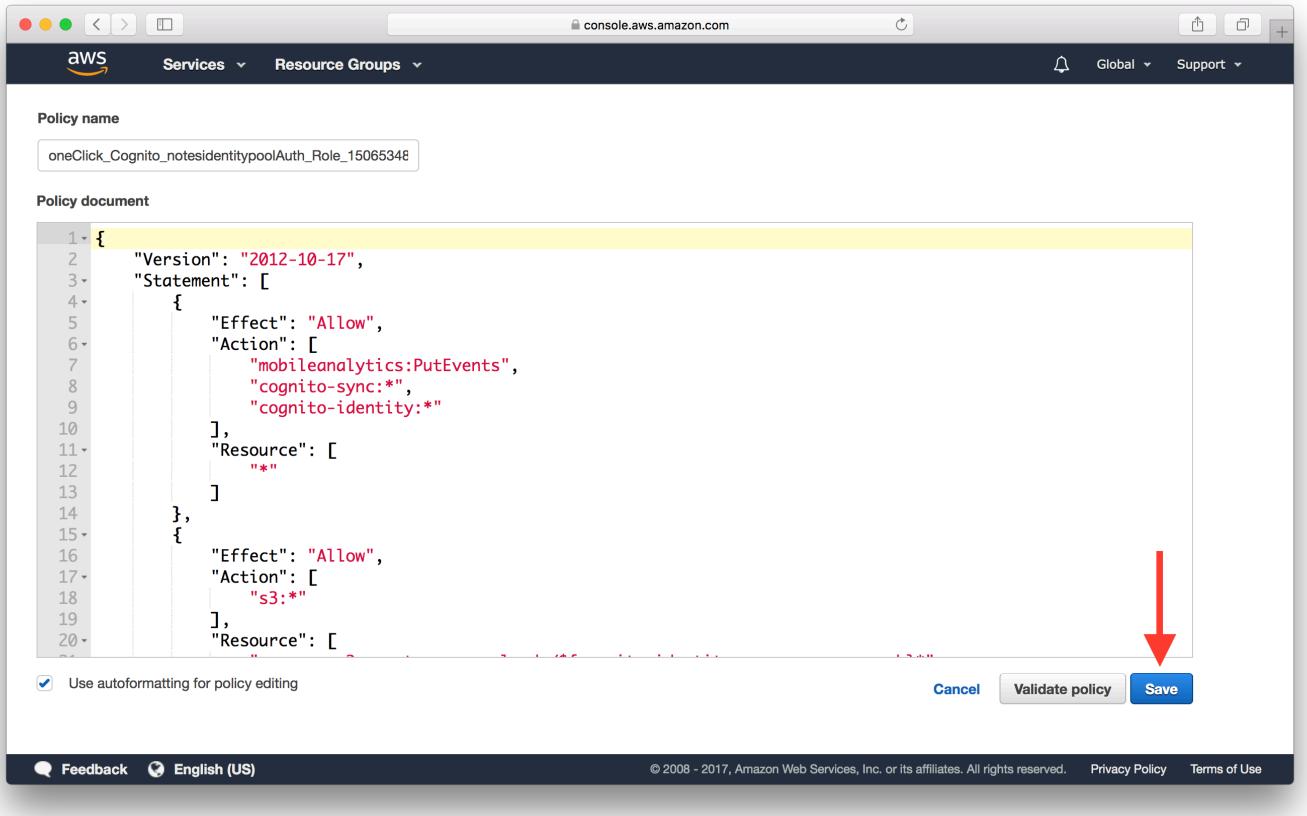


Here you can edit the policy to ensure that it has the right permission to

invoke API Gateway. Ensure that there is a block in your policy like the one below.

```
...
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-
api:YOUR_API_GATEWAY_REGION:*:YOUR_API_GATEWAY_ID/*"
      ]
    }
...
```

Finally, hit **Save** to update the policy.



Now if you test your policy, it should show that you are allowed to invoke your API Gateway endpoint.

Lambda Function Error

Now if you are able to invoke your Lambda function but it fails to execute properly due to uncaught exceptions, it'll error out. These are pretty straightforward to debug. When this happens, AWS Lambda will attempt to convert the error object to a string, and then send it to CloudWatch along with the stacktrace. This can be observed in both Lambda and API Gateway CloudWatch log groups.

Lambda Function Timeout

Sometimes we might run into a case where the Lambda function just times out. Normally, a Lambda function will end its execution by invoking the **callback** function that was passed in. By default, the callback will wait until the Node.js runtime event loop is empty before returning the results to the caller. If the Lambda function has an open connection to, let's say a database server, the event loop is not empty, and the callback will wait indefinitely until the connection is closed or the Lambda function times out.

To get around this issue, you can set this **callbackWaitsForEmptyEventLoop** property to false to request AWS Lambda to freeze the process as soon as the callback is called, even if there are events in the event loop.

```
export async function handler(event, context, callback)
{
```

```
  context.callbackWaitsForEmptyEventLoop = false;

  ...
};
```

This effectively allows a Lambda function to return its result to the caller without requiring that the database connection be closed. This allows the Lambda function to reuse the same connection across calls, and it reduces the execution time as well.

These are just a few of the common issues we see folks running into while working with serverless APIs. Feel free to let us know via the comments if there are any other issues you'd like us to cover.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/debugging-serverless-api-issues/143)

Serverless Environment Variables

In Node.js we use the `process.env` to get access to environment variables of the current process. In AWS Lambda, we can set environment variables that we can access via the `process.env` object.

Let's take a quick look at how to do that.

Defining Environment Variables

We can define our environment variables in our `serverless.yml` in two separate places. The first is in the `functions` section:

```
service: service-name

provider:
  name: aws
  stage: dev

functions:
  hello:
    handler: handler.hello
    environment:
      SYSTEM_URL: http://example.com/api/v1
```

Here `SYSTEM_URL` is the name of the environment variable we are defining and `http://example.com/api/v1` is its value. We can access this in our `hello` Lambda function using `process.env.SYSTEM_URL`, like so:

```
export function hello(event, context, callback) {
  callback(null, { body: process.env.SYSTEM_URL });
}
```

We can also define our environment variables globally in the `provider` section:

```
service: service-name

provider:
  name: aws
  stage: dev
  environment:
    SYSTEM_ID: jdoe

functions:
  hello:
    handler: handler.hello
    environment:
      SYSTEM_URL: http://example.com/api/v1
```

Just as before we can access the environment variable `SYSTEM_ID` in our `hello` Lambda function using `process.env.SYSTEM_ID`. The difference being that it is available to **all** the Lambda functions defined in our `serverless.yml`.

In the case where both the `provider` and `functions` section has an environment variable with the same name, the function specific environment variable takes precedence. As in, we can override the environment variables described in the `provider` section with the ones defined in the `functions` section.

Custom Variables in Serverless Framework

Serverless Framework builds on these ideas to make it easier to define and work with environment variables in our `serverless.yml` by generalizing the idea of variables (https://serverless.com/framework/docs/providers/aws/guide/variables/).

Let's take a quick look at how these work using an example. Say you had the following `serverless.yml`.

```
service: service-name

provider:
  name: aws
  stage: dev

functions:
  helloA:
    handler: handler.helloA
    environment:
      SYSTEM_URL: http://example.com/api/v1/pathA

  helloB:
    handler: handler.helloB
    environment:
```

```
      SYSTEM_URL: http://example.com/api/v1/pathB
```

In the case above we have the environment variable `SYSTEM_URL` defined in both the `helloA` and `helloB` Lambda functions. But the only difference between them is that the url ends with `pathA` or `pathB`. We can merge these two using the idea of variables.

A variable allows you to replace values in your `serverless.yml` dynamically. It uses the `${variableName}` syntax, where the value of `variableName` will be inserted.

Let's see how this works in practice. We can rewrite our example and simplify it by doing the following:

```
service: service-name

custom:
  systemUrl: http://example.com/api/v1/

provider:
  name: aws
  stage: dev

functions:
  helloA:
    handler: handler.helloA
    environment:
      SYSTEM_URL: ${self:custom.systemUrl}pathA

  helloB:
    handler: handler.helloB
```

```
    environment:
      SYSTEM_URL: ${self:custom.systemUrl}pathB
```

This should be pretty straightforward. We started by adding this section first:

```
custom:
  systemUrl: http://example.com/api/v1/
```

This defines a variable called `systemUrl` under the section `custom`. We can then reference the variable using the syntax `${self:custom.systemUrl}`.

We do this in the environment variables `SYSTEM_URL: ${self:custom.systemUrl}pathA`. Serverless Framework parses this and inserts the value of `self:custom.systemUrl` and that combined with `pathA` at the end gives us the original value of `http://example.com/api/v1/pathA`.

Variables can be referenced from a lot of different sources including CLI options, external YAML files, etc. You can read more about using variables in your `serverless.yml` here (https://serverless.com/framework/docs/providers/aws/guide/variables/).

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/serverless-environment-variables/25)

Stages in Serverless Framework

Serverless Framework allows you to create stages for your project to deploy to. Stages are useful for creating environments for testing and development. Typically you create a staging environment that is an independent clone of your production environment. This allows you to test and ensure that the version of code that you are about to deploy is good to go.

In this chapter we will take a look at how to configure stages in Serverless. Let's first start by looking at how stages can be implemented.

How Is Staging Implemented?

There are a couple of ways to set up stages for your project:

- Using the API Gateway built-in stages

  You can create multiple stages within a single API Gateway project. Stages within the same project share the same endpoint host, but have a different path. For example, say you have a stage called `prod` with the endpoint:

  ```
  https://abc12345.execute-api.us-east-
  1.amazonaws.com/prod
  ```

If you were to add a stage called `dev` to the same API Gateway API, the new stage will have the endpoint:

```
https://abc12345.execute-api.us-east-
1.amazonaws.com/dev
```

The downside is that both stages are part of the same project. You don't have the same level of flexibility to fine tune the IAM policies for stages of the same API, when compared to tuning different APIs. This leads to the next setup, each stage being its own API.

- Separate APIs for each stage

  You create an API Gateway project for each stage. Let's take the same example, your `prod` stage has the endpoint:

  ```
  https://abc12345.execute-api.us-east-
  1.amazonaws.com/prod
  ```

  To create the `dev` stage, you create a new API Gateway project and add the `dev` stage to the new project. The new endpoint will look something like:

  ```
  https://xyz67890.execute-api.us-east-
  1.amazonaws.com/dev
  ```

  Note that the `dev` stage carries a different endpoint host since it belongs to a different project. This is the approach Serverless Framework takes when configuring stages for your Serverless project. We will look at this in detail below.

- Separate AWS account for each stage

  Just like how having each stage being separate APIs give us more flexibility to fine tune the IAM policy. We can take it a step further and create the API project in a different AWS account. Most companies don't keep their production infrastructure in the same account as their development infrastructure. This helps reduce any cases where developers accidentally edit/delete production resources. We go in to more detail on how to deploy to multiple AWS accounts using different AWS profiles in the Configure Multiple AWS Profiles (/chapters/configure-multiple-aws-profiles.html) chapter.

Deploying to a Stage

Let's look at how the Serverless Framework helps us work with stages. As mentioned above, a new stage is a new API Gateway project. To deploy to a specific stage, you can either specify the stage in the `serverless.yml`.

```
service: service-name

provider:
  name: aws
  stage: dev
```

Or you can specify the stage by passing the `--stage` option to the `serverless deploy` command.

```
$ serverless deploy --stage dev
```

Stage Variables in Serverless Framework

Deploying to stages can be pretty simple but now let's look at how to configure our environment variables so that they work with our various stages. We went over the concept of environment variables in the chapter on Serverless Environment Variables (/chapters/serverless-environment-variables.html). Let's extend that to specify variables based on the stage we are deploying to.

Let's take a look at a sample `serverless.yml` below.

```
service: service-name

custom:
  myStage: ${opt:stage, self:provider.stage}
  myEnvironment:
    MESSAGE:
      prod: "This is production environment"
      dev: "This is development environment"

provider:
  name: aws
  stage: dev
  environment:
    MESSAGE:
${self:custom.myEnvironment.MESSAGE.${self:custom.myStage
```

There are a couple of things happening here. We first defined the `custom.myStage` variable as `${opt:stage, self:provider.stage}`. This is telling Serverless Framework to use the

`--stage` CLI option if it exists. And if it does not, then use the default stage specified by `provider.stage`. We also define the `custom.myEnvironment` section. This contains the value for `MESSAGE` defined for each stage. Finally, we set the environment variable `MESSAGE` as `${self:custom.myEnvironment.MESSAGE.${self:custom.myStage}}`. This sets the variable to pick the value of `self:custom.myEnvironment` depending on the current stage defined in `custom.myStage`.

You can easily extend this format to create separate sets of environment variables for the stages you are deploying to.

And we can access the `MESSAGE` in our Lambda functions via `process.env` object like so.

```
export function main(event, context, callback) {
  callback(null, { body: process.env.MESSAGE });
}
```

Hopefully, this chapter gives you a quick idea on how to set up stages in your Serverless project.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/stages-in-serverless-framework/35)

Backups in DynamoDB

An important (yet overlooked) aspect of having a database powering your web application are, backups! In this chapter we are going to take a look at how to configure backups for your DynamoDB tables.

For our demo notes app (https://demo.serverless-stack.com), we are using a DynamoDB table to store all our user's notes. DynamoDB achieves a high degree of data availability and durability by replicating your data across three different facilities within a given region. However, DynamoDB does not provide an SLA for the data durability. This means that you should backup your database tables.

Let's start by getting a quick background on how backups work in DynamoDB.

Backups in DynamoDB

There are two types of backups in DynamoDB:

1. **On-demand backups**

   This creates a full backup on-demand of your DynamoDB tables. It's useful for long-term data retention and archival. The backup is retained even if the table is deleted. You can use the backup to restore to a different table name. And this can make it useful for replicating tables.

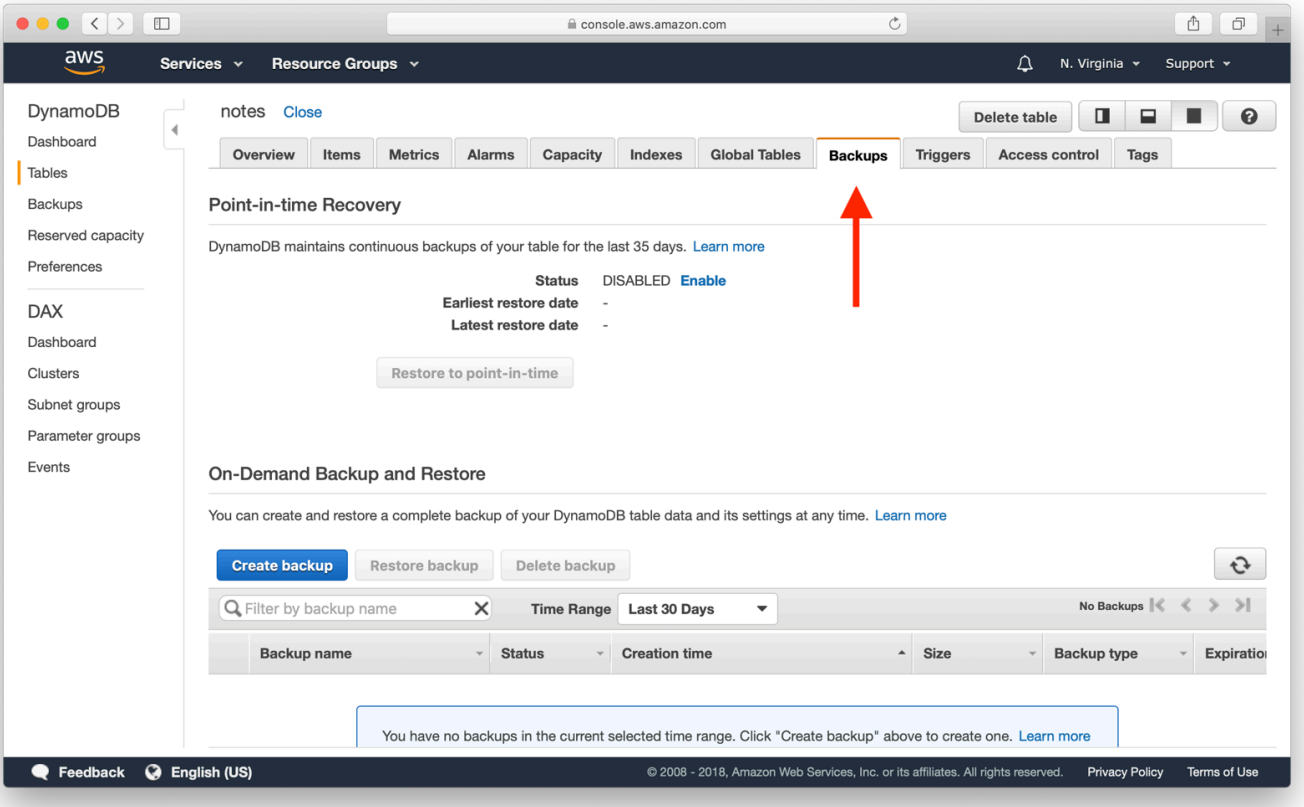
2. **Point-in-time recovery**

This type of backup on the other hand allows you to perform point-in-time restore. It's really helpful in protecting against accidental writes or delete operations. So for example, if you ran a script to transform the data within a table and it accidentally removed or corrupted your data; you could simply restore your table to any point in the last 35 days. DynamoDB does this by maintaining an incremental backup of your table. It even does this automatically, so you don't have to worry about creating, maintaining, or scheduling on-demand backups.

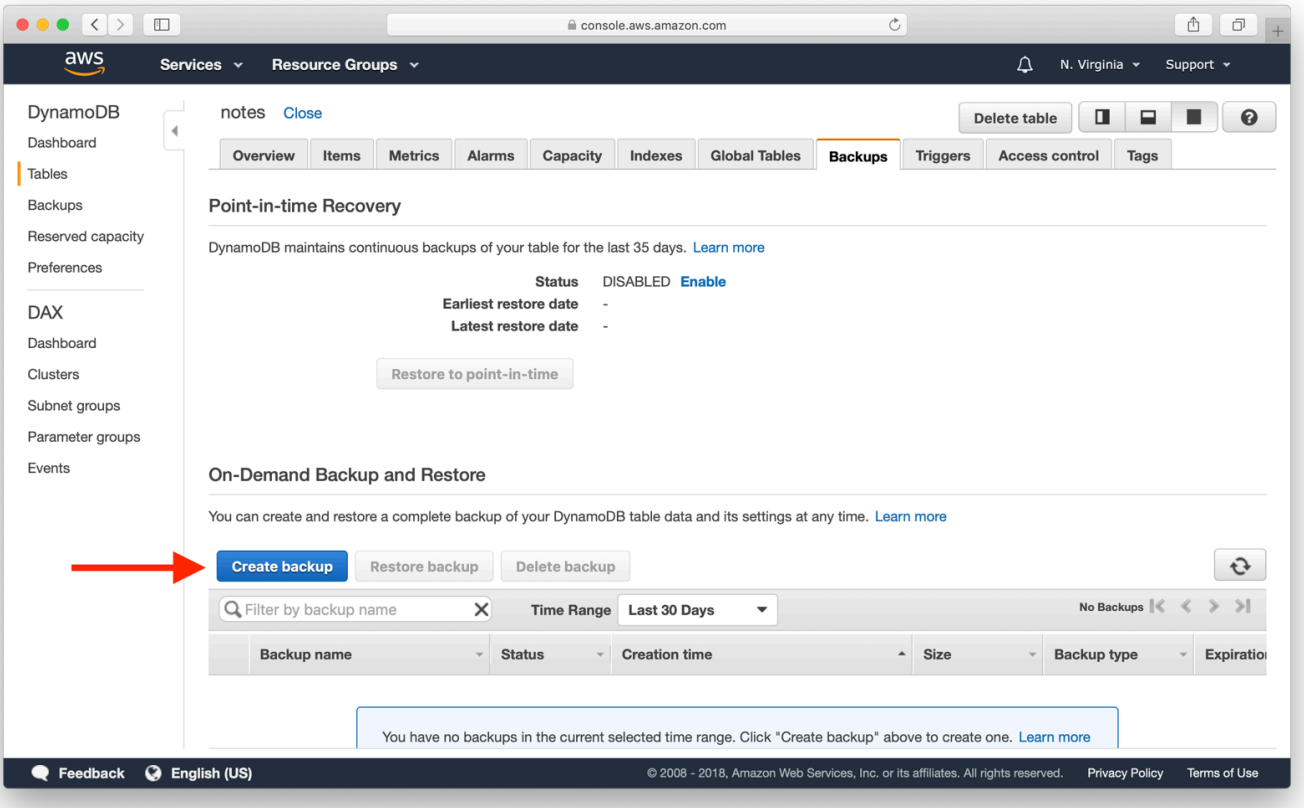
Let's look at how to use the two backup types.

On-Demand Backup

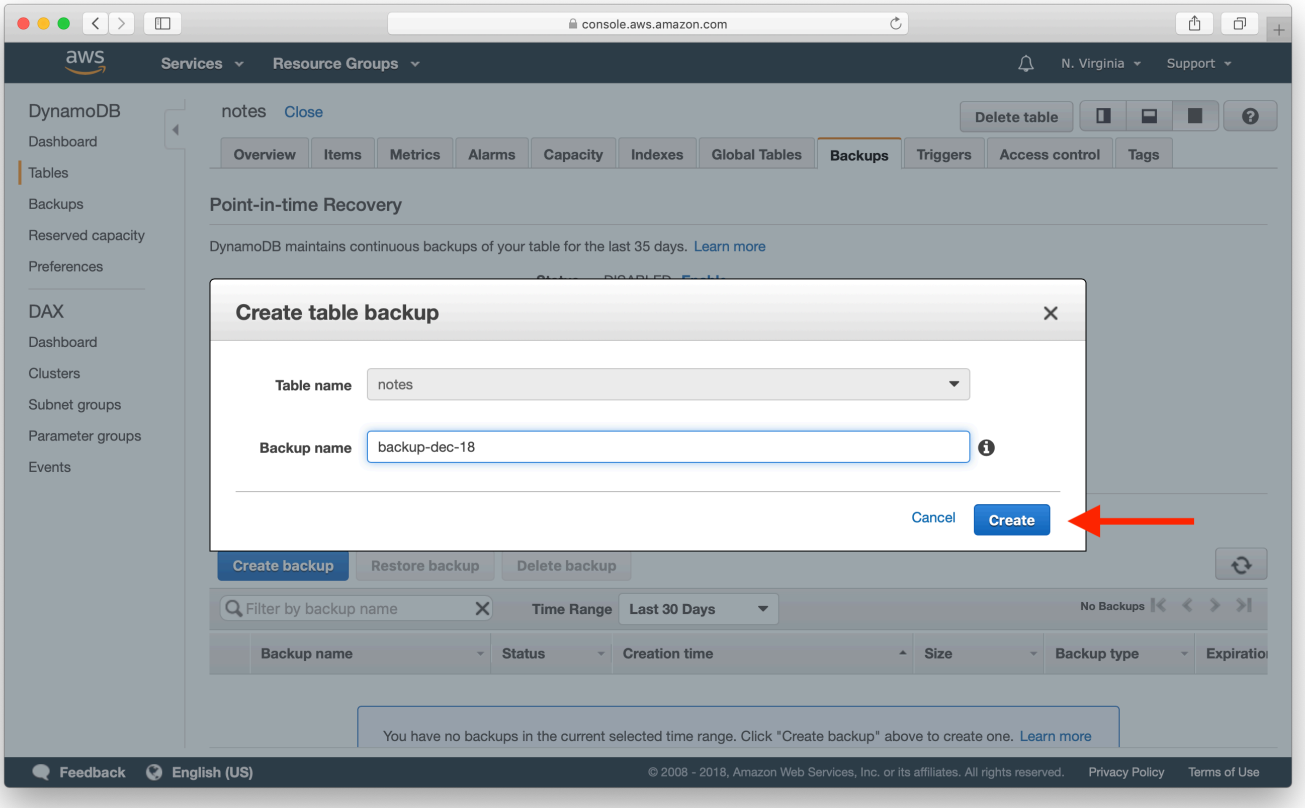
Head over to your table and click on the **Backups** tab.



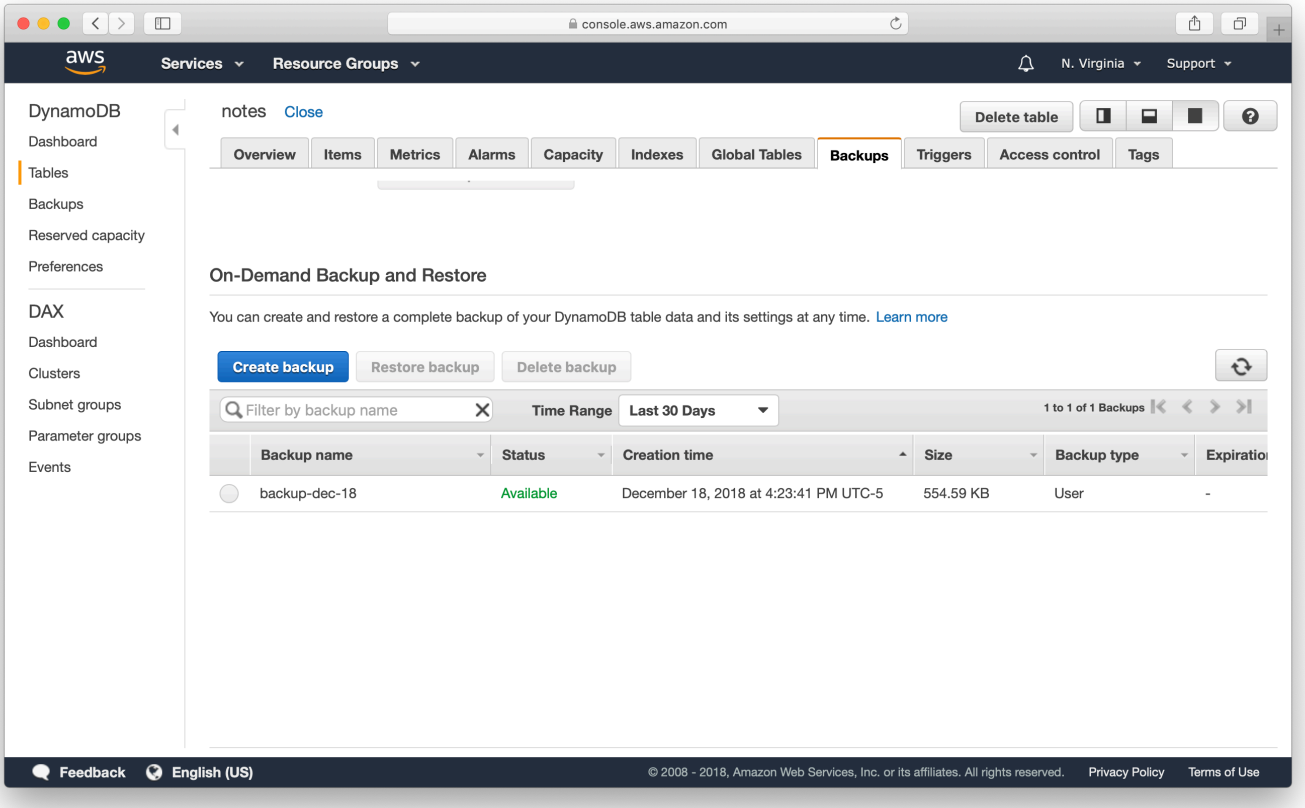
And just hit **Create backup**.



Give your backup a name and hit **Create**.

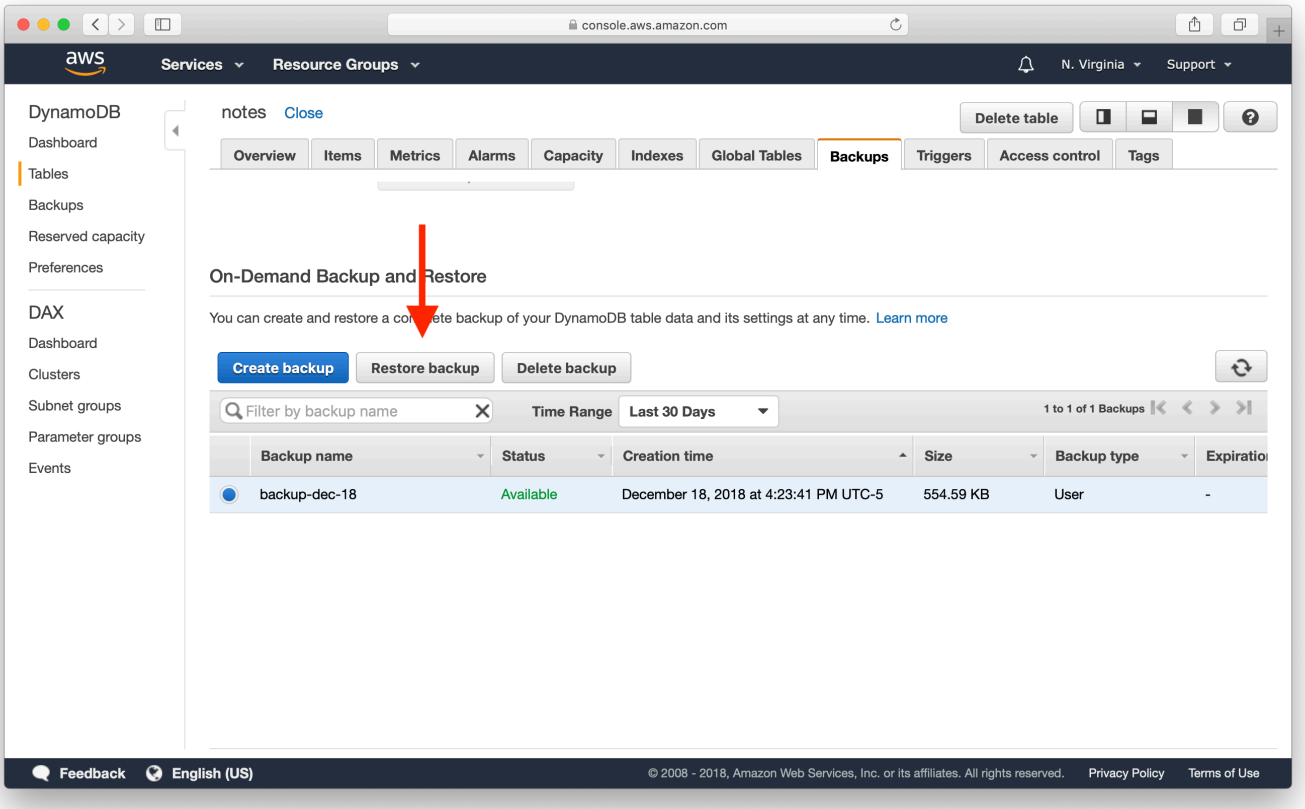


You should now be able to see your newly created backup.

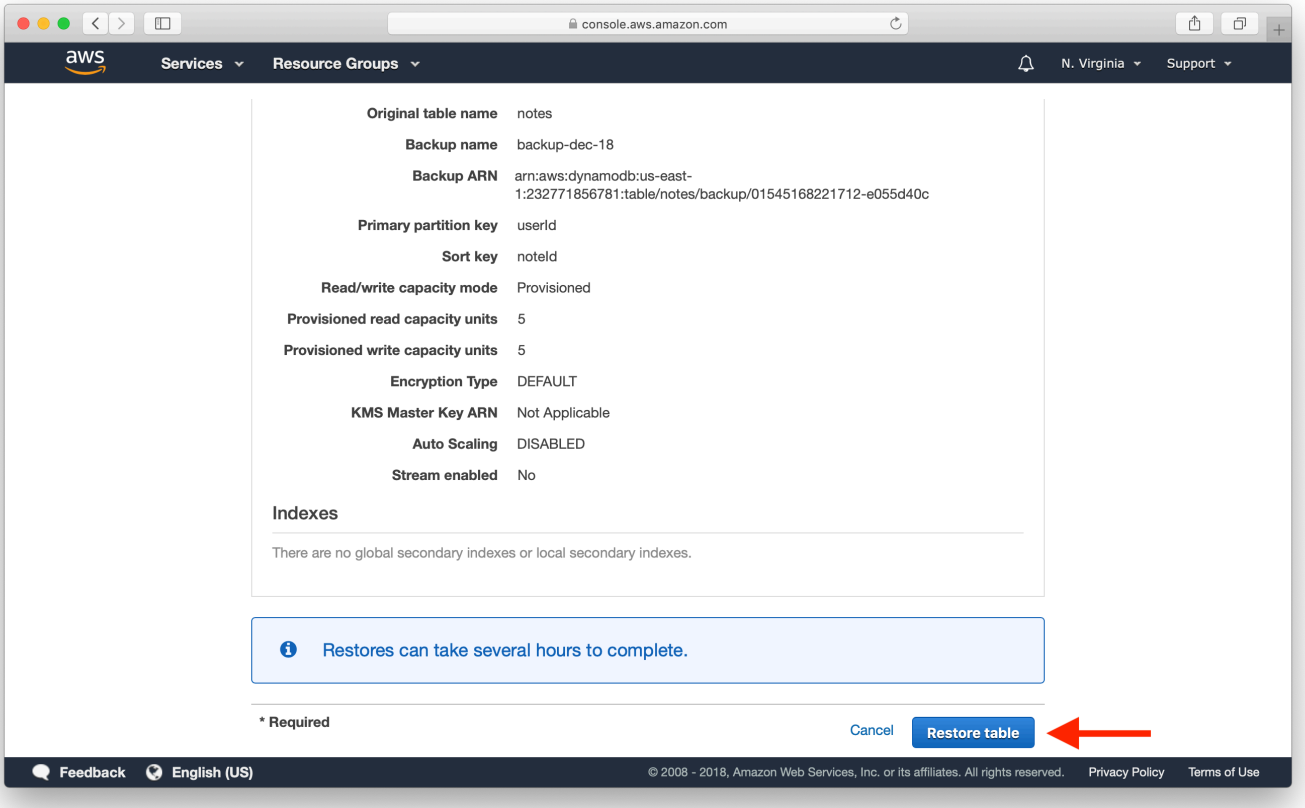


Restore Backup

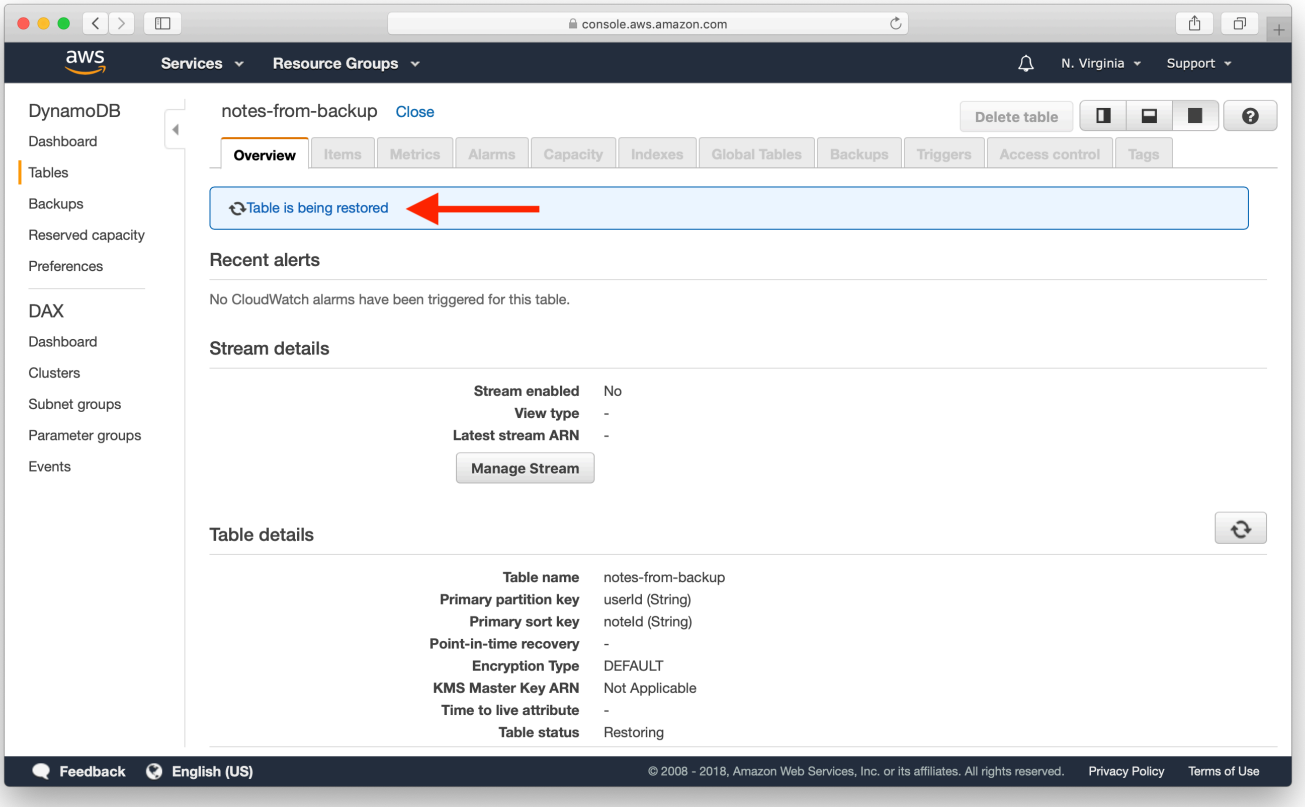
Now to restore your backup, simply select the backup and hit **Restore backup**.



Here you can type in the name of the new table you want to restore to and hit **Restore table**.



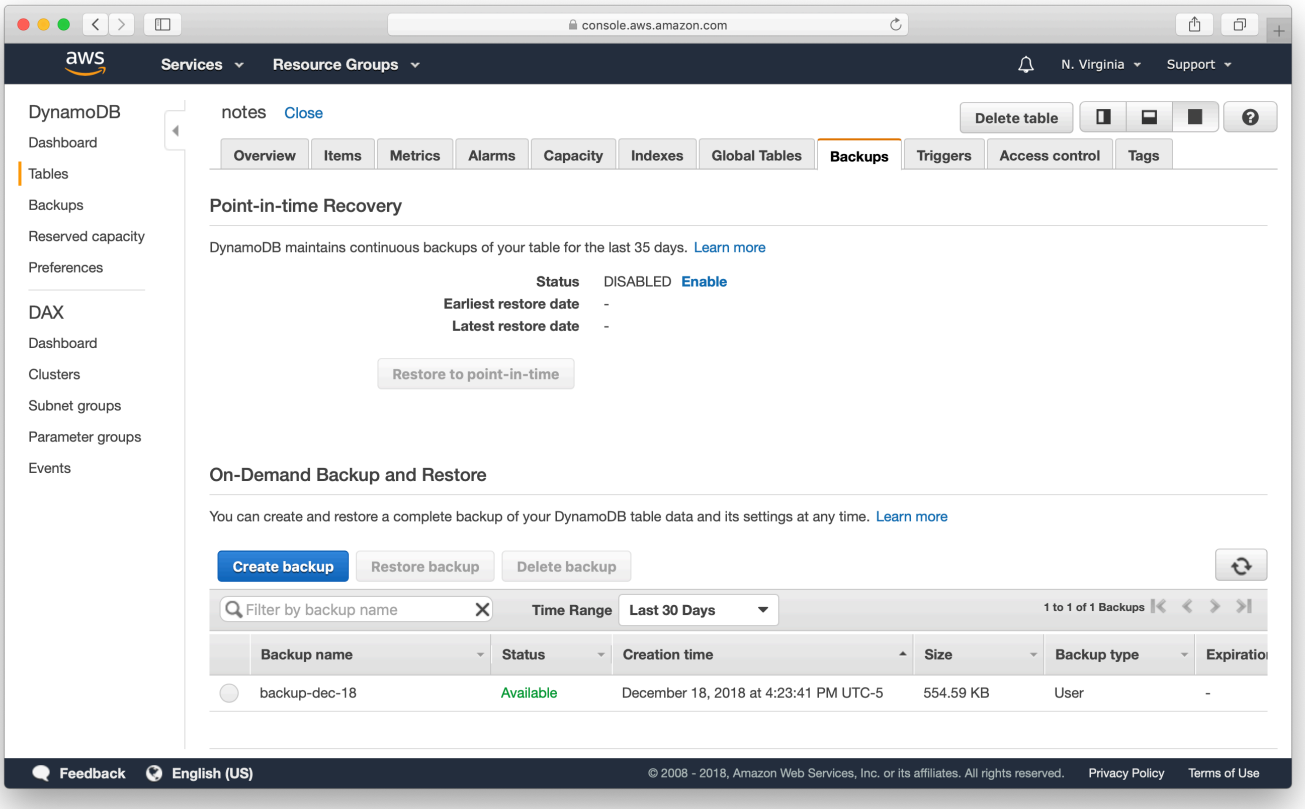
Depending on the size of the table, this might take some time. But you should notice a new table being created from the backup.



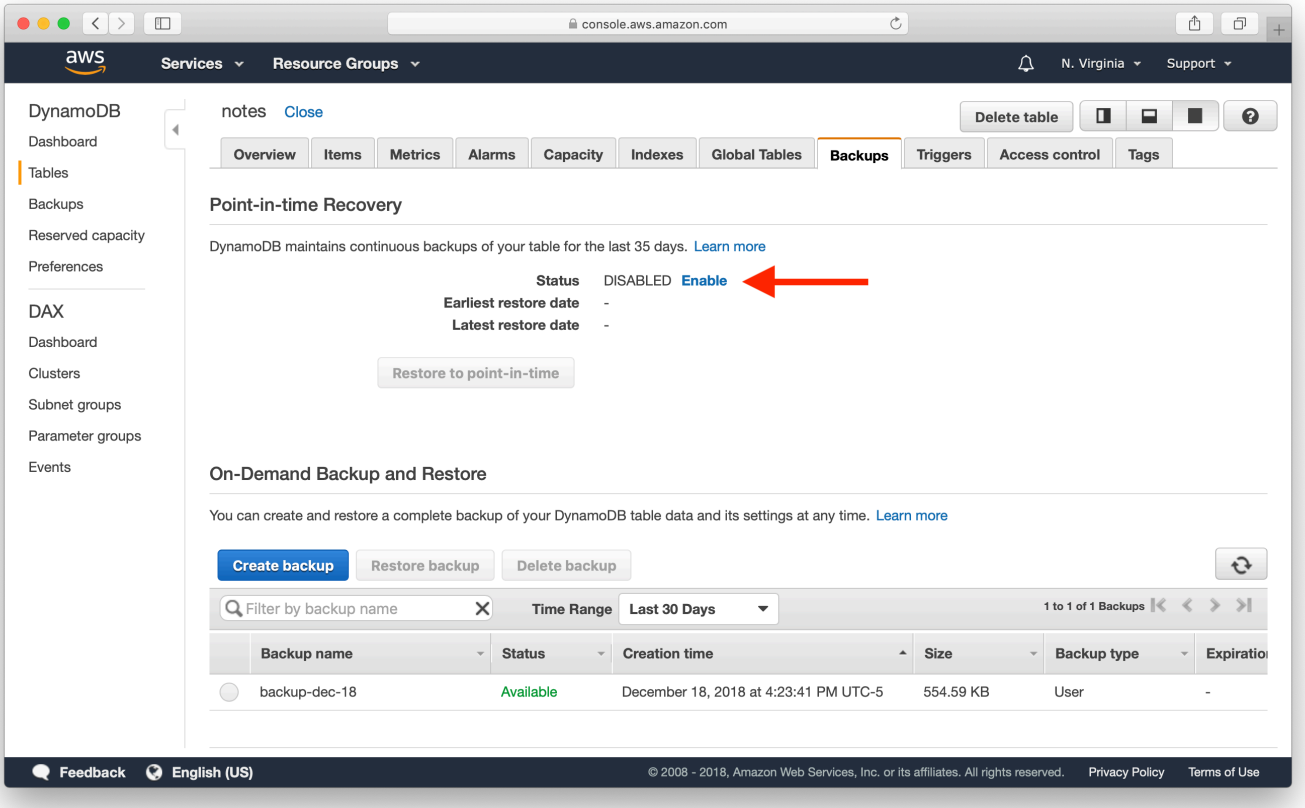
DynamoDB makes it easy to create and restore on-demand backups. You can also read more about on-demand backups here (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/BackupRestore.html).

Point-in-Time Recovery

To enable Point-in-time Recovery once again head over to the **Backups** tab.

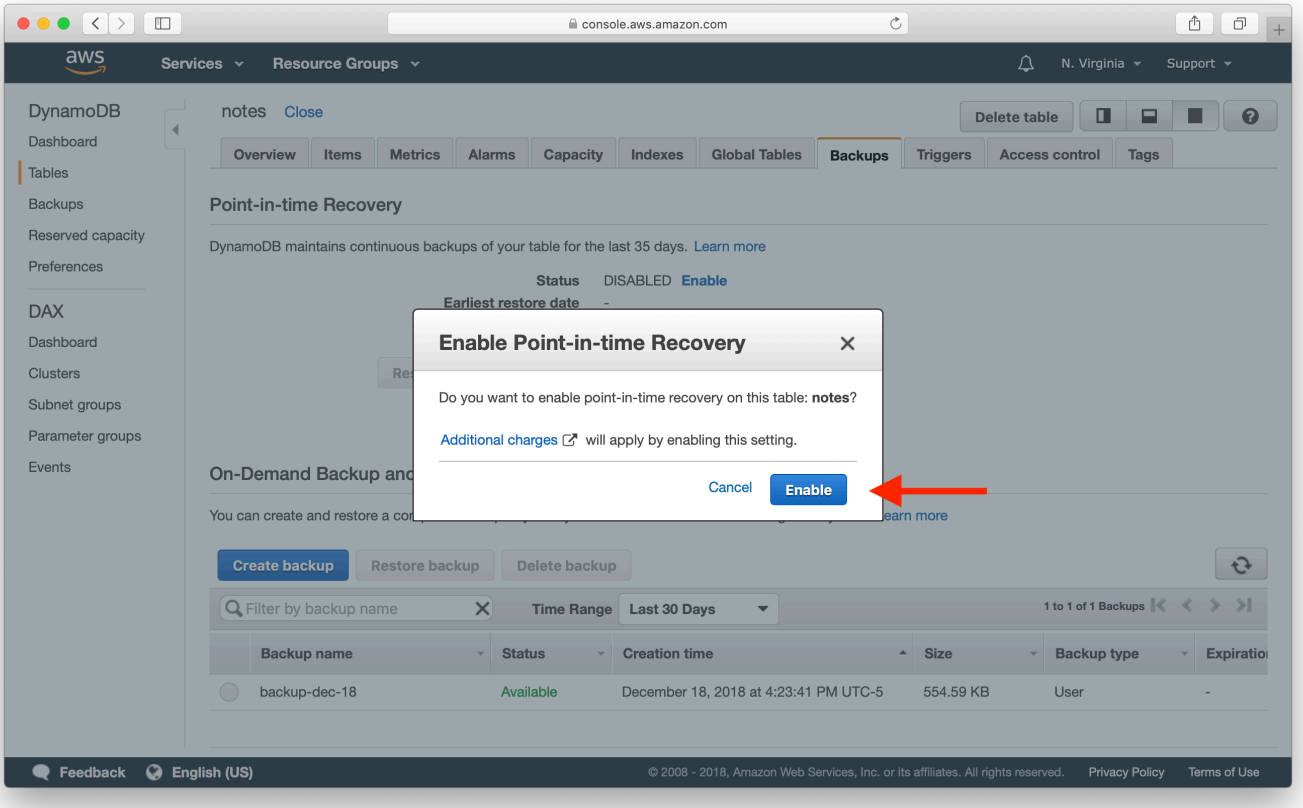


And hit **Enable** in the Point-in-time Recovery section.



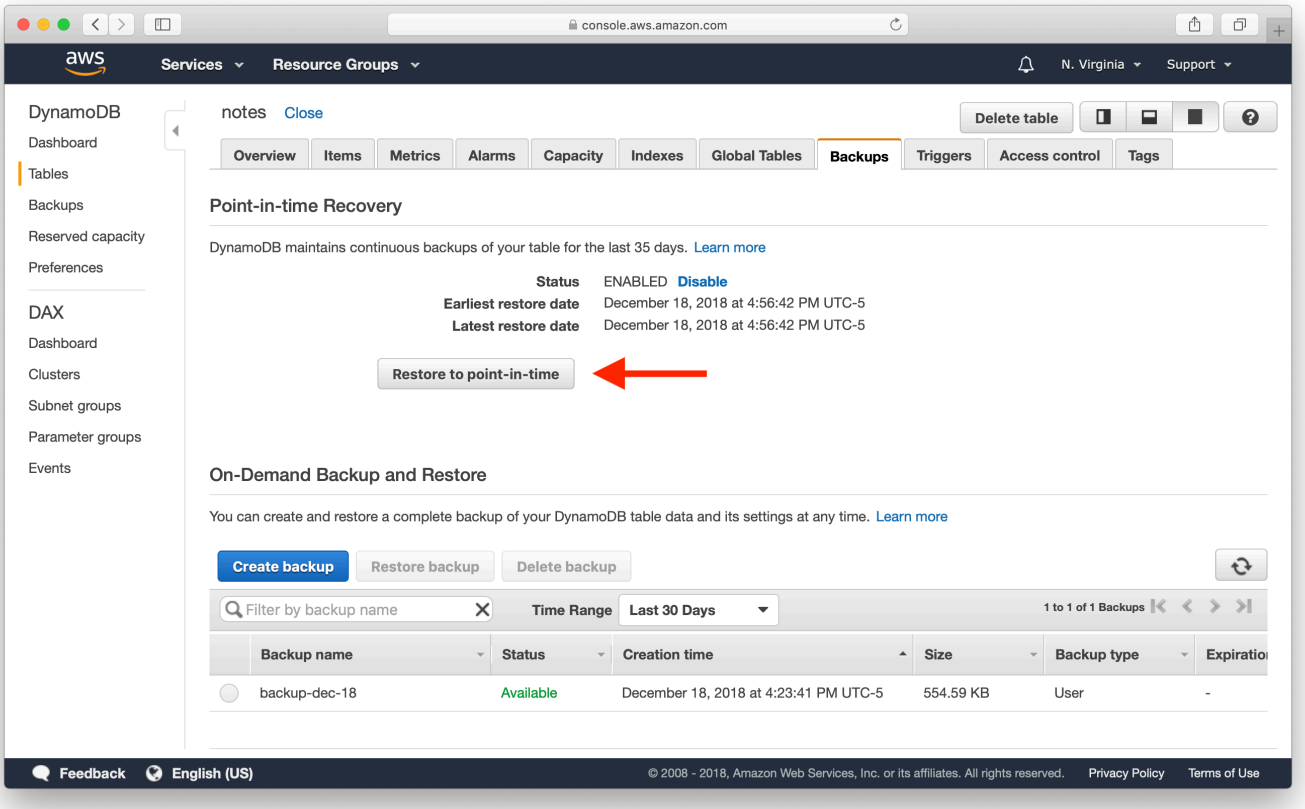
This will notify you that additional charges will apply for this setting. Click

Enable to confirm.

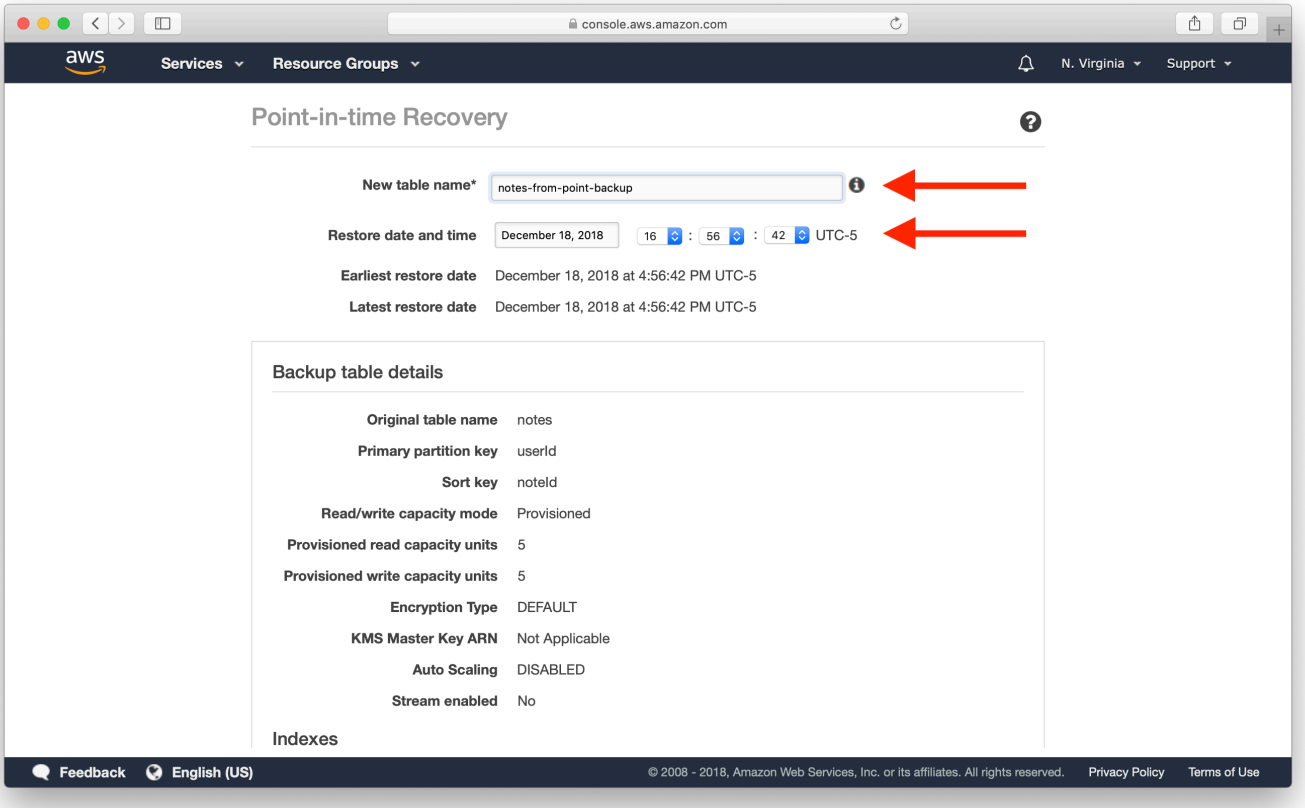


Restore to Point-in-Time

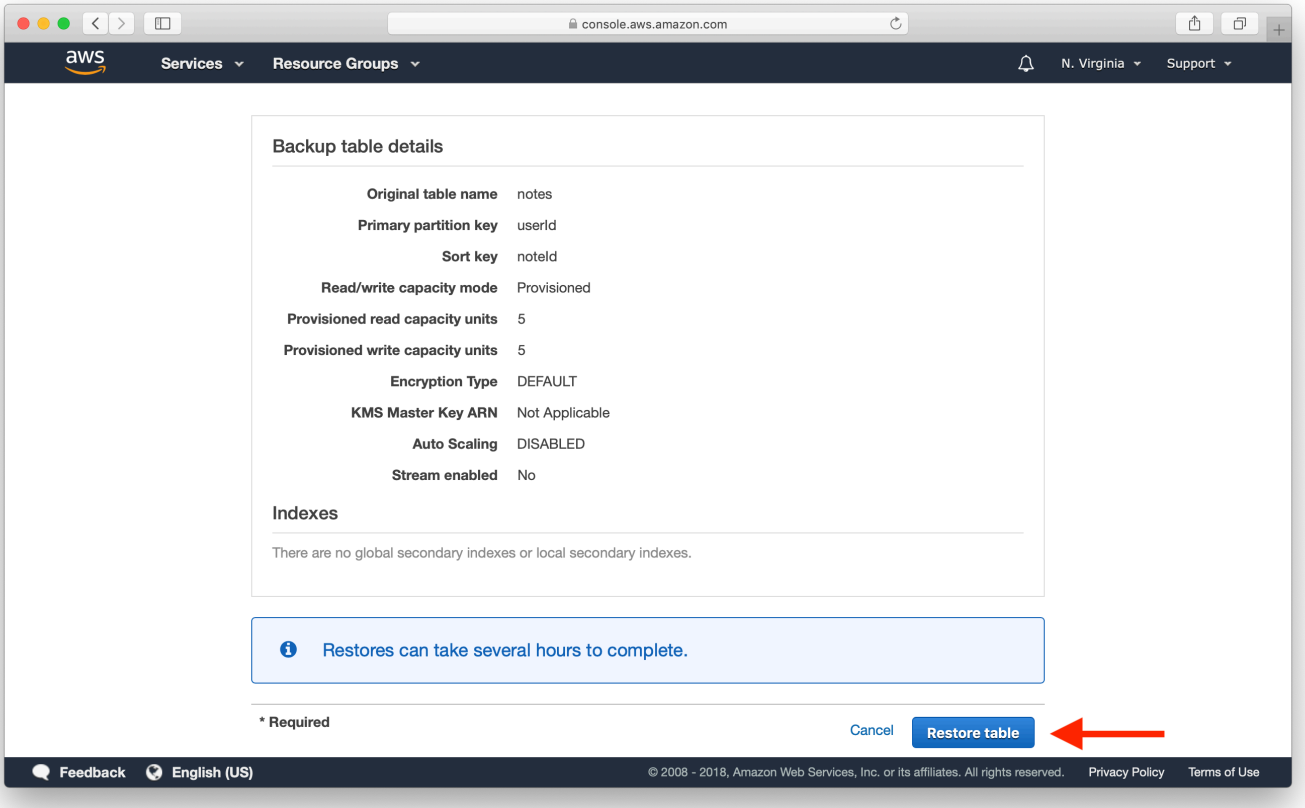
Once enabled, you can click **Restore to point-in-time** to restore to an older point.



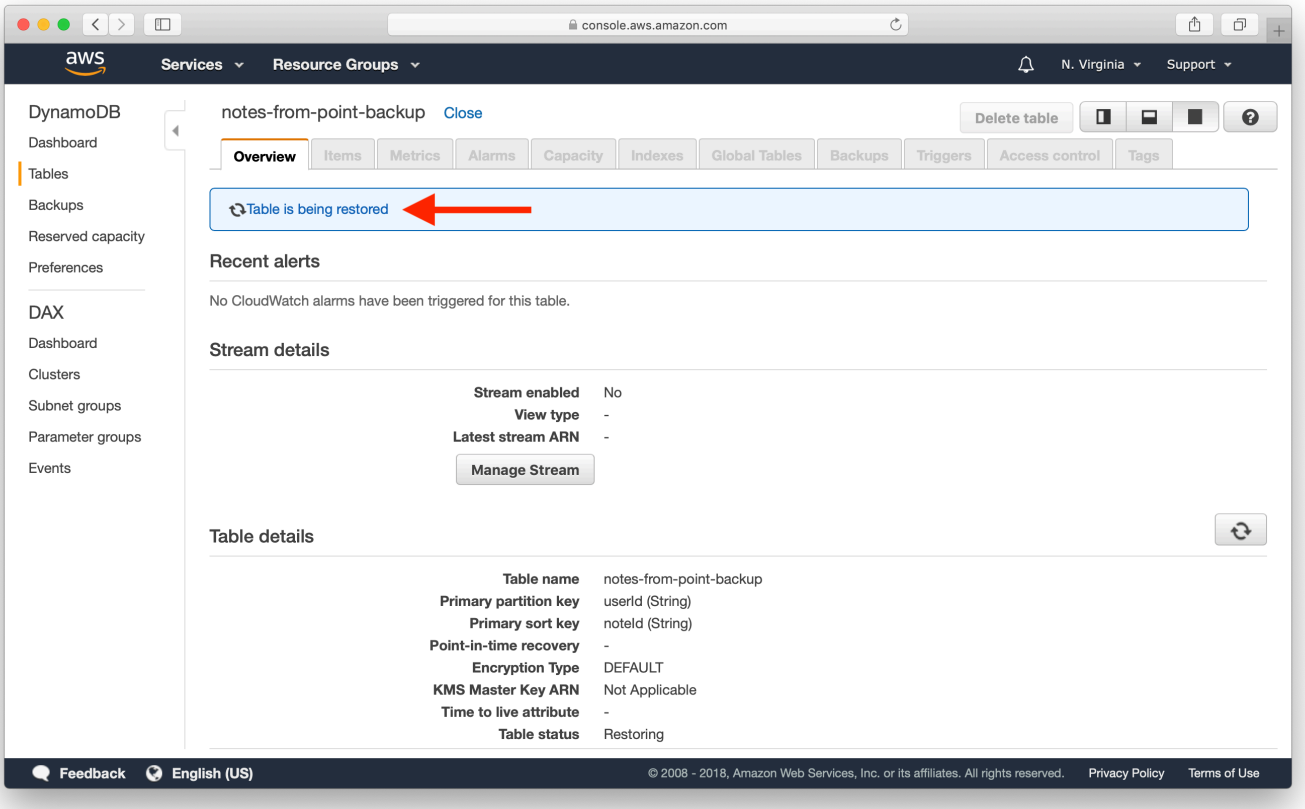
Here you can type in the name of the new table to be restored to and select the time you want to recover to.



And hit **Restore table**.



You should see your new table being restored.



You can read more about the details of Point-in-time Recovery here (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/PointInTimeRecovery.html).

Conclusion

Given, the two above types; a good strategy is to enable Point-in-time recovery and maintain a schedule of longer term On-demand backups. There are quite a few plugins and scripts that can help you with scheduling On-demand backups, here is one created by one of our readers - https://github.com/UnlyEd/serverless-plugin-dynamodb-backups.

Also worth noting, DynamoDB's backup and restore actions have no impact on the table performance or availability. No worrying about long backup processes that slow down performance for your active users.

So make sure to configure backups for the DynamoDB tables in your

applications.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/backups-in-dynamodb/705)

Configure Multiple AWS Profiles

When we configured our AWS CLI in the Configure the AWS CLI (/chapters/configure-the-aws-cli.html) chapter, we used the `aws configure` command to set the IAM credentials of the AWS account we wanted to use to deploy our serverless application to.

These credentials are stored in `~/.aws/credentials` and are used by the Serverless Framework when we run `serverless deploy`. Behind the scenes Serverless uses these credentials and the AWS SDK to create the necessary resources on your behalf to the AWS account specified in the credentials.

There are cases where you might have multiple credentials configured in your AWS CLI. This usually happens if you are working on multiple projects or if you want to separate the different stages of the same project.

In this chapter let's take a look at how you can work with multiple AWS credentials.

Create a New AWS Profile

Let's say you want to create a new AWS profile to work with. Follow the steps outlined in the Create an IAM User (/chapters/create-an-iam-user.html) chapter to create an IAM user in another AWS account and take

a note of the **Access key ID** and **Secret access key**.

To configure the new profile in your AWS CLI use:

```
$ aws configure --profile newAccount
```

Where `newAccount` is the name of the new profile you are creating. You can leave the **Default region name** and **Default output format** the way they are.

Set a Profile on Local

We mentioned how the Serverless Framework uses your AWS profile to deploy your resources on your behalf. But while developing on your local using the `serverless invoke local` command things are a little different.

In this case your Lambda function is run locally and has not been deployed yet. So any calls made in your Lambda function to any other AWS resources on your account will use the default AWS profile that you have. You can check your default AWS profile in `~/.aws/credentials` under the `[default]` tag.

To switch the default AWS profile to a new profile for the `serverless invoke local` command, you can run the following:

```
$ AWS_PROFILE=newAccount serverless invoke local --
function hello
```

Here `newAccount` is the name of the profile you want to switch to and `hello` is the name of the function that is being invoked locally. By adding

`AWS_PROFILE=newAccount` at the beginning of our `serverless invoke local` command we are setting the variable that the AWS SDK will use to figure out what your default AWS profile is.

If you want to set this so that you don't add it to each of your commands, you can use the following command:

```
$ export AWS_PROFILE=newAccount
```

Where `newAccount` is the profile you want to switch to. Now for the rest of your shell session, `newAccount` will be your default profile.

You can read more about this in the AWS Docs here (http://docs.aws.amazon.com/cli/latest/userguide/cli-multiple-profiles.html).

Set a Profile While Deploying

Now if we want to deploy using this newly created profile we can use the `--aws-profile` option for the `serverless deploy` command.

```
$ serverless deploy --aws-profile newAccount
```

Again, `newAccount` is the AWS profile Serverless Framework will be using to deploy.

If you don't want to set the profile every time you run `serverless deploy`, you can add it to your `serverless.yml`.

```
service: service-name
```

```
provider:
  name: aws
  stage: dev
  profile: newAccount
```

Note the `profile: newAccount` line here. This is telling Serverless to use the `newAccount` profile while running `serverless deploy`.

Set Profiles per Stage

There are cases where you would like to specify a different AWS profile per stage. A common scenario for this is when you have a completely separate staging environment than your production one. Each environment has its own API endpoint, database tables, and more importantly, the IAM policies to secure the environment. A simple yet effective way to achieve this is to keep the environments in separate AWS accounts. AWS Organizations (https://aws.amazon.com/organizations/) was in fact introduced to help teams to create and manage these accounts and consolidate the usage charges into a single bill.

Let's look at a quick example of how to work with multiple profiles per stage. So following the examples from before, if you wanted to deploy to your production environment, you would:

```
$ serverless deploy --stage prod --aws-profile
prodAccount
```

And to deploy to the staging environment you would:

```
$ serverless deploy --stage dev --aws-profile
devAccount
```

Here, `prodAccount` and `devAccount` are the AWS profiles for the production and staging environment respectively.

To simplify this process you can add the profiles to your `serverless.yml`. So you don't have to specify them in your `serverless deploy` commands.

```
service: service-name

custom:
  myStage: ${opt:stage, self:provider.stage}
  myProfile:
    prod: prodAccount
    dev: devAccount

provider:
  name: aws
  stage: dev
  profile:
${self:custom.myProfile.${self:custom.myStage}}
```

There are a couple of things happening here.

- We first defined `custom.myStage` as `${opt:stage, self:provider.stage}`. This is telling Serverless Framework to use the value from the `--stage` CLI option if it exists. If not, use the default stage specified in `provider.stage`.
- We also defined `custom.myProfile`, which contains the AWS profiles we want to use to deploy for each stage. Just as before we want to use the `prodAccount` profile if we are deploying to stage `prod` and the `devAccount` profile if we are deploying to stage

`dev`.

- Finally, we set the `provider.profile` to `${self:custom.myProfile.${self:custom.myStage}}`. This picks the value of our profile depending on the current stage defined in `custom.myStage`.

We used the concept of variables in Serverless Framework in this example. You can read more about this in the chapter on Serverless Environment Variables (/chapters/serverless-environment-variables.html).

Now, when you deploy to production, Serverless Framework is going to use the `prodAccount` profile. And the resources will be provisioned inside `prodAccount` profile user's AWS account.

```
$ serverless deploy --stage prod
```

And when you deploy to staging, the exact same set of AWS resources will be provisioned inside `devAccount` profile user's AWS account.

```
$ serverless deploy --stage dev
```

Notice that we did not have to set the `--aws-profile` option. And that's it, this should give you a good understanding of how to work with multiple AWS profiles and credentials.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/configure-multiple-aws-

```
profiles/21)
```

Customize the Serverless IAM Policy

Serverless Framework deploys using the policy attached to the IAM credentials in your AWS CLI profile. Back in the Create an IAM User (/chapters/create-an-iam-user.html) chapter we created a user that the Serverless Framework will use to deploy our project. This user was assigned **AdministratorAccess**. This means that Serverless Framework and your project has complete access to your AWS account. This is fine in trusted environments but if you are working as a part of a team you might want to fine-tune the level of access based on who is using your project.

In this chapter we will take a look at how to customize the IAM Policy that Serverless Framework is going to use.

The permissions required can be categorized into the following areas:

- Permissions required by Serverless Framework
- Permissions required by your Serverless Framework plugins
- Permissions required by your Lambda code

Granting **AdministratorAccess** policy ensures that your project will always have the necessary permissions. But if you want to create an IAM policy that grants the minimal set of permissions, you need to customize your IAM policy.

A basic Serverless project needs permissions to the following AWS

services:

- **CloudFormation** to create change set and update stack
- **S3** to upload and store Serverless artifacts and Lambda source code
- **CloudWatch Logs** to store Lambda execution logs
- **IAM** to manage policies for the Lambda IAM Role
- **API Gateway** to manage API endpoints
- **Lambda** to manage Lambda functions
- **EC2** to execute Lambda in VPC
- **CloudWatch Events** to manage CloudWatch event triggers

A simple IAM Policy template

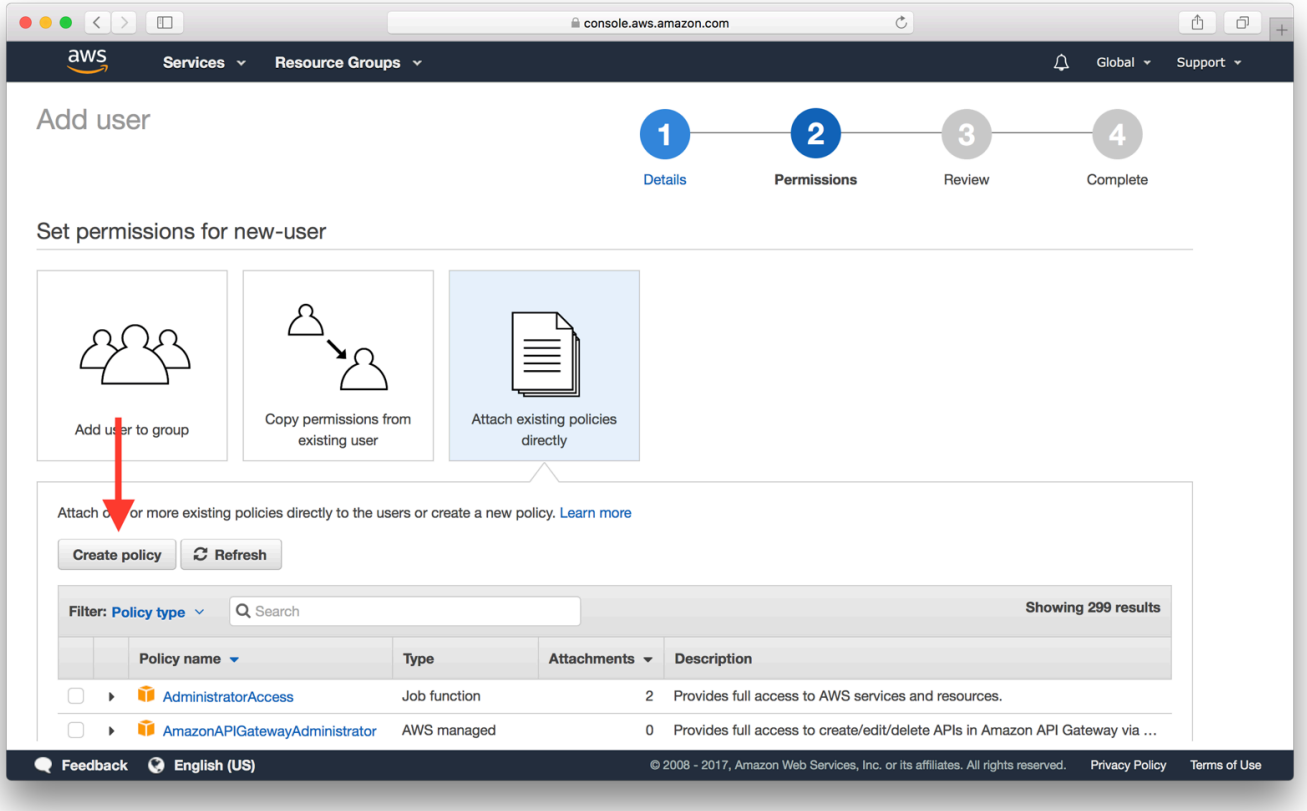
These can be defined and granted using a simple IAM policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "cloudformation:*",
        "s3:*",
        "logs:*",
        "iam:*",
        "apigateway:*",
        "lambda:*",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "events:*"
```

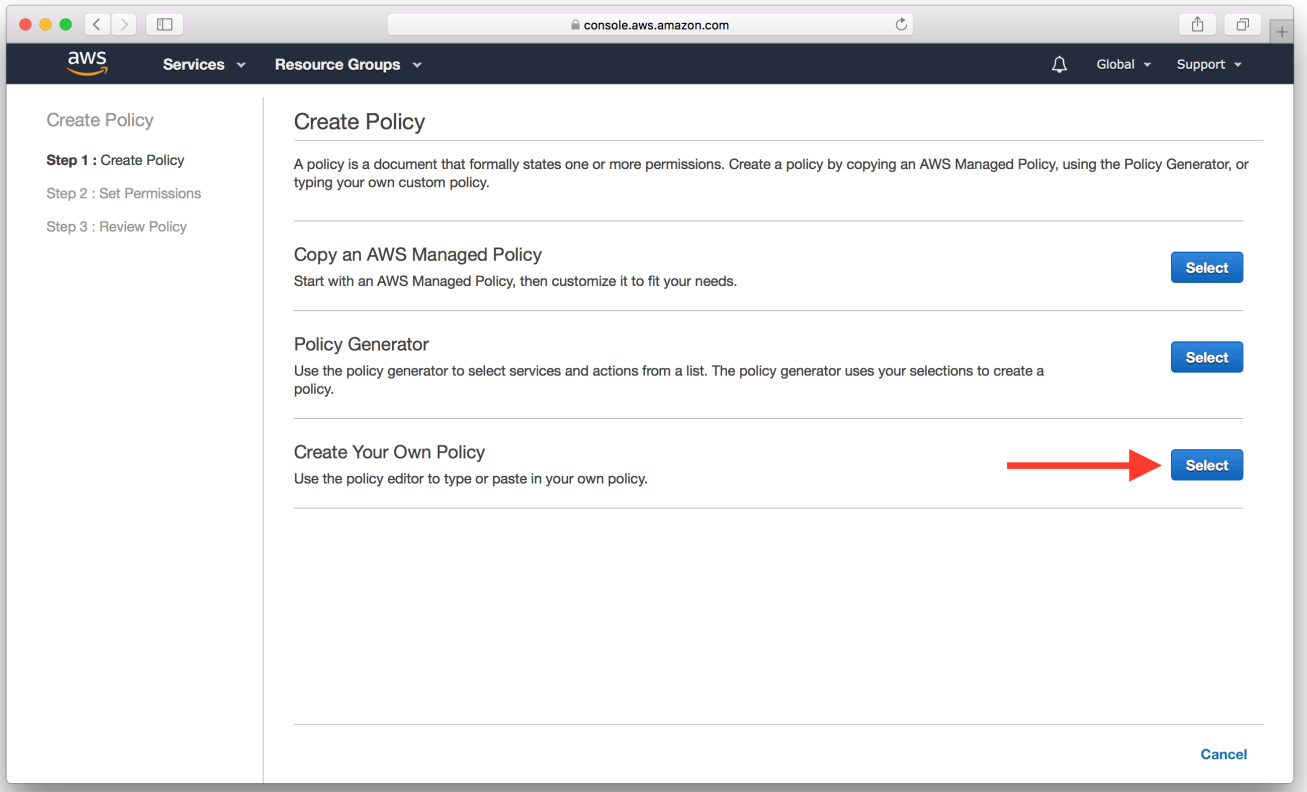
```
        ],
        "Resource": [
          "*"
        ]
      }
    ]
}
```

We can attach this policy to the IAM user we are creating by continuing from the **Attach existing policies directly** step in the Create an IAM User (/chapters/create-an-iam-user.html) chapter.

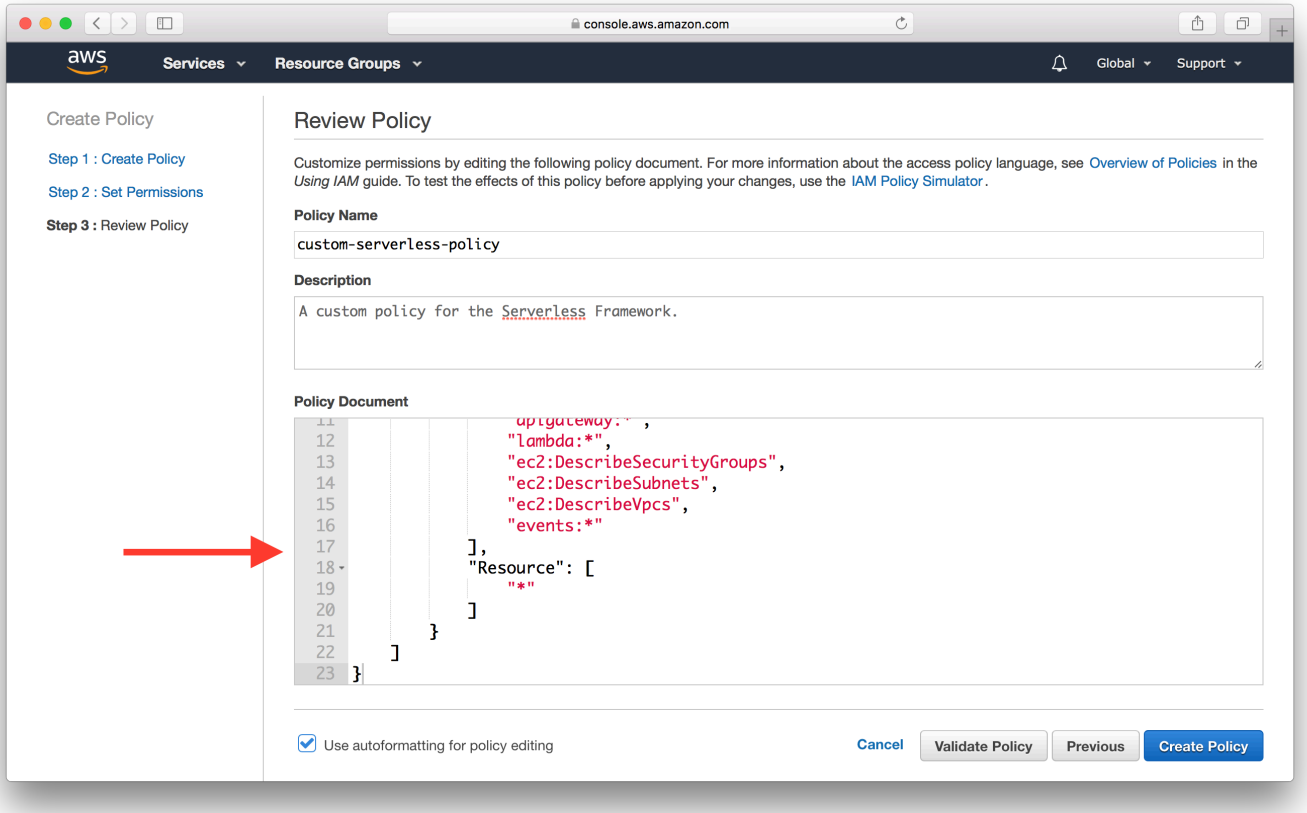
Hit the **Create policy** button.



And hit **Select** in the **Create Your Own Policy** section.



Here pick a name for your new policy and paste the policy created above in the **Policy Document** field.



Finally, hit **Create Policy**. You can now chose this policy while creating your IAM user instead of the **AdministratorAccess** one that we had used before.

This policy grants your Serverless Framework project access to all the resources listed above. But we can narrow this down further by restricting them to specific **Actions** for the specific **Resources** in each AWS service.

An advanced IAM Policy template

Below is a more nuanced policy template that restricts access to the Serverless project that is being deployed. Make sure to replace `<region>`, `<account_no>` and `<service_name>` for your specific project.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "cloudformation:Describe*",
        "cloudformation:List*",
        "cloudformation:Get*",
        "cloudformation:CreateStack",
        "cloudformation:UpdateStack",
        "cloudformation:DeleteStack"
      ],
      "Resource": "arn:aws:cloudformation:<region>:
<account_no>:stack/<service_name>*/*"
    },
    {
```

```
    "Effect": "Allow",
    "Action": [
      "cloudformation:ValidateTemplate"
    ],
    "Resource": "*"
  },
  {
    "Effect": "Allow",
    "Action": [
      "s3:CreateBucket",
      "s3:DeleteBucket",
      "s3:Get*",
      "s3:List*"
    ],
    "Resource": [
      "arn:aws:s3:::*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "s3:*"
    ],
    "Resource": [
      "arn:aws:s3:::*/*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "logs:DescribeLogGroups"
```

```
      ],
      "Resource": "arn:aws:logs:<region>:
<account_no>:log-group::log-stream:*"
    },
    {
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:DeleteLogGroup",
        "logs:DeleteLogStream",
        "logs:DescribeLogStreams",
        "logs:FilterLogEvents"
      ],
      "Resource": "arn:aws:logs:<region>:
<account_no>:log-group:/aws/lambda/<service_name>*:log-
stream:*",
      "Effect": "Allow"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:GetRole",
        "iam:PassRole",
        "iam:CreateRole",
        "iam:DeleteRole",
        "iam:DetachRolePolicy",
        "iam:PutRolePolicy",
        "iam:AttachRolePolicy",
        "iam:DeleteRolePolicy"
      ],
      "Resource": [
```

```
        "arn:aws:iam::
<account_no>:role/<service_name>*-lambdaRole"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:GET",
        "apigateway:POST",
        "apigateway:PUT",
        "apigateway:DELETE"
      ],
      "Resource": [
        "arn:aws:apigateway:<region>::/restapis"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:GET",
        "apigateway:POST",
        "apigateway:PUT",
        "apigateway:DELETE"
      ],
      "Resource": [
        "arn:aws:apigateway:<region>::/restapis/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
```

```
        "lambda:GetFunction",
        "lambda:CreateFunction",
        "lambda:DeleteFunction",
        "lambda:UpdateFunctionConfiguration",
        "lambda:UpdateFunctionCode",
        "lambda:ListVersionsByFunction",
        "lambda:PublishVersion",
        "lambda:CreateAlias",
        "lambda:DeleteAlias",
        "lambda:UpdateAlias",
        "lambda:GetFunctionConfiguration",
        "lambda:AddPermission",
        "lambda:RemovePermission",
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:*:<account_no>:function:
<service_name>*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs"
      ],
      "Resource": [
        "*"
      ]
    },
```

```
    {
      "Effect": "Allow",
      "Action": [
        "events:Put*",
        "events:Remove*",
        "events:Delete*",
        "events:Describe*"
      ],
      "Resource": "arn:aws:events::
<account_no>:rule/<service_name>*"
    }
  ]
}
```

The `<account_no>` is your AWS Account ID and you can follow these instructions (http://docs.aws.amazon.com/IAM/latest/UserGuide/console_account-alias.html) to look it up.

Also, recall that the `<region>` and `<service_name>` are defined in your `serverless.yml` like so.

```
service: my-service

provider:
  name: aws
  region: us-east-1
```

In the above `serverless.yml`, the `<region>` is `us-east-1` and the `<service_name>` is `my-service`.

The above IAM policy template restricts access to the AWS services based on the name of your Serverless project and the region it is deployed in.

It provides sufficient permissions for a minimal Serverless project. However, if you provision any additional resources in your **serverless.yml**, or install Serverless plugins, or invoke any AWS APIs in your application code; you would need to update the IAM policy to accommodate for those changes. If you are looking for details on where this policy comes from; here is an in-depth discussion on the minimal Serverless IAM Deployment Policy (https://github.com/serverless/serverless/issues/1439) required for a Serverless project.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/customize-the-serverless-iam-policy/18)

Mapping Cognito Identity Id and User Pool Id

If you are using the Cognito User Pool to manage your users while using the Identity Pool to secure your AWS resources; you might run into an interesting issue. How do you find the user's User Pool User Id in your Lambda function?

Identity Pool User Id vs User Pool User Id

You might recall (from the chapters where we work with our Lambda functions (/chapters/add-a-create-note-api.html)), that we used the `event.requestContext.identity.cognitoIdentityId` as the user Id. This is the Id that a user is assigned through the Identity Pool. However, you cannot use this Id to look up information for this user from the User Pool. This is because to access your Lambda function, your user needs to:

1. Authenticate through your User Pool
2. And then federate their identity through the Identity Pool

At this second step, their User Pool information is no longer available to us. To better understand this flow you can take a look at the Cognito user pool vs identity pool (/chapters/cognito-user-pool-vs-identity-pool.html) chapter. But in a nutshell, you can have multiple authentication providers at step 1 and the Identity Pool just ensures that they are all given a *global* user id that you can use.

Finding the User Pool User Id

However, you might find yourself looking for a user's User Pool user id in your Lambda function. While the process below isn't documented, it is something we have been using and it solves this problem pretty well.

Below is a sample Lambda function where we find the user's User Pool user id.

```
export async function main(event, context, callback) {
  const authProvider =
event.requestContext.identity.cognitoAuthenticationProvid

  // Cognito authentication provider looks like:
  // cognito-idp.us-east-1.amazonaws.com/us-east-
1_xxxxxxxxx,cognito-idp.us-east-1.amazonaws.com/us-
east-1_aaaaaaaaa:CognitoSignIn:qqqqqqqq-1111-2222-3333-
rrrrrrrrrrr
  // Where us-east-1_aaaaaaaaa is the User Pool id
  // And qqqqqqqq-1111-2222-3333-rrrrrrrrrrr is the
User Pool User Id
  const parts = authProvider.split(':');
  const userPoolIdParts = parts[parts.length -
3].split('/');

  const userPoolId =
userPoolIdParts[userPoolIdParts.length - 1];
  const userPoolUserId = parts[parts.length - 1];

  ...
}
```

The `event.requestContext.identity.cognitoAuthenticationProvider` gives us a string that contains the authentication details from the User Pool. Note that this info will be different depending on the authentication provider you are using. This string has the following format:

```
cognito-idp.us-east-1.amazonaws.com/us-east-1_xxxxxxxxx,cognito-idp.us-east-1.amazonaws.com/us-east-1_aaaaaaaaa:CognitoSignIn:qqqqqqqq-1111-2222-3333-rrrrrrrrrrr
```

Where `us-east-1_aaaaaaaaa` is the User Pool id and `qqqqqqqq-1111-2222-3333-rrrrrrrrrrr` is the User Pool User Id. We can extract these out with some simple JavaScript as we detailed above.

And that's it! You now have access to a user's User Pool user Id even though we are using AWS IAM and Federated Identities to secure our Lambda function.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/mapping-cognito-identity-id-and-user-pool-id/500)

Connect to API Gateway with IAM Auth

Connecting to an API Gateway endpoint secured using AWS IAM can be challenging. You need to sign your requests using Signature Version 4 (http://docs.aws.amazon.com/general/latest/gr/signature-version-4.html). You can use:

- Generated API Gateway SDK (https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-generate-sdk.html)
- AWS Amplify (https://github.com/aws/aws-amplify)

The generated SDK can be hard to use since you need to re-generate it every time a change is made. And we cover how to configure your app using AWS Amplify in the Configure AWS Amplify (/chapters/configure-aws-amplify.html) chapter.

However if you are looking to simply connect to API Gateway using the AWS JS SDK, we've create a standalone `sigV4Client.js` (https://github.com/AnomalyInnovations/sigV4Client) that you can use. It is based on the client that comes pre-packaged with the generated SDK.

In this chapter we'll go over how to use the the `sigV4Client.js`. The basic flow looks like this:

1. Authenticate a user with Cognito User Pool and acquire a user token.

2. With the user token get temporary IAM credentials from the Identity Pool.
3. Use the IAM credentials to sign our API request with Signature Version 4 (http://docs.aws.amazon.com/general/latest/gr/signature-version-4.html).

Authenticate a User with Cognito User Pool

The following method can authenticate a user to Cognito User Pool.

```
function login(username, password) {
  const userPool = new CognitoUserPool({
    UserPoolId: USER_POOL_ID,
    ClientId: APP_CLIENT_ID
  });
  const user = new CognitoUser({ Username: username,
Pool: userPool });
  const authenticationData = { Username: username,
Password: password };
  const authenticationDetails = new
AuthenticationDetails(authenticationData);

  return new Promise((resolve, reject) =>
    user.authenticateUser(authenticationDetails, {
      onSuccess: result => resolve(),
      onFailure: err => reject(err)
    })
  );
}
```

Ensure to use your `USER_POOL_ID` and `APP_CLIENT_ID`. And given

their Cognito `username` and `password` you can log a user in by calling:

```
await login('my_username', 'my_password');
```

Generate Temporary IAM Credentials

Once your user is authenticated you can generate a set of temporary credentials. To do so you need to first get their JWT user token using the following:

```
function getUserToken(currentUser) {
  return new Promise((resolve, reject) => {
    currentUser.getSession(function(err, session) {
      if (err) {
        reject(err);
        return;
      }
      resolve(session.getIdToken().getJwtToken());
    });
  });
}
```

Where you can get the current logged in user using:

```
function getCurrentUser() {
  const userPool = new CognitoUserPool({
    UserPoolId: config.cognito.USER_POOL_ID,
    ClientId: config.cognito.APP_CLIENT_ID
  });
  return userPool.getCurrentUser();
}
```

And with the JWT token you can generate their temporary IAM credentials using:

```
function getAwsCredentials(userToken) {
  const authenticator = `cognito-idp.${config.cognito

.REGION}.amazonaws.com/${config.cognito.USER_POOL_ID}`;

  AWS.config.update({ region: config.cognito.REGION });

  AWS.config.credentials = new
AWS.CognitoIdentityCredentials({
    IdentityPoolId: config.cognito.IDENTITY_POOL_ID,
    Logins: {
      [authenticator]: userToken
    }
  });

  return AWS.config.credentials.getPromise();
}
```

Sign API Gateway Requests with Signature Version 4

The `sigV4Client.js` needs **crypto-js** (https://github.com/brix/crypto-js) installed.

Install it by running the following in your project root.

```
$ npm install crypto-js --save
```

And to use the `sigV4Client.js` simply copy it over to your project.

→ **`sigV4Client.js`**
(https://raw.githubusercontent.com/AnomalyInnovations/sigV4Client/master/sigV4Client.js)

This file can look a bit intimidating at first but it is just using the temporary credentials and the request parameters to create the necessary signed headers. To create a new `sigV4Client` we need to pass in the following:

```
// Pseudocode

sigV4Client.newClient({
  // Your AWS temporary access key
  accessKey,
  // Your AWS temporary secret key
  secretKey,
  // Your AWS temporary session token
  sessionToken,
  // API Gateway region
  region,
  // API Gateway URL
  endpoint
});
```

And to sign a request you need to use the `signRequest` method and pass in:

```
// Pseudocode

const signedRequest = client.signRequest({
```

```
  // The HTTP method
  method,
  // The request path
  path,
  // The request headers
  headers,
  // The request query parameters
  queryParams,
  // The request body
  body
});
```

And `signedRequest.headers` should give you the signed headers that you need to make the request.

Call API Gateway with the sigV4Client

Let's put it all together. The following gives you a simple helper function to call an API Gateway endpoint.

```
function invokeApig({
  path,
  method = "GET",
  headers = {},
  queryParams = {},
  body
}) {

  const currentUser = getCurrentUser();

  const userToken = await getUserToken(currentUser);
```

```
  await getAwsCredentials(userToken);

  const signedRequest = sigV4Client
    .newClient({
      accessKey: AWS.config.credentials.accessKeyId,
      secretKey:
AWS.config.credentials.secretAccessKey,
      sessionToken:
AWS.config.credentials.sessionToken,
      region: YOUR_API_GATEWAY_REGION,
      endpoint: YOUR_API_GATEWAY_URL
    })
    .signRequest({
      method,
      path,
      headers,
      queryParams,
      body
    });

  body = body ? JSON.stringify(body) : body;
  headers = signedRequest.headers;

  const results = await fetch(signedRequest.url, {
    method,
    headers,
    body
  });

  if (results.status !== 200) {
```

```
    throw new Error(await results.text());
  }

  return results.json();
}
```

Make sure to replace `YOUR_API_GATEWAY_URL` and `YOUR_API_GATEWAY_REGION`. Post in the comments if you have any questions.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/comments-signup-with-aws-cognito/113)

Serverless Node.js Starter

Based on what we have gone through in this guide, it makes sense that we have a good starting point for our future projects. For this we created a couple of Serverless starter projects that you can use called, Serverless Node.js Starter (https://github.com/AnomalyInnovations/serverless-nodejs-starter). We also have a Python version called Serverless Python Starter (https://github.com/AnomalyInnovations/serverless-python-starter). Our starter projects also work really well with Seed (https://seed.run); a fully-configured CI/CD pipeline for Serverless Framework.

Serverless Node.js Starter (https://github.com/AnomalyInnovations/serverless-nodejs-starter) uses the serverless-bundle (https://github.com/AnomalyInnovations/serverless-bundle) plugin (an extension of the serverless-webpack (https://github.com/serverless-heaven/serverless-webpack) plugin) and the serverless-offline (https://github.com/dherault/serverless-offline) plugin. It supports:

- **Generating optimized Lambda packages with Webpack**
- **Use ES7 syntax in your handler functions**
- **Run API Gateway locally**
  - Use `serverless offline start`
- **Support for unit tests**
  - Run `npm test` to run your tests
- **Sourcemaps for proper error messages**

    - Error message show the correct line numbers
    - Works in production with CloudWatch
- **Add environment variables for your stages**
- **No need to manage Webpack or Babel configs**

Demo

A demo version of this service is hosted on AWS - `https://z6pv80ao4l.execute-api.us-east-1.amazonaws.com/dev/hello` (https://z6pv80ao4l.execute-api.us-east-1.amazonaws.com/dev/hello).

And here is the ES7 source behind it.

```
export const hello = async (event, context, callback)
=> {
  const response = {
    statusCode: 200,
    body: JSON.stringify({
      message: `Go Serverless v1.0! ${(await message({
time: 1, copy: 'Your function executed
successfully!'}))}`,
      input: event,
    }),
  };

  callback(null, response);
};

const message = ({ time, ...rest }) => new
Promise((resolve, reject) =>
```

```
  setTimeout(() => {
    resolve(`${rest.copy} (with a delay)`);
  }, time * 1000)
);
```

Requirements

- Configure your AWS CLI (/chapters/configure-the-aws-cli.html)
- Install the Serverless Framework `npm install serverless -g`

Installation

To create a new Serverless project.

```
$ serverless install --url
https://github.com/AnomalyInnovations/serverless-
nodejs-starter --name my-project
```

Enter the new directory.

```
$ cd my-project
```

Install the Node.js packages.

```
$ npm install
```

Usage

To run a function on your local

```
$ serverless invoke local --function hello
```

To simulate API Gateway locally using serverless-offline (https://github.com/dherault/serverless-offline)

```
$ serverless offline start
```

Run your tests

```
$ npm test
```

We use Jest to run our tests. You can read more about setting up your tests here (https://facebook.github.io/jest/docs/en/getting-started.html#content).

Deploy your project

```
$ serverless deploy
```

Deploy a single function

```
$ serverless deploy function --function hello
```

To add environment variables to your project

1. Rename `env.example` to `env.yml`.
2. Add environment variables for the various stages to `env.yml`.
3. Uncomment `environment: ${file(env.yml):${self:provider.stage}}` in the `serverless.yml`.
4. Make sure to not commit your `env.yml`.

So give it a try and send us an email (mailto:contact@anoma.ly) if you have any questions or open a new issue (https://github.com/AnomalyInnovations/serverless-nodejs-starter/issues/new) if you've found a bug.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/serverless-node-js-starter/22)

Package Lambdas with serverless-bundle

AWS Lambda functions are stored as zip files in an S3 bucket. They are loaded up onto a container when the function is invoked. The time it takes to do this is called the cold start time. If a function has been recently invoked, the container is kept around. In this case, your functions get invoked a lot quicker and this delay is referred to as the warm start time. One of the factors that affects cold starts, is the size of your Lambda function package. The larger the package, the longer it takes to invoke your Lambda function.

Optimizing Lambda Packages

Serverless Framework (https://github.com/serverless/serverless) handles all the packaging and deployments for our Lambda functions. By default, it will create one package per service and use that for all the Lambda functions in that service. This means that each Lambda function in your service loads the code that is used by all the other functions as well! Fortunately there is an option to override this.

```
# Create an individual package for our functions
package:
  individually: true
```

By adding the above to your `serverless.yml`, you are telling Serverless

Framework to generate individual packages for each of your Lambda functions. Note that, this isn't the default behavior because individual packaging takes a lot longer. However, the performance benefit makes this well worth it.

While individual packaging is a good start, for Node.js apps, Serverless Framework will add your `node_modules/` directory in the package. This can balloon the size of your Lambda function packages astronomically. To fix this you can optimize your packages further by using the serverless-webpack (https://github.com/serverless-heaven/serverless-webpack) plugin to apply Webpack's tree shaking algorithm (https://webpack.js.org/guides/tree-shaking/) to only include the relevant bits of code needed for your Lambda function. Also, with the serverless-webpack plugin, you can use Babel (https://babeljs.io) to transpile your JavaScript functions so that you can use a more modern syntax including `import/export` statements.

However, using Webpack and Babel require you to manage their respective configs, plugins, and NPM packages in your Serverless app. Additionally, you might want to lint your code before your functions get packaged. This means that your projects can end up with a long list of packages and config files before you even write your first line of code! And they need to be updated over time. Which can be really hard to do across multiple projects.

```
"eslint"
"webpack"
"@babel/core"
"babel-eslint"
"babel-loader"
"eslint-loader"
```

```
"@babel/runtime"
"@babel/preset-env"
"serverless-webpack"
"source-map-support"
"webpack-node-externals"
"eslint-config-strongloop"
"@babel/plugin-transform-runtime"
"babel-plugin-source-map-support"
```

We created a new plugin to solve all of these issues.

Optimized Lambda packages without any configuration

Enter serverless-bundle (https://github.com/AnomalyInnovations/serverless-bundle); a plugin that will generate an optimized Lambda function package using serverless-webpack without you having to manage any Webpack, Babel, or ESLint configs!

```
-    "eslint"
-    "webpack"
-    "@babel/core"
-    "babel-eslint"
-    "babel-loader"
-    "eslint-loader"
-    "@babel/runtime"
-    "@babel/preset-env"
-    "serverless-webpack"
-    "source-map-support"
```

```
-    "webpack-node-externals"
-    "eslint-config-strongloop"
-    "@babel/plugin-transform-runtime"
-    "babel-plugin-source-map-support"

+    "serverless-bundle": "^1.2.2"
```

The serverless-bundle plugin supports:

- Linting via ESLint (https://eslint.org)
- Caching for faster builds
- Use ES6 `import/export`
- Transpiling unit tests with babel-jest (https://github.com/facebook/jest/tree/master/packages/babel-jest)
- Adding source maps for proper error messages

To get started with serverless-bundle, simply install it:

```
$ npm install --save-dev serverless-bundle
```

Then add it to your `serverless.yml`.

```
plugins:
  - serverless-bundle
```

And to run your tests using the same Babel config used in the plugin add the following to your `package.json`:

```
"scripts": {
  "test": "serverless-bundle test"
}
```

You can read more on the advanced options over on the GitHub README (https://github.com/AnomalyInnovations/serverless-bundle/blob/master/README.md).

Our ever popular Serverless Node.js Starter (https://github.com/AnomalyInnovations/serverless-nodejs-starter) has now been updated to use the serverless-bundle plugin.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/package-lambdas-with-serverless-bundle/1150)

Understanding React Hooks

React Hooks are a way for your function components to “hook” into React’s lifecycle and state. They were introduced in React 16.8.0 (https://github.com/facebook/react/releases/tag/v16.8.0). Previously, only Class based components were able to use React’s lifecycle and state. Aside from enabling function components to do this, Hooks make it incredibly easy to reuse stateful logic between components.

If you are moving to Hooks for the first time, the change can be a little jarring. This chapter is here to help you understand how they work and how to think about them. We want to help you transition from the mental model of Class components to function components with React Hooks. Here is what we’ll be covering:

1. A quick refresher on the lifecycle of Class components
2. An overview of the lifecycle of function components with Hooks
3. A good mental model to understand React Hooks in function components
4. A subtle difference between Class and function components

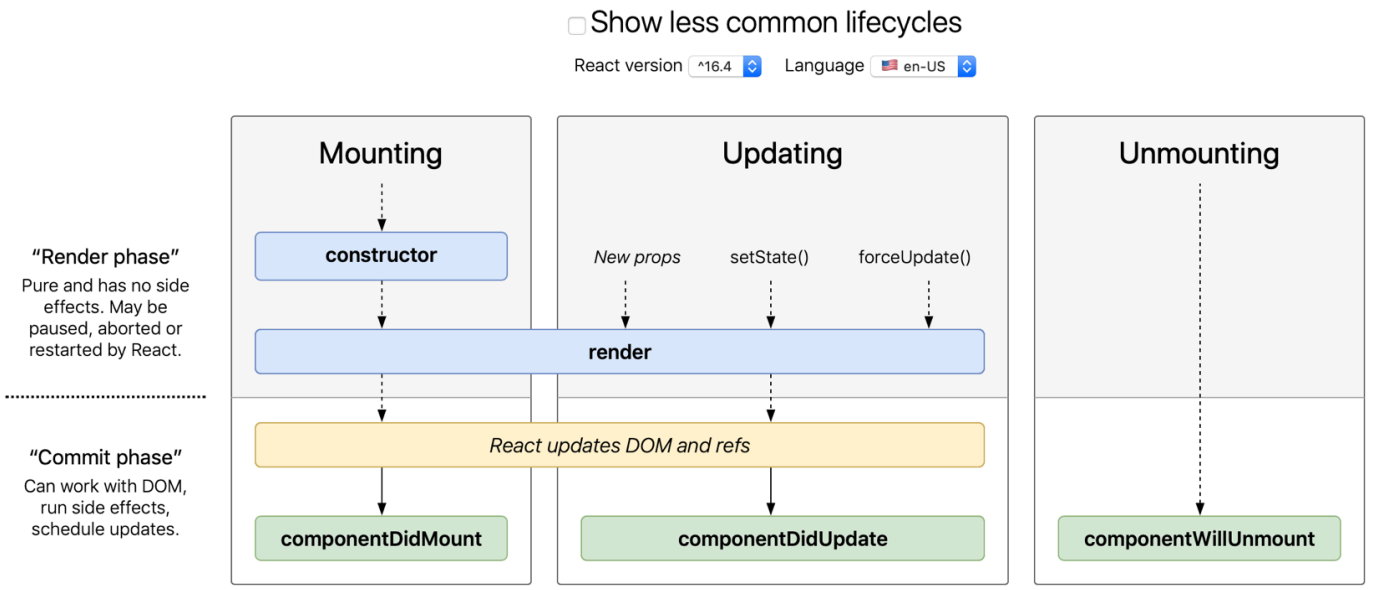
Note that, this chapter does not cover specific React Hooks in detail, the React Docs are a great place for that (https://reactjs.org/docs/hooks-reference.html).

Let’s get started.

The React Class Component Lifecycle

If you are used to using React Class components, you'll be familiar with some of the main lifecycle methods.

- `constructor`
- `render`
- `componentDidMount`
- `componentDidUpdate`
- `componentWillUnmount`
- etc.



You can view the above in detail here (http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/).

Let's understand this quickly with an example. Say you have a component called `Hello`:

```
class Hello extends React.Component {
  constructor(props) {
    super(props);
  }
```

```
  componentDidMount() {
  }

  componentDidUpdate() {
  }

  componentWillUnmount() {
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
      </div>
    );
  }
}
```

This is roughly what React does when creating the `Hello` component. Note that, this is a simplified model and it isn't exactly what happens behind the scenes.

1. React will create a new instance of your component.
   ```
   const HelloInstance = new Hello(someProps);
   ```
2. This calls your component's `constructor(someProps)`.
3. It'll then call `HelloInstance.render()`, to render it for the first time.
4. Next it'll call `HelloInstance.componentDidMount()`. Here you

can run any API calls and call `setState` to update your component.

5. Calling `setState` will in turn cause React to call `HelloInstance.render()`. This is also the case if React wants to re-render the component (maybe because its parent is being re-rendered).

6. After the updated render, React will call `HelloInstance.componentDidUpdate()`.

7. Finally, when it's time to remove your component (maybe the user navigates to a different screen), React will call `HelloInstance.componentWillUnmount()`.

The key thing to understand about the lifecycle is that your Class component is instantiated ONCE and the various lifecycle methods are then called on the SAME instance. This means that you can save some sort of "state" locally in your class instance using class variables. This has some interesting implications that we'll talk about below.

But for now, let's look at the flow for a function component.

The React Function Component Lifecycle

Let's start with a basic React function component and look at how React renders it.

```
function Hello(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
    </div>
```

```
  );
}
```

React will render this by simply running the function!

```
Hello(someProps);
```

And if it needs to be re-rendered, React will run your function again!

Again we are using a simplified React model but the concept is straightforward. For a function component, React simply runs your function every time it needs to render or re-render it.

You'll notice that our simple function component has no control over itself. Also, we can't really do anything in regards to the React render lifecycle like our class component above.

This is where Hooks come in!

Adding React Hooks

React Hooks allows function components to "hook" into the React state and lifecycle. Let's look at an example.

```
function Hello(props) {
  const [ stateVariable, setStateVariable ] =
useState(0);

  useEffect(() => {
    console.log('mount and update');

    return () => {
```

```
      console.log('cleanup');
    };
  });

  return (
    <div>
      <h1>Hello, world!</h1>
    </div>
  );
}
```

We are using two Hooks here; `useState` and `useEffect`. One tells React to store some state for us. While the other, tells React to call us during the render lifecycle.

- When our component gets rendered, we tell React that we want to store something in the state by calling `useState(<VARIABLE>)`. React gives us back `[ stateVariable, setStateVariable ]`, where `stateVariable` is the current value of this variable in the state. And `setStateVariable` is a function that we can call to set the new value of this variable. You can read about how useState works here (https://reactjs.org/docs/hooks-state.html).

- Next we the `useEffect` Hook. We pass in a function that we want React to run every time our component gets rendered or updated. This function can also return a function that'll get called when our component needs to cleanup the old render. So if React renders our component, and we call `setStateVariable` at some point, React will need to re-render it. Here is what roughly what happens:

  ```
  // React renders the component
  ```

```
Hello(someProps);
// Console shows: mount and update

...

// React re-renders the component

// Console shows: cleanup
Hello(someProps);
// Console shows: mount and update
```

- And finally when your component is unmounted or removed, React will call your cleanup function once again.

You'll notice that the lifecycle flow here is not exactly the same as before. And that the `useEffect` Hook is run (and cleans up) on every render. This is by design. The main change you need to make mentally is that unlike Class components, function components are run for every single render. And since they are just simple functions, they internally have no state of their own.

As an aside, you can optionally make `useEffect` call you only for the initial mount and final unmount by passing in an empty array (`[]`) as another argument.

```
useEffect(() => {
  console.log('mount');
  return () => {
    console.log('will unmount');
  }
}, []);
```

You can read about useEffect in detail here (https://reactjs.org/docs/hooks-effect.html).

React Hooks Mental Model

So when you are thinking about function components with Hooks, they are very simple in that they are rerun every time. As you are looking at your code, imagine that it is run in order over single time. And since there is no local state for your functions, the values available are only what React has stored in its state.

As opposed to Class components, where specific methods in your class are called upon render. Additionally, you might have stored some state locally in a state variable. This means that as you are debugging your code, you've to try and keep in mind what the current value of a local state variable is.

This slight difference in local state can introduce some very subtle bugs in the Class component version that is worth understanding in detail. On the other hand thanks to JavaScript Closures (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures), functional components have a more straight forward execution model.

Let's look at this next.

Subtle Differences Between Class & Function Components

This section is based on a great post by Dan Abramov (https://twitter.com/dan_abramov?ref_src=twsrc%5Egoogle%7Ctwcamp%5Eserp%7Ctwgr%5Eauthor), title "How Are Function Components Different from Classes?"

(https://overreacted.io/how-are-function-components-different-from-classes/) that we recommend you read. This isn't specifically related to React Hooks. But we'll go over the key takeaway from that post because it'll help you make the trasition from the Class components mental model to the function components one. This is something you'll need to do as you start using React Hooks.

Using the example from Dan's post; let's compare similar versions of the same component first as a Class.

```
class ProfilePage extends React.Component {
  showMessage = () => {
    alert('Followed ' + this.props.user);
  };

  handleClick = () => {
    setTimeout(this.showMessage, 3000);
  };

  render() {
    return <button onClick=
{this.handleClick}>Follow</button>;
  }
}
```

And now as a function component.

```
function ProfilePage(props) {
  const showMessage = () => {
    alert('Followed ' + props.user);
  };
```

```
  const handleClick = () => {
    setTimeout(showMessage, 3000);
  };

  return (
    <button onClick={handleClick}>Follow</button>
  );
}
```

Take a second to understand what the component does. Imagine that instead of the `setTimeout` call, we are doing some sort of an API call. Both these versions are doing pretty much the same thing.

However, the Class version is buggy in a very subtle way. Dan has a demo version (https://codesandbox.io/s/pjqnl16lm7) of this code for you to try out. Simply click the follow button, try changing the selected profile within 3 seconds and check out what is alerted.

Here is the bug in the Class version. If you click the button and `this.props.user` changes before 3 seconds, then the alerted message is the new user! This isn't surprising if you've followed along this chapter so far. React is using the SAME instance of your Class component between re-renders. Meaning that within our code the `this` object refers to that same instance. So conceptually React changes the `ProfilePage` instance prop by doing something like this:

```
// Create an instance
const ProfilePageInstance = new ProfilePage({ user:
"First User" });
// First render
```

```
ProfilePageInstance.render();

// Button click
this.handleClick();
// Timer is started

// Update prop
ProfilePageInstance.props.user = "New User";
// Re-render
ProfilePageInstance.render();

// Timer completes
// where this <=> ProfilePageInstance
alert('Followed ' + this.props.user);
```

So when the `alert` is run, `this.props.user` is `New User` instead!

Let's look at how the functional version handles this.

```
// First render
ProfilePage({ user: "First User" });

// Button click
handleClick();
// Timer is started

// Re-render with updated props
ProfilePage({ user: "New User" });

// Timer completes
// from the first ProfilePage() call scope
alert('Followed ' + props.user);
```

Here is the critical difference, the `alert` call here is from the scope of the first `ProfilePage()` call scope. This is happens thanks to JavaScript Closures (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures). Since there is no "instance" here, your code is just a regular JavaScript function and is scoped to where it was run.

The above pattern is not specific to React Hooks, it's just how JavaScript functions work. However, if you've been using Class components so far and are transitioning to using React Hooks; we strongly encourage you to really understand this pattern.

Summary

This allows us to think of our components just as regular JavaScript functions. No special order of our lifecycle methods being called and no local state to track. Here's the key takeaway:

"***React simply calls your function components over and over again when it needs to render it. You'll need to use React Hooks to store state and plug into the React render lifecycle. And thanks to JavaScript Closures, your variables are scoped to the specific function call.***"

We hope this chapter helps you create a better mental model for understanding function components with React Hooks. Leave us a comment in the discussion thread below if you want us to expand on something further.

For help and discussion

Comments on this chapter

```
(https://discourse.serverless-
stack.com/t/understanding-react-hooks/1338)
```

Code Splitting in Create React App

Code Splitting is not a necessary step for building React apps. But feel free to follow along if you are curious about what Code Splitting is and how it can help larger React apps.

Code Splitting

While working on React.js single page apps, there is a tendency for apps to grow quite large. A section of the app (or route) might import a large number of components that are not necessary when it first loads. This hurts the initial load time of our app.

You might have noticed that Create React App will generate one large `.js` file while we are building our app. This contains all the JavaScript our app needs. But if a user is simply loading the login page to sign in; it doesn't make sense that we load the rest of the app with it. This isn't a concern early on when our app is quite small but it becomes an issue down the road. To address this, Create React App has a very simple built-in way to split up our code. This feature unsurprisingly, is called Code Splitting.

Create React App (from 1.0 onwards) allows us to dynamically import parts of our app using the `import()` proposal. You can read more about it here (https://facebook.github.io/react/blog/2017/05/18/whats-new-in-create-react-app.html#code-splitting-with-dynamic-import).

While, the dynamic `import()` can be used for any component in our React app; it works really well with React Router. Since, React Router is figuring out which component to load based on the path; it would make sense that we dynamically import those components only when we navigate to them.

Code Splitting and React Router v4

The usual structure used by React Router to set up routing for your app looks something like this.

```
/* Import the components */
import Home from "./containers/Home";
import Posts from "./containers/Posts";
import NotFound from "./containers/NotFound";

/* Use components to define routes */
export default () =>
  <Switch>
    <Route path="/" exact component={Home} />
    <Route path="/posts/:id" exact component={Posts} />
    <Route component={NotFound} />
  </Switch>;
```

We start by importing the components that will respond to our routes. And then use them to define our routes. The `Switch` component renders the route that matches the path.

However, we import all of the components in the route statically at the top. This means, that all these components are loaded regardless of which route is matched. To implement Code Splitting here we are going to want

to only load the component that responds to the matched route.

Create an Async Component

To do this we are going to dynamically import the required component.

✦ CHANGE Add the following to `src/components/AsyncComponent.js`.

```
import React, { Component } from "react";

export default function asyncComponent(importComponent)
{
  class AsyncComponent extends Component {
    constructor(props) {
      super(props);

      this.state = {
        component: null
      };
    }

    async componentDidMount() {
      const { default: component } = await
importComponent();

      this.setState({
        component: component
      });
    }

    render() {
```

```
      const C = this.state.component;

      return C ? <C {...this.props} /> : null;
    }
  }

  return AsyncComponent;
}
```

We are doing a few things here:

1. The `asyncComponent` function takes an argument; a function (`importComponent`) that when called will dynamically import a given component. This will make more sense below when we use `asyncComponent` .
2. On `componentDidMount` , we simply call the `importComponent` function that is passed in. And save the dynamically loaded component in the state.
3. Finally, we conditionally render the component if it has completed loading. If not we simply render `null` . But instead of rendering `null` , you could render a loading spinner. This would give the user some feedback while a part of your app is still loading.

Use the Async Component

Now let's use this component in our routes. Instead of statically importing our component.

```
import Home from "./containers/Home";
```

We are going to use the `asyncComponent` to dynamically import the

component we want.

```
const AsyncHome = asyncComponent(() =>
import("./containers/Home"));
```

It's important to note that we are not doing an import here. We are only passing in a function to `asyncComponent` that will dynamically `import()` when the `AsyncHome` component is created.

Also, it might seem weird that we are passing a function here. Why not just pass in a string (say `./containers/Home`) and then do the dynamic `import()` inside the `AsyncComponent`? This is because we want to explicitly state the component we are dynamically importing. Webpack splits our app based on this. It looks at these imports and generates the required parts (or chunks). This was pointed out by @wSokra (https://twitter.com/wSokra/status/866703557323632640) and @dan_abramov (https://twitter.com/dan_abramov/status/866646657437491201).

We are then going to use the `AsyncHome` component in our routes. React Router will create the `AsyncHome` component when the route is matched and that will in turn dynamically import the `Home` component and continue just like before.

```
<Route path="/" exact component={AsyncHome} />
```

Now let's go back to our Notes project and apply these changes.

✦ CHANGE Your `src/Routes.js` should look like this after the changes.

```
import React from "react";
```

```
import { Route, Switch } from "react-router-dom";
import asyncComponent from
"./components/AsyncComponent";
import AppliedRoute from "./components/AppliedRoute";
import AuthenticatedRoute from
"./components/AuthenticatedRoute";
import UnauthenticatedRoute from
"./components/UnauthenticatedRoute";

const AsyncHome = asyncComponent(() =>
import("./containers/Home"));
const AsyncLogin = asyncComponent(() =>
import("./containers/Login"));
const AsyncNotes = asyncComponent(() =>
import("./containers/Notes"));
const AsyncSignup = asyncComponent(() =>
import("./containers/Signup"));
const AsyncNewNote = asyncComponent(() =>
import("./containers/NewNote"));
const AsyncNotFound = asyncComponent(() =>
import("./containers/NotFound"));

export default ({ childProps }) =>
  <Switch>
    <AppliedRoute
      path="/"
      exact
      component={AsyncHome}
      props={childProps}
    />
    <UnauthenticatedRoute
```

```
      path="/login"
      exact
      component={AsyncLogin}
      props={childProps}
    />
    <UnauthenticatedRoute
      path="/signup"
      exact
      component={AsyncSignup}
      props={childProps}
    />
    <AuthenticatedRoute
      path="/notes/new"
      exact
      component={AsyncNewNote}
      props={childProps}
    />
    <AuthenticatedRoute
      path="/notes/:id"
      exact
      component={AsyncNotes}
      props={childProps}
    />
    {/* Finally, catch all unmatched routes */}
    <Route component={AsyncNotFound} />
  </Switch>
;
```

It is pretty cool that with just a couple of changes, our app is all set up for code splitting. And without adding a whole lot more complexity either! Here is what our `src/Routes.js` looked like before.

```
import React from "react";
import { Route, Switch } from "react-router-dom";
import AppliedRoute from "./components/AppliedRoute";
import AuthenticatedRoute from
"./components/AuthenticatedRoute";
import UnauthenticatedRoute from
"./components/UnauthenticatedRoute";

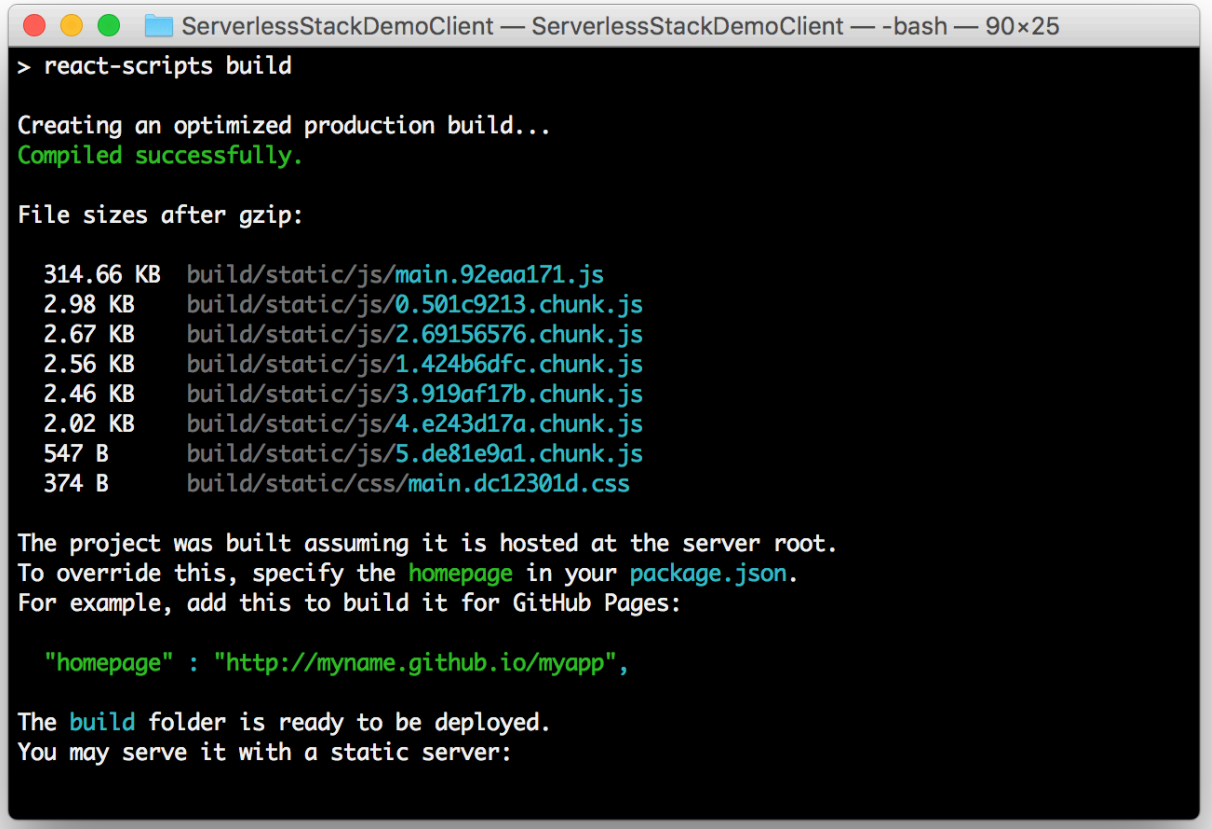
import Home from "./containers/Home";
import Login from "./containers/Login";
import Notes from "./containers/Notes";
import Signup from "./containers/Signup";
import NewNote from "./containers/NewNote";
import NotFound from "./containers/NotFound";

export default ({ childProps }) =>
  <Switch>
    <AppliedRoute
      path="/"
      exact
      component={Home}
      props={childProps}
    />
    <UnauthenticatedRoute
      path="/login"
      exact
      component={Login}
      props={childProps}
    />
    <UnauthenticatedRoute
      path="/signup"
```

```
      exact
      component={Signup}
      props={childProps}
    />
    <AuthenticatedRoute
      path="/notes/new"
      exact
      component={NewNote}
      props={childProps}
    />
    <AuthenticatedRoute
      path="/notes/:id"
      exact
      component={Notes}
      props={childProps}
    />
    {/* Finally, catch all unmatched routes */}
    <Route component={NotFound} />
  </Switch>
;
```

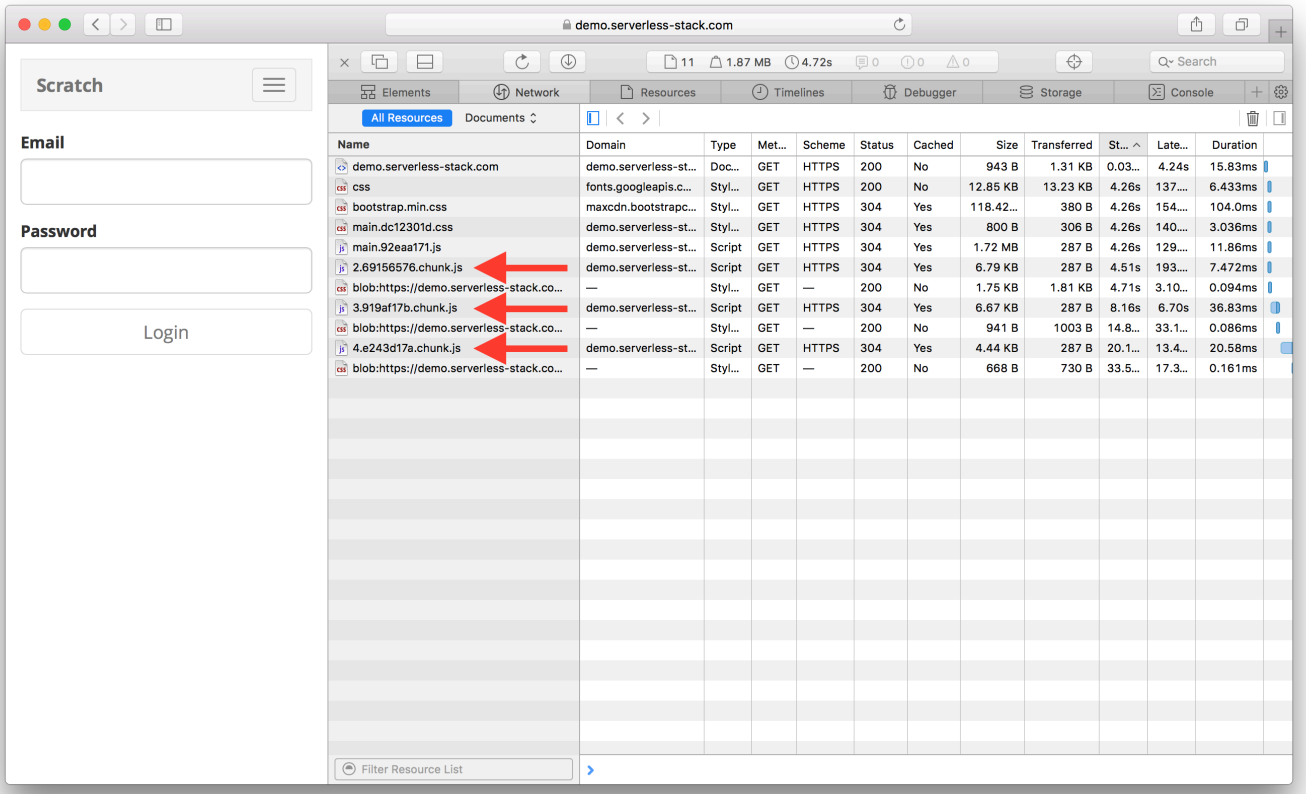
Notice that instead of doing the static imports for all the containers at the top, we are creating these functions that are going to do the dynamic imports for us when necessary.

Now if you build your app using `npm run build`; you'll see the code splitting in action.



Each of those `.chunk.js` files are the different dynamic `import()` calls that we have. Of course, our app is quite small and the various parts that are split up are not significant at all. However, if the page that we use to edit our note included a rich text editor; you can imagine how that would grow in size. And it would unfortunately affect the initial load time of our app.

Now if we deploy our app using `npm run deploy`; you can see the browser load the different chunks on-demand as we browse around in the demo (https://demo.serverless-stack.com).



That's it! With just a few simple changes our app is completely set up to use the code splitting feature that Create React App has.

Next Steps

Now this seems really easy to implement but you might be wondering what happens if the request to import the new component takes too long, or fails. Or maybe you want to preload certain components. For example, a user is on your login page about to login and you want to preload the homepage.

It was mentioned above that you can add a loading spinner while the import is in progress. But we can take it a step further and address some of these edge cases. There is an excellent higher order component that does a lot of this well; it's called **react-loadable** (https://github.com/thejameskyle/react-loadable).

All you need to do to use it is install it.

```
$ npm install --save react-loadable
```

Use it instead of the `asyncComponent` that we had above.

```
const AsyncHome = Loadable({
  loader: () => import("./containers/Home"),
  loading: MyLoadingComponent
});
```

And `AsyncHome` is used exactly as before. Here the `MyLoadingComponent` would look something like this.

```
const MyLoadingComponent = ({isLoading, error}) => {
  // Handle the loading state
  if (isLoading) {
    return <div>Loading...</div>;
  }
  // Handle the error state
  else if (error) {
    return <div>Sorry, there was a problem loading the
page.</div>;
  }
  else {
    return null;
  }
};
```

It's a simple component that handles all the different edge cases gracefully.

To add preloading and to further customize this; make sure to check out the other options and features that react-loadable (https://github.com/thejameskyle/react-loadable) has. And have fun code splitting!

CHECKPOINT

For reference, here is the code we have so far.

Frontend Source :code-splitting-in-create-react-app (https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/code-splitting-in-create-react-app)

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/code-splitting-in-create-react-app/98)

Environments in Create React App

While developing your frontend React app and working with an API backend, you'll often need to create multiple environments to work with. For example, you might have an environment called dev that might be connected to the dev stage of your serverless backend. This is to ensure that you are working in an environment that is isolated from your production version.

Aside from isolating the resources used, having a separate environment that mimics your production version can really help with testing your changes before they go live. You can take this idea of environments further by having a staging environment that can even have snapshots of the live database to give you as close to a production setup as possible. This type of setup can sometimes help track down bugs and issues that you might run into only on our live environment and not on local.

In this chapter we will look at some simple ways to configure multiple environments in our React app. There are many different ways to do this but here is a simple one based on what we have built in first part of this guide (/#the-basics).

Custom Environment Variables

Create React App (https://github.com/facebookincubator/create-react-

app/blob/master/packages/react-scripts/template/README.md#adding-custom-environment-variables) has support for custom environment variables baked into the build system. To set a custom environment variable, simply set it while starting the Create React App build process.

```
$ REACT_APP_TEST_VAR=123 npm start
```

Here `REACT_APP_TEST_VAR` is the custom environment variable and we are setting it to the value `123`. In our app we can access this variable as `process.env.REACT_APP_TEST_VAR`. So the following line in our app:

```
console.log(process.env.REACT_APP_TEST_VAR);
```

Will print out `123` in our console.

Note that, these variables are embedded during build time. Also, only the variables that start with `REACT_APP_` are embedded in our app. All the other environment variables are ignored.

Configuring Environments

We can use this idea of custom environment variables to configure our React app for specific environments. Say we used a custom environment variable called `REACT_APP_STAGE` to denote the environment our app is in. And we wanted to configure two environments for our app:

- One that we will use for our local development and also to test before pushing it to live. Let's call this one `dev`.
- And our live environment that we will only push to, once we are comfortable with our changes. Let's call it `production`.

The first thing we can do is to configure our build system with the `REACT_APP_STAGE` environment variable. Currently the `scripts` portion of our `package.json` looks something like this:

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --env=jsdom",
  "predeploy": "npm run build",
  "deploy": "aws s3 sync build/
s3://YOUR_S3_DEPLOY_BUCKET_NAME",
  "postdeploy": "aws cloudfront create-invalidation --
distribution-id YOUR_CF_DISTRIBUTION_ID --paths '/*' &&
aws cloudfront create-invalidation --distribution-id
YOUR_WWW_CF_DISTRIBUTION_ID --paths '/*'",
  "eject": "react-scripts eject"
}
```

Recall that the `YOUR_S3_DEPLOY_BUCKET_NAME` is the S3 bucket we created to host our React app back in the Create an S3 bucket (/chapters/create-an-s3-bucket.html) chapter. And `YOUR_CF_DISTRIBUTION_ID` and `YOUR_WWW_CF_DISTRIBUTION_ID` are the CloudFront Distributions for the apex (/chapters/create-a-cloudfront-distribution.html) and www (/chapters/setup-www-domain-redirect.html) domains.

Here we only have one environment and we use it for our local development and on live. The `npm start` command runs our local server and `npm run deploy` command deploys our app to live.

To set our two environments we can change this to:

```
"scripts": {
  "start": "REACT_APP_STAGE=dev react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --env=jsdom",

  "predeploy": "REACT_APP_STAGE=dev npm run build",
  "deploy": "aws s3 sync build/
s3://YOUR_DEV_S3_DEPLOY_BUCKET_NAME",
  "postdeploy": "aws cloudfront create-invalidation --
distribution-id YOUR_DEV_CF_DISTRIBUTION_ID --paths
'/*' && aws cloudfront create-invalidation --
distribution-id YOUR_DEV_WWW_CF_DISTRIBUTION_ID --paths
'/*'",

  "predeploy:prod": "REACT_APP_STAGE=production npm run
build",
  "deploy:prod": "aws s3 sync build/
s3://YOUR_PROD_S3_DEPLOY_BUCKET_NAME",
  "postdeploy:prod": "aws cloudfront create-
invalidation --distribution-id
YOUR_PROD_CF_DISTRIBUTION_ID --paths '/*' && aws
cloudfront create-invalidation --distribution-id
YOUR_PROD_WWW_CF_DISTRIBUTION_ID --paths '/*'",

  "eject": "react-scripts eject"
}
```

We are doing a few things of note here:

1. We use the `REACT_APP_STAGE=dev` for our `npm start` command.
2. We also have dev versions of our S3 and CloudFront Distributions

called `YOUR_DEV_S3_DEPLOY_BUCKET_NAME`, `YOUR_DEV_CF_DISTRIBUTION_ID`, and `YOUR_DEV_WWW_CF_DISTRIBUTION_ID`.

3. We default `npm run deploy` to the dev environment and dev versions of our S3 and CloudFront Distributions. We also build using the `REACT_APP_STAGE=dev` environment variable.
4. We have production versions of our S3 and CloudFront Distributions called `YOUR_PROD_S3_DEPLOY_BUCKET_NAME`, `YOUR_PROD_CF_DISTRIBUTION_ID`, and `YOUR_PROD_WWW_CF_DISTRIBUTION_ID`.
5. Finally, we create a specific version of the deploy script for the production environment with `npm run deploy:prod`. And just like the dev version of this command, it builds using the `REACT_APP_STAGE=production` environment variable and the production versions of the S3 and CloudFront Distributions.

Note that you don't have to replicate the S3 and CloudFront Distributions for the dev version. But it does help if you want to mimic the live version as much as possible.

Using Environment Variables

Now that we have our build commands set up with the custom environment variables, we are ready to use them in our app.

Currently, our `src/config.js` looks something like this:

```
export default {
  MAX_ATTACHMENT_SIZE: 5000000,
  s3: {
```

```
    BUCKET: "YOUR_S3_UPLOADS_BUCKET_NAME"
  },
  apiGateway: {
    REGION: "YOUR_API_GATEWAY_REGION",
    URL: "YOUR_API_GATEWAY_URL"
  },
  cognito: {
    REGION: "YOUR_COGNITO_REGION",
    USER_POOL_ID: "YOUR_COGNITO_USER_POOL_ID",
    APP_CLIENT_ID: "YOUR_COGNITO_APP_CLIENT_ID",
    IDENTITY_POOL_ID: "YOUR_IDENTITY_POOL_ID"
  }
};
```

To use the `REACT_APP_STAGE` variable, we are just going to set the config conditionally.

```
const dev = {
  s3: {
    BUCKET: "YOUR_DEV_S3_UPLOADS_BUCKET_NAME"
  },
  apiGateway: {
    REGION: "YOUR_DEV_API_GATEWAY_REGION",
    URL: "YOUR_DEV_API_GATEWAY_URL"
  },
  cognito: {
    REGION: "YOUR_DEV_COGNITO_REGION",
    USER_POOL_ID: "YOUR_DEV_COGNITO_USER_POOL_ID",
    APP_CLIENT_ID: "YOUR_DEV_COGNITO_APP_CLIENT_ID",
    IDENTITY_POOL_ID: "YOUR_DEV_IDENTITY_POOL_ID"
  }
```

```
};

const prod = {
  s3: {
    BUCKET: "YOUR_PROD_S3_UPLOADS_BUCKET_NAME"
  },
  apiGateway: {
    REGION: "YOUR_PROD_API_GATEWAY_REGION",
    URL: "YOUR_PROD_API_GATEWAY_URL"
  },
  cognito: {
    REGION: "YOUR_PROD_COGNITO_REGION",
    USER_POOL_ID: "YOUR_PROD_COGNITO_USER_POOL_ID",
    APP_CLIENT_ID: "YOUR_PROD_COGNITO_APP_CLIENT_ID",
    IDENTITY_POOL_ID: "YOUR_PROD_IDENTITY_POOL_ID"
  }
};

const config = process.env.REACT_APP_STAGE ===
'production'
  ? prod
  : dev;

export default {
  // Add common config values here
  MAX_ATTACHMENT_SIZE: 5000000,
  ...config
};
```

This is pretty straightforward. We simply have a set of configs for dev and for production. The configs point to a separate set of resources for our dev

and production environments. And using `process.env.REACT_APP_STAGE` we decide which one to use.

Again, it might not be necessary to replicate the resources for each of the environments. But it is pretty important to separate your live resources from your dev ones. You do not want to be testing your changes directly on your live database.

So to recap:

- The `REACT_APP_STAGE` custom environment variable is set to either `dev` or `production`.
- While working locally we use the `npm start` command which uses our dev environment.
- The `npm run deploy` command then deploys by default to dev.
- Once we are comfortable with the dev version, we can deploy to production using the `npm run deploy:prod` command.

This entire setup is fairly straightforward and can be extended to multiple environments. You can read more on custom environment variables in Create React App here (https://github.com/facebookincubator/create-react-app/blob/master/packages/react-scripts/template/README.md#adding-custom-environment-variables).

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/environments-in-create-react-app/30)

Deploy a React App to AWS

In this section we'll be looking at how to deploy your React app as a static website on AWS.

The basic setup we are going to be using will look something like this:

1. Upload the assets of our app
2. Use a CDN to serve out our assets
3. Point our domain to the CDN distribution
4. Switch to HTTPS with a SSL certificate

AWS provides quite a few services that can help us do the above. We are going to use S3 (https://aws.amazon.com/s3/) to host our assets, CloudFront (https://aws.amazon.com/cloudfront/) to serve it, Route 53 (https://aws.amazon.com/route53/) to manage our domain, and Certificate Manager (https://aws.amazon.com/certificate-manager/) to handle our SSL certificate.

So let's get started by first configuring our S3 bucket to upload the assets of our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploy-the-frontend/39)

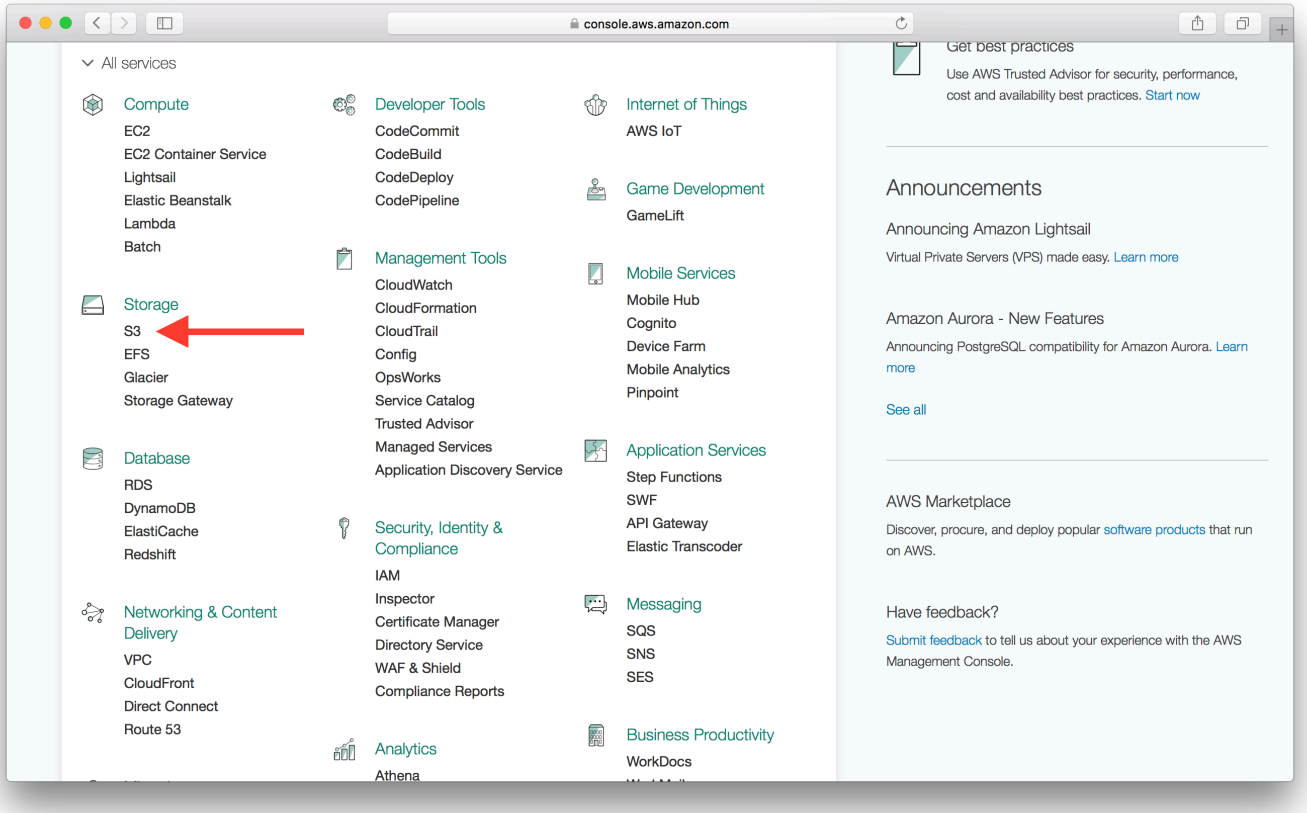
Create an S3 Bucket

To be able to host our note taking app, we need to upload the assets that are going to be served out statically on S3. S3 has a concept of buckets (or folders) to separate different types of files.

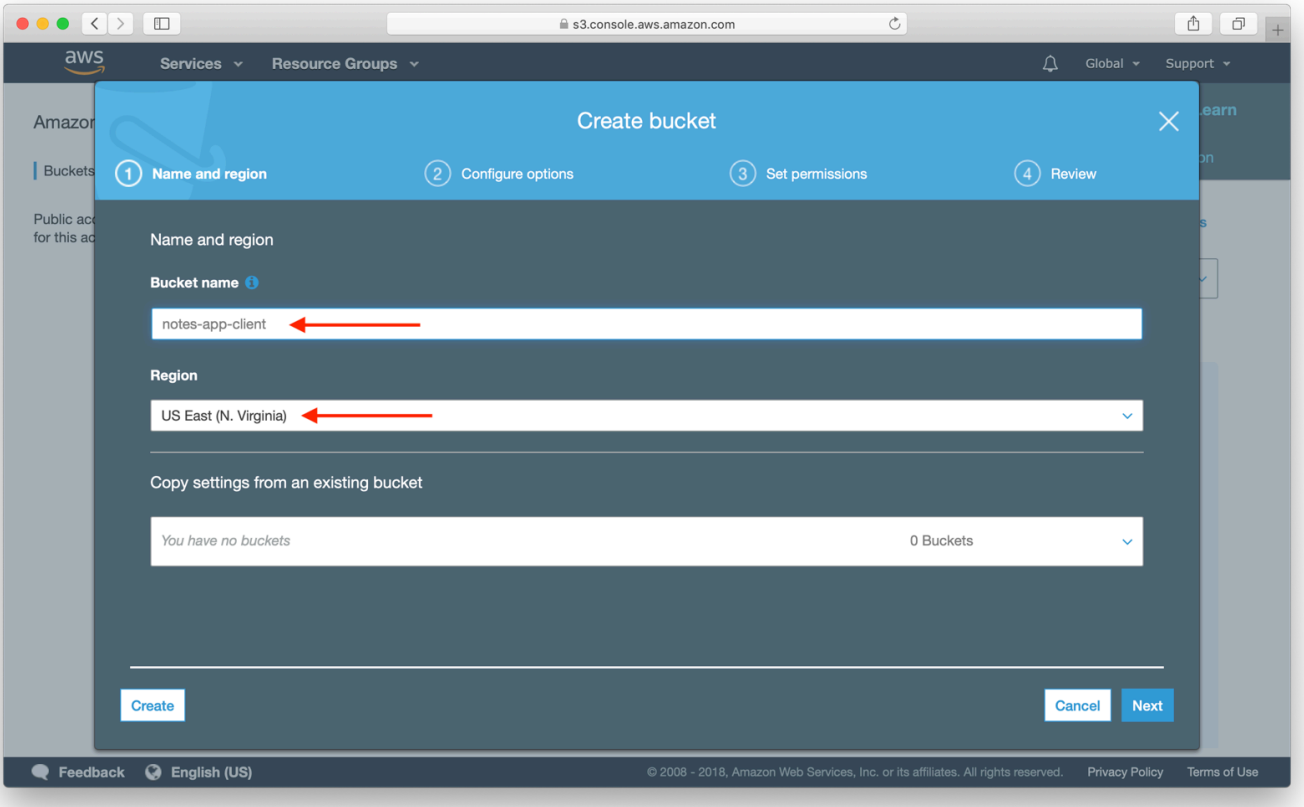
A bucket can also be configured to host the assets in it as a static website and is automatically assigned a publicly accessible URL. So let's get started.

Create the Bucket

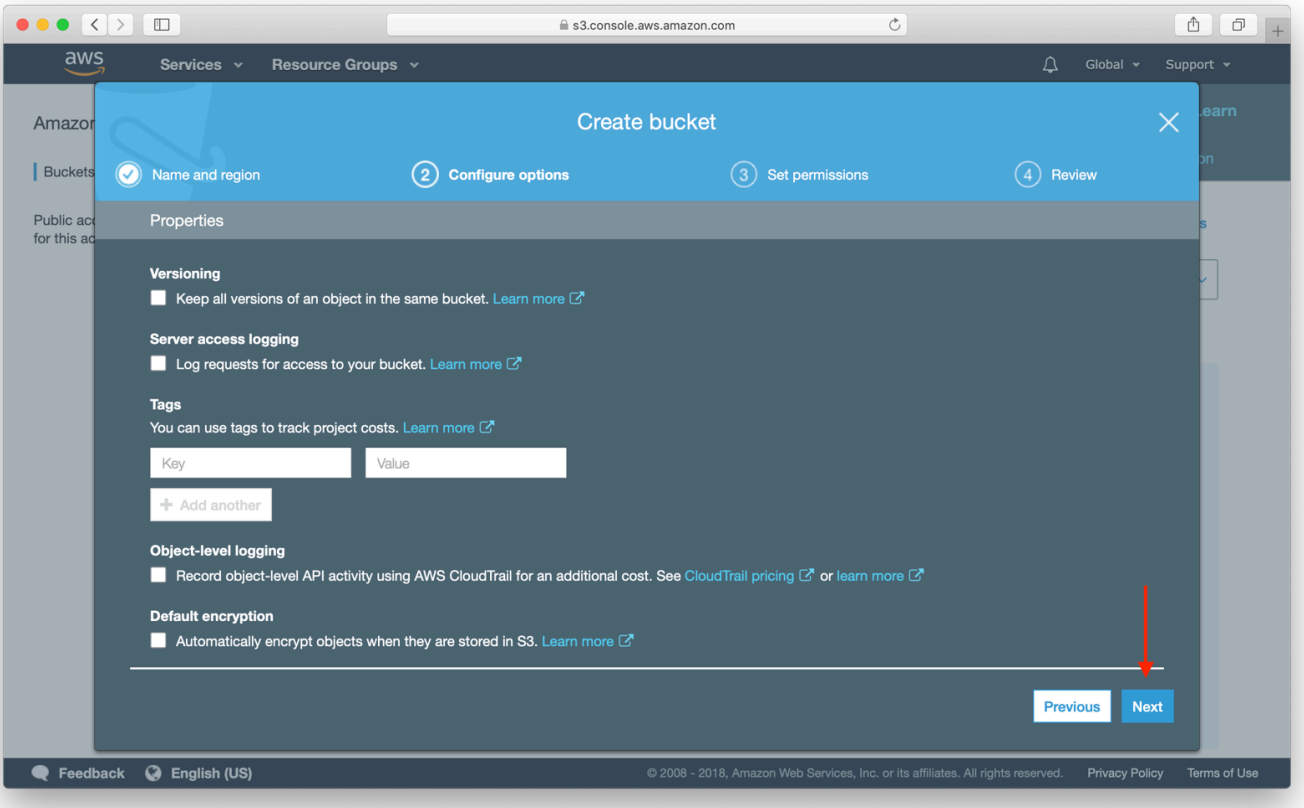
First, log in to your AWS Console (https://console.aws.amazon.com) and select S3 from the list of services.



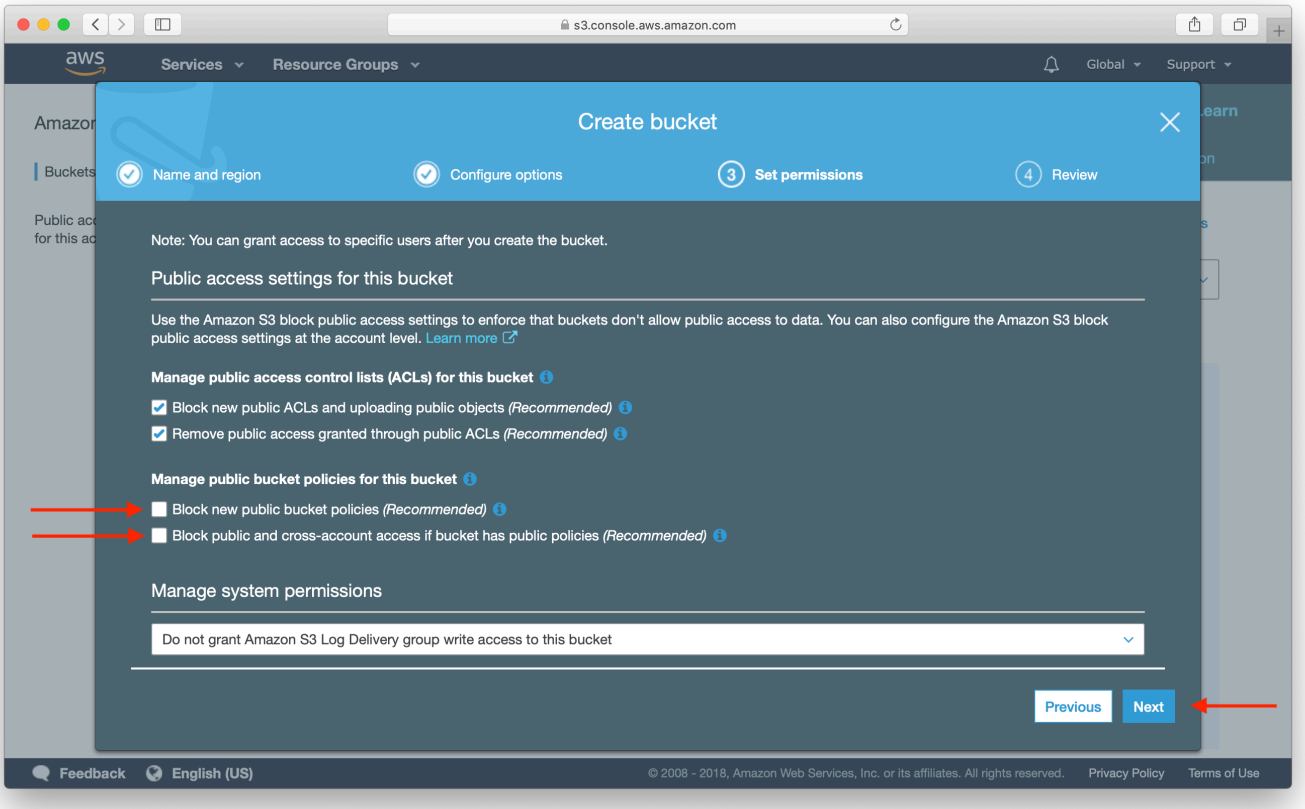
Select **Create Bucket** and pick a name for your application and select the **US East (N. Virginia) Region** Region. Since our application is being served out using a CDN, the region should not matter to us.



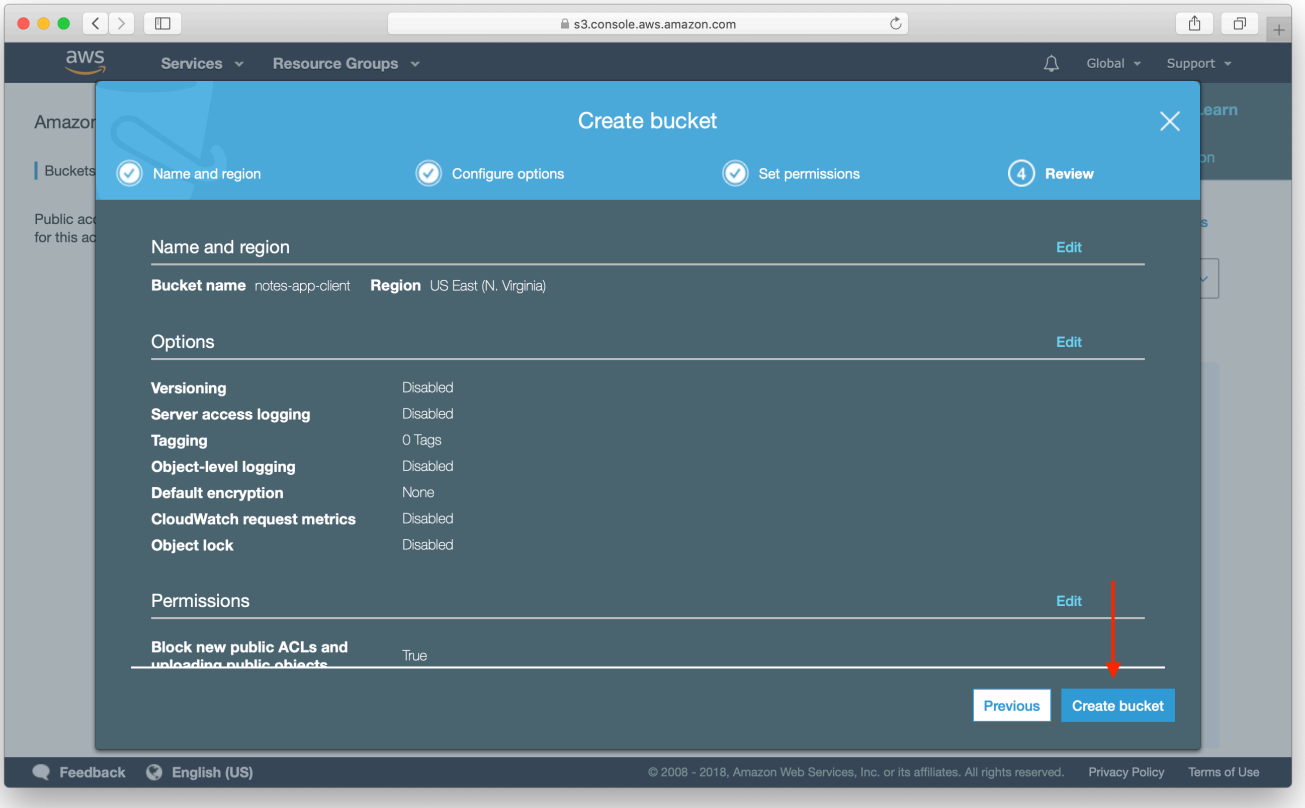
Click **Next** through the configure options step.



In the permissions step, make sure to uncheck **Block new public bucket policies** and **Block public and cross-account access if bucket has public policies**. Making buckets public is a common security error, but in our case we'll be serving our app from the bucket, so want it to be public.

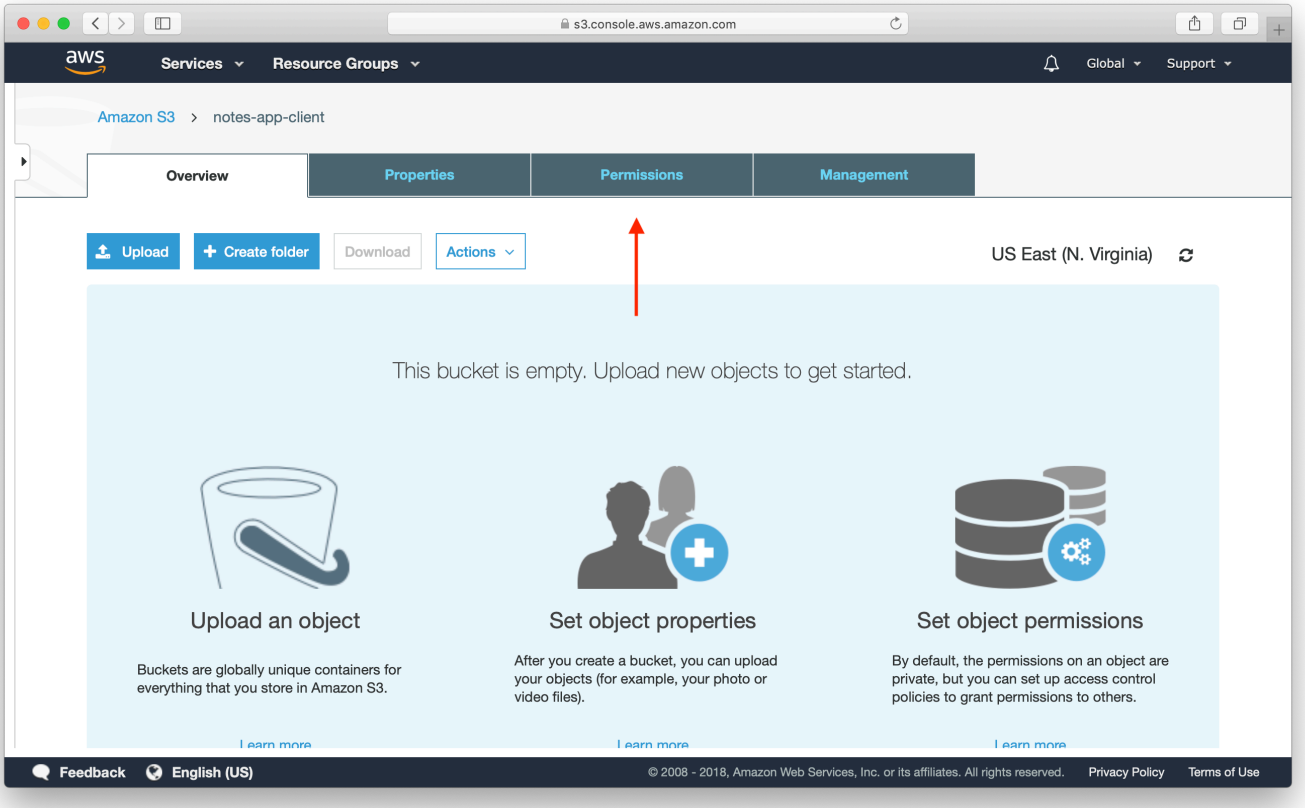


Click **Create bucket** on the review page to create the bucket.



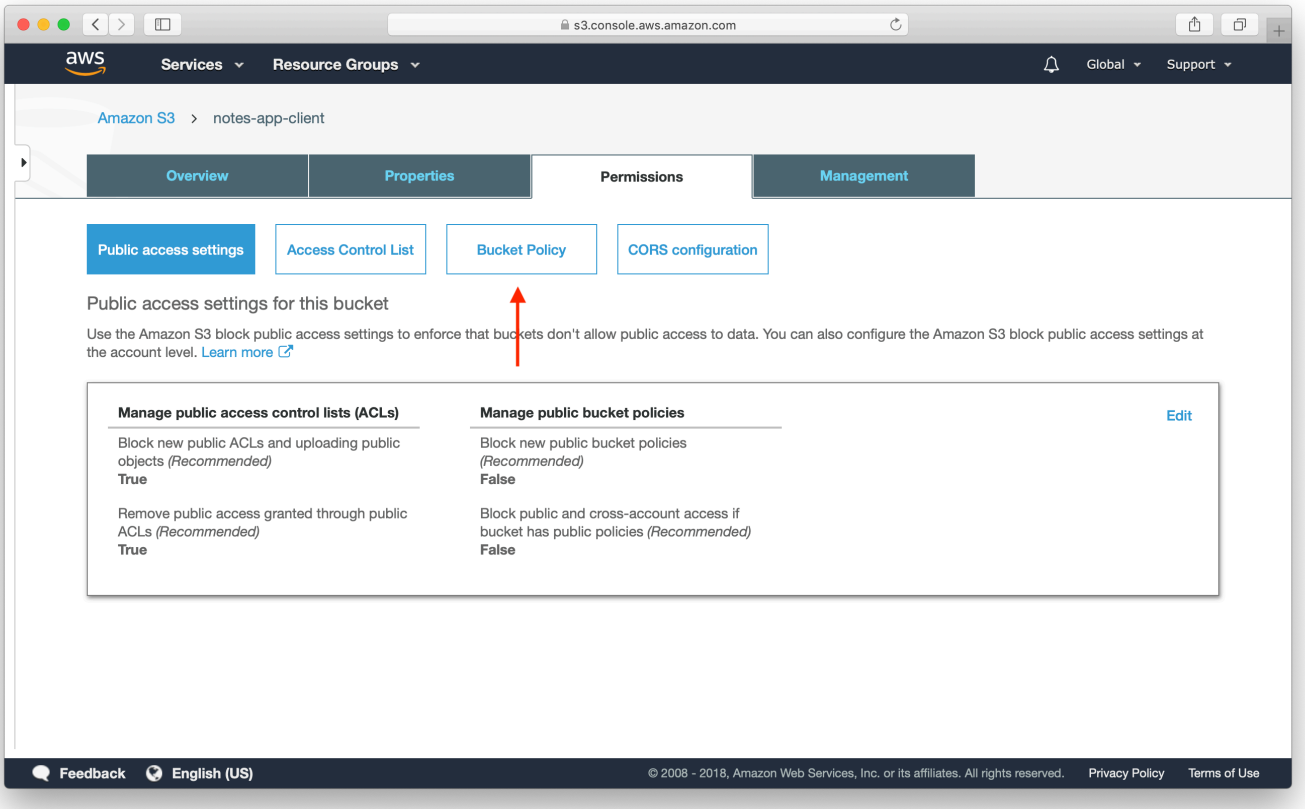
Now click on your newly created bucket from the list and navigate to its

permissions panel by clicking **Permissions**.



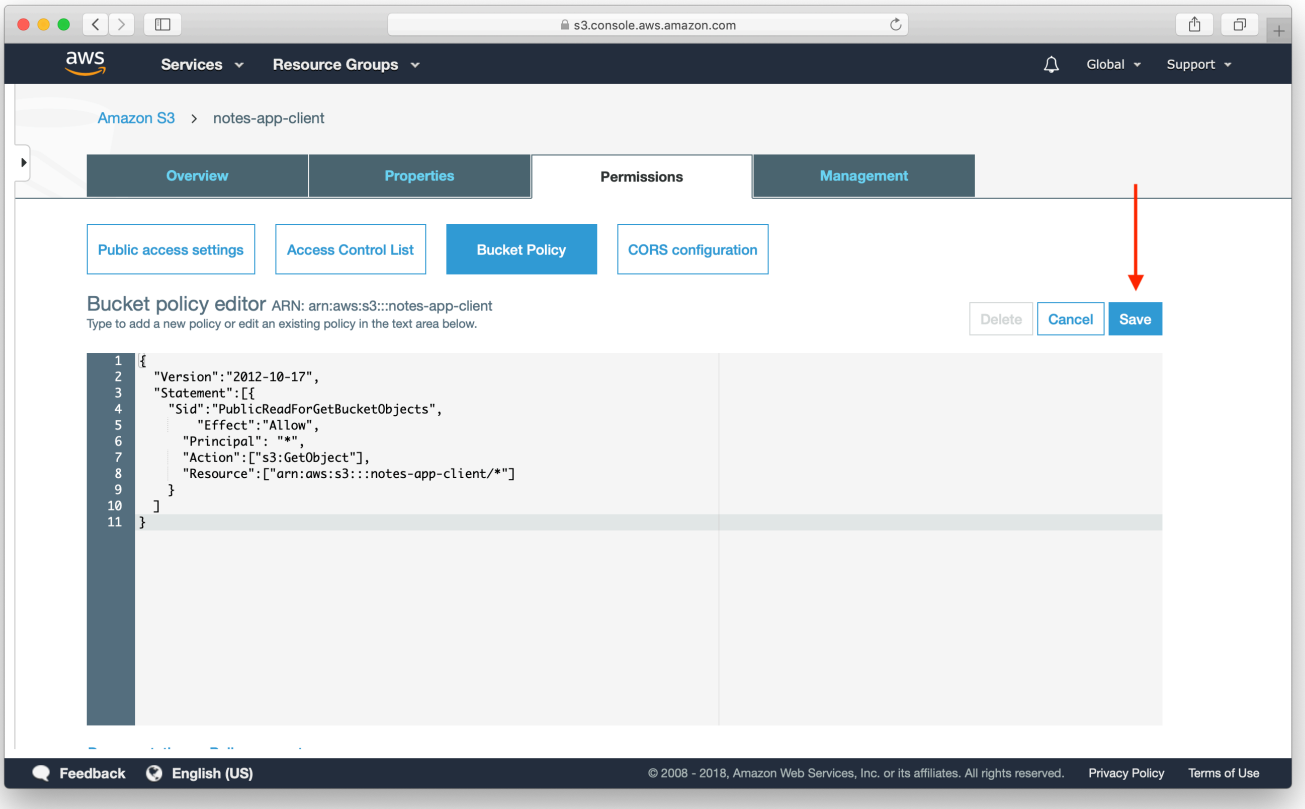
Add Permissions

Buckets by default are not publicly accessible, so we need to change the S3 Bucket Permission. Select the **Bucket Policy** from the permissions panel.



✦ CHANGE Add the following bucket policy into the editor. Where `notes-app-client` is the name of our S3 bucket. Make sure to use the name of your bucket here.

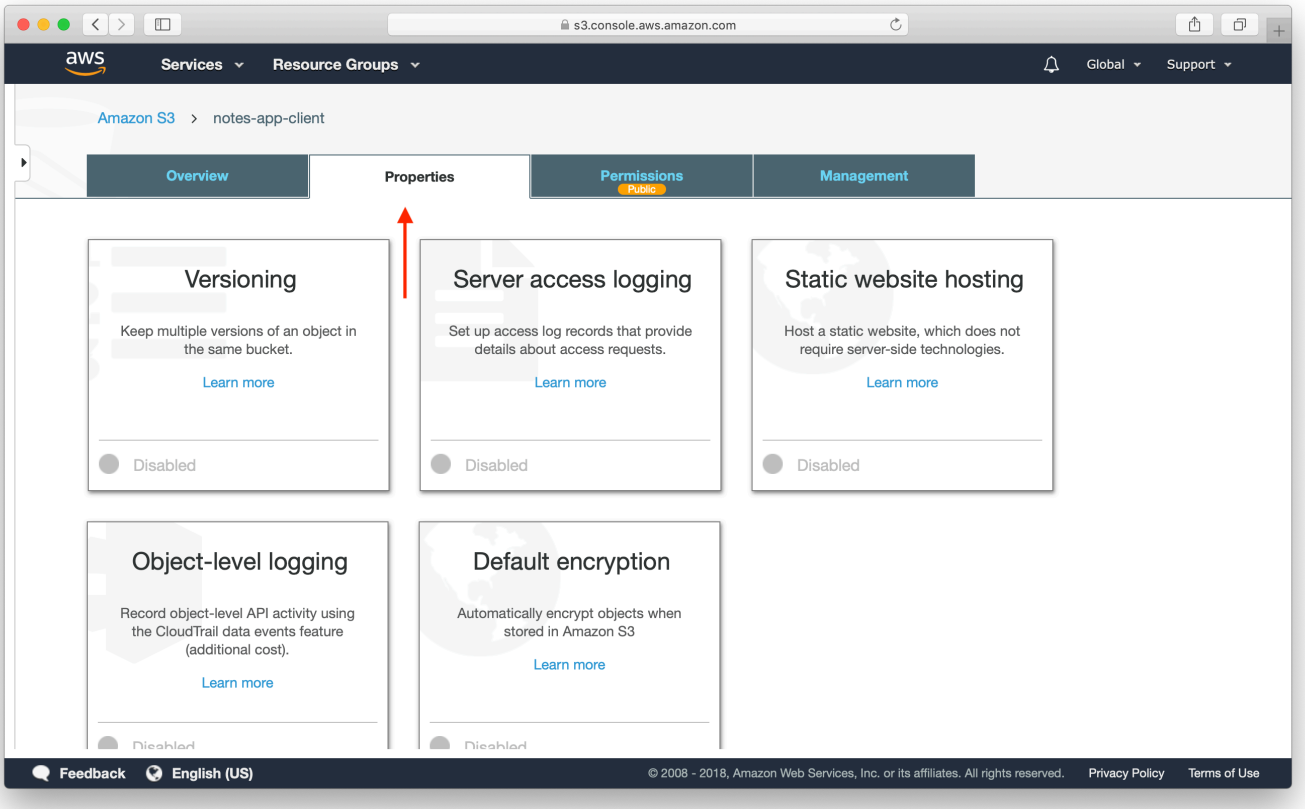
```
{
  "Version":"2012-10-17",
  "Statement":[{
        "Sid":"PublicReadForGetBucketObjects",
        "Effect":"Allow",
          "Principal": "*",
      "Action":["s3:GetObject"],
      "Resource":["arn:aws:s3:::notes-app-client/*"]
    }
  ]
}
```



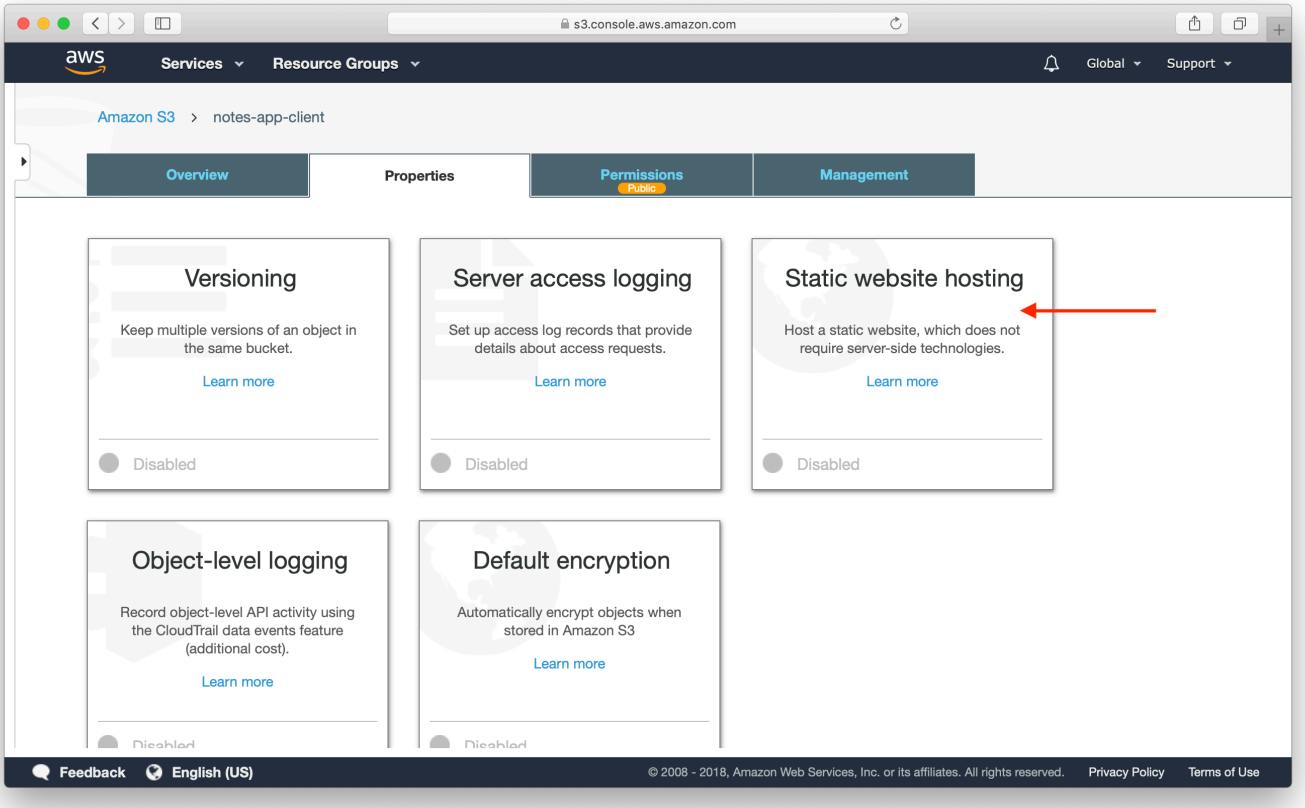
And hit **Save**.

Enable Static Web Hosting

And finally we need to turn our bucket into a static website. Select the **Properties** tab from the top panel.

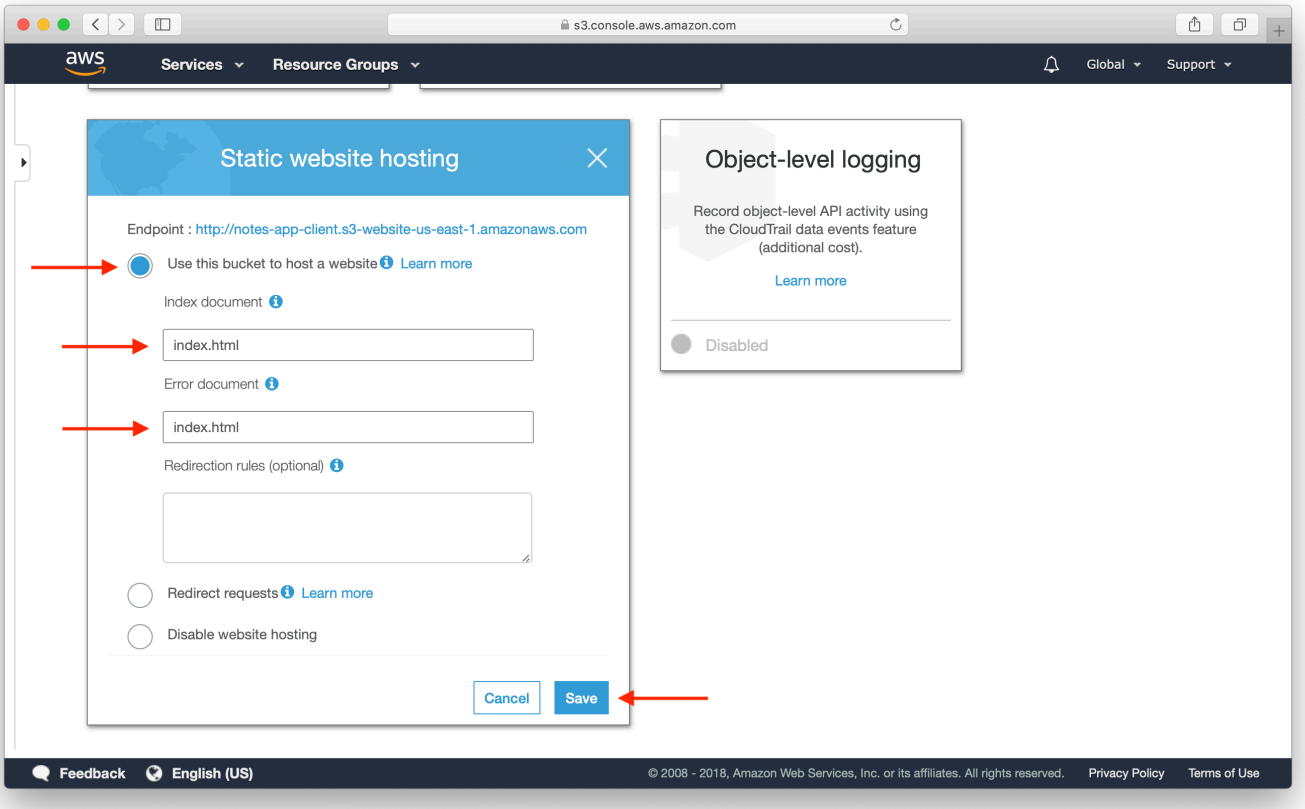


Select **Static website hosting.**



Now select **Use this bucket to host a website** and add our `index.html` as the **Index Document** and the **Error Document**. Since we are letting React handle 404s, we can simply redirect our errors to our `index.html` as well. Hit **Save** once you are done.

This panel also shows us where our app will be accessible. AWS assigns us a URL for our static website. In this case the URL assigned to me is `notes-app-client.s3-website-us-east-1.amazonaws.com`.



Now that our bucket is all set up and ready, let's go ahead and upload our assets to it.

For help and discussion

Comments on this chapter (https://discourse.serverless-

```
stack.com/t/create-an-s3-bucket/48)
```

Deploy to S3

Now that our S3 Bucket is created we are ready to upload the assets of our app.

Build Our App

Create React App comes with a convenient way to package and prepare our app for deployment. From our working directory simply run the following command.

```
$ npm run build
```

This packages all of our assets and places them in the `build/` directory.

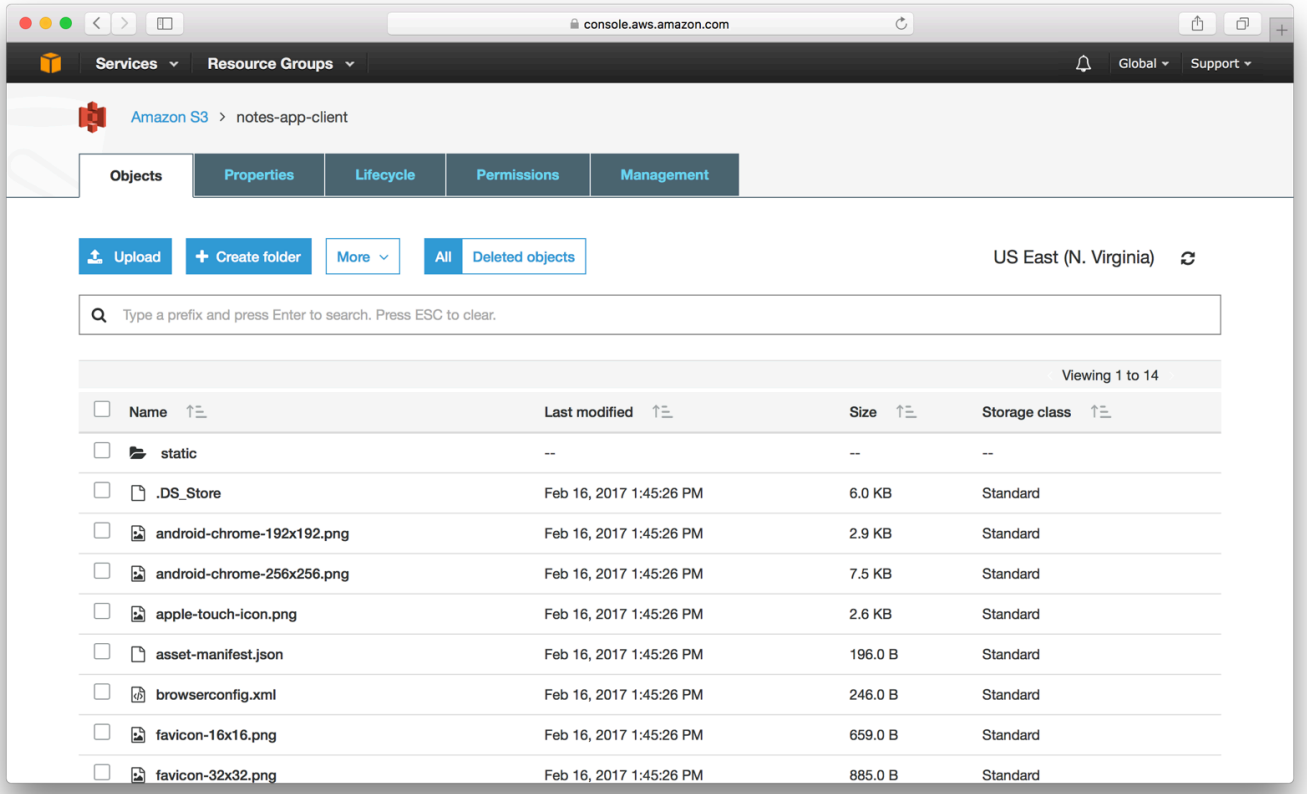
Upload to S3

Now to deploy simply run the following command; where `YOUR_S3_DEPLOY_BUCKET_NAME` is the name of the S3 Bucket we created in the Create an S3 bucket (/chapters/create-an-s3-bucket.html) chapter.

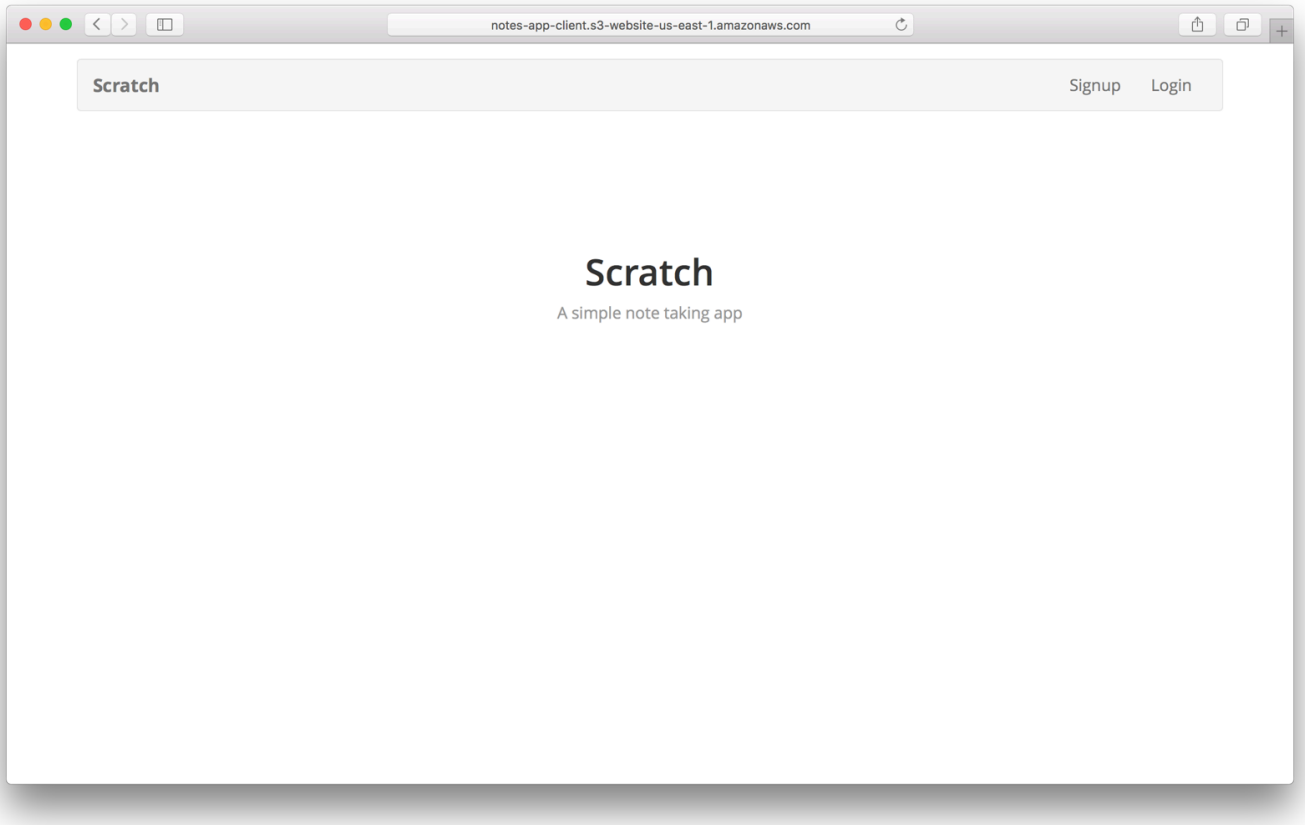
```
$ aws s3 sync build/ s3://YOUR_S3_DEPLOY_BUCKET_NAME
```

All this command does is that it syncs the `build/` directory with our bucket on S3. Just as a sanity check, go into the S3 section in your AWS Console (https://console.aws.amazon.com/console/home) and check if

your bucket has the files we just uploaded.



And our app should be live on S3! If you head over to the URL assigned to you (in my case it is http://notes-app-client.s3-website-us-east-1.amazonaws.com (http://notes-app-client.s3-website-us-east-1.amazonaws.com)), you should see it live.



Next we'll configure CloudFront to serve our app out globally.

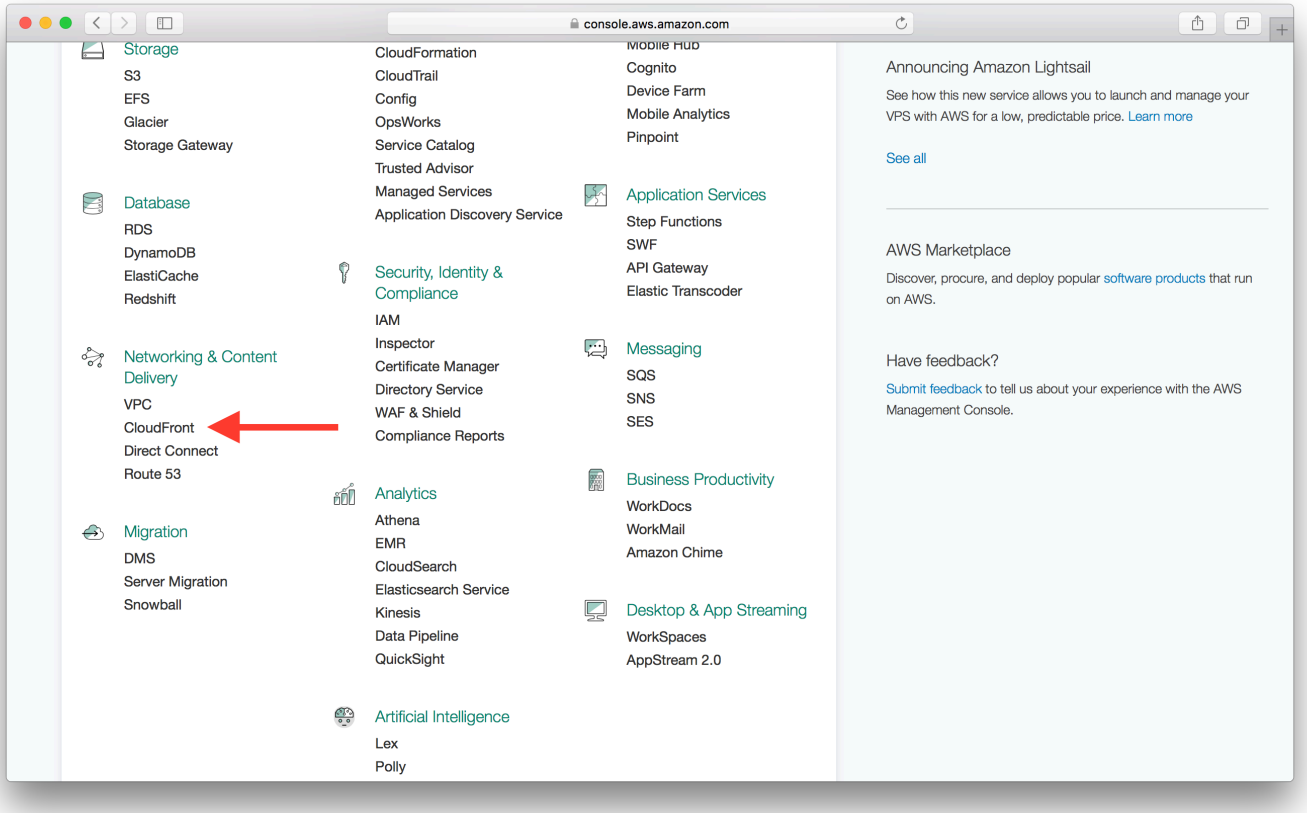
For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploy-to-s3/134)

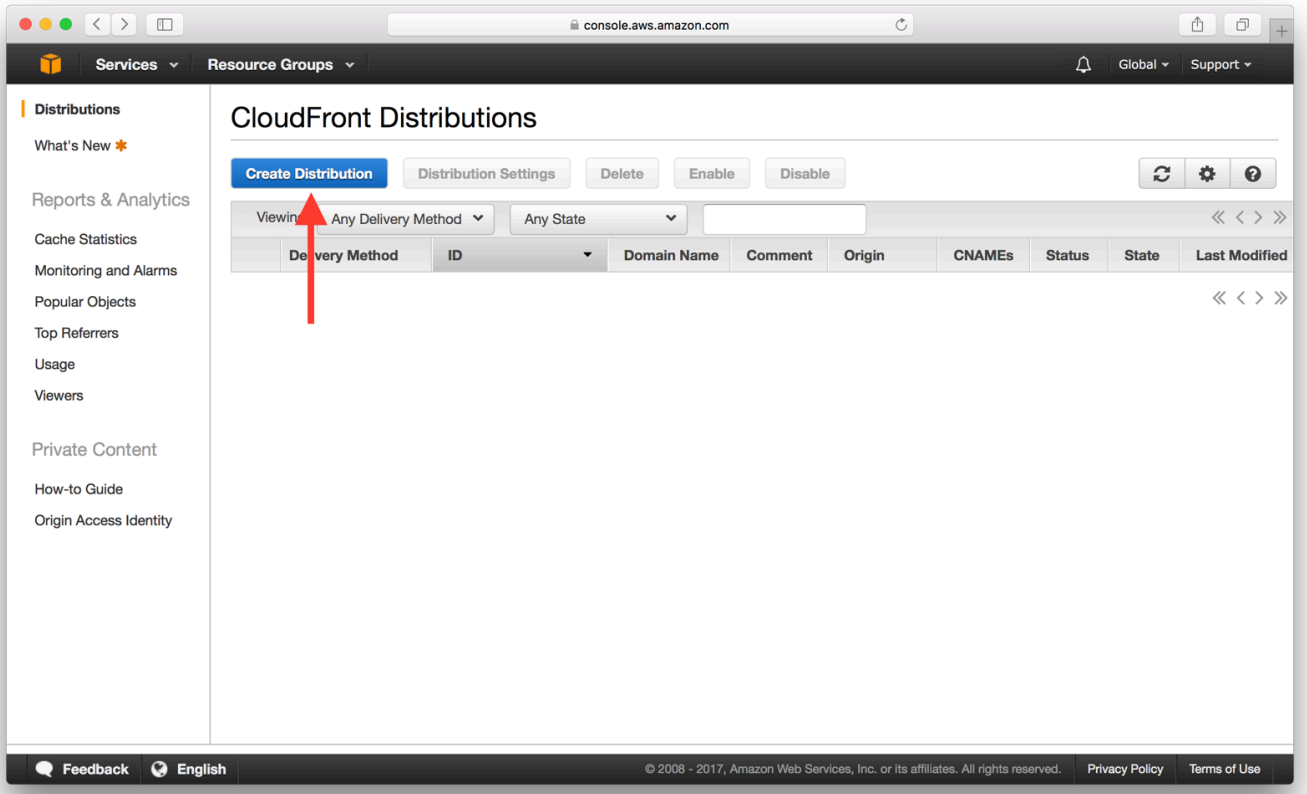
Create a CloudFront Distribution

Now that we have our app up and running on S3, let's serve it out globally through CloudFront. To do this we need to create an AWS CloudFront Distribution.

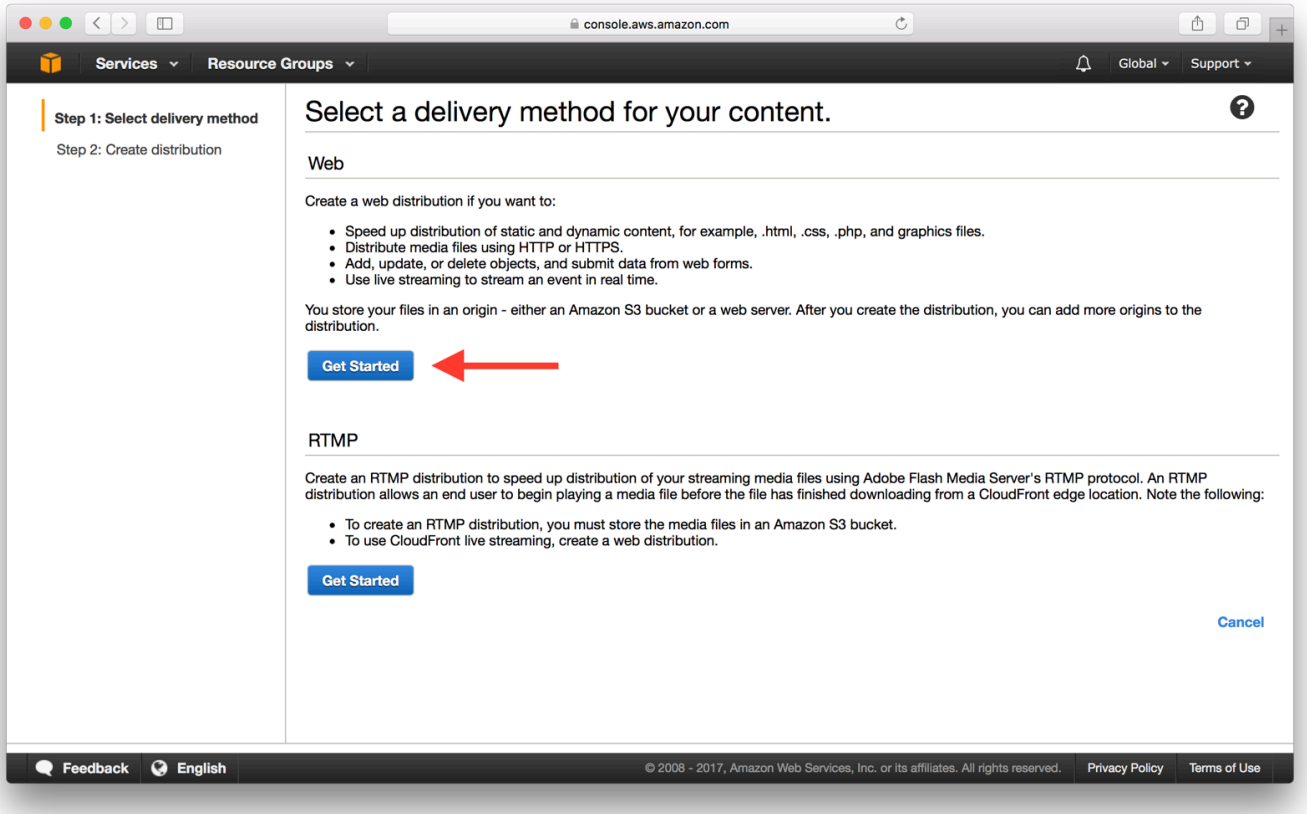
Select CloudFront from the list of services in your AWS Console (https://console.aws.amazon.com).



Then select **Create Distribution**.



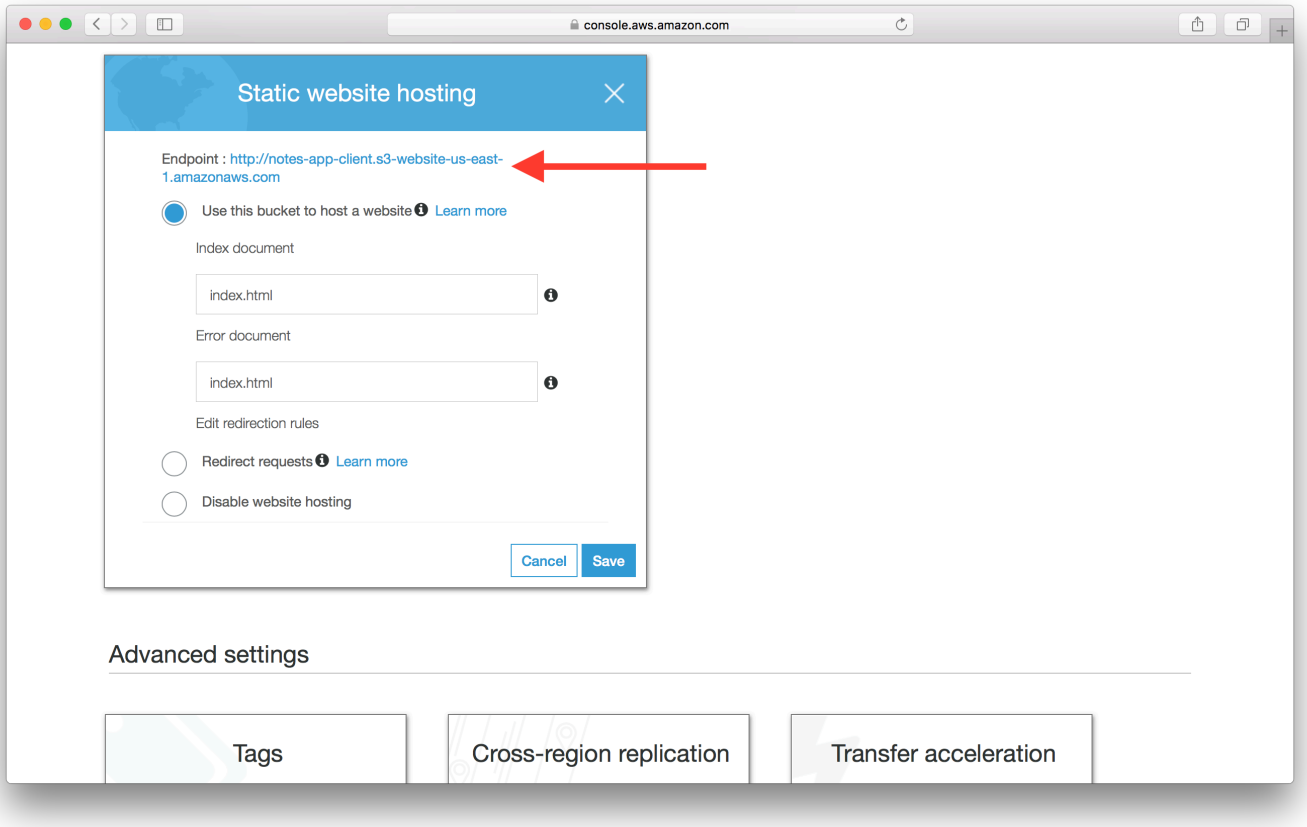
And then in the **Web** section select **Get Started**.



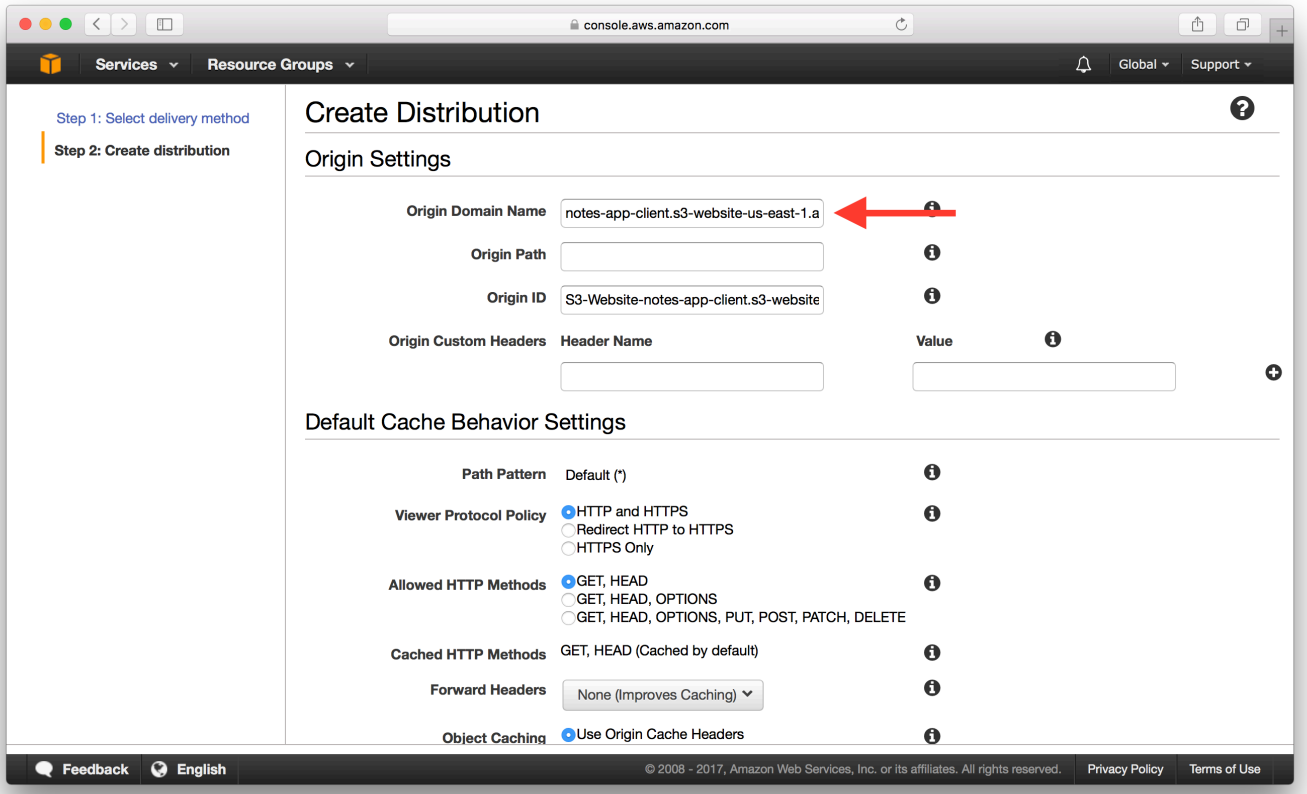
In the Create Distribution form we need to start by specifying the Origin

Domain Name for our Web CloudFront Distribution. This field is pre-filled with a few options including the S3 bucket we created. But we are **not** going to select on the options in the dropdown. This is because the options here are the REST API endpoints for the S3 bucket instead of the one that is set up as a static website.

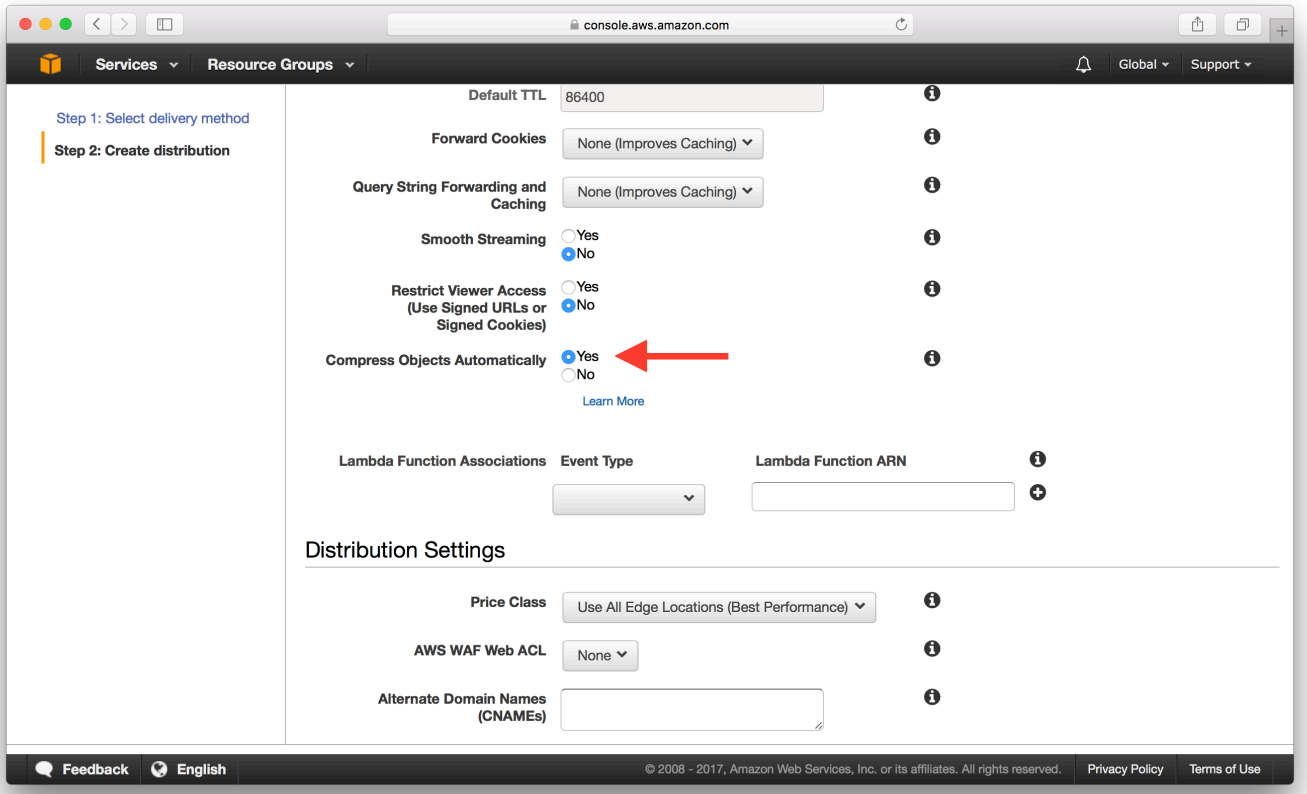
You can grab the S3 website endpoint from the **Static website hosting** panel for your S3 bucket. We had configured this in the previous chapter. Copy the URL in the **Endpoint** field.



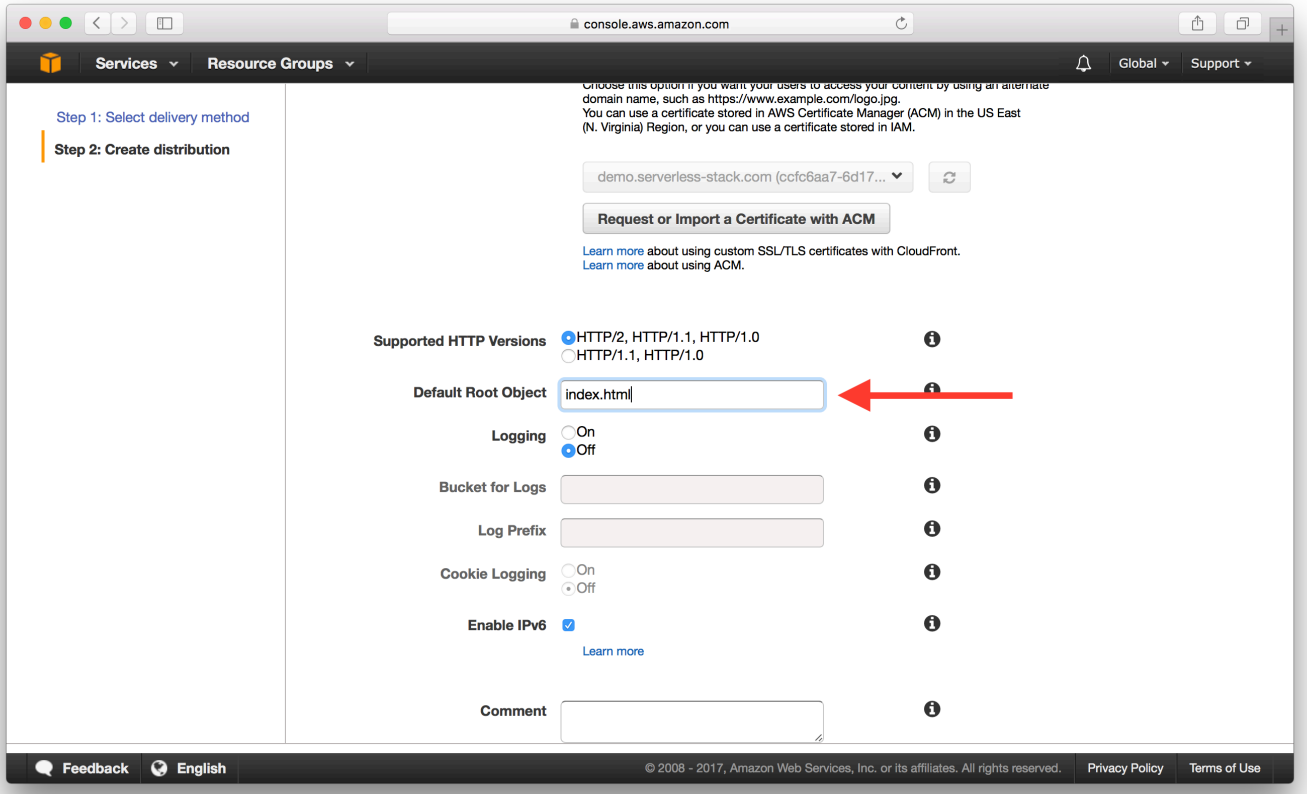
And paste that URL in the **Origin Domain Name** field. In my case it is, `http://notes-app-client.s3-website-us-east-1.amazonaws.com`.



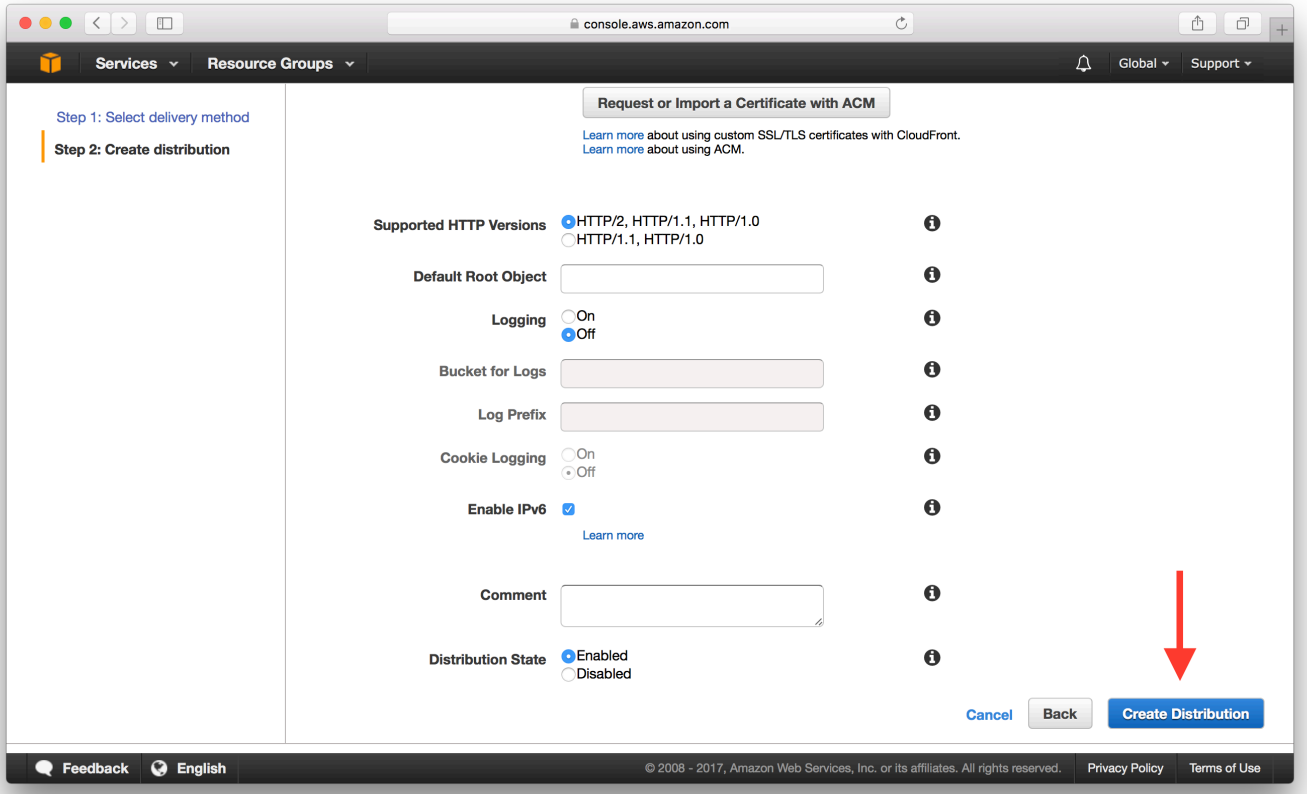
And now scroll down the form and switch **Compress Objects Automatically** to **Yes**. This will automatically Gzip compress the files that can be compressed and speed up the delivery of our app.



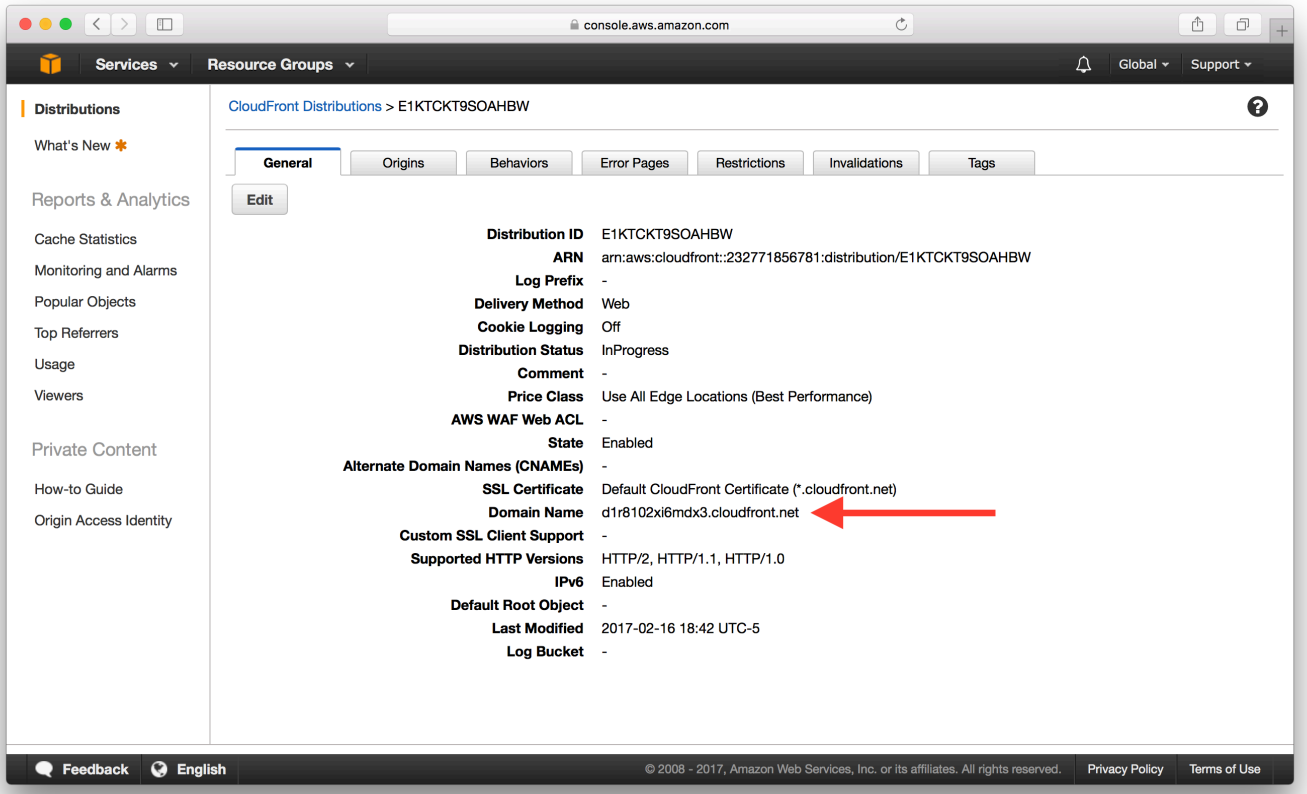
Next, scroll down a bit further to set the **Default Root Object** to `index.html`.



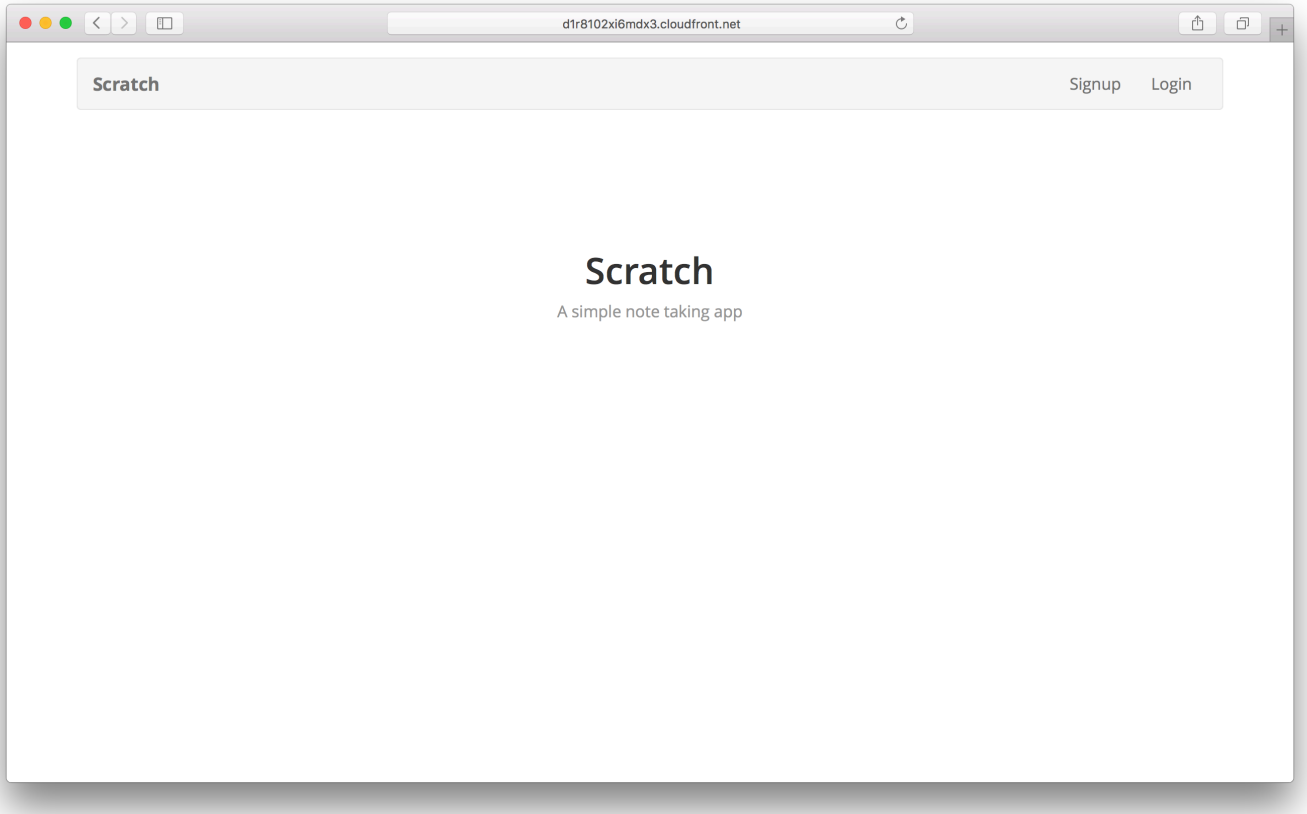
And finally, hit **Create Distribution**.



It takes AWS a little while to create a distribution. But once it is complete you can find your CloudFront Distribution by clicking on your newly created distribution from the list and looking up its domain name.



And if you navigate over to that in your browser, you should see your app live.

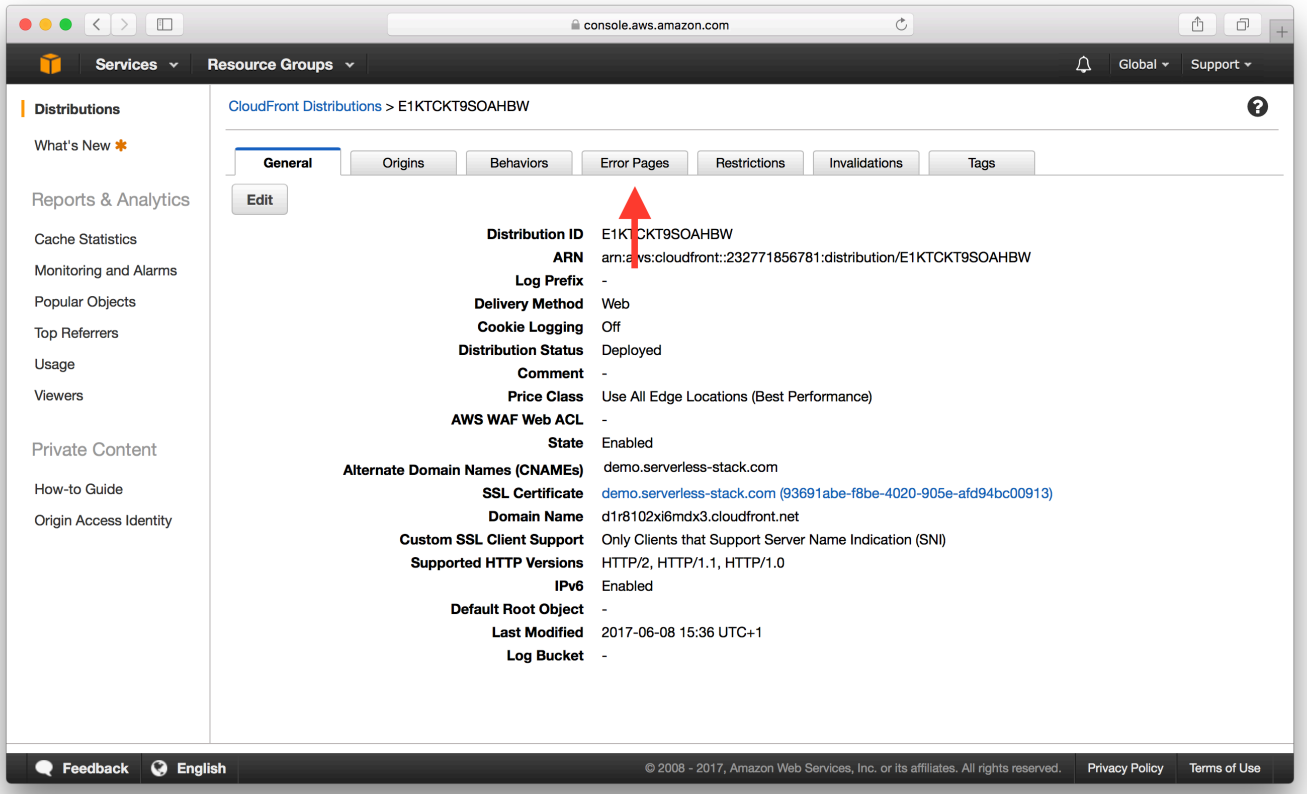


Now before we move on there is one last thing we need to do. Currently, our static website returns our `index.html` as the error page. We set this up back in the chapter where we created our S3 bucket. However, it returns a HTTP status code of 404 when it does so. We want to return the `index.html` but since the routing is handled by React Router; it does not make sense that we return the 404 HTTP status code. One of the issues with this is that certain corporate firewalls and proxies tend to block 4xx and 5xx responses.

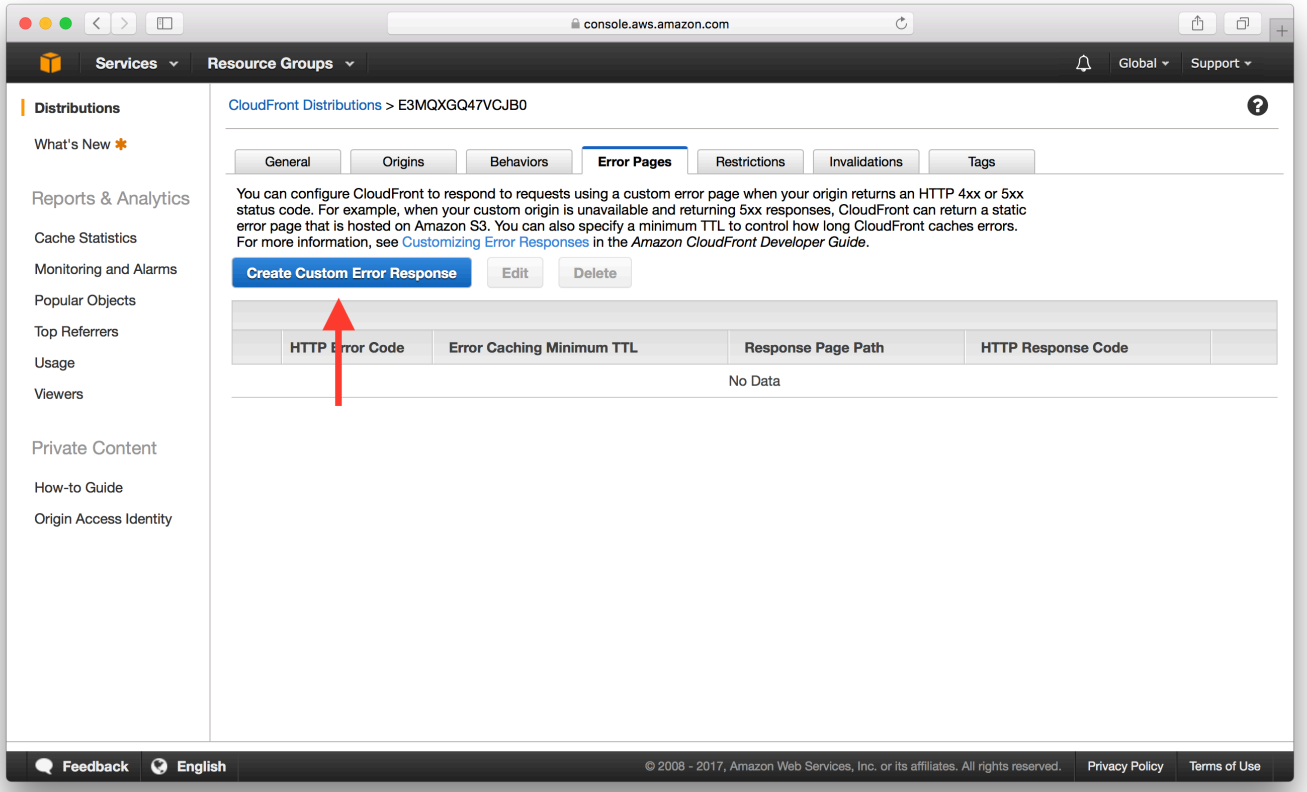
Custom Error Responses

So we are going to create a custom error response and return a 200 status code instead. The downside of this approach is that we are going to be returning 200 even for cases where we don't have a route in our React Router. Unfortunately, there isn't a way around this. This is because CloudFront or S3 are not aware of the routes in our React Router.

To set up a custom error response, head over to the **Error Pages** tab in our Distribution.

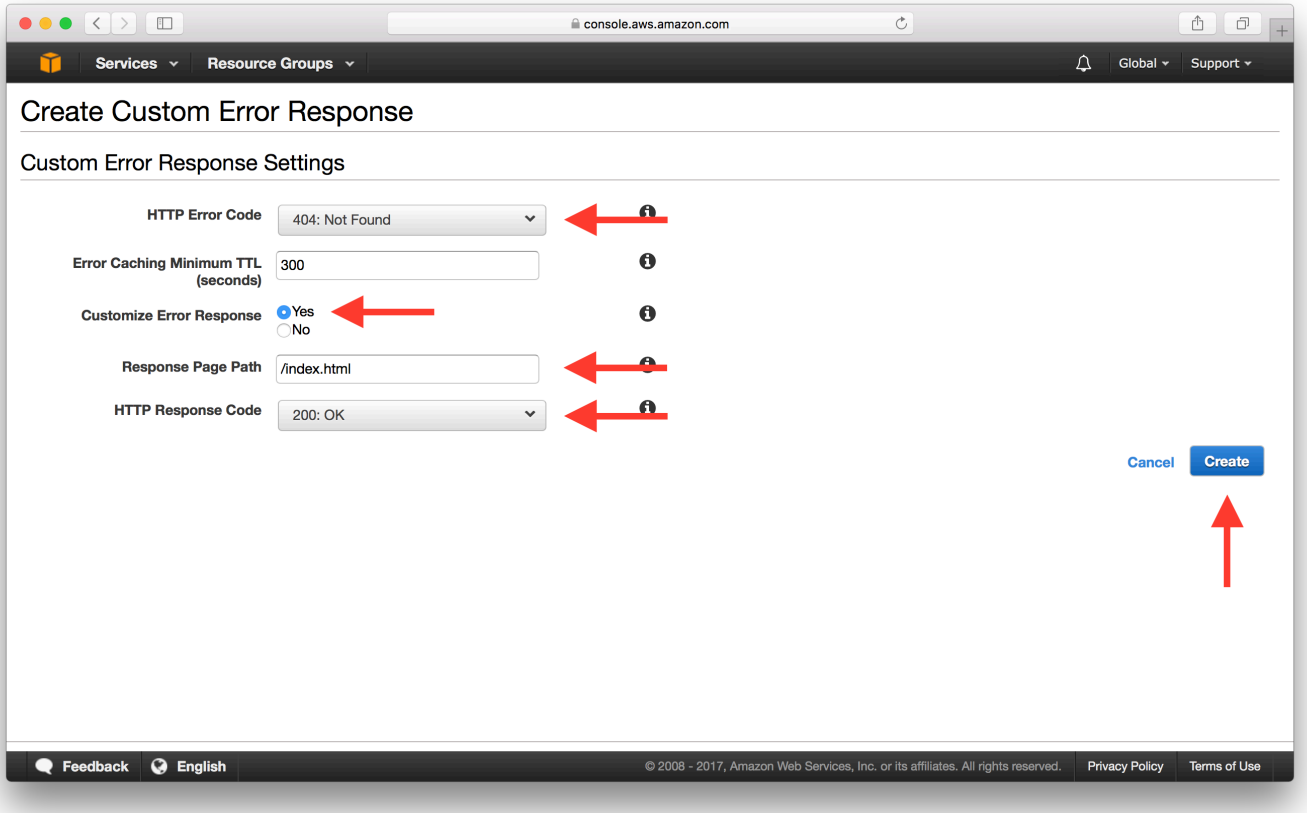


And select **Create Custom Error Response.**



Pick **404** for the **HTTP Error Code** and select **Customize Error Response.**

Enter `/index.html` for the **Response Page Path** and **200** for the **HTTP Response Code**.



And hit **Create**. This is basically telling CloudFront to respond to any 404 responses from our S3 bucket with the `index.html` and a 200 status code. Creating a custom error response should take a couple of minutes to complete.

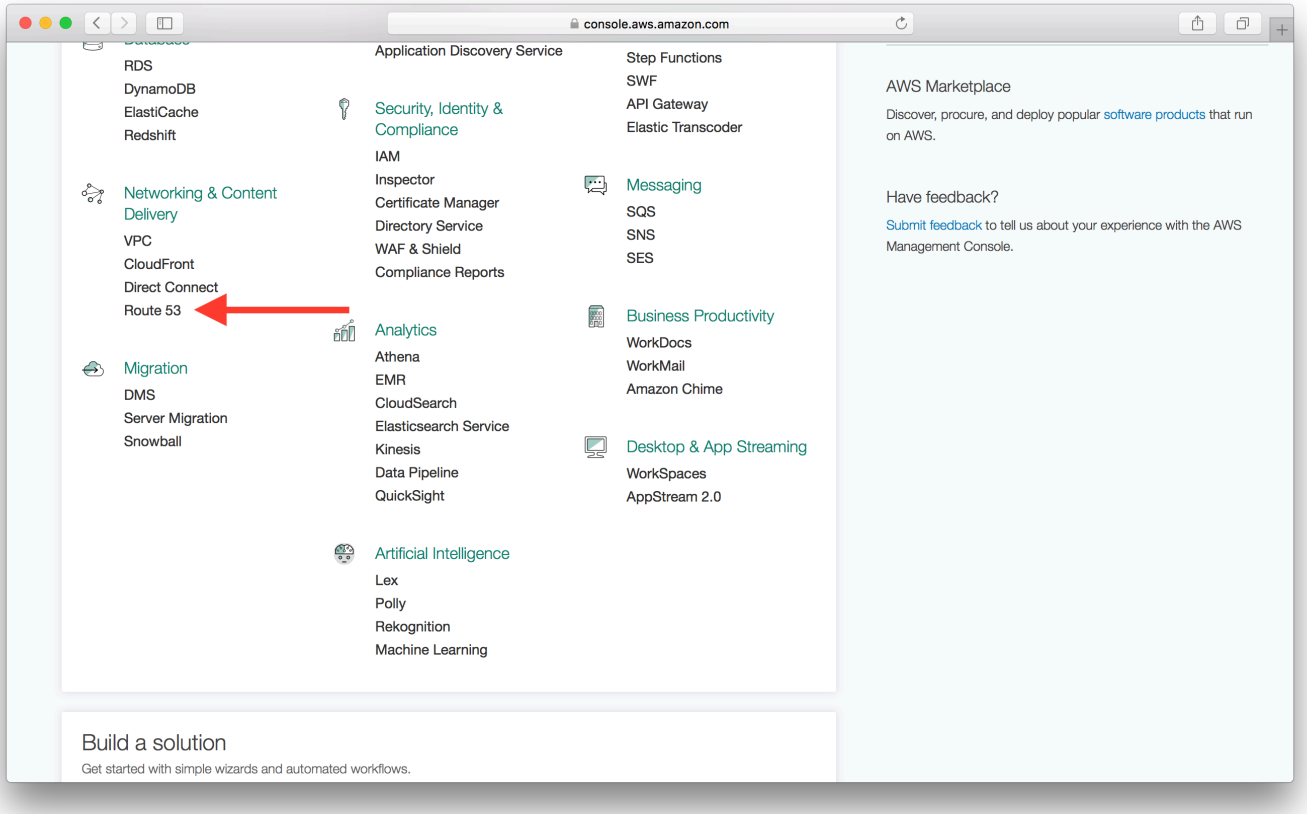
Next up, let's point our domain to our CloudFront Distribution.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/create-a-cloudfront-distribution/104)

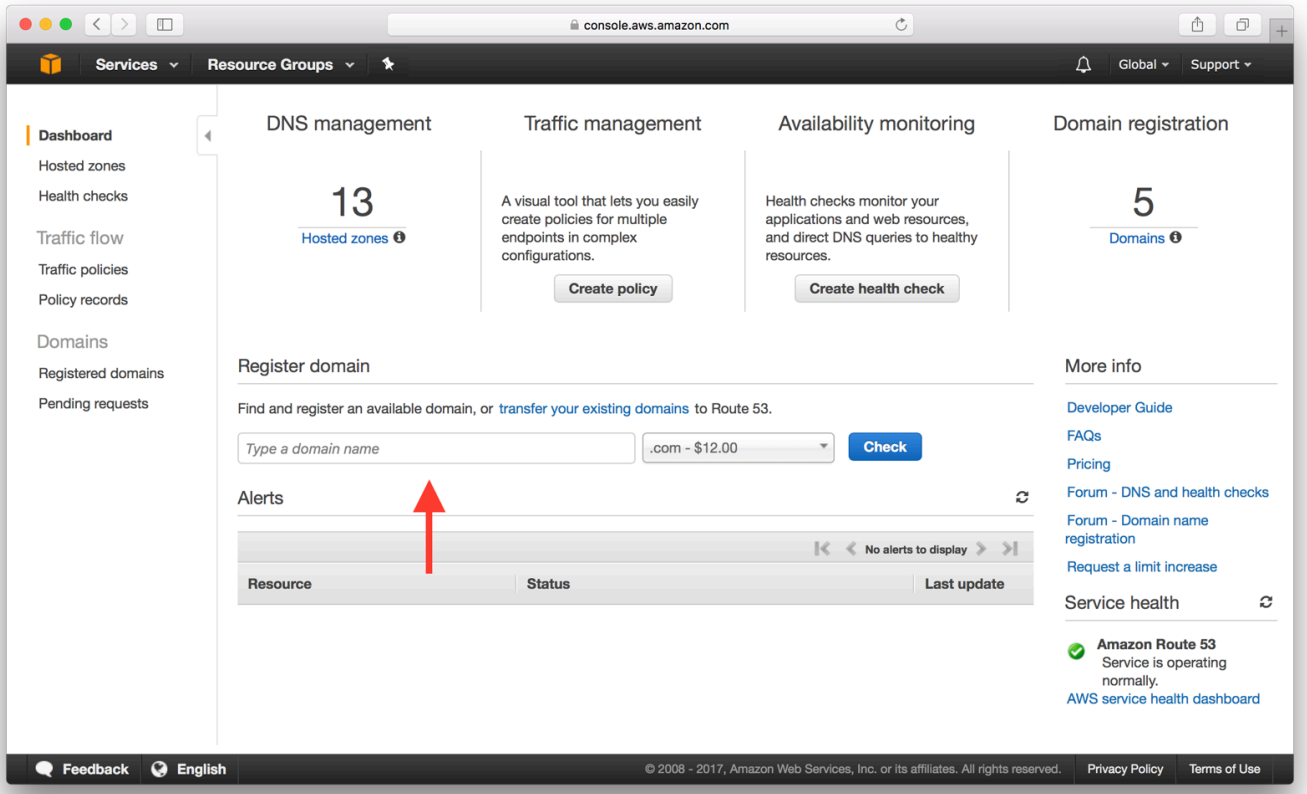
Set up Your Domain with CloudFront

Now that we have our CloudFront distribution live, let's set up our domain with it. You can purchase a domain right from the AWS Console (https://console.aws.amazon.com) by heading to the Route 53 section in the list of services.

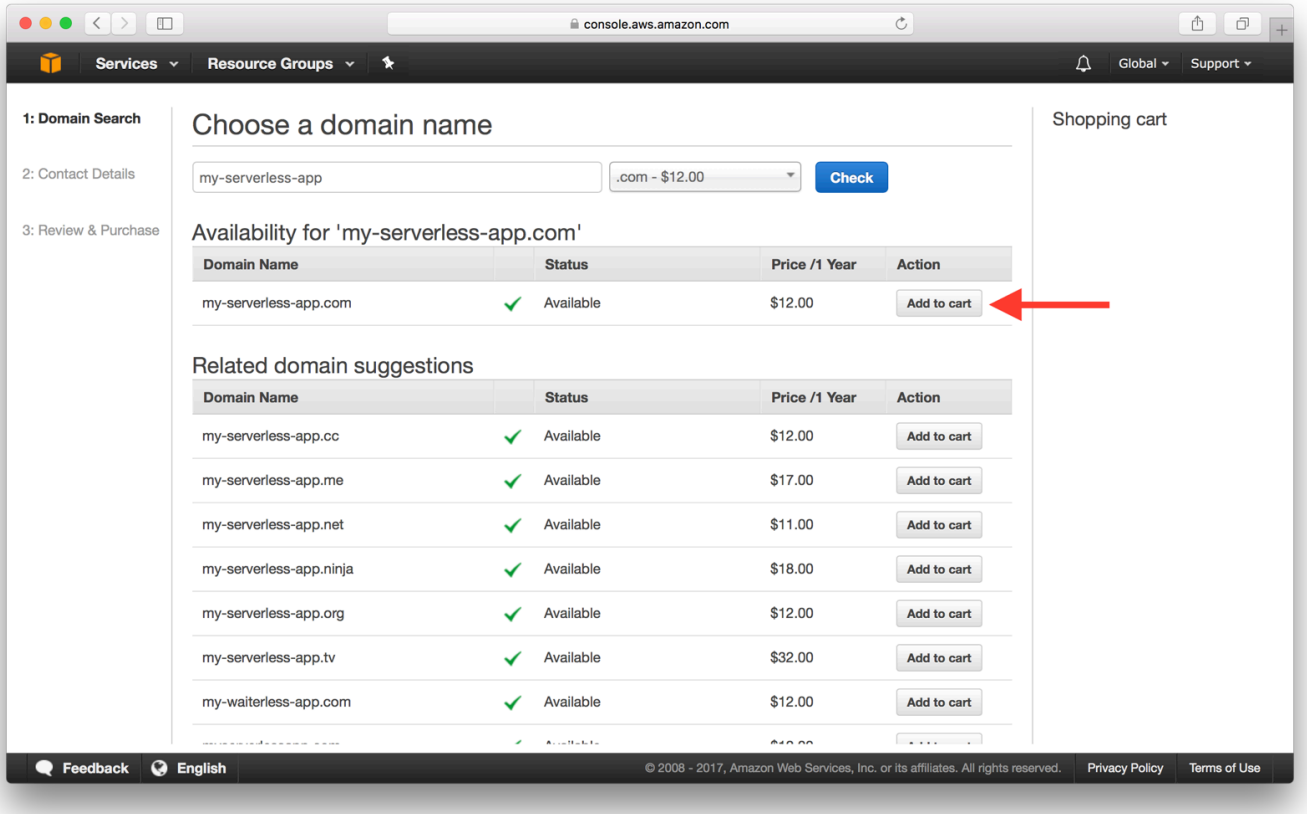


Purchase a Domain with Route 53

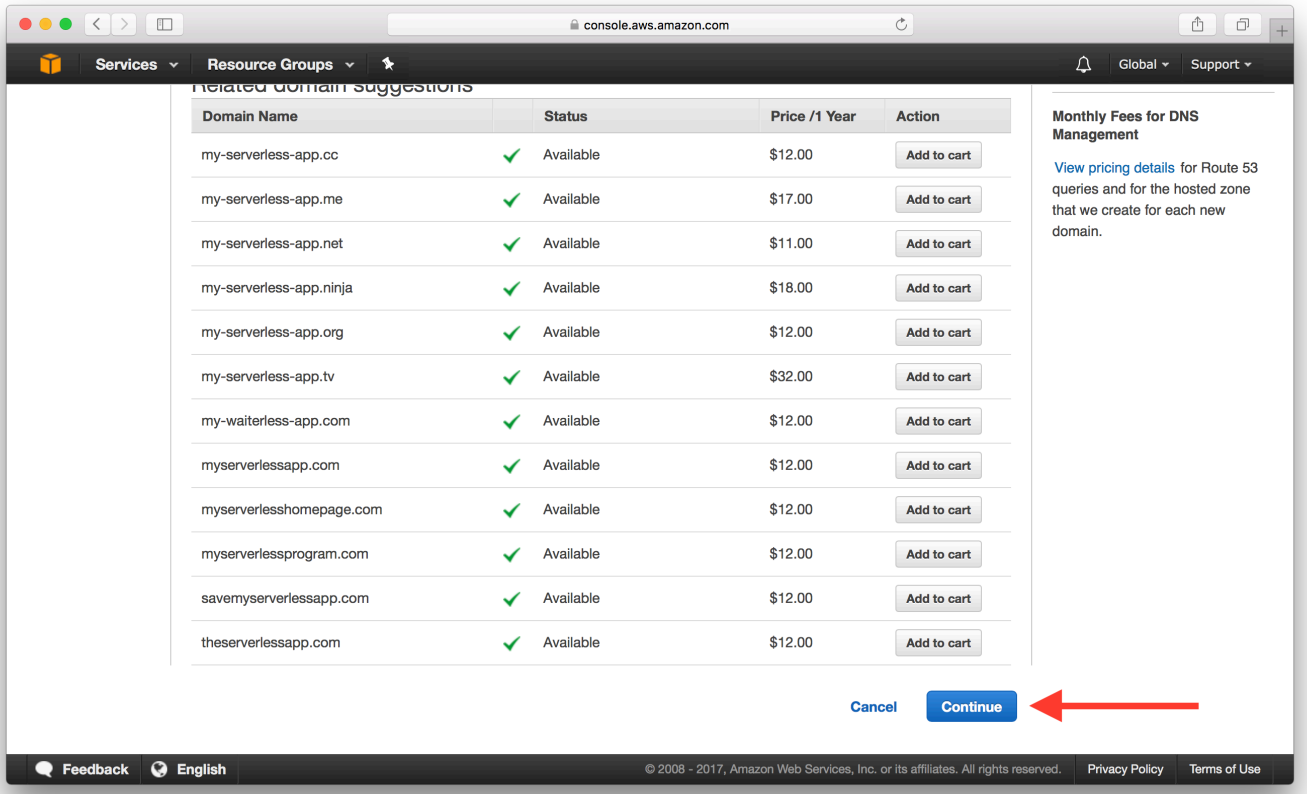
Type in your domain in the **Register domain** section and click **Check**.



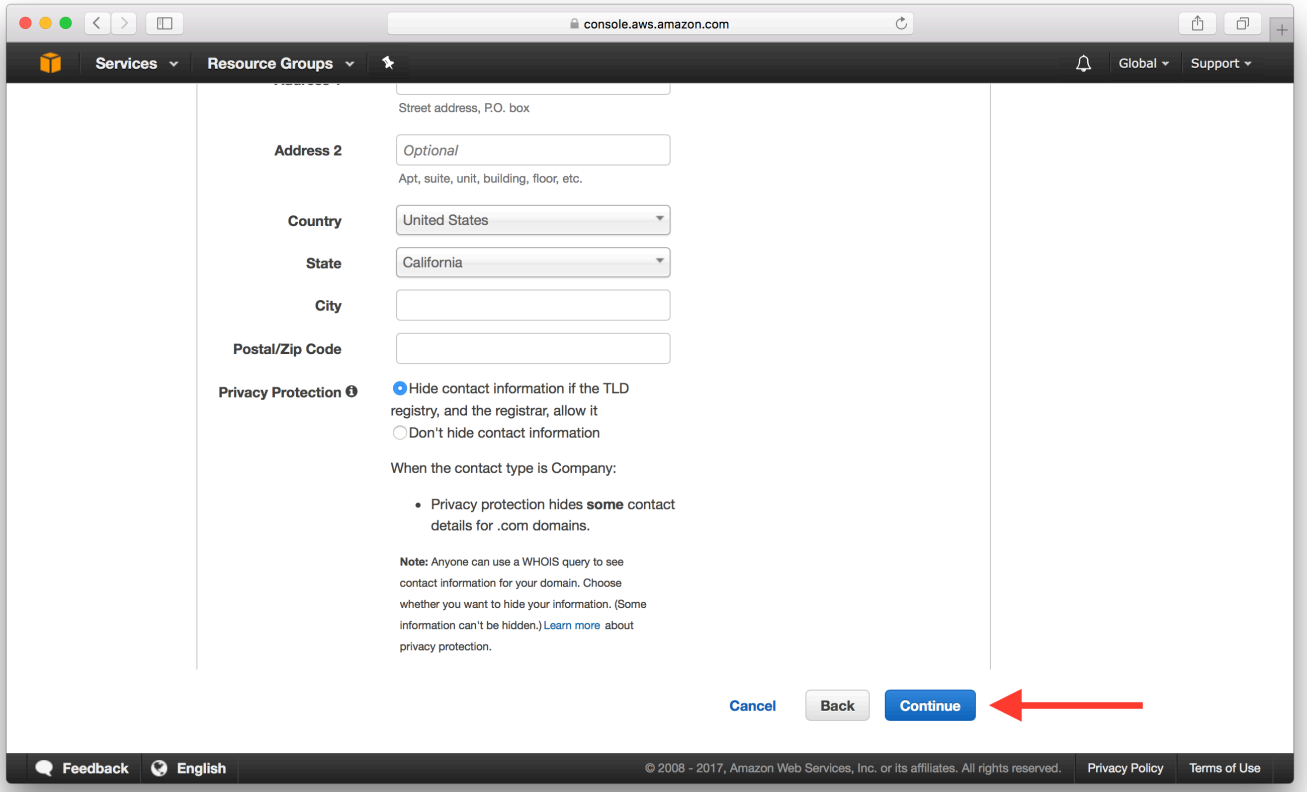
After checking its availability, click **Add to cart**.



And hit **Continue** at the bottom of the page.

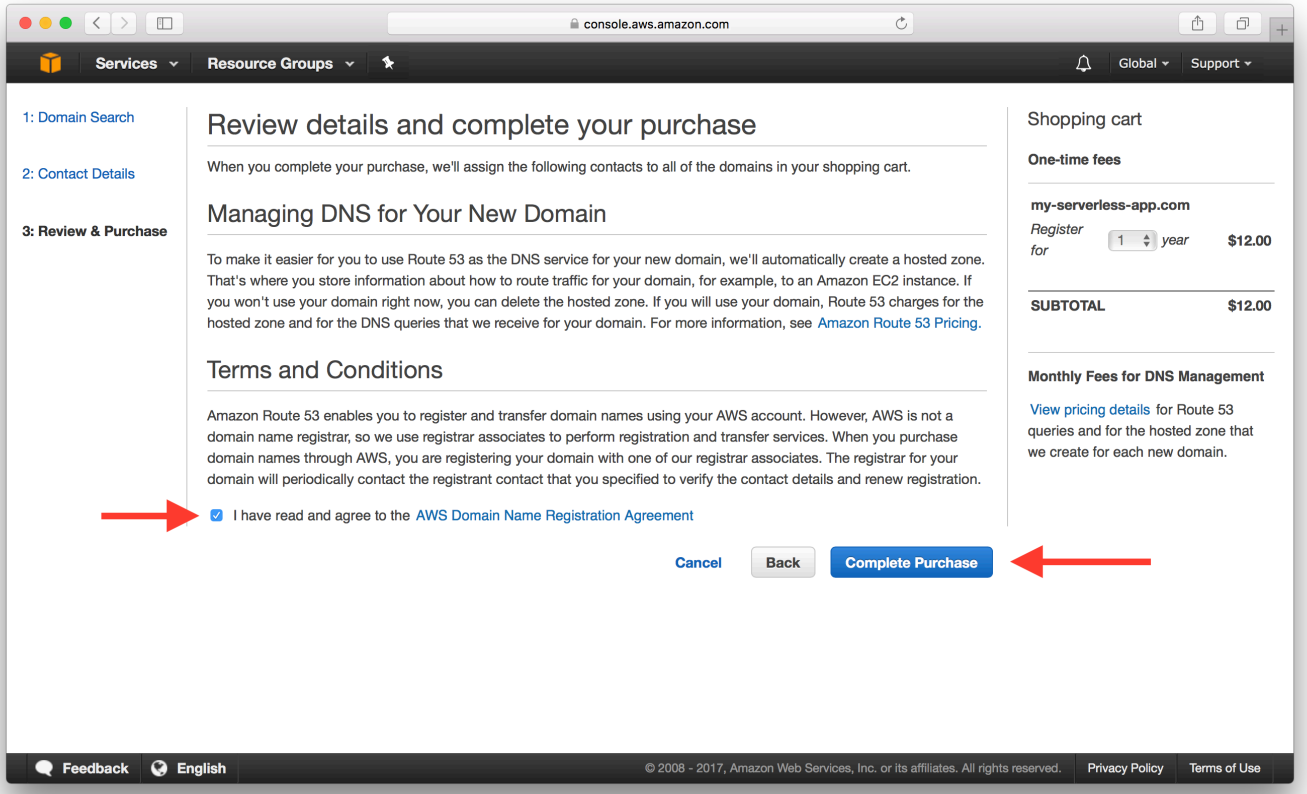


Fill in your contact details and hit **Continue** once again.



Finally, review your details and confirm the purchase by hitting **Complete**

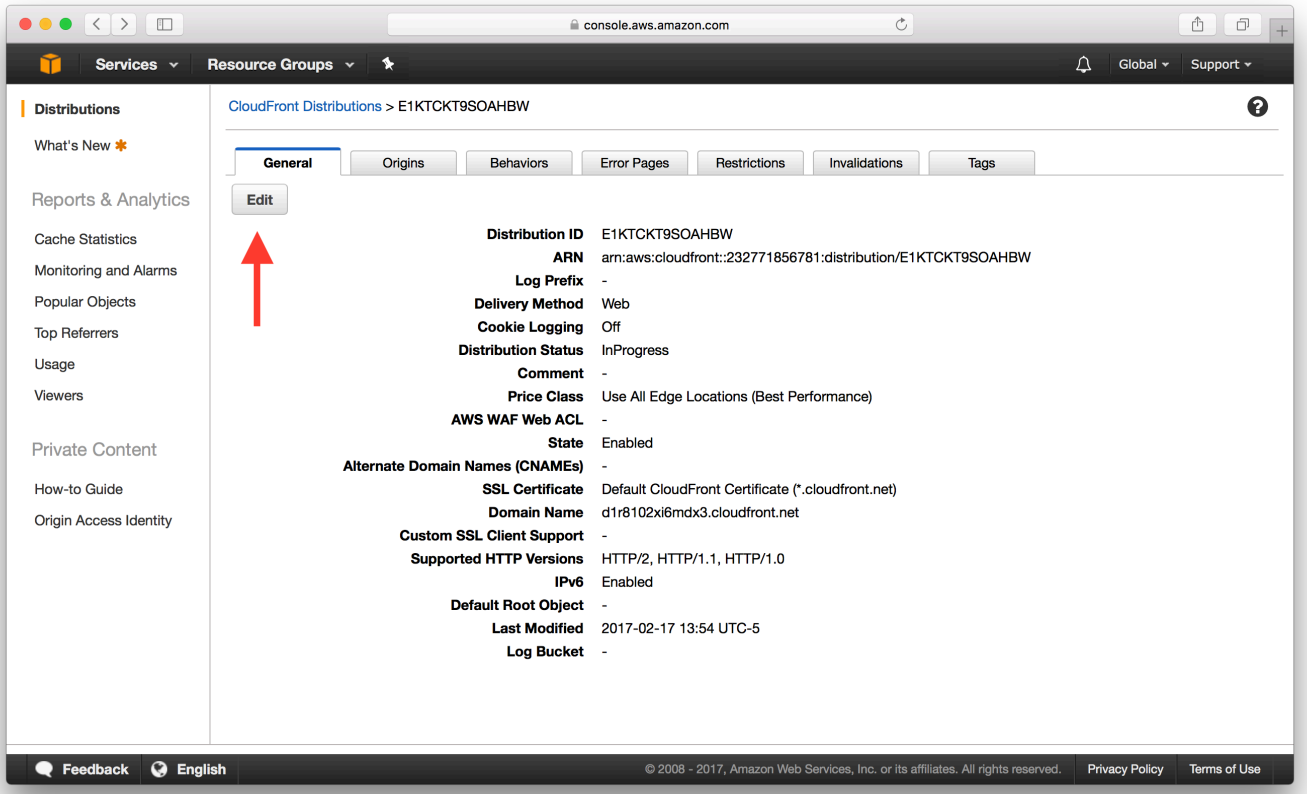
Purchase.



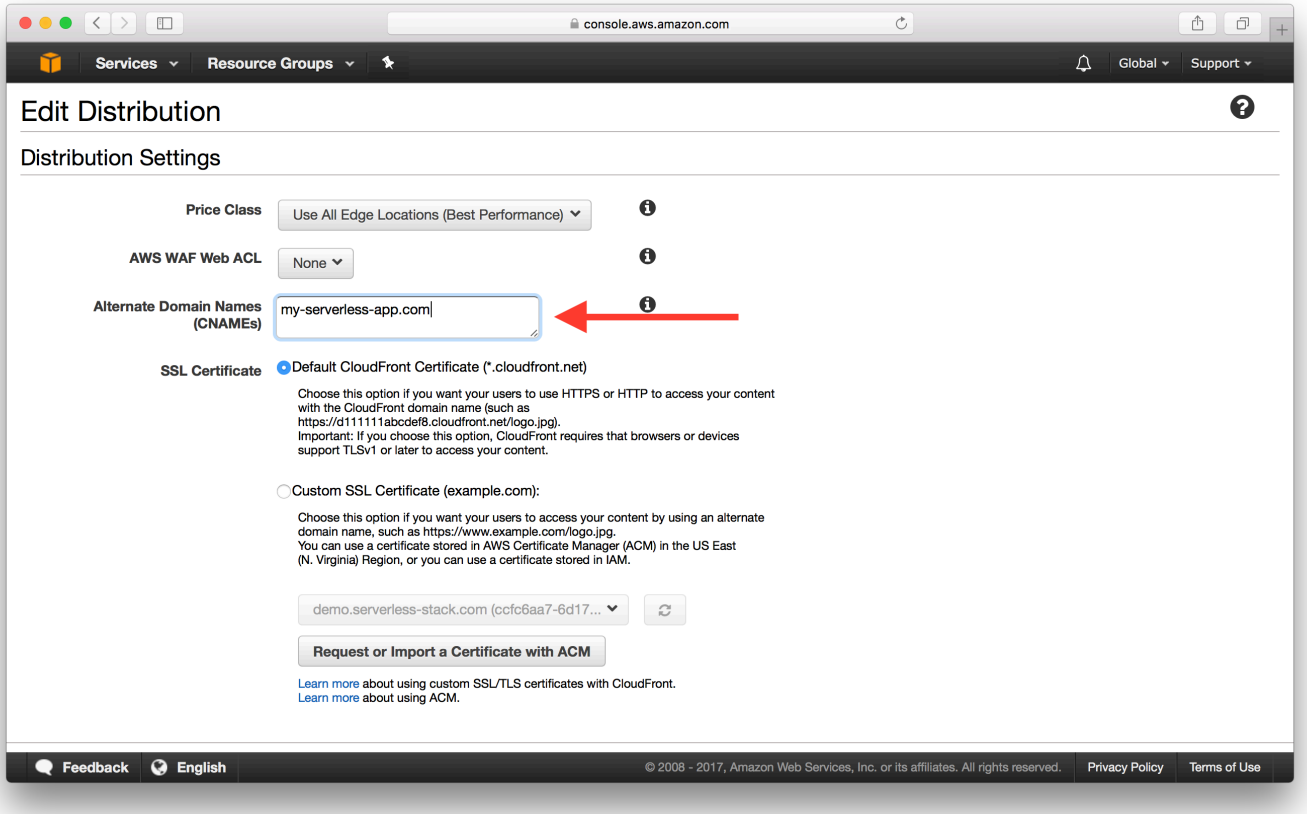
Next, we'll add an alternate domain name for our CloudFront Distribution.

Add Alternate Domain for CloudFront Distribution

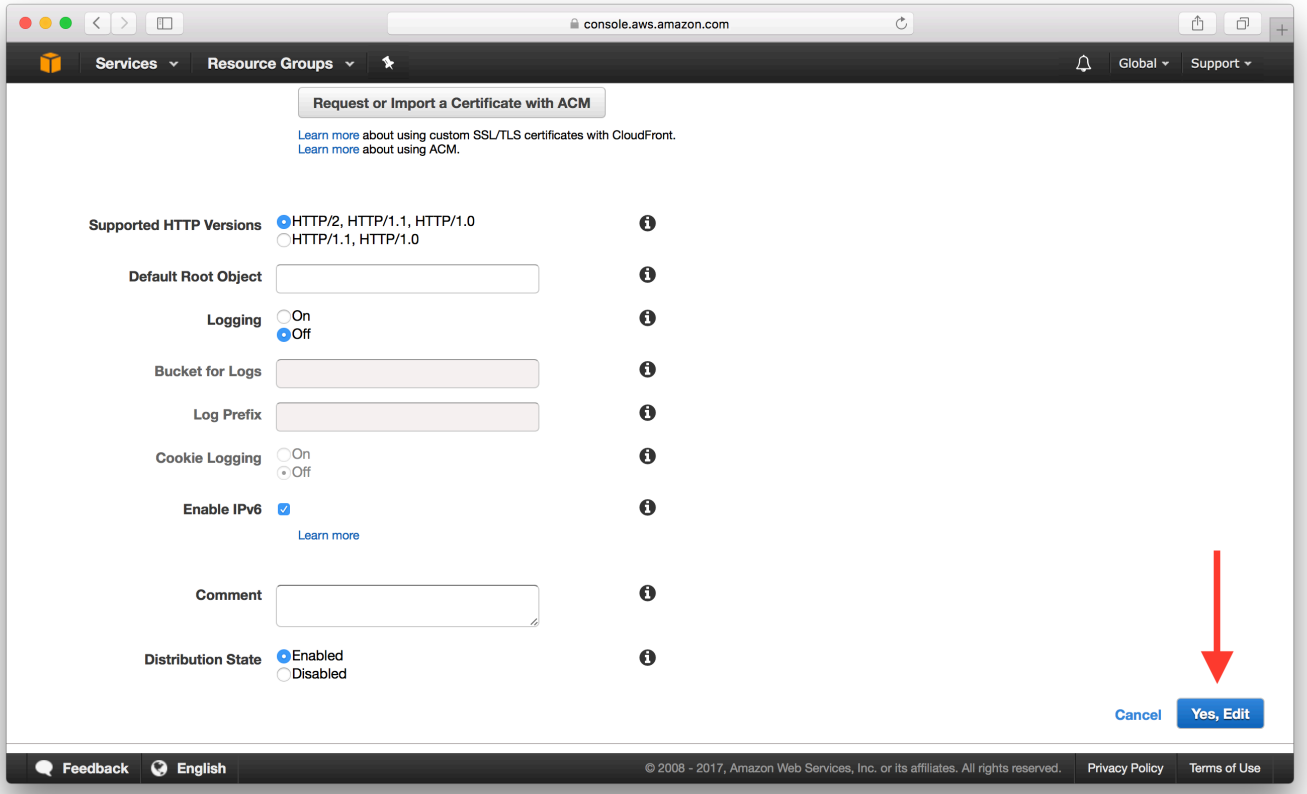
Head over to the details of your CloudFront Distribution and hit **Edit**.



And type in your new domain name in the **Alternate Domain Names (CNAMEs)** field.



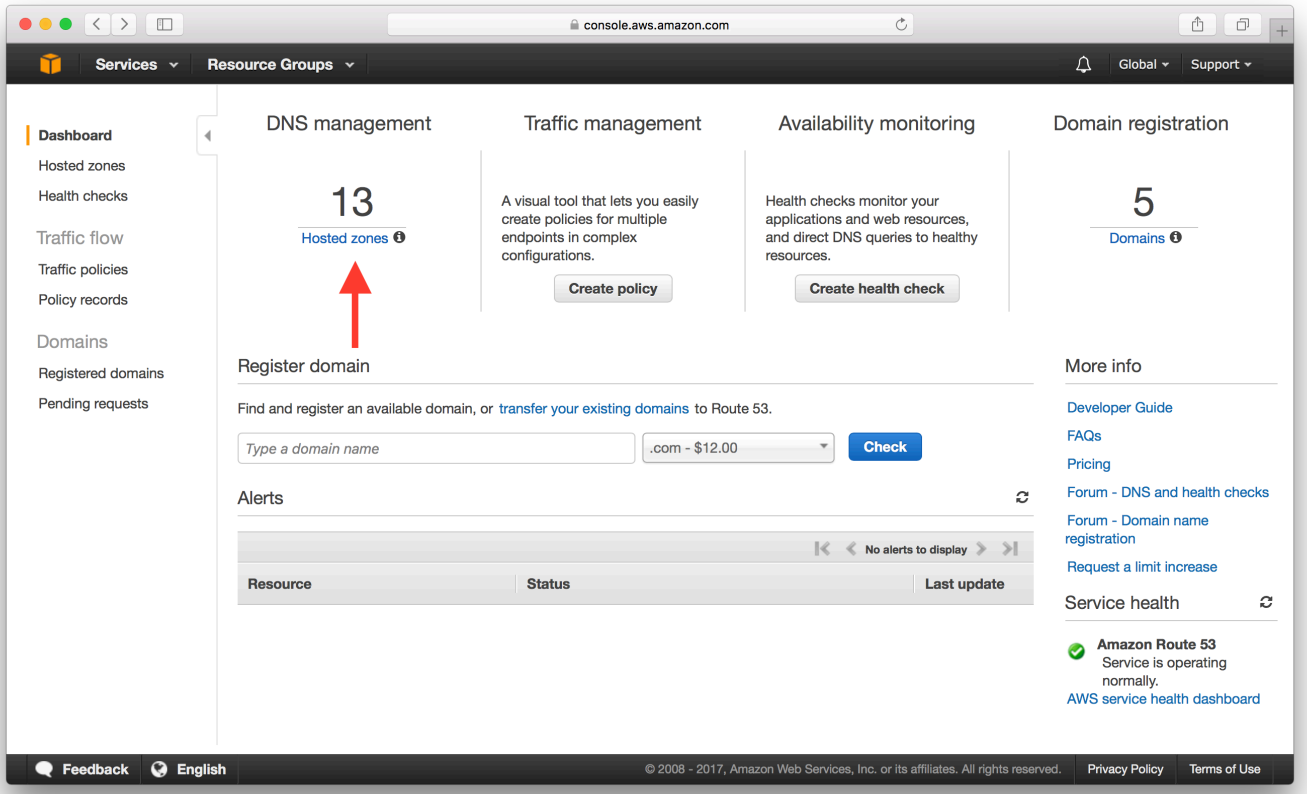
Scroll down and hit **Yes, Edit** to save the changes.



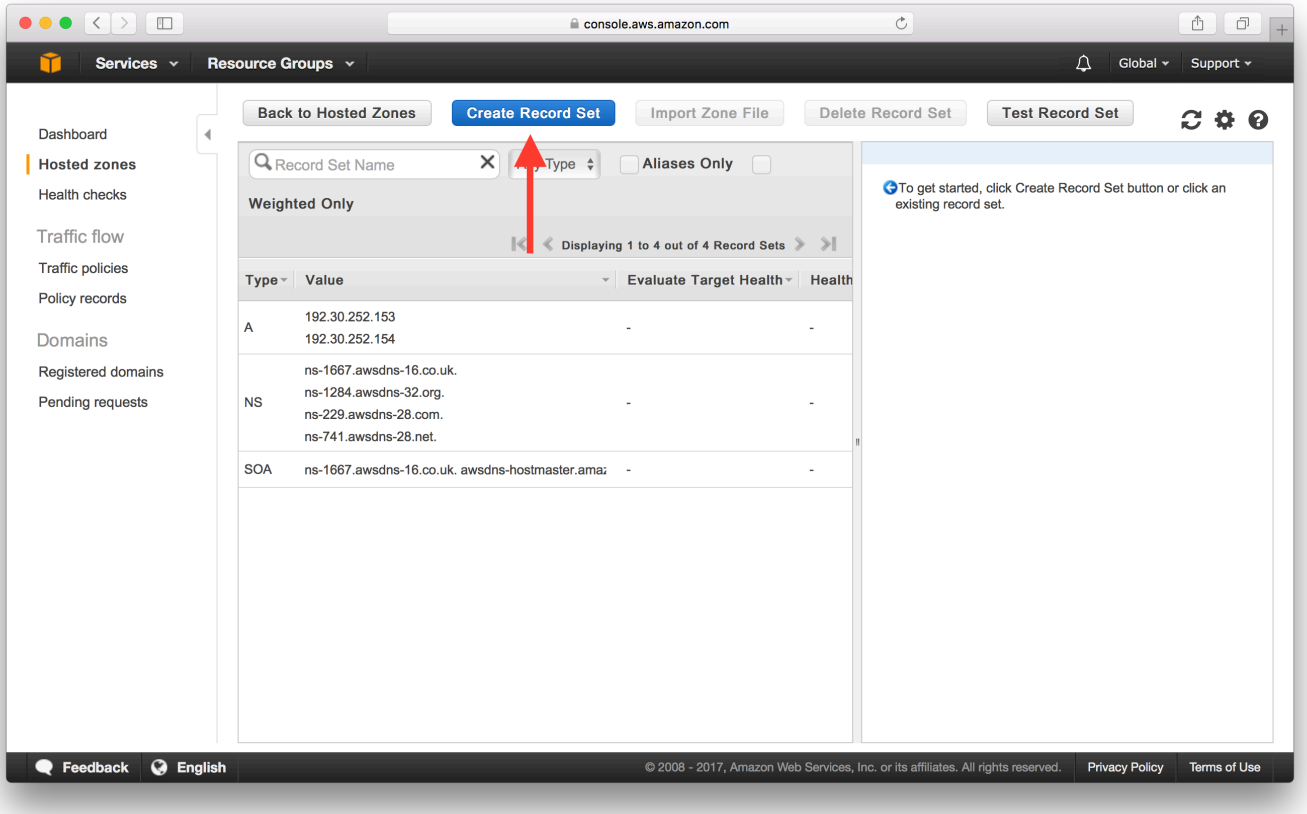
Next, let's point our domain to the CloudFront Distribution.

Point Domain to CloudFront Distribution

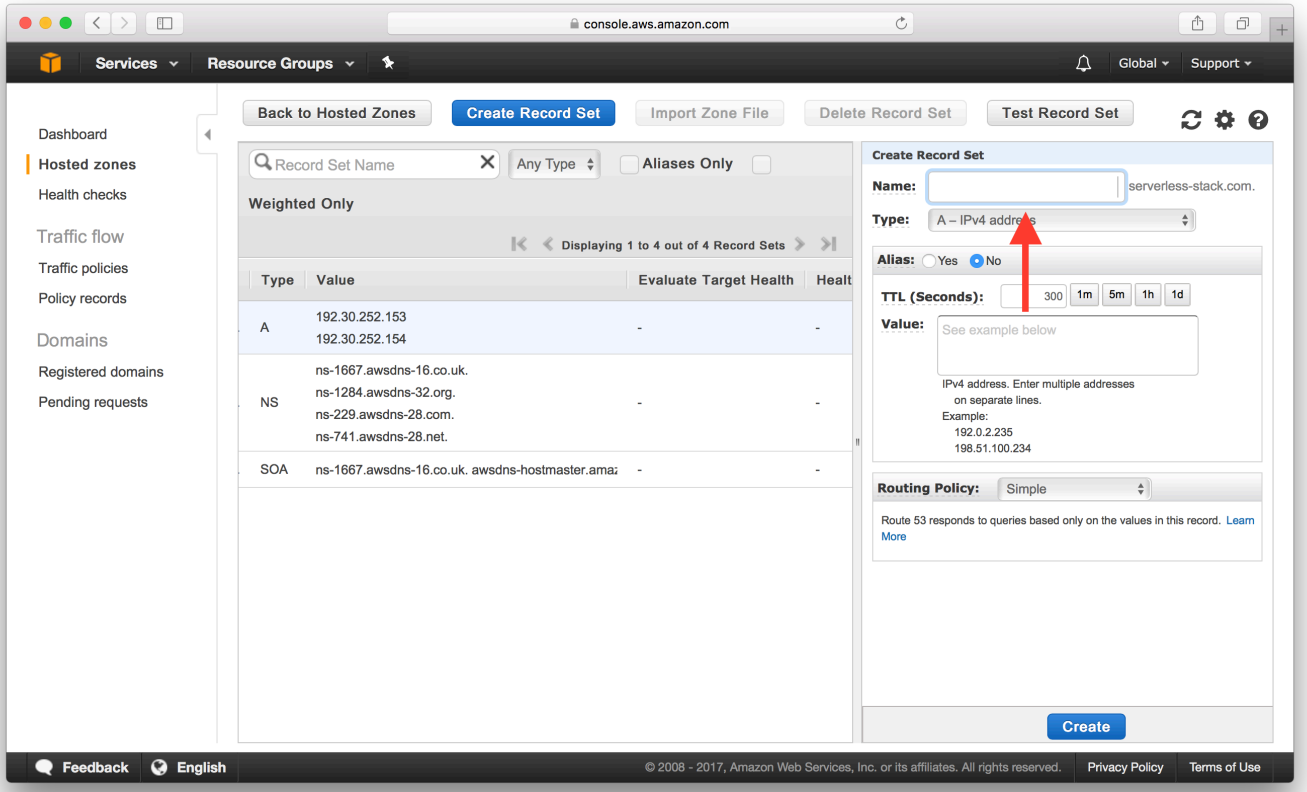
Head back into Route 53 and hit the **Hosted Zones** button. If you don't have an existing **Hosted Zone**, you'll need to create one by adding the **Domain Name** and selecting **Public Hosted Zone** as the **Type**.



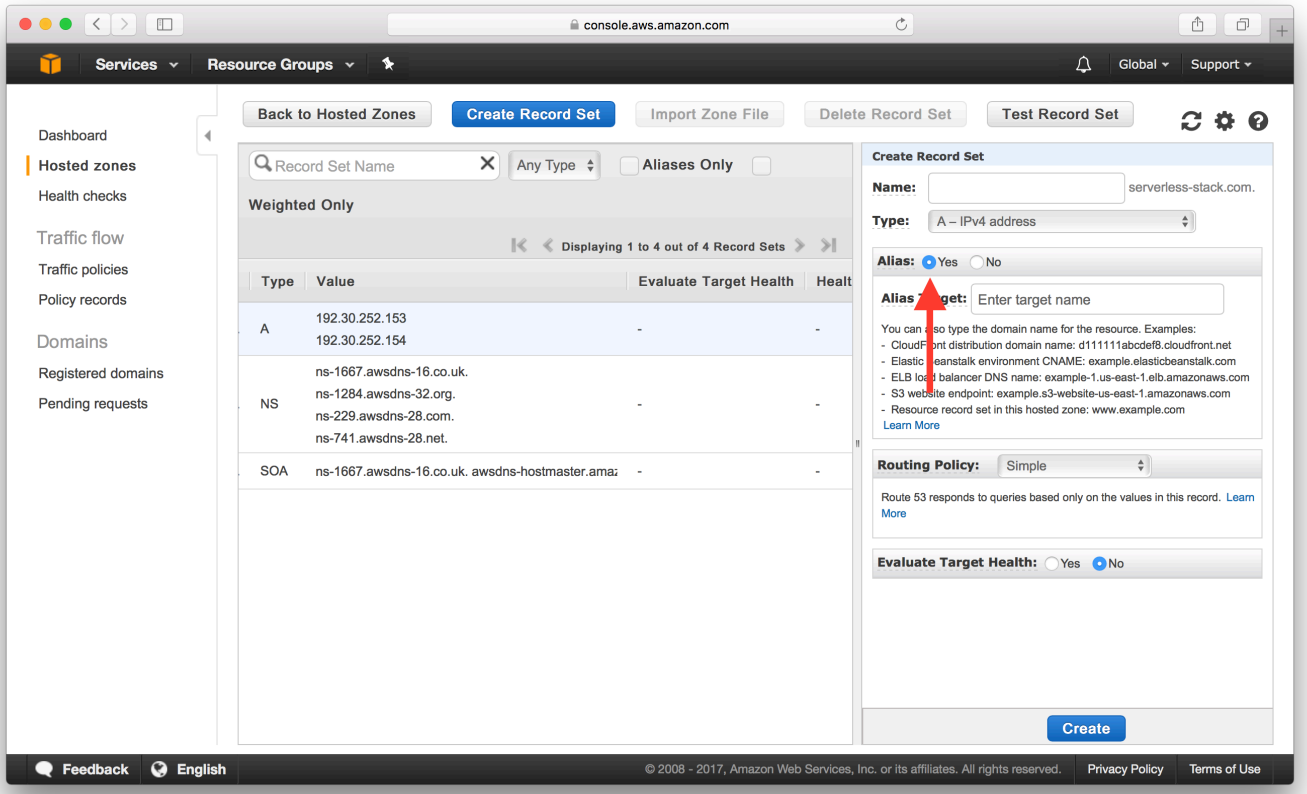
Select your domain from the list and hit **Create Record Set** in the details screen.



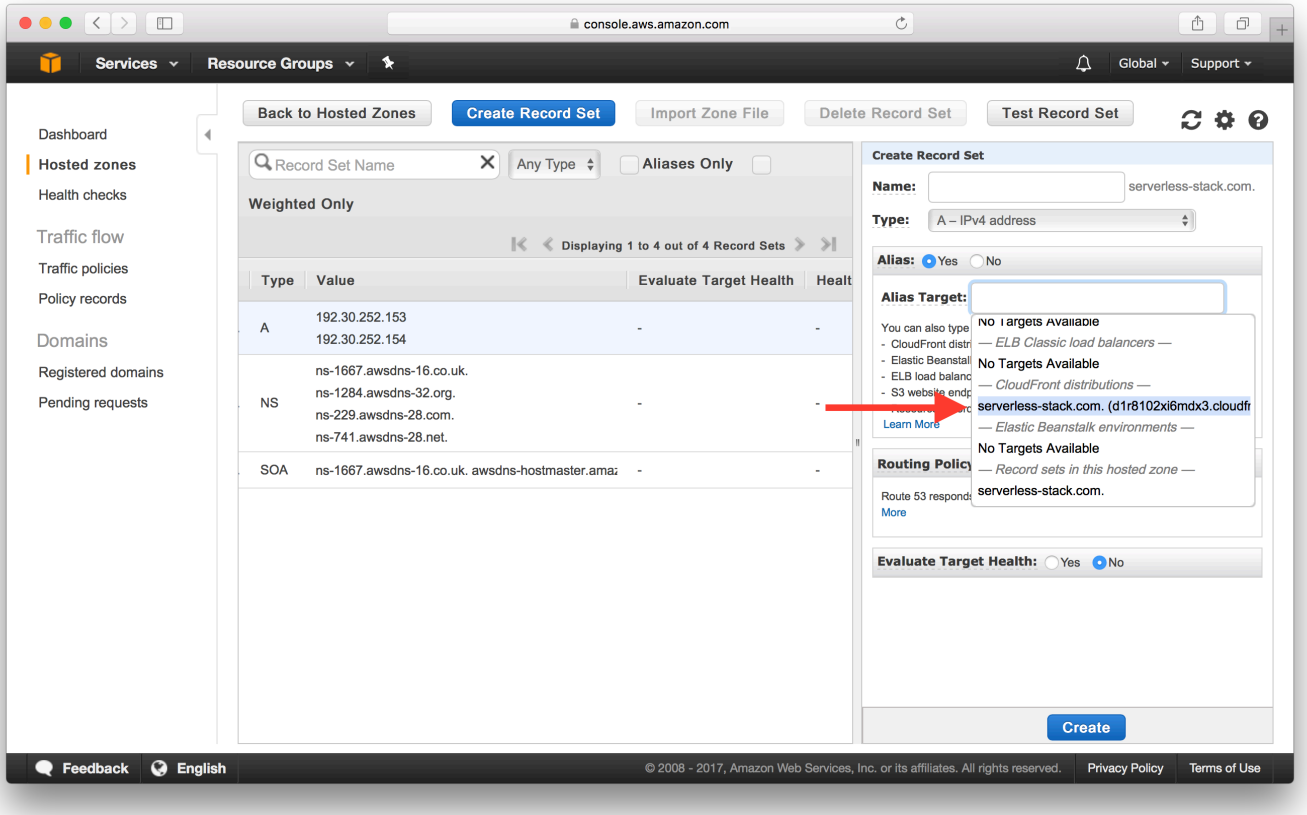
Leave the **Name** field empty since we are going to point our bare domain (without the www.) to our CloudFront Distribution.



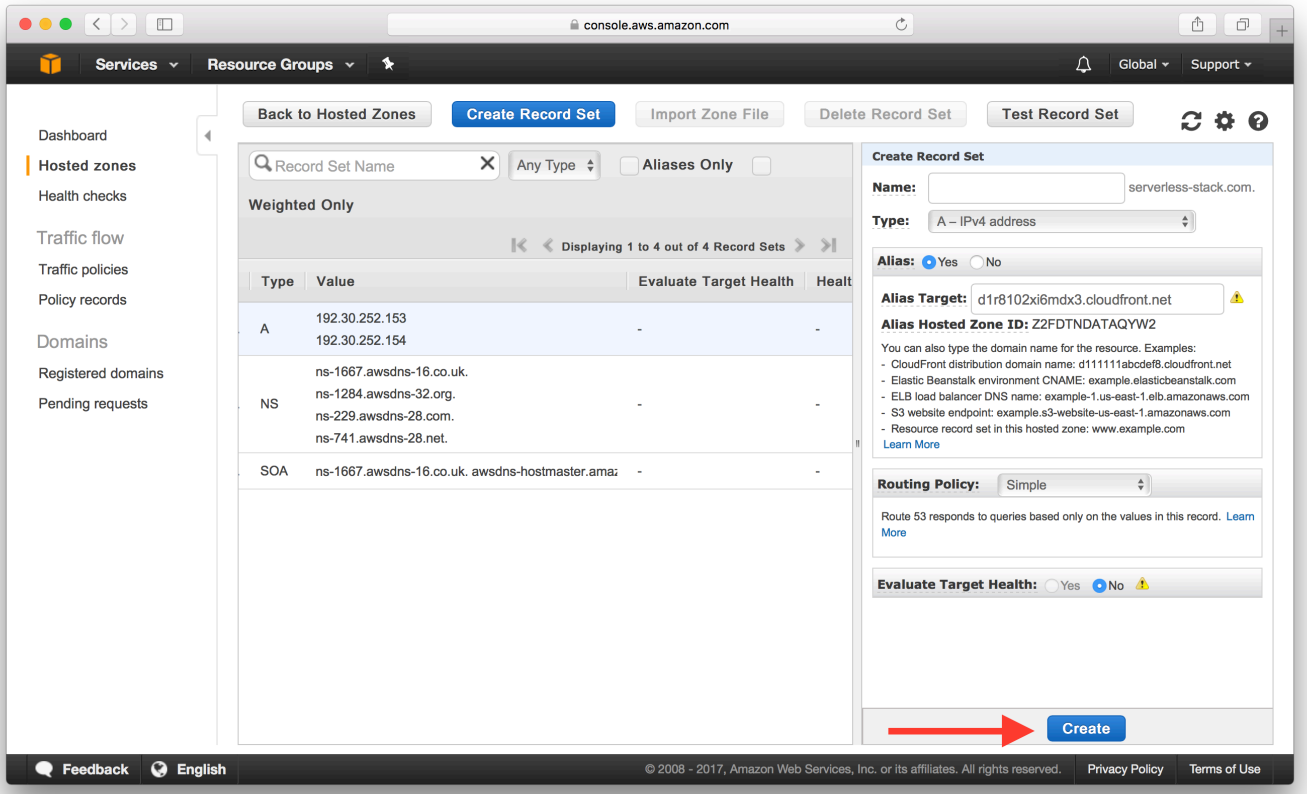
And select **Alias** as **Yes** since we are going to simply point this to our CloudFront domain.



In the **Alias Target** dropdown, select your CloudFront Distribution.



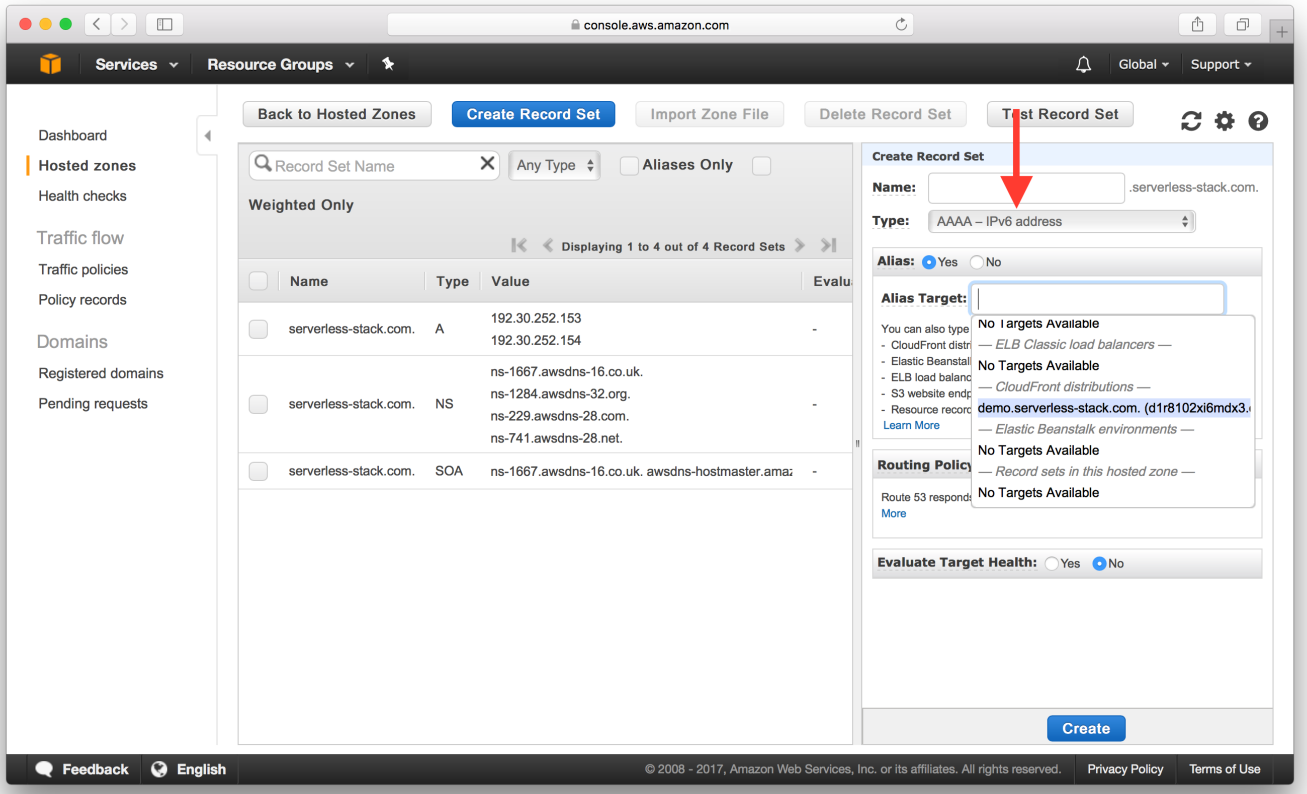
Finally, hit **Create** to add your new record set.



Add IPv6 Support

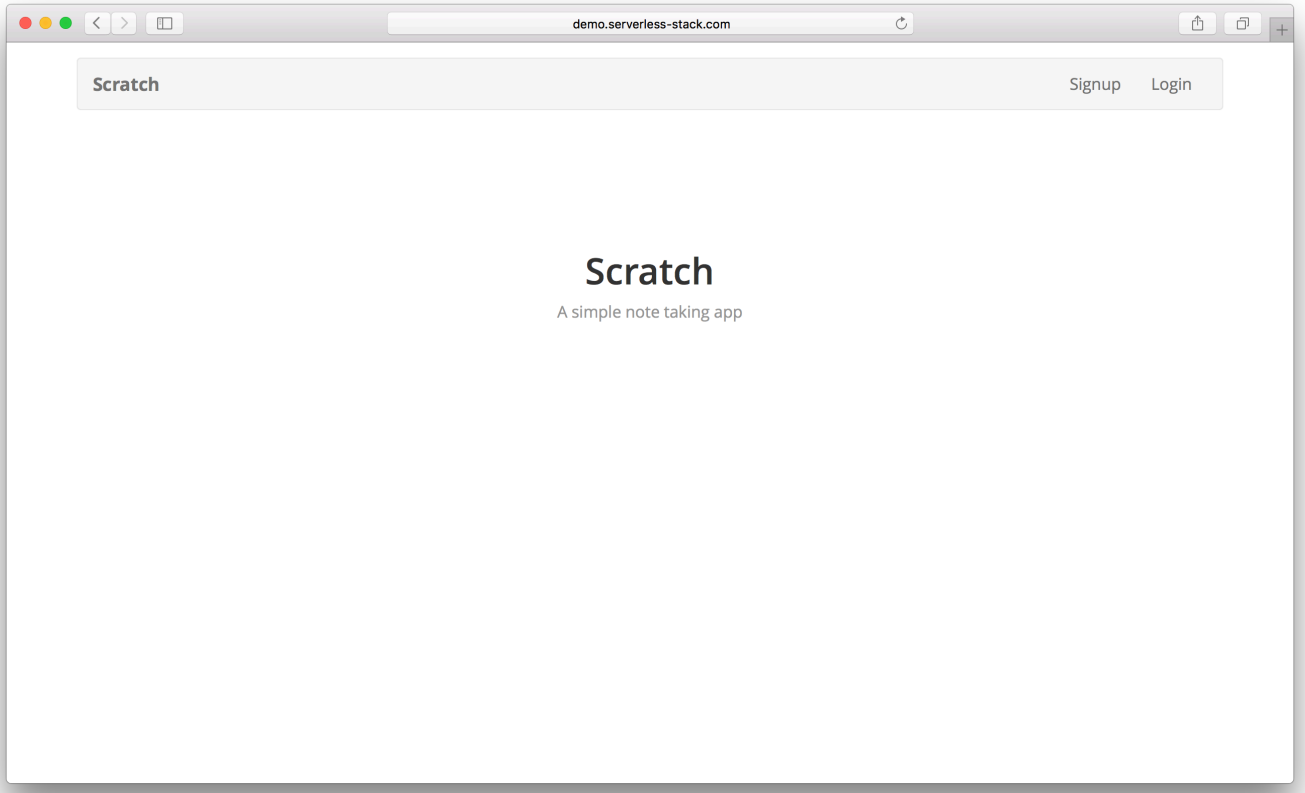
CloudFront Distributions have IPv6 enabled by default and this means that we need to create an AAAA record as well. It is set up exactly the same way as the Alias record.

Create a new Record Set with the exact settings as before, except make sure to pick **AAAA - IPv6 address** as the **Type**.



And hit **Create** to add your AAAA record set.

It can take around an hour to update the DNS records but once it's done, you should be able to access your app through your domain.



Next up, we'll take a quick look at ensuring that our www. domain also directs to our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/set-up-your-domain-with-cloudfront/149)

Set up WWW Domain Redirect

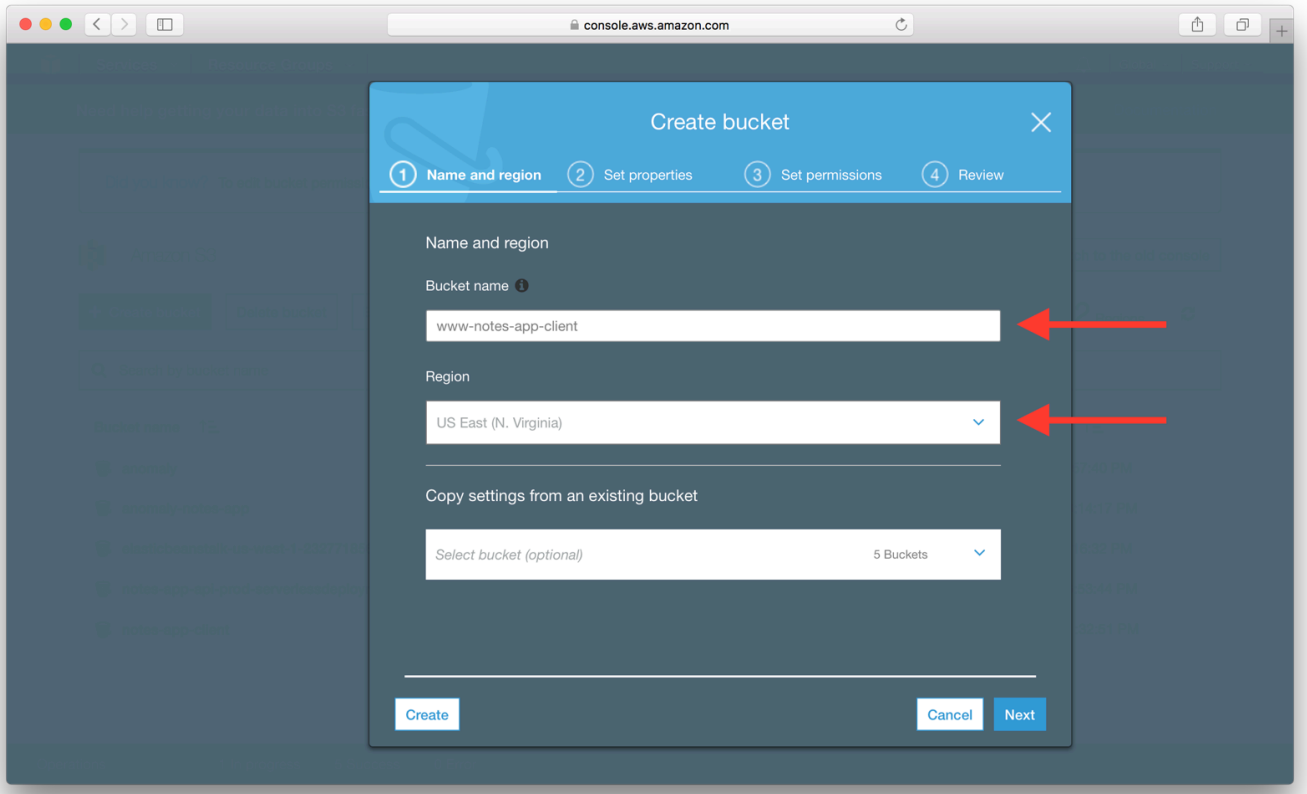
There's plenty of debate over the www vs non-www domains and while both sides have merit; we'll go over how to set up another domain (in this case the www) and redirect it to our original. The reason we do a redirect is to tell the search engines that we only want one version of our domain to appear in the search results. If you prefer having the www domain as the default simply swap this step with the last one where we created a bare domain (non-www).

To create a www version of our domain and have it redirect we are going to create a new S3 Bucket and a new CloudFront Distribution. This new S3 Bucket will simply respond with a redirect to our main domain using the redirection feature that S3 Buckets have.

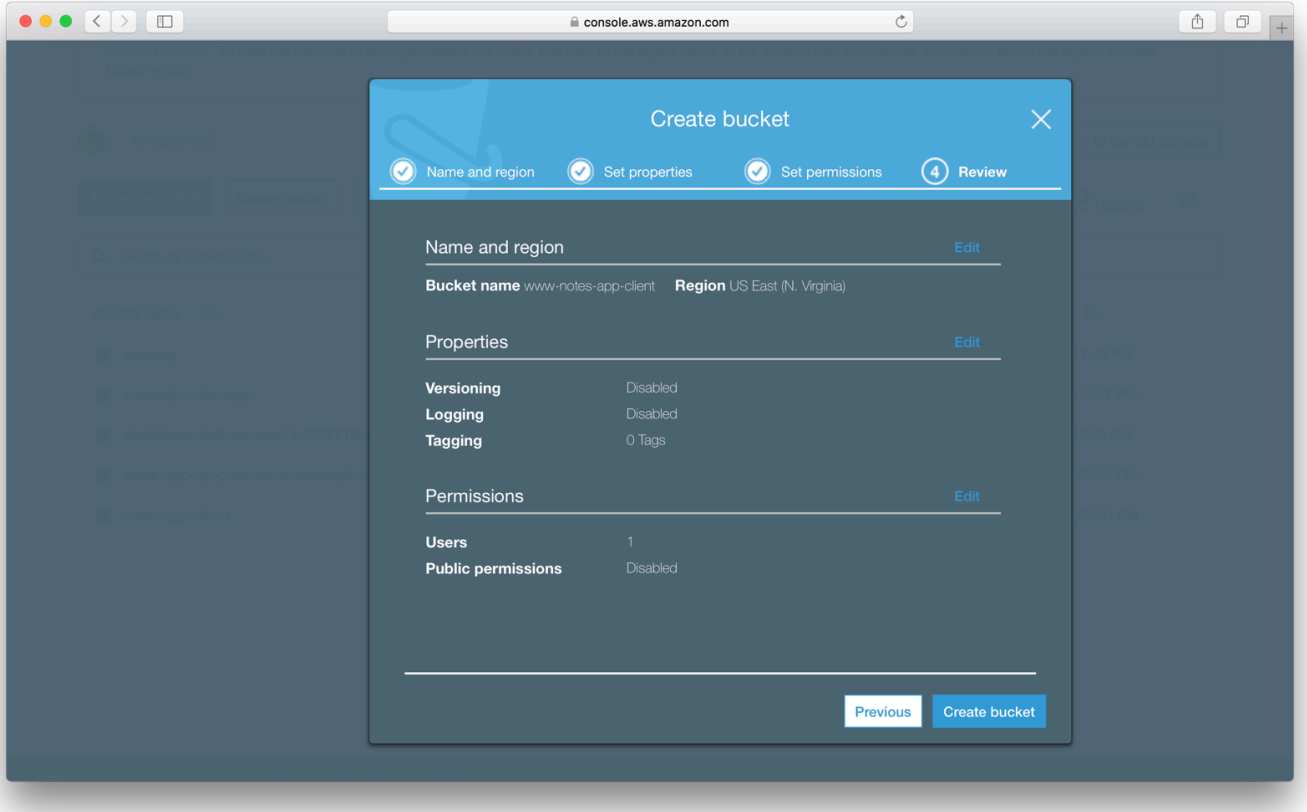
So let's start by creating a new S3 redirect Bucket for this.

Create S3 Redirect Bucket

Create a **new S3 Bucket** through the AWS Console (https://console.aws.amazon.com). The name doesn't really matter but it pick something that helps us distinguish between the two. Again, remember that we need a separate S3 Bucket for this step and we cannot use the original one we had previously created.

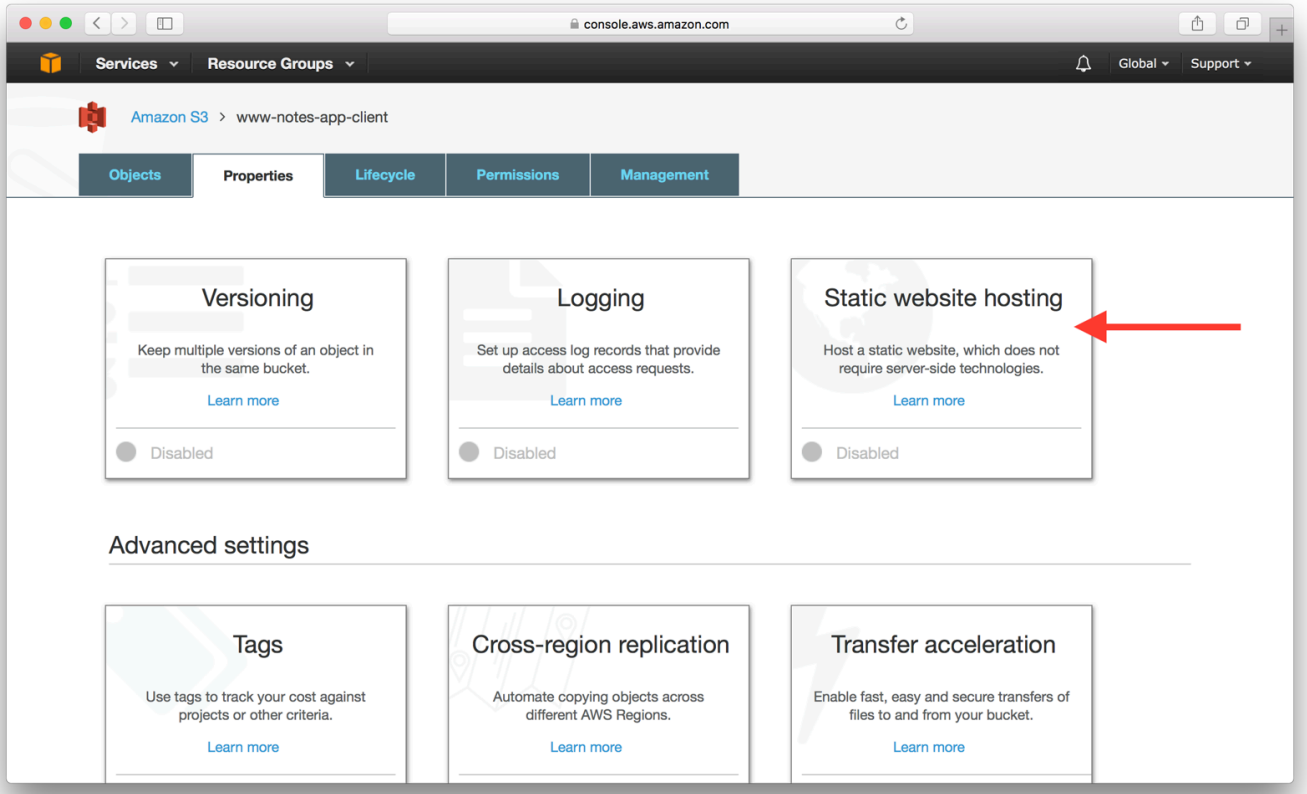


Next just follow through the steps and leave the defaults intact.



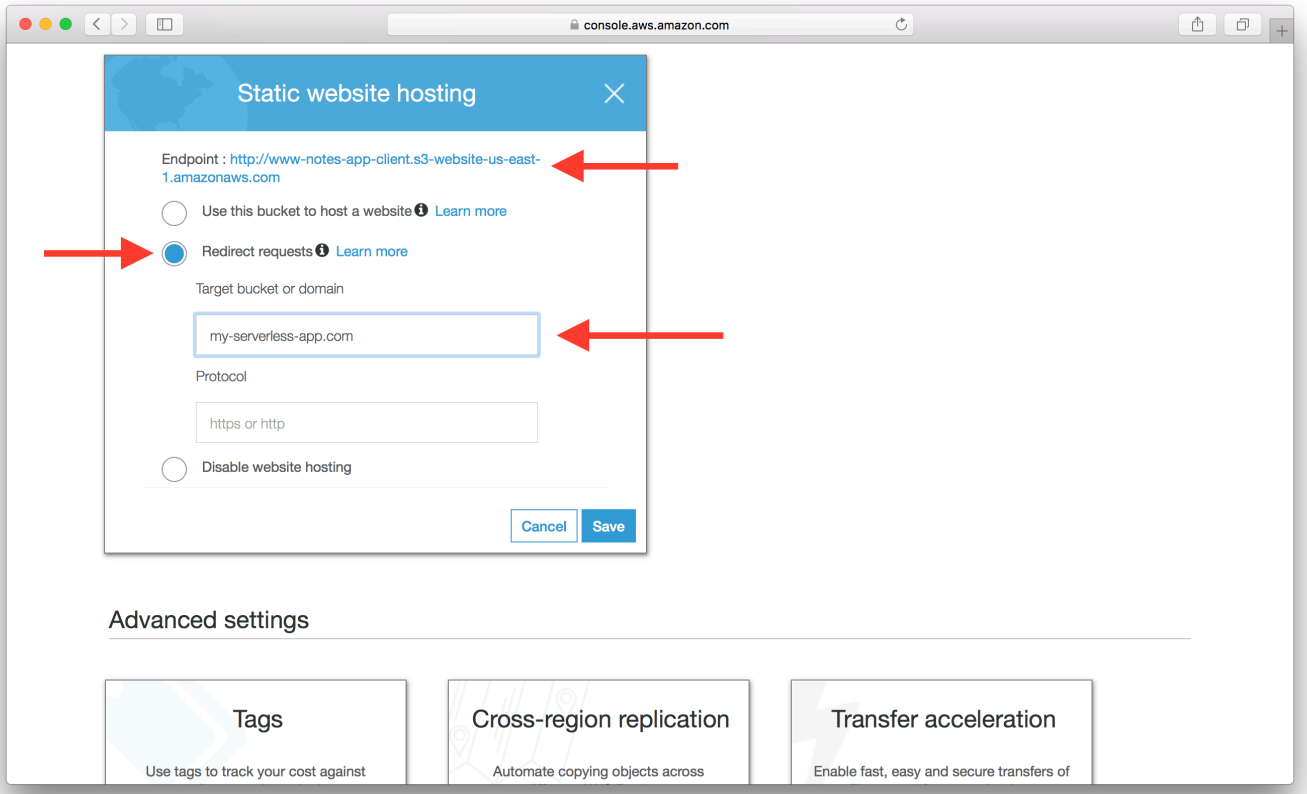
Now go into the **Properties** of the new bucket and click on the **Static**

website hosting.



But unlike last time we are going to select the **Redirect requests** option and fill in the domain we are going to be redirecting towards. This is the domain that we set up in our last chapter.

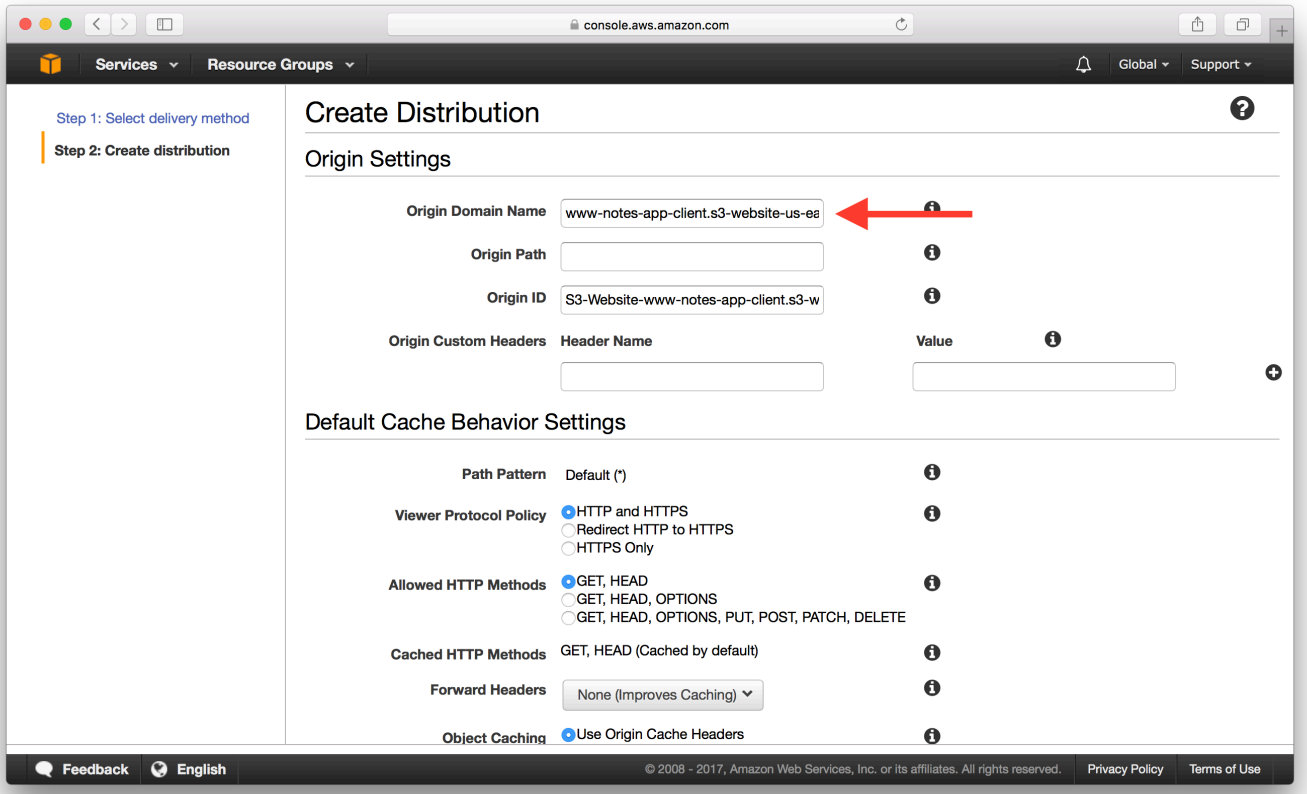
Also, make sure to copy the **Endpoint** as we'll be needing this later.



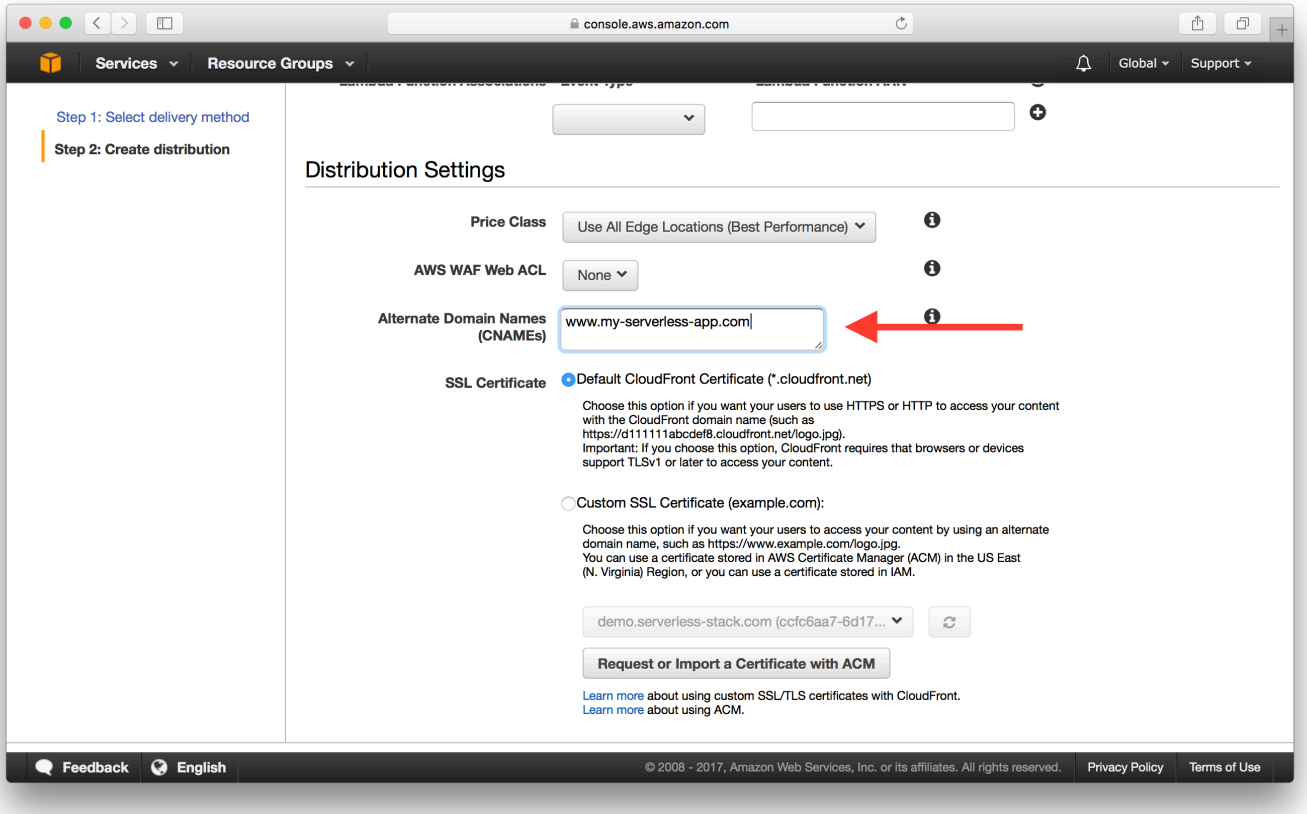
And hit **Save** to make the changes. Next we'll create a CloudFront Distribution to point to this S3 redirect Bucket.

Create a CloudFront Distribution

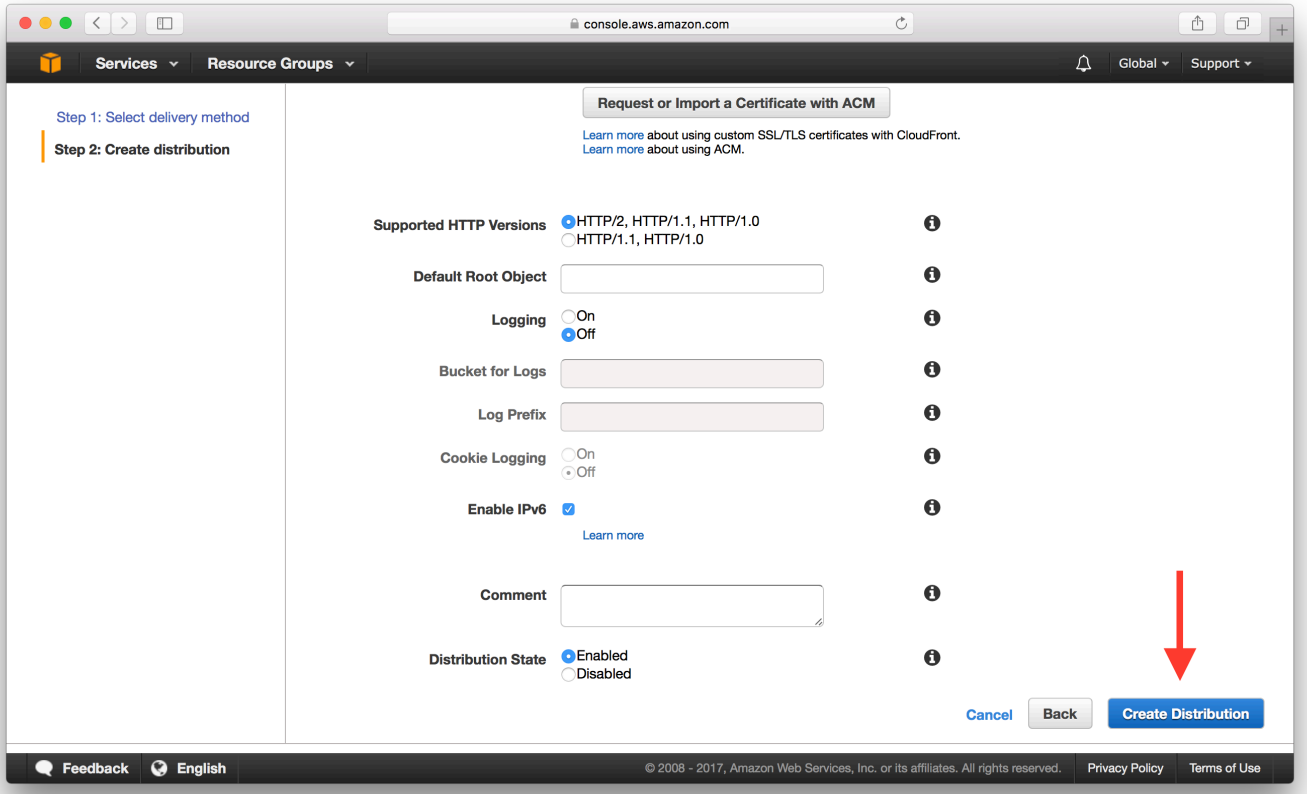
Create **a new CloudFront Distribution**. And copy the S3 **Endpoint** from the step above as the **Origin Domain Name**. Make sure to **not** use the one from the dropdown. In my case, it is `http://www-notes-app-client.s3-website-us-east-1.amazonaws.com`.



Scroll down to the **Alternate Domain Names (CNAMEs)** and use the www version of our domain name here.



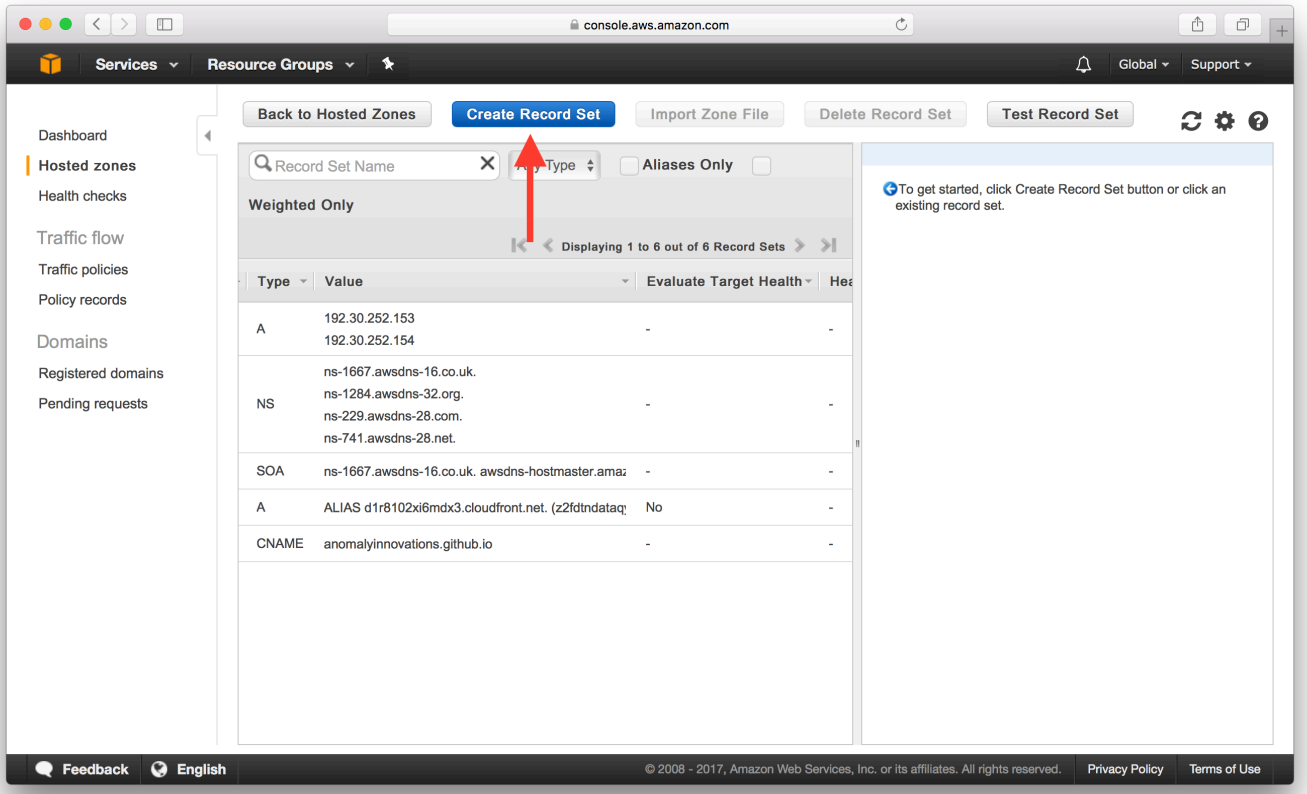
And hit **Create Distribution**.



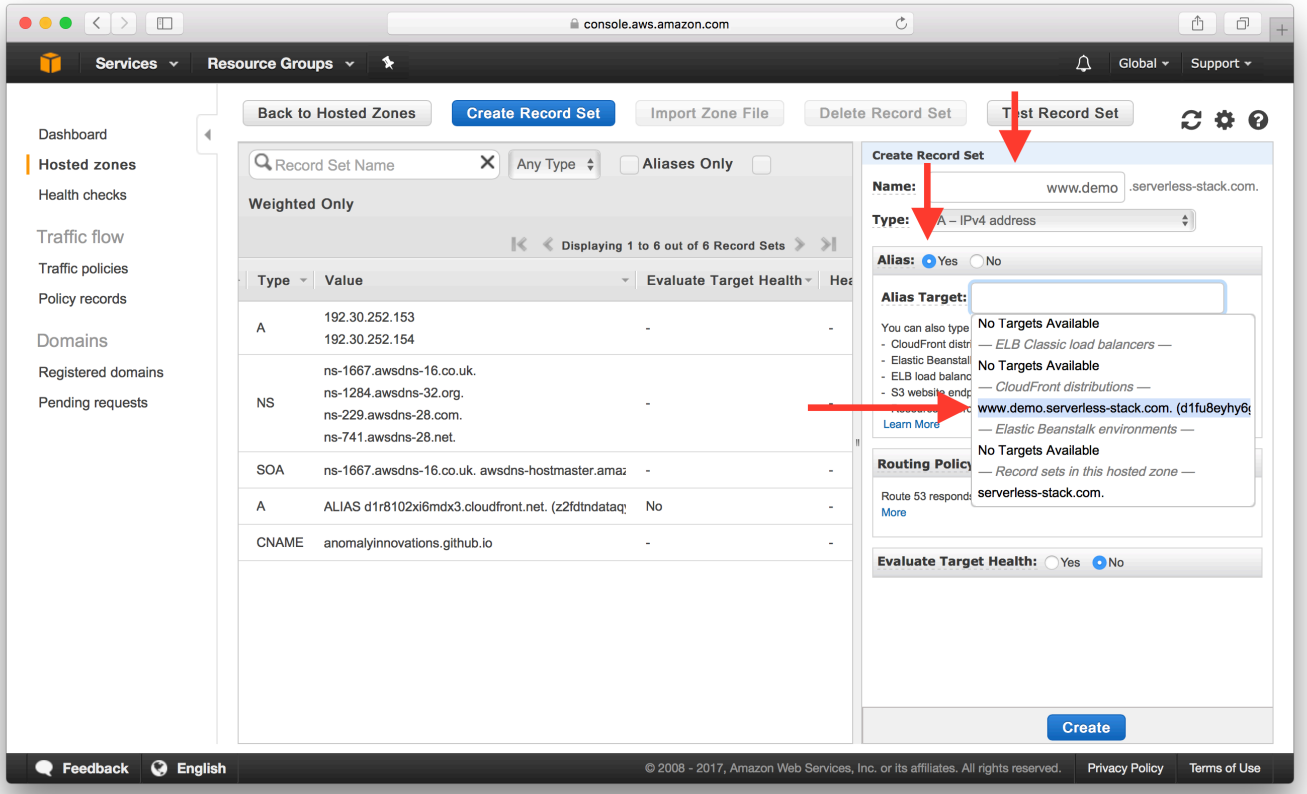
Finally, we'll point our www domain to this CloudFront Distribution.

Point WWW Domain to CloudFront Distribution

Head over to your domain in Route 53 and hit **Create Record Set**.



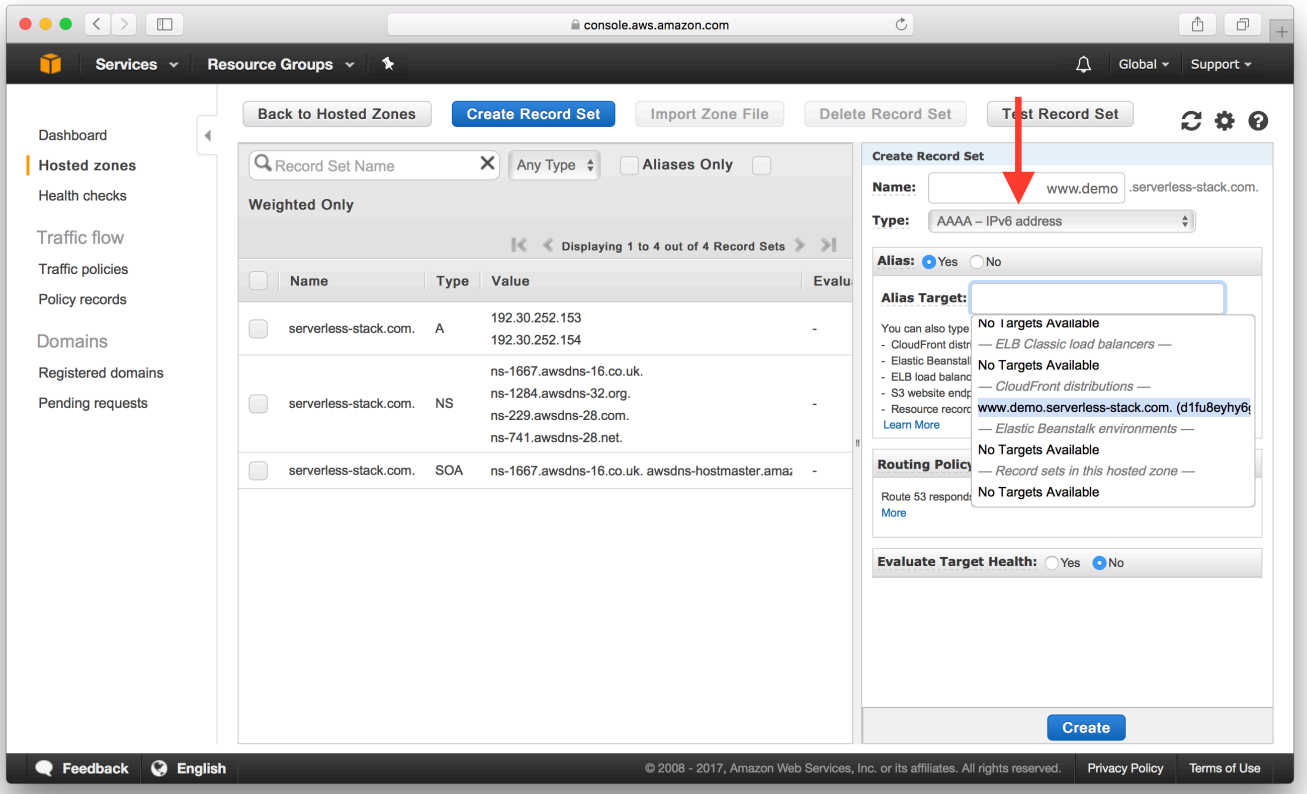
This time fill in `www` as the **Name** and select **Alias** as **Yes**. And pick your new CloudFront Distribution from the **Alias Target** dropdown.



Add IPv6 Support

Just as before, we need to add an AAAA record to support IPv6.

Create a new Record Set with the exact same settings as before, except make sure to pick **AAAA - IPv6 address** as the **Type**.



And that's it! Just give it some time for the DNS to propagate and if you visit your www version of your domain, it should redirect you to your non-www version.

Next, we'll set up SSL and add HTTPS support for our domains.

For help and discussion

Comments on this chapter

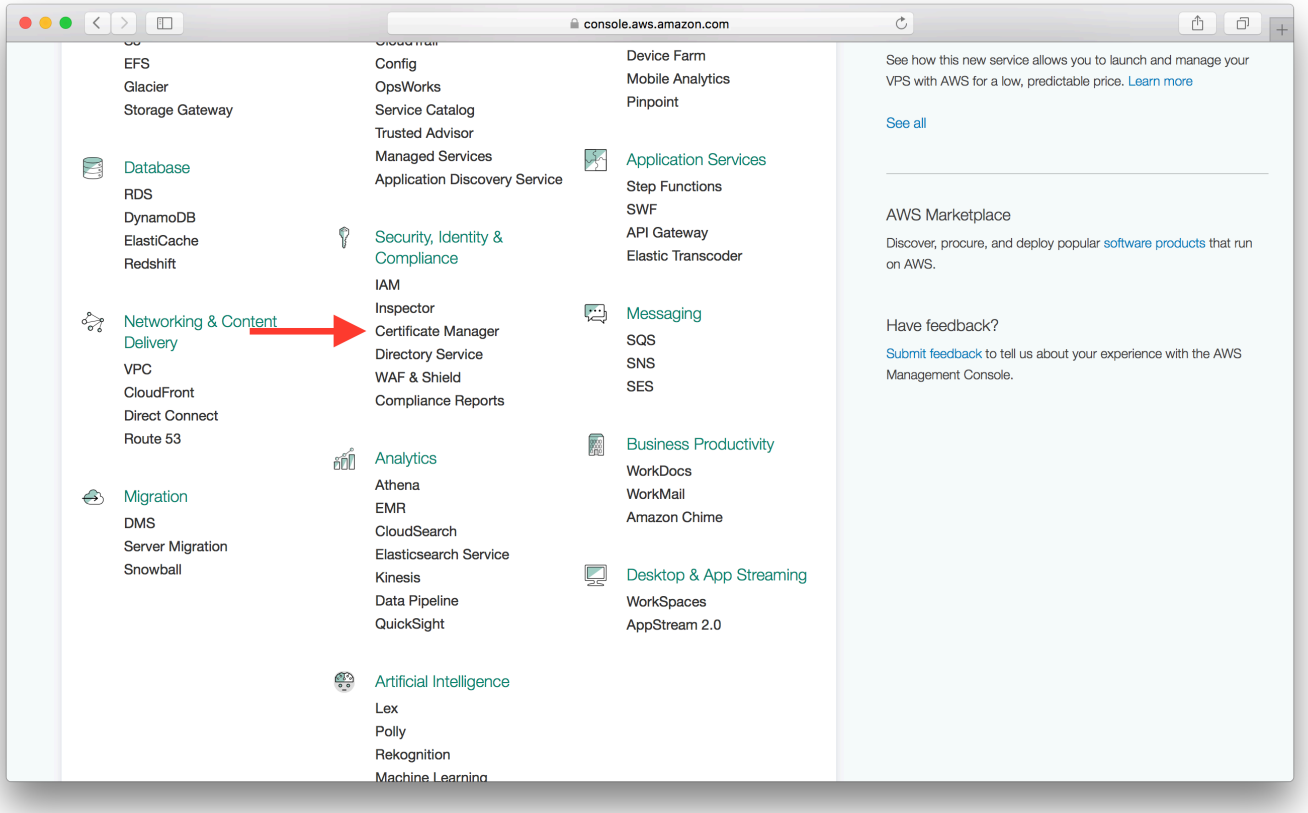
```
(https://discourse.serverless-
stack.com/t/set-up-www-domain-redirect/142)
```

Set up SSL

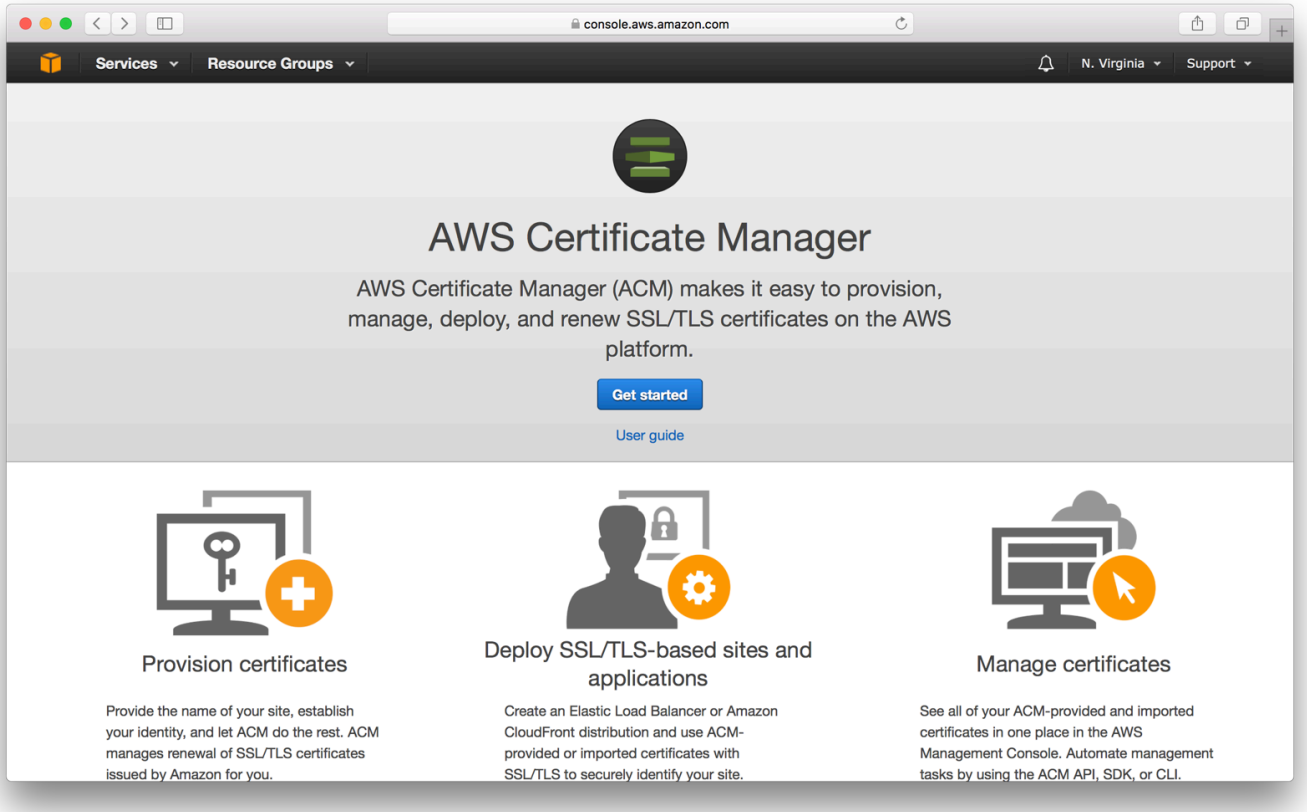
Now that our app is being served through our domain, let's add a layer of security to it by switching to HTTPS. AWS makes this fairly easy to do, thanks to Certificate Manager.

Request a Certificate

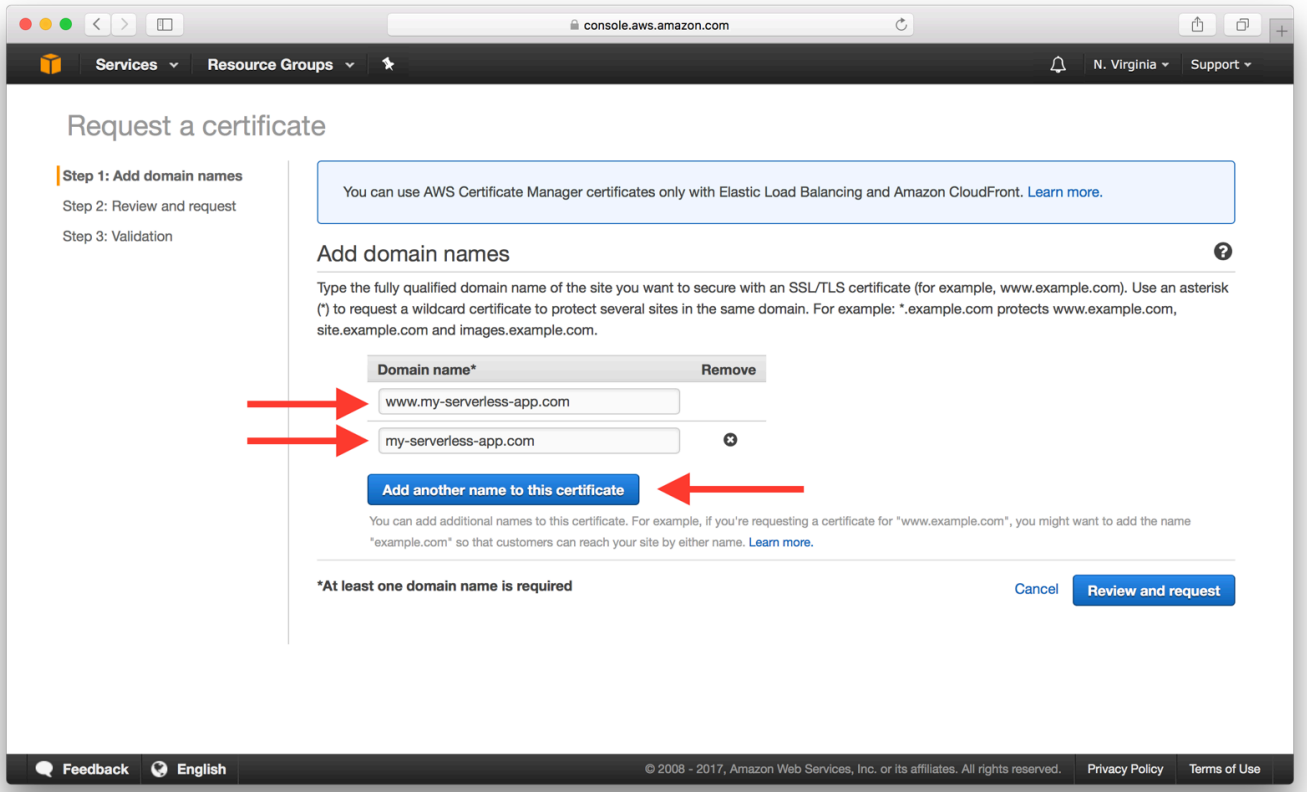
Select **Certificate Manager** from the list of services in your AWS Console (https://console.aws.amazon.com). Ensure that you are in the **US East (N. Virginia)** region. This is because a certificate needs to be from this region for it to work with CloudFront (http://docs.aws.amazon.com/acm/latest/userguide/acm-regions.html).



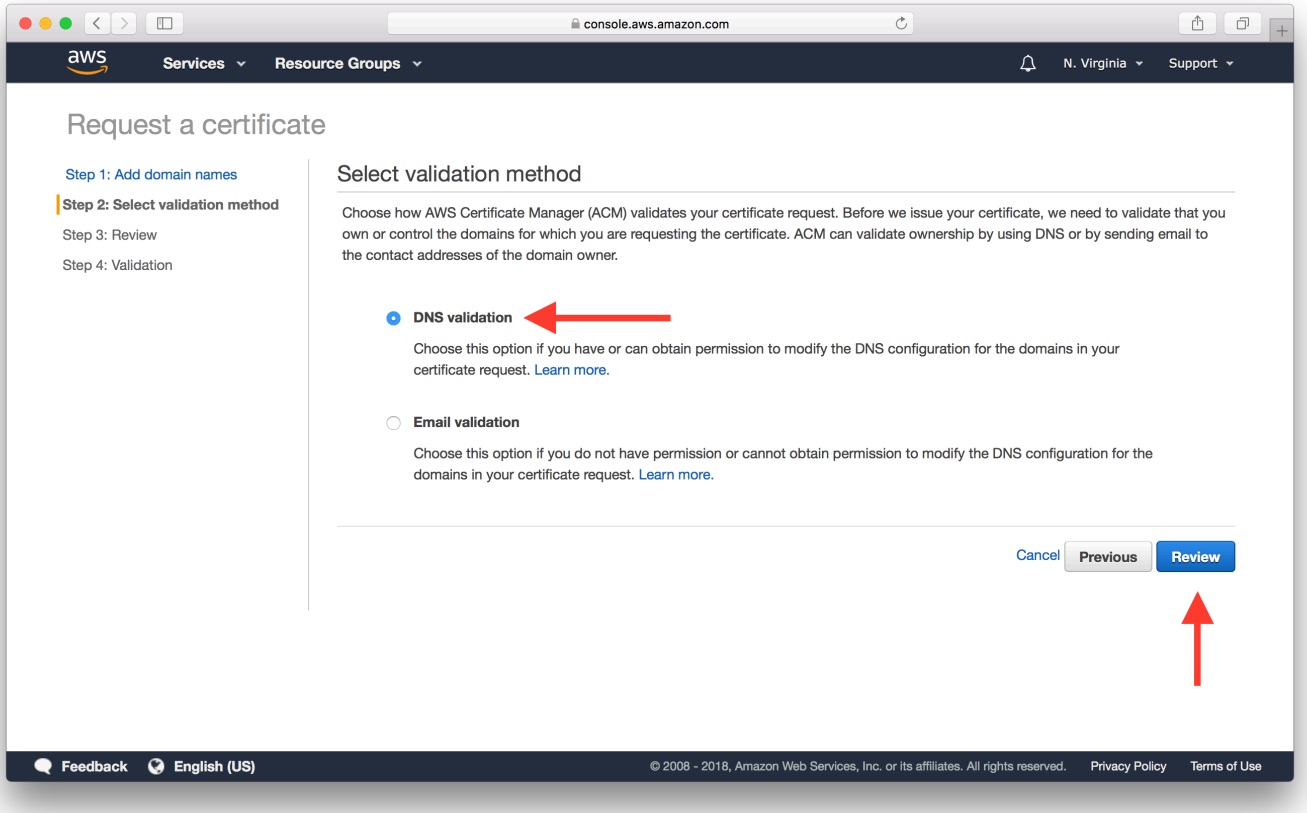
If this is your first certificate, you'll need to hit **Get started**. If not then hit **Request a certificate** from the top.



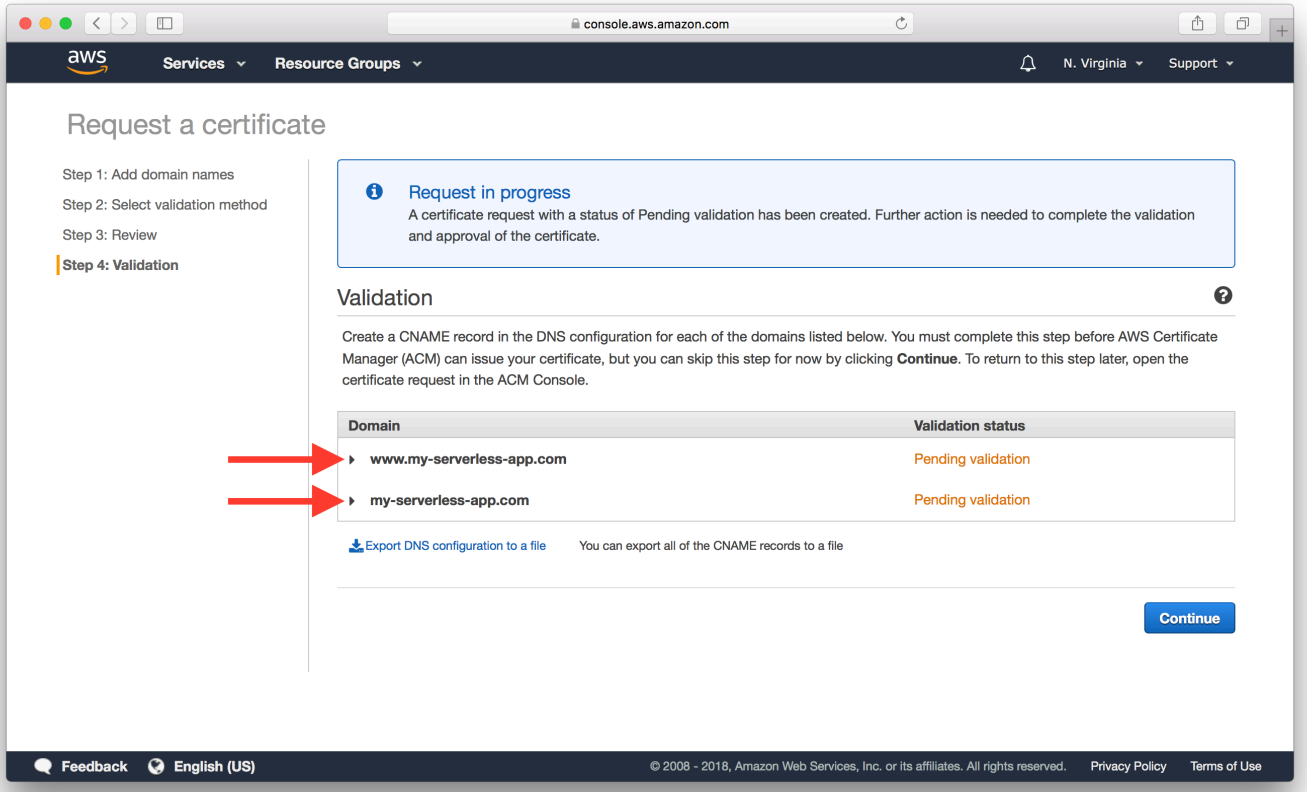
And type in the name of our domain. Hit **Add another name to this certificate** and add our www version of our domain as well. Hit **Review and request** once you are done.



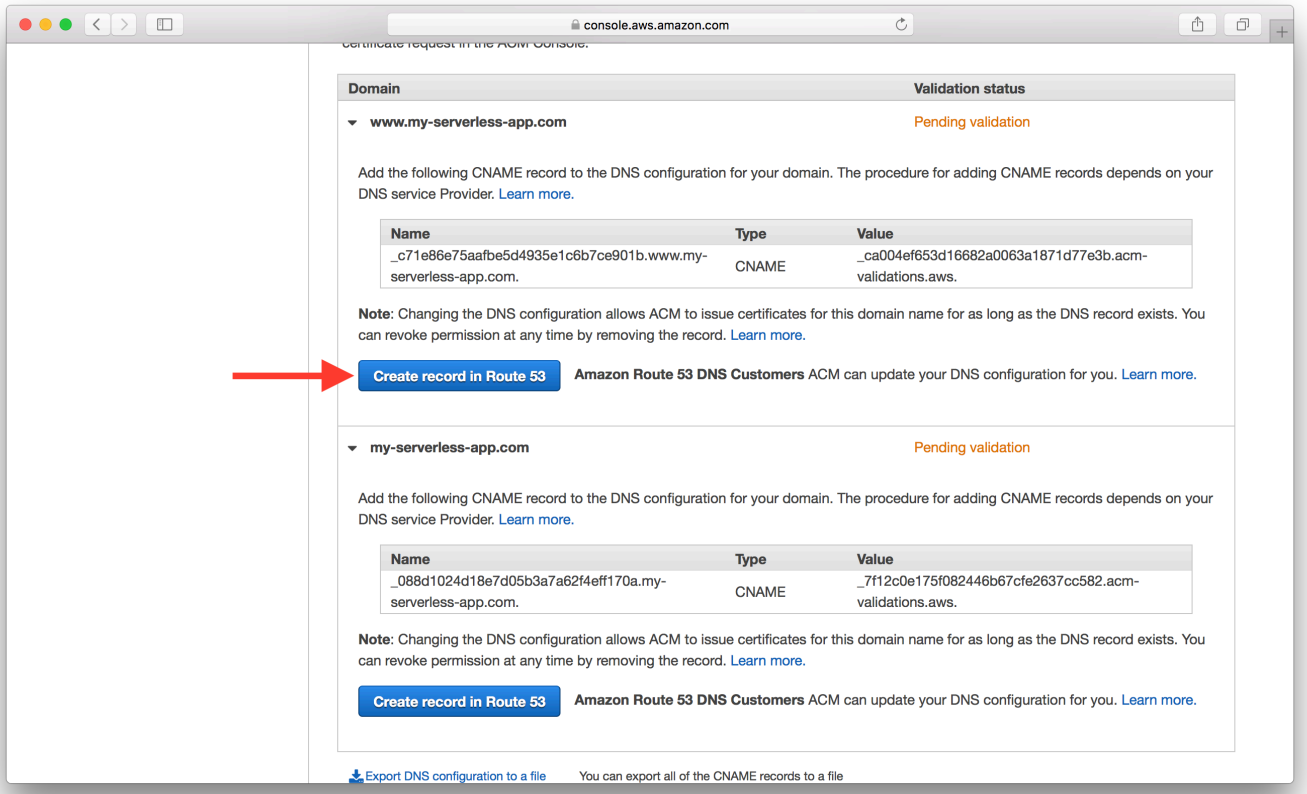
Now to confirm that we control the domain, select the **DNS validation** method and hit **Review**.



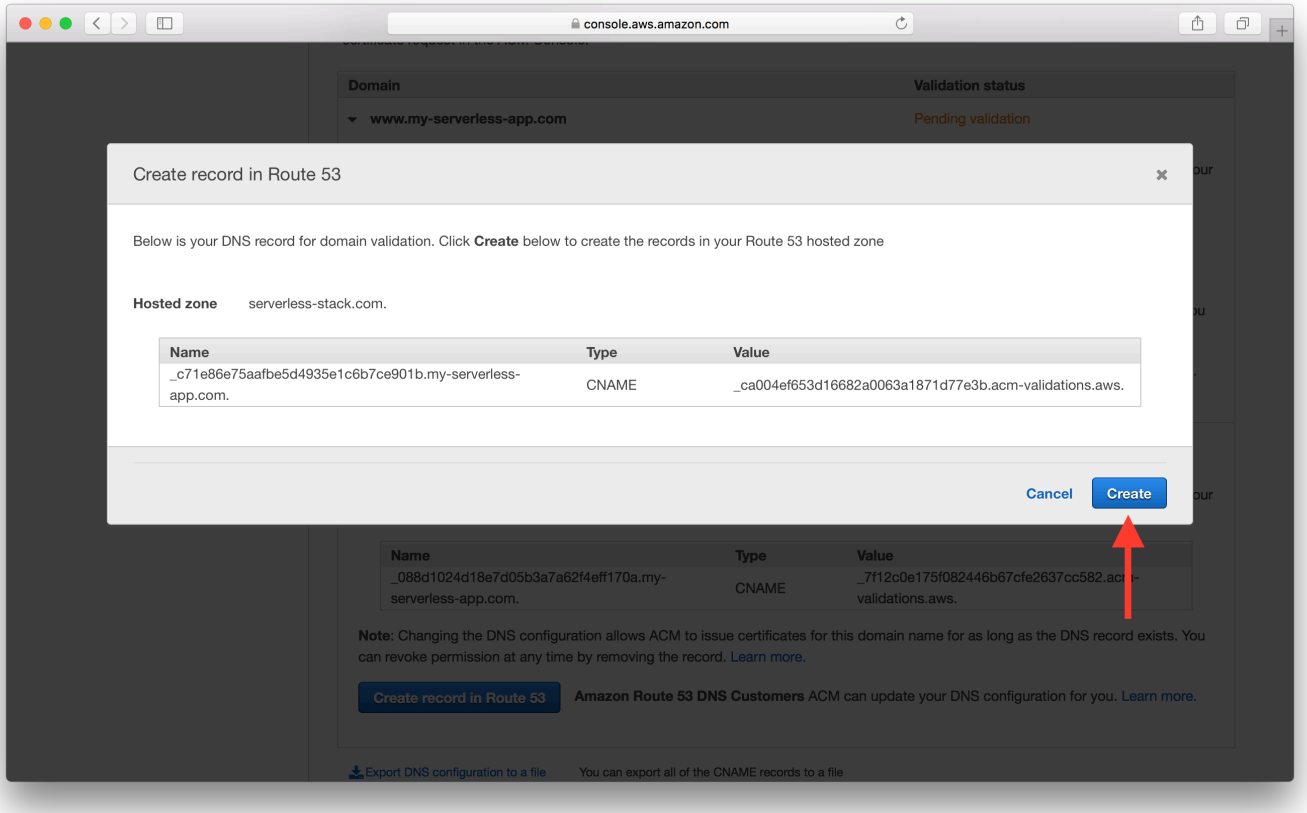
On the validation screen expand the two domains we are trying to validate.



Since we control the domain through Route 53, we can directly create the DNS record through here by hitting **Create record in Route 53**.



And confirm that you want the record to be created by hitting **Create**.



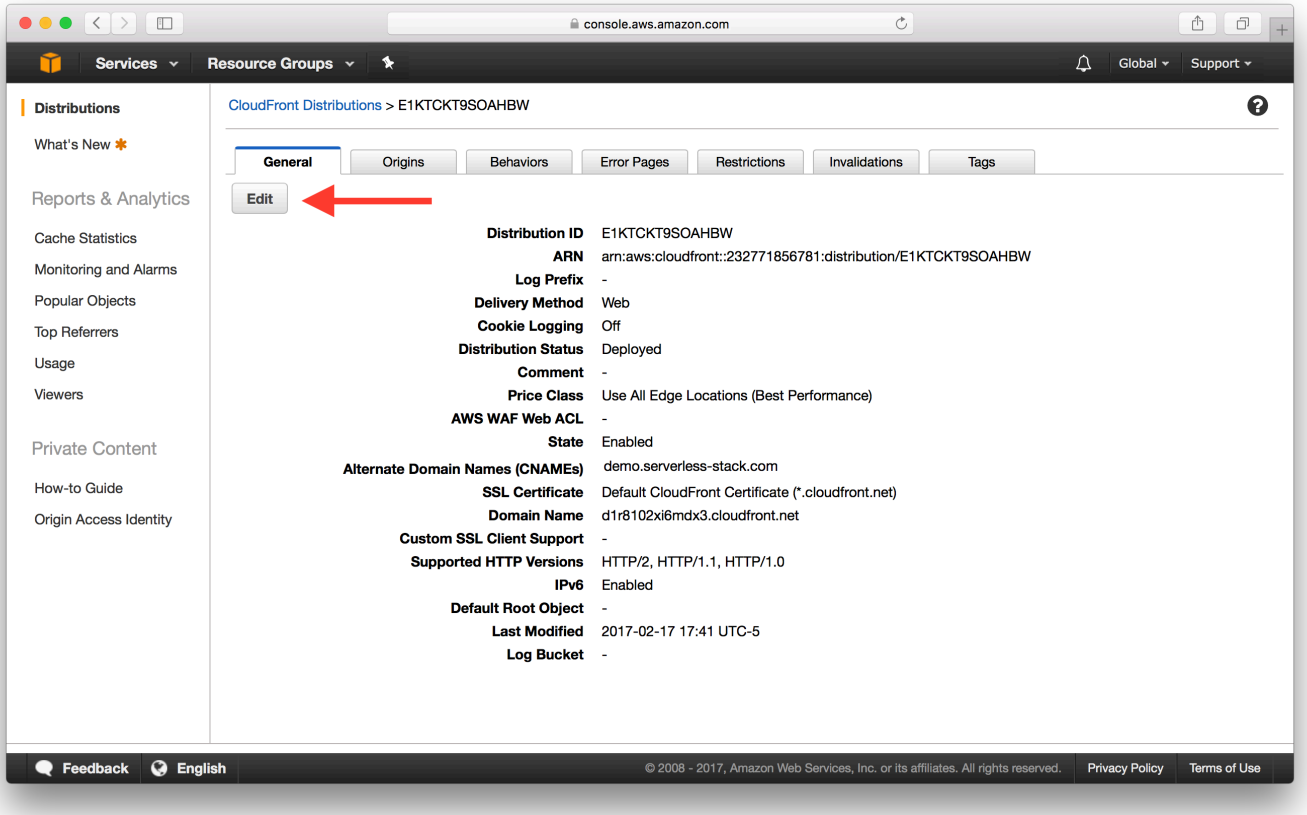
Also, make sure to do this for the other domain.

The process of creating a DNS record and validating it can take around 30 minutes.

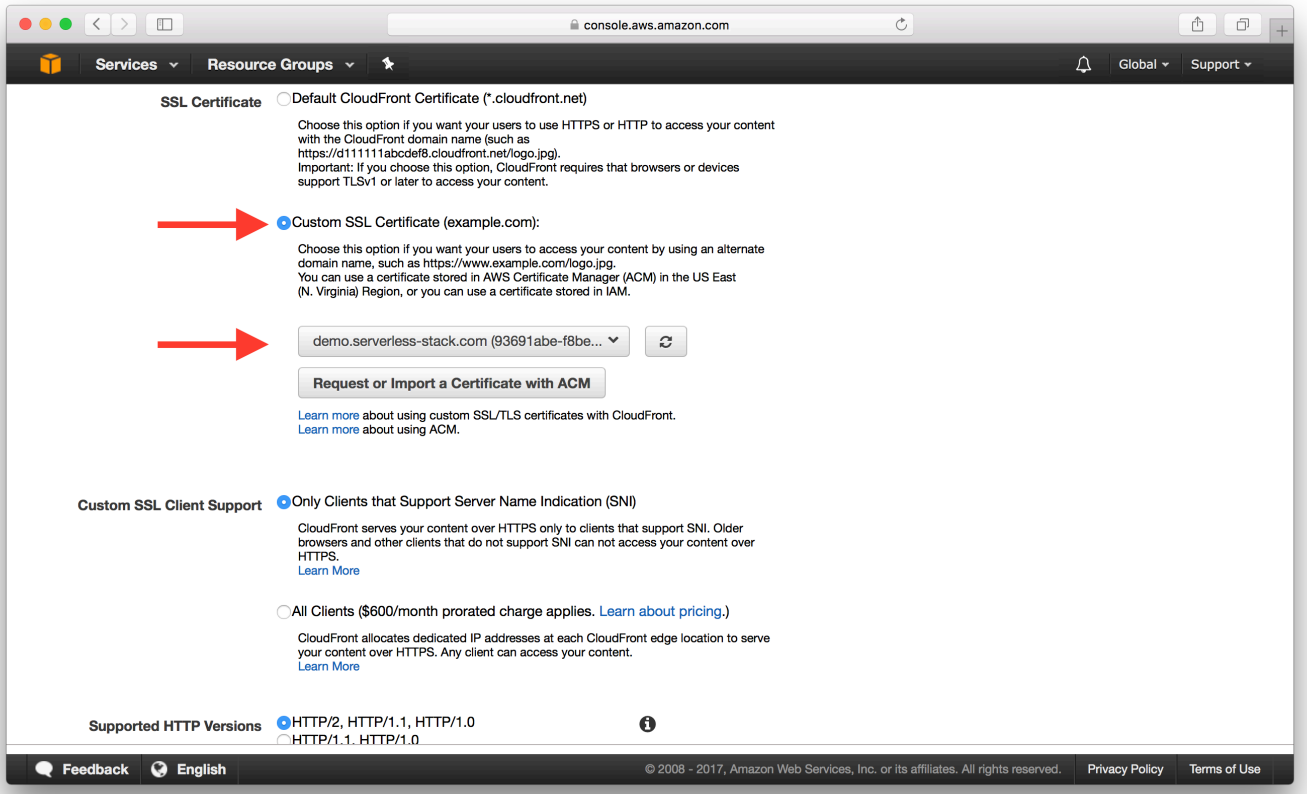
Next, we'll associate this certificate with our CloudFront Distributions.

Update CloudFront Distributions with Certificate

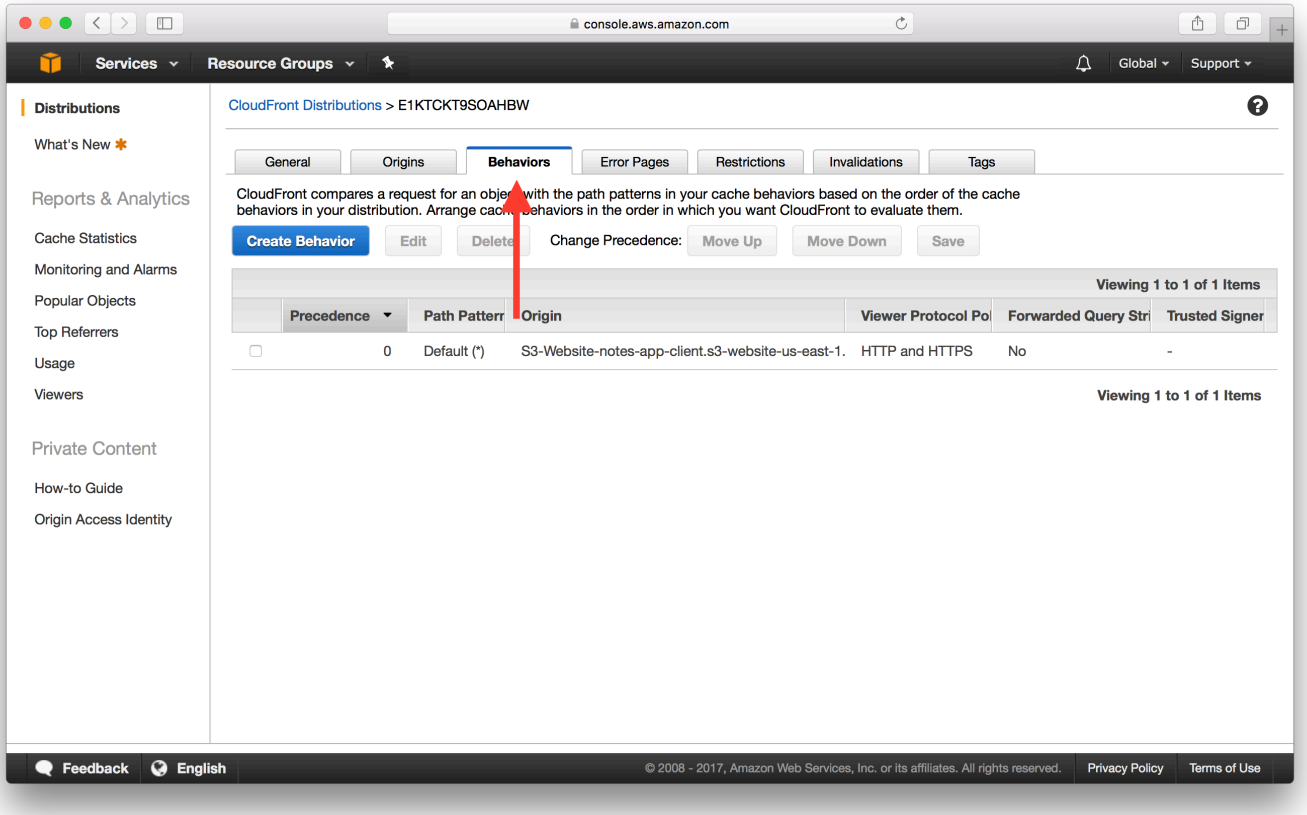
Open up our first CloudFront Distribution from our list of distributions and hit the **Edit** button.



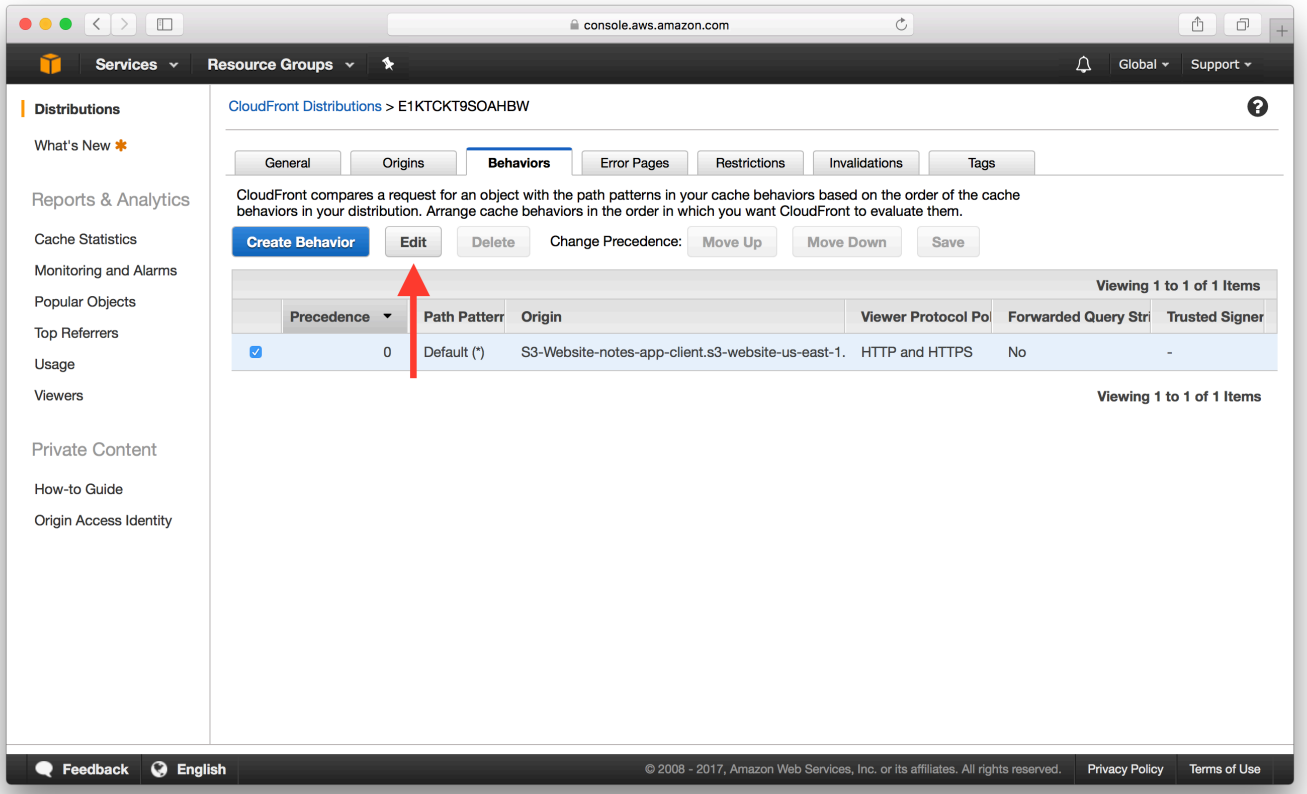
Now switch the **SSL Certificate** to **Custom SSL Certificate** and select the certificate we just created from the drop down. And scroll down to the bottom and hit **Yes, Edit**.



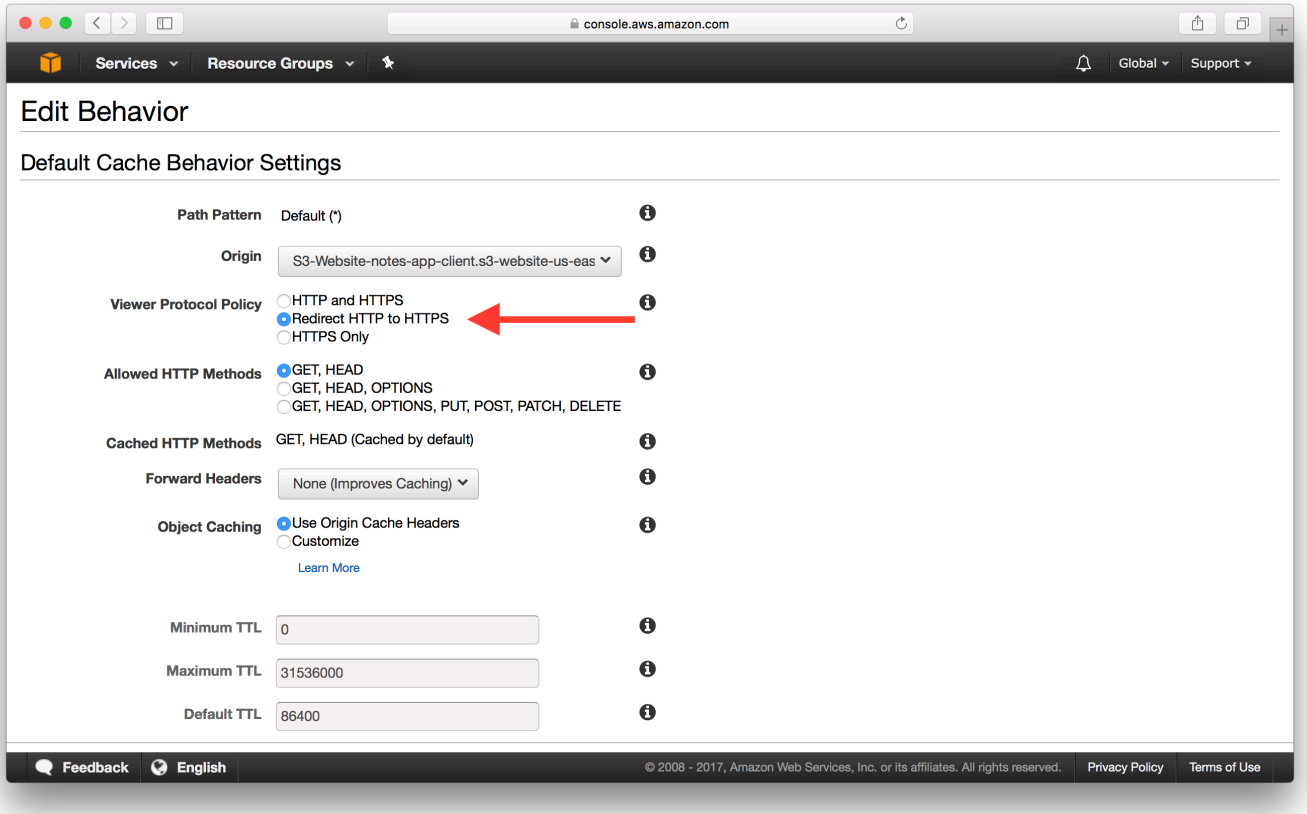
Next, head over to the **Behaviors** tab from the top.



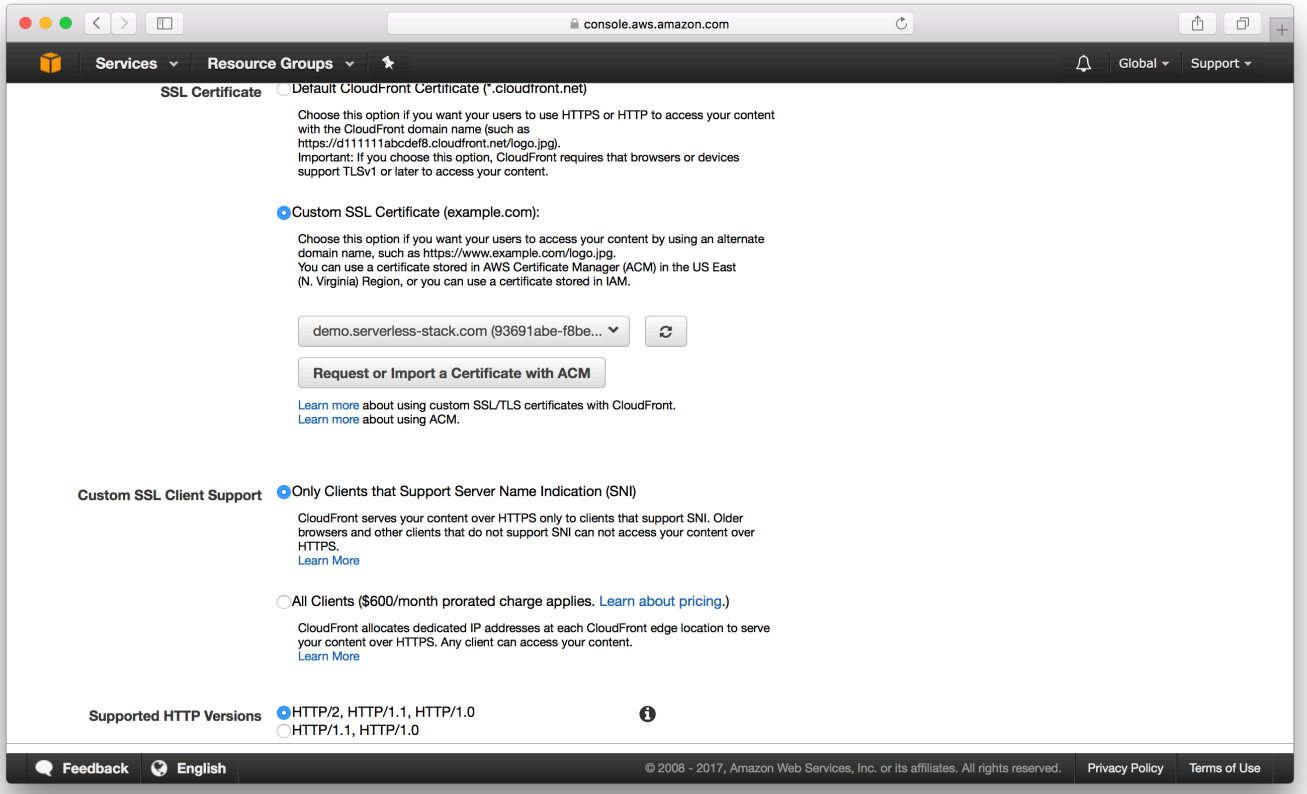
And select the only one we have and hit **Edit**.



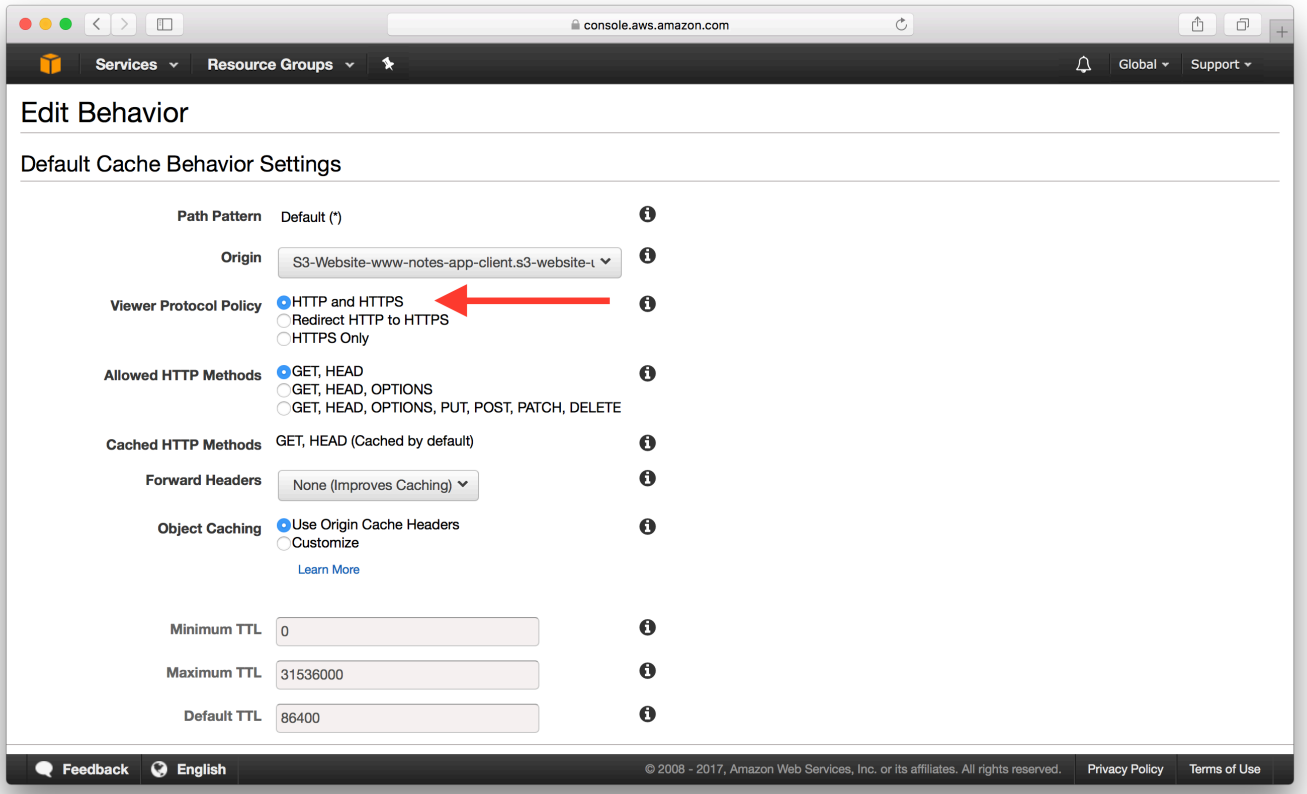
Then switch the **Viewer Protocol Policy** to **Redirect HTTP to HTTPS**. And scroll down to the bottom and hit **Yes, Edit**.



Now let's do the same for our other CloudFront Distribution.



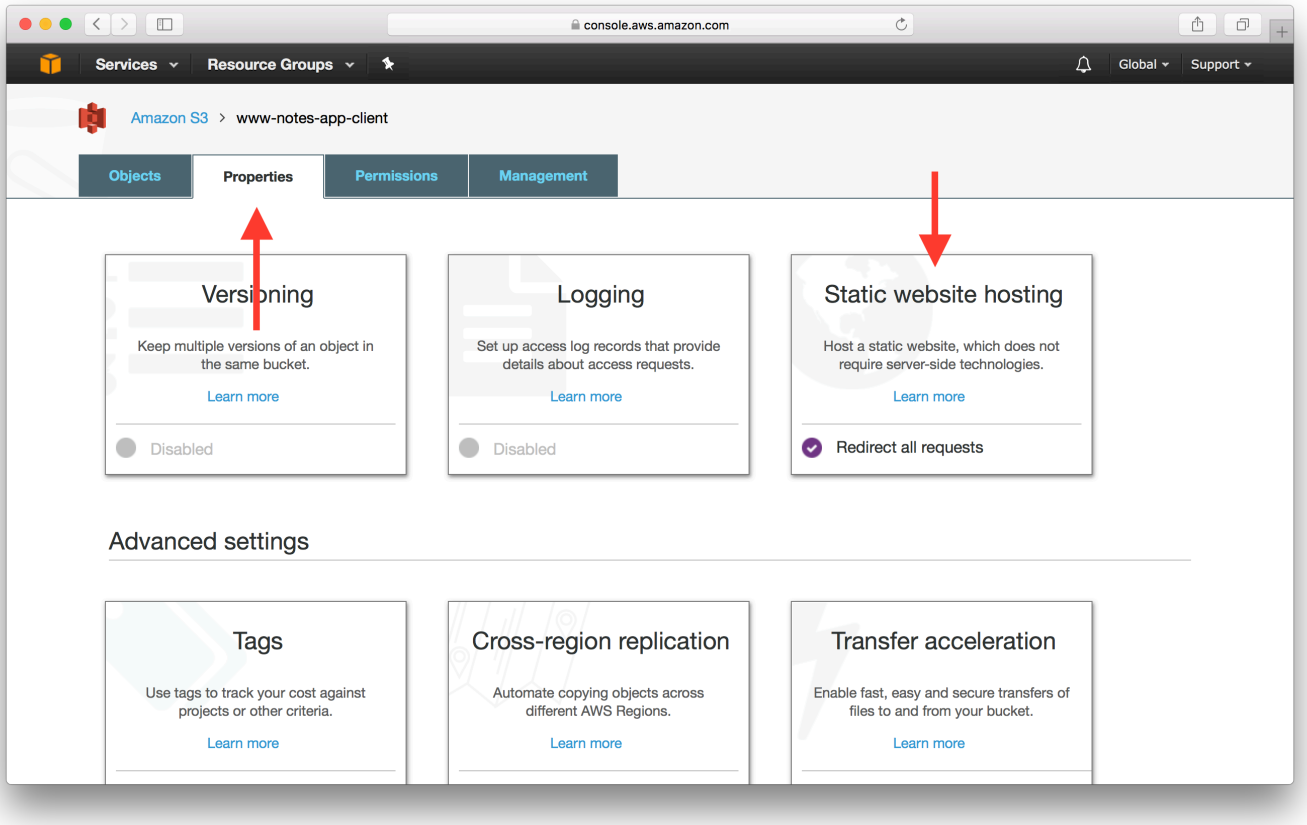
But leave the **Viewer Protocol Policy** as **HTTP and HTTPS**. This is because we want our users to go straight to the HTTPS version of our non-www domain. As opposed to redirecting to the HTTPS version of our www domain before redirecting again.



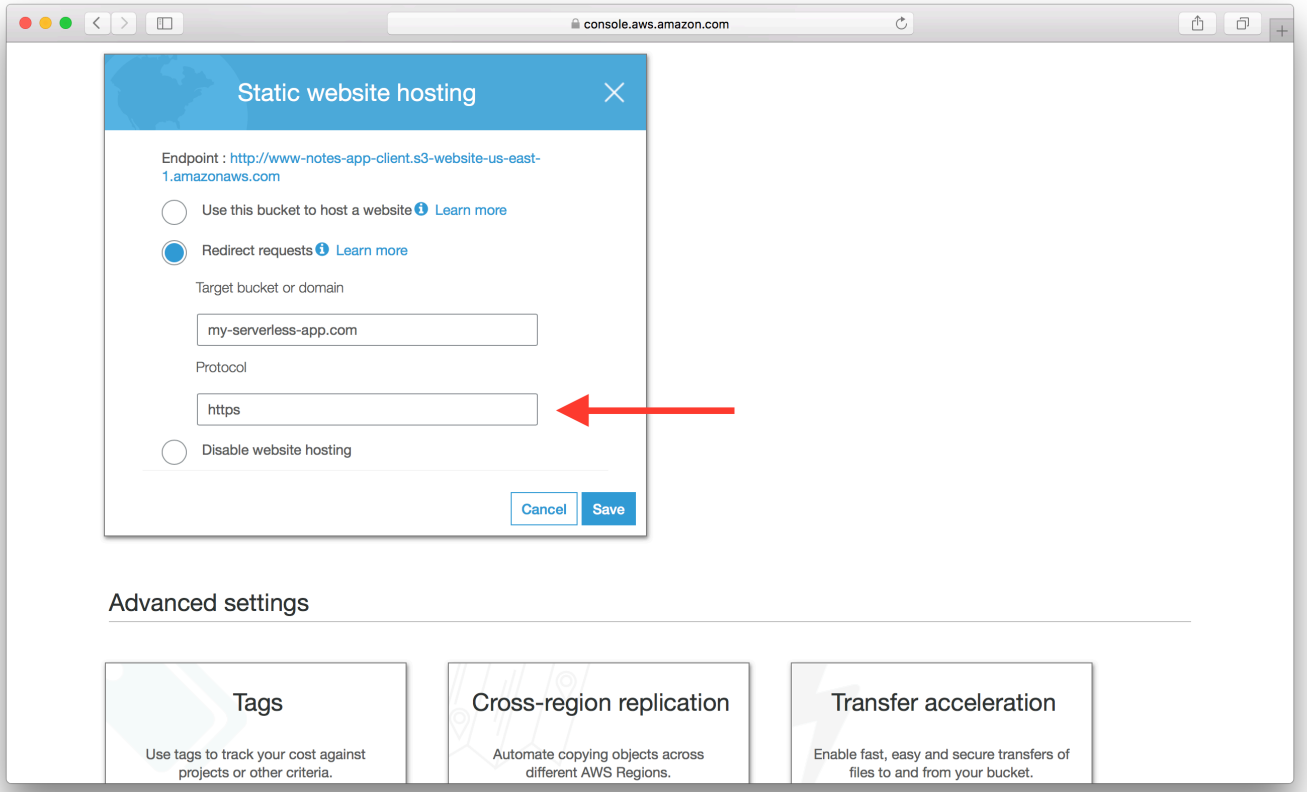
Update S3 Redirect Bucket

The S3 Redirect Bucket that we created in the last chapter is redirecting to the HTTP version of our non-www domain. We should switch this to the HTTPS version to prevent the extra redirect.

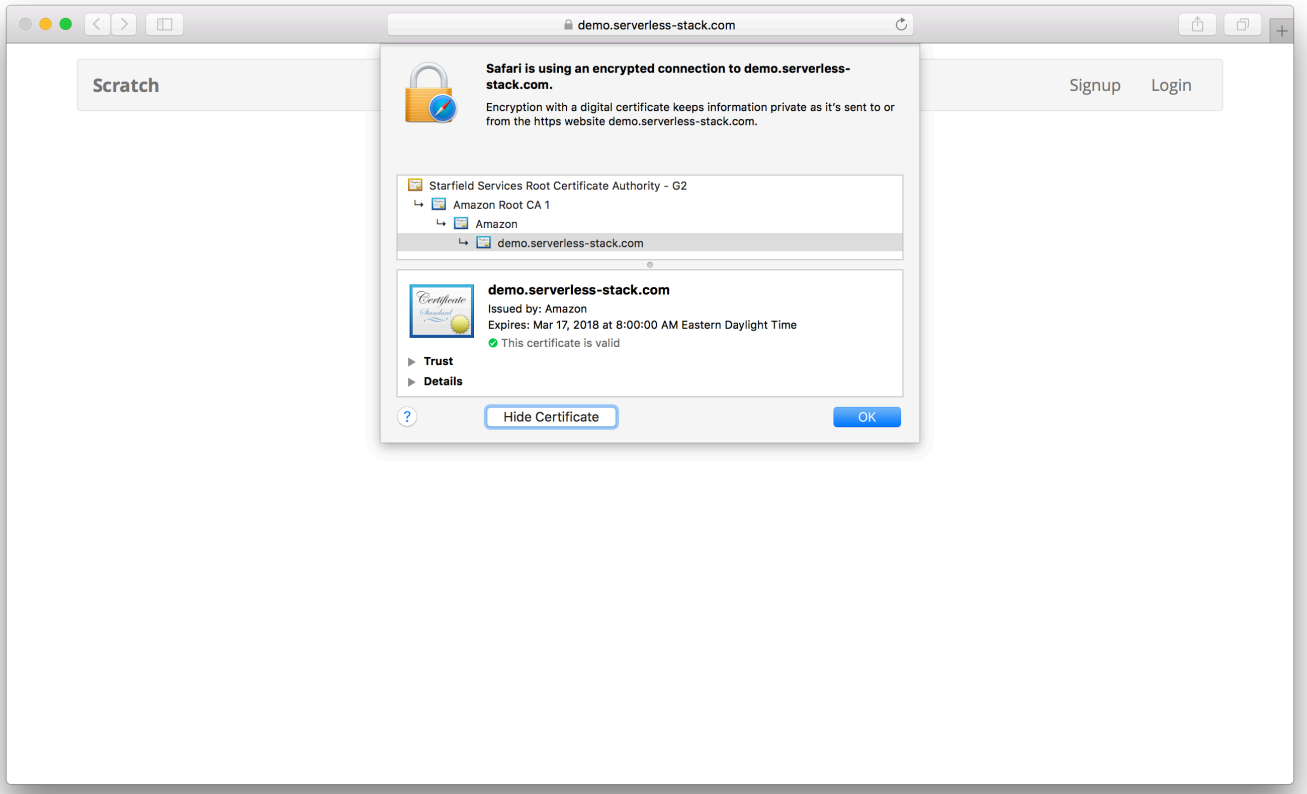
Open up the S3 Redirect Bucket we created in the last chapter. Head over to the **Properties** tab and select **Static website hosting**.



Change the **Protocol** to **https** and hit **Save**.



And that's it. Our app should be served out on our domain through HTTPS.



Next up, let's look at the process of deploying updates to our app.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/comments-for-set-up-ssl/133)

Deploy Updates

Now let's look at how we make changes and update our app. The process is very similar to how we deployed our code to S3 but with a few changes. Here is what it looks like.

1. Build our app with the changes
2. Deploy to the main S3 Bucket
3. Invalidate the cache in both our CloudFront Distributions

We need to do the last step since CloudFront caches our objects in its edge locations. So to make sure that our users see the latest version, we need to tell CloudFront to invalidate it's cache in the edge locations.

Let's assume you've made some changes to your app; you'll need to build these changes first.

Build Our App

First let's prepare our app for production by building it. Run the following in your working directory.

```
$ npm run build
```

Now that our app is built and ready in the `build/` directory, let's deploy to S3.

Upload to S3

Run the following from our working directory to upload our app to our main S3 Bucket. Make sure to replace `YOUR_S3_DEPLOY_BUCKET_NAME` with the S3 Bucket we created in the Create an S3 bucket (/chapters/create-an-s3-bucket.html) chapter.

```
$ aws s3 sync build/ s3://YOUR_S3_DEPLOY_BUCKET_NAME --
delete
```

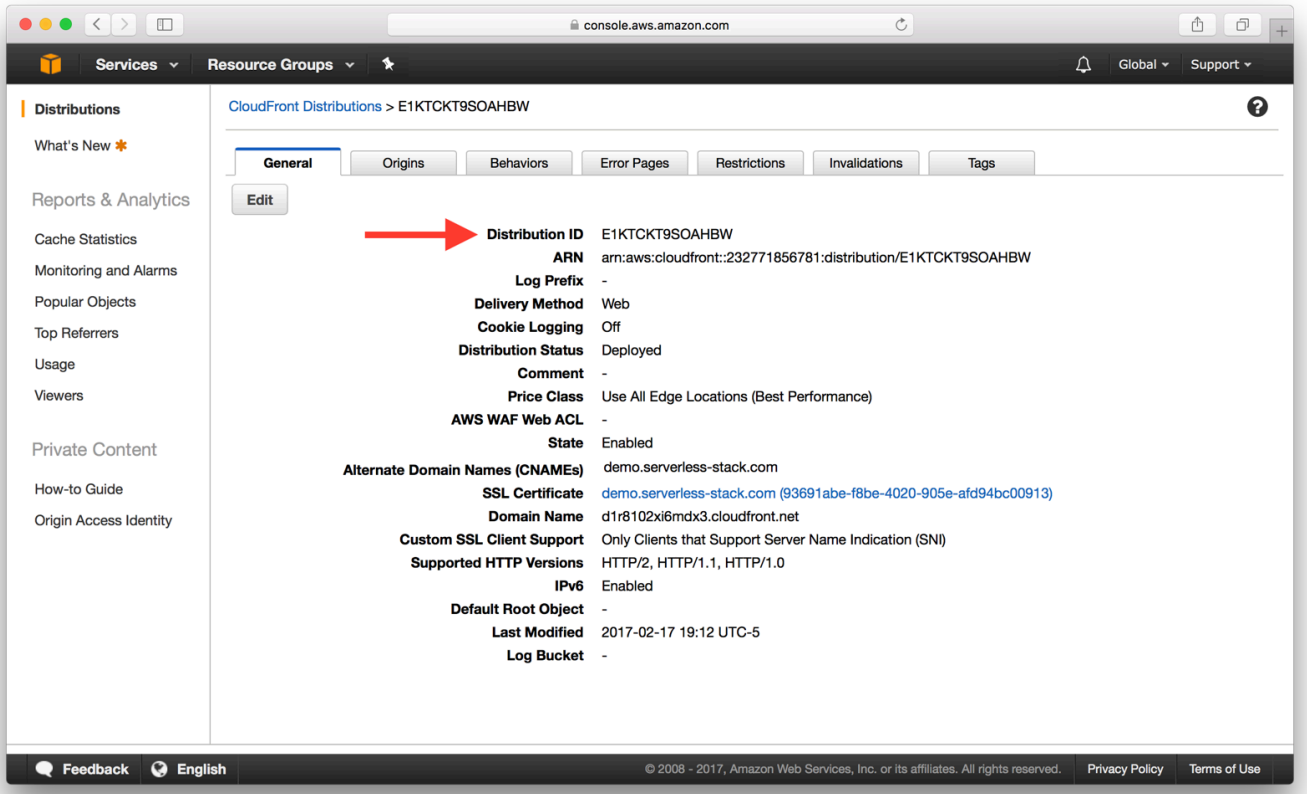
Note the `--delete` flag here; this is telling S3 to delete all the files that are in the bucket that we aren't uploading this time around. Create React App generates unique bundles when we build it and without this flag we'll end up retaining all the files from the previous builds. Our changes should be live on S3.

Now to ensure that CloudFront is serving out the updated version of our app, let's invalidate the CloudFront cache.

Invalidate the CloudFront Cache

CloudFront allows you to invalidate objects in the distribution by passing in the path of the object. But it also allows you to use a wildcard (`/*`) to invalidate the entire distribution in a single command. This is recommended when we are deploying a new version of our app.

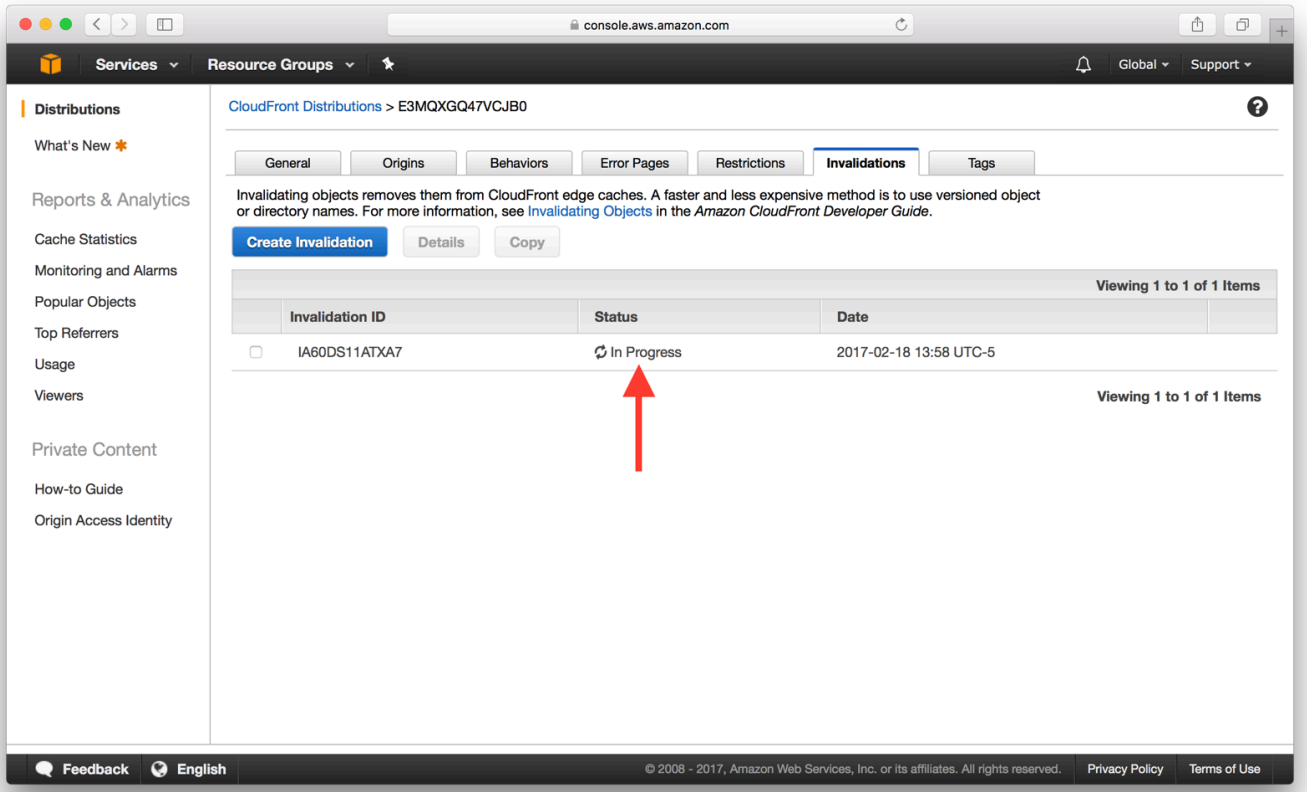
To do this we'll need the **Distribution ID** of **both** of our CloudFront Distributions. You can get it by clicking on the distribution from the list of CloudFront Distributions.



Now we can use the AWS CLI to invalidate the cache of the two distributions. Make sure to replace `YOUR_CF_DISTRIBUTION_ID` and `YOUR_WWW_CF_DISTRIBUTION_ID` with the ones from above.

```
$ aws cloudfront create-invalidation --distribution-id
YOUR_CF_DISTRIBUTION_ID --paths "/*"
$ aws cloudfront create-invalidation --distribution-id
YOUR_WWW_CF_DISTRIBUTION_ID --paths "/*"
```

This invalidates our distribution for both the www and non-www versions of our domain. If you click on the **Invalidations** tab, you should see your invalidation request being processed.



It can take a few minutes to complete. But once it is done, the updated version of our app should be live.

And that's it! We now have a set of commands we can run to deploy our updates. Let's quickly put them together so we can do it with one command.

Add a Deploy Command

NPM allows us to add a `deploy` command in our `package.json`.

✦ CHANGE Add the following in the `scripts` block above `eject` in the `package.json`.

```
"predeploy": "npm run build",
"deploy": "aws s3 sync build/
s3://YOUR_S3_DEPLOY_BUCKET_NAME --delete",
```

```
"postdeploy": "aws cloudfront create-invalidation --
distribution-id YOUR_CF_DISTRIBUTION_ID --paths '/*' &&
aws cloudfront create-invalidation --distribution-id
YOUR_WWW_CF_DISTRIBUTION_ID --paths '/*'",
```

Make sure to replace `YOUR_S3_DEPLOY_BUCKET_NAME`, `YOUR_CF_DISTRIBUTION_ID`, and `YOUR_WWW_CF_DISTRIBUTION_ID` with the ones from above.

For Windows users, if `postdeploy` returns an error like.

```
An error occurred (InvalidArgument) when calling the
CreateInvalidation operation: Your request contains one
or more invalid invalidation paths.
```

Make sure that there is no quote in the `/*`.

```
"postdeploy": "aws cloudfront create-invalidation --
distribution-id YOUR_CF_DISTRIBUTION_ID --paths /* &&
aws cloudfront create-invalidation --distribution-id
YOUR_WWW_CF_DISTRIBUTION_ID --paths /*",
```

Now simply run the following command from your project root when you want to deploy your updates. It'll build your app, upload it to S3, and invalidate the CloudFront cache.

```
$ npm run deploy
```

And that's it! Now you have a workflow for deploying and updating your React app on AWS using S3 and CloudFront.

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/deploy-updates/16)

Manage User Accounts in AWS Amplify

If you've followed along with the first part of Serverless Stack (/#the-basics) guide, you might be looking to add ways your users can better manage their accounts. This includes the ability to:

- Reset their password in case they forget it
- Change their password once they are logged in
- And change the email they are logging in with

As a quick refresher, we are using AWS Cognito (https://aws.amazon.com/cognito/) as our authentication and user management provider. And on the frontend we are using AWS Amplify (https://aws-amplify.github.io/) with our Create React App (https://github.com/facebook/create-react-app).

In the next few chapters we are going to look at how to add the above functionality to our Serverless notes app (https://demo.serverless-stack.com). For these chapters we are going to use a forked version of the notes app. You can view the hosted version here (https://demo-user-mgmt.serverless-stack.com) and the source is available in a repo here (https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client).

Let's get started by allowing users to reset their password in case they have forgotten it.

CHECKPOINT

For reference, here is the code we have so far.

`User Management Frontend Source (https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client)`

For help and discussion

`Comments on this chapter (https://discourse.serverless-stack.com/t/manage-user-accounts-in-aws-amplify/505)`

Handle Forgot and Reset Password

In our Serverless notes app (https://demo.serverless-stack.com) we've used Cognito User Pool (https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools.html) to sign up and login our users. In the frontend we've used AWS Amplify (https://aws-amplify.github.io/) in our React app. However, if our users have forgotten their passwords, we need to have a way for them to reset their password. In this chapter we will look at how to do this.

The version of the notes app used in this chapter is hosted in a:

- Separate GitHub repository: **https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client** (https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client)
- And can be accessed through: **https://demo-user-mgmt.serverless-stack.com** (https://demo-user-mgmt.serverless-stack.com)

Let's look at the main changes we need to make to allow users to reset their password.

Add a Reset Password Form

✦ CHANGE We are going to create a

`src/containers/ResetPassword.js`.

```
import React, { Component } from "react";
import { Auth } from "aws-amplify";
import { Link } from "react-router-dom";
import {
  HelpBlock,
  FormGroup,
  Glyphicon,
  FormControl,
  ControlLabel
} from "react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import "./ResetPassword.css";

export default class ResetPassword extends Component {
  constructor(props) {
    super(props);

    this.state = {
      code: "",
      email: "",
      password: "",
      codeSent: false,
      confirmed: false,
      confirmPassword: "",
      isConfirming: false,
      isSendingCode: false
    };
  }
```

```
  validateCodeForm() {
    return this.state.email.length > 0;
  }

  validateResetForm() {
    return (
      this.state.code.length > 0 &&
      this.state.password.length > 0 &&
      this.state.password ===
this.state.confirmPassword
    );
  }

  handleChange = event => {
    this.setState({
      [event.target.id]: event.target.value
    });
  };

  handleSendCodeClick = async event => {
    event.preventDefault();

    this.setState({ isSendingCode: true });

    try {
      await Auth.forgotPassword(this.state.email);
      this.setState({ codeSent: true });
    } catch (e) {
      alert(e.message);
      this.setState({ isSendingCode: false });
    }
```

```
};

handleConfirmClick = async event => {
  event.preventDefault();

  this.setState({ isConfirming: true });

  try {
    await Auth.forgotPasswordSubmit(
      this.state.email,
      this.state.code,
      this.state.password
    );
    this.setState({ confirmed: true });
  } catch (e) {
    alert(e.message);
    this.setState({ isConfirming: false });
  }
};

renderRequestCodeForm() {
  return (
    <form onSubmit={this.handleSendCodeClick}>
      <FormGroup bsSize="large" controlId="email">
        <ControlLabel>Email</ControlLabel>
        <FormControl
          autoFocus
          type="email"
          value={this.state.email}
          onChange={this.handleChange}
        />
```

```
        </FormGroup>
        <LoaderButton
          block
          type="submit"
          bsSize="large"
          loadingText="Sending…"
          text="Send Confirmation"
          isLoading={this.state.isSendingCode}
          disabled={!this.validateCodeForm()}
        />
      </form>
    );
  }

  renderConfirmationForm() {
    return (
      <form onSubmit={this.handleConfirmClick}>
        <FormGroup bsSize="large" controlId="code">
          <ControlLabel>Confirmation
Code</ControlLabel>
          <FormControl
            autoFocus
            type="tel"
            value={this.state.code}
            onChange={this.handleChange}
          />
          <HelpBlock>
            Please check your email
({this.state.email}) for the confirmation
            code.
          </HelpBlock>
```

```
        </FormGroup>
        <hr />
        <FormGroup bsSize="large" controlId="password">
          <ControlLabel>New Password</ControlLabel>
          <FormControl
            type="password"
            value={this.state.password}
            onChange={this.handleChange}
          />
        </FormGroup>
        <FormGroup bsSize="large"
controlId="confirmPassword">
          <ControlLabel>Confirm Password</ControlLabel>
          <FormControl
            type="password"
            onChange={this.handleChange}
            value={this.state.confirmPassword}
          />
        </FormGroup>
        <LoaderButton
          block
          type="submit"
          bsSize="large"
          text="Confirm"
          loadingText="Confirm…"
          isLoading={this.state.isConfirming}
          disabled={!this.validateResetForm()}
        />
      </form>
    );
  }
```

```
  renderSuccessMessage() {
    return (
      <div className="success">
        <Glyphicon glyph="ok" />
        <p>Your password has been reset.</p>
        <p>
          <Link to="/login">
            Click here to login with your new
credentials.
          </Link>
        </p>
      </div>
    );
  }

  render() {
    return (
      <div className="ResetPassword">
        {!this.state.codeSent
          ? this.renderRequestCodeForm()
          : !this.state.confirmed
            ? this.renderConfirmationForm()
            : this.renderSuccessMessage()}
      </div>
    );
  }
}
```

Let's quickly go over the flow here:

- We ask the user to put in the email address for their account in the

`this.renderRequestCodeForm()`.

- Once the user submits this form, we start the process by calling `Auth.forgotPassword(this.state.email)`. Where `Auth` is a part of the AWS Amplify library.
- This triggers Cognito to send a verification code to the specified email address.
- Then we present a form where the user can input the code that Cognito sends them. This form is rendered in `this.renderConfirmationForm()`. And it also allows the user to put in their new password.
- Once they submit this form with the code and their new password, we call `Auth.forgotPasswordSubmit(this.state.email, this.state.code, this.state.password)`. This resets the password for the account.
- Finally, we show the user a sign telling them that their password has been successfully reset. We also link them to the login page where they can login using their new details.

Let's also add a couple of styles.

✦ CHANGE Add the following to `src/containers/ResetPassword.css`.

```
@media all and (min-width: 480px) {
  .ResetPassword {
    padding: 60px 0;
  }

  .ResetPassword form {
    margin: 0 auto;
    max-width: 320px;
```

```
  }

  .ResetPassword .success {
    max-width: 400px;
  }
}

.ResetPassword .success {
  margin: 0 auto;
  text-align: center;
}
.ResetPassword .success .glyphicon {
  color: grey;
  font-size: 30px;
  margin-bottom: 30px;
}
```

Add the Route

Finally, let's link this up with the rest of our app.

✦ CHANGE Add the route to `src/Routes.js`.

```
<UnauthenticatedRoute
  path="/login/reset"
  exact
  component={ResetPassword}
  props={childProps}
/>
```

✦ CHANGE And import it in the header.

```
import ResetPassword from "./containers/ResetPassword";
```

Link from the Login Page

Now we want to make sure that our users are directed to this page when they are trying to login.

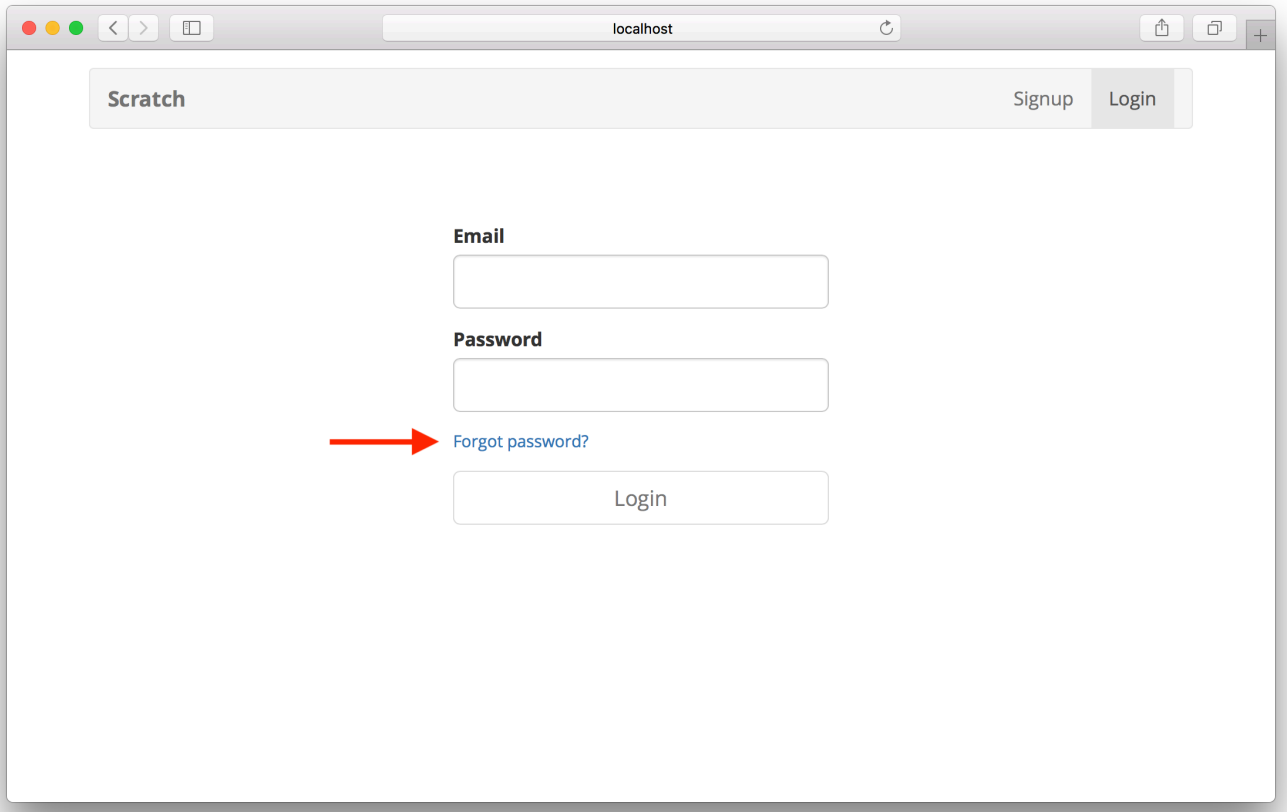
CHANGE So let's add a link in our `src/containers/Login.js`. Add it above our login button.

```
<Link to="/login/reset">Forgot password?</Link>
```

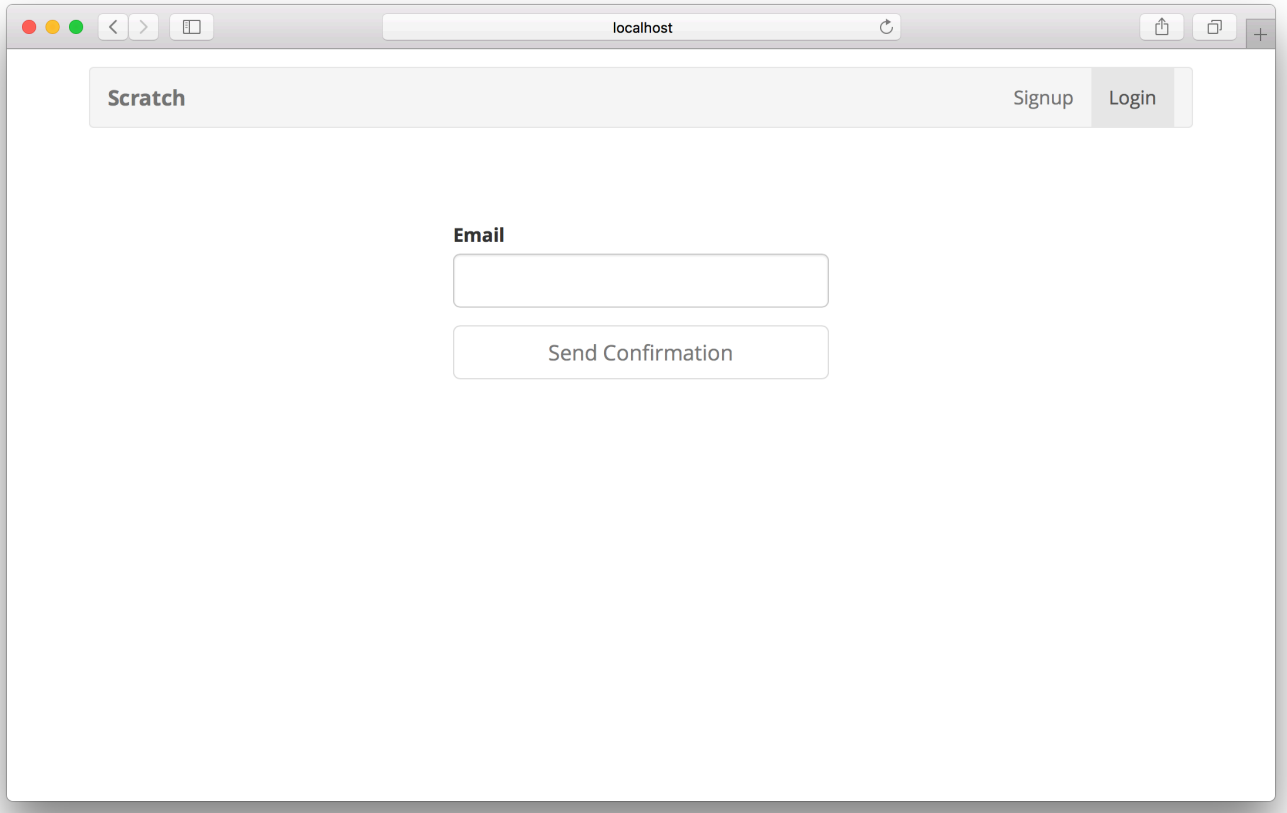
CHANGE And import the `Link` component in the header.

```
import { Link } from "react-router-dom";
```

That's it! We should now be able to navigate to `/login/reset` or go to it from the login page in case we need to reset our password.



And from there they can put in their email to reset their password.



Next, let's look at how our logged in users can change their password.

CHECKPOINT

For reference, here is the code we have so far.

`User Management Frontend Source (https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client)`

For help and discussion

`Comments on this chapter (https://discourse.serverless-stack.com/t/handle-forgot-and-reset-password/506)`

Allow Users to Change Passwords

For our Serverless notes app (https://demo.serverless-stack.com), we want to allow our users to change their password. Recall that we are using Cognito to manage our users and AWS Amplify in our React app. In this chapter we will look at how to do that.

For reference, we are using a forked version of the notes app with:

- A separate GitHub repository: **https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client** (https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client)
- And it can be accessed through: **https://demo-user-mgmt.serverless-stack.com** (https://demo-user-mgmt.serverless-stack.com)

Let's start by creating a settings page that our users can use to change their password.

Add a Settings Page

✦ CHANGE Add the following to `src/containers/Settings.js`.

```
import React, { Component } from "react";
import { LinkContainer } from "react-router-bootstrap";
import LoaderButton from "../components/LoaderButton";
```

```
import "./Settings.css";

export default class Settings extends Component {
  constructor(props) {
    super(props);

    this.state = {
    };
  }

  render() {
    return (
      <div className="Settings">
        <LinkContainer to="/settings/email">
          <LoaderButton
            block
            bsSize="large"
            text="Change Email"
          />
        </LinkContainer>
        <LinkContainer to="/settings/password">
          <LoaderButton
            block
            bsSize="large"
            text="Change Password"
          />
        </LinkContainer>
      </div>
    );
  }
}
```

All this does is add two links to a page that allows our users to change their password and email.

✦ CHANGE Let's also add a couple of styles for this page.

```
@media all and (min-width: 480px) {
  .Settings {
    padding: 60px 0;
    margin: 0 auto;
    max-width: 320px;
  }
}
.Settings .LoaderButton:last-child {
  margin-top: 15px;
}
```

✦ CHANGE Add a link to this settings page to the navbar of our app by changing `src/App.js`.

```
<Navbar fluid collapseOnSelect>
  <Navbar.Header>
    <Navbar.Brand>
      <Link to="/">Scratch</Link>
    </Navbar.Brand>
    <Navbar.Toggle />
  </Navbar.Header>
  <Navbar.Collapse>
    <Nav pullRight>
      {this.state.isAuthenticated
        ? <Fragment>
            <LinkContainer to="/settings">
```

```
              <NavItem>Settings</NavItem>
            </LinkContainer>
            <NavItem onClick=
{this.handleLogout}>Logout</NavItem>
          </Fragment>
        : <Fragment>
            <LinkContainer to="/signup">
              <NavItem>Signup</NavItem>
            </LinkContainer>
            <LinkContainer to="/login">
              <NavItem>Login</NavItem>
            </LinkContainer>
          </Fragment>
      }
    </Nav>
  </Navbar.Collapse>
</Navbar>
```

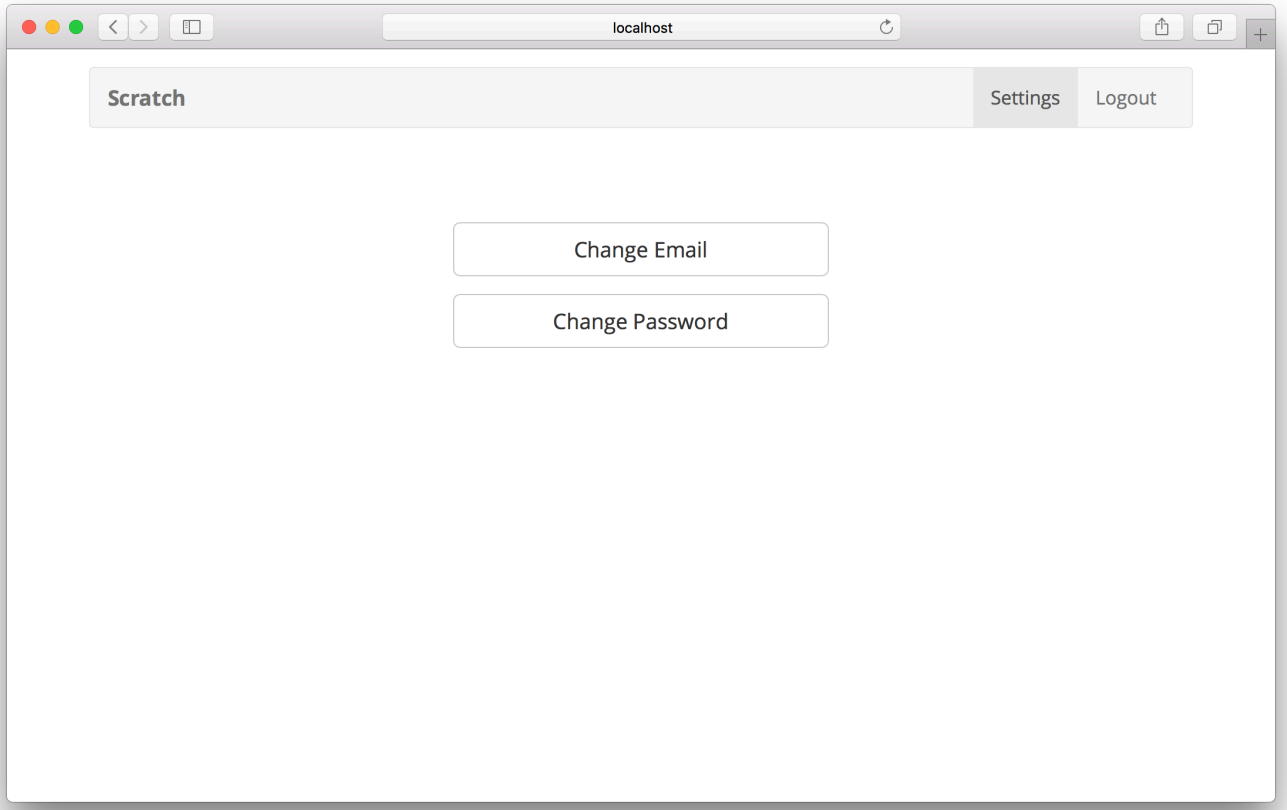
✦ CHANGE Also, add the route to our `src/Routes.js`.

```
<AuthenticatedRoute
  path="/settings"
  exact
  component={Settings}
  props={childProps}
/>
```

✦ CHANGE And don't forget to import it.

```
import Settings from "./containers/Settings";
```

This should give us a settings page that our users can get to from the app navbar.



Change Password Form

Now let's create the form that allows our users to change their password.

✦ CHANGE Add the following to `src/containers/ChangePassword.js`.

```
import React, { Component } from "react";
import { Auth } from "aws-amplify";
import { FormGroup, FormControl, ControlLabel } from
"react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import "./ChangePassword.css";
```

```
export default class ChangePassword extends Component {
  constructor(props) {
    super(props);

    this.state = {
      password: "",
      oldPassword: "",
      isChanging: false,
      confirmPassword: ""
    };
  }

  validateForm() {
    return (
      this.state.oldPassword.length > 0 &&
      this.state.password.length > 0 &&
      this.state.password ===
this.state.confirmPassword
    );
  }

  handleChange = event => {
    this.setState({
      [event.target.id]: event.target.value
    });
  };

  handleChangeClick = async event => {
    event.preventDefault();
```

```
    this.setState({ isChanging: true });

    try {
      const currentUser = await
Auth.currentAuthenticatedUser();
      await Auth.changePassword(
        currentUser,
        this.state.oldPassword,
        this.state.password
      );

      this.props.history.push("/settings");
    } catch (e) {
      alert(e.message);
      this.setState({ isChanging: false });
    }
  };

  render() {
    return (
      <div className="ChangePassword">
        <form onSubmit={this.handleChangeClick}>
          <FormGroup bsSize="large"
controlId="oldPassword">
            <ControlLabel>Old Password</ControlLabel>
            <FormControl
              type="password"
              onChange={this.handleChange}
              value={this.state.oldPassword}
            />
          </FormGroup>
```

```
          <hr />
          <FormGroup bsSize="large"
controlId="password">
            <ControlLabel>New Password</ControlLabel>
            <FormControl
              type="password"
              value={this.state.password}
              onChange={this.handleChange}
            />
          </FormGroup>
          <FormGroup bsSize="large"
controlId="confirmPassword">
            <ControlLabel>Confirm
Password</ControlLabel>
            <FormControl
              type="password"
              onChange={this.handleChange}
              value={this.state.confirmPassword}
            />
          </FormGroup>
          <LoaderButton
            block
            type="submit"
            bsSize="large"
            text="Change Password"
            loadingText="Changing…"
            disabled={!this.validateForm()}
            isLoading={this.state.isChanging}
          />
        </form>
      </div>
```

```
    );
  }
}
```

Most of this should be very straightforward. The key part of the flow here is that we ask the user for their current password along with their new password. Once they enter it, we can call the following:

```
const currentUser = await
Auth.currentAuthenticatedUser();
await Auth.changePassword(
  currentUser,
  this.state.oldPassword,
  this.state.password
);
```

The above snippet uses the `Auth` module from Amplify to get the current user. And then uses that to change their password by passing in the old and new password. Once the `Auth.changePassword` method completes, we redirect the user to the settings page.

✦ CHANGE Let's also add a couple of styles.

```
@media all and (min-width: 480px) {
  .ChangePassword {
    padding: 60px 0;
  }

  .ChangePassword form {
    margin: 0 auto;
    max-width: 320px;
```

```
  }
}
```

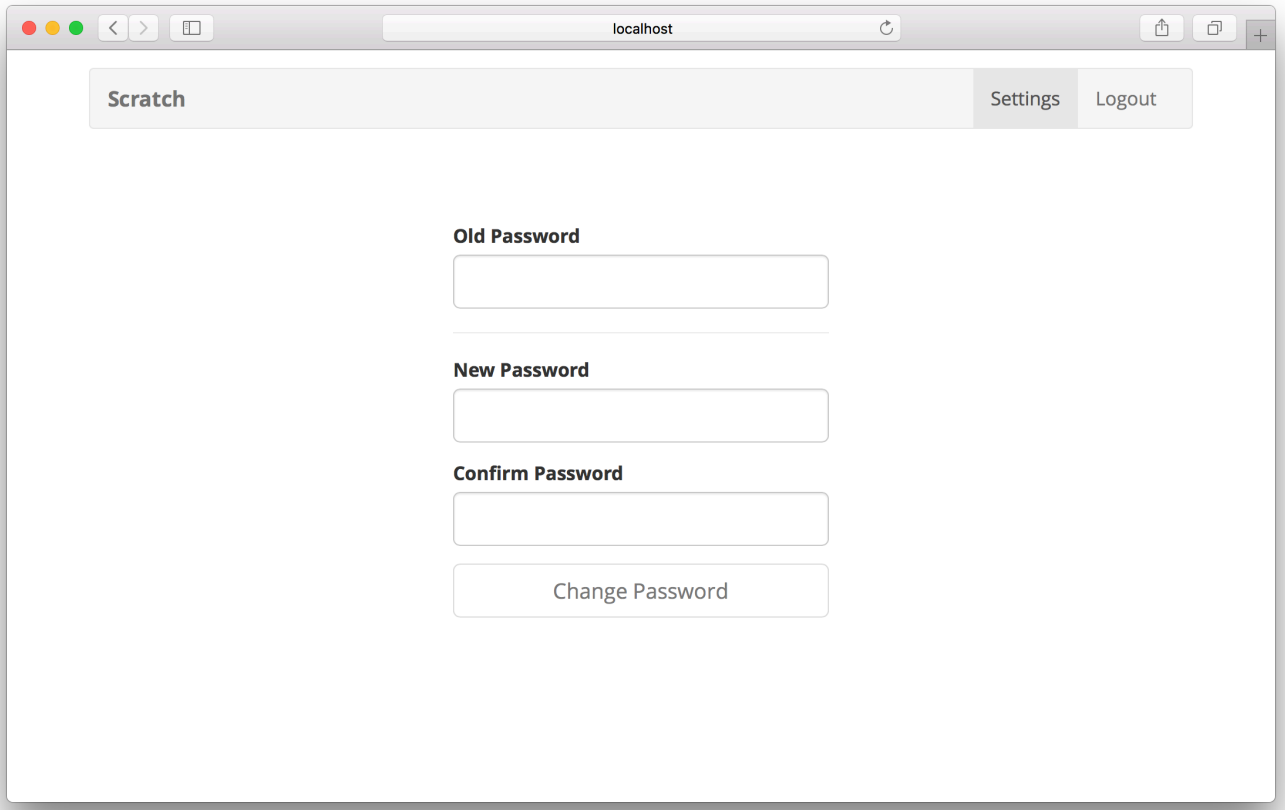
✦ CHANGE Let's add our new page to `src/Routes.js`.

```
<AuthenticatedRoute
  path="/settings/password"
  exact
  component={ChangePassword}
  props={childProps}
/>
```

✦ CHANGE And import it.

```
import ChangePassword from
"./containers/ChangePassword";
```

That should do it. The `/settings/password` page should allow us to change our password.



Next, let's look at how to implement a change email form for our users.

CHECKPOINT

For reference, here is the code we have so far.

`User Management Frontend Source (https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client)`

For help and discussion

`Comments on this chapter (https://discourse.serverless-`

stack.com/t/allow-users-to-change-passwords/507)

Allow Users to Change Their Email

We want the users of our Serverless notes app (https://demo.serverless-stack.com) to be able to change their email. Recall that we are using Cognito to manage our users and AWS Amplify in our React app. In this chapter we will look at how to do that.

For reference, we are using a forked version of the notes app with:

- A separate GitHub repository: **https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client** (https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client)
- And it can be accessed through: **https://demo-user-mgmt.serverless-stack.com** (https://demo-user-mgmt.serverless-stack.com)

In the previous chapter we created a settings page that links to `/settings/email`. Let's implement that.

Change Email Form

✦ CHANGE Add the following to `src/containers/ChangeEmail.js`.

```
import React, { Component } from "react";
import { Auth } from "aws-amplify";
import {
```

```
  HelpBlock,
  FormGroup,
  FormControl,
  ControlLabel
} from "react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import "./ChangeEmail.css";

export default class ChangeEmail extends Component {
  constructor(props) {
    super(props);

    this.state = {
      code: "",
      email: "",
      codeSent: false,
      isConfirming: false,
      isSendingCode: false
    };
  }

  validatEmailForm() {
    return this.state.email.length > 0;
  }

  validateConfirmForm() {
    return this.state.code.length > 0;
  }

  handleChange = event => {
    this.setState({
```

```
      [event.target.id]: event.target.value
    });
  };

  handleUpdateClick = async event => {
    event.preventDefault();

    this.setState({ isSendingCode: true });

    try {
      const user = await
Auth.currentAuthenticatedUser();
      await Auth.updateUserAttributes(user, { email:
this.state.email });

      this.setState({ codeSent: true });
    } catch (e) {
      alert(e.message);
      this.setState({ isSendingCode: false });
    }
  };

  handleConfirmClick = async event => {
    event.preventDefault();

    this.setState({ isConfirming: true });

    try {
      await
Auth.verifyCurrentUserAttributeSubmit("email",
this.state.code);
```

```
      this.props.history.push("/settings");
    } catch (e) {
      alert(e.message);
      this.setState({ isConfirming: false });
    }
  };

  renderUpdateForm() {
    return (
      <form onSubmit={this.handleUpdateClick}>
        <FormGroup bsSize="large" controlId="email">
          <ControlLabel>Email</ControlLabel>
          <FormControl
            autoFocus
            type="email"
            value={this.state.email}
            onChange={this.handleChange}
          />
        </FormGroup>
        <LoaderButton
          block
          type="submit"
          bsSize="large"
          text="Update Email"
          loadingText="Updating…"
          disabled={!this.validatEmailForm()}
          isLoading={this.state.isSendingCode}
        />
      </form>
    );
```

```
  }

  renderConfirmationForm() {
    return (
      <form onSubmit={this.handleConfirmClick}>
        <FormGroup bsSize="large" controlId="code">
          <ControlLabel>Confirmation
Code</ControlLabel>
          <FormControl
            autoFocus
            type="tel"
            value={this.state.code}
            onChange={this.handleChange}
          />
          <HelpBlock>
            Please check your email
({this.state.email}) for the confirmation
            code.
          </HelpBlock>
        </FormGroup>
        <LoaderButton
          block
          type="submit"
          bsSize="large"
          text="Confirm"
          loadingText="Confirm…"
          isLoading={this.state.isConfirming}
          disabled={!this.validateConfirmForm()}
        />
      </form>
    );
```

```
  }

  render() {
    return (
      <div className="ChangeEmail">
        {!this.state.codeSent
          ? this.renderUpdateForm()
          : this.renderConfirmationForm()}
      </div>
    );
  }
}
```

The flow for changing a user's email is pretty similar to how we sign a user up.

1. We ask a user to put in their new email.
2. Cognito sends them a verification code.
3. They enter the code and we confirm that their email has been changed.

We start by rendering a form that asks our user to enter their new email in `this.renderUpdateForm()`. Once the user submits this form, we call:

```
const user = await Auth.currentAuthenticatedUser();
Auth.updateUserAttributes(user, { email:
this.state.email });
```

This gets the current user and updates their email using the `Auth` module from Amplify. Next we render the form where they can enter the code in `this.renderConfirmationForm()`. Upon submitting this form we call:

```
Auth.verifyCurrentUserAttributeSubmit("email",
this.state.code);
```

This confirms the change on Cognito's side. Finally, we redirect the user to the settings page.

✦ CHANGE Let's add a couple of styles to `src/containers/ChangeEmail.css`.

```
@media all and (min-width: 480px) {
  .ChangeEmail {
    padding: 60px 0;
  }

  .ChangeEmail form {
    margin: 0 auto;
    max-width: 320px;
  }
}
```

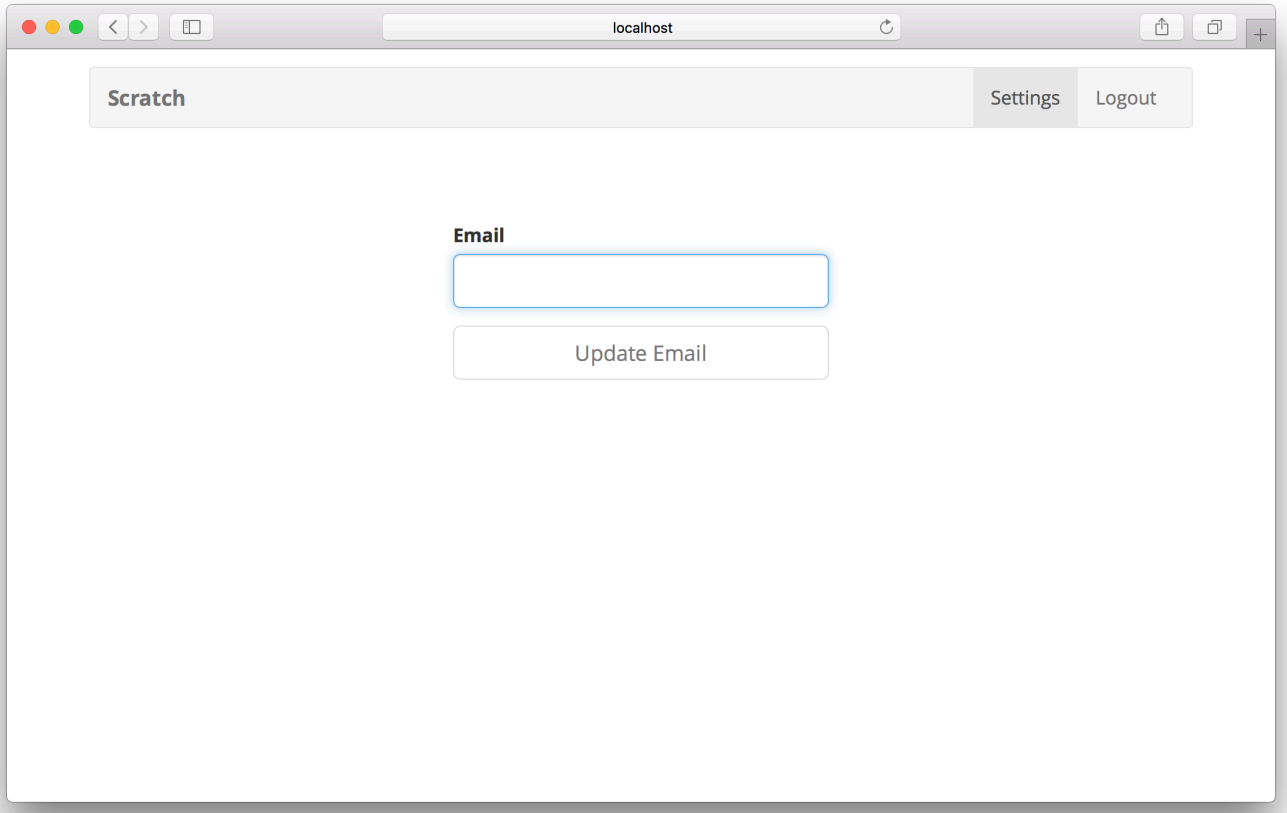
✦ CHANGE Finally, let's add our new page to `src/Routes.js`.

```
<AuthenticatedRoute
  path="/settings/email"
  exact
  component={ChangeEmail}
  props={childProps}
/>
```

✦ CHANGE And import it in the header.

```
import ChangeEmail from "./containers/ChangeEmail";
```

That should do it. Our users should now be able to change their email.



Finer Details

You might notice that the change email flow is interrupted if the user does not confirm the new email. In this case, the email appears to have been changed but Cognito marks it as not being verified. We will let you handle this case on your own but here are a couple of hints on how to do so.

- You can get the current user's Cognito attributes by calling `Auth.userAttributes(currentUser)`. Looking for the email attribute and checking if it is not verified using `attributes["email_verified"] !== "false"`.

- In this case show a simple sign that allows users to resend the verification code. You can do this by calling `Auth.verifyCurrentUserAttribute("email")`.
- Next you can simply display the confirm code form from above and follow the same flow by calling `Auth.verifyCurrentUserAttributeSubmit("email", this.state.code)`.

This can make your change email flow more robust and handle the case where a user forgets to verify their new email.

CHECKPOINT

For reference, here is the code we have so far.

User Management Frontend Source (https://github.com/AnomalyInnovations/serverless-stack-demo-user-mgmt-client)

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/allow-users-to-change-their-email/508)

Facebook Login with Cognito using AWS Amplify

In our guide so far we have used the Cognito User Pool (https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools.html) to sign up users to our demo notes app (https://demo.serverless-stack.com). This means that our users have to sign up for an account with their email and password. But you might want your users to use their Facebook or Google account to sign up for your app. It also means that your users won't have to remember another email and password combination for the sites they use. In this chapter we will look at how to add a *"Login with Facebook"* option to our demo app.

The version of the notes app used in this chapter is hosted in :

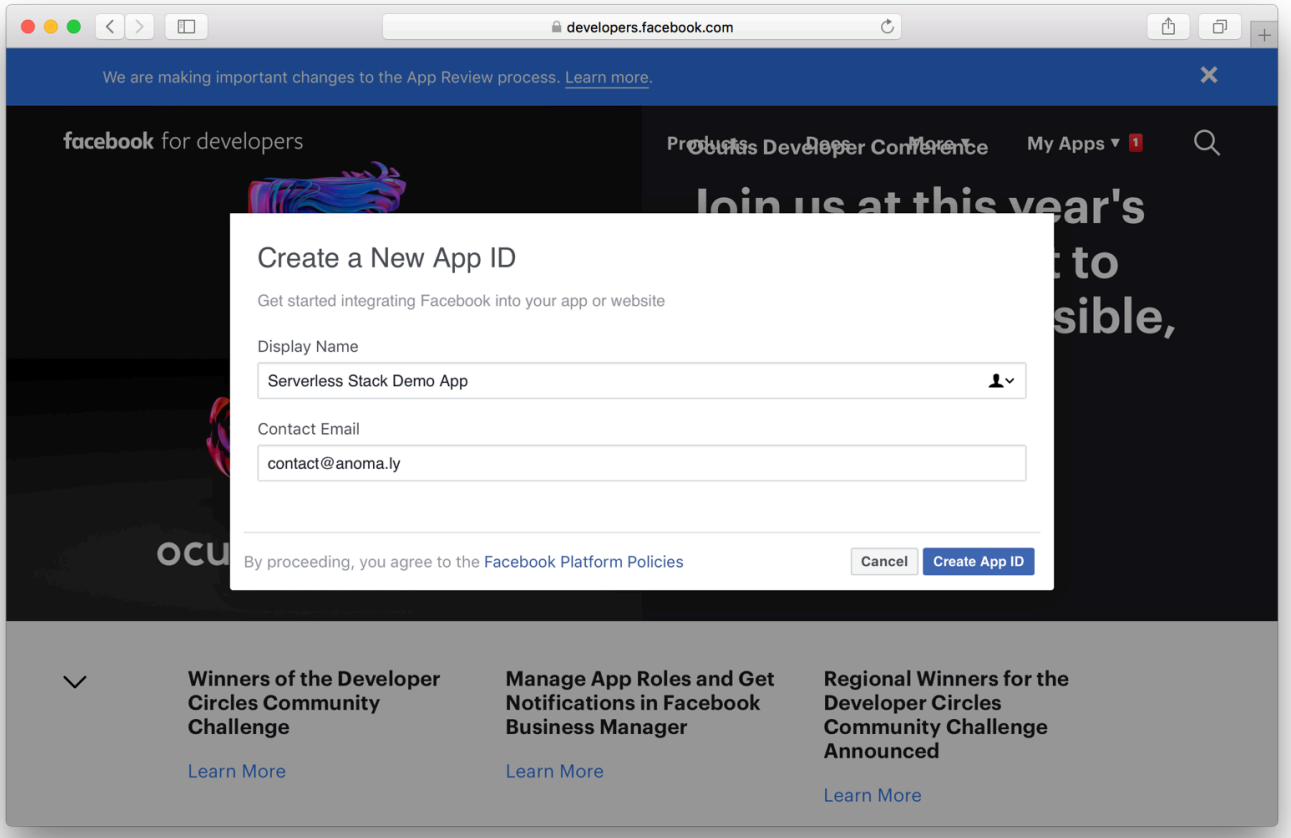
- A separate GitHub repository: **https://github.com/AnomalyInnovations/serverless-stack-demo-fb-login-client** (https://github.com/AnomalyInnovations/serverless-stack-demo-fb-login-client)
- And can be accessed through: **https://demo-fb-login.serverless-stack.com** (https://demo-fb-login.serverless-stack.com)

The main ideas and code for this chapter have been contributed by our long time reader and contributor Peter Eman Paver Abastillas (https://github.com/jatazoulja).

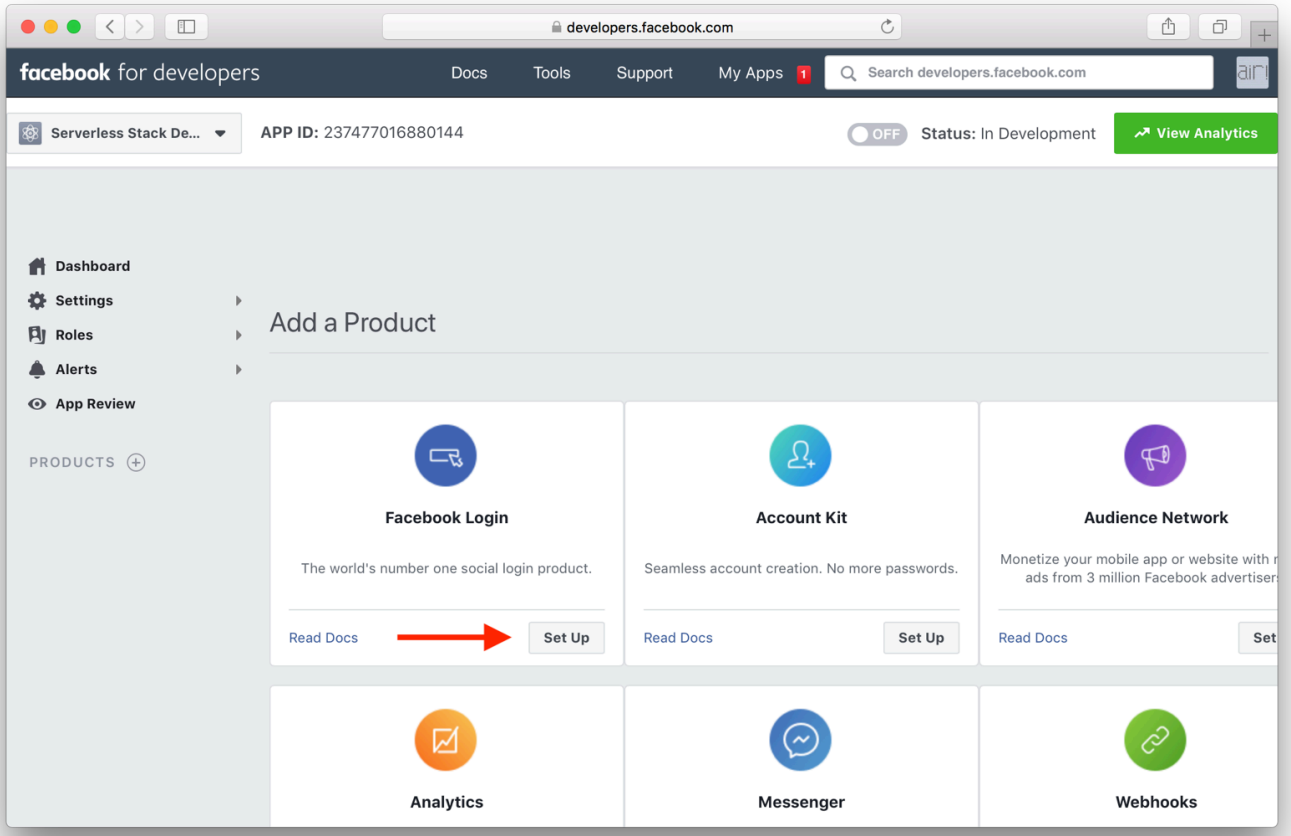
To get started let's create a Facebook app that our users will use to login.

Creating a Facebook App

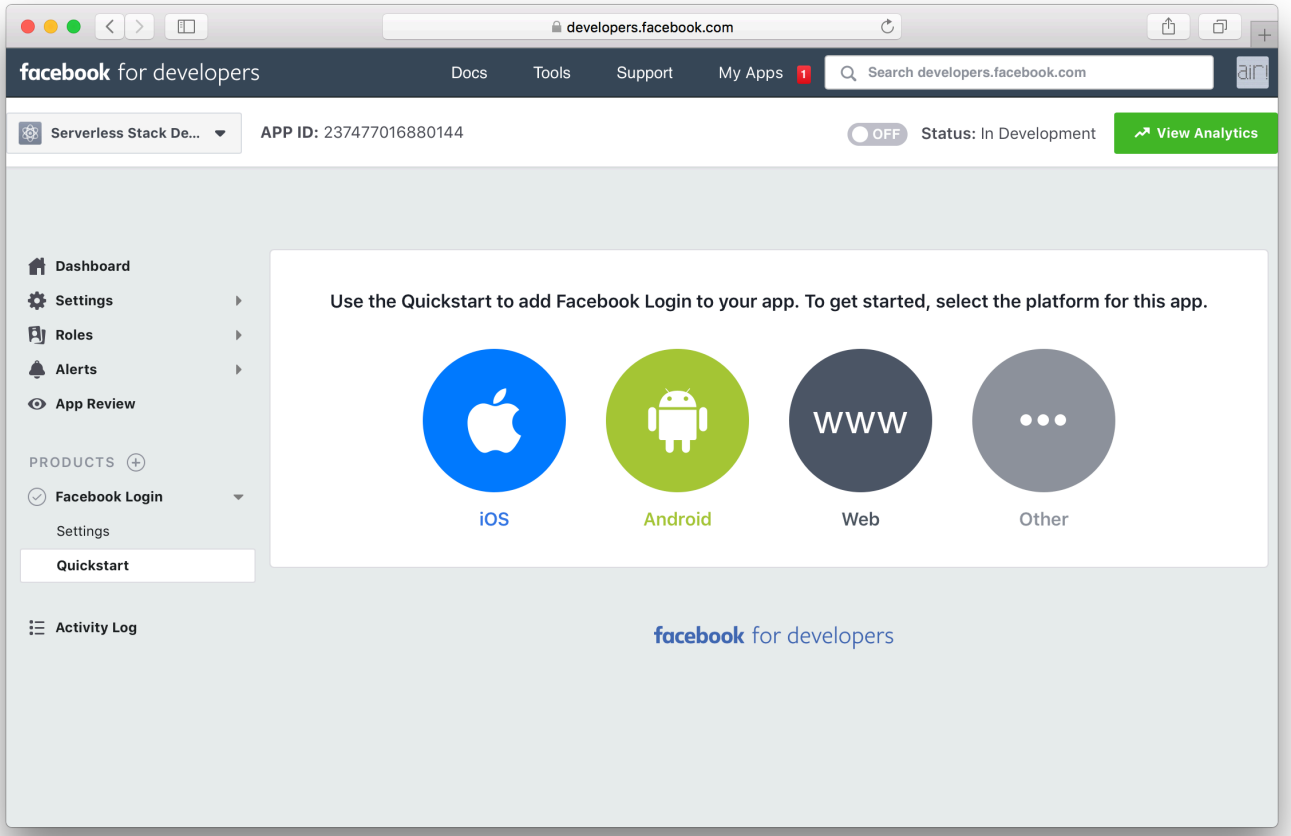
Head over to http://developers.facebook.com/ (http://developers.facebook.com/) and create a new app by clicking **My Apps** > **Add New App**.



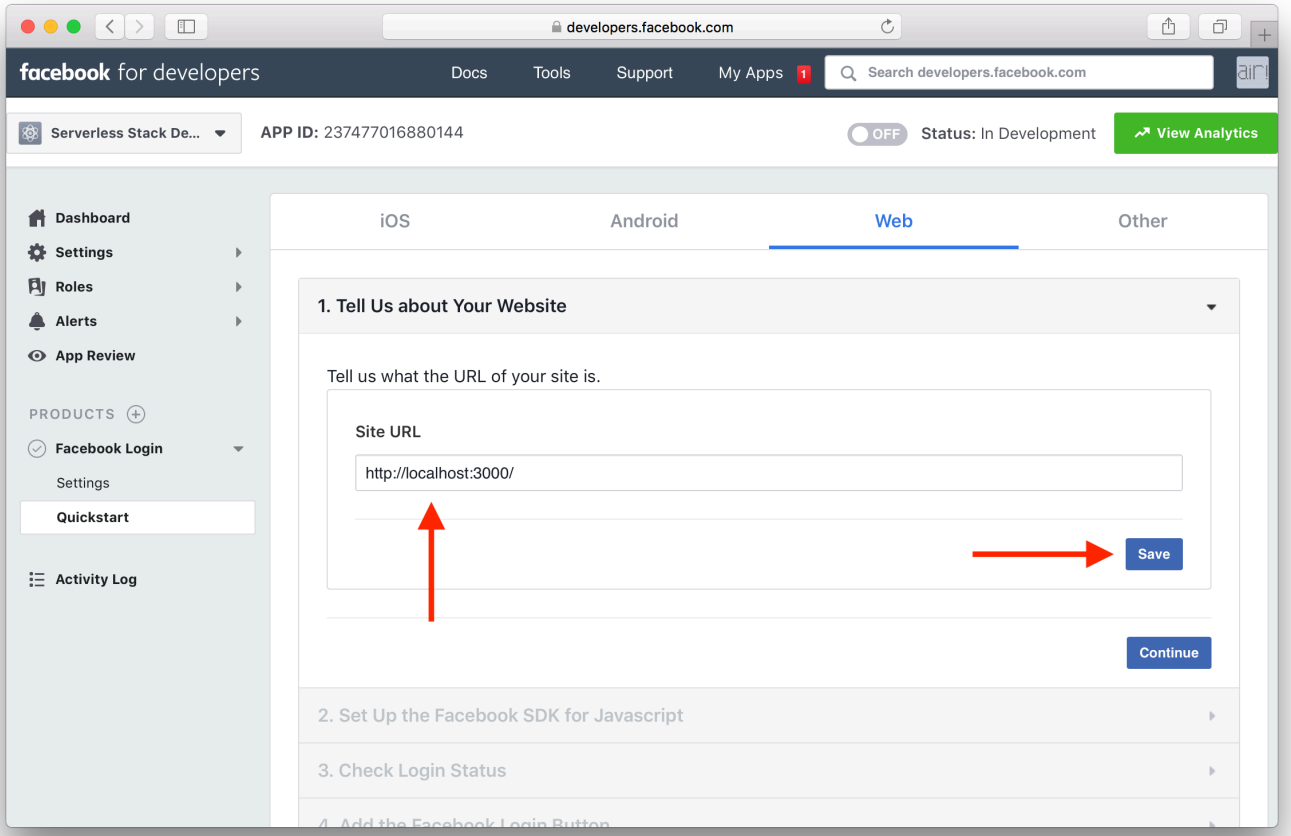
Under **Facebook Login**, select **Set Up**.



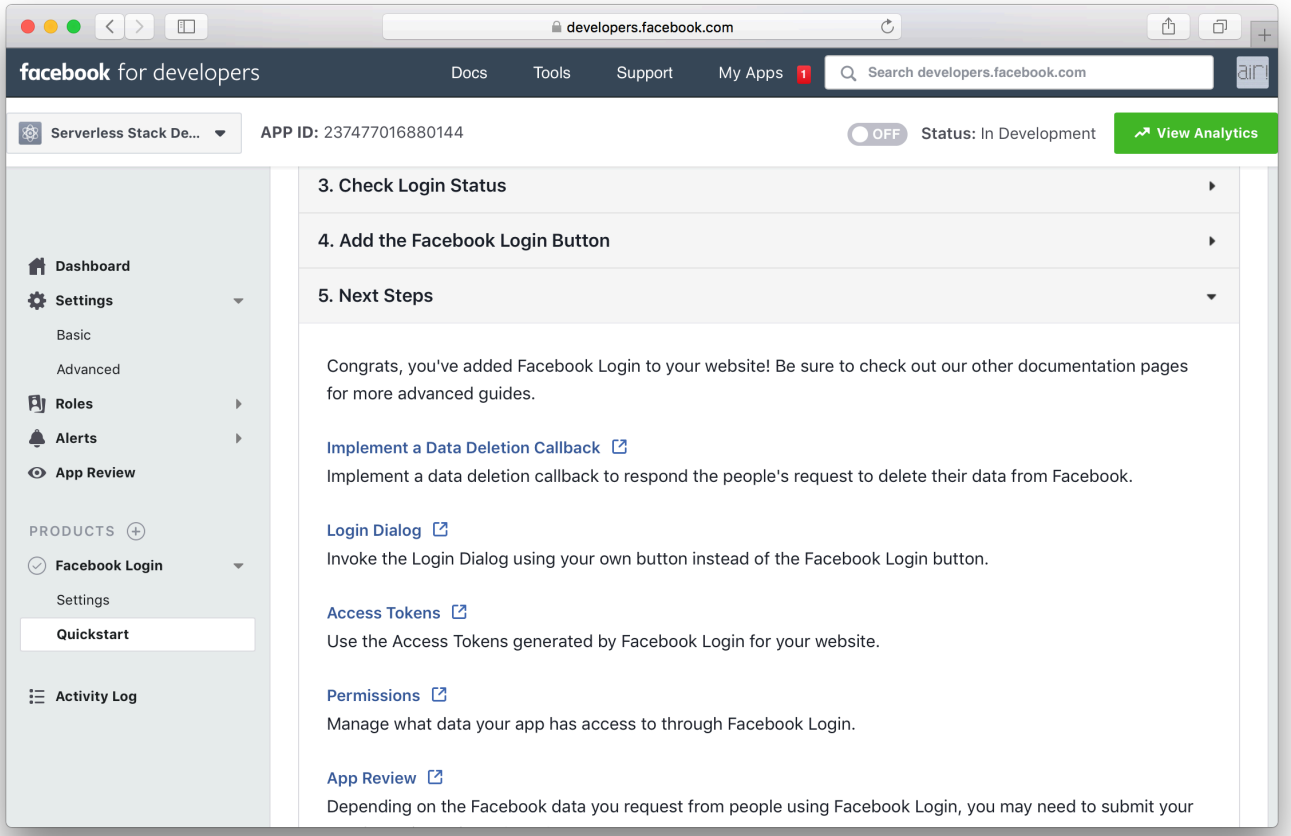
And select **Web**.



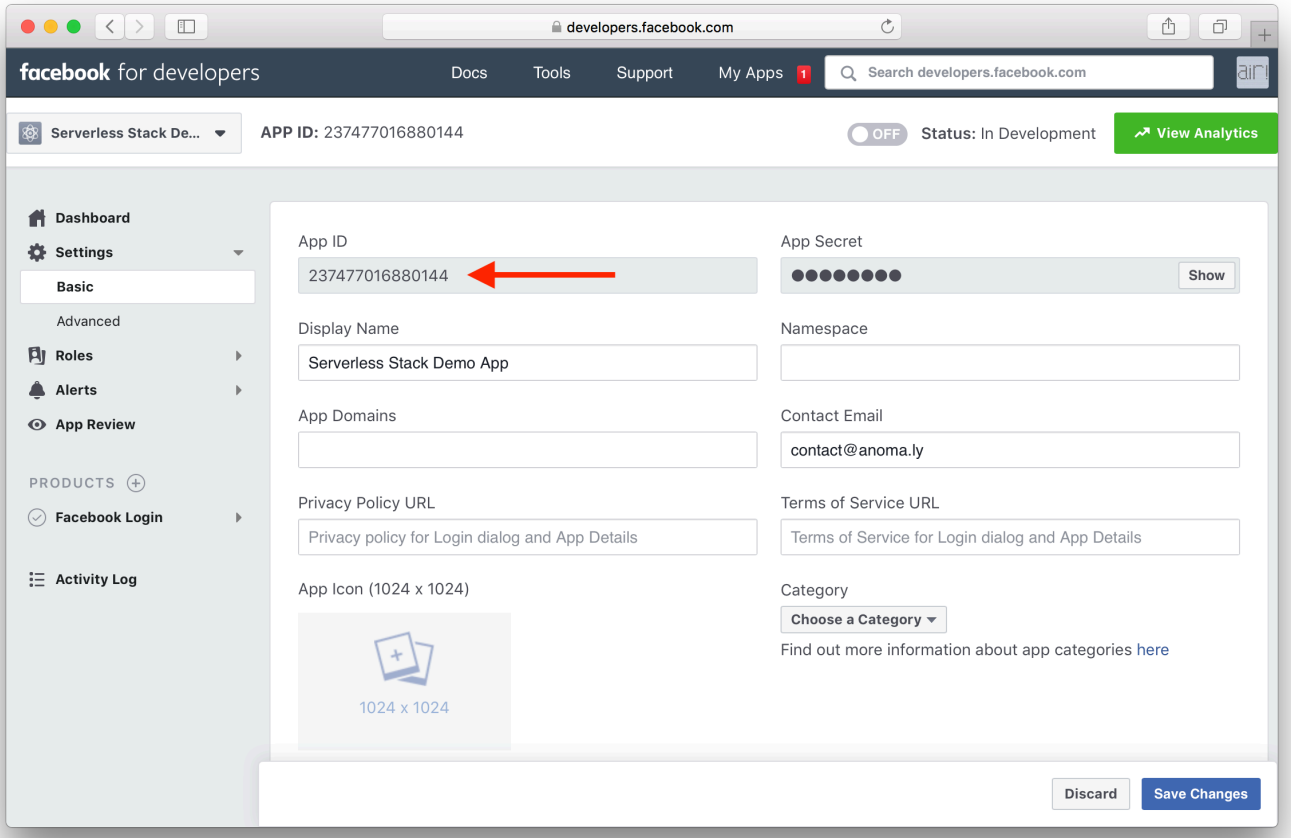
In the first step of the Quickstart, set the URL for your app to be `http://localhost:3000`. Or `https://localhost:3000` if you use the HTTPS option in Create React App (https://github.com/facebook/create-react-app/blob/master/packages/react-scripts/template/README.md#using-https-in-development). Hit **Save**.



You can hit **Continue** to go through the rest of the Quickstart.



Finally, head over to **Settings** > **Basic** and make a note of your **App ID**.



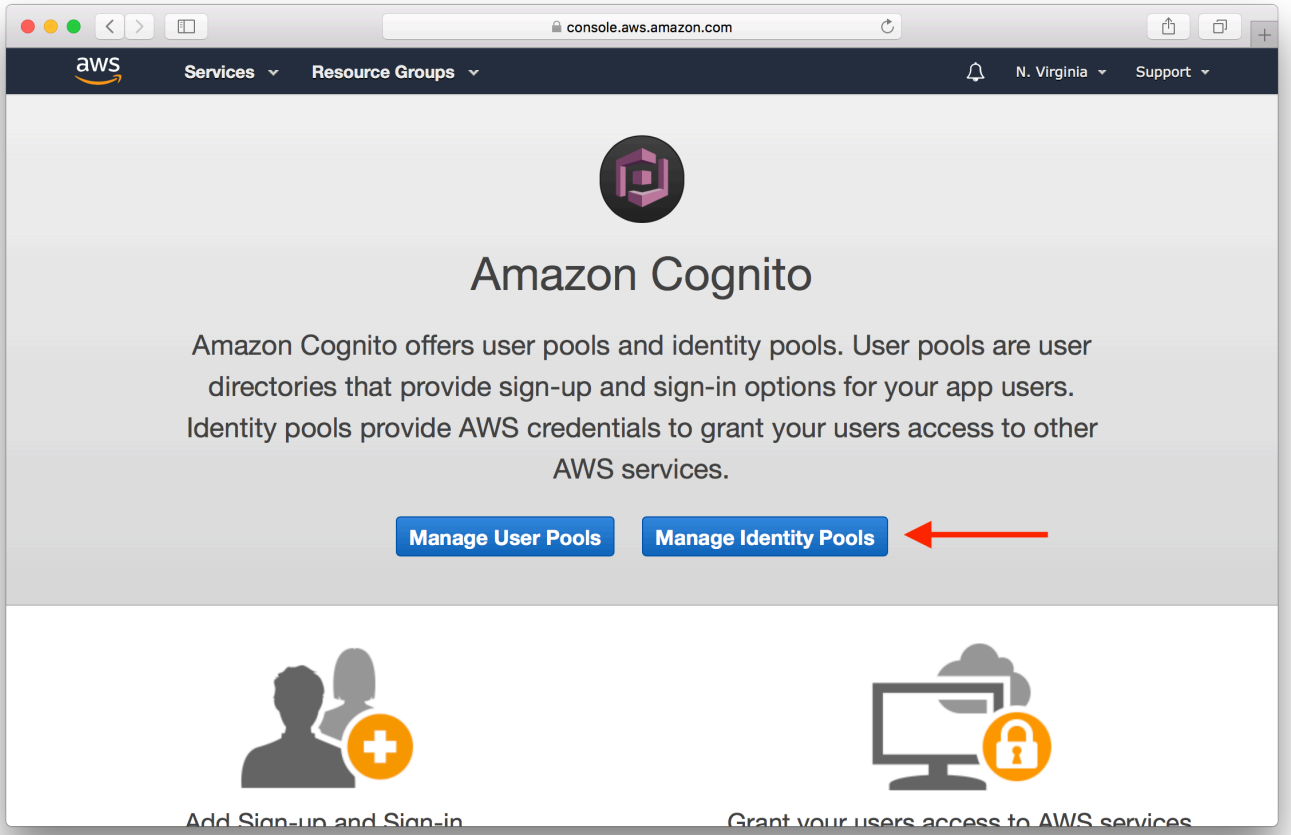
We are going to need this when we configure the AWS and React portion of our app.

Next we are going to use Cognito Identity Pool (https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-identity.html) to federate our identities. This means that when a user signs up with their Facebook account, they will get added to our Identity Pool. And our Serverless Backend API will get an Id that we can use. This Id will remain the same if the user signs in later at any point. If you are a little confused about how the Identity Pool is different from the User Pool, you can take a quick look at our Cognito user pool vs identity pool (/chapters/cognito-user-pool-vs-identity-pool.html) chapter.

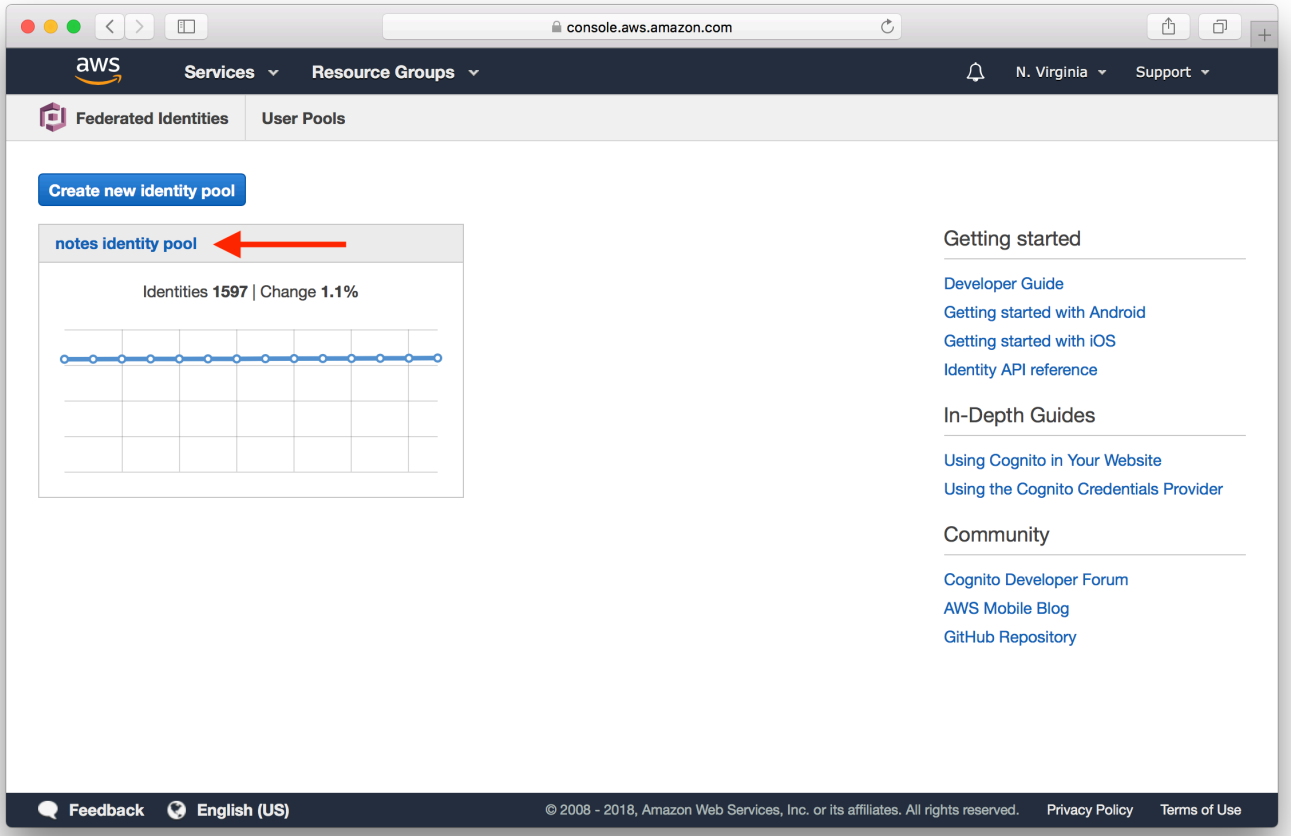
Add Facebook as an Authentication Provider

Head over to your AWS Console (https://console.aws.amazon.com), and go

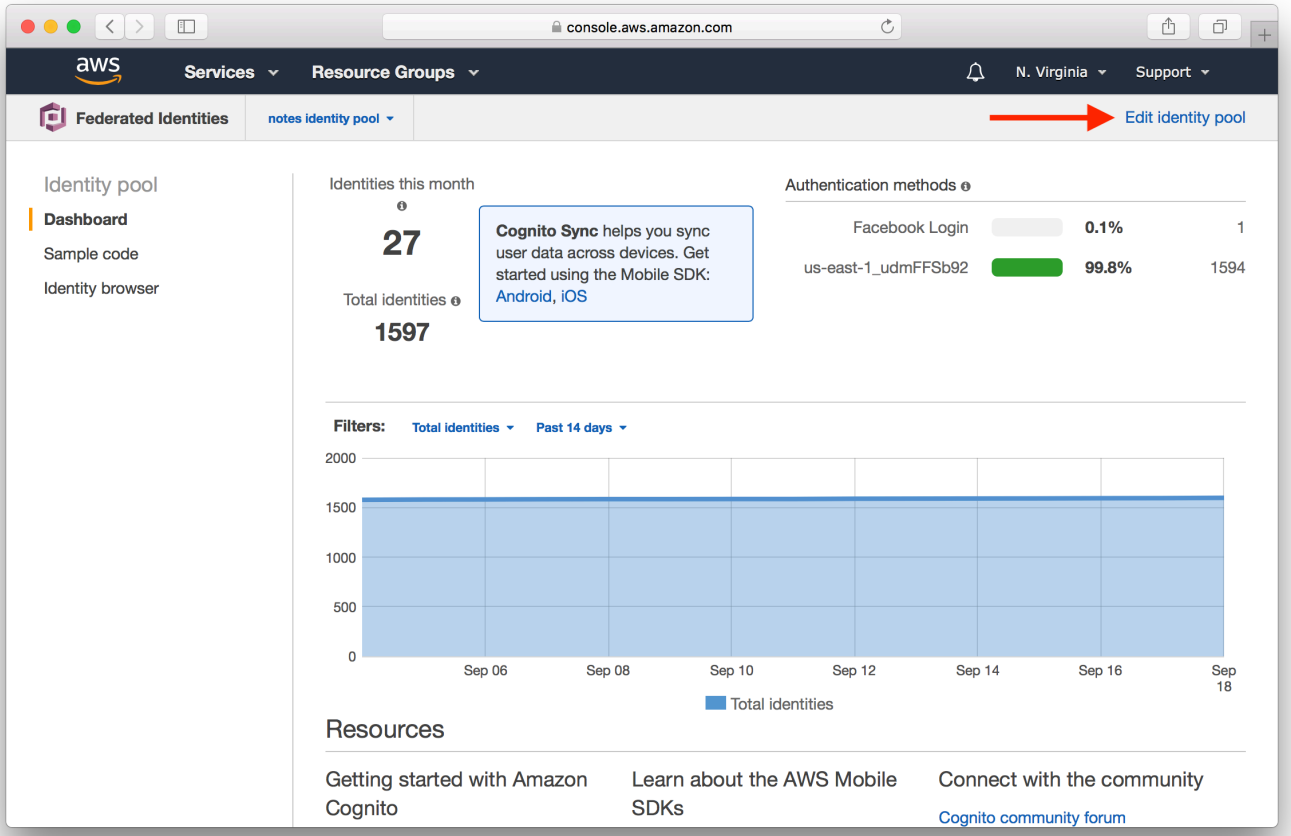
to **Cognito** and click **Manage Identity Pools**.



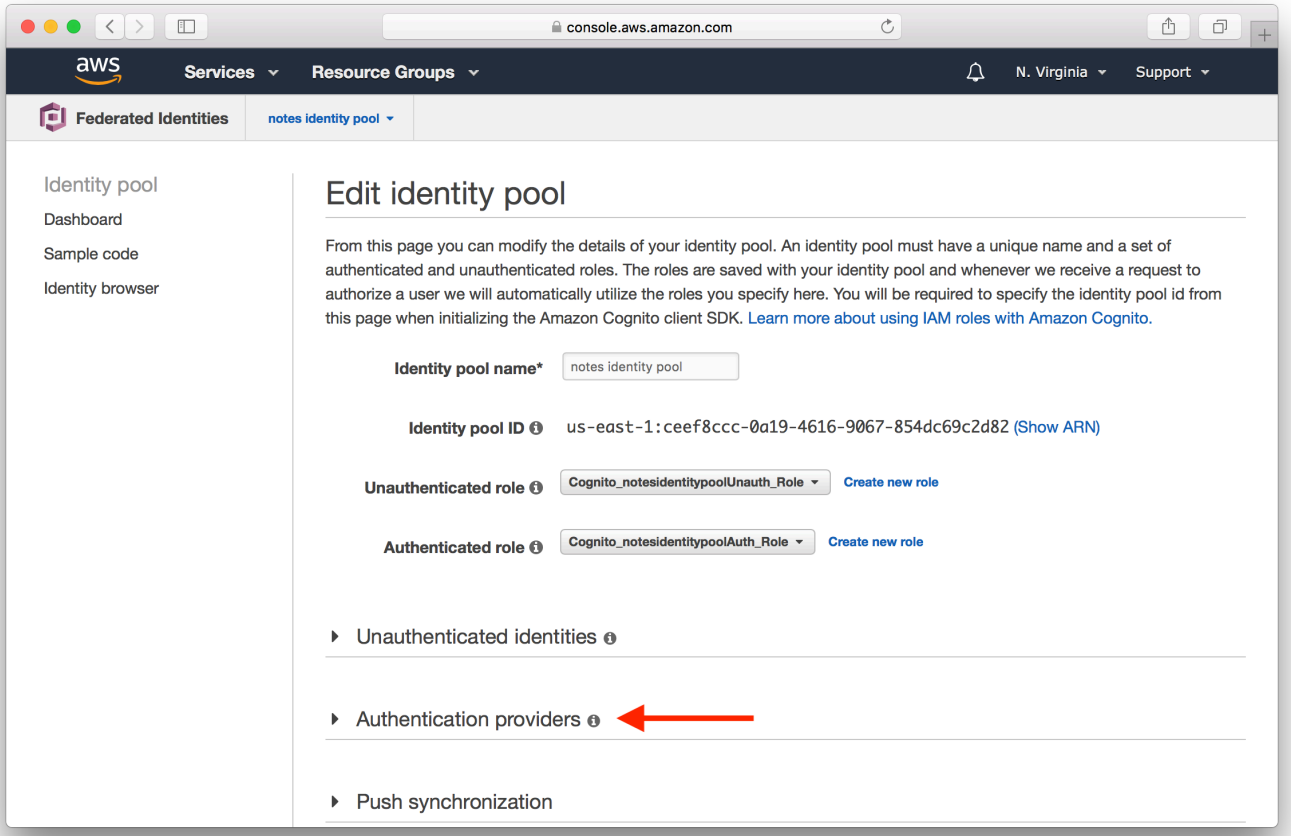
Select the Identity Pool that you are using for your app.



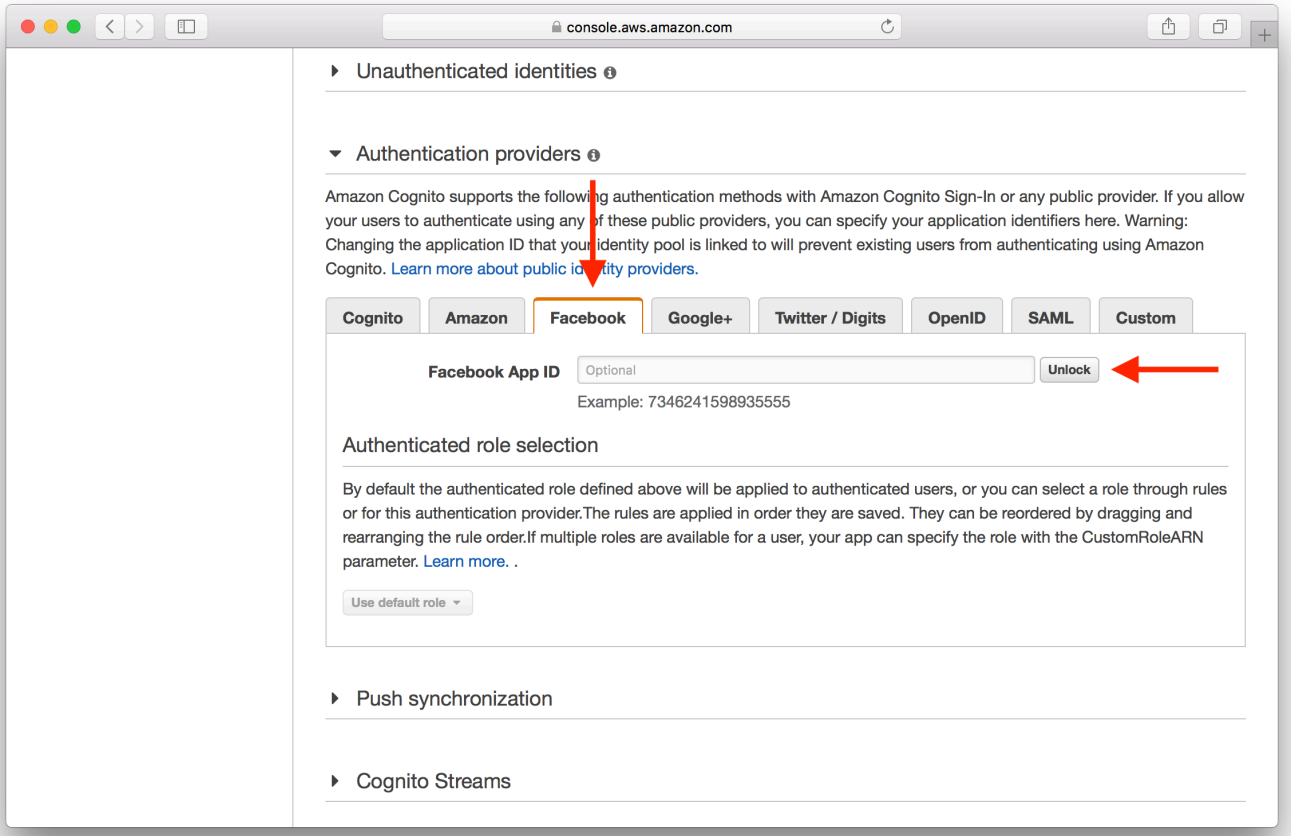
Hit **Edit identity pool** from the top.



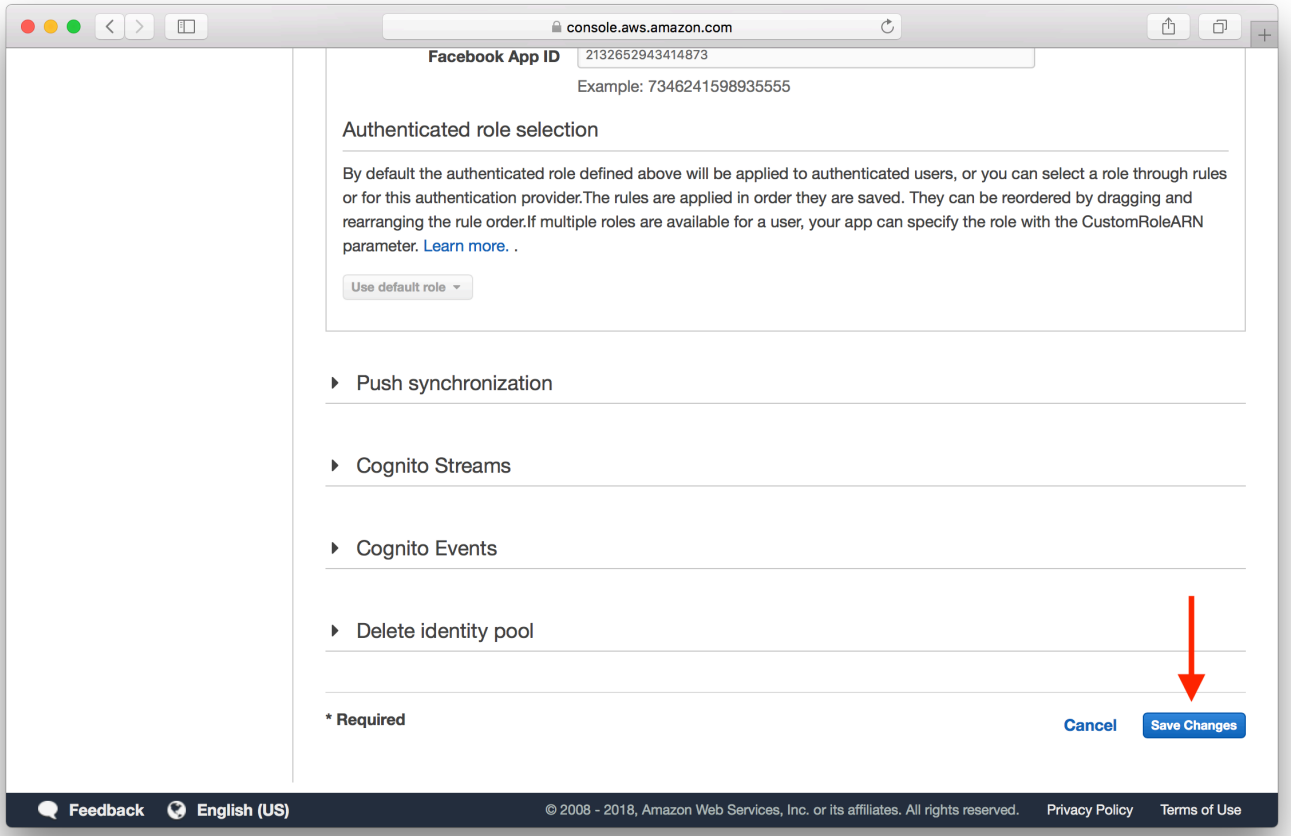
Scroll down and expand the **Authentication providers**.



You'll notice that you have **Cognito** as the default option. Select the **Facebook** tab. And Hit **Unlock** and paste your **Facebook App ID** from above.



And scroll down and hit **Save Changes**.



Now that we have the AWS side configured, let's head over to our React app.

Configure Facebook Login with AWS Amplify

In our React app we are going to use the Facebook JS SDK and AWS Amplify to configure our Facebook login. A working version of our app is available in the GitHub repo here (https://github.com/AnomalyInnovations/serverless-stack-demo-fb-login-client).

Let's take a quick look at the key changes that were made.

✦ CHANGE To start we add our Facebook App ID to our `src/config.js`. So it should look something like this.

```
export default {
  s3: {
    REGION: "YOUR_S3_UPLOADS_BUCKET_REGION",
    BUCKET: "YOUR_S3_UPLOADS_BUCKET_NAME"
  },
  apiGateway: {
    REGION: "YOUR_API_GATEWAY_REGION",
    URL: "YOUR_API_GATEWAY_URL"
  },
  cognito: {
    REGION: "YOUR_COGNITO_REGION",
    USER_POOL_ID: "YOUR_COGNITO_USER_POOL_ID",
    APP_CLIENT_ID: "YOUR_COGNITO_APP_CLIENT_ID",
    IDENTITY_POOL_ID: "YOUR_IDENTITY_POOL_ID"
  },
  social: {
    FB: "YOUR_FACEBOOK_APP_ID"
  }
};
```

Make sure to replace `YOUR_FACEBOOK_APP_ID` with the one from above.

✦ CHANGE Next we load the Facebook JS SDK in the our `src/App.js` in the `componentDidMount` method.

```
async componentDidMount() {
  this.loadFacebookSDK();

  try {
    await Auth.currentAuthenticatedUser();
    this.userHasAuthenticated(true);
```

```
  } catch (e) {
    if (e !== "not authenticated") {
      alert(e);
    }
  }

  this.setState({ isAuthenticating: false });
}

loadFacebookSDK() {
  window.fbAsyncInit = function() {
    window.FB.init({
      appId            : config.social.FB,
      autoLogAppEvents : true,
      xfbml            : true,
      version          : 'v3.1'
    });
  };

  (function(d, s, id){
     var js, fjs = d.getElementsByTagName(s)[0];
     if (d.getElementById(id)) {return;}
     js = d.createElement(s); js.id = id;
     js.src =
"https://connect.facebook.net/en_US/sdk.js";
     fjs.parentNode.insertBefore(js, fjs);
   }(document, 'script', 'facebook-jssdk'));
}
```

And we also load the current authenticated user using the `Auth.currentAuthenticatedUser` method. Where `Auth` is a part of

the AWS Amplify package.

✦ CHANGE Make sure to import the config at the top of `src/App.js`.

```
import config from "./config";
```

✦ CHANGE Now we'll create a Facebook login button component in `src/components/FacebookButton.js`.

```
import React, { Component } from "react";
import { Auth } from "aws-amplify";
import LoaderButton from "./LoaderButton";

function waitForInit() {
  return new Promise((res, rej) => {
    const hasFbLoaded = () => {
      if (window.FB) {
        res();
      } else {
        setTimeout(hasFbLoaded, 300);
      }
    };
    hasFbLoaded();
  });
}

export default class FacebookButton extends Component {
  constructor(props) {
    super(props);

    this.state = {
```

```
      isLoading: true
    };
  }

  async componentDidMount() {
    await waitForInit();
    this.setState({ isLoading: false });
  }

  statusChangeCallback = response => {
    if (response.status === "connected") {
      this.handleResponse(response.authResponse);
    } else {
      this.handleError(response);
    }
  };

  checkLoginState = () => {

window.FB.getLoginStatus(this.statusChangeCallback);
  };

  handleClick = () => {
    window.FB.login(this.checkLoginState, {scope:
"public_profile,email"});
  };

  handleError(error) {
    alert(error);
  }
```

```
  async handleResponse(data) {
    const { email, accessToken: token, expiresIn } =
data;
    const expires_at = expiresIn * 1000 + new
Date().getTime();
    const user = { email };

    this.setState({ isLoading: true });

    try {
      const response = await Auth.federatedSignIn(
        "facebook",
        { token, expires_at },
        user
      );
      this.setState({ isLoading: false });
      this.props.onLogin(response);
    } catch (e) {
      this.setState({ isLoading: false });
      this.handleError(e);
    }
  }

  render() {
    return (
      <LoaderButton
        block
        bsSize="large"
        bsStyle="primary"
        className="FacebookButton"
        text="Login with Facebook"
```

```
        onClick={this.handleClick}
        disabled={this.state.isLoading}
      />
    );
  }
}
```

Let's look at what we are doing here very quickly.

1. We first wait for the Facebook JS SDK to load in the `waitForInit` method. Once it has loaded, we enable the *Login with Facebook* button.
2. Once our user clicks the button, we kick off the login process using `FB.login` and listen for the login status to change in the `statusChangeCallback`. While calling this method, we are specifying that we want the user's public profile and email address by setting `{scope: "public_profile,email"}`.
3. If the user has given our app the permissions, then we use the information we receive from Facebook (the user's email) and call the `Auth.federatedSignIn` AWS Amplify method. This effectively logs the user in.

✦ CHANGE Finally, we can use the `FacebookButton.js` in our `src/containers/Login.js` and `src/containers/Signup.js`.

```
<FacebookButton
  onLogin={this.handleFbLogin}
/>
<hr />
```

Add the button above our login and signup form. And don't forget to

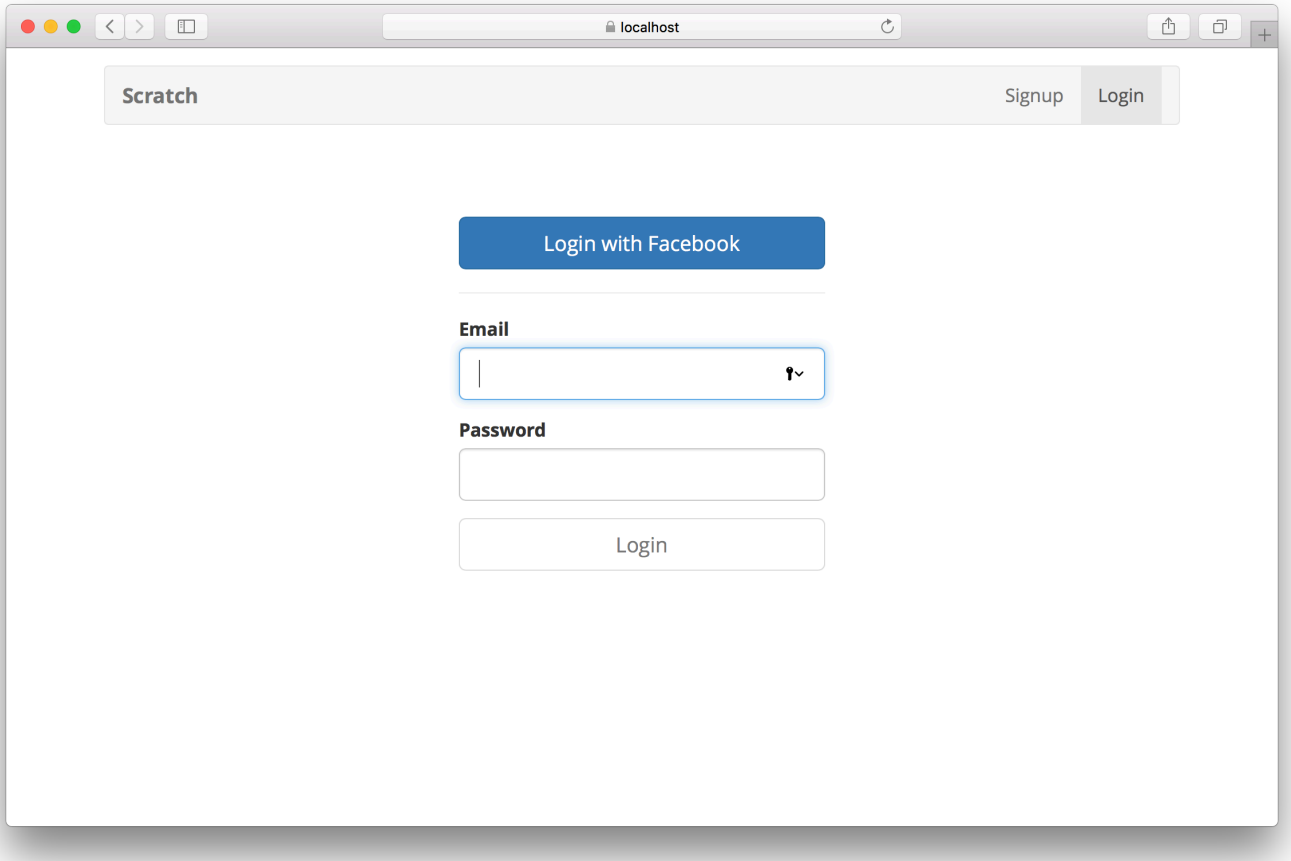
import it using `import FacebookButton from "../components/FacebookButton";`.

CHANGE Also, add the handler method as well.

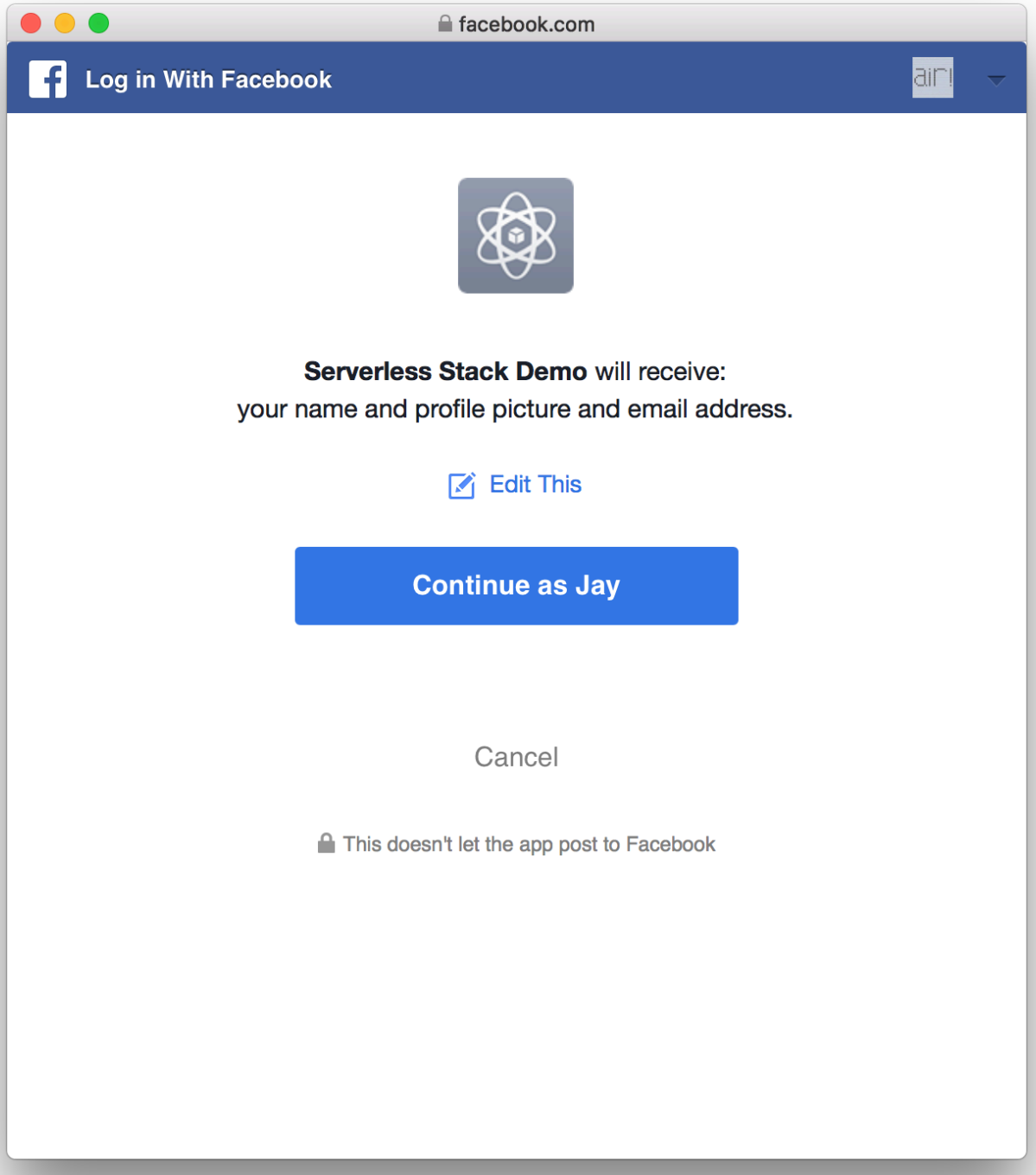
```
handleFbLogin = () => {
  this.props.userHasAuthenticated(true);
};
```

The above logs the user in to our React app, once the Facebook sign up process is complete. Make sure to add these to `src/containers/Signup.js` as well.

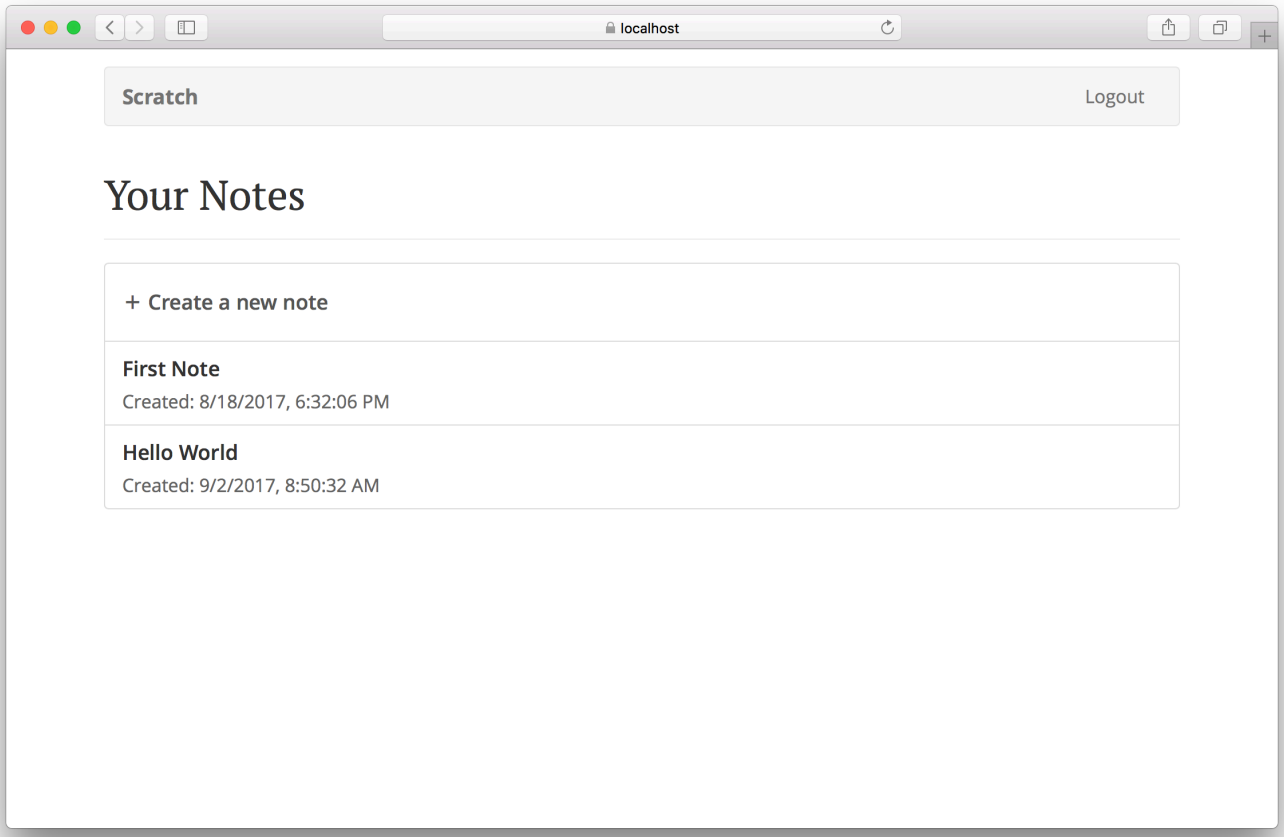
And that's it, if you head over to your app you should see the login with Facebook option.



Clicking on it should bring up the Facebook dialog asking you to login with your app.



Once you are logged in, you should be able to interact with the app just as before.



A final note on deploying your app. You might recall from above that we are telling Facebook to use the `https://localhost:3000` URL. This needs to be changed to the live URL once you deploy your React app. A good practice here is to create two Facebook apps, one for your live users and one for your local testing. That way you won't need to change the URL and you will have an environment where you can test your changes.

CHECKPOINT

For reference, here is the code we have so far.

`Facebook Login Frontend Source (https://github.com/AnomalyInnovations/serverless-stack-demo-fb-login-client)`

For help and discussion

Comments on this chapter (https://discourse.serverless-stack.com/t/facebook-login-with-cognito-using-aws-amplify/466)