



Serverless Stack

Open source guide to building full-stack apps using
Serverless and React

Anomaly Innovations

October 22, 2020 - v5.0.1

Contents

Preface	8
Who Is This Guide For?	9
What Does This Guide Cover?	10
How to Get Help?	15
The Basics	
Introduction	17
What is Serverless?	18
What is AWS Lambda?	21
Why Create Serverless Apps?	25
Set up your AWS account	27
Create an AWS Account.	28
Create an IAM User	29
What is IAM	40
What is an ARN	49
Configure the AWS CLI	51
Setting up the Serverless backend	53
Create a DynamoDB Table.	54
Create an S3 Bucket for File Uploads	62
Create a Cognito User Pool	71
Create a Cognito Test User.	88
Set up the Serverless Framework	90
Add Support for ES6 and TypeScript	93
Initialize the Backend Repo	95
Building a Serverless REST API	99
Add a Create Note API	100
Add a Get Note API	110
Add a List All the Notes API	113
Add an Update Note API	116

Add a Delete Note API	119
Working with 3rd Party APIs	122
Setup a Stripe Account	123
Add a Billing API.	127
Load Secrets from .env	130
Test the Billing API	132
Unit Tests in Serverless	134
Handle API Gateway CORS Errors.	137
Deploying the backend	140
Deploy the APIs	141
Create a Cognito Identity Pool	143
Cognito User Pool vs Identity Pool.	154
Test the APIs	158
Setting up a React app	162
Create a New React.js App	163
Initialize the Frontend Repo	165
Add App Favicons	169
Set up Custom Fonts	175
Set up Bootstrap.	178
Handle Routes with React Router	180
Create Containers	182
Adding Links in the Navbar.	188
Handle 404s	192
Configure AWS Amplify	195
Building a React app	199
Create a Login Page	200
Login with AWS Cognito.	204
Add the Session to the State	206
Load the State from the Session.	211
Clear the Session on Logout	216
Redirect on Login and Logout.	217
Give Feedback While Logging In.	220
Create a Custom React Hook to Handle Form Fields	226
Create a Signup Page	231
Create the Signup Form	232
Signup with AWS Cognito	238
Add the Create Note Page	242
Call the Create API.	247

Upload a File to S3	250
List All the Notes	253
Call the List API	256
Display a Note	260
Render the Note Form	264
Save Changes to a Note	269
Delete a Note	272
Create a Settings Page	274
Add Stripe Keys to Config	278
Create a Billing Form	280
Connect the Billing Form	284
Set up Secure Pages	289
Create a Route That Redirects	290
Use the Redirect Routes	293
Redirect on Login	296
Deploying the backend to production	299
Getting Production Ready	300
What Is Infrastructure as Code	303
What is AWS CDK?	306
Using AWS CDK with Serverless Framework	309
Building a CDK app with SST	314
Configure DynamoDB in CDK	316
Configure S3 in CDK	321
Configure Cognito User Pool in CDK	324
Configure Cognito Identity Pool in CDK	327
Connect Serverless Framework and CDK with SST	337
Deploy Your Serverless Infrastructure	343
Automating Serverless Deployments	347
Setting up Your Project on Seed	348
Configure Secrets in Seed	363
Deploying Through Seed	369
Set Custom Domains Through Seed	383
Test the Configured APIs	395
Deploying the frontend to production	399
Automating React Deployments	400
Manage Environments in Create React App	401
Create a Build Script	405
Setting up Your Project on Netlify	408

Custom Domains in Netlify.	415
Frontend Workflow	434
Monitoring and debugging errors	447
Debugging Full-Stack Serverless Apps	448
Setup Error Reporting in React	450
Report API Errors in React.	458
Setup an Error Boundary in React	460
Setup Error Logging in Serverless	467
Logic Errors in Lambda Functions	475
Unexpected Errors in Lambda Functions	485
Errors Outside Lambda Functions	493
Errors in API Gateway	501
Conclusion	511
Wrapping Up	512
Further Reading.	514
Translations	515
Giving Back	517
Changelog	519
Staying up to date	525

Best Practices

Introduction	526
Best Practices for Building Serverless Apps	527
Organize a Serverless app	531
Organizing Serverless Projects	532
Cross-Stack References in Serverless	539
Share Code Between Services	542
Share an API Endpoint Between Services	547
Deploy a Serverless App with Dependencies	552
Manage environments	555
Environments in Serverless Apps	556
Structure Environments Across AWS Accounts.	558
Manage AWS Accounts Using AWS Organizations	561
Parameterize Serverless Resources Names	571
Deploying to Multiple AWS Accounts	574
Deploy the Resources Repo	579
Deploy the API Services Repo	589

Manage Environment Related Config.	601
Storing Secrets in Serverless Apps	605
Share Route 53 Domains Across AWS Accounts	616
Monitor Usage for Environments	632
Development lifecycle	636
Working on Serverless Apps	637
Invoke Lambda Functions Locally	638
Invoke API Gateway Endpoints Locally	642
Creating Feature Environments	647
Creating Pull Request Environments	665
Promoting to Production	679
Rollback Changes	683
Deploying Only Updated Services	688
Observability	691
Tracing Serverless Apps with X-Ray	692
Conclusion	699
Wrapping up the Best Practices	700

Extra Credit

Serverless	701
API Gateway and Lambda Logs	702
Debugging Serverless API Issues	726
Serverless Environment Variables	747
Stages in Serverless Framework.	751
Backups in DynamoDB	755
Configure Multiple AWS Profiles	771
Customize the Serverless IAM Policy.	776
Mapping Cognito Identity Id and User Pool Id	786
Connect to API Gateway with IAM Auth	788
Serverless Node.js Starter	794
Package Lambdas with serverless-bundle	798
Using Lerna and Yarn Workspaces with Serverless	802
React	809
Understanding React Hooks	810
Code Splitting in Create React App	819
Environments in Create React App.	830
Deploy a React App to AWS	836

Create an S3 Bucket	837
Deploy to S3	848
Create a CloudFront Distribution	851
Purchase a Domain with Route 53	864
Set up SSL	870
Set up Your Domain with CloudFront.	877
Set up WWW Domain Redirect	892
Deploy Updates	906
Manage User Accounts in AWS Amplify.	911
Handle Forgot and Reset Password	912
Allow Users to Change Passwords	921
Allow Users to Change Their Email.	928
Facebook Login with Cognito using AWS Amplify	935

Preface

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](#). And [check out a sample of what folks have built with Serverless Stack](#).

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

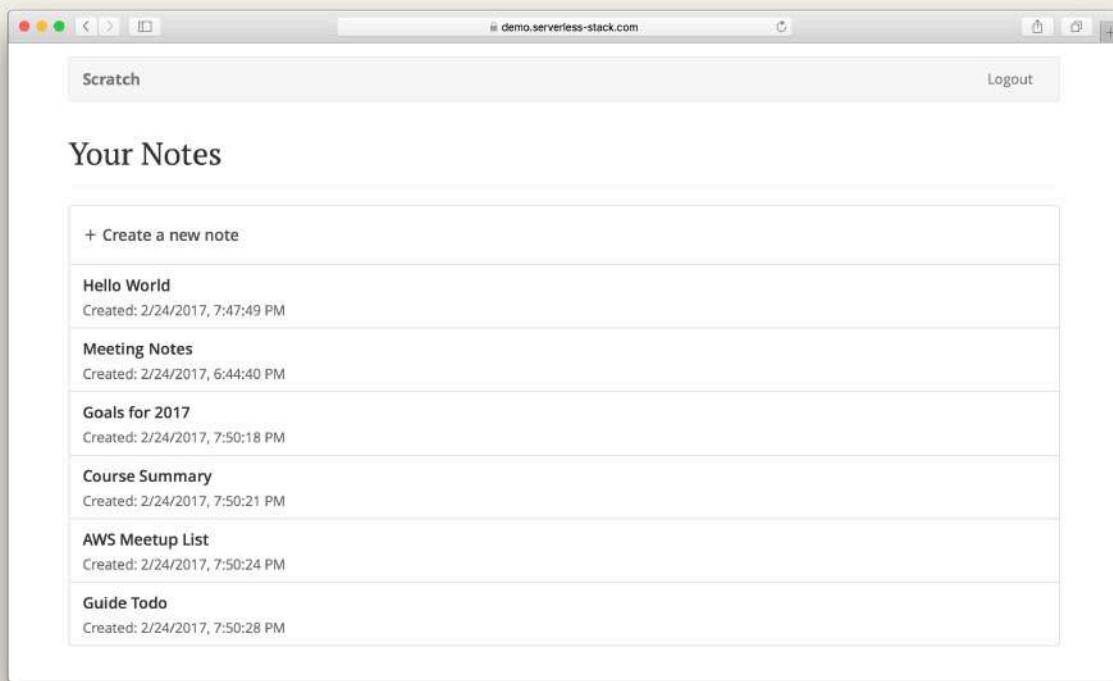


Help and discussion

View the [comments for this chapter on our forums](#)

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**. However, unlike most tutorials out there, our goal is to go into the details of what it takes to build a full-stack application for production.



Completed app desktop screenshot

It is a single page application powered by a serverless API written completely in JavaScript. Here is the complete source for the [backend](#) and the [frontend](#). 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
- The app should be deployed when we `git push`
- We should be able to monitor and debug any errors

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 & API Gateway](#) for our serverless API
- [DynamoDB](#) for our database
- [Cognito](#) for user authentication and securing our APIs
- [S3](#) for hosting our app and file uploads
- [CloudFront](#) for serving out our app
- [Route 53](#) for our domain
- [Certificate Manager](#) for SSL
- [CloudWatch](#) for Lambda and API access logs
- [React.js](#) for our single page app
- [React Router](#) for routing
- [Bootstrap](#) for the UI Kit
- [Stripe](#) for processing credit card payments
- [Seed](#) for automating Serverless deployments
- [Netlify](#) for automating React deployments
- [GitHub](#) for hosting our project repos
- [Sentry](#) for error reporting

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+ installed on your machine.
- A free [GitHub account](#).
- 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 deploy 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 topics.

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 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 and debugging Serverless apps:

- Set up error reporting in React using Sentry
- Configure an Error Boundary in React
- Add error logging to our Serverless APIs
- Cover the debugging workflow for common Serverless errors

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](#).



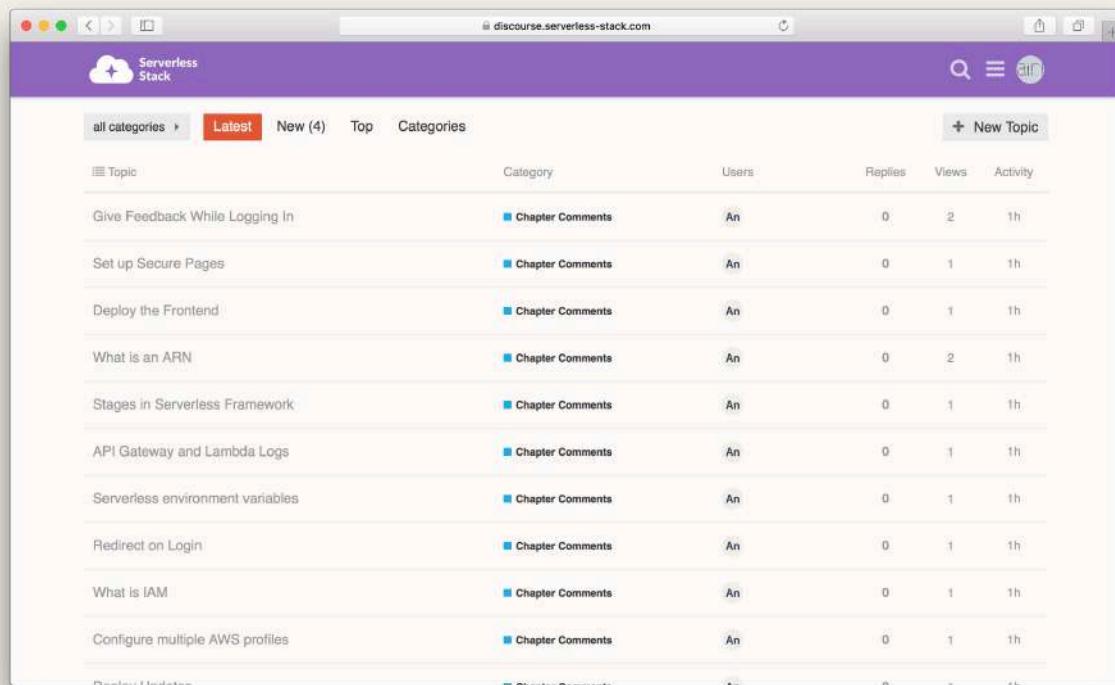
Help and discussion

View the [comments for this chapter on our forums](#)

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](#) 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.



The screenshot shows a web browser displaying the discourse.serverless-stack.com website. The page has a purple header with the "Serverless Stack" logo. Below the header, there's a navigation bar with links for "all categories", "Latest" (which is highlighted in red), "New (4)", "Top", and "Categories". On the right side of the navigation bar is a "+ New Topic" button. The main content area displays a table of topics. The columns are "Topic", "Category", "Users", "Replies", "Views", and "Activity". The topics listed are: "Give Feedback While Logging In" (Chapter Comments, 0 replies, 2 views, 1h activity), "Set up Secure Pages" (Chapter Comments, 0 replies, 1 view, 1h activity), "Deploy the Frontend" (Chapter Comments, 0 replies, 1 view, 1h activity), "What is an ARN" (Chapter Comments, 0 replies, 2 views, 1h activity), "Stages in Serverless Framework" (Chapter Comments, 0 replies, 1 view, 1h activity), "API Gateway and Lambda Logs" (Chapter Comments, 0 replies, 1 view, 1h activity), "Serverless environment variables" (Chapter Comments, 0 replies, 1 view, 1h activity), "Redirect on Login" (Chapter Comments, 0 replies, 1 view, 1h activity), "What is IAM" (Chapter Comments, 0 replies, 1 view, 1h activity), and "Configure multiple AWS profiles" (Chapter Comments, 0 replies, 1 view, 1h activity). At the bottom of the table, there's a link to "View all topics".

Serverless Stack Discourse Forums screenshot

This entire guide is hosted on [GitHub](#). So if you find an error you can always:

- Open a [new issue](#)

- Or if you've found a typo, edit the page and submit a pull request!



Help and discussion

View the [comments for this chapter on our forums](#)

Introduction

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](#)
- Microsoft Azure: [Azure Functions](#)
- Google Cloud: [Cloud 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 architected 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](#) 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](#) which we are going to be using in this tutorial has a few plugins to [help keep your functions warm](#).

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.



Help and discussion

View the [comments for this chapter on our forums](#)

What is AWS Lambda?

[AWS 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 12.13.0, 10.16.3, and 8.10
- Java 11 and 8
- Python 3.8, 3.7, 3.6, and 2.7
- .NET Core 2.1, 2.2, 3.0, and 3.1
- Go 1.x
- Ruby 2.5
- Rust

Note that, [.NET Core 2.2 and 3.0 are supported through custom runtimes](#).

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.

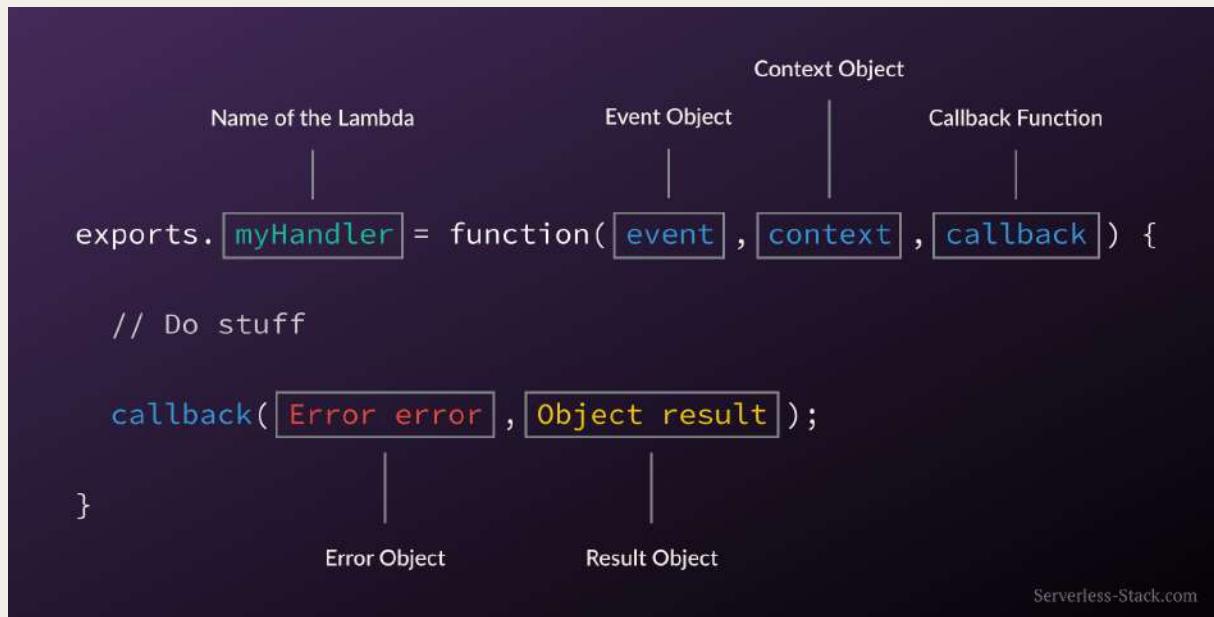
The ephemeral disk space is available in the form of the `/tmp` directory. You can only use this space for temporary storage since subsequent invocations will not have access to this. We'll talk a bit more on the stateless nature of the Lambda functions below.

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.



Anatomy of a Lambda Function image

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 an 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](#). 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](#).

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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

Service	Rate	Cost
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.



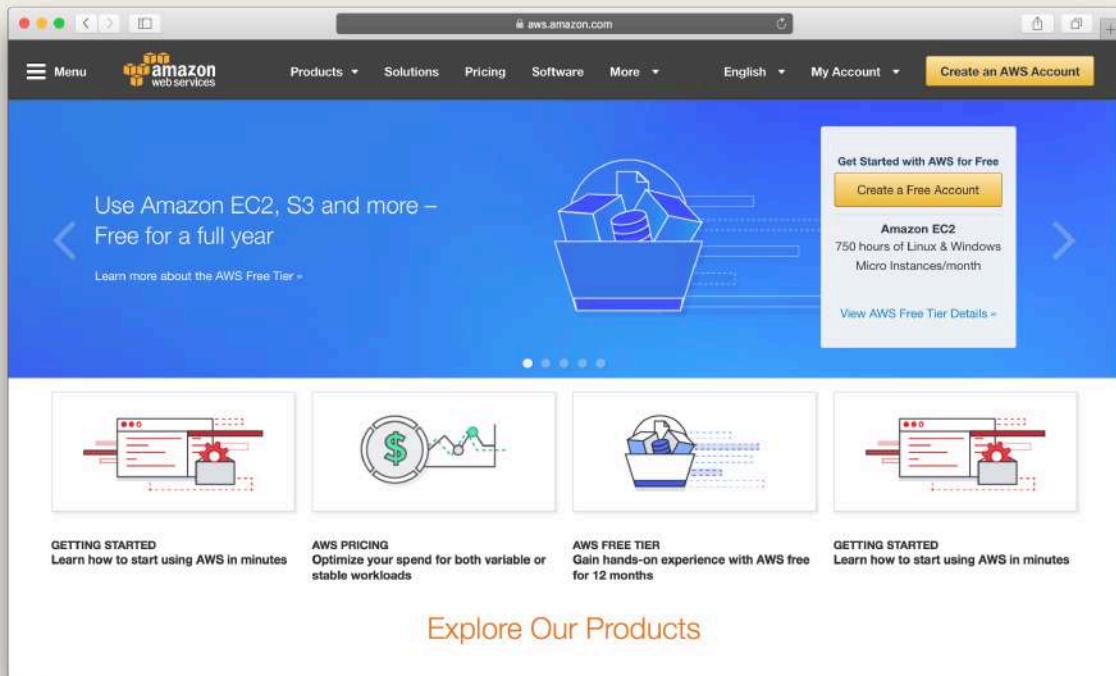
Help and discussion

View the [comments for this chapter on our forums](#)

Set up your AWS account

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](#) and hit the **Create a Free Account** and follow the steps to create your account.



Create an aws account Screenshot

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



Help and discussion

View the [comments for this chapter on our forums](#)

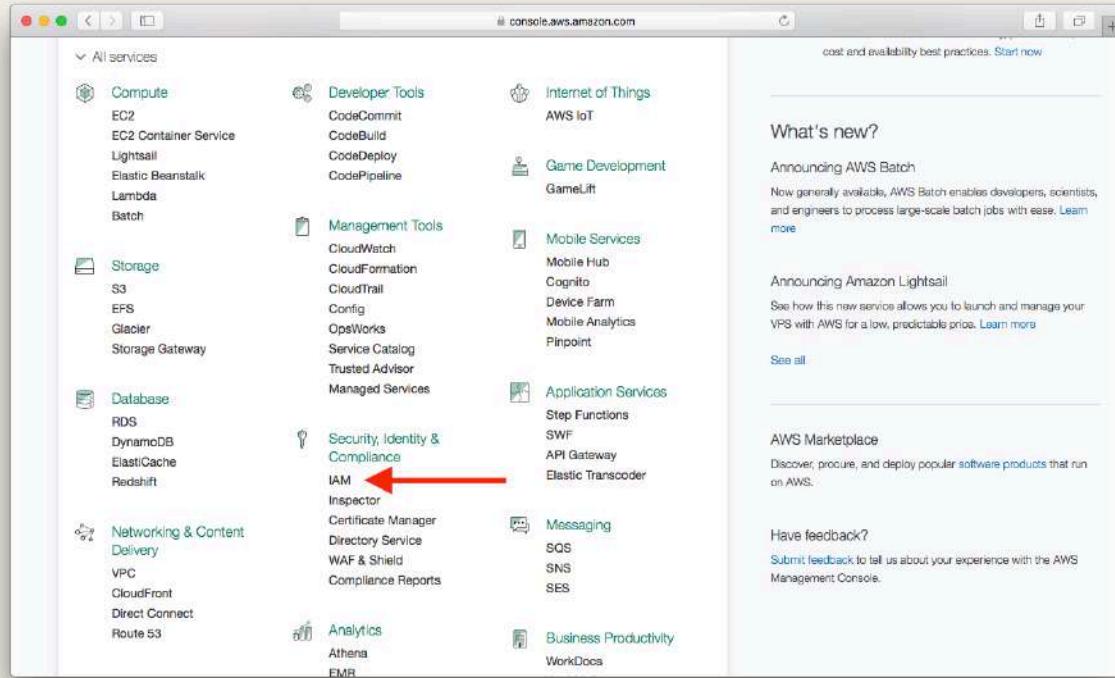
Create an IAM User

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.

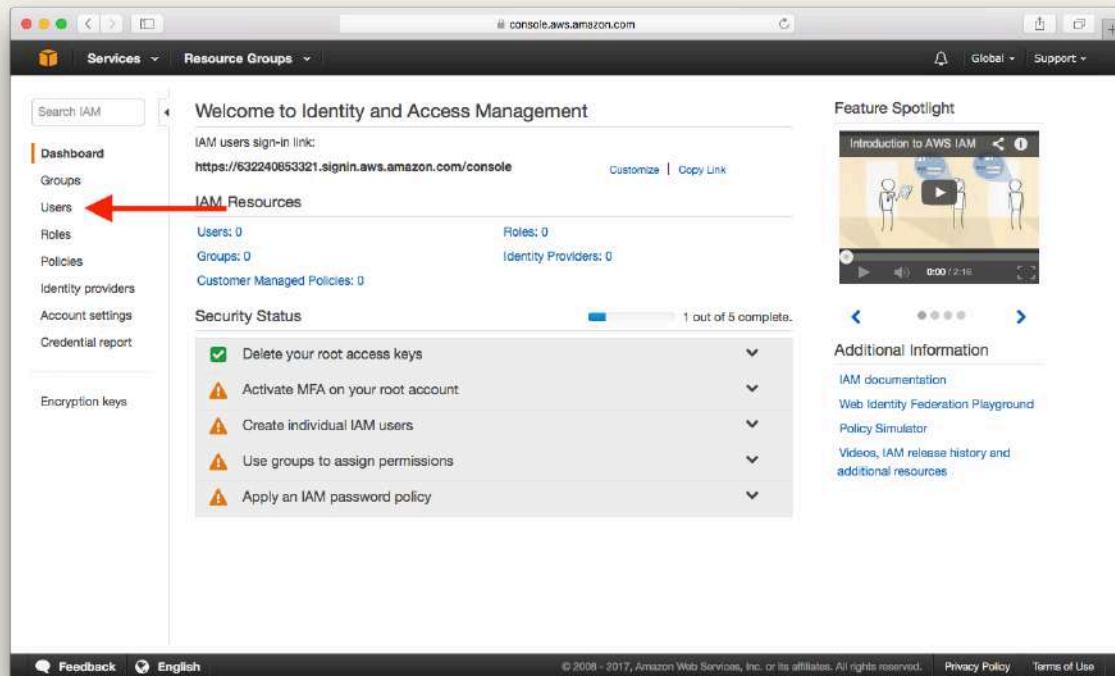
Create User

First, log in to your [AWS Console](#) and select IAM from the list of services.



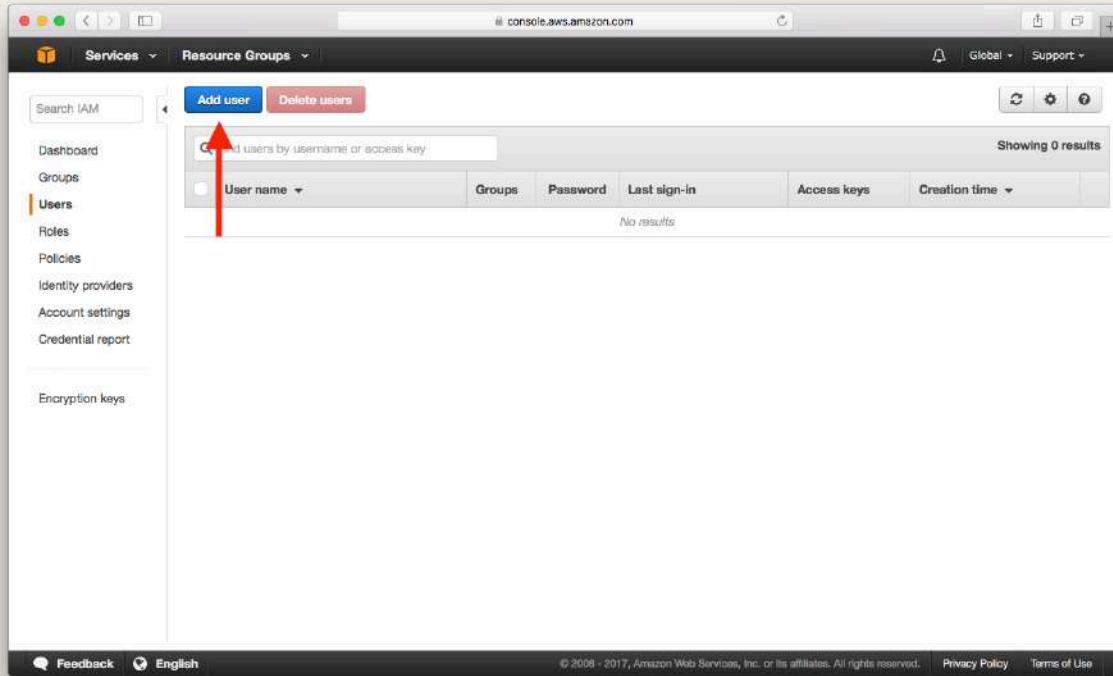
Select IAM Service Screenshot

Select **Users**.



Select IAM Users Screenshot

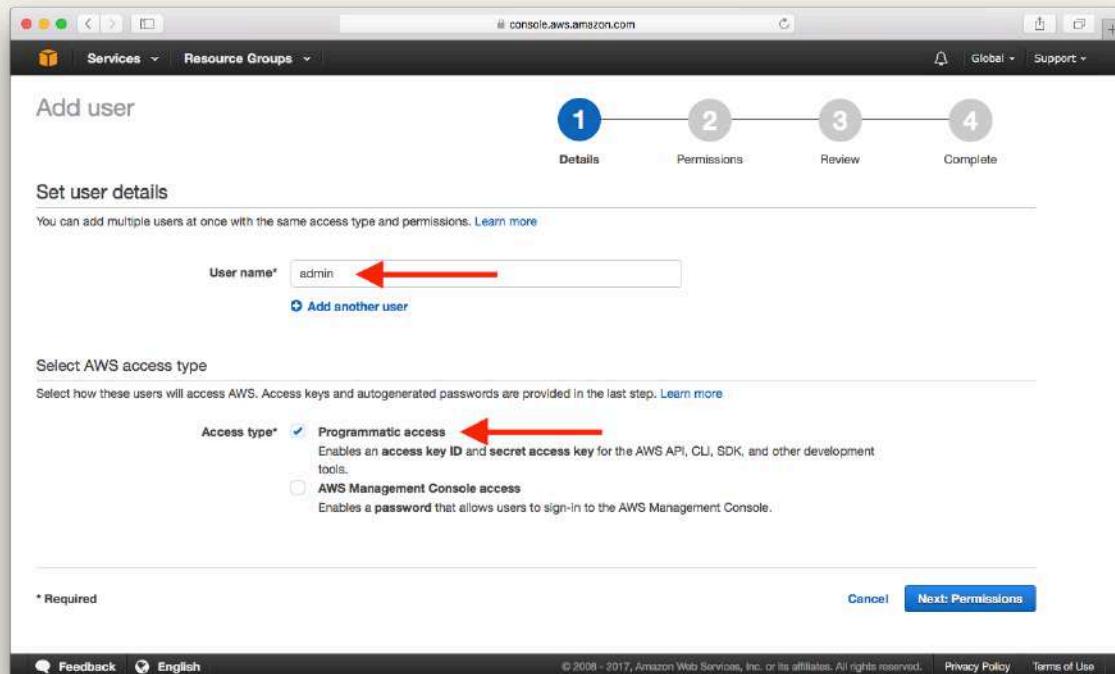
Select **Add User**.



Add IAM User Screenshot

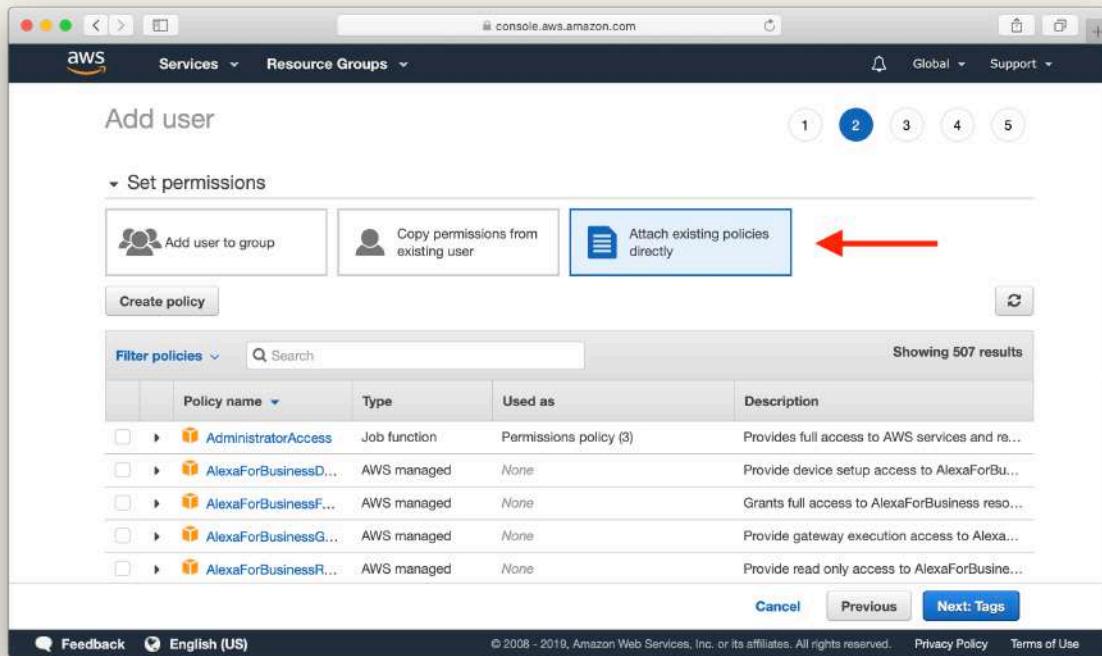
Enter a **User name** and check **Programmatic access**, then select **Next: Permissions**.

This account will be used by our [AWS CLI](#) and [Serverless Framework](#). They'll be connecting to the AWS API directly and will not be using the Management Console.



Fill in IAM User Info Screenshot

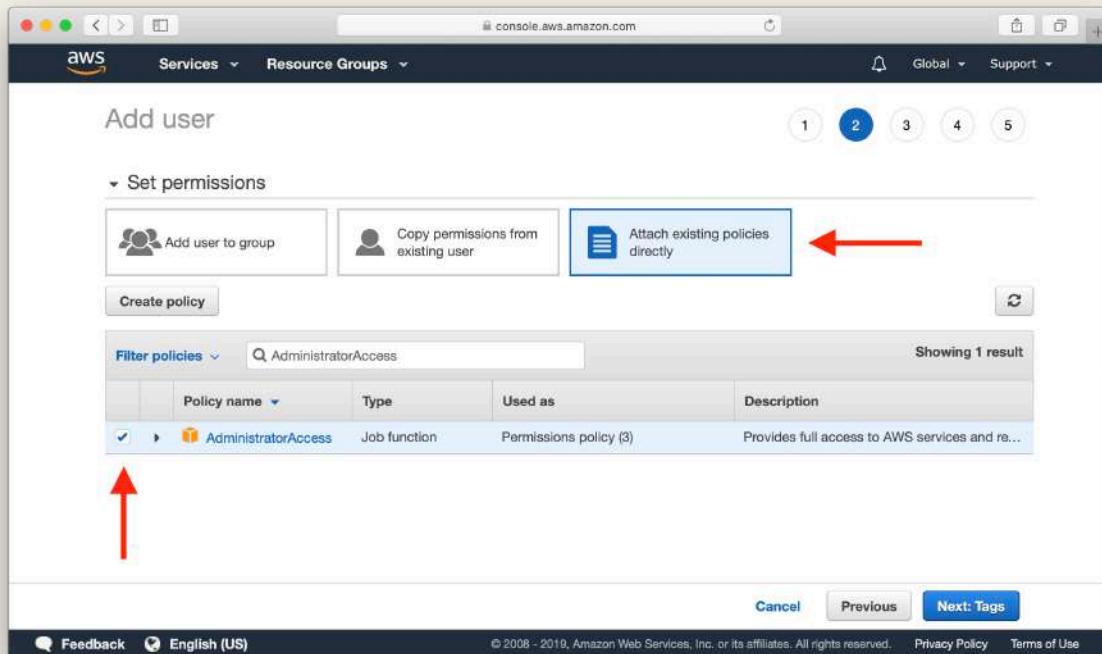
Select **Attach existing policies directly**.



Add IAM User Policy Screenshot

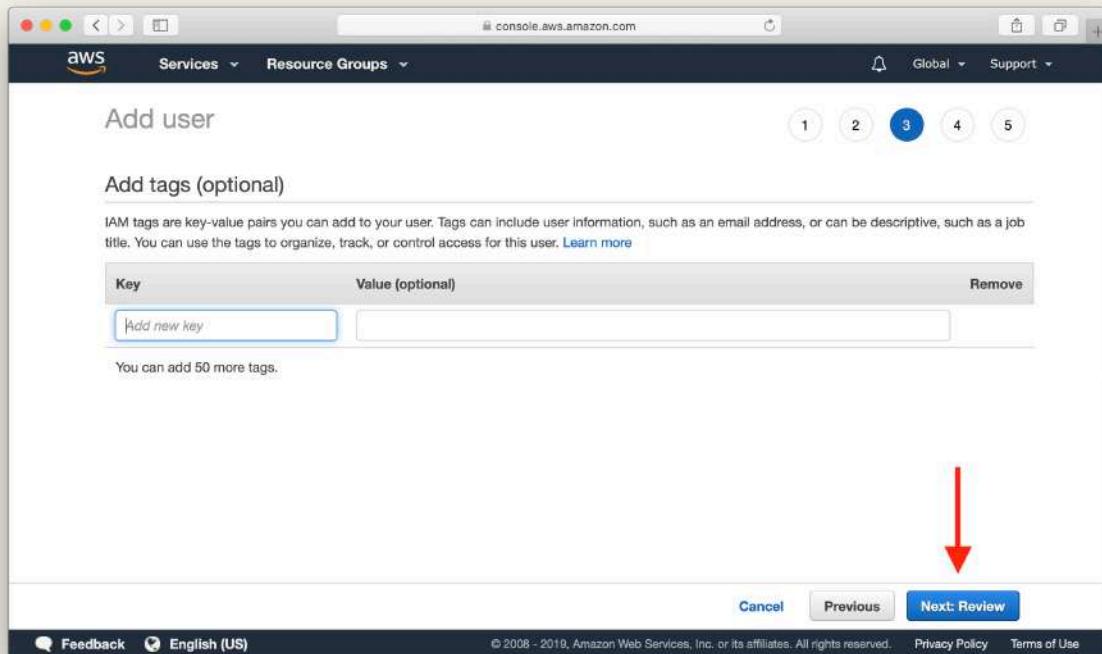
Search for **AdministratorAccess** and select the policy, then select **Next: Tags**.

We can provide a more fine-grained policy here and we cover this later in the [Customize the Serverless IAM Policy](#) chapter. But for now, let's continue with this.



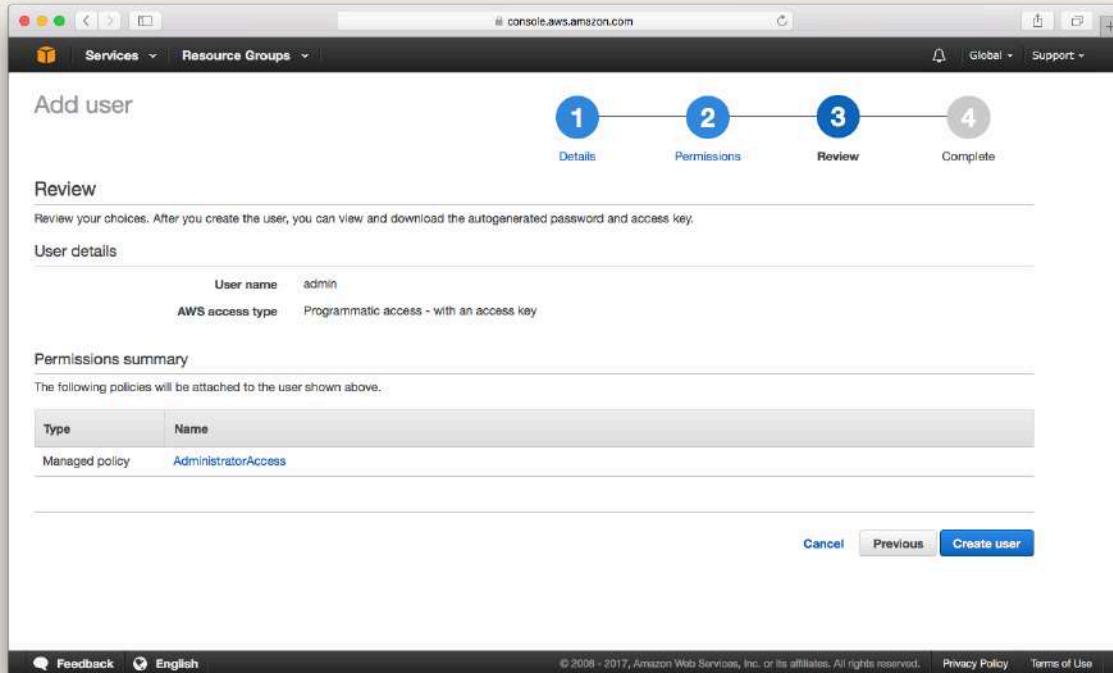
Added Admin Policy Screenshot

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



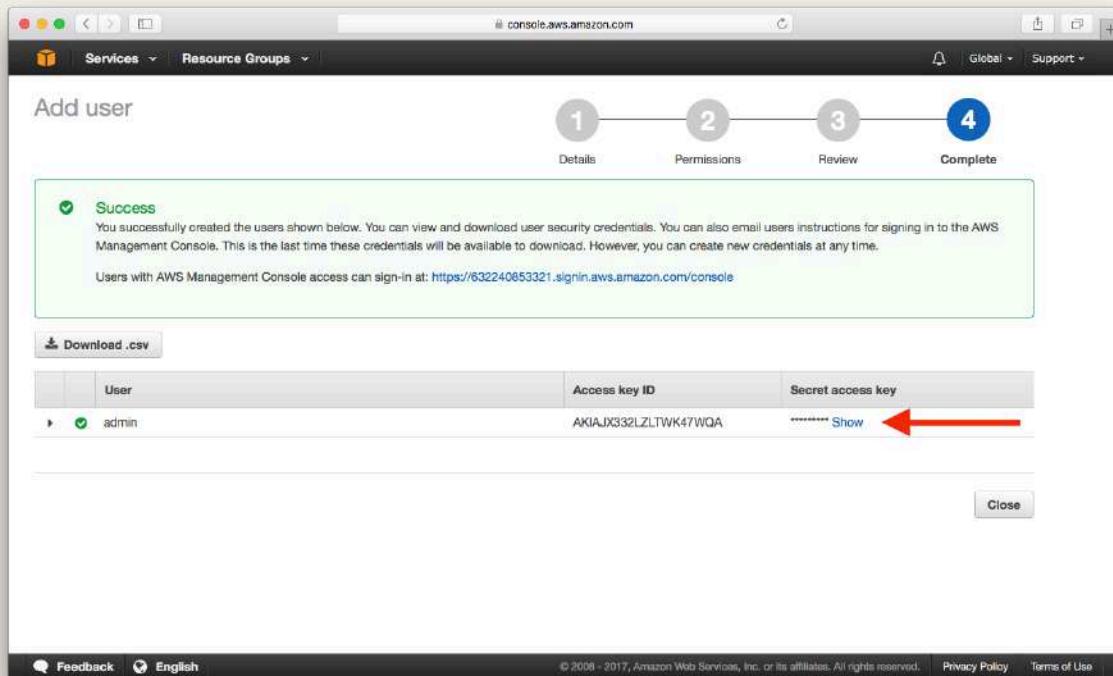
Skip IAM tags Screenshot

Select **Create user**.



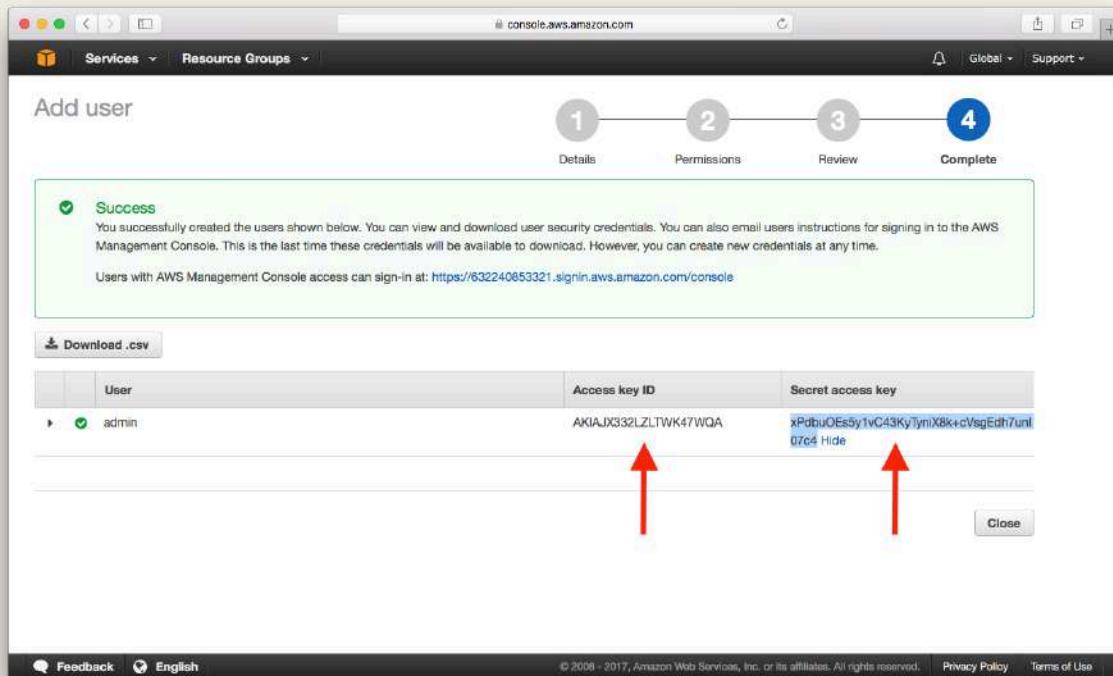
Review IAM User Screenshot

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



Added IAM User Screenshot

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



IAM User Credentials Screenshot

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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.

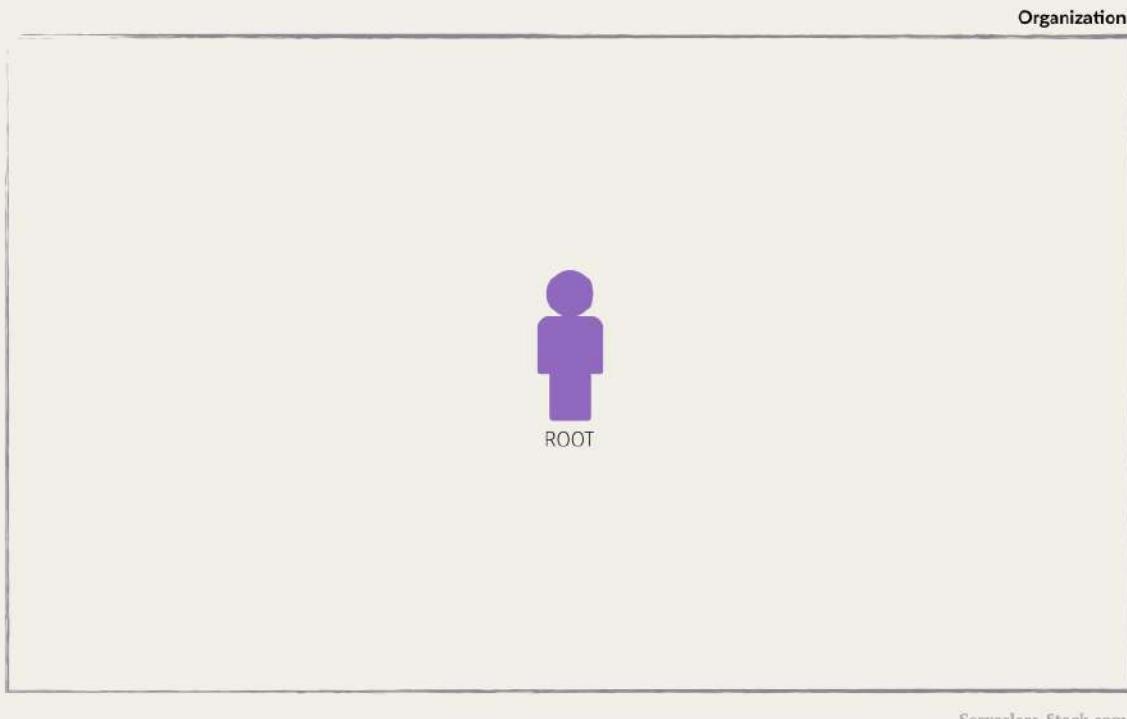
Let's start with the official definition of IAM.

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.



Serverless-Stack.com

IAM Root user diagram

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.



Serverless-Stack.com

IAM user diagram

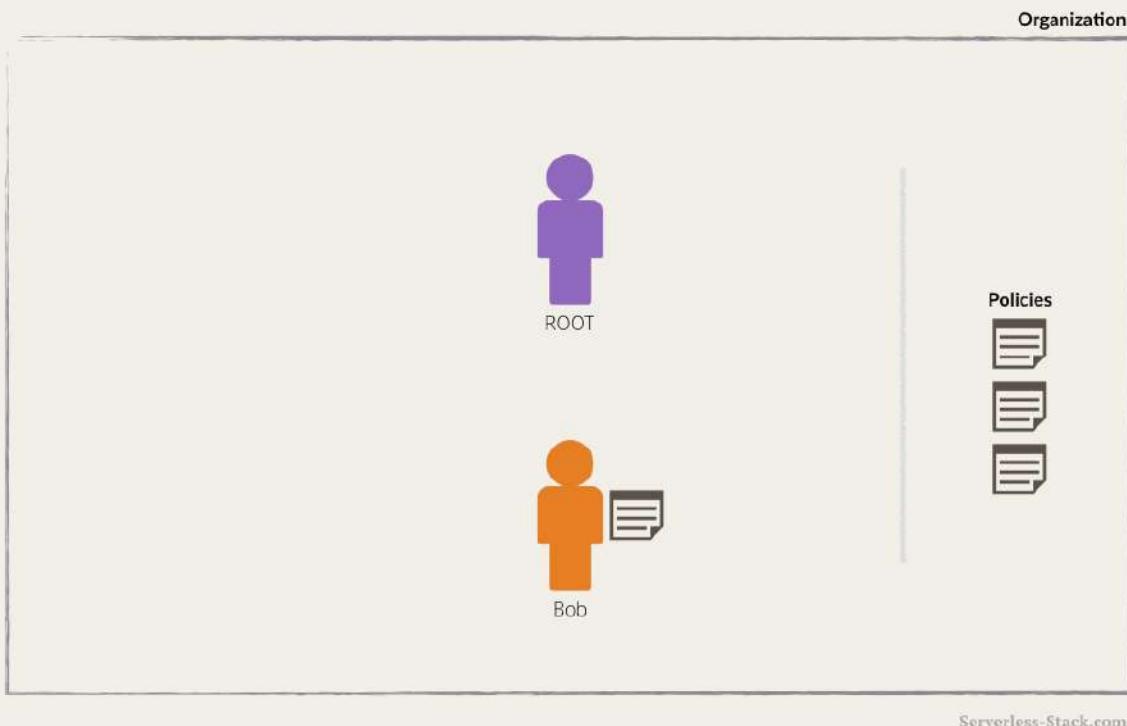
By default, users can't access anything in your account. You grant permissions to a user by creating a policy and attaching the policy to the user. You can grant one or more of these policies to restrict what the user can and cannot access.

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.



Serverless-Stack.com

IAM policy diagram

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/Bobs-*"  
  }  
}
```

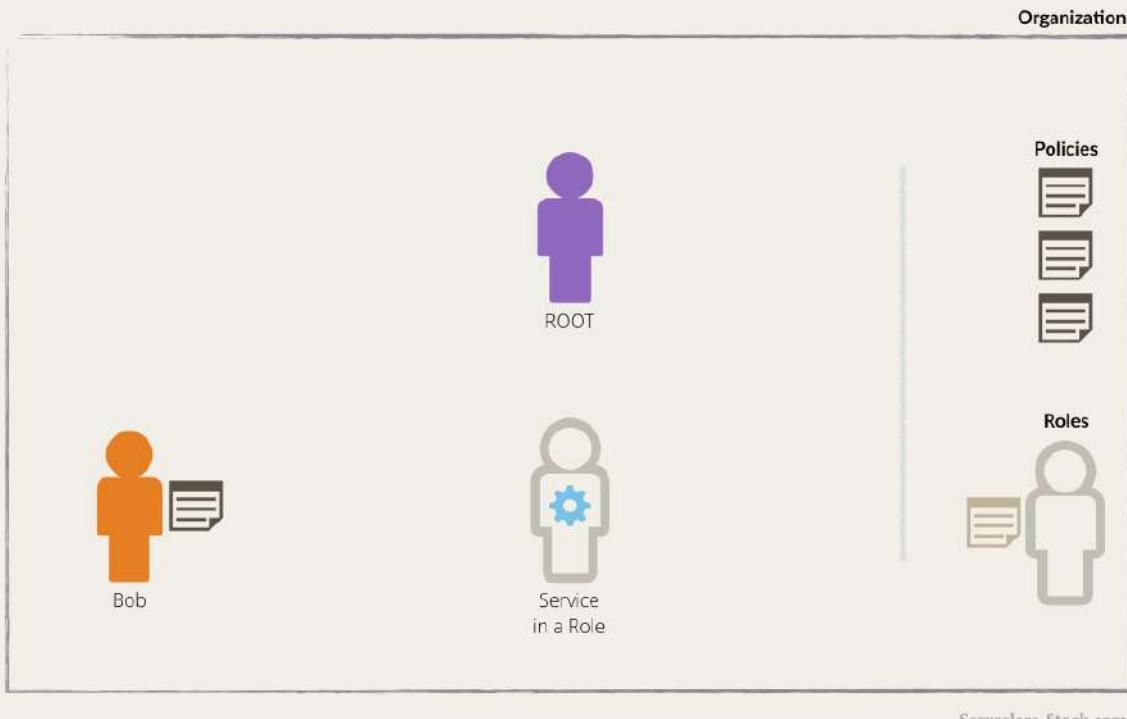
```
"Effect": "Allow",
"Action": ["s3:GetObject"],
"Resource": "arn:aws:s3:::Hello-bucket/*",
"Condition": {"StringEquals": {"s3:prefix": "Bobs-"}}
}
```

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](#). 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.



Serverless-Stack.com

AWS service with IAM Role diagram

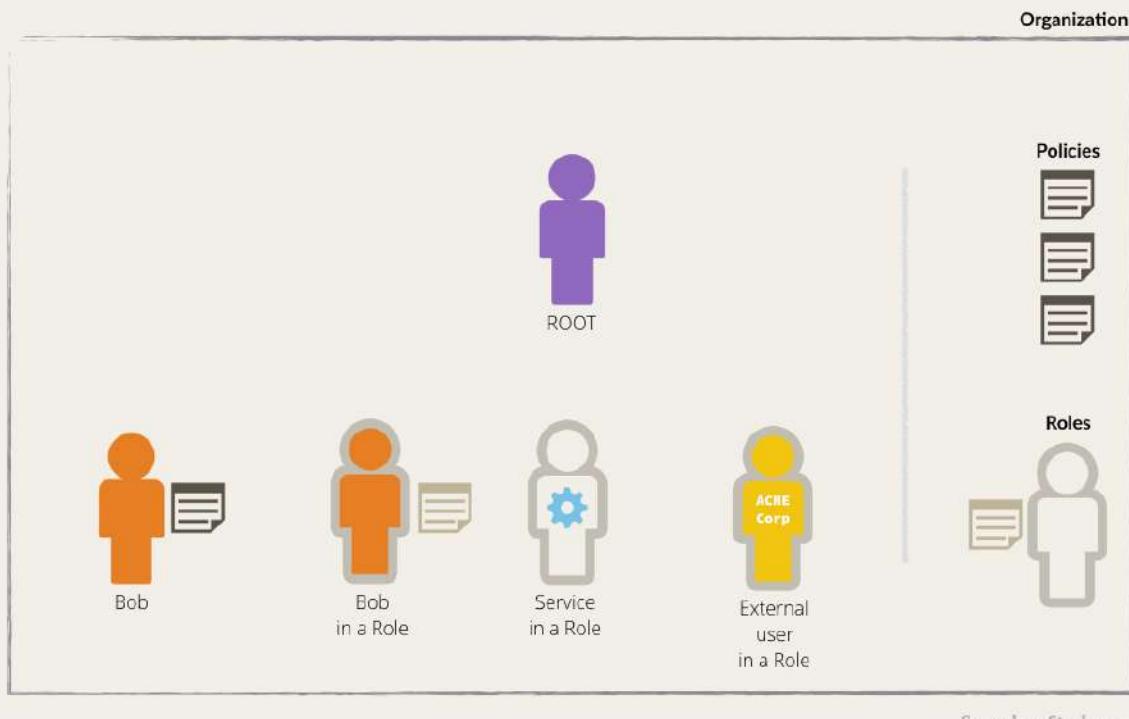
Roles can be applied to users as well. In this case, the user is taking on the policy set for the IAM role. This is useful for cases where a user is wearing multiple “hats” in the organization. Roles make this easy since you only need to create these roles once and they can be re-used for anybody else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



Serverless-Stack.com

IAM User with IAM Role diagram

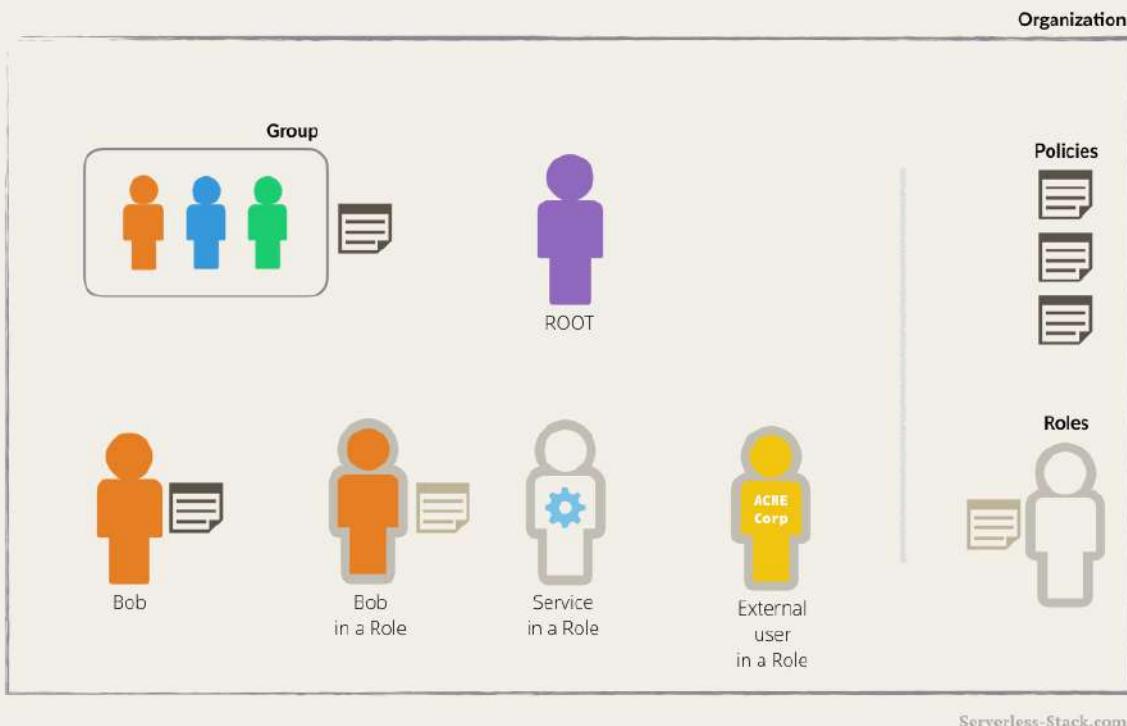
You can also have a role tied to the ARN of a user from a different organization. This allows the external user to assume that role as a part of your organization. This is typically used when you have a third party service that is acting on your AWS Organization. You'll be asked to create a **Cross-Account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



External IAM User with IAM Role diagram

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.



Serverless-Stack.com

Complete IAM Group, IAM Role, IAM User, and IAM Policy diagram

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.



Help and discussion

View the [comments](#) for this chapter on our forums

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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](#).

Install the AWS CLI

AWS CLI needs Python 2 version 2.6.5+ or Python 3 version 3.3+ and [Pip](#). Use the following if you need help installing Python or Pip.

- [Installing Python](#)
- [Installing Pip](#)

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

```
$ sudo pip install awscli
```

Or using [Homebrew](#) 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](#).

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**



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.



Help and discussion

View the [comments](#) for this chapter on our forums

Setting up the Serverless backend

Create a DynamoDB Table

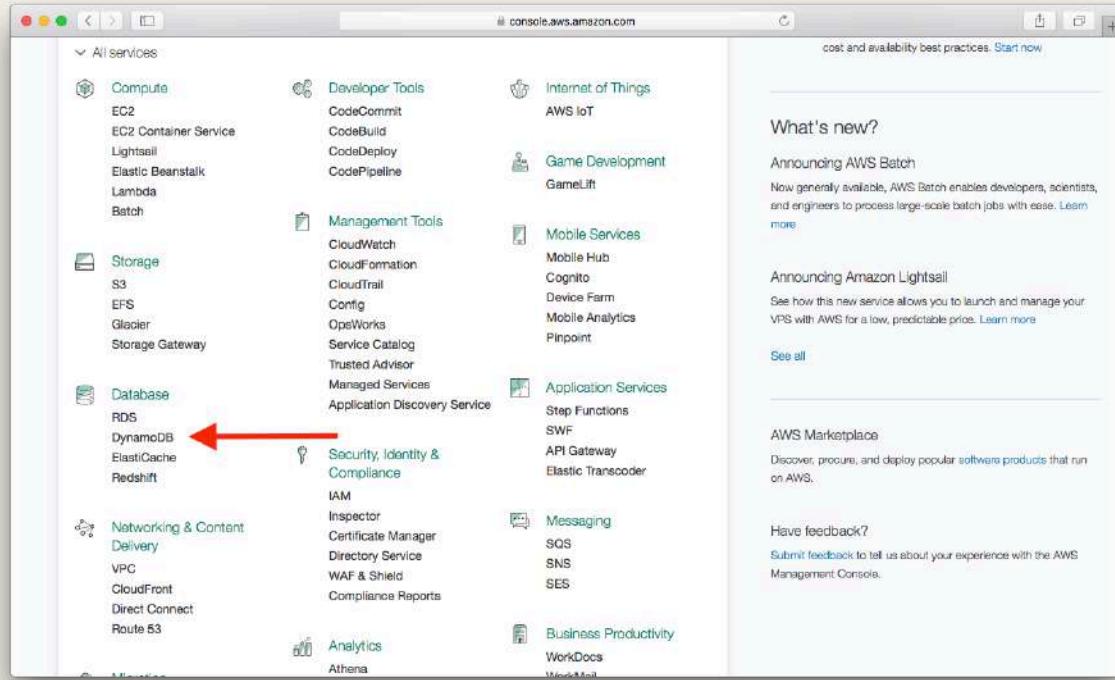
To build the backend for our notes app, it makes sense that we first start by thinking about how the data is going to be stored. We are going to use [DynamoDB](#) to do this.

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](#).

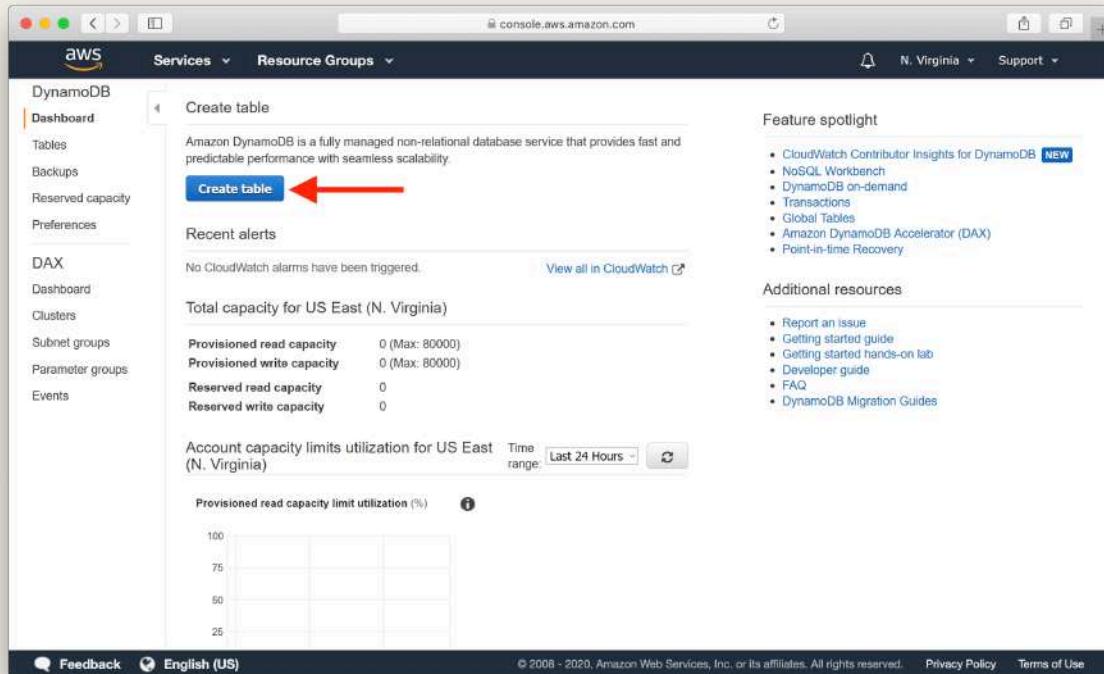
Create Table

First, log in to your [AWS Console](#) and select **DynamoDB** from the list of services.



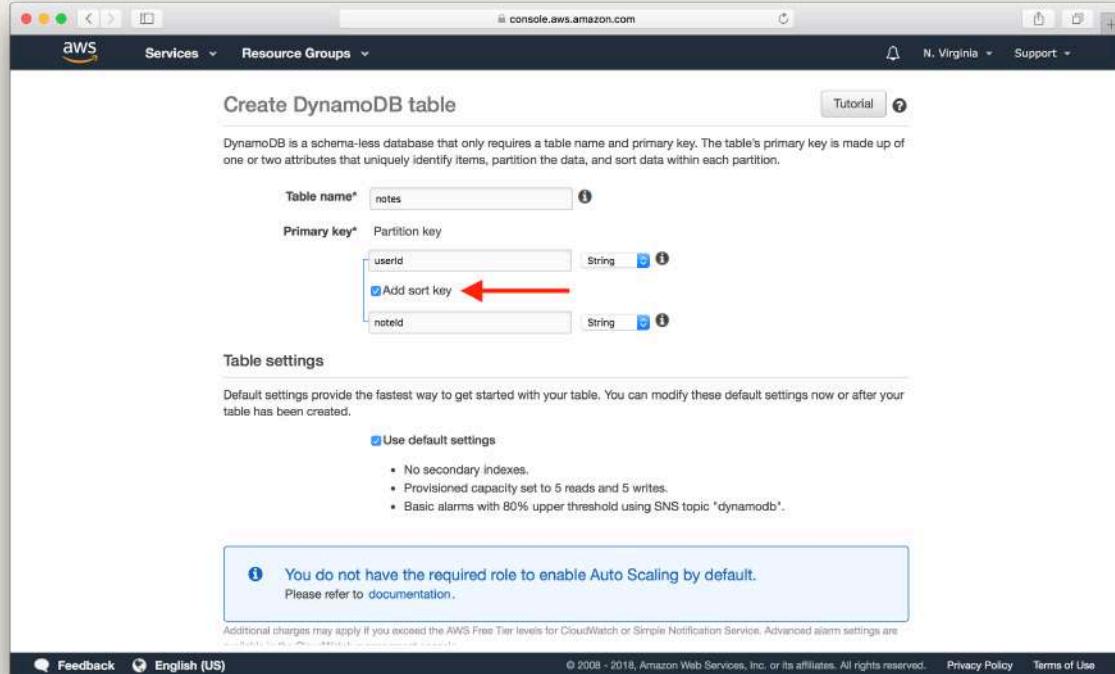
Select DynamoDB Service screenshot

Select **Create table**.



Create DynamoDB Table screenshot

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



Set Table Primary Key screenshot

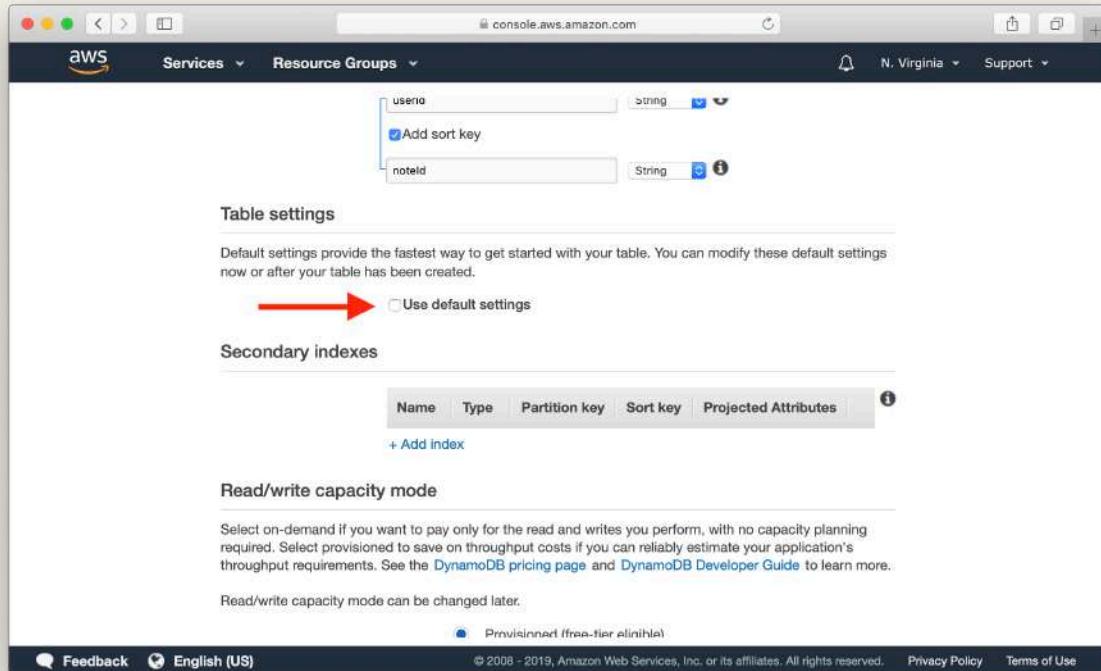
Each DynamoDB table has a primary key, which cannot be changed once set. The primary key uniquely identifies each item in the table, so that no two items can have the same key. DynamoDB supports two different kinds of primary keys:

- 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.

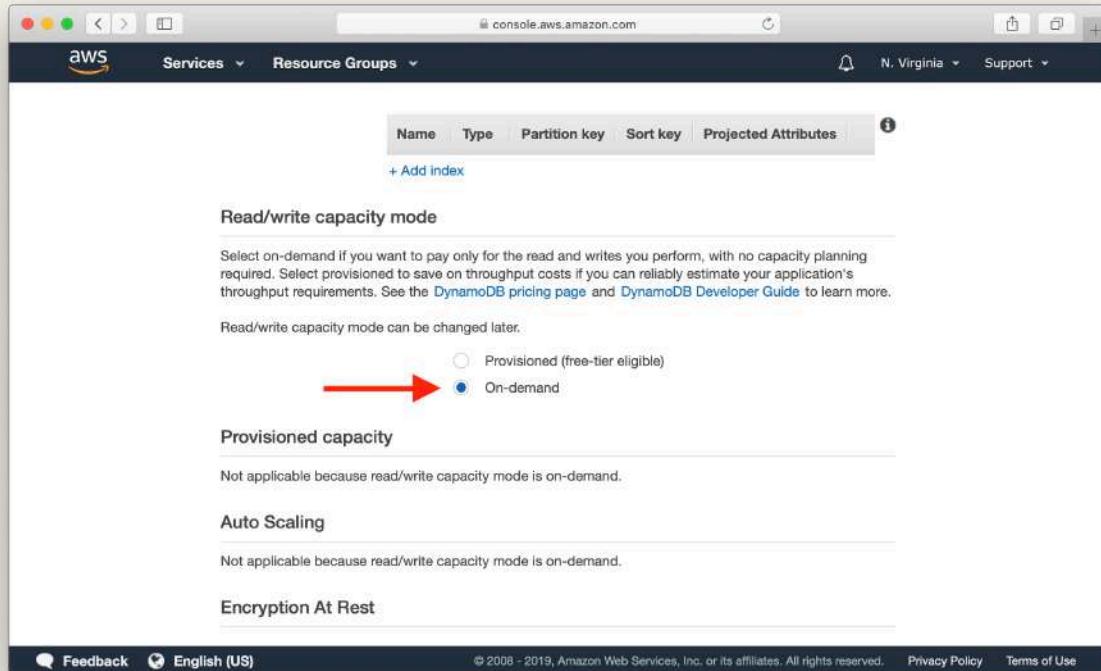
To further your understanding of how indexes work in DynamoDB, you can read more here: [DynamoDB Core Components](#)

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



Deselect Use default settings screenshot

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

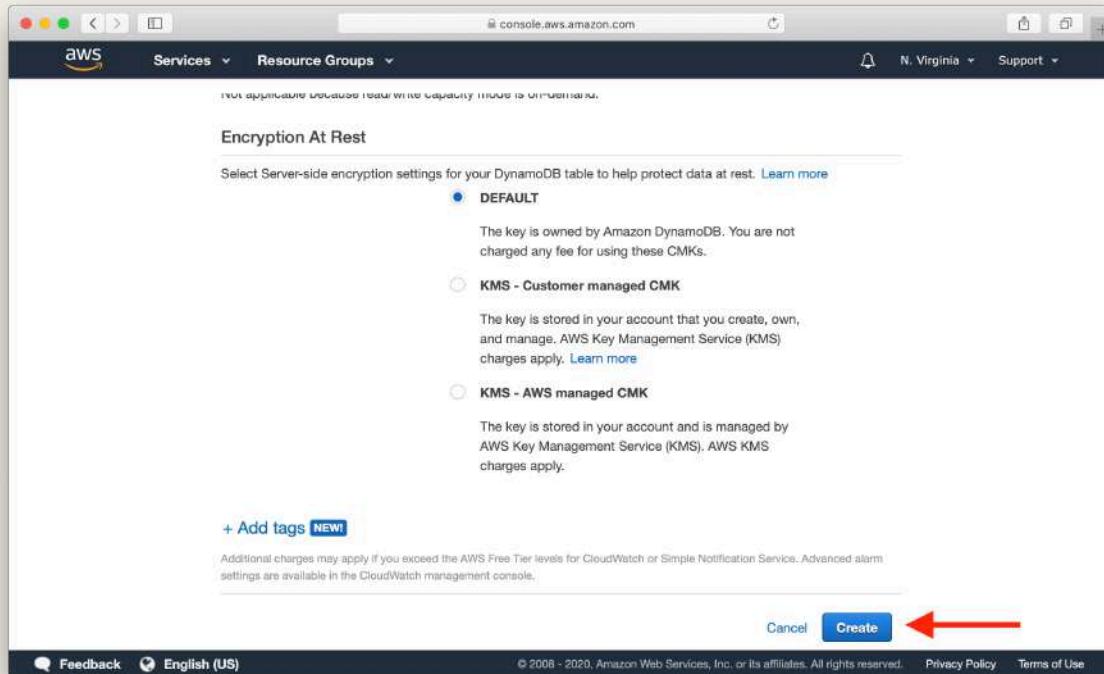


Select On-Demand Capacity screenshot

On-Demand Capacity 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** mode.

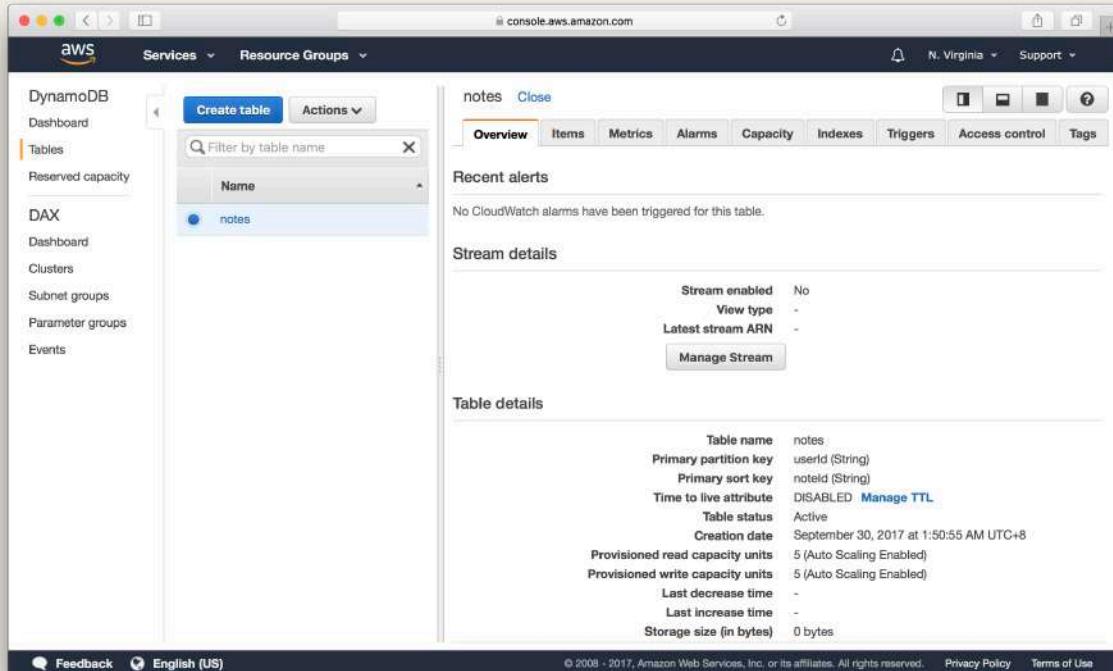
If the On-Demand Capacity option is missing and there is an info box containing the message "You do not have the required role to enable Auto Scaling by default", you can create the table and afterwards modify the setting from the "Capacity" tab of the table settings page. The role mentioned by the info box is automatically created by the table creation process.

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



Create DynamoDB table screenshot

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



Select DynamoDB Service screenshot

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](#).

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



Help and discussion

View the [comments for this chapter on our forums](#)

Create an S3 Bucket for File Uploads

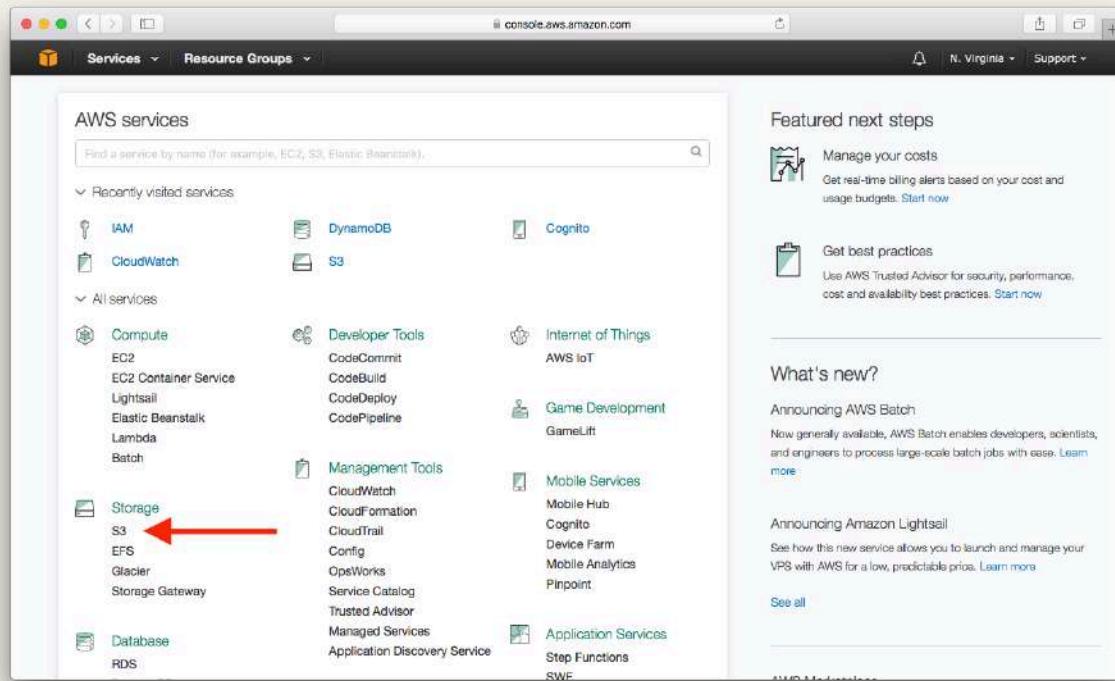
Now that we have our database table ready; let's get things set up for handling file uploads. We need to handle file uploads because each note can have an uploaded file as an attachment.

[Amazon S3](#) (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.

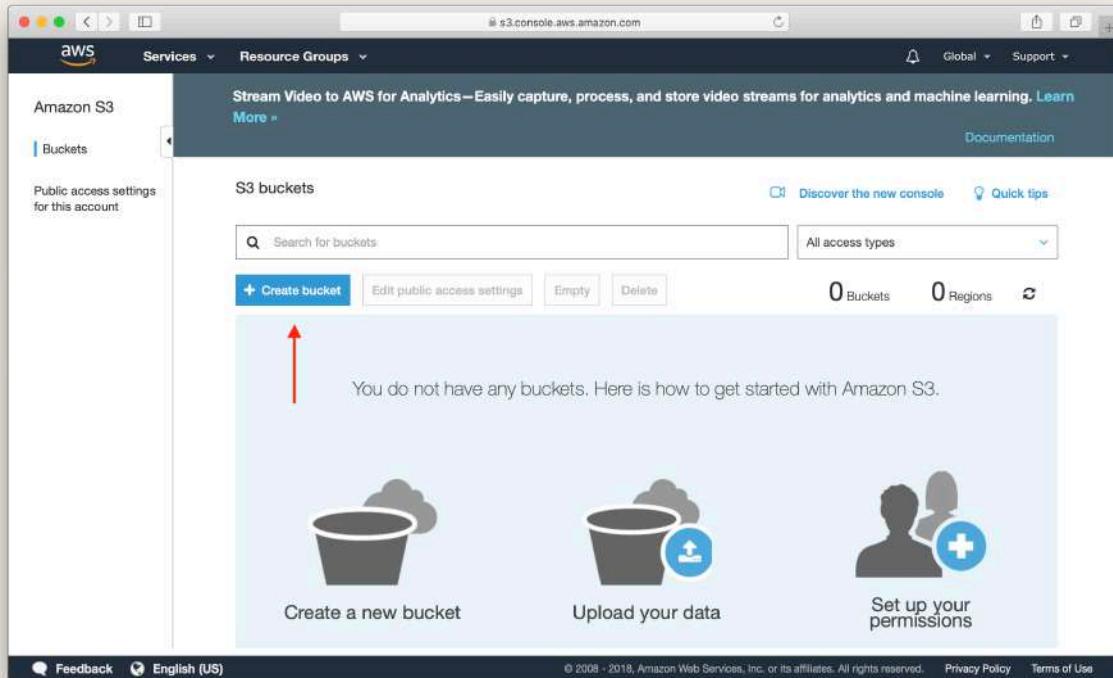
Create Bucket

First, log in to your [AWS Console](#) and select **S3** from the list of services.



Select S3 Service screenshot

Select **Create bucket**.

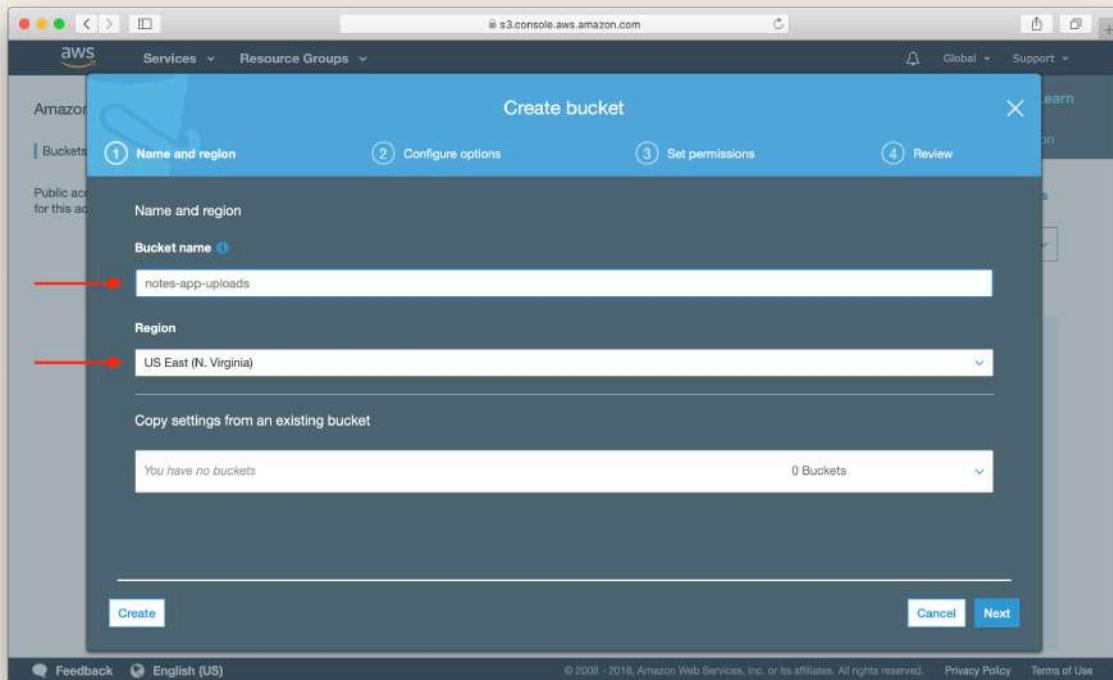


Select Create Bucket screenshot

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

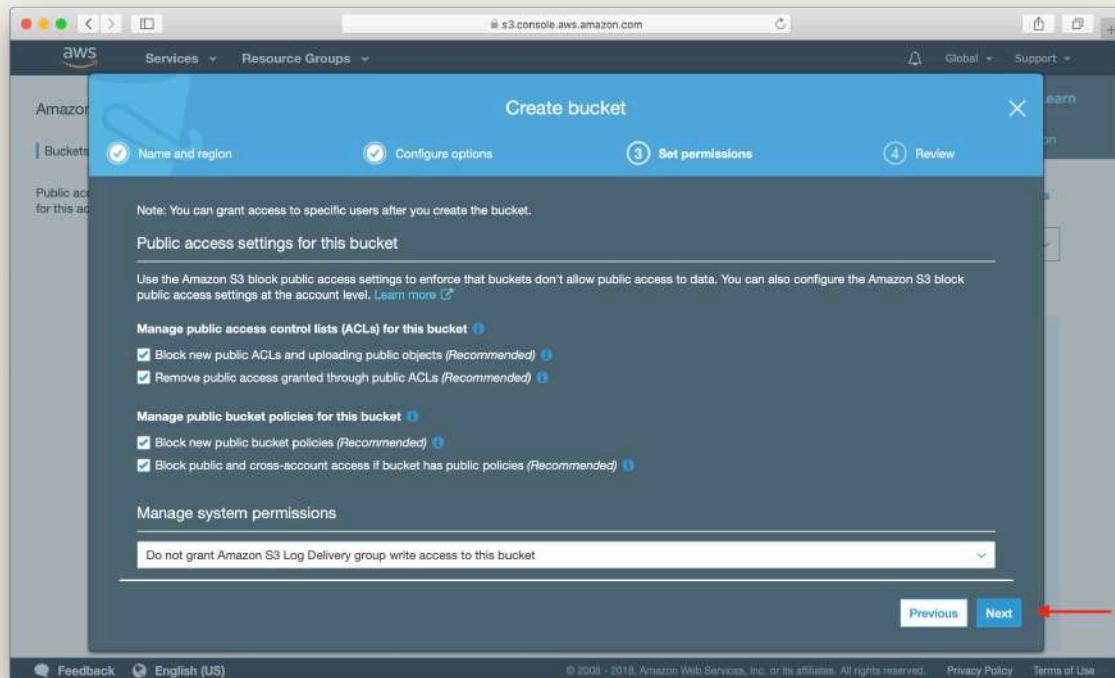
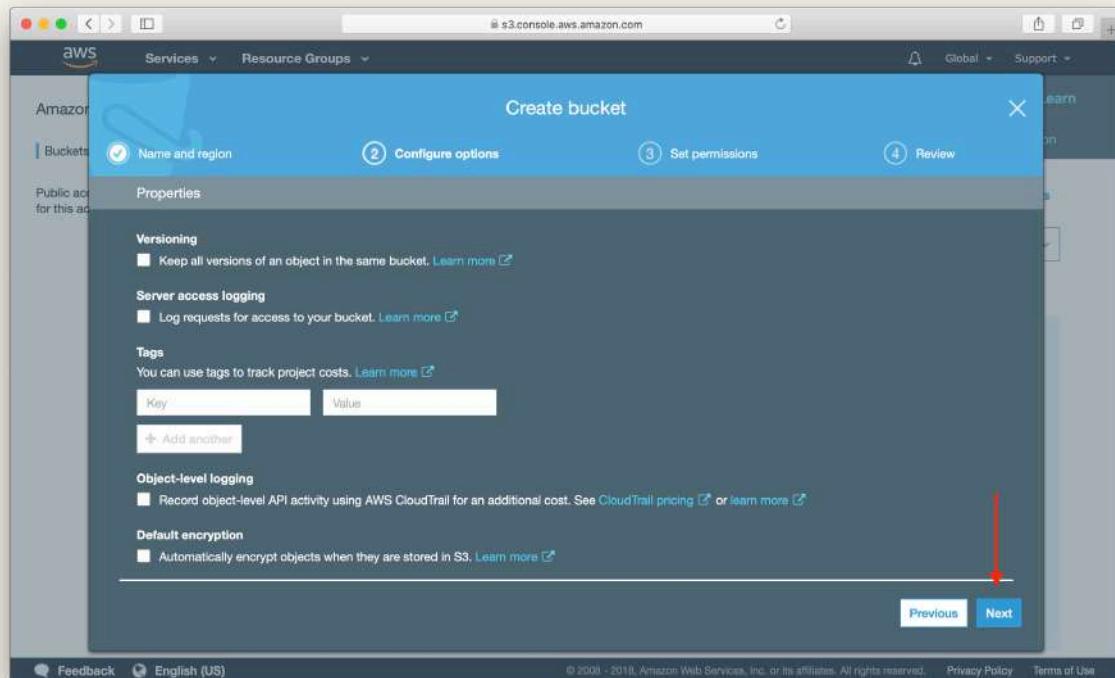
- **Bucket names** are globally unique, which means you cannot pick the same name as this tutorial.
- **Region** is the physical geographical region where the files are stored. We will use **US East (N. Virginia)** for this guide.

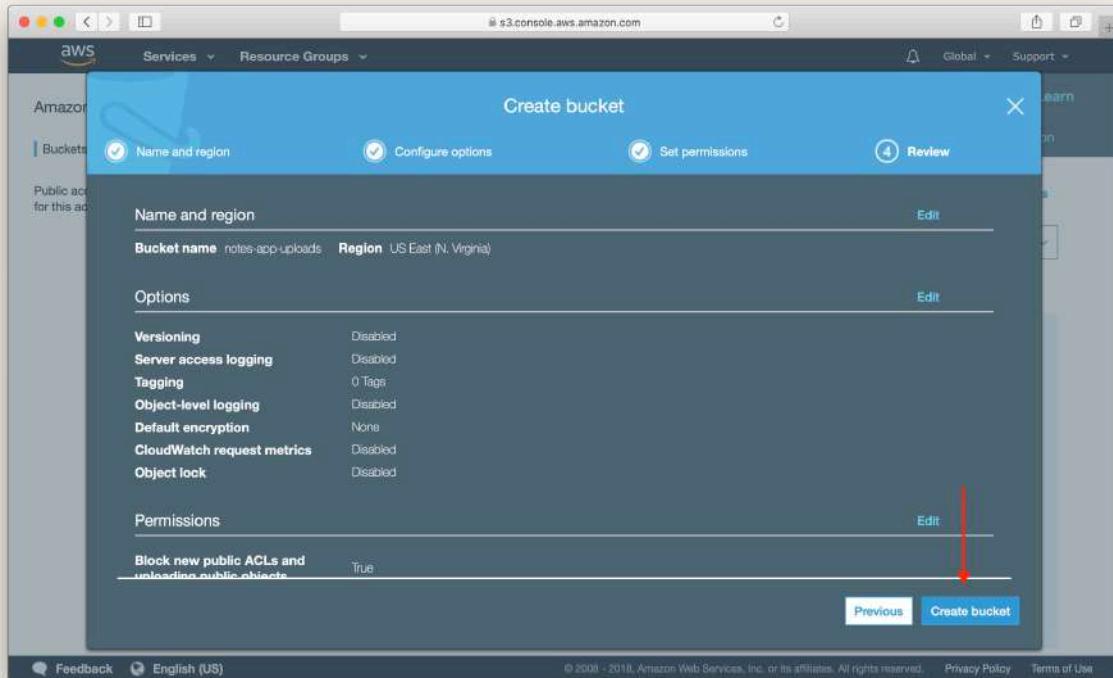
Make a note of the name and region as we'll be using it later in the guide.



Enter S3 Bucket Info screenshot

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

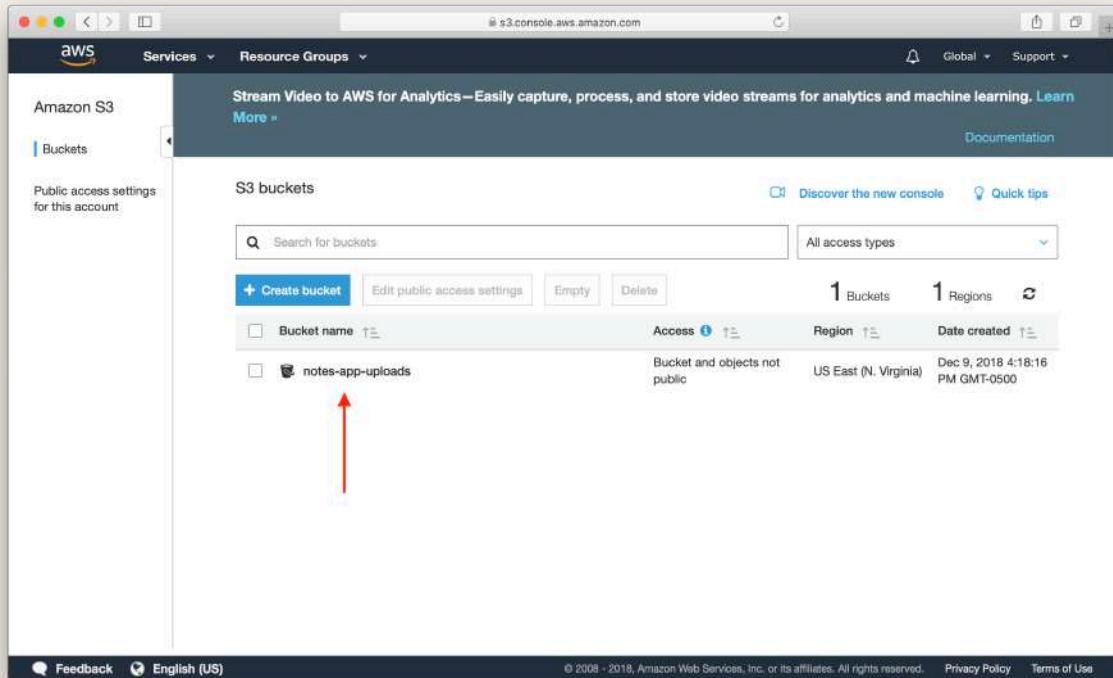




Enable CORS

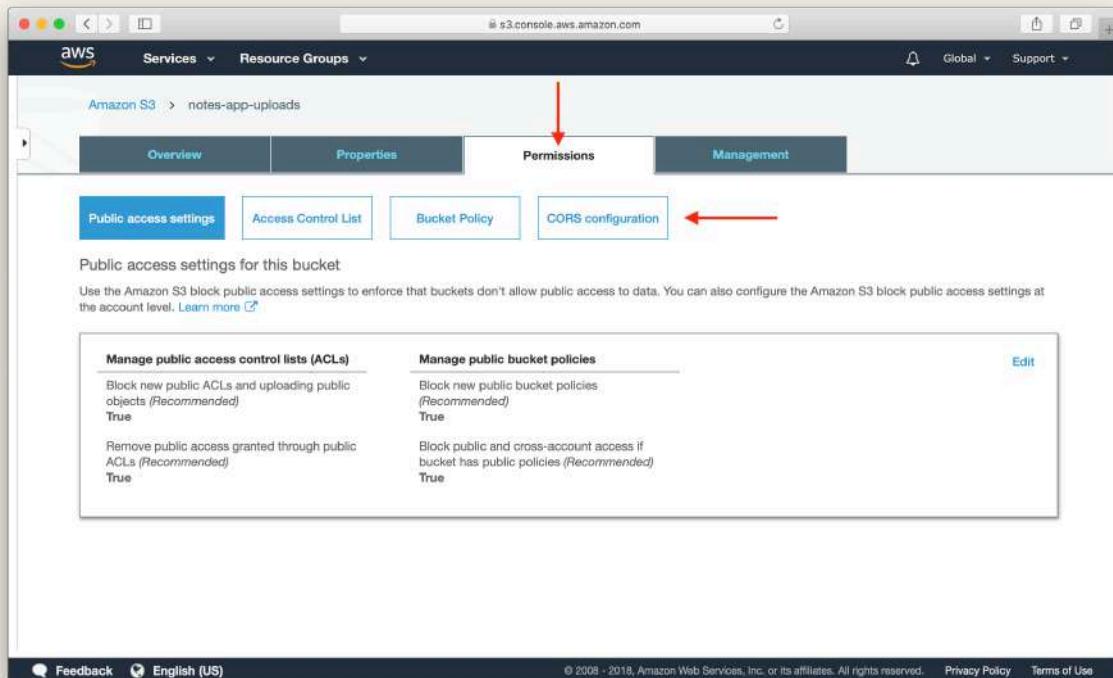
In the notes app we'll be building, users will be uploading files to the bucket we just created. And since our app will be served through our custom domain, it'll be communicating across domains while it does the uploads. By default, S3 does not allow its resources to be accessed from a different domain. 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 Created S3 Bucket screenshot

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

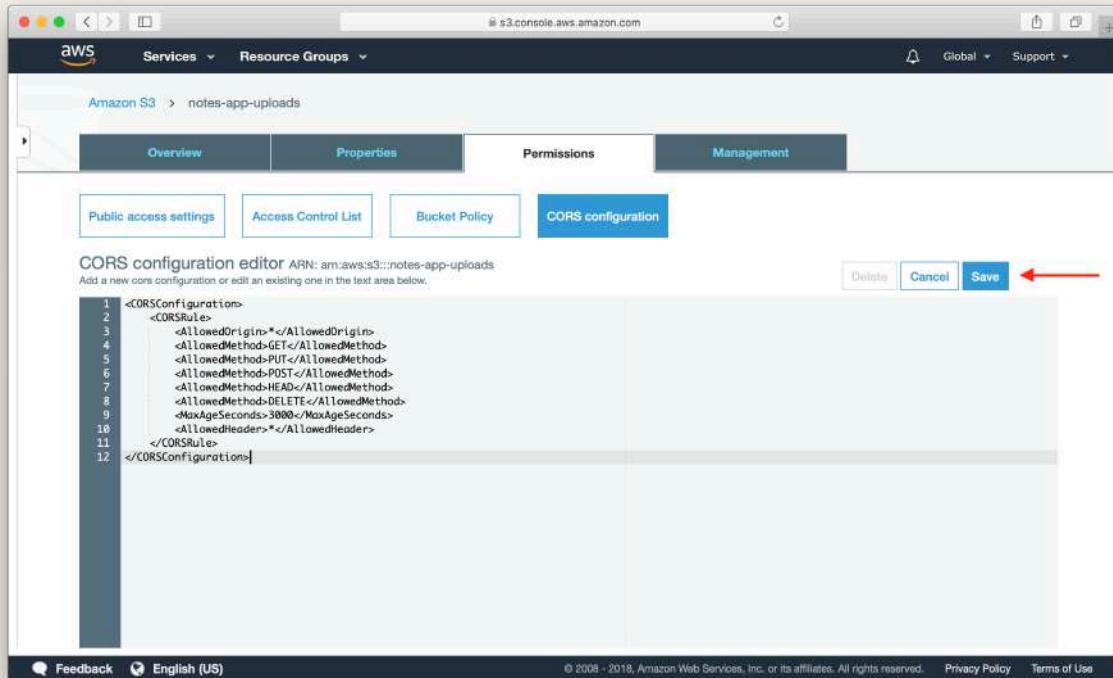


Select S3 Bucket CORS Configuration screenshot

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.



Save S3 Bucket CORS Configuration screenshot

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



Help and discussion

View the [comments for this chapter on our forums](#)

Create a Cognito User Pool

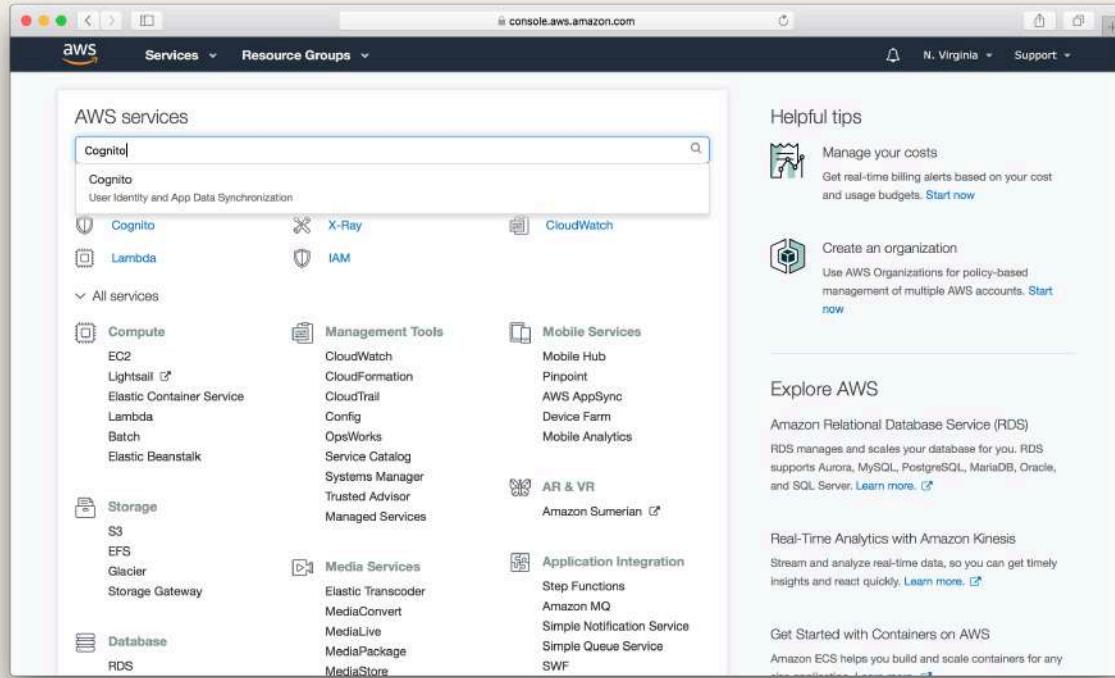
Our notes app needs to handle user accounts and authentication in a secure and reliable way. To do this we are going to use [Amazon Cognito](#).

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.

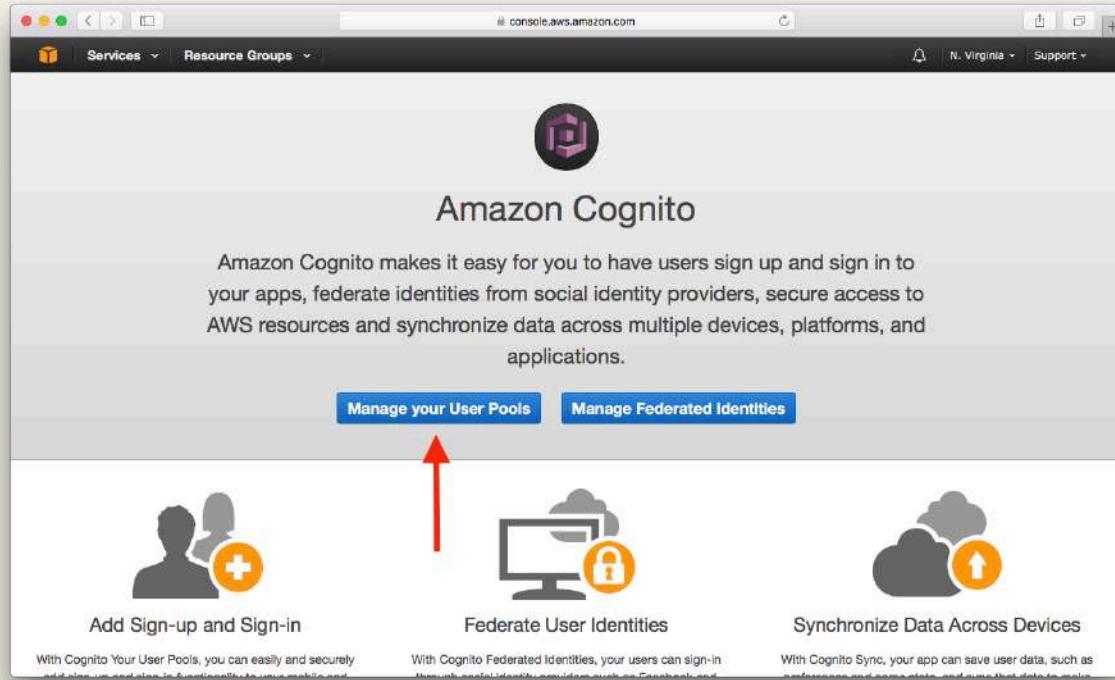
Create User Pool

From your [AWS Console](#), select **Cognito** from the list of services.



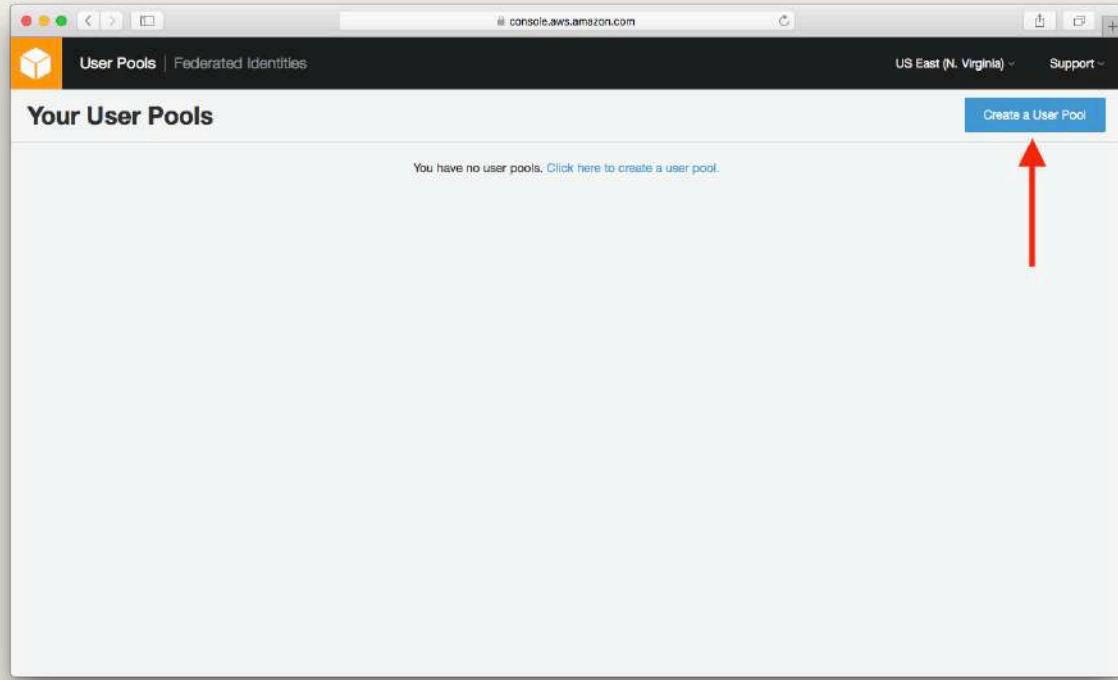
Select Amazon Cognito Service screenshot

Select **Manage your User Pools**.



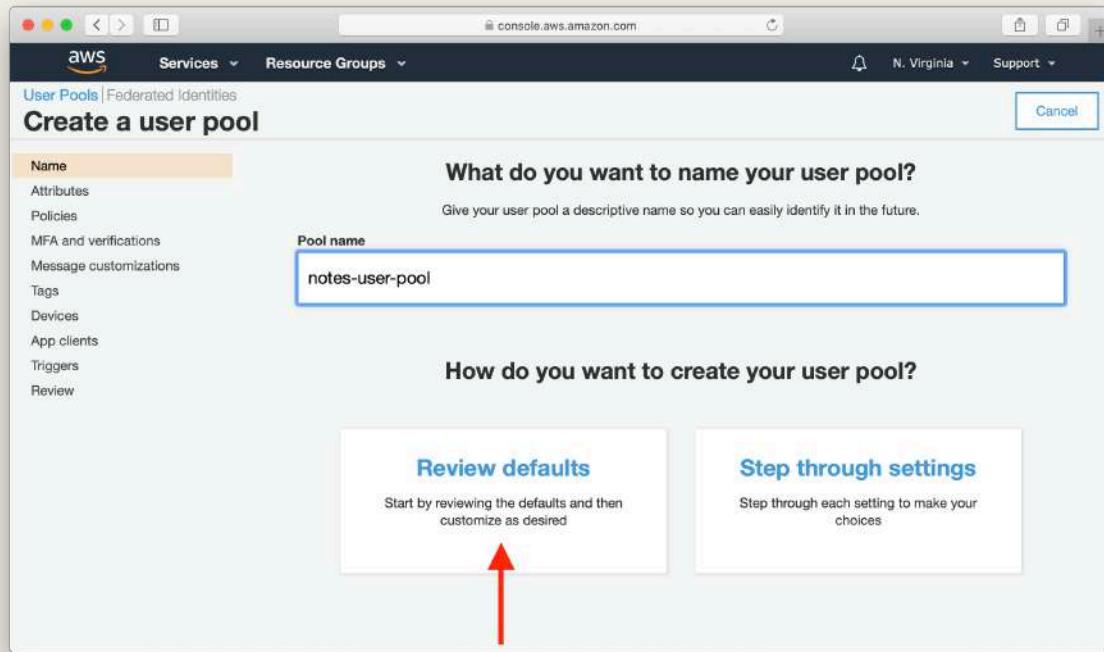
Select Manage Your Cognito User Pools screenshot

Select **Create a User Pool**.



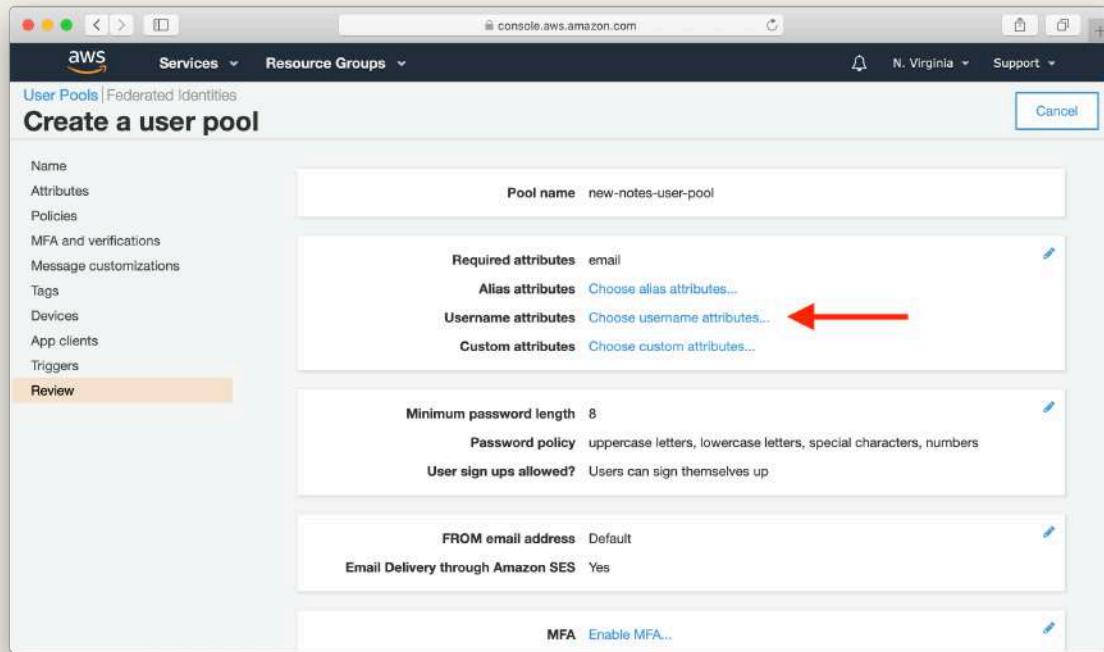
Select Create a Cognito User Pool screenshot

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



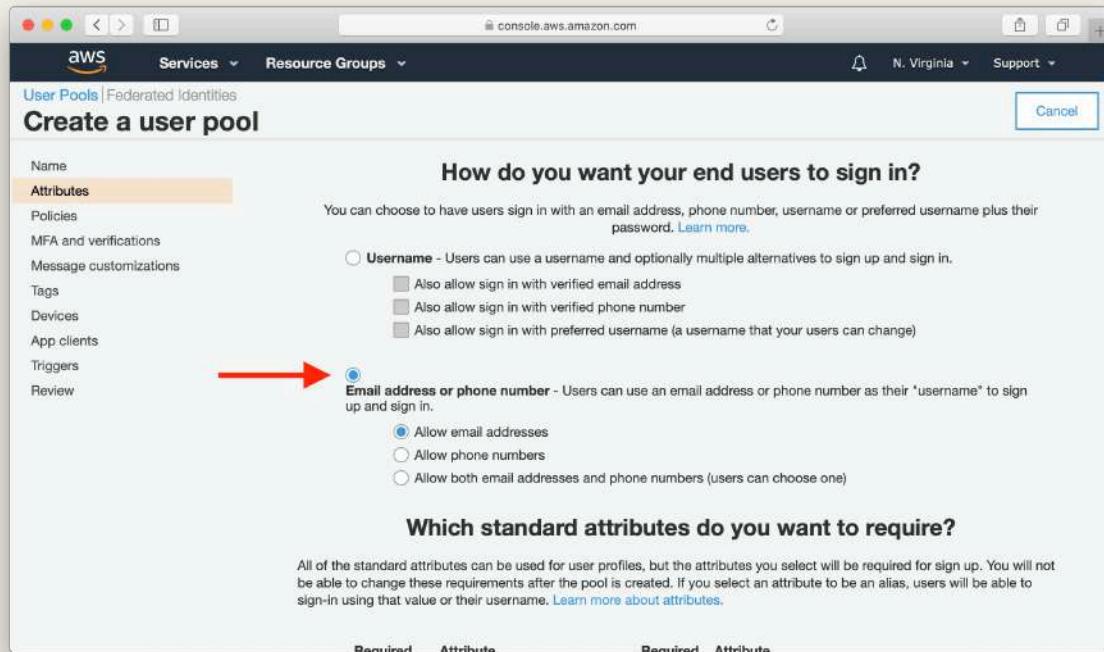
Fill in Cognito User Pool info screenshot

Select **Choose username attributes....**



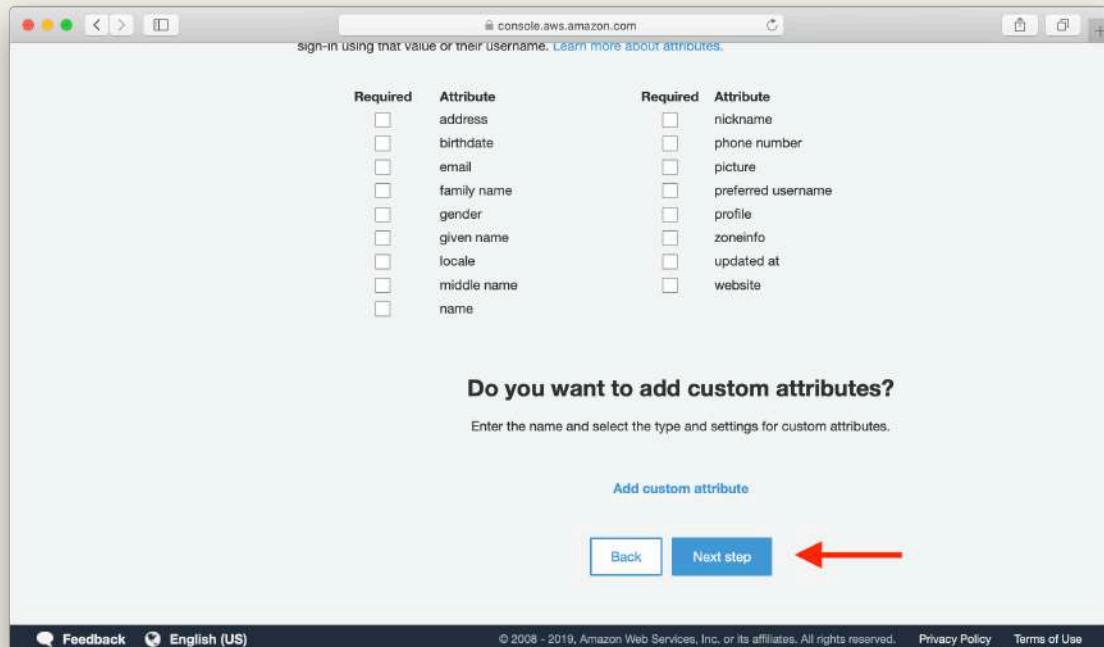
Choose username attribute screenshot

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.



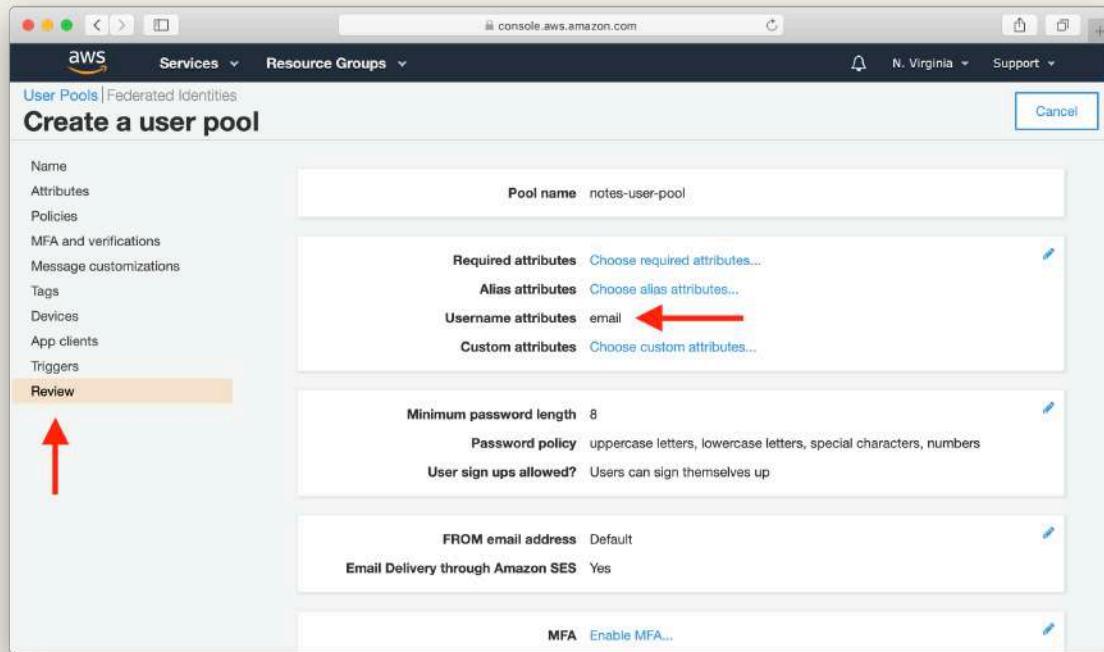
Select email address as username screenshot

Scroll down and select **Next step**.



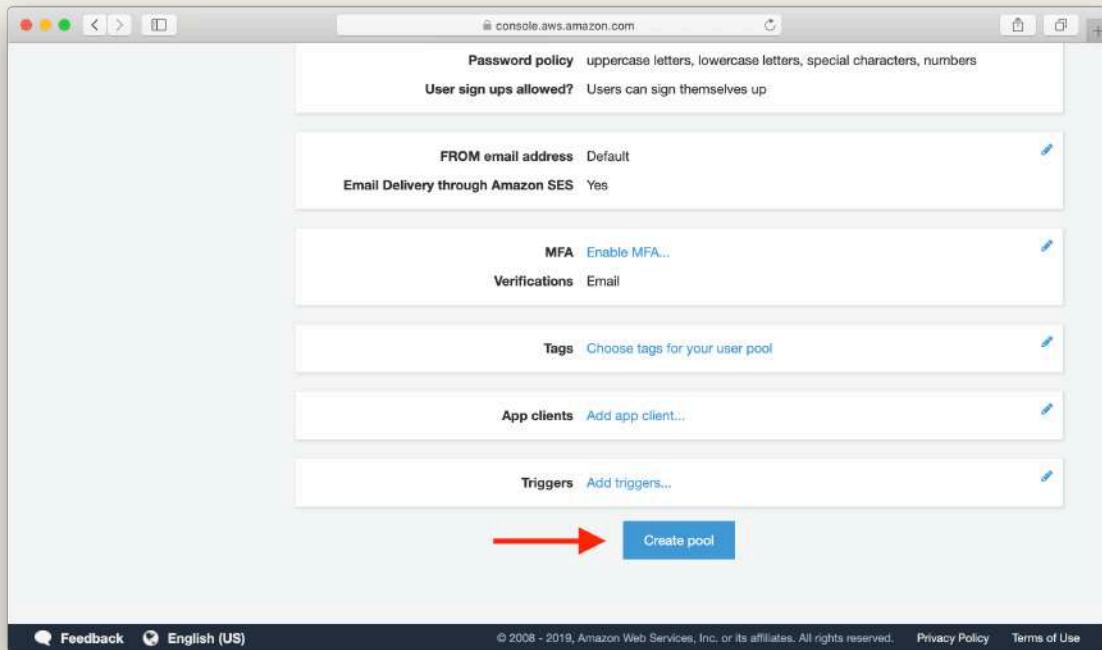
Select attributes next step screenshot

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



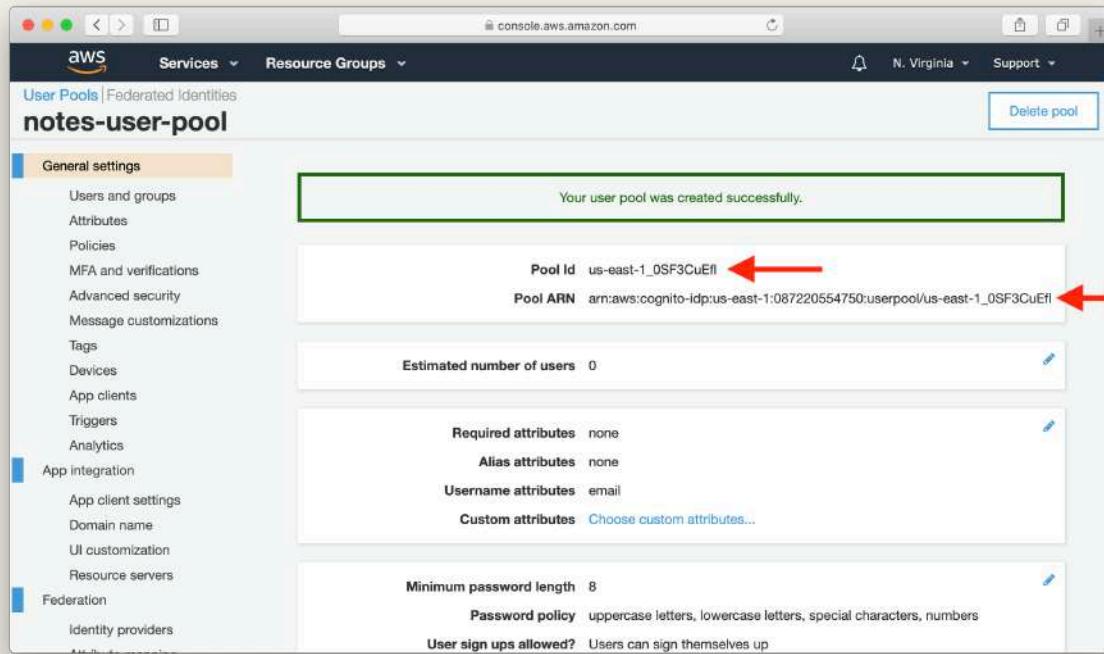
Review User Pool settings screenshot

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



Select Create pool screenshot

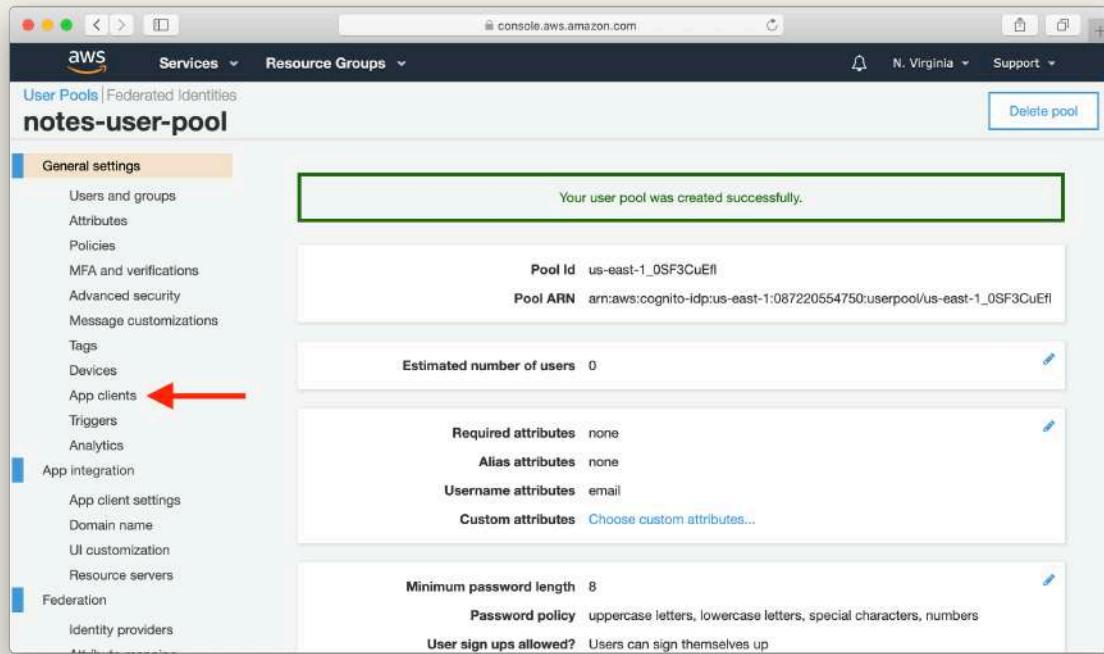
Your User Pool has been created. Take a note of the **Pool Id** and **Pool ARN** which will be required later. Also, note the region that your User Pool is created in – in our case it's us-east-1.



Cognito User Pool Created Screenshot

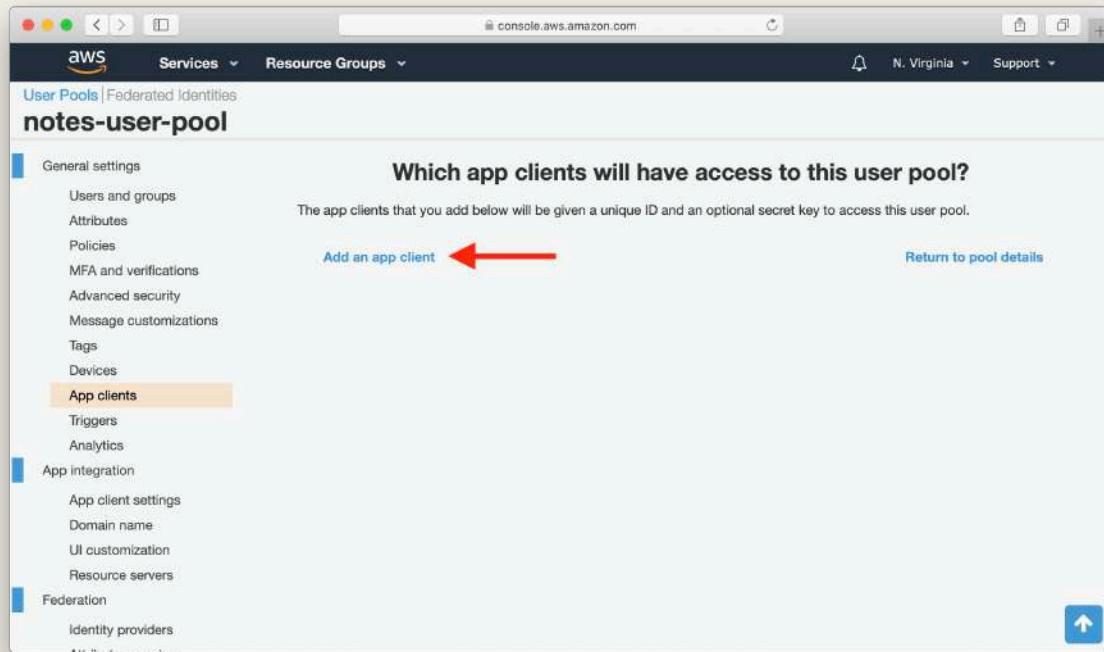
Create App Client

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



Select Congito User Pool Apps Screenshot

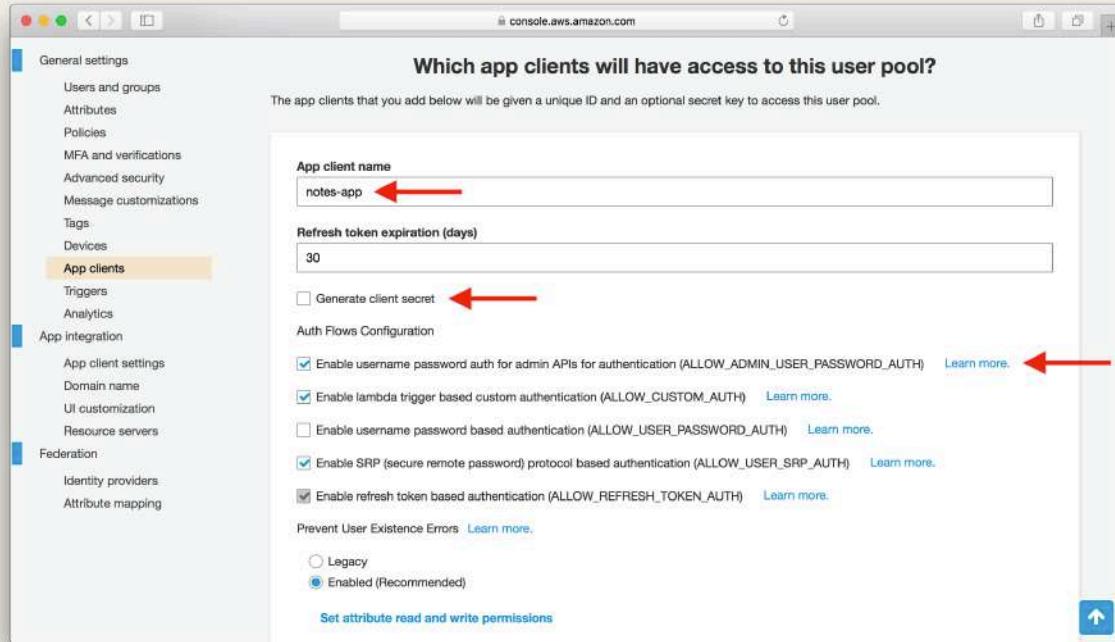
Select **Add an app client**.



Select Add An App Screenshot

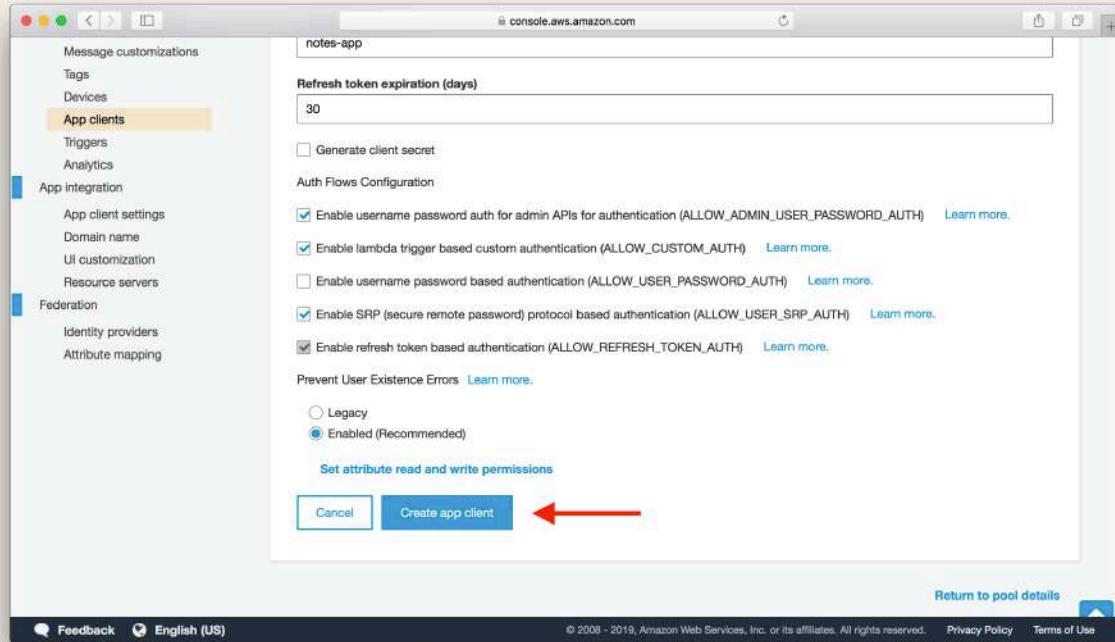
Enter **App client name**, un-select **Generate client secret**, select **Enable sign-in API for server-based authentication**, then select **Create app client**.

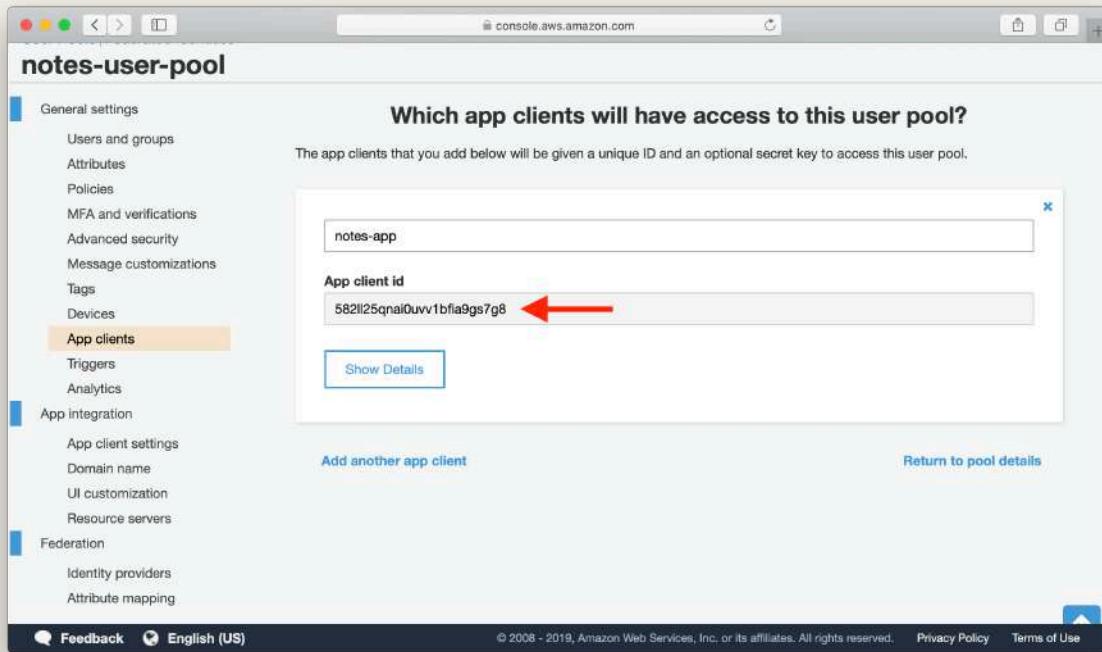
- **DISABLE client secret generation:** user pool apps with a client secret are not supported by the JavaScript SDK. We need to un-select the option.
- **Enable username password auth for admin APIs for authentication:** required by AWS CLI when managing the pool users via command line interface. We will be creating a test user through the command line interface in the next chapter.



Fill Cognito User Pool App Info Screenshot

Now select **Create app client**.

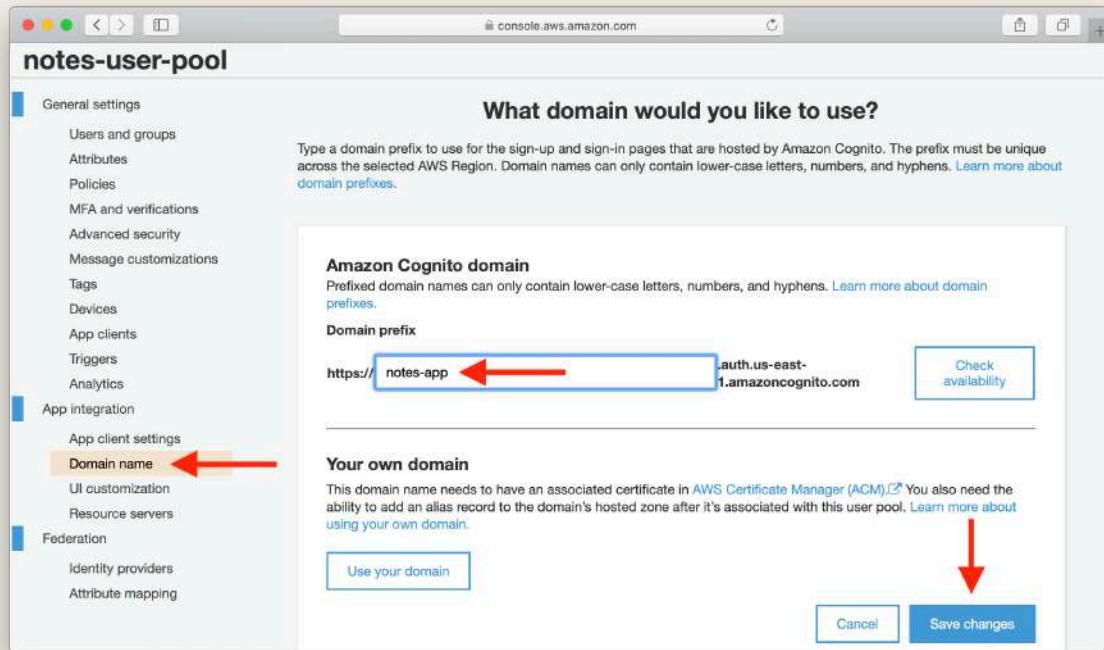




Cognito User Pool App Created Screenshot

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.



Select Congito User Pool Apps Screenshot

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.



Help and discussion

View the [comments](#) for this chapter on our forums

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.



Help and discussion

View the [comments](#) for this chapter on our forums

Set up the Serverless Framework

We are going to be using [AWS Lambda](#) and [Amazon 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](#) 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](#), a package manager for JavaScript. Follow [this](#) 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-api
```

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

```
$ cd notes-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@7.0.3 --save
```

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

Update Service Name

Let's change the name of our service from the one in the starter.

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

```
service: notes-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: nodejs12.x
  stage: prod
  region: us-east-1
```

The service name is pretty important. We are calling our service the notes-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 that we've included — serverless-bundle, serverless-offline, and serverless-dotenv-plugin. The `serverless-offline` plugin is helpful for local development. While the `serverless-dotenv-plugin` will be used later to load the `.env` files as Lambda environment variables.

On the other hand, we use the `serverless-bundle` plugin to allow us to write our Lambda functions using a flavor of JavaScript that's similar to the one we'll be using in our frontend React app.

Let's look at this in detail.



Help and discussion

View the [comments for this chapter on our forums](#)

Add Support for ES6 and TypeScript

AWS Lambda supports Node.js v10.x and v12.x. However, the supported syntax is a little different when compared to the more advanced ECMAScript flavor of JavaScript that our frontend React app supports. 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.

Additionally, our frontend React app automatically supports TypeScript, via [Create React App](#). And while we are not using TypeScript in this guide, it makes sense to have a similar setup for your backend Lambda functions. So you can use it in your future projects.

To do this we typically need to install [Babel](#), [TypeScript](#), [Webpack](#), and a long list of other packages. This can add a ton of extra config and complexity to your project.

To help with this we created, [serverless-bundle](#). This is a Serverless Framework plugin that has a few key advantages:

- Only one dependency
- Supports ES6 and TypeScript
- Generates optimized packages
- Linting Lambda functions using [ESLint](#)
- Supports transpiling unit tests with [babel-jest](#)
- Source map support for proper error messages

It's automatically included in the starter project we used in the previous chapter – '[serverless-nodejs-starter](#)'. For TypeScript, we have a starter for that as well – [serverless-typescript-starter](#).

However, if you are looking to add ES6 and TypeScript support to your existing Serverless Framework projects, you can do this by installing [serverless-bundle](#):

```
$ npm install --save-dev serverless-bundle
```

And including it in your `serverless.yml` using:

```
plugins:  
  - serverless-bundle
```

To run your tests, add this to your `package.json`.

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

Optimized Packages

By default Serverless Framework creates a single package for all your Lambda functions. This means that when a Lambda function is invoked, it'll load all the code in your app. Including all the other Lambda functions. This negatively affects performance as your app grows in size. The larger your Lambda function packages, the longer [the cold starts](#).

To turn this off and ensure that Serverless Framework is packaging our functions individually, add the following to your `serverless.yml`.

```
package:  
  individually: true
```

This should be on by default in our starter project.

Note that, with the above option enabled, `serverless-bundle` can use Webpack to generate optimized packages using a [tree shaking algorithm](#). It'll only include the code needed to run your Lambda function and nothing else!

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



Help and discussion

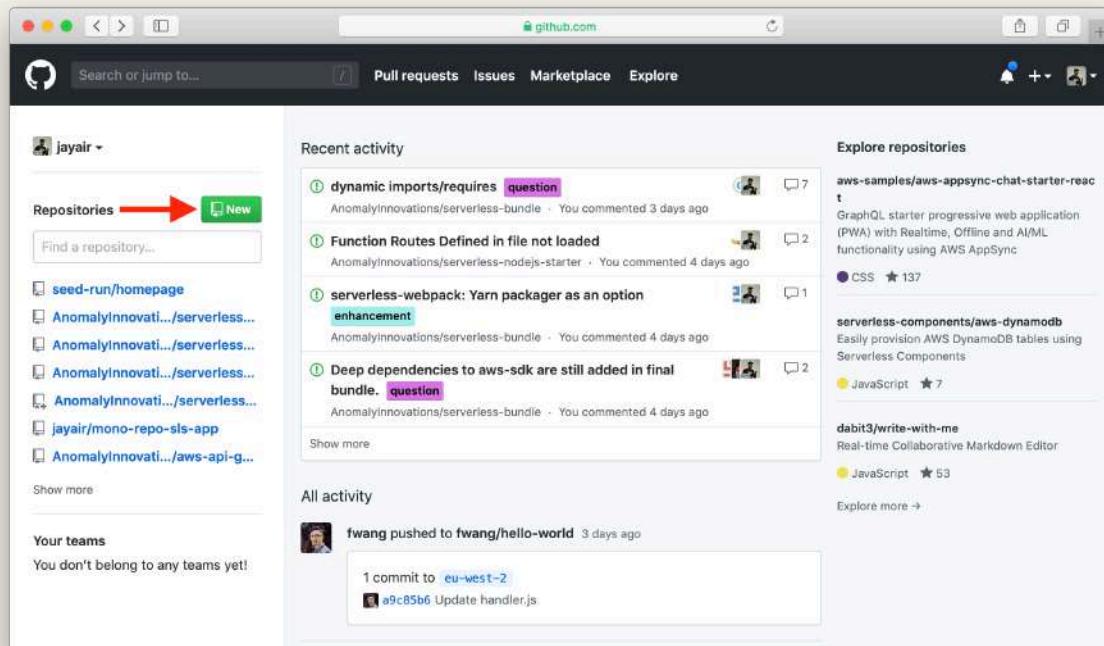
View the [comments for this chapter](#) on our forums

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.

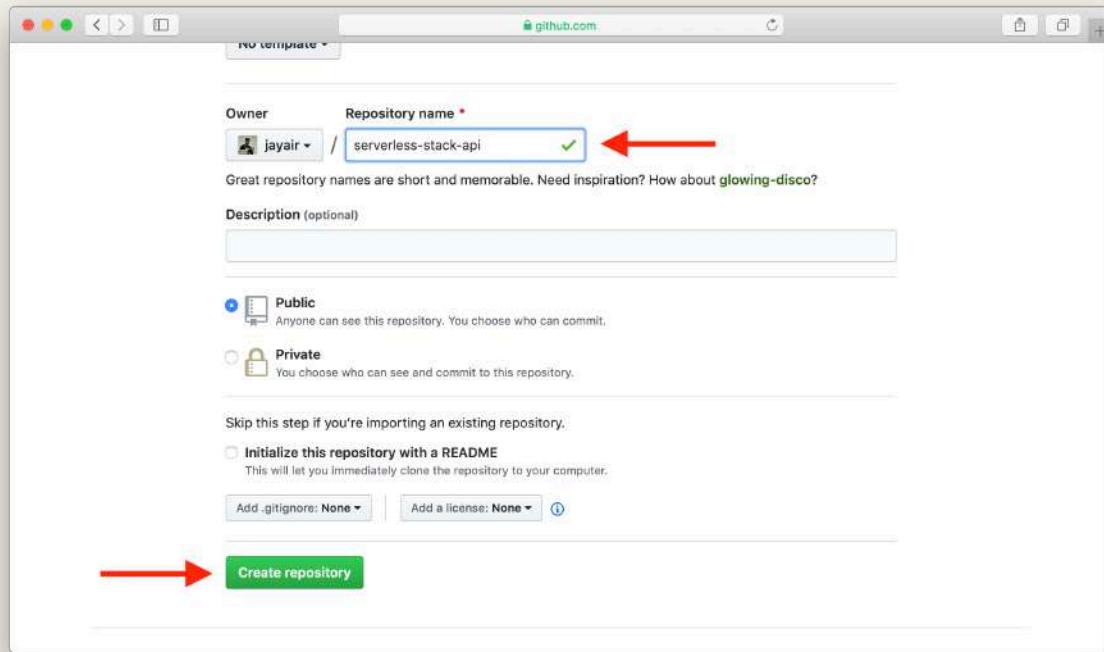
Create a New Github Repo

Let's head over to [GitHub](#). Make sure you are signed in and hit **New repository**.



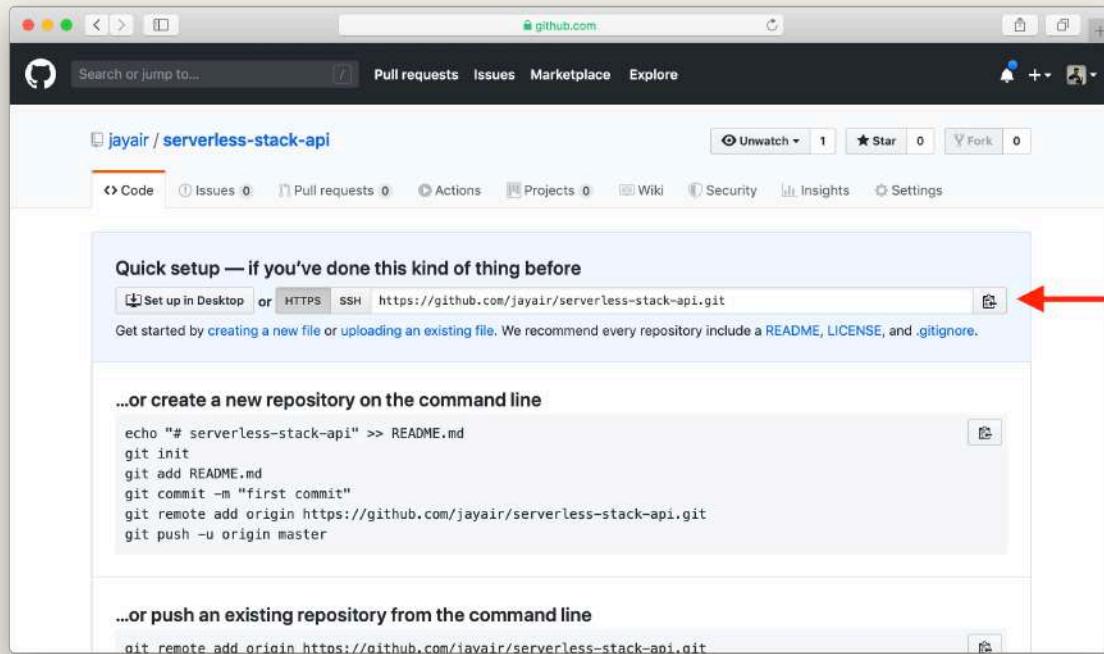
Create new GitHub repository screenshot

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



Name new GitHub repository screenshot

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



Copy new GitHub repo url screenshot

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!



Help and discussion

View the [comments for this chapter](#) on our forums



For reference, here is the code we are using

Backend Source: [initialize-the-backend-repo](#)

Building a Serverless REST API

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 * as 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 initializing 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](#) 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-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: nodejs12.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:Scan
      - dynamodb:Query
      - dynamodb:GetItem
      - dynamodb:PutItem
      - dynamodb:UpdateItem
      - dynamodb:DeleteItem
      - dynamodb:DescribeTable
    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](#). We discuss this and a few other patterns in the chapter on [organizing Serverless Framework projects](#). 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](#) 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 Start by replacing our create.js with the following.

```
import * as uuid from "uuid";
import handler from "./libs/handler-lib";
import dynamoDb from "./libs/dynamodb-lib";

export const main = handler(async (event, context) => {
  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()
    }
};
```

```
await dynamoDb.put(params);  
  
return params.Item;  
});
```

This code doesn't work just yet but it shows you what we want to accomplish:

- We want to make our Lambda function `async`, and simply return the results. Without having to call the `callback` method.
- We want to simplify how we make calls to DynamoDB. We don't want to have to create a new `AWS.DynamoDB.DocumentClient()`. We also want to use `async/await` when working with our database calls.
- We want to centrally handle any errors in our Lambda functions.
- Finally, since all of our Lambda functions will be handling API endpoints, we want to handle our HTTP responses in one place.

To do all of this let's first create our `dynamodb-lib`.

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

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

◆ CHANGE Create a `libs/dynamodb-lib.js` file with:

```
import AWS from "aws-sdk";  
  
const client = new AWS.DynamoDB.DocumentClient();  
  
export default {  
  get: (params) => client.get(params).promise(),  
  put: (params) => client.put(params).promise(),  
  query: (params) => client.query(params).promise(),  
  update: (params) => client.update(params).promise(),  
  delete: (params) => client.delete(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. And we are exposing the DynamoDB client methods that we are going to need in this guide.

◆ CHANGE Also create a `libs/handler-lib.js` file with the following.

```
export default function handler(lambda) {
  return async function (event, context) {
    let body, statusCode;

    try {
      // Run the Lambda
      body = await lambda(event, context);
      statusCode = 200;
    } catch (e) {
      body = { error: e.message };
      statusCode = 500;
    }

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

Let's go over this in detail.

- We are creating a `handler` function that we'll use as a wrapper around our Lambda functions.
- It takes our Lambda function as the argument.
- We then run the Lambda function in a `try/catch` block.
- On success, we `JSON.stringify` the result and return it with a `200` status code.
- If there is an error then we return the error message with a `500` status code.

It's **important to note** that the `handler-lib.js` needs to be **imported before we import anything else**. This is because we'll be adding some error handling to it later that needs to be initialized when our Lambda function is first invoked.

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` in our `libs/handler-lib.js`, should give you a clue about what the issue is.

```
    } catch (e) {  
        // Print out the full error  
        console.log(e);  
  
        body = { error: e.message };  
        statusCode = 500;  
    }
```



Help and discussion

View the [comments](#) for this chapter on our forums

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



Create a new file `get.js` in your project root and paste the following code:

```
import handler from "./libs/handler-lib";
import dynamoDb from "./libs/dynamodb-lib";

export const main = handler(async (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
    }
  };

  const result = await dynamoDb.get(params);
  if (!result.Item) {
    throw new Error("Item not found.");
  }

  // Return the retrieved item
```

```
    return result.Item;  
});
```

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

Configure the API Endpoint



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

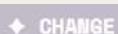
```
get:  
  # Defines an HTTP API endpoint that calls the main function in get.js  
  # - path: url path is /notes/{id}  
  # - method: GET request  
  handler: get.main  
  events:  
    - http:  
        path: notes/{id}  
        method: get  
        cors: true  
        authorizer: aws_iam
```

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.



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 invoke our newly created function from the root directory of the project.

```
$ 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.



Help and discussion

View the [comments](#) for this chapter on our forums

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



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

```
import handler from "./libs/handler-lib";
import dynamoDb from "./libs/dynamodb-lib";

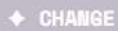
export const main = handler(async (event, context) => {
  const params = {
    TableName: process.env.tableName,
    // 'KeyConditionExpression' defines the condition for the query
    // - 'userId = :userId': only return items with matching 'userId'
    //   partition key
    // 'ExpressionAttributeValues' 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
    }
  };

  const result = await dynamoDb.query(params);

  // Return the matching list of items in response body
  return result.Items;
})
```

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



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



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.



Help and discussion

View the [comments for this chapter](#) on our forums

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 handler from "./libs/handler-lib";
import dynamoDb from "./libs/dynamodb-lib";

export const main = handler(async (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"
};

await dynamoDb.update(params);

return { status: true };
});
```

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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



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

```
import handler from "./libs/handler-lib";
import dynamoDb from "./libs/dynamodb-lib";

export const main = handler(async (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
    }
  };

  await dynamoDb.delete(params);

  return { status: true };
});
```

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.



Help and discussion

View the [comments](#) for this chapter on our forums

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.

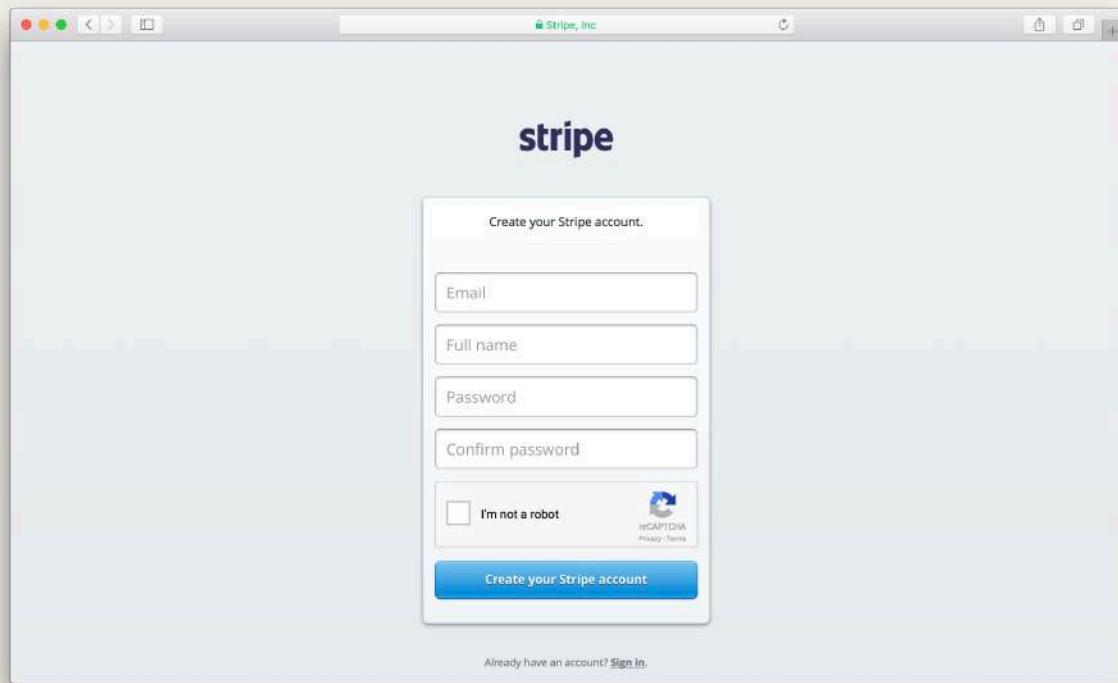


Help and discussion

View the [comments for this chapter on our forums](#)

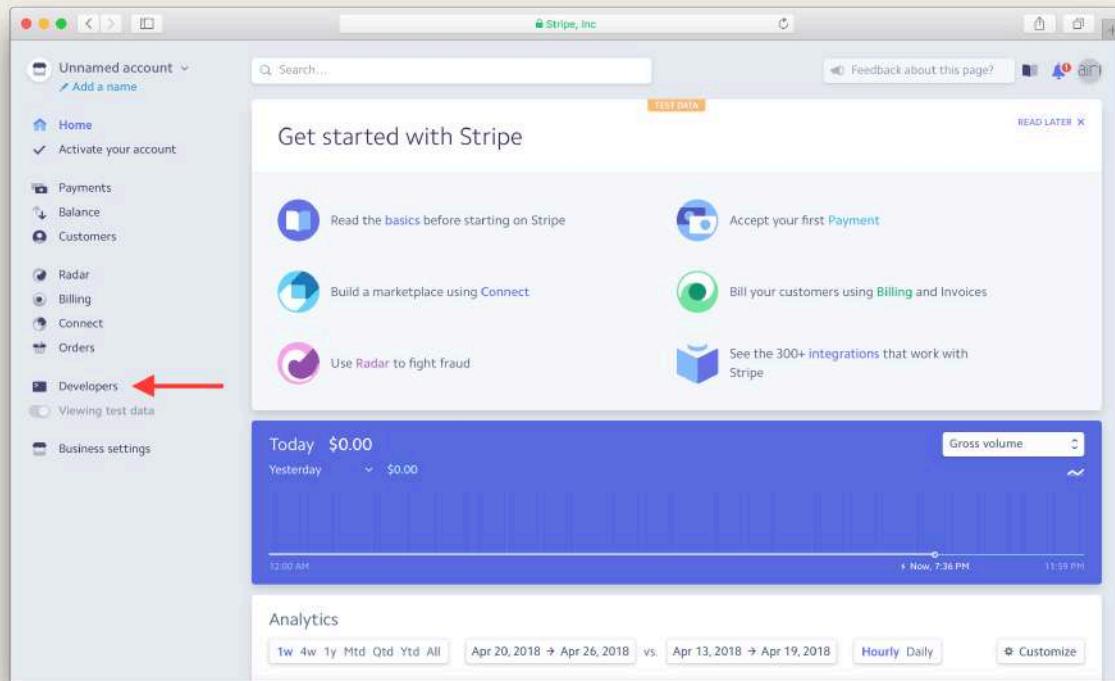
Setup a Stripe Account

Let's start by creating a free Stripe account. Head over to [Stripe](#) and register for an account.



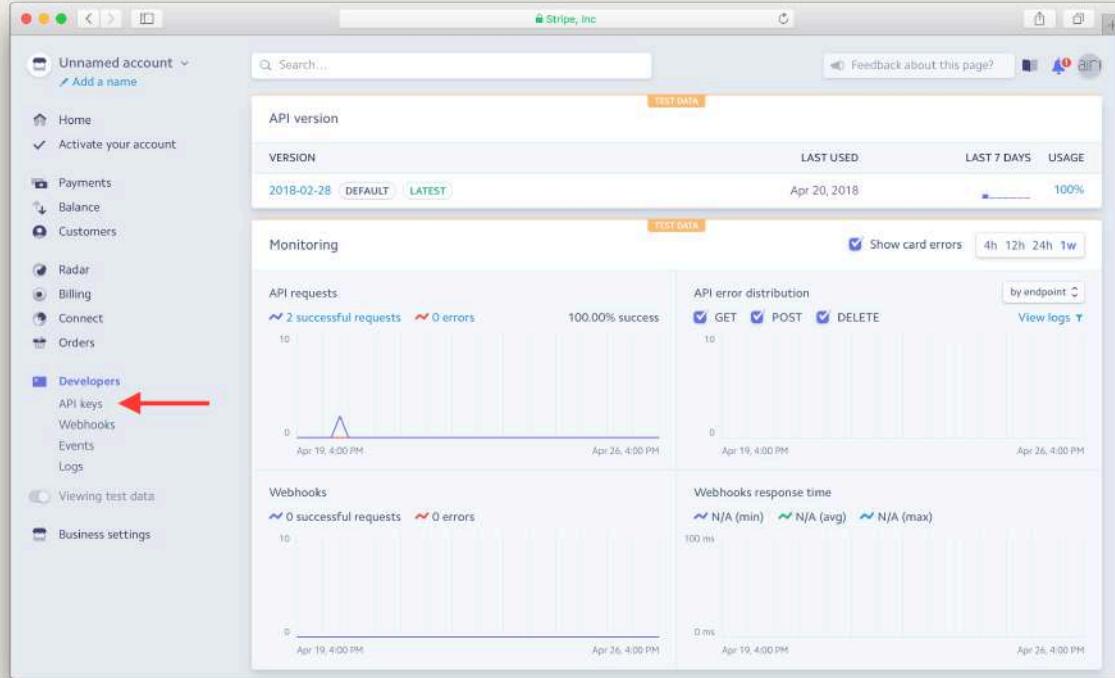
Create a Stripe account screenshot

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



Stripe dashboard screenshot

And hit **API keys**.

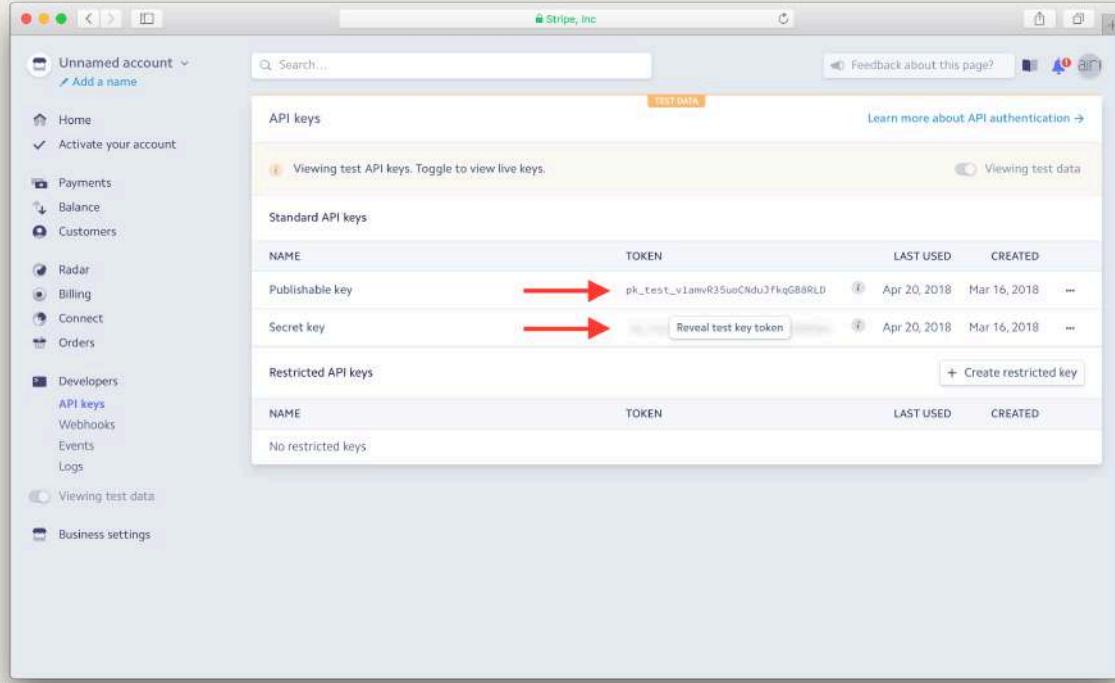


Developer section in Stripe dashboard screenshot

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**.



Stripe dashboard Stripe API keys screenshot

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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 Create a new file called 'billing.js' with the following.

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

export const main = handler(async (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);

  await stripe.charges.create({
    source,
    amount,
    description,
    currency: "usd"
```

```
});  
return { status: true };  
});
```

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.

Note, if you are testing this from India, you'll need to add some shipping information as well. Check out the [details from our forums](#).

Add the Business Logic

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

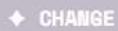
◆ CHANGE Create a libs/billing-lib.js and add the following.

```
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.



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.



Help and discussion

View the [comments for this chapter](#) on our forums

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](#) chapter.

We are using the `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 committing 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 loaded as an environment variable when we test our code locally.
- This allows us to add a Lambda environment variable 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 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.



Help and discussion

View the [comments for this chapter on our forums](#)

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](#).

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.

**Help and discussion**

View the [comments](#) for this chapter on our forums

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](#).

We are going to use [Jest](#) for this and it is already a part of [our starter project](#).

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:

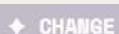
```
"scripts": {  
  "test": "jest"  
},
```

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

Alternatively, if you are using the [serverless-bundle](#) 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



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](#).

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 command in the root of our project.

```
$ 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  
  \faCheck Lowest tier (4ms)  
  \faCheck Middle tier  
  \faCheck 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.



Help and discussion

View the [comments for this chapter on our forums](#)

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](#) 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](#) supports [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



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

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



Help and discussion

View the [comments](#) for this chapter on our forums



For reference, here is the code we are using

Backend Source: [handle-api-gateway-cors-errors](#)

Deploying the backend

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](#) chapter.

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

```
Service Information
service: notes-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-api-prod-create  
get: notes-api-prod-get  
list: notes-api-prod-list  
update: notes-api-prod-update  
delete: notes-api-prod-delete  
billing: notes-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](#) over on [Seed](#).

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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.

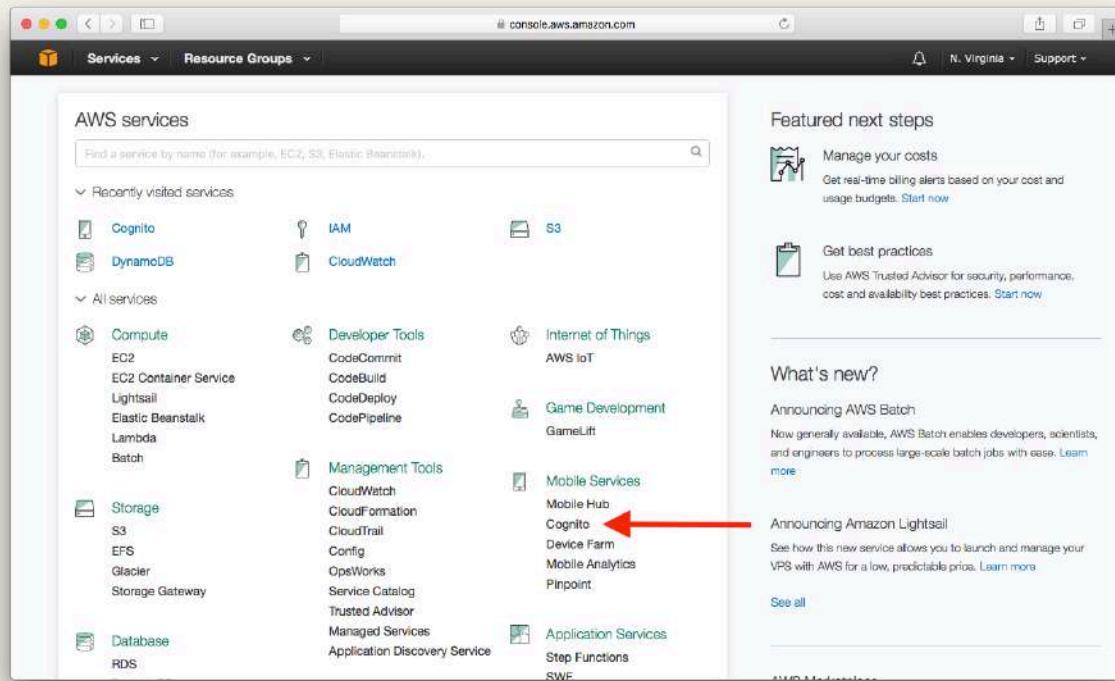
Amazon Cognito Federated Identities enables developers to create unique identities for your users and authenticate them with federated identity providers. With a federated identity, you can obtain temporary, limited-privilege AWS credentials to securely access other AWS services such as Amazon DynamoDB,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

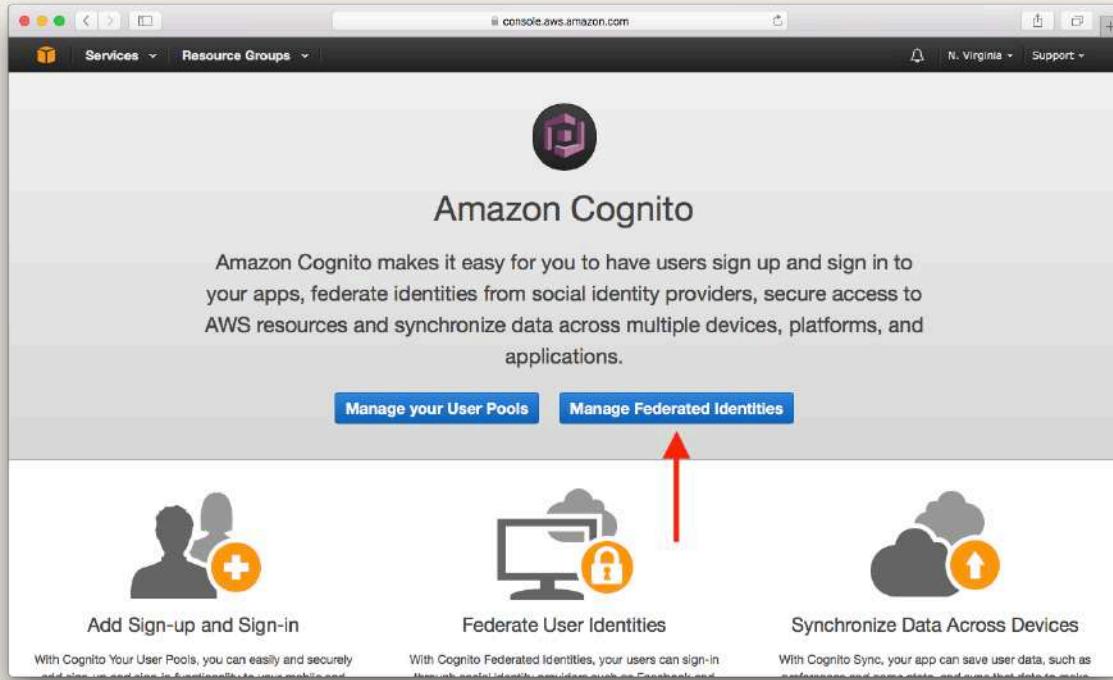
Create Pool

From your [AWS Console](#) and select **Cognito** from the list of services.



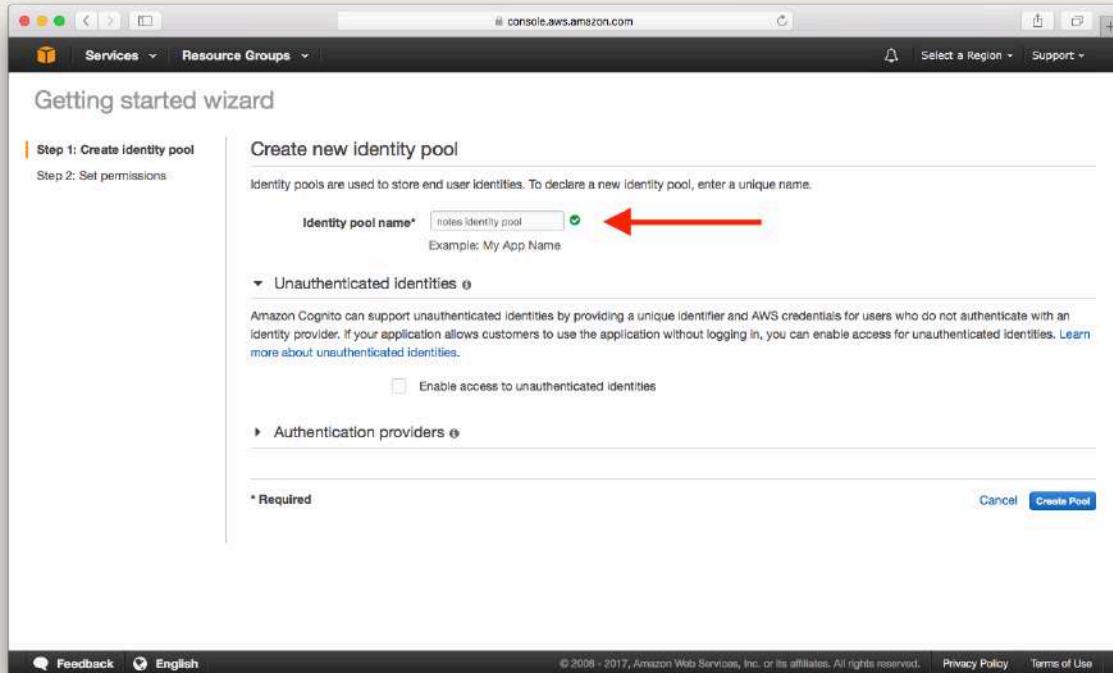
Select Cognito Service screenshot

Select **Manage Federated Identities**.



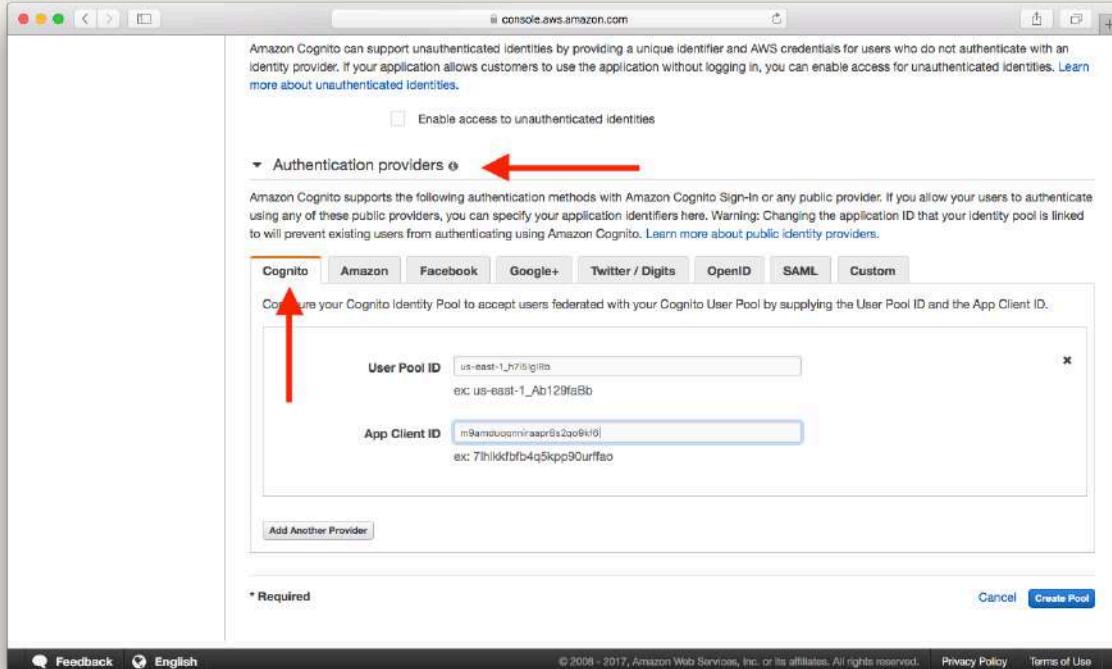
Select Manage Federated Identities Screenshot

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



Fill Cognito Identity Pool Info Screenshot

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](#) chapter. Select **Create Pool**.

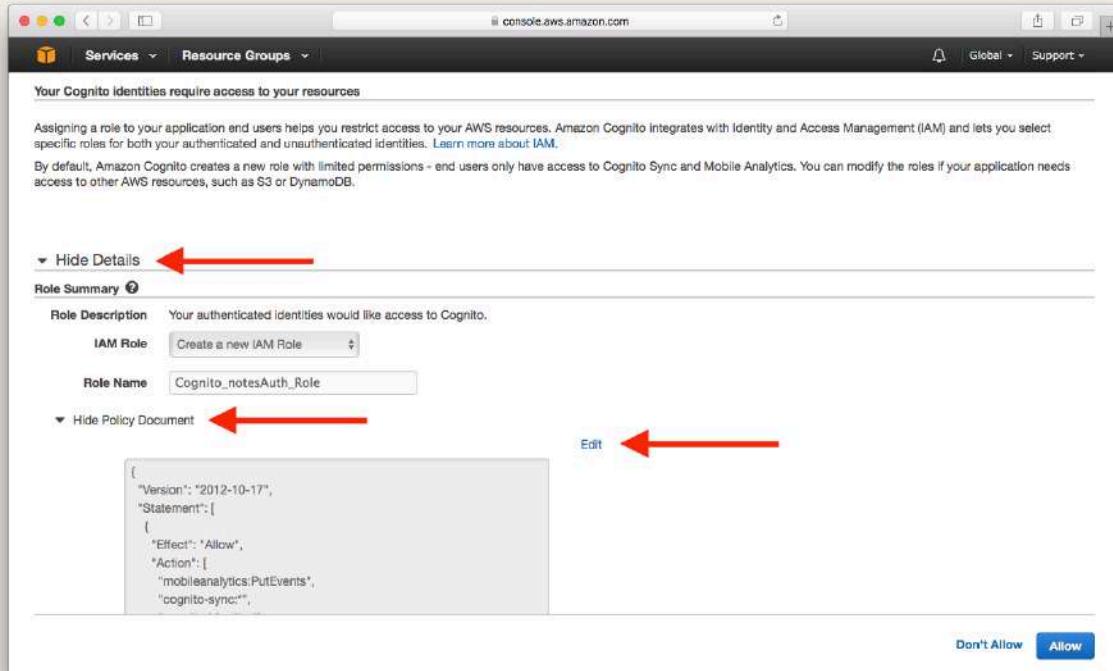


Fill Authentication Provider Info Screenshot

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

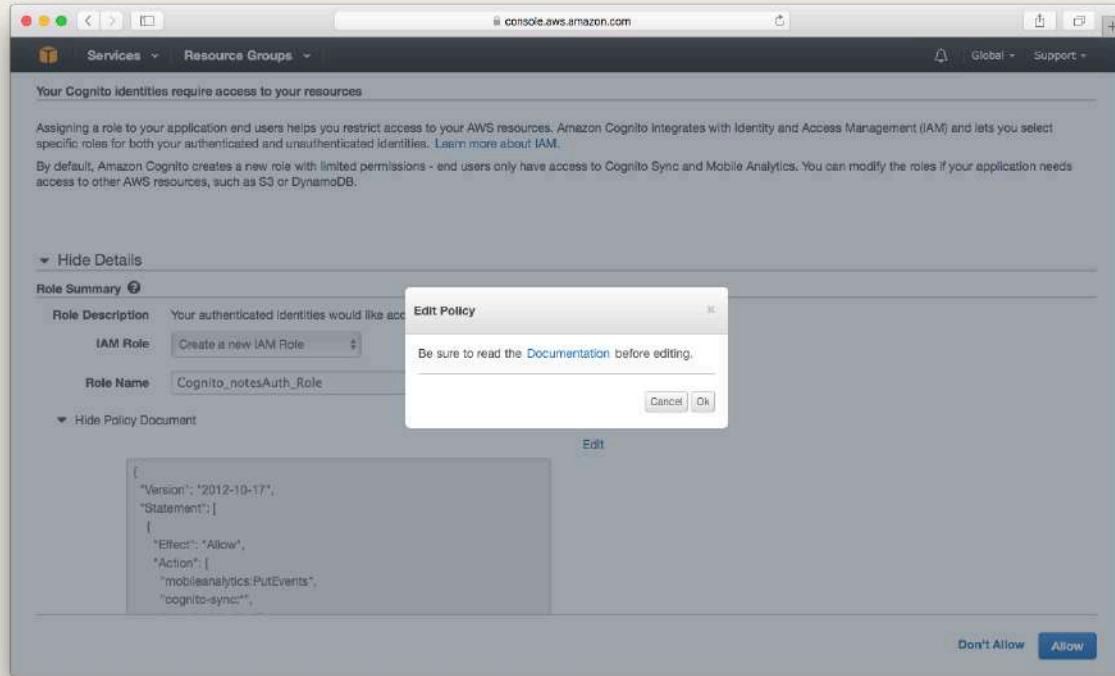
Select **View Details**. Two **Role Summary** sections are expanded. The top section summarizes the permission policy for authenticated users, and the bottom section summarizes that for unauthenticated users.

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



Select Edit Policy Document Screenshot

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



Select Confirm Edit Policy Screenshot

◆ 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](#) 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/${cognito-
            identity.amazonaws.com:sub}/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "execute-api:Invoke"
    ],
    "Resource": [
        "arn:aws:execute-
            api:YOUR_API_GATEWAY_REGION::YOUR_API_GATEWAY_ID/*/*/*"
    ]
}
]
```

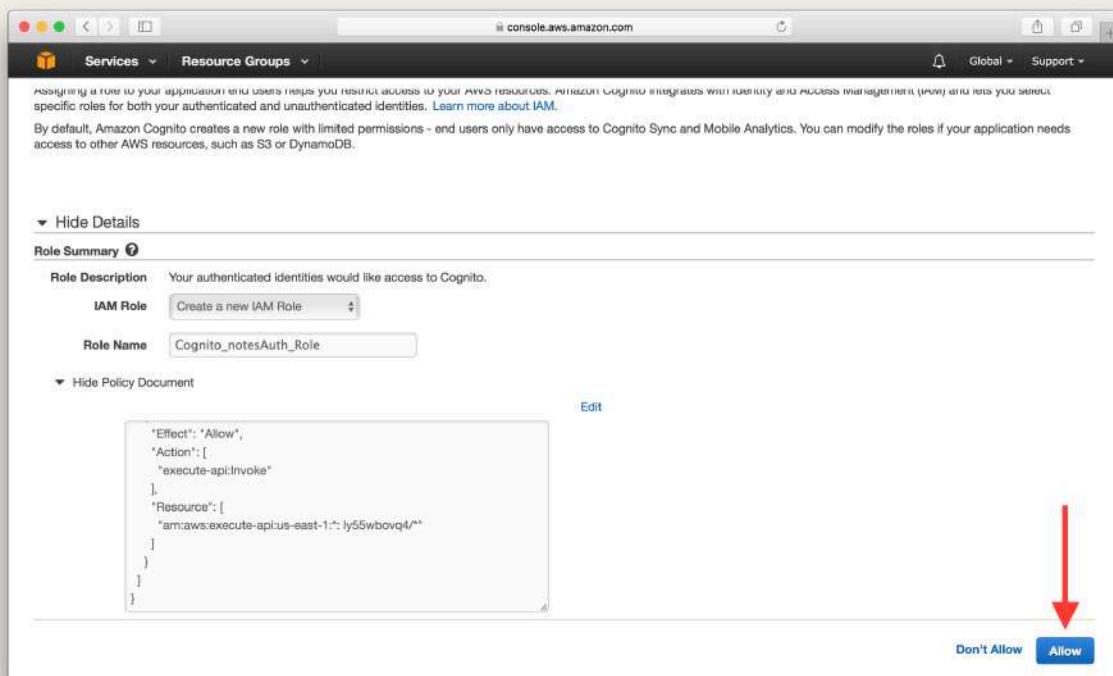
A quick note on the block that relates to the S3 Bucket. In the above policy we are granting our logged in users access to the path `private/${cognito-identity.amazonaws.com:sub}/`. Where `cognito-identity.amazonaws.com:sub` is the authenticated user's federated identity ID (their user id). So a user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

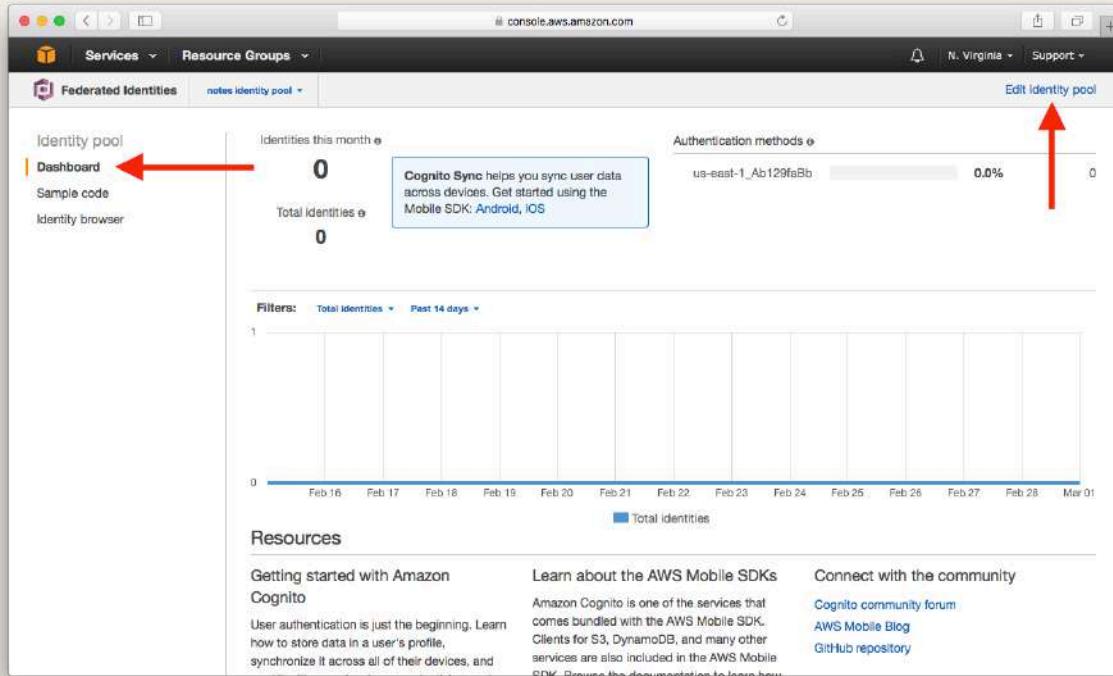
Select **Allow**.



Submit Cognito Identity Pool Policy Screenshot

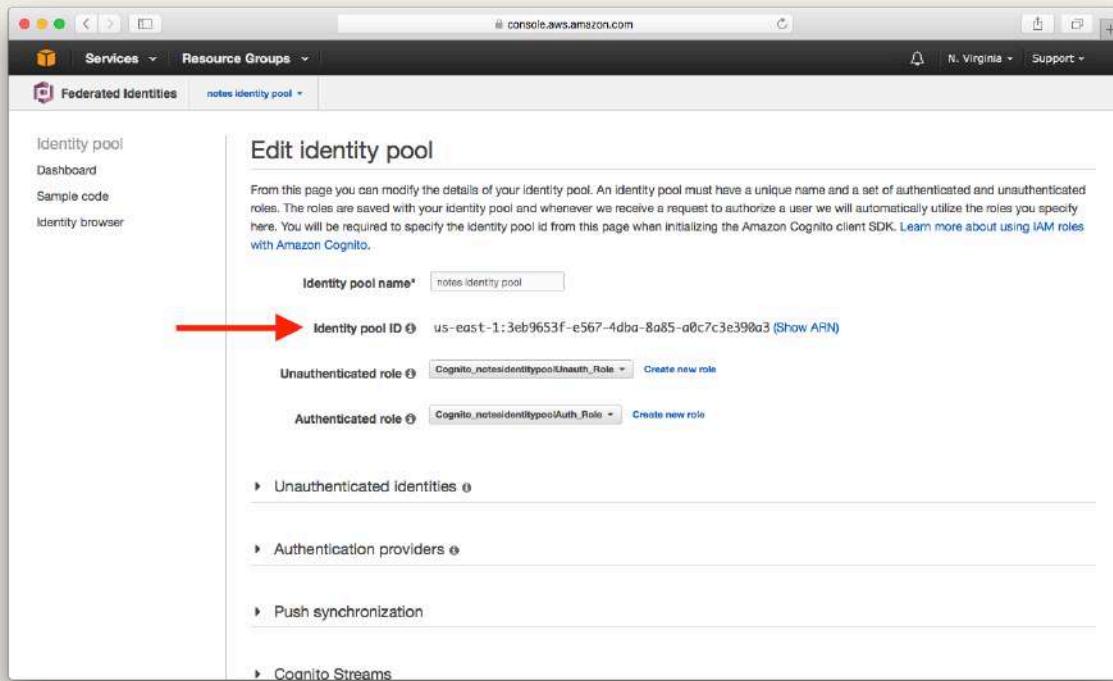
Our Cognito Identity Pool should now be created. Let's find out the Identity Pool ID.

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



Cognito Identity Pool Created Screenshot

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



Cognito Identity Pool Created Screenshot

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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

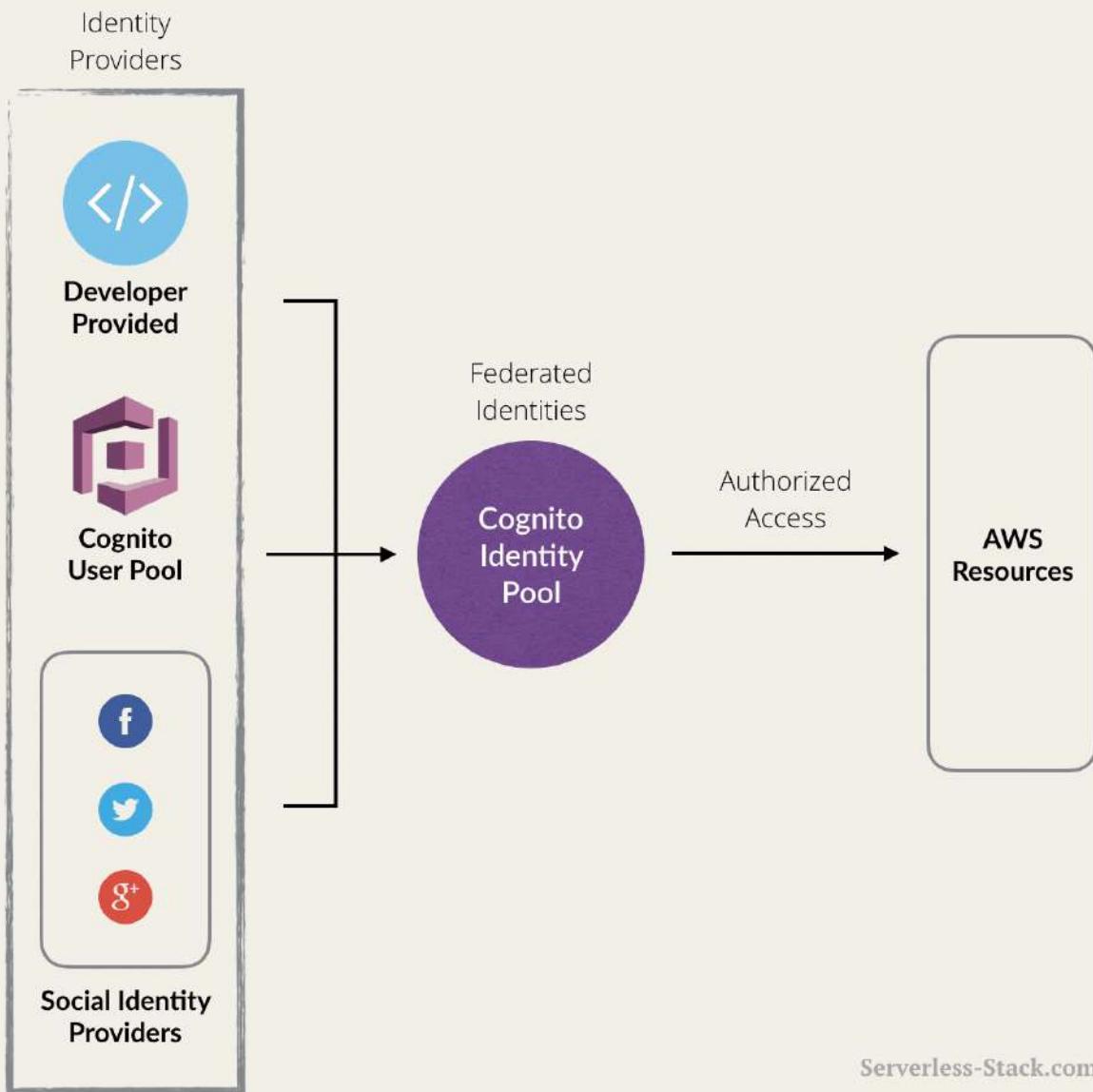
Say you were creating a new web or mobile app and you were thinking about how to handle user registration, authentication, and account recovery. This is where Cognito User Pools would come in. Cognito User Pool handles all of this and as a developer you just need to use the SDK to retrieve user related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



Amazon Cognito User Pool vs Identity Pool screenshot

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.

**Help and discussion**

View the [comments](#) for this chapter on our forums

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](#).

These steps can be a bit tricky to do by hand. So we created a simple tool called [AWS API Gateway Test CLI](#).

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](#) 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](#) 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](#) chapter.
- Use the `YOUR_API_GATEWAY_URL` and `YOUR_API_GATEWAY_REGION` with the ones from the [Deploy the APIs](#) 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](#).

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](#) chapter to ensure that your IAM Roles have the right set of permissions.

Next, you can [enable API Gateway logs](#) and follow [these instructions](#) 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 {error: "Some error message"}

If instead your command fails with the {error: "Some error message"} 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 `libs/handler-lib.js` error handler.

```
// On failure
.catch((e) => {
  console.log(e);
  return [500, { error: e.message }];
})
```

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](#) on working with API Gateway and Lambda logs and you can read about how to check your debug messages [here](#).

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](#).

- '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](#).



Help and discussion

View the [comments for this chapter on our forums](#)

Setting up a React app

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](#). We'll use the [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.

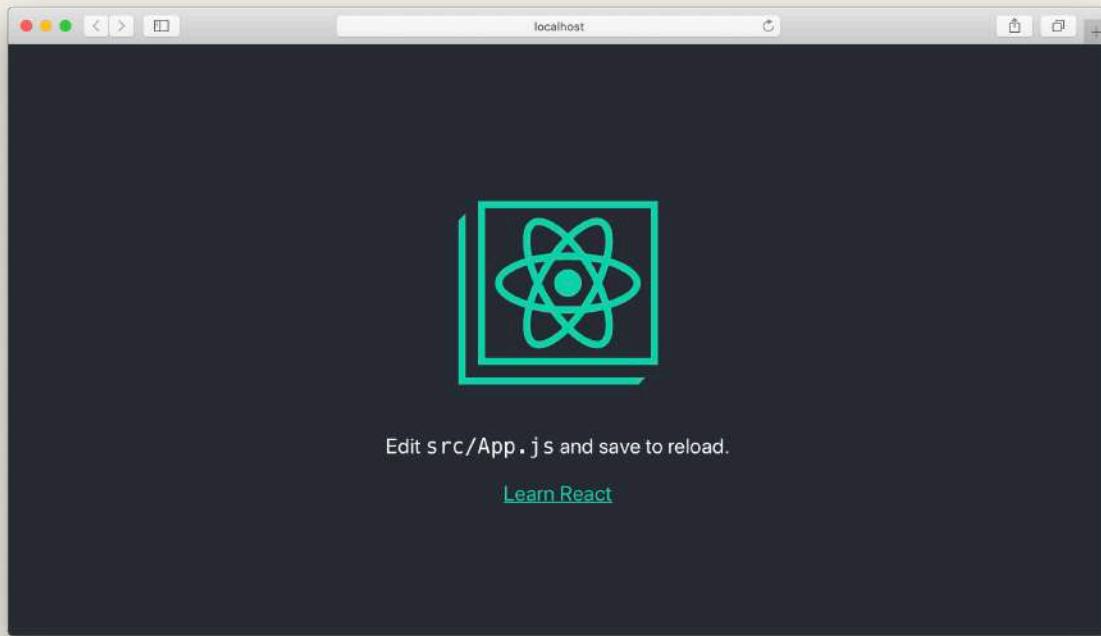
```
$ npx create-react-app notes-app-client --use-npm
```

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.



New Create React App screenshot

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](#).

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.



Help and discussion

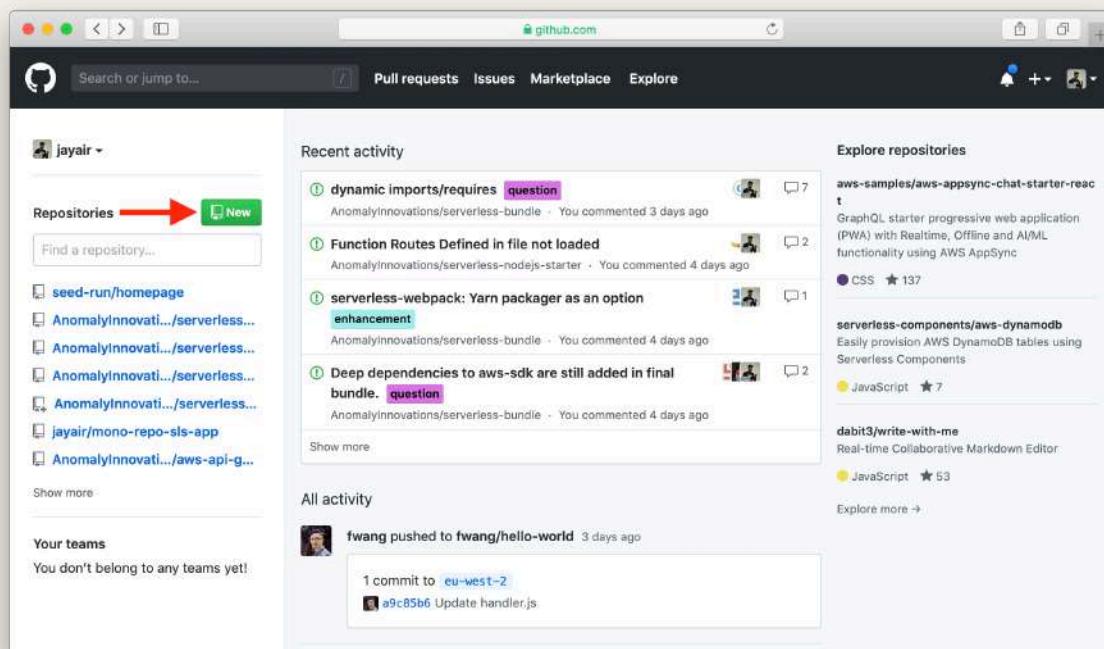
View the [comments for this chapter on our forums](#)

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.

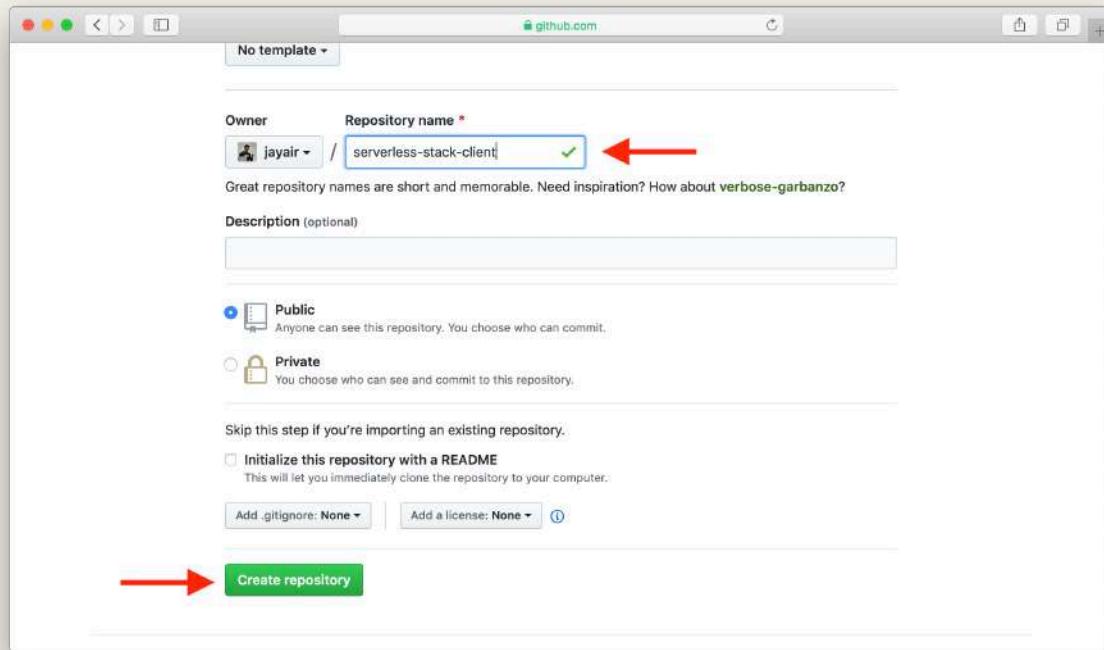
Create a New GitHub Repo

Let's head over to [GitHub](#). Make sure you are signed in and hit **New repository**.



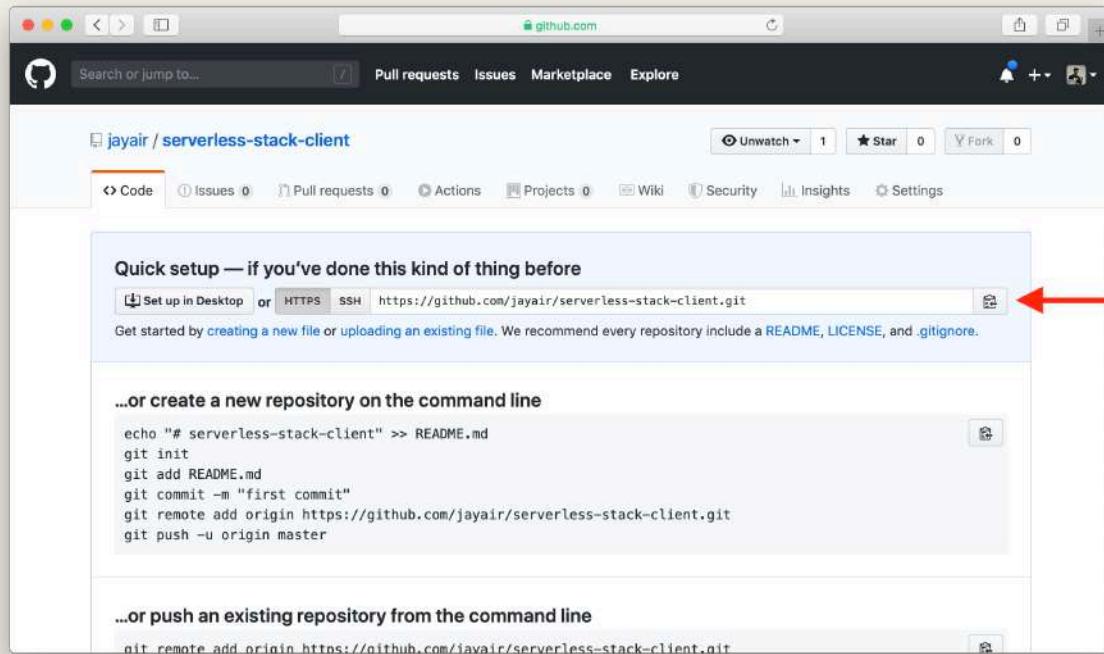
Create new GitHub repository screenshot

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



Name new client GitHub repository screenshot

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



Copy new client GitHub repo url screenshot

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.



Help and discussion

View the [comments for this chapter on our forums](#)



For reference, here is the code we are using

Frontend Source: [initialize-the-frontend-repo](#)

Add App Favicons

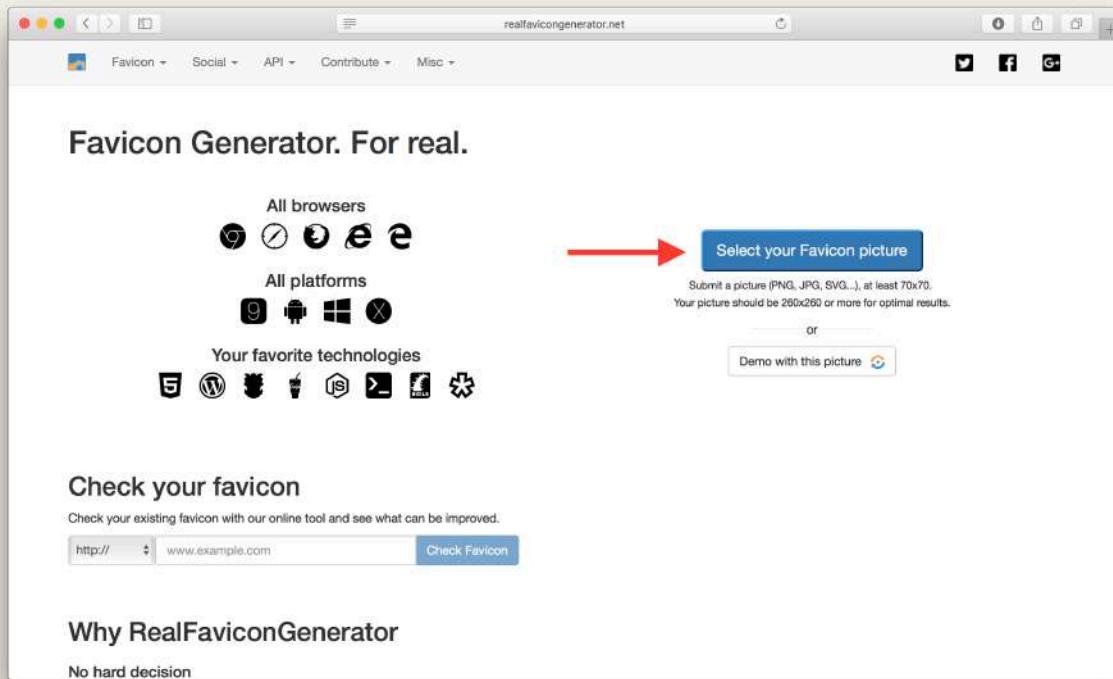
Create React App generates a simple favicon for our app and places it in `public/favicon.ico`. However, getting the favicon to work on all browsers and mobile platforms requires a little more work. There are quite a few different requirements and dimensions. And this gives us a good opportunity to learn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

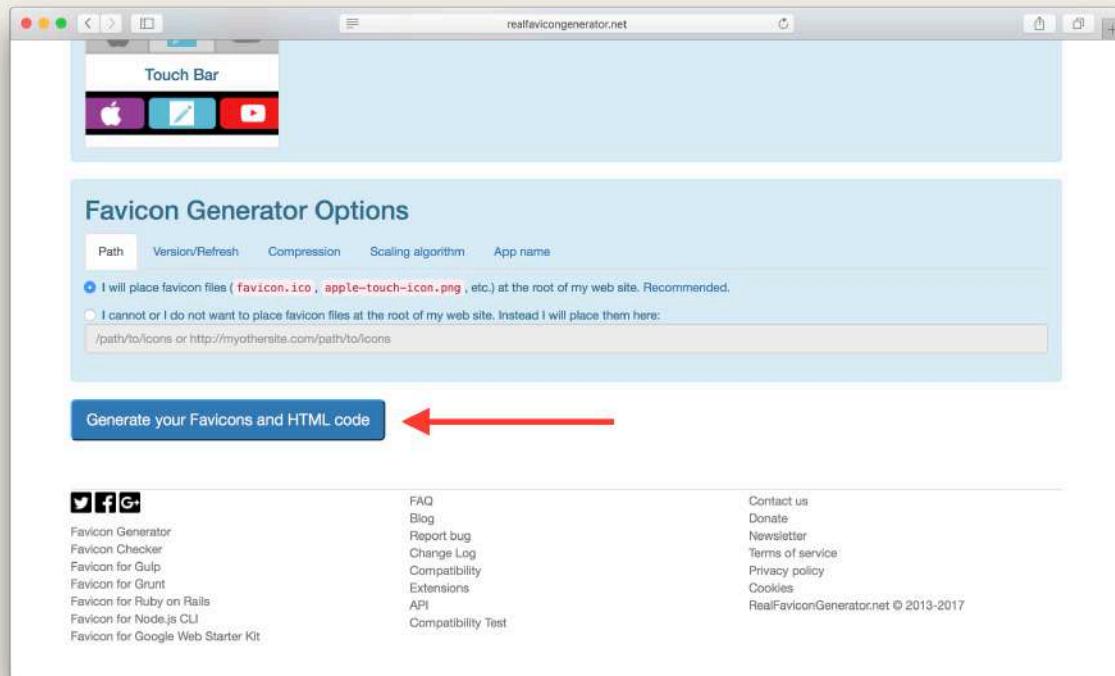
To ensure that our icon works for most of our targeted platforms we'll use a service called the [Favicon Generator](#).

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



Realfavicongenerator.net screenshot

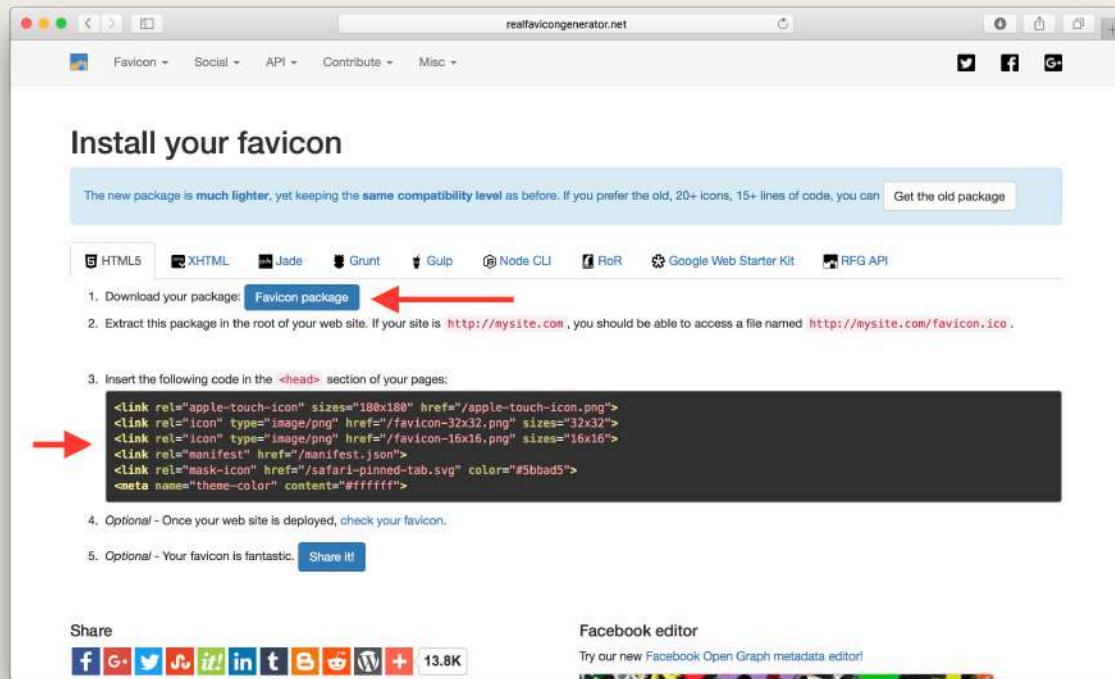
Once you upload your icon, it'll show you a preview of your icon on various platforms. Scroll down the page and hit the **Generate your Favicons and HTML code** button.



Realfavicongenerator.net screenshot

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.



Realfavicongenerator.net completed screenshot

◆ 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": "apple-touch-icon-180x180.png",
      "sizes": "180x180",
      "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.

**Help and discussion**

View the [comments for this chapter on our forums](#)

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](#).

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](#)) and Sans-Serif ([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`.

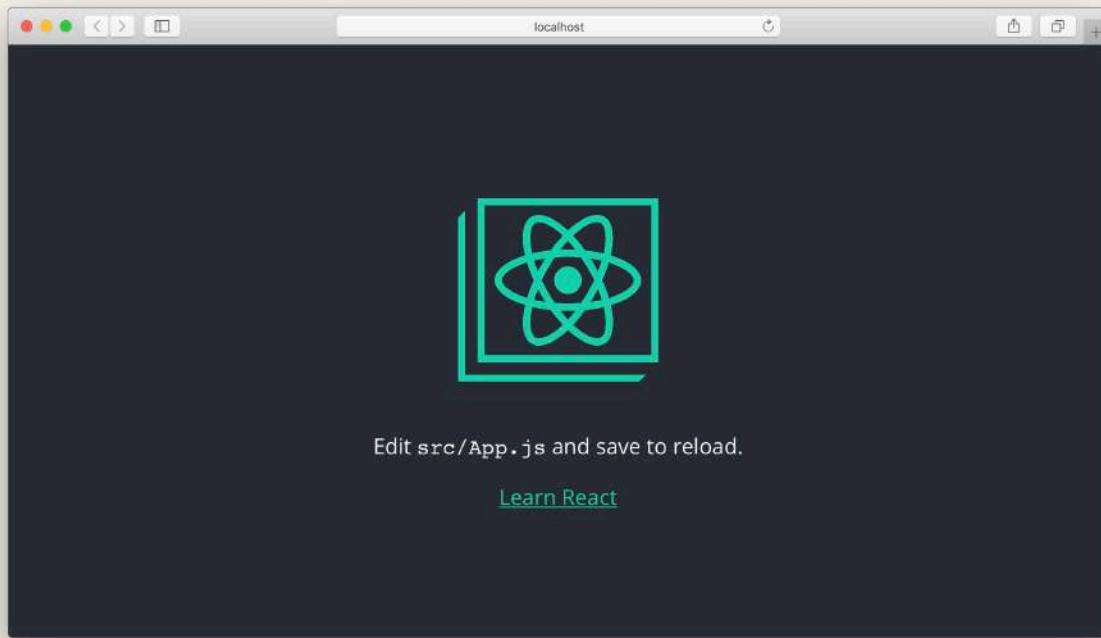
◆ **CHANGE** Let's change the current font in `src/index.css` for the `body` tag to the following.

```
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.



Custom fonts updated screenshot

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.



Help and discussion

View the [comments](#) for this chapter on our forums

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](#) for our note taking app. While Bootstrap can be used directly with React; the preferred way is to use it with the [React-Bootstrap](#) 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@0.33.1 --save
```

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

Note that, we'll be upgrading the guide to React Bootstrap v1 once it is out of beta.

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/bootstrap.min.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](#) 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.



Help and discussion

View the [comments for this chapter](#) on our forums

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](#) 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

◆ CHANGE Run the following command in your working directory.

```
$ npm install react-router-dom@5.1.2 --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` 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.



Help and discussion

View the [comments for this chapter on our forums](#)

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` 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() {
  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).

Let's also add a couple lines of style 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 exact path="/">  
        <Home />  
      </Route>  
    </Switch>  
  );  
}
```

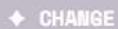
This component uses this `Switch` component from React-Router that renders the first matching route that is defined within it. For now we only have a single route, it looks for / and renders the `Home` component when matched. We are also using the `exact` prop to ensure that it matches the / route exactly.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



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

```
<Routes />
```

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

```
function App() {
  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.



New homepage loaded screenshot

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



Help and discussion

View the [comments for this chapter on our forums](#)

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() {
  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>
  );
}
```

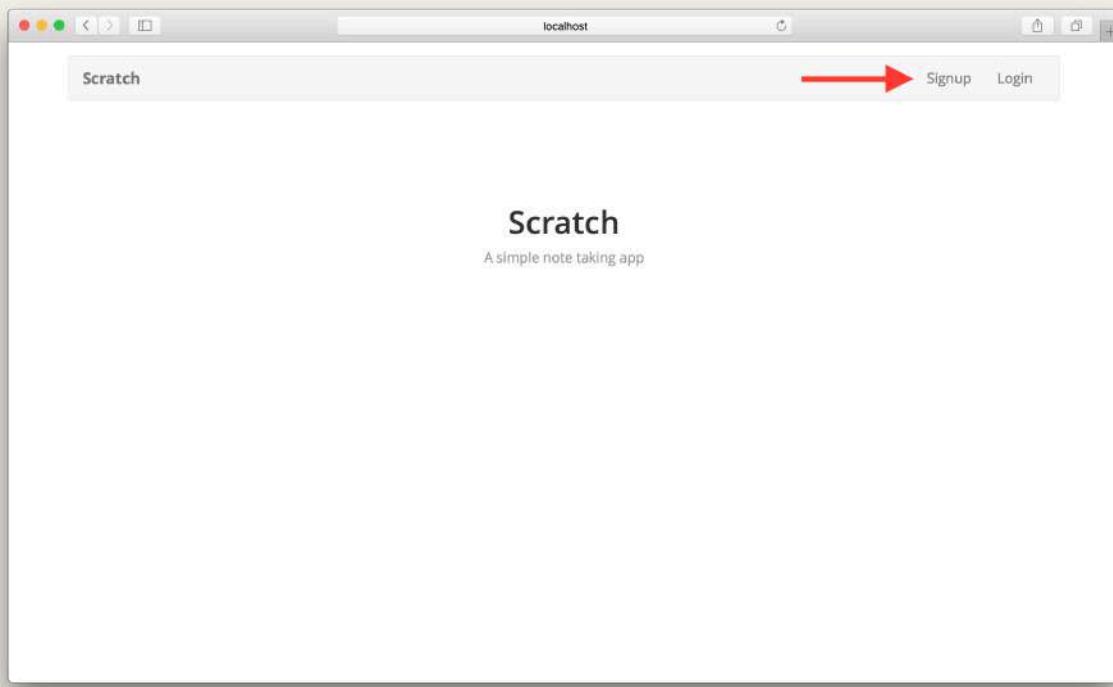
This adds two links to our navbar using the `NavItem` Bootstrap component. The `Navbar.Collapse` component ensures that on mobile devices the two links will be collapsed.

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.



Navbar links added screenshot

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](#). 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
```

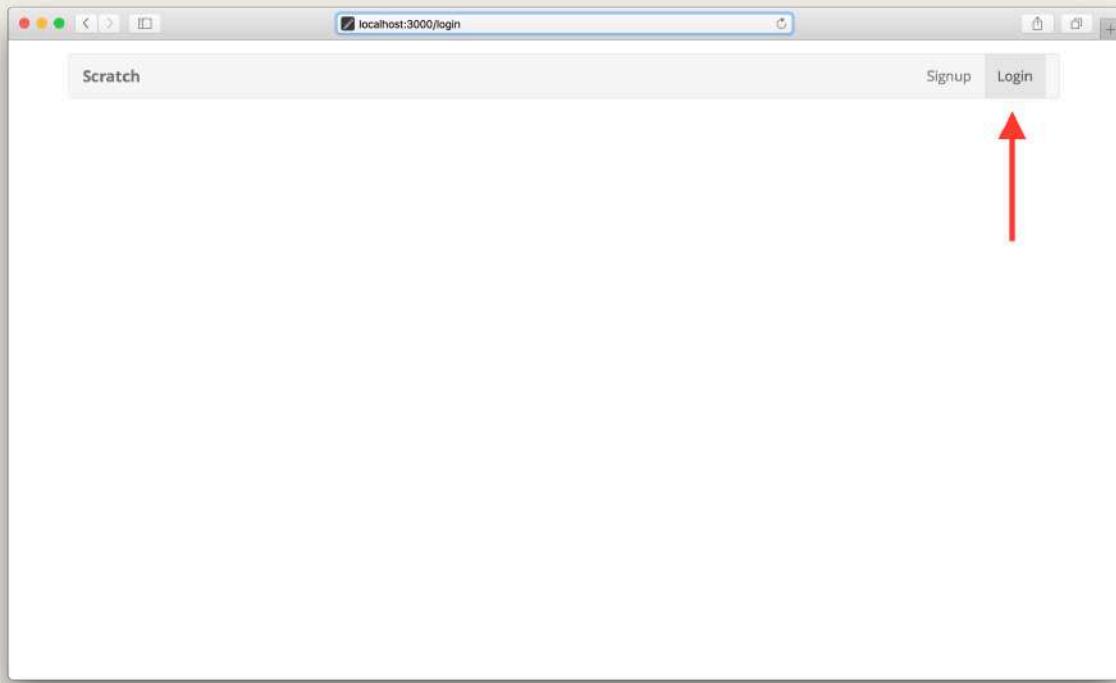
◆ CHANGE And include it at the top of your src/App.js.

```
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() {
  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.



Navbar link highlighted screenshot

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.



Help and discussion

View the [comments for this chapter](#) on our forums

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.

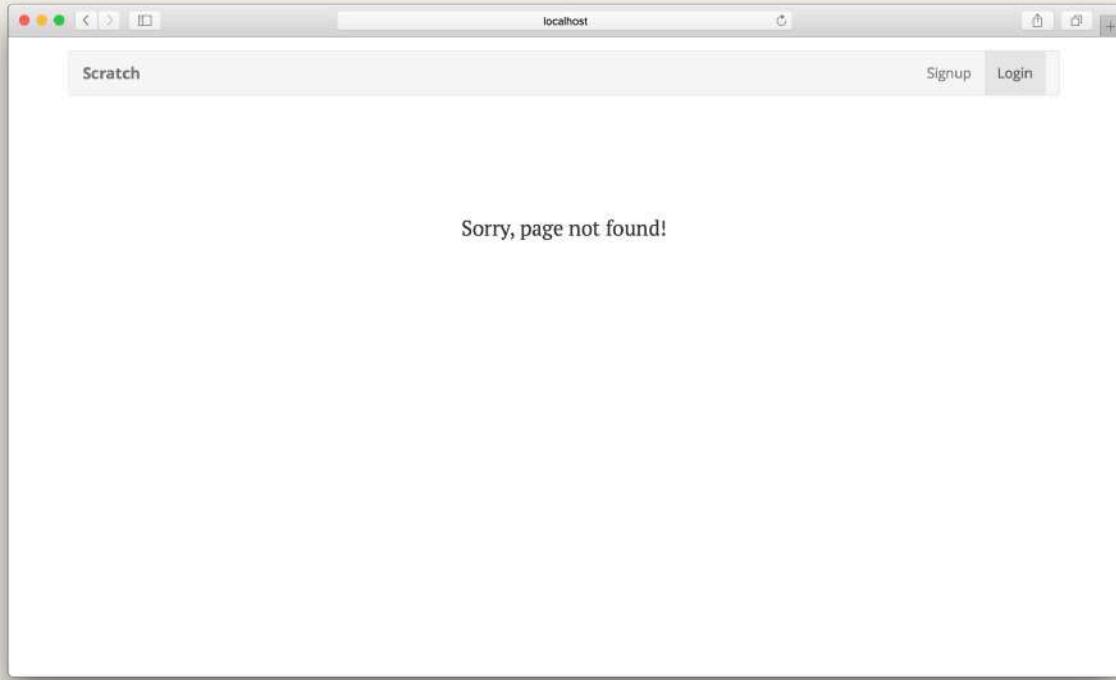
```
{/* Finally, catch all unmatched routes */}
<Route>
  <NotFound />
</Route>
```

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.



Router 404 page screenshot

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



Help and discussion

View the [comments for this chapter on our forums](#)

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](#).

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

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](#) 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](#) 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](#) chapter.
4. YOUR_IDENTITY_POOL_ID with your **Identity pool ID** from the [Create a Cognito identity pool](#) 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.



Help and discussion

View the [comments for this chapter on our forums](#)



For reference, here is the code we are using

Frontend Source: [configure-aws-amplify](#)

Building a React app

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() {
  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 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](#).
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](#).
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 browser's 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

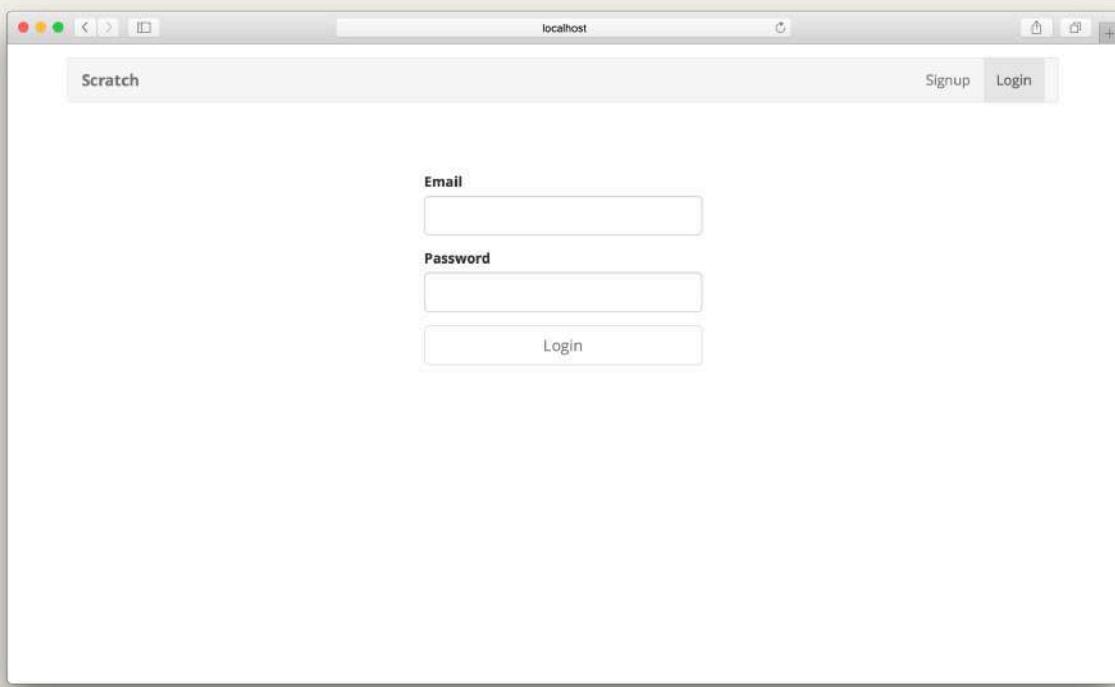
◆ CHANGE Now we link this container up with the rest of our app by adding the following line to `src/Routes.js` below our home `<Route>`.

```
<Route exact path="/login">  
  <Login />  
</Route>
```

◆ 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.



Login page added screenshot

Next, let's connect our login form to our AWS Cognito set up.



Help and discussion

View the [comments for this chapter on our forums](#)

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 Auth module 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.

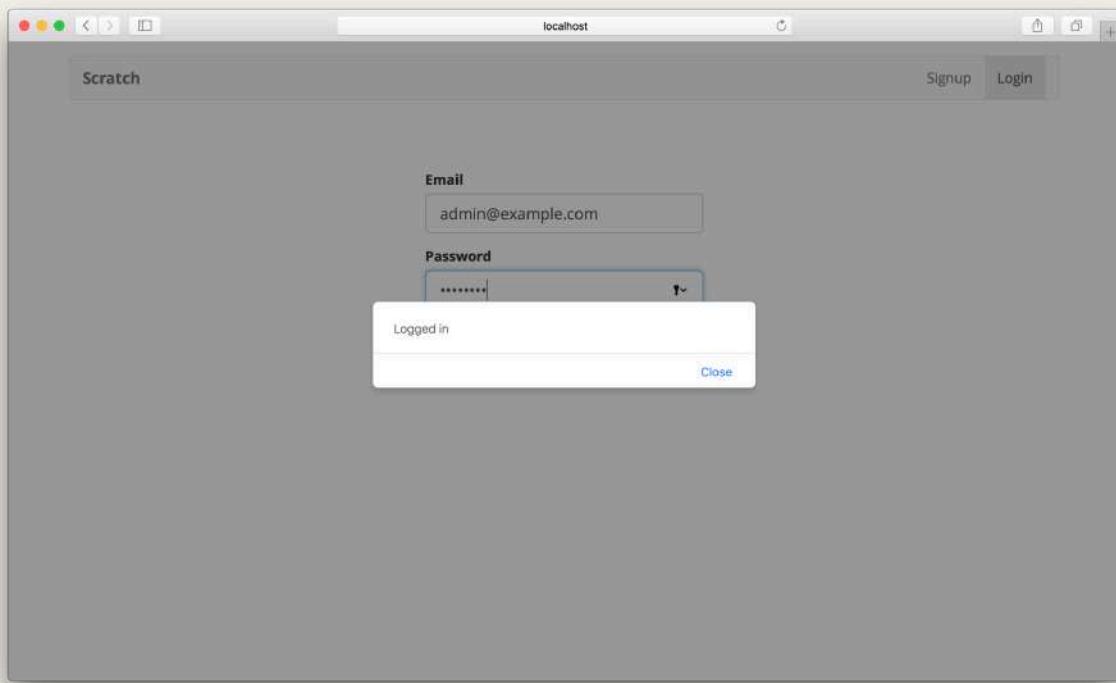
```
async function handleSubmit(event) {
  event.preventDefault();

  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](#) chapter), you should see the browser alert that tells you that the login was successful.



Login success screenshot

Next, we'll take a look at storing the login state in our app.



Help and discussion

View the [comments](#) for this chapter on our forums

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.

Store the Session in the Context

We are going to have to pass the session related info to all of our containers. This is going to be tedious if we pass it in as a prop, since we'll have to do that manually for each component. Instead let's use [React Context](#) for this.

We'll create a context for our entire app that all of our containers will use.

◆ CHANGE Create a `src/libs/` directory. We'll use this to store all our common code.

```
$ mkdir src/libs/
```

◆ CHANGE Add the following to `src/libs/contextLib.js`.

```
import { useContext, createContext } from "react";

export const AppContext = createContext(null);

export function useAppContext() {
  return useContext(AppContext);
}
```

This really simple bit of code is creating and exporting two things: 1. Using the `createContext` API to create a new context for our app. 2. Using the `useContext` React Hook to access the context.

If you are not sure how Contexts work, don't worry, it'll make more sense once we use it.

◆ CHANGE Import our new app context in the header of `src/App.js`.

```
import { AppContext } from "./libs/contextLib";
```

Now to add our session to the context and to pass it to our containers:

◆ CHANGE Wrap our `Routes` component in the `return` statement of `src/App.js`.

```
<Routes />
```

◆ CHANGE With this.

```
<ApplicationContext.Provider value={{ isAuthenticated, userHasAuthenticated }}>
  <Routes />
</ApplicationContext.Provider>
```

React Context's are made up of two parts. The first is the Provider. This is telling React that all the child components inside the Context Provider should be able to access what we put in it. In this case we are putting in the following object:

```
{ isAuthenticated, userHasAuthenticated }
```

Use the Context to Update the State

The second part of the Context API is the consumer. We'll add that to the Login container:

◆ CHANGE Start by importing it in the header of `src/containers/Login.js`.

```
import { useAppContext } from "../libs/contextLib";
```

◆ CHANGE Include the hook by adding it below the `export default function Login() {` line.

```
const { userHasAuthenticated } = useAppContext();
```

This is telling React that we want to use our app context here and that we want to be able to use the `userHasAuthenticated` function.

◆ CHANGE Finally, replace the `alert('Logged in');` line with the following in `src/containers/Login.js`.

```
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:

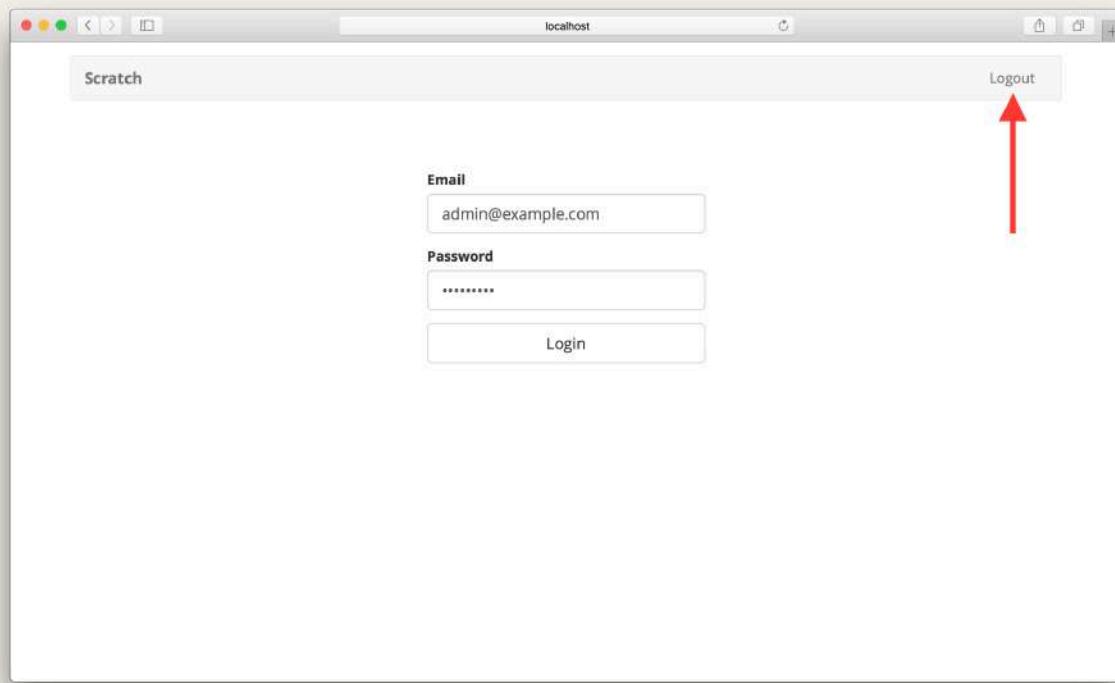
```
{isAuthenticated
? <NavItem onClick={handleLogout}>Logout</NavItem>
: <>
  <LinkContainer to="/signup">
    <NavItem>Signup</NavItem>
  </LinkContainer>
  <LinkContainer to="/login">
    <NavItem>Login</NavItem>
  </LinkContainer>
</>
}
```

The `<>` or [Fragment component](#) 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 `return` statement 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](#) chapter. You should see the Logout button appear right away.



Login state updated screenshot

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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`. 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 takes 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 gets 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() {
  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.

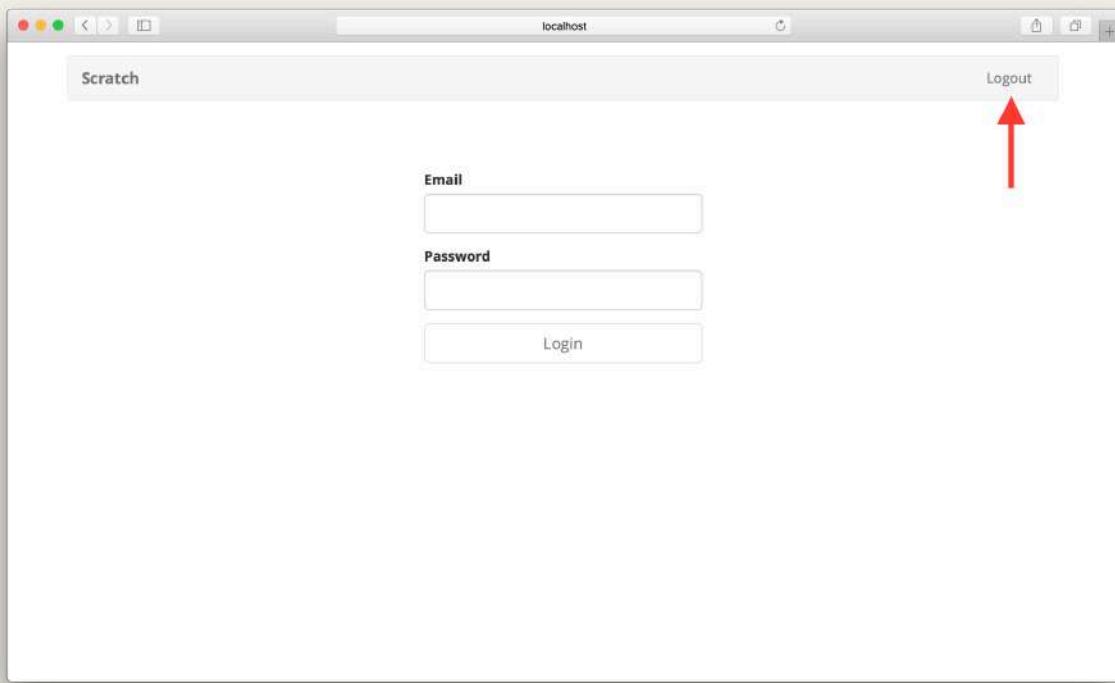
We'll conditionally render our app based on the `isAuthenticating` flag.

◆ 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>
<AppContext.Provider
  value={{ isAuthenticated, userHasAuthenticated }}>
  <Routes />
</AppContext.Provider>
</div>
);
```

Now if you head over to your browser and refresh the page, you should see that a user is logged in.



Login from session loaded screenshot

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.



Help and discussion

View the [comments](#) for this chapter on our forums

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 throughout the entire process. Next, we'll look at redirecting the page after we login and logout to make the flow make more sense.



Help and discussion

View the [comments for this chapter on our forums](#)

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 `useHistory` hook that comes with React Router.

Redirect to Home on Login

◆ CHANGE First, initialize `useHistory` hook in the beginning of `src/containers/Login.js`.

```
const history = useHistory();
```

◆ CHANGE Then 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);
    userHasAuthenticated(true);
    history.push("/");
  } catch (e) {
    alert(e.message);
  }
}
```

◆ CHANGE Also, import `useHistory` from React Router in the header of `src/containers/Login.js`.

```
import { useHistory } from "react-router-dom";
```

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.



React Router v4 redirect home after login screenshot

Redirect to Login After Logout

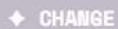
Now we'll do something very similar for the logout process.

◆ CHANGE Add the `useHistory` hook in the beginning of `App` component.

```
const history = useHistory();
```

◆ CHANGE Import `useHistory` by replacing the `import { Link }` line in the header of `src/App.js` with this:

```
import { Link, useHistory } from "react-router-dom";
```



Add the following to the bottom of the handleLogout function in our `src/App.js`.

```
history.push("/login");
```

So our `handleLogout` function should now look like this.

```
async function handleLogout() {
  await Auth.signOut();

  userHasAuthenticated(false);

  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. Also, we are not doing a whole lot with the errors that the `Auth` package might throw. Let's look at those next.



Help and discussion

View the [comments for this chapter on our forums](#)

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);
    userHasAuthenticated(true);
    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 `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 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" className="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>
```

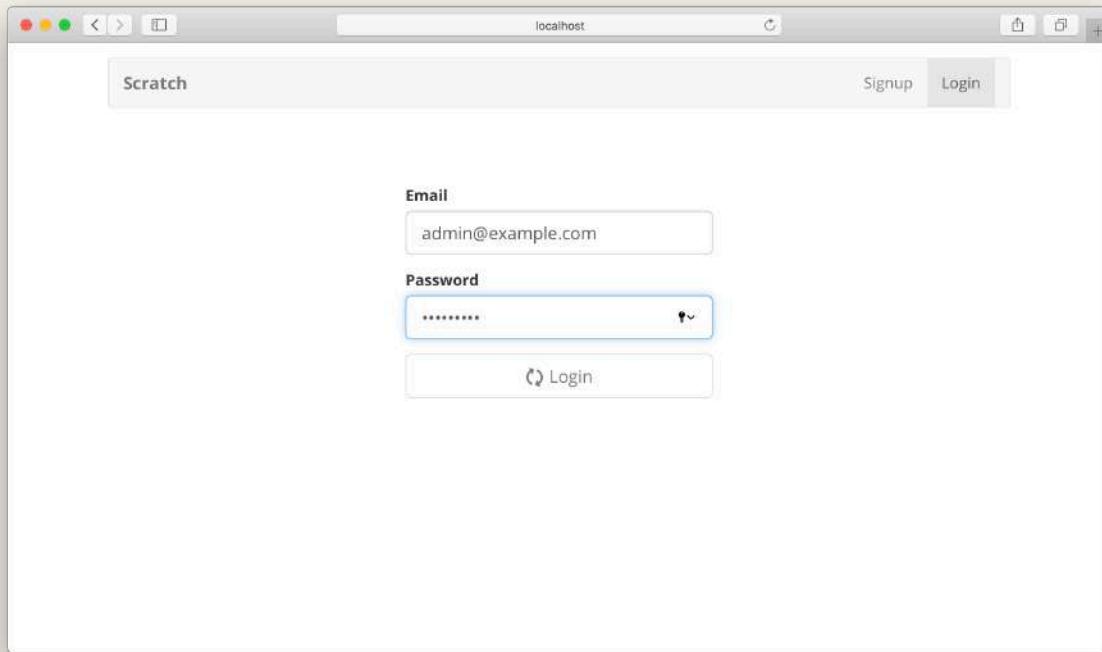
◆ CHANGE And replace it with this.

```
<LoaderButton
  block
  type="submit"
  bsSize="large"
  isLoading={isLoading}
  disabled={!validateForm()}>
  </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.



Login loading state screenshot

Handling Errors

You might have noticed in our Login and App components that we simply `alert` when there is an error. We are going to keep our error handling simple. But it'll help us further down the line if we handle all of our errors in one place.

◆ CHANGE To do that, add the following to `src/libs/errorLib.js`.

```
export function onError(error) {
  let message = error.toString();

  // Auth errors
  if (!(error instanceof Error) && error.message) {
    message = error.message;
  }

  alert(message);
```

```
}
```

The Auth package throws errors in a different format, so all this code does is `alert` the error message we need. And in all other cases simply `alert` the error object itself.

Let's use this in our Login container.

◆ CHANGE Import the new error lib in the header of `src/containers/Login.js`.

```
import { onError } from "../libs/errorLib";
```

◆ CHANGE And replace `alert(e.message)`; in the `handleSubmit` function with:

```
onError(e);
```

We'll do something similar in the App component.

◆ CHANGE Import the error lib in the header of `src/App.js`.

```
import { onError } from "./libs/errorLib";
```

◆ CHANGE And replace `alert(e)`; in the `onLoad` function with:

```
onError(e);
```

We'll improve our error handling a little later on in the guide.

Also, 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](#).

For now, we are ready to move on to the sign up process for our app.



Help and discussion

View the [comments for this chapter on our forums](#)

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

◆ 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. In fact, we did this back when we created our app context. But let's go over in detail 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`. The initial state in our case is an object where the `keys` are the `ids` of the form fields and the `values` are what the user enters.
3. So our hook returns an array with `fields` and a callback function that sets the new state based on the `event` object. The callback function takes the `event` object and gets the form field id from `event.target.id` and the value from `event.target.value`. In the case of our form the elements, the `event.target.id` comes from the `controlId` that's set in the `FormGroup` element:

```

<FormGroup controlId="email" bsSize="large">
  <ControlLabel>Email</ControlLabel>
  <FormControl
    autoFocus
    type="email"
    value={email}
    onChange={e => setEmail(e.target.value)}
  />
</FormGroup>

```

4. The callback function is directly using `setValues`, the function that we get from `useState`. So `onChange` we take what the user has entered and call `setValues` to update the state of `fields, { ...fields, [event.target.id]: event.target.value }`. This updated object is now set as our new form field state.

And that's it! We can now use this in our Login component.

Using Our Custom Hook



Replace our `src/containers/Login.js` with the following:

```
import React, { useState } from "react";
import { Auth } from "aws-amplify";
import { useHistory } from "react-router-dom";
import { FormGroup, FormControl, ControlLabel } from "react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import { useAppContext } from "../libs/contextLib";
import { useFormFields } from "../libs/hooksLib";
import { onError } from "../libs/errorLib";
import "./Login.css";

export default function Login() {
  const history = useHistory();
  const { userHasAuthenticated } = useAppContext();
  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);
      userHasAuthenticated(true);
    } catch (err) {
      onError(err);
    }
  }
}
```

```
    history.push("/");
} catch (e) {
  onError(e);
  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.



Help and discussion

View the [comments for this chapter](#) on our forums

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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 { useHistory } from "react-router-dom";
import {
  HelpBlock,
  FormGroup,
  FormControl,
  ControlLabel
} from "react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import { useContext } from "../libs/contextLib";
import { useFormFields } from "../libs/hooksLib";
import { onError } from "../libs/errorLib";
import "./Signup.css";

export default function Signup() {
  const [fields, handleFieldChange] = useFormFields({
    email: "",
    password: "",
    confirmPassword: "",
    confirmationCode: ""
  });
  const history = useHistory();
  const [newUser, setNewUser] = useState(null);
```

```
const { userHasAuthenticated } = useAppContext();
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}
        </FormControl>
      </FormGroup>
    </form>
  );
}
```

```
        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 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

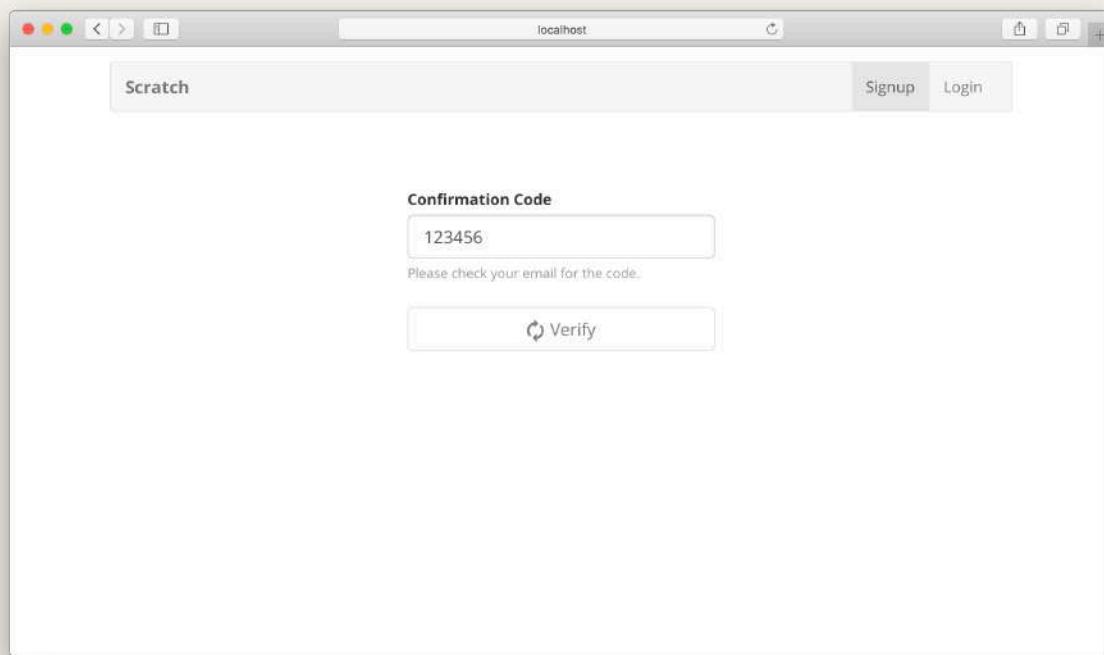
◆ CHANGE Finally, add our container as a route in `src/Routes.js` below our login route.

```
<Route exact path="/signup">  
  <Signup />  
</Route>
```

◆ 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.



Signup page added screenshot

Next, let's connect our signup form to Amazon Cognito.



Help and discussion

View the [comments for this chapter](#) on our forums

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.signIn({
      username: fields.email,
      password: fields.password,
    });
    setIsLoading(false);
    setNewUser(newUser);
  } catch (e) {
    onError(e);
    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);

userHasAuthenticated(true);
history.push("/");
} catch (e) {
onError(e);
setIsLoading(false);
}
}
```

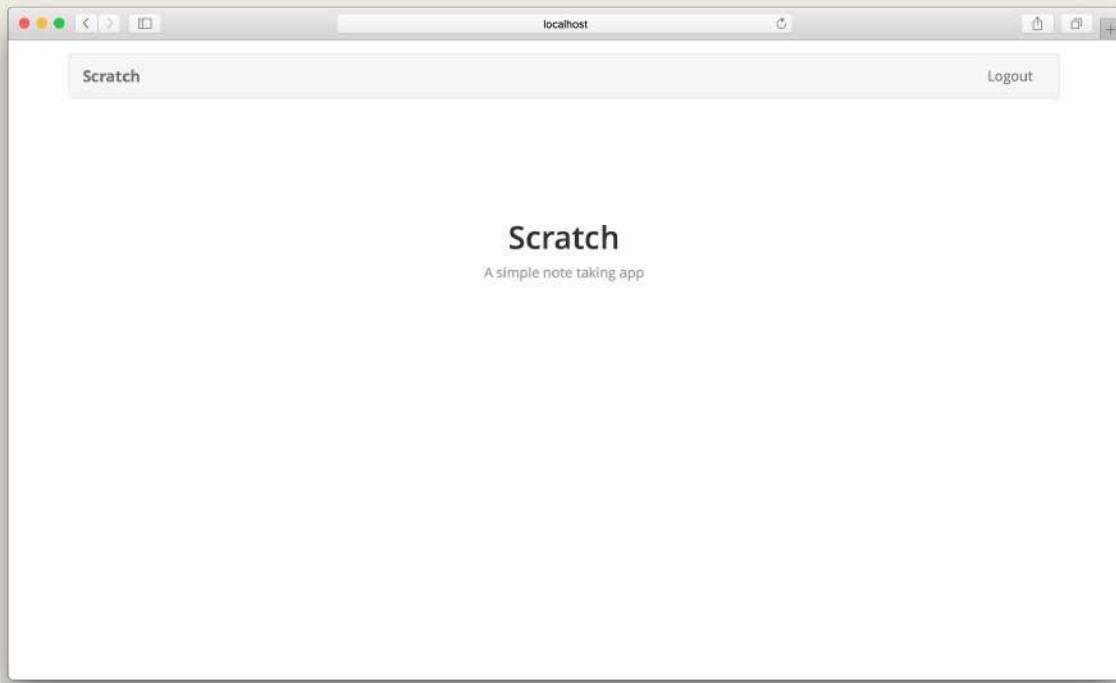
◆ CHANGE Also, include the Amplify Auth in our header.

```
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 context 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.



Redirect home after signup screenshot

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](#).
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](#).

Next up, we are going to create our first note.



Help and discussion

View the [comments for this chapter on our forums](#)

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



Create a new file `src/containers/NewNote.js` and add the following.

```
import React, { useRef, useState } from "react";
import { useHistory } from "react-router-dom";
import { FormGroup, FormControl, ControlLabel } from "react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import { onError } from "../libs/errorLib";
import config from "../config";
import "./NewNote.css";

export default function NewNote() {
  const file = useRef(null);
  const history = useHistory();
  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](#), 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



Finally, add our container as a route in `src/Routes.js` below our signup route.

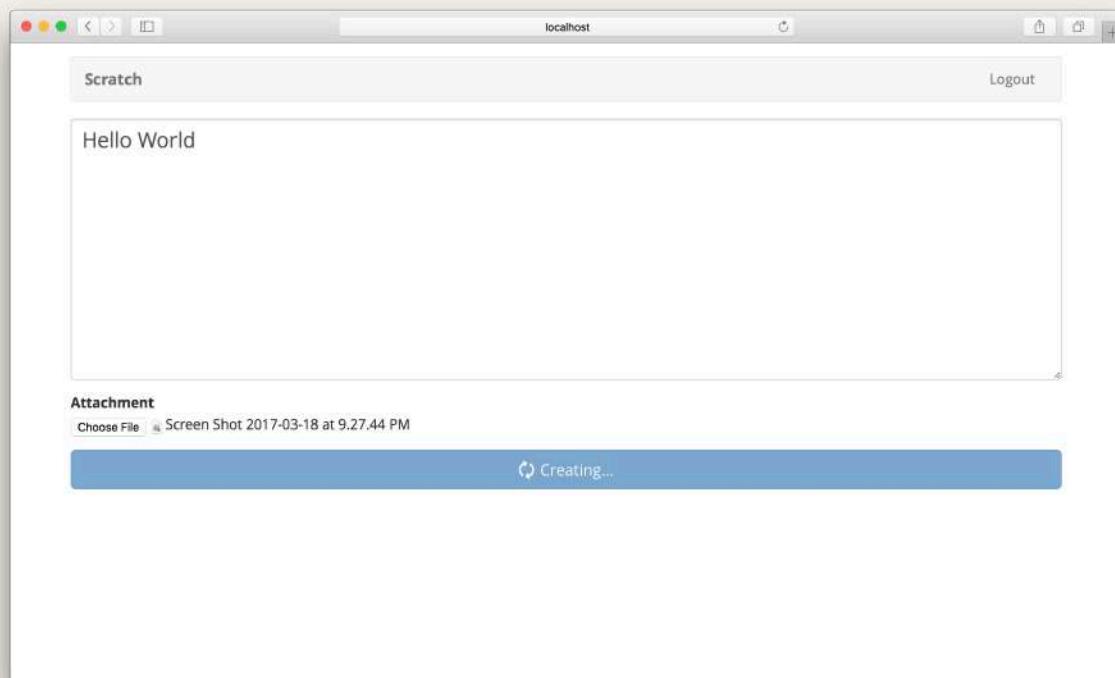
```
<Route exact path="/notes/new">
  <NewNote />
</Route>
```



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.



New note page added screenshot

Next, let's get into connecting this form to our API.



Help and discussion

[View the comments for this chapter on our forums](#)

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.

◆ 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 });
  history.push("/");
} catch (e) {
  onError(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](#) 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.



New note created screenshot

Next let's upload our file to S3 and add an attachment to our note.



Help and discussion

View the [comments for this chapter](#) on our forums

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](#) 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 Create `src/libs/awsLib.js` and add the following:

```
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 });
  history.push("/");
} catch (e) {
  onError(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.



Help and discussion

View the [comments for this chapter](#) on our forums

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, ListGroupItem } from "react-bootstrap";
import { useAppContext } from "../libs/contextLib";
import { onError } from "../libs/errorLib";
import "./Home.css";

export default function Home() {
  const [notes, setNotes] = useState([]);
  const { isAuthenticated } = useAppContext();
  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>
    );
  }
}
```

```
}

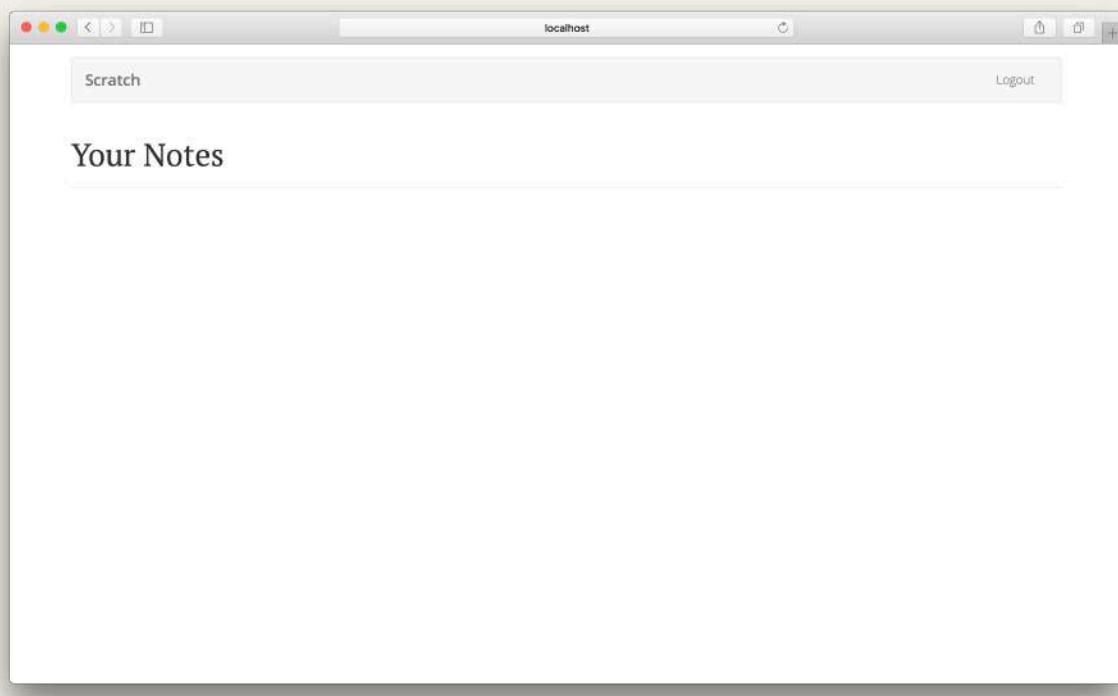
function renderNotes() {
  return (
    <div className="notes">
      <PageHeader>Your Notes</PageHeader>
      <ListGroup>
        {!isLoading && renderNotesList(notes)}
      </ListGroup>
    </div>
  );
}

return (
  <div className="Home">
    {isAuthenticated ? renderNotes() : renderLander()}
  </div>
);
}
```

We are doing a few things of note here:

1. Rendering the lander or the list of notes based on `isAuthenticated` flag in our app context.
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.



Empty homepage loaded screenshot

Next we are going to fill it up with our API.



Help and discussion

View the [comments for this chapter on our forums](#)

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 (!isAuthenticated) {
      return;
    }

    try {
      const notes = await loadNotes();
      setNotes(notes);
    } catch (e) {
      onError(e);
    }

    setIsLoading(false);
  }

  onLoad();
}, [isAuthenticated]);

function loadNotes() {
  return API.get("notes", "/notes");
}
```

We are using the `useEffect` React Hook. We covered how this works back in the [Load the State from the Session](#) 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 `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 `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>
    )
  );
}
```



Include the LinkContainer from react-router-bootstrap.

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

The code above does a few things.

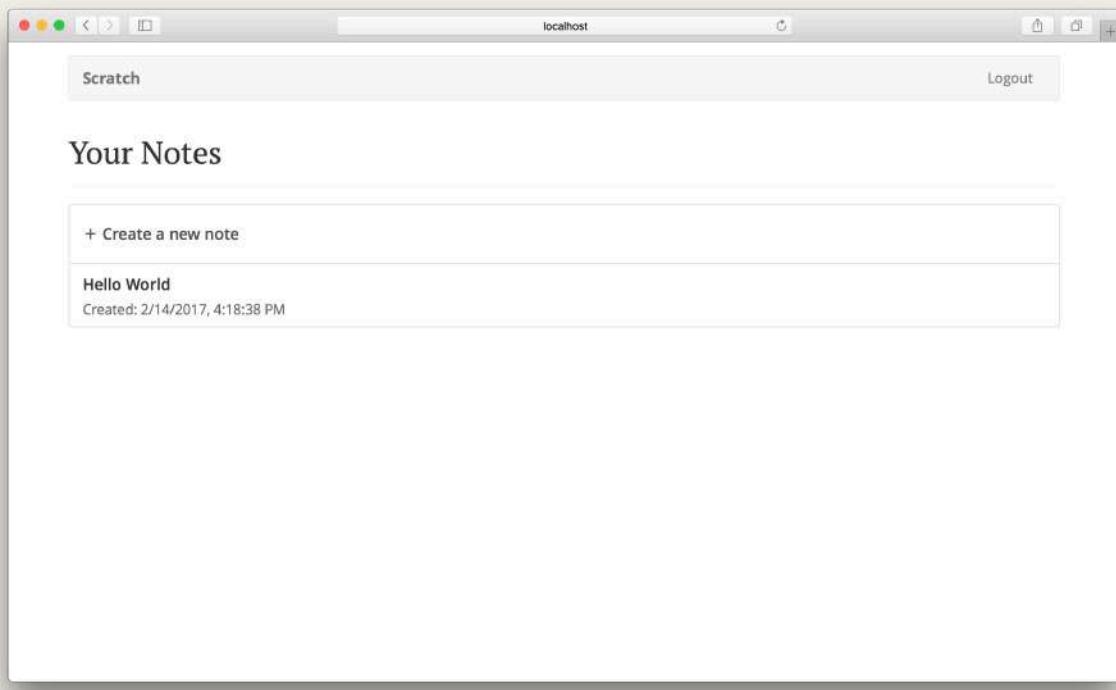
1. It always renders a **Create a new note** button as the first item in the list (even if the list is empty). We do this by concatenating an array with an empty object with our notes array.
2. We render the first line of each note as the `ListGroupItem` header by doing `note.content.trim().split('\n')[0]`.
3. And the `LinkContainer` component directs our app to each of the items.



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.



Homepage list loaded screenshot

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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.

```
<Route exact path="/notes/:id">
  <Notes />
</Route>
```

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



Create a new file `src/containers/Notes.js` and add the following.

```
import React, { useRef, useState, useEffect } from "react";
import { useParams, useHistory } from "react-router-dom";
import { API, Storage } from "aws-amplify";
import { onError } from "../libs/errorLib";

export default function Notes() {
  const file = useRef(null);
  const { id } = useParams();
  const history = useHistory();
  const [note, setNote] = useState(null);
  const [content, setContent] = useState("");

  useEffect(() => {
    function loadNote() {
      return API.get("notes", `/notes/${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) {
        onError(e);
      }
    }

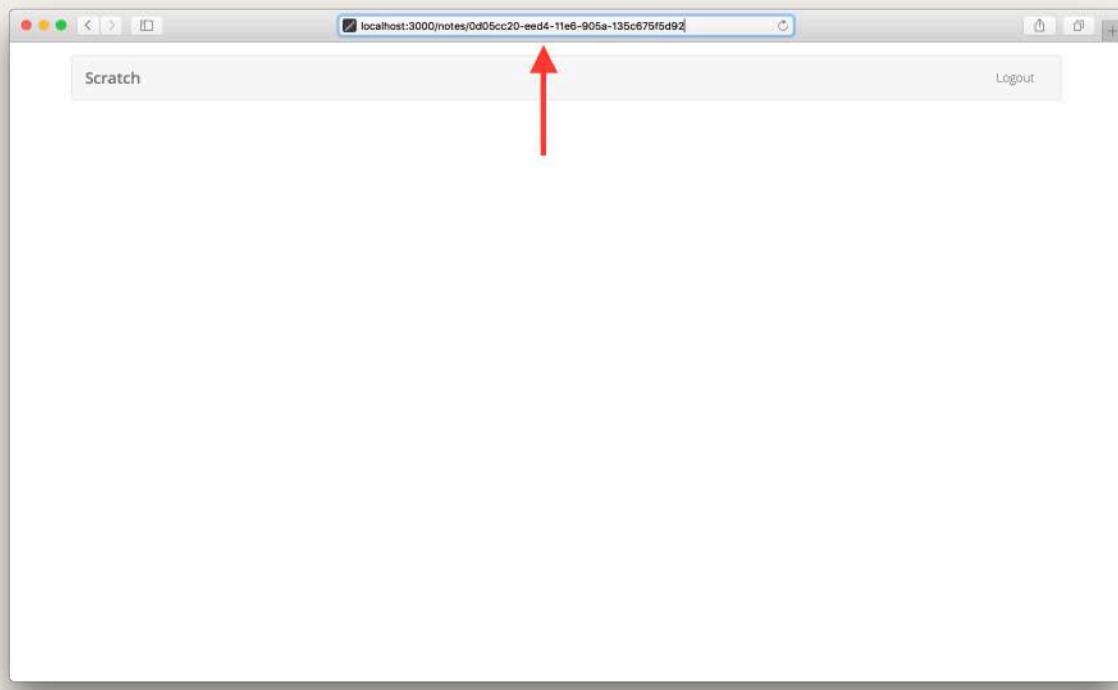
    onLoad();
  }
}
```

```
}, [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 `useParams` hook that comes with React Router. The `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.



Empty notes page loaded screenshot

Next up, we are going to render the note we just loaded.



Help and discussion

View the [comments for this chapter on our forums](#)

Render the Note Form

Now that our container loads a note using the `useEffect` method, 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 { id } = useParams();
const history = useHistory();
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`.

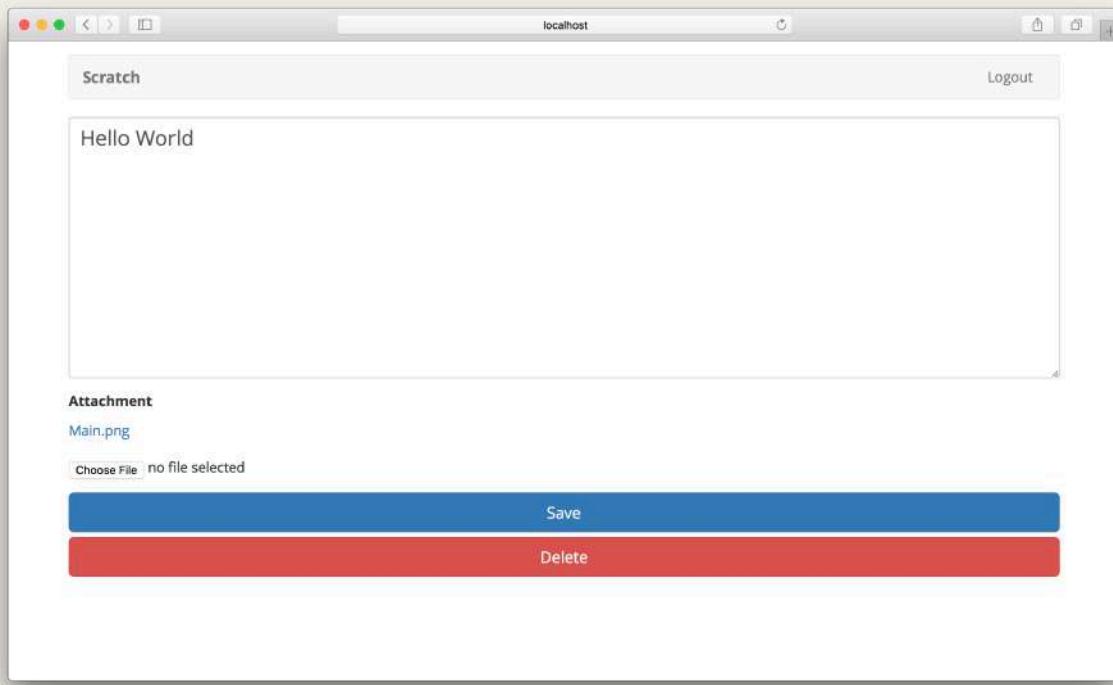
```
.Notes form {
  padding-bottom: 15px;
}

.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.



Notes page loaded screenshot

Next, we'll look at saving the changes we make to our note.



Help and discussion

View the [comments for this chapter](#) on our forums

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/${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  
});  
history.push("/");  
} catch (e) {  
  onError(e);  
  setIsLoading(false);  
}  
}  
}
```



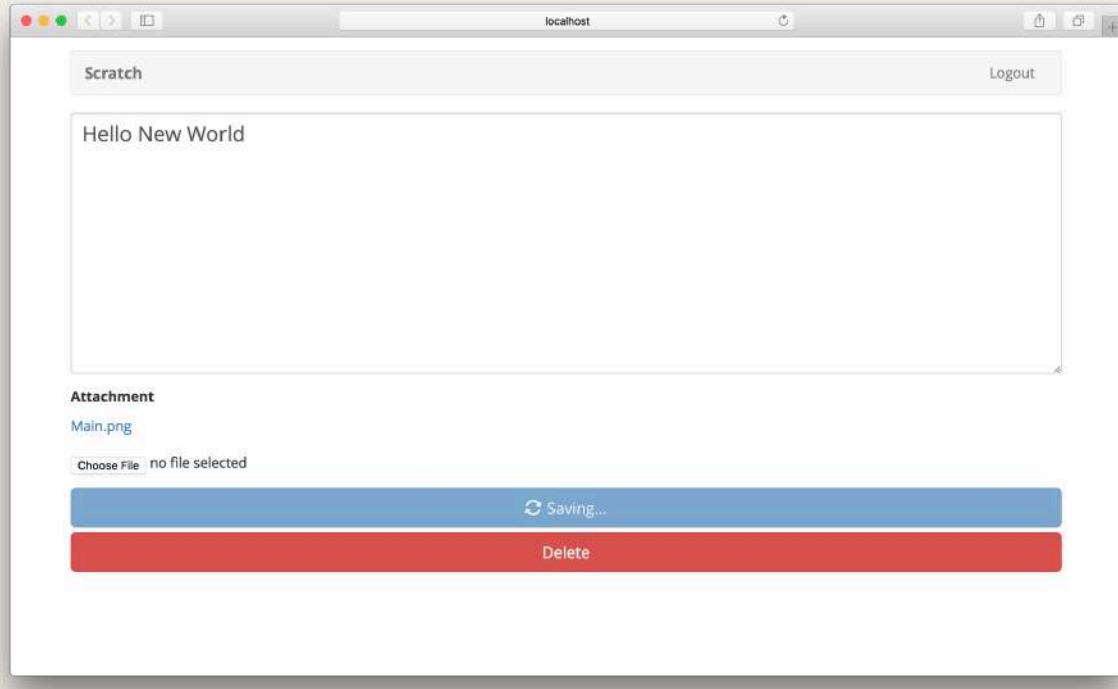
And include our s3Upload helper method in the header:

```
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 the useParams hook. 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.



Notes page saving screenshot

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](#) on how to a delete file from S3.

Next up, let's allow users to delete their note.



Help and discussion

View the [comments](#) for this chapter on our forums

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/${id}`);
}

async function handleDelete(event) {
  event.preventDefault();

  const confirmed = window.confirm(
    "Are you sure you want to delete this note?"
  );

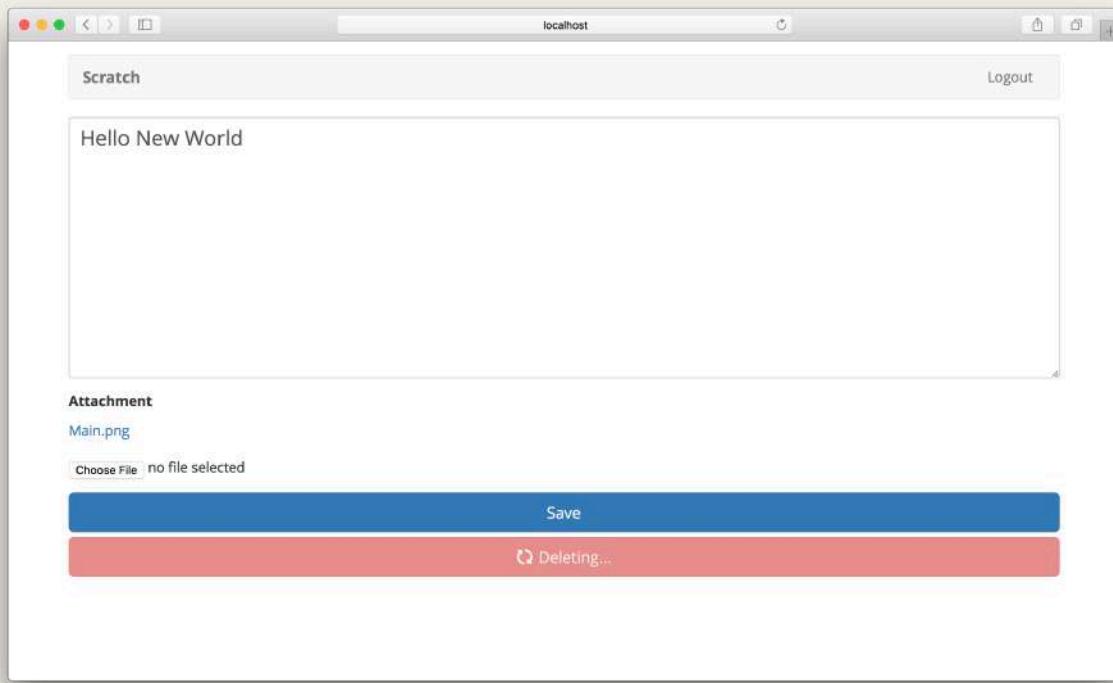
  if (!confirmed) {
    return;
  }

  setIsDeleting(true);

  try {
    await deleteNote();
    history.push("/");
  } catch (e) {
    onError(e);
    setIsDeleting(false);
  }
}
```

We are simply making a `DELETE` request to `/notes/:id` where we get the `id` from `useParams` hook provided by React Router. 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.



Note page deleting screenshot

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](#) 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!



Help and discussion

View the [comments for this chapter on our forums](#)

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, useEffect } from "react";
import { useHistory } from "react-router-dom";
import { API } from "aws-amplify";
import { onError } from "../libs/errorLib";
import config from "../config";

export default function Settings() {
  const history = useHistory();
  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`.

```
<Route exact path="/settings">  
  <Settings />  
</Route>
```

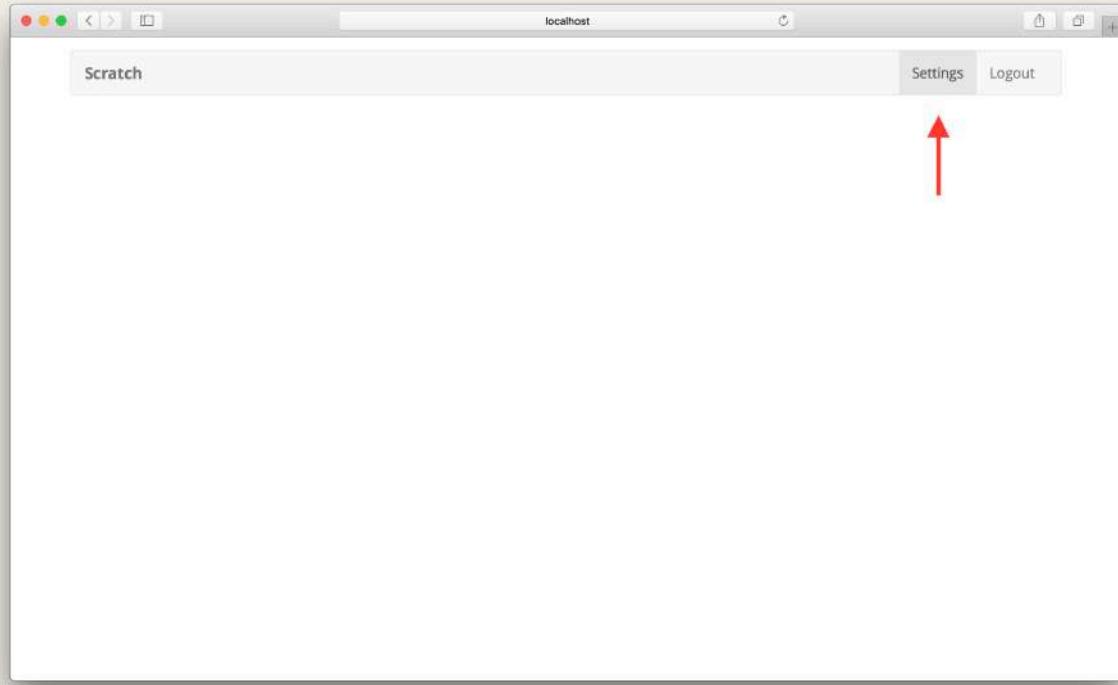
◆ CHANGE Next add a link to our settings page in the navbar by replacing the `return` statement in `src/App.js` with this.

```
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>  
              </>  
            ) : null}  
          </Nav>  
        </Navbar.Collapse>  
      </Navbar>  
    </div>  
  )  
)
```

```
    ) : (
      <>
        <LinkContainer to="/signup">
          <NavItem>Signup</NavItem>
        </LinkContainer>
        <LinkContainer to="/login">
          <NavItem>Login</NavItem>
        </LinkContainer>
      </>
    )}
  </Nav>
</Navbar.Collapse>
</Navbar>
<AppContext.Provider
  value={{ isAuthenticated, userHasAuthenticated }}>
  <Routes />
</AppContext.Provider>
</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.



Add empty settings page screenshot

Next, we'll add our Stripe SDK keys to our config.



Help and discussion

View the [comments for this chapter](#) on our forums

Add Stripe Keys to Config

Back in the [Setup a Stripe account](#) 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](#) chapter.

Let's also include 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>
```

And load the Stripe config in our settings page.

◆ CHANGE Add the following at top of the `Settings` function in `src/containers/Settings.js`.

```
const [stripe, setStripe] = useState(null);

useEffect(() => {
  setStripe(window.Stripe(config.STRIPE_KEY));
}, []);
```

This loads the Stripe object from Stripe.js with the Stripe key when our settings page loads. And saves it to the state.

Next, we'll build our billing form.



Help and discussion

View the [comments](#) for this chapter on our forums

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's name</ControlLabel>
      <FormControl
        type="text"
        value={fields.name}
        onChange={handleFieldChange}
      />
    </FormGroup>
  </form>
)
```

```
        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](#).

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.



Help and discussion

View the [comments for this chapter on our forums](#)

Connect the Billing Form

Now all we have left to do is to connect our billing form to our billing API.

◆ CHANGE Replace our return statement in `src/containers/Settings.js` with this.

```
async function handleFormSubmit(storage, { token, error }) {
  if (error) {
    onError(error);
    return;
  }

  setIsLoading(true);

  try {
    await billUser({
      storage,
      source: token.id
    });

    alert("Your card has been charged successfully!");
    history.push("/");
  } catch (e) {
    onError(e);
    setIsLoading(false);
  }
}

return (
  <div className="Settings">
    <StripeProvider stripe={stripe}>
      <Elements>
        <BillingForm isLoading={isLoading} onSubmit={handleFormSubmit} />
    </Elements>
  </StripeProvider>
</div>
)
```

```
</Elements>
</StripeProvider>
</div>
);
```

◆ CHANGE And add the following to the header.

```
import { Elements, StripeProvider } from "react-stripe-elements";
import BillingForm from "../components/BillingForm";
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 our Stripe key. In the [Add Stripe Keys to Config](#), we load the Stripe object with our key into the state. We use it 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.

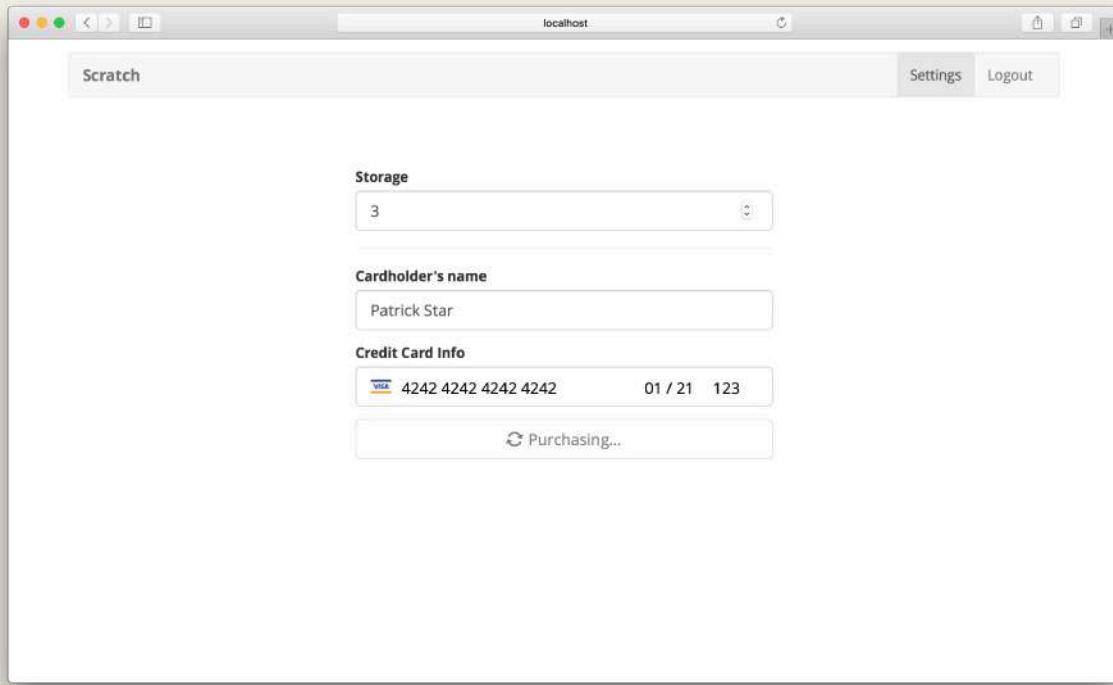
Finally, let's handle some styles for our settings page as a whole.

◆ 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.



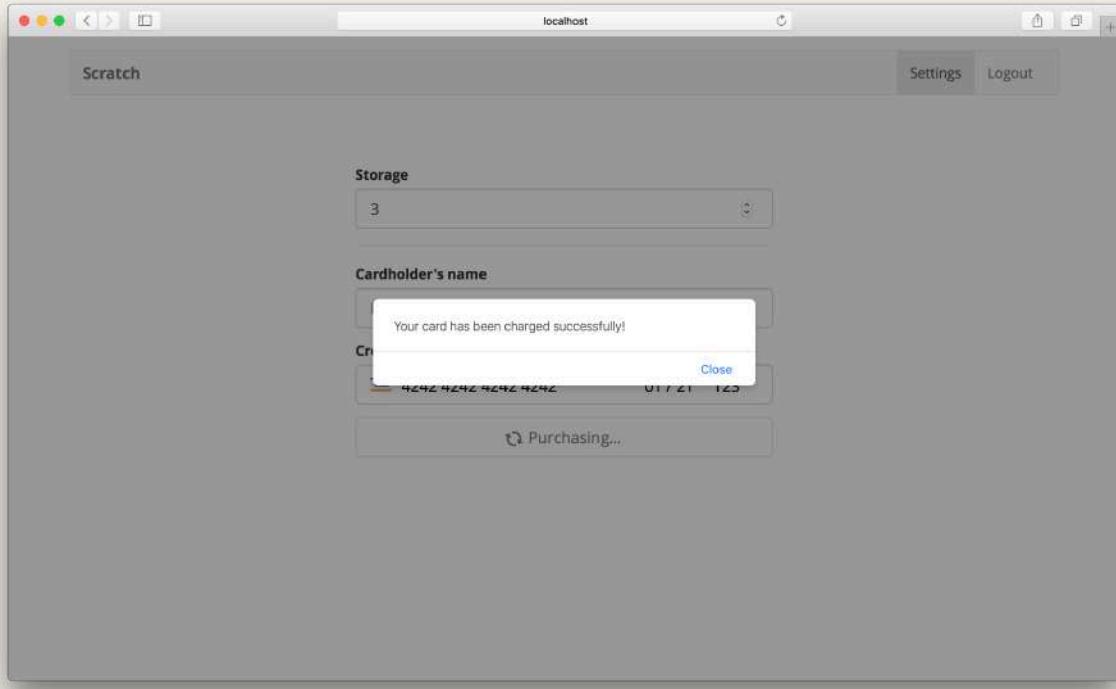
Settings screen with billing form screenshot

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.

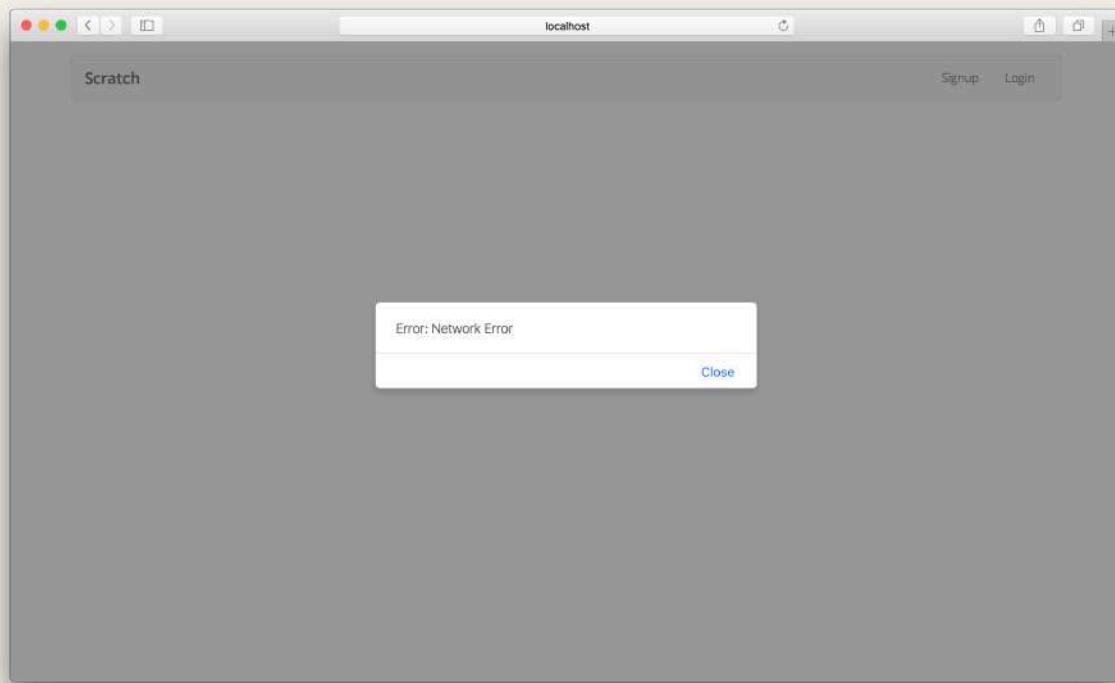
You can read more about the Stripe test cards in the [Stripe API Docs here](#).

If everything is set correctly, you should see the success message and you'll be redirected to the homepage.



Settings screen billing success screenshot

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.



Note page logged out error screenshot

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.



Help and discussion

View the [comments](#) for this chapter on our forums

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 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.



Help and discussion

View the [comments for this chapter on our forums](#)

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, useLocation } from "react-router-dom";
import { useAppContext } from "../libs/contextLib";

export default function AuthenticatedRoute({ children, ...rest }) {
  const { pathname, search } = useLocation();
  const { isAuthenticated } = useAppContext();
  return (
    <Route {...rest}>
      {isAuthenticated ? (
        children
      ) : (
        <Redirect to={`/login?redirect=${pathname}${search}`}>
        </Redirect>
      )}
    </Route>
  );
}
```

This simple component creates a Route where its children are rendered only if the user is authenticated. If the user is not authenticated, then it redirects to the login page. Let's take a closer look at it:

- Like all components in React, `AuthenticatedRoute` has a prop called `children` that represents all child components. Example child components in our case would be `NewNote`, `Notes` and `Settings`.

- The `AuthenticatedRoute` component returns a React Router `Route` component.
- We use the `useApplicationContext` hook to check if the user is authenticated.
- If the user is authenticated, then we simply render the `children` component. And if the user is not authenticated, then we use the `Redirect` React Router component to redirect the user to the login page.
- We also pass in the current path to the login page (`redirect` in the query string). We will use this later to redirect us back after the user logs in. We use the `useLocation` React Router hook to get this info.

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";
import { useApplicationContext } from "../libs/contextLib";

export default function UnauthenticatedRoute({ children, ...rest }) {
  const { isAuthenticated } = useApplicationContext();
  return (
    <Route {...rest}>
      {!isAuthenticated ? (
        children
      ) : (
        <Redirect to="/" />
      )}
    </Route>
  );
}
```

Here we are checking to ensure that the user is **not** authenticated before we render the child components. Example child components here would be `Login` and `Signup`. 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.

**Help and discussion**

View the [comments](#) for this chapter on our forums

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.

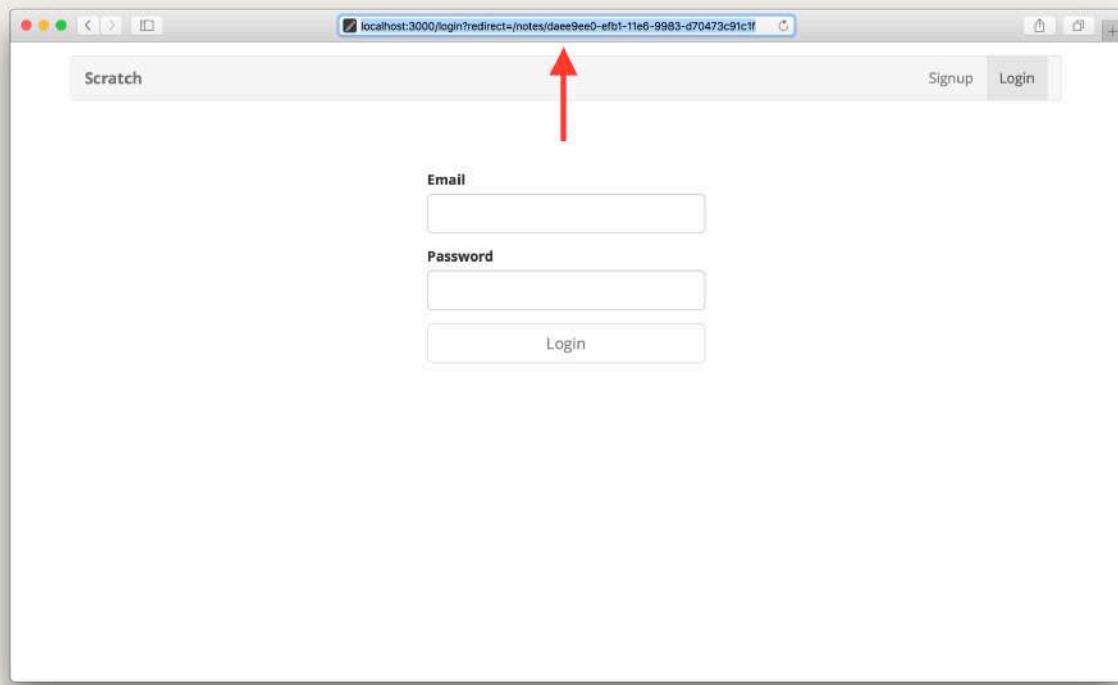
So the following routes in `src/Routes.js` would be affected.

```
<Route exact path="/login">
  <Login />
</Route>
<Route exact path="/signup">
  <Signup />
</Route>
<Route exact path="/settings">
  <Settings />
</Route>
<Route exact path="/notes/new">
  <NewNote />
</Route>
<Route exact path="/notes/:id">
  <Notes />
</Route>
```

◆ CHANGE They should now look like so:

```
<UnauthenticatedRoute exact path="/login">
  <Login />
</UnauthenticatedRoute>
<UnauthenticatedRoute exact path="/signup">
  <Signup />
</UnauthenticatedRoute>
<AuthenticatedRoute exact path="/settings">
  <Settings />
</AuthenticatedRoute>
<AuthenticatedRoute exact path="/notes/new">
  <NewNote />
</AuthenticatedRoute>
<AuthenticatedRoute exact path="/notes/:id">
  <Notes />
</AuthenticatedRoute>
```

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.



Note page redirected to login screenshot

Next, we are going to use the reference to redirect to the note page after we login.



Help and discussion

View the [comments for this chapter on our forums](#)

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({ children, ...rest }) {
  const { isAuthenticated } = useAppContext();
  const redirect = querystring("redirect");
  return (
    <Route {...rest}>
      {!isAuthenticated ? (
        children
      ) : (
        <Redirect to={redirect === "" || redirect === null ? "/" : redirect} />
      )}
    </Route>
  );
}
```

◆ CHANGE And remove the following from the handleSubmit method in src/containers/Login.js.

```
history.push("/");
```

◆ CHANGE Also, remove the hook declaration.

```
const history = useHistory();
```

◆ CHANGE Finally, remove the import.

```
import { useHistory } from "react-router-dom";
```

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](#) chapter. It builds on what we have covered so far.



Help and discussion

View the [comments](#) for this chapter on our forums



For reference, here is the code we are using

Frontend Source: [redirect-on-login](#)

Deploying the backend to production

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 every time you create a new environment or app. Thankfully, 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](#) (for the backend) and [Netlify](#) (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 the 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.

Reorganize Your Repo

In the next few chapters we are going to be using [AWS CDK](#) to configure our Serverless infrastructure. So let's reorganize our backend repo around a bit.

◆ **CHANGE** Create a new services/notes/ directory. Run the following in the root of our backend repo.

```
$ mkdir -p services/notes
```

This is a common organizational pattern in Serverless Framework projects. You'll have multiple services in the future. So we'll create a services directory and add a notes service in it.

◆ **CHANGE** Let's move our files to the new directory.

```
$ mv *.js *.json *.yml .env services/notes  
$ mv tests libs mocks node_modules services/notes
```

If you are on Windows or if the above commands don't work, make sure to copy over these files and directories to services/notes.

In the coming chapters, we'll also be creating an infrastructure/ directory for our CDK app.

Update the serverless.yml

We'll also be deploying our app to multiple environments. This makes it so that when we make changes or test our app while developing, we don't affect our users. So let's start by defaulting our API to deploy to the development environment, instead of production.

◆ **CHANGE** Open the services/notes/serverless.yml and find the following line:

```
stage: prod
```

◆ **CHANGE** And replace it with:

```
stage: dev
```

We are defaulting the stage to dev instead of prod. This will become more clear later when we create multiple environments.

Commit the Changes

Let's quickly commit these to Git.

```
$ git add .  
$ git commit -m "Reorganizing the repo"
```

Note that, we are going to be creating new versions of our resources (DynamoDB, Cognito, etc.). Instead of using the ones that we created in the previous sections. This is because we want to define and create them programmatically. You can remove the resources we previously created. But for the purpose of this guide, we are going to leave it as is. In case you want to refer back to it at some point.

Let's get started by getting a quick feel for how *infrastructure as code* works.



Help and discussion

View the [comments for this chapter on our forums](#)

What Is Infrastructure as Code

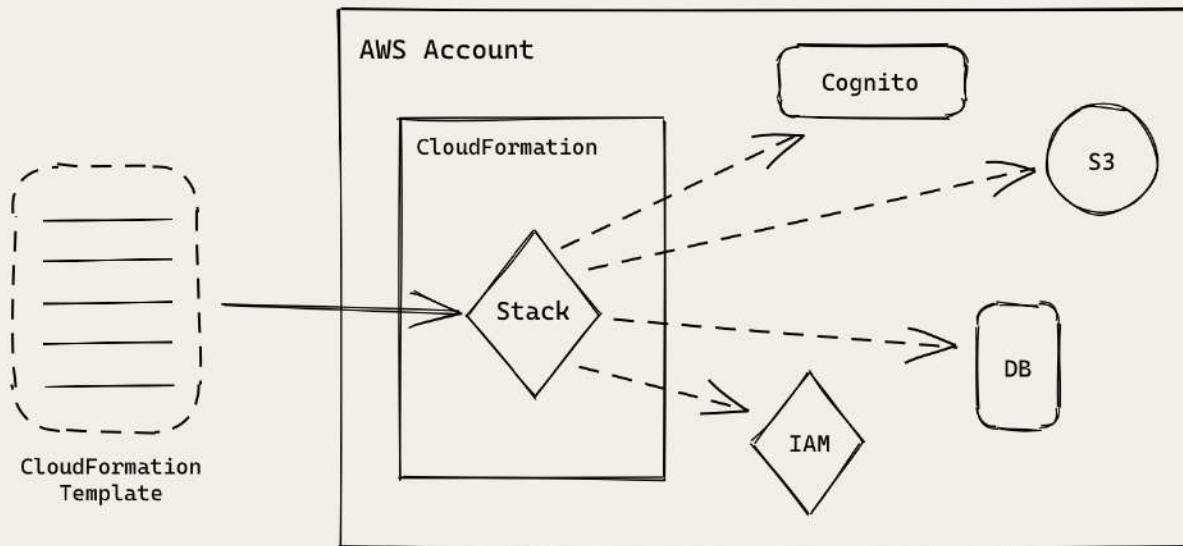
[Serverless Framework](#) converts your `serverless.yml` into a [CloudFormation](#) template. This is a description of the infrastructure that you are trying to configure as a part of your serverless project. In our case we were describing the Lambda functions and API Gateway endpoints that we were trying 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 configured 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.

AWS CloudFormation

To do this we are going to be using [AWS CloudFormation](#). CloudFormation is an AWS service that takes a template (written in JSON or YAML), and provisions your resources based on that.



How CloudFormation works

It creates a CloudFormation **stack** from the submitted **template**, and that stack is directly tied to the resources that have been created. So if you remove the stack, the services that it created will be removed as well.

As an example, here is what the CloudFormation template for a DynamoDB table (like the one [we created manually before](#)) looks like.

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

Serverless Framework internally uses CloudFormation as well. It converts your `serverless.yml` into a CloudFormation template and submits it to AWS when you run `serverless deploy`. And when you run `serverless remove`, it'll ask CloudFormation to remove the stack that it previously created.

Problems with CloudFormation

CloudFormation is great for defining your AWS resources. However it has a few major drawbacks.

In a CloudFormation template you need to define all the resources that your app needs. This includes quite a large number of minor resources that you won't be directly interacting with. So your templates can easily be a few hundred lines long.

YAML and JSON are really easy to get started with. But it can be really hard to maintain large CloudFormation templates. And since these are just simple definition files, it makes it hard to reuse and compose them.

Finally, the learning curve for CloudFormation templates can be really steep. You'll find yourself constantly looking at the documentation to figure out how to define your resources. ## Introducing AWS CDK

To fix these issues, AWS launched the [AWS CDK project back in August 2018](#). It allows you to use modern programming languages like JavaScript or Python, instead of YAML or JSON. We'll be using CDK in the coming chapters. So let's take a quick look at how it works.



Help and discussion

View the [comments for this chapter on our forums](#)

What is AWS CDK?

AWS CDK (Cloud Development Kit), released in Developer Preview back in August 2018; allows you to use TypeScript, JavaScript, Java, .NET, and Python to create AWS infrastructure.

So for example, a CloudFormation template that creates our DynamoDB table would now look like.

```
- 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  
  
+ const table = new dynamodb.Table(this, "notes", {  
+   partitionKey: { name: 'userId', type: dynamodb.AttributeType.STRING },  
+   sortKey: { name: 'noteId', type: dynamodb.AttributeType.STRING },  
+   billingMode: dynamodb.BillingMode.PAY_PER_REQUEST,  
+ });
```

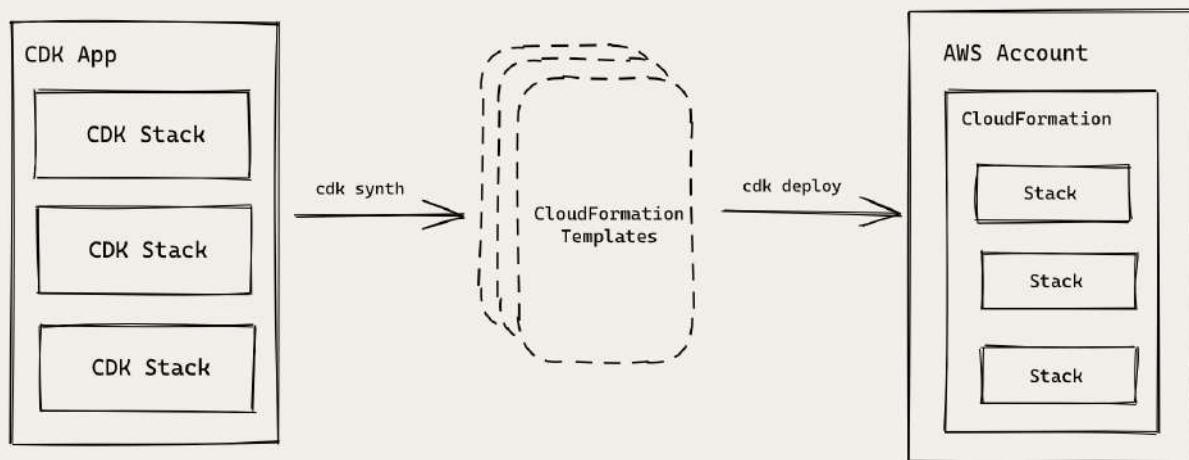
The first thing to notice is that the resources are defined as class instances in JavaScript. That's

great because we are used to thinking in terms of objects in programming languages. And now we can do the same for our infrastructure. Second, we can reuse these objects. We can combine and compose them. So if you always find yourself creating the same set of resources, you can make that into a new class and reuse it!

CDK is truly, *infrastructure as code*.

How CDK works

CDK internally uses CloudFormation. It converts your code into a CloudFormation template. So in the above example, you write the code at the bottom and it generates the CloudFormation template at the top.



How CDK works

A CDK app is made up of multiple stacks. Or more specifically, multiple instances of the `cdk.Stack` class. While these do get converted into CloudFormation stacks down the road. It's more appropriate to think of them as representations of your CloudFormation stacks, but in code.

When you run `cdk synth`, it converts these stacks into CloudFormation templates. And when you run `cdk deploy`, it'll submit these to CloudFormation. CloudFormation creates these stacks and all the resources that are defined in them.

It's fairly straightforward. The key bit here is that even though we are using CloudFormation internally, we are not working directly with the YAML or JSON templates anymore.

Next we'll look at how we can use CDK alongside our Serverless Framework services.



Help and discussion

View the [comments](#) for this chapter on our forums

Using AWS CDK with Serverless Framework

To quickly recap, we are using [Serverless Framework](#) to deploy our Serverless backend API. And we are going to use [AWS CDK](#) to deploy the rest of the infrastructure for our notes app.

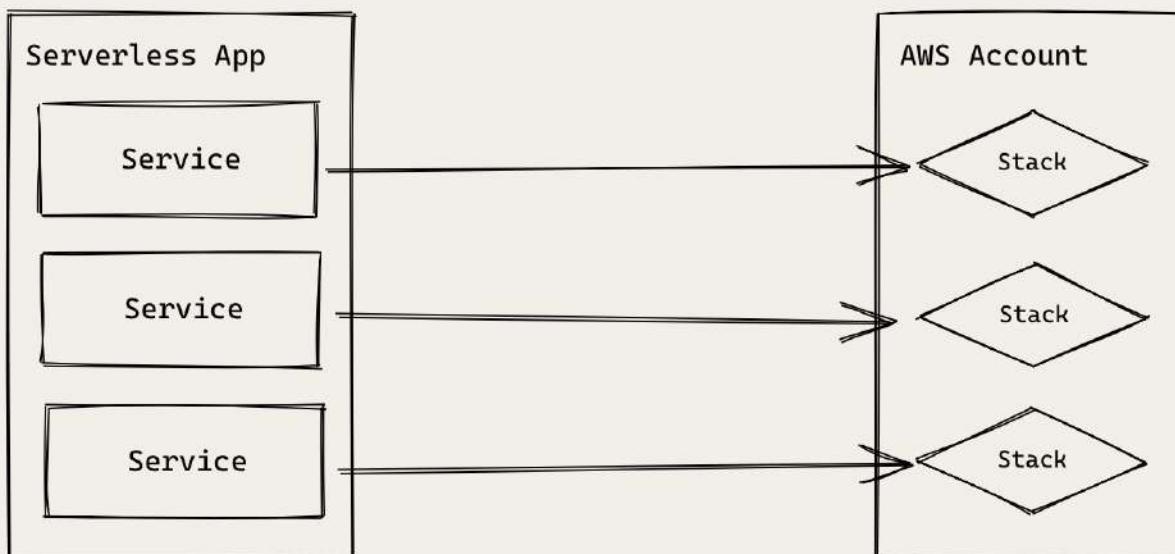
In this chapter we'll look at how we can use the two together.

Background

To understand how we can use Serverless Framework and CDK together, let's look at how their apps are structured.

Serverless Framework App Architecture

So far in this guide we've only created a single Serverless service. But Serverless apps can be made up of multiple services and the app as a whole is deployed to the same environment.



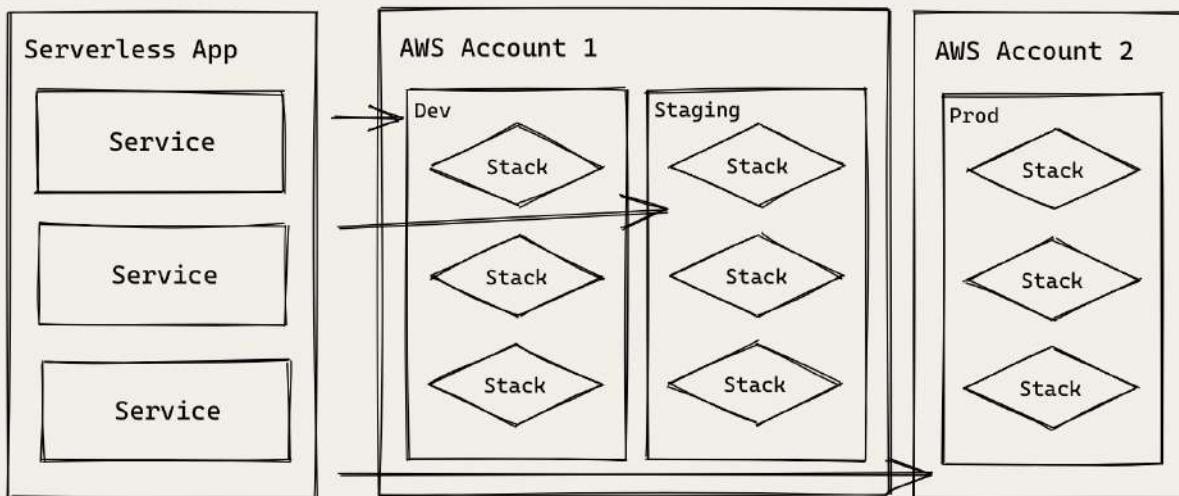
Serverless Framework App Architecture

You might recall that Serverless Framework internally uses CloudFormation. So each service is deployed as a CloudFormation stack to the target AWS account. You can specify a stage, region, and AWS profile to customize this.

```
$ AWS_PROFILE=development serverless deploy --stage dev --region us-east-1
```

The `--stage` option here prefixes your stack names with the stage name. So if you are deploying multiple stages to the same AWS account, the resource names will not thrash.

This allows you to easily deploy your Serverless app to multiple environments. Even if they are in the same AWS account.



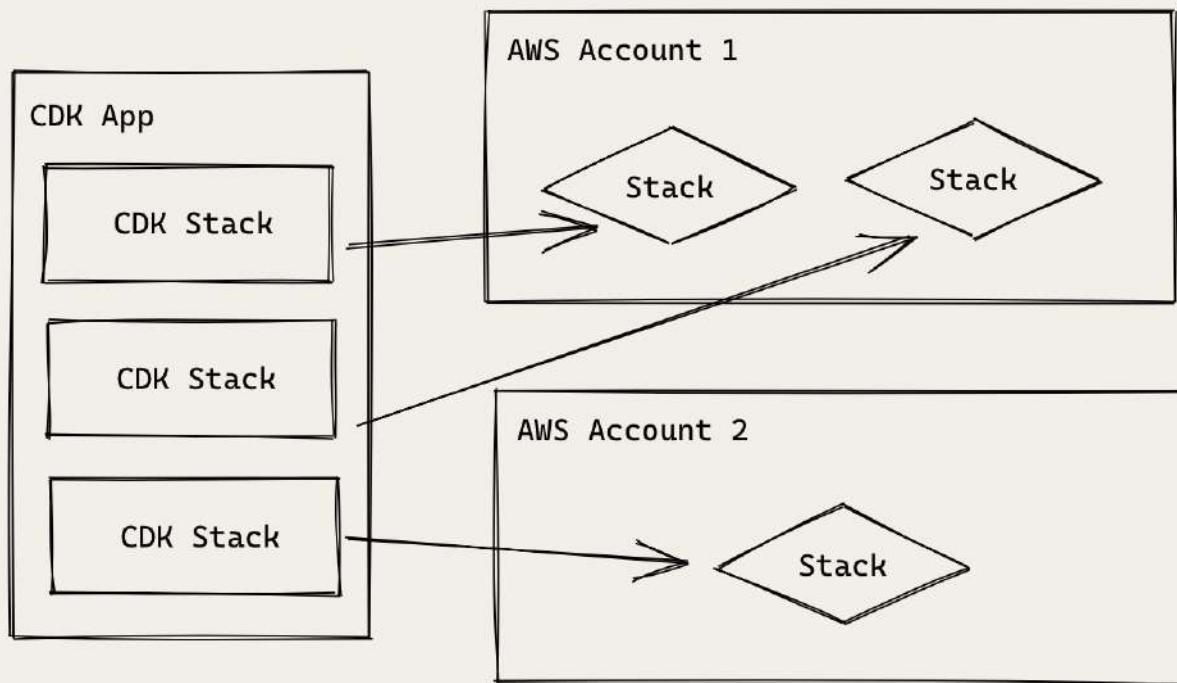
Serverless Framework app deployed to multiple stages

In the example above, the same app is deployed **three times to three different stages**. And two of the stages are in the same AWS account. While the third is in its own account.

We are able to do this by simply changing the options in the `serverless deploy` command. This allows us to deploy to multiple environments/stages without making any changes to our code.

CDK App Architecture

AWS CDK apps on the other hand are made up of multiple stacks. And each stack is deployed to the target AWS account as a CloudFormation stack. However, unlike Serverless apps, each stack can be deployed to a different AWS account or region.



AWS CDK App Architecture

We haven't had a chance to look at some CDK code in detail yet, but you can define the AWS account and region that you want your CDK stack to be deployed to.

```
new MyStack(app, "my-stack", { env: { account: "1234", region: "us-east-1" }  
});
```

This means that each time you deploy your CDK app, it could potentially create a stack in multiple environments. This critical design difference prevents us from directly using CDK apps alongside our Serverless services.

You can fix this issue by following a certain convention in your CDK app. However, this is only effective if these conventions are enforced.

Ideally, we'd like our CDK app to work the same way as our Serverless Framework app. So we can deploy them together. This will matter a lot more when we are going to `git push` to deploy our apps automatically.

To fix this issue, we created the [Serverless Stack Toolkit \(SST\)](#).

Enter, Serverless Stack Toolkit

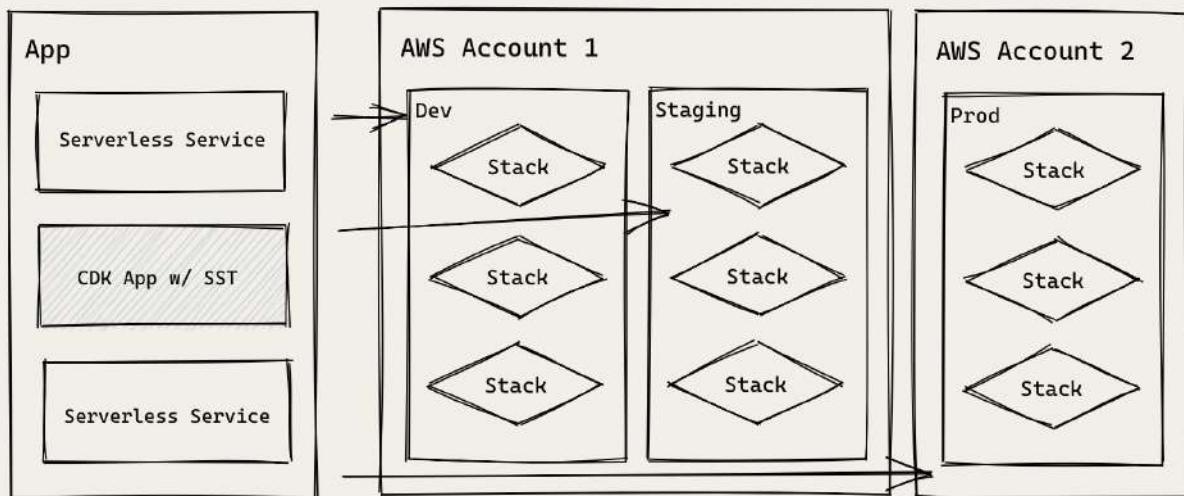
SST allows you to follow the same conventions as Serverless Framework. This means that you can deploy your Lambda functions using.

```
$ AWS_PROFILE=production serverless deploy --stage prod --region us-east-1
```

And use CDK for the rest of your AWS infrastructure.

```
$ AWS_PROFILE=production npx sst deploy --stage prod --region us-east-1
```

Just like Serverless Framework, the stacks in your CDK app are prefixed with the stage name. Now you can use Serverless Framework and CDK together! Allowing you to do something like this.



Serverless Framework with CDK using SST

Here, just like the Serverless Framework example above; our app is made up of three services. Except, one of those services is a CDK app deployed using SST!

We'll be deploying it using the `sst deploy` command, instead of the standard `cdk deploy` command. This'll make more sense in the coming chapters once we look at our infrastructure code.

Let's start by creating our SST project.

**Help and discussion**

View the [comments](#) for this chapter on our forums

Building a CDK app with SST

We are going to be using [AWS CDK](#) to create and deploy the infrastructure our Serverless app is going to need. We are using [Serverless Framework](#) for our APIs. And to use CDK with it, we'll be using the [Serverless Stack Toolkit \(SST\)](#). It's an extension of CDK that allows us to deploy it alongside our Serverless Framework service.

Let's get started.

Create a new SST app

◆ CHANGE In the root of your Serverless app run the following.

```
$ npx create-serverless-stack resources infrastructure
```

This will create your SST app in the `infrastructure/` directory inside your Serverless project.

◆ CHANGE Now let's go in there and do a quick build.

```
$ cd infrastructure  
$ npx sst build
```

You should see something like this.

```
Successfully compiled 1 stack to build/cdk.out:
```

```
dev-infrastructure-my-stack
```

There are template files created for us to use. We'll be overwriting them in the next chapter.

Update your config

Let's also quickly change the config a bit. It has your app name, the default stage and region we are deploying to.

◆ CHANGE Replace your `infrastructure/sst.json` with.

```
{  
  "name": "notes-infra",  
  "type": "@serverless-stack/resources",  
  "stage": "dev",  
  "region": "us-east-1"  
}
```

Now we are ready to configure our infrastructure. We'll look at DynamoDB first.



Help and discussion

View the [comments](#) for this chapter on our forums

Configure DynamoDB in CDK

We are now going to start creating our resources using CDK. Starting with DynamoDB.

Create a Stack



Add the following to `infrastructure/lib/DynamoDBStack.js`.

```
import { CfnOutput } from "@aws-cdk/core";
import * as dynamodb from "@aws-cdk/aws-dynamodb";
import * as sst from "@serverless-stack/resources";

export default class DynamoDBStack extends sst.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const app = this.node.root;

    const table = new dynamodb.Table(this, "Table", {
      billingMode: dynamodb.BillingMode.PAY_PER_REQUEST, // Use on-demand
      ↵ billing mode
      sortKey: { name: "noteId", type: dynamodb.AttributeType.STRING },
      partitionKey: { name: "userId", type: dynamodb.AttributeType.STRING },
    });

    // Output values
    new CfnOutput(this, "TableName", {
      value: table.tableName,
      exportName: app.logicalPrefixedName("TableName"),
    });
    new CfnOutput(this, "TableArn", {
```

```
    value: table.tableArn,
    exportName: app.logicalPrefixedName("TableArn"),
  });
}
}
```

Let's quickly go over what we are doing here.

1. We are creating a new stack for our DynamoDB table by extending `sst.Stack` instead of `cdk.Stack`. This is what allows us to deploy CDK alongside our Serverless services.
2. We are defining the table we created back in the [Create a DynamoDB Table](#) chapter. By default, CDK will generate a table name for us.
3. We add `userId` as a `partitionKey` and `noteId` as our `sortKey`.
4. We also set our `BillingMode` to `PAY_PER_REQUEST`. This is the **On-demand** option that we had selected in the AWS console.
5. We need to use the table name in our API. Also, we'll use the table ARN to ensure that our Lambda functions have access to this table. We don't want to hardcode these values. So we'll use a CloudFormation export, using the `CfnOutput` method with the `exportName` option. We'll later import these values in our API using [cross-stack references](#). The output names need to be unique per stack. While the `exportName` needs to be unique for a given region in the AWS account. To ensure that it'll be unique when we deploy to multiple environments, we'll use the `app.logicalPrefixedName` method. It's a convenience method in `sst.App` that prefixes a given name with the name of the stage (environment) and the name of the app. We'll use this method whenever we need to ensure uniqueness across environments.

You can refer to the CDK docs for more details on the [dynamodb.Table](#) construct.

Note that, we don't need to create a separate stack for each resource. We could use a single stack for all our resources. But for the purpose of illustration, we are going to split them all up.

◆ CHANGE Let's add the DynamoDB CDK package. Run the following in your `infrastructure`/ directory.

```
$ npx sst add-cdk @aws-cdk/aws-dynamodb
```

The reason we are using the `add-cdk` command instead of using an `npm install`, is because of a known issue with AWS CDK. Using mismatched versions of CDK packages can cause some unexpected problems down the road. The `sst add-cdk` command ensures that we install the right version of the package.

Add the Stack

Now let's add this stack to our app.

◆ CHANGE Replace your `infrastructure/lib/index.js` with this.

```
import DynamoDBStack from "./DynamoDBStack";  
  
// Add stacks  
export default function main(app) {  
  new DynamoDBStack(app, "dynamodb");  
}
```

We are now ready to deploy the DynamoDB stack in our app.

Deploy the Stack

◆ CHANGE To deploy your app run the following in the `infrastructure/` directory.

```
$ npx sst deploy
```

You should see something like this at the end of the deploy process.

```
Stack dev-notes-infra-dynamodb  
Status: deployed  
Outputs:  
  TableName: dev-notes-infra-dynamodb-TableCD117FA1-RBR93WLG5IQH  
  TableArn: arn:aws:dynamodb:us-east-1:087220554750:table/dev-notes-infra-  
    ↳ dynamodb-TableCD117FA1-RBR93WLG5IQH  
Exports:  
  dev-notes-infra-TableName:  
    ↳ dev-notes-infra-dynamodb-TableCD117FA1-RBR93WLG5IQH  
  dev-notes-infra-TableArn:  
    ↳ arn:aws:dynamodb:us-east-1:087220554750:table/dev-notes-infra-  
    ↳ dynamodb-TableCD117FA1-RBR93WLG5IQH
```

You'll notice the table name and ARN in the output and exported values.

Note that, we created a completely new DynamoDB table here. If you want to remove the old table we created manually through the console, you can do so now. We are going to leave it as is, in case you want to refer back to it at some point.

Remove Template Files

There are a couple of files that come with the template, that we can now remove.

◆ CHANGE Run this from the `infrastructure/` directory.

```
$ rm lib/MyStack.js  
$ rm README.md
```

Fix the Unit Tests

You can also setup unit tests for your stacks. We'll add a simple one here to show you how it works.

◆ CHANGE Start by renaming the `infrastructure/MyStack.test.js`.

```
$ mv test/MyStack.test.js test/DynamoDBStack.test.js
```

◆ CHANGE And replace `infrastructure/test/DynamoDBStack.test.js` with.

```
import { expect, haveResource } from "@aws-cdk/assert";  
import * as sst from "@serverless-stack/resources";  
import DynamoDBStack from "../lib/DynamoDBStack";  
  
test("Test Stack", () => {  
  const app = new sst.App();  
  // WHEN  
  const stack = new DynamoDBStack(app, "test-stack");  
  // THEN  
  expect(stack).to(  
    haveResource("AWS::DynamoDB::Table", {
```

```
    BillingMode: "PAY_PER_REQUEST",  
  })  
);  
});
```

This is a really simple test that ensure that our DynamoDBStack class is creating a DynamoDB table with the BillingMode set to PAY_PER_REQUEST.

And we can run the test using.

```
$ npx sst test
```

You should see something like this as your test output.

```
PASS  test/DynamoDBStack.test.js  
  \faCheck Test Stack (1022 ms)  
  
Test Suites: 1 passed, 1 total  
Tests:       1 passed, 1 total  
Snapshots:   0 total  
Time:        5.473 s  
Ran all test suites.
```

You can build on these tests later when you stack becomes more complicated.

For now let's move on to S3 and create a bucket for file uploads.



Help and discussion

View the [comments for this chapter on our forums](#)

Configure S3 in CDK

Now that we have our DynamoDB table configured, let's look at how we can configure our S3 file uploads bucket using CDK.

Create a Stack

◆ CHANGE Add the following to `infrastructure/lib/S3Stack.js`.

```
import * as cdk from "@aws-cdk/core";
import * as s3 from "@aws-cdk/aws-s3";
import * as sst from "@serverless-stack/resources";

export default class S3Stack extends sst.Stack {
    // Public reference to the S3 bucket
    bucket;

    constructor(scope, id, props) {
        super(scope, id, props);

        this.bucket = new s3.Bucket(this, "Uploads", {
            // Allow client side access to the bucket from a different domain
            cors: [
                {
                    maxAge: 3000,
                    allowedOrigins: ["*"],
                    allowedHeaders: ["*"],
                    allowedMethods: ["GET", "PUT", "POST", "DELETE", "HEAD"],
                },
            ],
        });
    }
}
```

```
// Export values
new cdk.CfnOutput(this, "AttachmentsBucketName", {
  value: this.bucket.bucketName,
})};
}
```

If you recall from the [Create an S3 bucket for file uploads](#) chapter, we had created a bucket and configured the CORS policy for it. The CORS policy is necessary because we are uploading directly from our frontend client. We'll configure the same policy here.

Just like the DynamoDB stack that [we created in the last chapter](#), we are going to output the name of the bucket that we created. However, we don't need to create a CloudFormation export because we need this value in our React app. And there isn't really a way to import CloudFormation exports there.

The one thing that we are doing differently here is that we are creating a public class property called `bucket`. It holds a reference to the bucket that is created in this stack. We'll refer to this later when [creating our Cognito IAM policies](#).

You can refer to the CDK docs for more details on the [s3.Bucket](#) construct.

◆ **CHANGE** Let's add the S3 CDK package. Run the following in your `infrastructure`/directory.

```
$ npx sst add-cdk @aws-cdk/aws-s3
```

This will do an `npm install` using the right CDK version.

Add the Stack

◆ **CHANGE** Let's add this stack to our CDK app. Replace your `infrastructure/lib/index.js` with this.

```
import S3Stack from './S3Stack';
import DynamoDBStack from './DynamoDBStack';
```

```
// Add stacks
export default function main(app) {
  new DynamoDBStack(app, "dynamodb");

  new S3Stack(app, "s3");
}
```

Deploy the Stack

◆ CHANGE Now let's deploy our new stack by running the following from the `infrastructure/` directory.

```
$ npx sst deploy
```

You should see something like this at the end of your deploy output.

```
Stack dev-notes-infra-s3
  Status: deployed
  Outputs:
    AttachmentsBucketName: dev-notes-infra-s3-uploads4f6eb0fd-18yd4altuql9g
```

You'll notice the output has the name of our newly created S3 bucket.

And that's it. Next let's look into configuring our Cognito User Pool.



Help and discussion

View the [comments for this chapter on our forums](#)

Configure Cognito User Pool in CDK

So far we've configured our DynamoDB table and S3 bucket in CDK. We are now ready to setup our Cognito User Pool. The User Pool stores our user credentials and allows our users to sign up and login to our app.

Create a Stack



Add the following to `infrastructure/lib/CognitoStack.js`.

```
import { CfnOutput } from "@aws-cdk/core";
import * as cognito from "@aws-cdk/aws-cognito";
import * as sst from "@serverless-stack/resources";

export default class CognitoStack extends sst.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const userPool = new cognito.UserPool(this, "UserPool", {
      selfSignUpEnabled: true, // Allow users to sign up
      autoVerify: { email: true }, // Verify email addresses by sending a
        ↴ verification code
      signInAliases: { email: true }, // Set email as an alias
    });

    const userPoolClient = new cognito.UserPoolClient(this, "UserPoolClient",
      ↴ {
        userPool,
        generateSecret: false, // Don't need to generate secret for web app
          ↴ running on browsers
      });
  }
}
```

```
// Export values
new CfnOutput(this, "UserPoolId", {
  value: userPool.userPoolId,
}) ;
new CfnOutput(this, "UserPoolClientId", {
  value: userPoolClient.userPoolClientId,
}) ;
}
}
```

Let's quickly go over what we are doing here:

- We are creating a new stack for our Cognito related resources. We don't have to create a separate stack here. We could've used one of our existing stacks. But this setup allows us to illustrate how you would use multiple stacks together.
- We are creating a new instance of the `cognito.UserPool` class. And setting up the options to match what we did back in the [Create a Cognito user pool](#) chapter.
- We then create a new instance of the `cognito.UserPoolClient` class and link it to the User Pool we defined above.
- Finally, we output the `UserPoolId` and `UserPoolClientId`. We'll be using this later in our React app.

You can refer to the CDK docs to learn more about the `cognito.UserPool` and the `cognito.UserPoolClient` constructs.

◆ CHANGE Let's add the Cognito CDK package. Run the following in your `infrastructure`/ directory.

```
$ npx sst add-cdk @aws-cdk/aws-cognito
```

This will do an `npm install` using the right CDK version.

Add the Stack

◆ CHANGE Let's add this stack to our CDK app. Replace your `infrastructure/lib/index.js` with this.

```
import S3Stack from "./S3Stack";
import CognitoStack from "./CognitoStack";
import DynamoDBStack from "./DynamoDBStack";

// Add stacks
export default function main(app) {
  new DynamoDBStack(app, "dynamodb");

  new S3Stack(app, "s3");

  new CognitoStack(app, "cognito");
}
```

Deploy the Stack

◆ CHANGE Now let's deploy our new Cognito User Pool by running the following from the infrastructure/ directory.

```
$ npx sst deploy
```

You should see something like this at the end of your deploy output.

```
Stack dev-notes-infra-cognito
Status: deployed
Outputs:
  UserPoolClientId: 68clr7ilru7rheikb3g8gvgvfq
  UserPoolId: us-east-1_LqVQhcQDe
```

We'll be copying over these values in one of our later chapters.

Next, let's look at configuring our Cognito Identity Pool.



Help and discussion

View the [comments for this chapter on our forums](#)

Configure Cognito Identity Pool in CDK

Over the past few chapters we've created our DynamoDB table, S3 bucket, and Cognito User Pool in CDK. We are now ready to tie them together using a Cognito Identity Pool. This tells AWS which of our resources are available to our logged in users. You can read more about Identity Pools in the [Cognito User Pool vs Identity Pool](#) chapter.

Add the Identity Pool

◆ CHANGE Replace your infrastructure/lib/CognitoStack.js with the following.

```
import { CfnOutput } from "@aws-cdk/core";
import * as cognito from "@aws-cdk/aws-cognito";
import * as sst from "@serverless-stack/resources";

export default class CognitoStack extends sst.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const userPool = new cognito.UserPool(this, "UserPool", {
      selfSignUpEnabled: true, // Allow users to sign up
      autoVerify: { email: true }, // Verify email addresses by sending a
        ↴ verification code
      signInAliases: { email: true }, // Set email as an alias
    });

    const userPoolClient = new cognito.UserPoolClient(this, "UserPoolClient",
      ↴ {
        userPool,
        generateSecret: false, // Don't need to generate secret for web app
          ↴ running on browsers
    });
  }
}
```

```
});  
  
const identityPool = new cognito.CfnIdentityPool(this, "IdentityPool", {  
    allowUnauthenticatedIdentities: false, // Don't allow unauthenticated  
    // users  
    cognitoIdentityProviders: [  
        {  
            clientId: userPoolClient.userPoolClientId,  
            providerName: userPool.userPoolProviderName,  
        },  
    ],  
});  
  
// Export values  
new CfnOutput(this, "UserPoolId", {  
    value: userPool.userPoolId,  
});  
new CfnOutput(this, "UserPoolClientId", {  
    value: userPoolClient.userPoolClientId,  
});  
new CfnOutput(this, "IdentityPoolId", {  
    value: identityPool.ref,  
});  
}  
}
```

Let's quickly highlight the changes and go over them.

We are creating a new CfnIdentityPool and linking it to the User Pool that we created [in the last chapter](#).

```
+ const identityPool = new cognito.CfnIdentityPool(this, "IdentityPool", {  
+   allowUnauthenticatedIdentities: false, // Don't allow unauthenticated users  
+   cognitoIdentityProviders: [  
+     {  
+       clientId: userPoolClient.userPoolClientId,  
+       providerName: userPool.userPoolProviderName,  
+     },  
+
```

```
+  ] ,  
+ });
```

And we output the id of the Identity Pool that we just created.

```
+ new CfnOutput(this, "IdentityPoolId", {  
+   value: identityPool.ref,  
+ });
```

You can refer to the CDK docs to learn more about the [cognito.CfnIdentityPool](#) construct.

Deploy the Stack

◆ **CHANGE** Let's quickly deploy this. Run the following from the `infrastructure/` directory.

```
$ npx sst deploy
```

You should see something like this at the end of your deploy output.

```
Stack dev-notes-infra-cognito  
Status: deployed  
Outputs:  
  UserPoolClientId: 1jh98ercq1aksvmlq0sla1qm9n  
  UserPoolId: us-east-1_Nzpw587R8  
  IdentityPoolId: us-east-1:9bf24959-2085-4802-add3-183c8842e6ae
```

Add the Cognito Authenticated Role

Now we are ready to add the IAM role our authenticated users will assume. We could simply add it directly in the `CognitoStack` class. But let's use this opportunity to explore an aspect of CDK that really sets it apart from the old CloudFormation way of using YAML or JSON. The ability to easily create custom constructs.

Create a Construct in CDK

So far we've been using the built-in constructs that come with AWS CDK. Now let's create one of our own. We are going to abstract out the process of creating an authenticated IAM role. It allows us to separate the complexity involved in creating this role and ensure that we can reuse it later when we are working with Identity Pools.

◆ CHANGE Create a new file in `infrastructure/lib/CognitoAuthRole.js` and add:

```
import * as cdk from "@aws-cdk/core";
import * as iam from "@aws-cdk/aws-iam";
import * as cognito from "@aws-cdk/aws-cognito";

export default class CognitoAuthRole extends cdk.Construct {
    // Public reference to the IAM role
    role;

    constructor(scope, id, props) {
        super(scope, id);

        const { identityPool } = props;

        // IAM role used for authenticated users
        this.role = new iam.Role(this, "CognitoDefaultAuthenticatedRole", {
            assumedBy: new iam.FederatedPrincipal(
                "cognito-identity.amazonaws.com",
                {
                    StringEquals: {
                        "cognito-identity.amazonaws.com:aud": identityPool.ref,
                    },
                    "ForAnyValue:StringLike": {
                        "cognito-identity.amazonaws.com:amr": "authenticated",
                    },
                },
                "sts:AssumeRoleWithWebIdentity"
            ),
        });
        this.role.addToPolicy(
            new iam.PolicyStatement({
```

```
    effect: iam.Effect.ALLOW,
    actions: [
        "mobileanalytics:PutEvents",
        "cognito-sync:*",
        "cognito-identity:*",
    ],
    resources: ["*"],
)
);

new cognito.CfnIdentityPoolRoleAttachment(
    this,
    "IdentityPoolRoleAttachment",
{
    identityPoolId: identityPool.ref,
    roles: { authenticated: this.role.roleArn },
}
);
}

}
```

Here's what we are doing here:

- We are creating a construct called `CognitoAuthRole` by extending `cdk.Construct`.
- It takes an `identityPool` as a prop.
- Then we create a new IAM role using `iam.Role`. We also specify that it can be assumed by users that our authenticated with the Identity Pool that's passed in.
- We assign our newly created role to `this.role`, a public class property. This is so that we can access this role in our Cognito stack.
- We also import the `aws-iam` construct up top.
- We add a policy to this role, using the `addToPolicy` method. It's a standard Cognito related policy. We did this back in the [Create a Cognito Identity Pool](#) chapter.
- Finally, we attach this newly created role to our Identity Pool by creating a new `cognito.CfnIdentityPoolRoleAttachment`.

You can refer to the CDK docs to learn more about the `iam.Role` and `cognito.CfnIdentityPoolRoleAttachment` constructs.

◆ CHANGE Let's add the IAM CDK package. Run the following in your infrastructure/ directory.

```
$ npx sst add-cdk @aws-cdk/aws-iam
```

Using the New Construct

We are now ready to add the authenticated role to our Cognito stack. We'll use our newly created construct. And use the S3 bucket that we previously created to restrict access for logged in users.

◆ CHANGE Replace your infrastructure/lib/CognitoStack.js with this.

```
import { CfnOutput } from "@aws-cdk/core";
import * as iam from "@aws-cdk/aws-iam";
import * as cognito from "@aws-cdk/aws-cognito";
import * as sst from "@serverless-stack/resources";
import CognitoAuthRole from "./CognitoAuthRole";

export default class CognitoStack extends sst.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const { bucketArn } = props;

    const app = this.node.root;

    const userPool = new cognito.UserPool(this, "UserPool", {
      selfSignUpEnabled: true, // Allow users to sign up
      autoVerify: { email: true }, // Verify email addresses by sending a
        verification code
      signInAliases: { email: true }, // Set email as an alias
    });

    const userPoolClient = new cognito.UserPoolClient(this, "UserPoolClient",
      {
        ...
      }
    );
  }
}
```

```
userPool,  
  generateSecret: false, // Don't need to generate secret for web app  
  ↪ running on browsers  
};  
  
const identityPool = new cognito.CfnIdentityPool(this, "IdentityPool", {  
  allowUnauthenticatedIdentities: false, // Don't allow unauthenticated  
  ↪ users  
  cognitoIdentityProviders: [  
    {  
      clientId: userPoolClient.userPoolClientId,  
      providerName: userPool.userPoolProviderName,  
    },  
    ],  
  };  
  
const authenticatedRole = new CognitoAuthRole(this, "CognitoAuthRole", {  
  identityPool,  
});  
  
authenticatedRole.role.addToPolicy(  
  // IAM policy granting users permission to a specific folder in the S3  
  ↪ bucket  
  new iam.PolicyStatement({  
    actions: ["s3:*"],  
    effect: iam.Effect.ALLOW,  
    resources: [  
      bucketArn + "/private/${cognito-identity.amazonaws.com:sub}/*",  
    ],  
  })  
);  
  
// Export values  
new CfnOutput(this, "UserPoolId", {  
  value: userPool.userPoolId,  
});  
new CfnOutput(this, "UserPoolClientId", {
```

```
        value: userPoolClient.userPoolClientId,  
    );  
    new CfnOutput(this, "IdentityPoolId", {  
        value: identityPool.ref,  
    });  
    new CfnOutput(this, "AuthenticatedRoleName", {  
        value: authenticatedRole.role.roleName,  
        exportName: app.logicalPrefixedName("CognitoAuthRole"),  
    });  
}  
}  
}
```

Let's go over the changes we are making here.

We first import our new construct.

```
+ import CognitoAuthRole from "./CognitoAuthRole";
```

We then get a reference to the bucketArn of our previously created S3 bucket. We'll be passing this in shortly.

```
+ const { bucketArn } = props;  
  
+ const app = this.node.root;
```

Then we create a new instance of our CognitoAuthRole and assign it to authenticatedRole.

```
+ const authenticatedRole = new CognitoAuthRole(this, "CognitoAuthRole", {  
    +   identityPool,  
+});
```

We access the new IAM role we are creating through authenticatedRole.role. And add a new policy to it. It grants permission to a specific folder in the S3 bucket we created. This ensures that authenticated users can only access their uploaded files (and not any other user's uploads). We talked about how this works back in the [Create a Cognito Identity Pool](#) chapter.

```
+ authenticatedRole.role.addToPolicy(
+   new iam.PolicyStatement({
+     actions: ["s3:*"],
+     effect: iam.Effect.ALLOW,
+     resources: [
+       bucketArn + "/private/${cognito-identity.amazonaws.com:sub}/*",
+     ],
+   })
+ );
```

Finally, we export the name of the role that we just created. We'll use this later in our Serverless API. We use the `app.logicalPrefixedName` method to ensure that our exported name is unique across multiple environments.

```
+ new CfnOutput(this, "AuthenticatedRoleName", {
+   value: authenticatedRole.role.roleName,
+   exportName: app.logicalPrefixedName("CognitoAuthRole"),
+});
```

Now let's pass in the S3 bucket info from our other stack.

◆ CHANGE Replace your `infrastructure/lib/index.js` with this.

```
import S3Stack from "./S3Stack";
import CognitoStack from "./CognitoStack";
import DynamoDBStack from "./DynamoDBStack";

// Add stacks
export default function main(app) {
  new DynamoDBStack(app, "dynamodb");

  const s3 = new S3Stack(app, "s3");

  new CognitoStack(app, "cognito", { bucketArn: s3.bucket.bucketArn });
}
```

You'll notice we are taking the `bucket` property of the `S3Stack` and passing it in as the `bucketArn` to our `CognitoStack`.

Redeploy the Stack

Now let's redeploy to update our Cognito Identity Pool.



Run the following from the `infrastructure/` directory.

```
$ npx sst deploy
```

You should now see the newly exported auth role name.

```
Stack dev-notes-infra-cognito
Status: deployed
Outputs:
  AuthenticatedRoleName:
    ↳ dev-notes-infra-cognito-CognitoAuthRoleCognitoDefa-14TSUK0GNJIBU
  UserPoolClientId: 1jh98ercq1aksvmlq0sla1qm9n
  UserPoolId: us-east-1_Nzpw587R8
  IdentityPoolId: us-east-1:9bf24959-2085-4802-add3-183c8842e6ae
Exports:
  dev-notes-infra-CognitoAuthRole:
    ↳ dev-notes-infra-cognito-CognitoAuthRoleCognitoDefa-14TSUK0GNJIBU
```

And the infrastructure of our app has now been completely configured in code, thanks to CDK! And it's deployed in a way that is compatible with Serverless Framework, thanks to SST!

Next let's connect Serverless Framework to our CDK SST app.



Help and discussion

View the [comments for this chapter](#) on our forums

Connect Serverless Framework and CDK with SST

Now that we have configured the infrastructure for our Serverless app using CDK. Let's look at how we can connect it to our Serverless Framework project. The conventions enforced by [SST](#) makes this easy to do.

Reference Your SST App

Start by adding a reference to your SST app in your `serverless.yml`.

◆ CHANGE Add the following `custom:` block at the top of our `services/notes/serverless.yml` above the `provider:` block.

```
custom:  
  # Our stage is based on what is passed in when running serverless  
  # commands. Or falls back to what we have set in the provider section.  
  stage: ${opt:stage, self:provider.stage}  
  # Name of the SST app that's deploying our infrastructure  
  sstApp: ${self:custom.stage}-notes-infra
```

Here `notes-infra` is the name of our SST app as defined in `infrastructure/sst.json`.

```
{  
  "name": "notes-infra",  
  "type": "@serverless-stack/resources",  
  "stage": "dev",  
  "region": "us-east-1"  
}
```

Let's look at what we are defining in your `serverless.yml` in a little more detail.

1. 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).
2. Next, we set the name of our SST app as a custom variable. This includes the name of the stage as well — \${self:custom.stage}-notes-infra. It's configured such that it references the SST app for the stage the current Serverless app is deployed to. So if you deploy your API app to dev, it'll reference the dev version of the SST notes app.

These two simple steps allow us to (loosely) link our Serverless Framework and CDK app using SST.

Just for reference, the top of our `serverless.yml` should look something like this.

```
service: notes-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

custom:
  # Our stage is based on what is passed in when running serverless
  # commands. Or falls back to what we have set in the provider section.
  stage: ${opt:stage, self:provider.stage}
  # Name of the SST app that's deploying our infrastructure
  sstApp: ${self:custom.stage}-notes-infra

provider:
  name: aws
  runtime: nodejs12.x
```

```
stage: dev
region: us-east-1

...
```

Reference DynamoDB

Next let's programmatically reference the DynamoDB table that we created using CDK.

◆ CHANGE Replace the environment and iamRoleStatements block with in your serverless.yml with.

```
# These environment variables are made available to our functions
# under process.env.

environment:
  stripeSecretKey: ${env:STRIPE_SECRET_KEY}
  tableName: !ImportValue '${self:custom.sstApp}-TableName'

iamRoleStatements:
  - Effect: Allow
    Action:
      - dynamodb:Scan
      - dynamodb:Query
      - dynamodb:GetItem
      - dynamodb:PutItem
      - dynamodb:UpdateItem
      - dynamodb:DeleteItem
      - dynamodb:DescribeTable
    # Restrict our IAM role permissions to
    # the specific table for the stage
  Resource:
    - !ImportValue '${self:custom.sstApp}-TableArn'
```

Make sure to **copy the indentation** correctly. Your provider block should look something like this.

```

provider:
  name: aws
  runtime: nodejs12.x
  stage: dev
  region: us-east-1

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

  environment:
    stripeSecretKey: ${env:STRIPE_SECRET_KEY}
    tableName: !ImportValue '${self:custom.sstApp}-TableName'

  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:Scan
        - dynamodb:Query
        - dynamodb:.GetItem
        - dynamodb:PutItem
        - dynamodb:UpdateItem
        - dynamodb:DeleteItem
        - dynamodb:DescribeTable
      # Restrict our IAM role permissions to
      # the specific table for the stage
      Resource:
        - !ImportValue '${self:custom.sstApp}-TableArn'

```

Let's look at what we are doing here.

- We'll use the name of the SST app to import the CloudFormation exports that we setup in our DynamoDBStack class back in the [Configure DynamoDB in CDK](#) chapter.
- We'll then change the `tableName` from the hardcoded notes to `!ImportValue '${self:custom.sstApp}-TableName'`. This imports the table name that we exported in CDK.
- Similarly, we'll import the table ARN using `!ImportValue '${self:custom.sstApp}-TableArn'`. Previously, we were giving our Lambda functions access to all DynamoDB tables in our region. Now we are able to lockdown our

permissions a bit more specifically.

You might have picked up that we are using the stage name extensively in our setup. This is because we want to ensure that we can deploy our app to multiple environments simultaneously. This setup allows us to create and destroy new environments simply by changing the stage name.

Add to the Cognito Authenticated Role

While we are on the topic of giving our Lambda functions IAM access. We'll need to do something similar for our API. In the [previous chapter](#), we created an IAM role our authenticated users will use. It allows them to uploads files to their folder in S3. But we also need to allow them to access our API endpoint.

Note that, we don't need to explicitly give them access to our Lambda functions or DynamoDB table. This is because we are securing access at the level of the API endpoint. We assume that if you can access our endpoint, you have access to our Lambda functions. And the DynamoDB permissions that we setup above are not for our users, but our Lambda functions. We don't do this for our S3 bucket because the user is directly uploading files to S3. So we need to secure access to it as well. Put another way, the two external touch points our user has is our API endpoint and S3 bucket. And that's what we need to secure access to.

So let's add our API endpoint to the authenticated role we previously created in CDK.

◆ CHANGE Add the following to `services/notes/resources/cognito-policy.yml`.

Resources:

```
CognitoAuthorizedApiPolicy:  
  Type: AWS::IAM::Policy  
  Properties:  
    PolicyName: ${self:custom.stage}-CognitoNotesAuthorizedApiPolicy  
    PolicyDocument:  
      Version: "2012-10-17"  
      Statement:  
        - Effect: "Allow"  
          Action:  
            - "execute-api:Invoke"  
          Resource:  
            !Sub 'arn:aws:execute-  
              ↵ api:${AWS::Region}:${AWS::AccountId}:#{ApiGatewayRestApi}/*'
```

Roles:

```
- !ImportValue '${self:custom.sstApp}-CognitoAuthRole'
```

While YAML can be a bit hard to read, here is what we are doing.

- We create a new policy called \${self:custom.stage}-CognitoNotesAuthorizedApiPolicy. We make sure it's unique when we deploy it to multiple environments.
- This policy has execute-api:Invoke access to the arn:aws:execute-api:\${AWS::Region}:\${AWS::AccountId}:\${ApiGatewayRestApi}/* resource. Once we attach this resource to our API, the ApiGatewayRestApi variable will be replaced with the API we are creating.
- Finally, we attach this policy to the role we previously created (and exported), !ImportValue '\${self:custom.sstApp}-CognitoAuthRole'. If you go back and look at the end of the [previous chapter](#), you'll notice the above export.

Now let's add this resource to our API.

◆ **CHANGE** Replace the resources: block at the bottom of our services/notes/serverless.yml with.

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

And now we are ready to deploy our (completely programmatically created) Serverless infrastructure!

**Help and discussion**

View the [comments for this chapter](#) on our forums

Deploy Your Serverless Infrastructure

Now that we have all our resources configured, let's go ahead and deploy our entire infrastructure.

Deploy Your SST CDK App

Let's deploy our SST app once to make sure all of our changes have been deployed.

◆ CHANGE Run the following from your `infrastructure` directory.

```
$ npx sst deploy
```

You should see something like this in your output.

```
Stack dev-notes-infra-dynamodb
  Status: deployed
  Outputs:
    TableName: dev-notes-infra-dynamodb-TableCD117FA1-1B701ZU6DS6IR
    TableArn: arn:aws:dynamodb:us-east-1:087220554750:table/dev-notes-infra-
      ↵ dynamodb-TableCD117FA1-1B701ZU6DS6IR
  Exports:
    dev-notes-infra-TableName:
      ↵ dev-notes-infra-dynamodb-TableCD117FA1-1B701ZU6DS6IR
    dev-notes-infra-TableArn:
      ↵ arn:aws:dynamodb:us-east-1:087220554750:table/dev-notes-infra-
      ↵ dynamodb-TableCD117FA1-1B701ZU6DS6IR
```

```
Stack dev-notes-infra-s3
  Status: deployed
  Outputs:
```

```
AttachmentsBucketName: dev-notes-infra-s3-uploads4f6eb0fd-1taash9pf6q1f
ExportsOutputFnGetAttUploads4F6EB0FDArn5513CBEA:
    ↳ arn:aws:s3:::dev-notes-infra-s3-uploads4f6eb0fd-1taash9pf6q1f
Exports:
  dev-notes-infra-s3:ExportsOutputFnGetAttUploads4F6EB0FDArn5513CBEA:
    ↳ arn:aws:s3:::dev-notes-infra-s3-uploads4f6eb0fd-1taash9pf6q1f

Stack dev-notes-infra-cognito
Status: deployed
Outputs:
  AuthenticatedRoleName:
    ↳ dev-notes-infra-cognito-CognitoAuthRoleCognitoDefa-14TSUK0GNJIBU
  UserPoolClientId: 1jh98ercqlaksvmlq0sla1qm9n
  UserPoolId: us-east-1_Nzpw587R8
  IdentityPoolId: us-east-1:9bf24959-2085-4802-add3-183c8842e6ae
Exports:
  dev-notes-infra-CognitoAuthRole:
    ↳ dev-notes-infra-cognito-CognitoAuthRoleCognitoDefa-14TSUK0GNJIBU
```

We'll be using these outputs later when we update our React app.

Deploy Your Serverless API

Now let's deploy your API.

◆ CHANGE From your services/notes/ directory run:

```
$ serverless deploy -v
```

Your output should look something like this:

```
Service Information
service: notes-api
stage: dev
region: us-east-1
stack: notes-api-dev
```

```
resources: 44
api keys:
  None
endpoints:
  POST - https://5opmr1alga.execute-api.us-east-1.amazonaws.com/dev/notes
  GET - https://5opmr1alga.execute-api.us-east-1.amazonaws.com/dev/notes/{id}
  GET - https://5opmr1alga.execute-api.us-east-1.amazonaws.com/dev/notes
  PUT - https://5opmr1alga.execute-api.us-east-1.amazonaws.com/dev/notes/{id}
  DELETE -
    ↳ https://5opmr1alga.execute-api.us-east-1.amazonaws.com/dev/notes/{id}
  POST - https://5opmr1alga.execute-api.us-east-1.amazonaws.com/dev/billing
functions:
  create: notes-api-dev-create
  get: notes-api-dev-get
  list: notes-api-dev-list
  update: notes-api-dev-update
  delete: notes-api-dev-delete
  billing: notes-api-dev-billing
layers:
  None
```

Stack Outputs

```
DeleteLambdaFunctionQualifiedArn:
  ↳ arn:aws:lambda:us-east-1:087220554750:function:notes-api-dev-delete:3
CreateLambdaFunctionQualifiedArn:
  ↳ arn:aws:lambda:us-east-1:087220554750:function:notes-api-dev-create:3
GetLambdaFunctionQualifiedArn:
  ↳ arn:aws:lambda:us-east-1:087220554750:function:notes-api-dev-get:3
UpdateLambdaFunctionQualifiedArn:
  ↳ arn:aws:lambda:us-east-1:087220554750:function:notes-api-dev-update:3
BillingLambdaFunctionQualifiedArn:
  ↳ arn:aws:lambda:us-east-1:087220554750:function:notes-api-dev-billing:1
ListLambdaFunctionQualifiedArn:
  ↳ arn:aws:lambda:us-east-1:087220554750:function:notes-api-dev-list:3
ServiceEndpoint: https://5opmr1alga.execute-api.us-east-1.amazonaws.com/dev
ServerlessDeploymentBucketName:
  ↳ notes-api-dev-serverlessdeploymentbucket-1323e6pius3a
```

We'll be using the `ServiceEndpoint` later when we update our React app.

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, our app should 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



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

```
$ git add .  
$ git commit -m "Setting up our Serverless infrastructure"  
$ git push
```



Help and discussion

View the [comments](#) for this chapter on our forums



For reference, here is the code we are using

Backend Source: [deploy-your-serverless-infrastructure](#)

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](#). Full disclosure, we also built Seed. You can replace most of this section with a service like [Travis CI](#) or [CircleCI](#). Using a CI service (like Travis CI or CircleCI) can a bit more cumbersome and needs some scripting. But we have a couple of posts on this over on the [Seed](#) blog to help you out:

- [Configure a CI/CD pipeline for Serverless apps on CircleCI](#)
- [Configure a CI/CD pipeline for Serverless apps on Travis CI](#)

Let's get started with setting up your project on Seed.



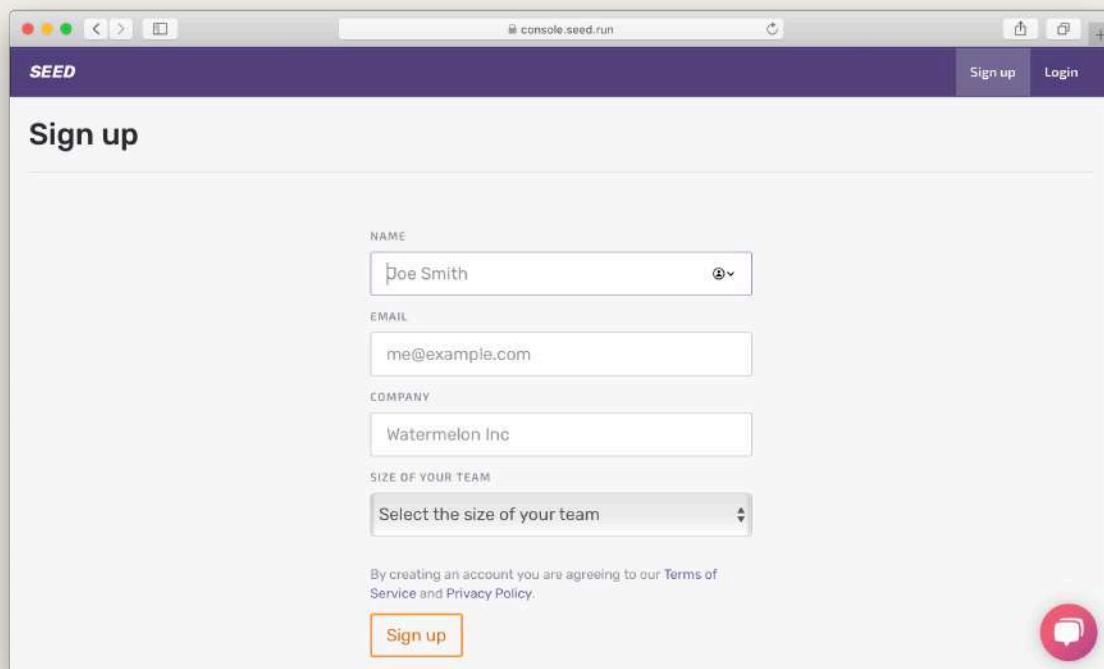
Help and discussion

View the [comments for this chapter on our forums](#)

Setting up Your Project on Seed

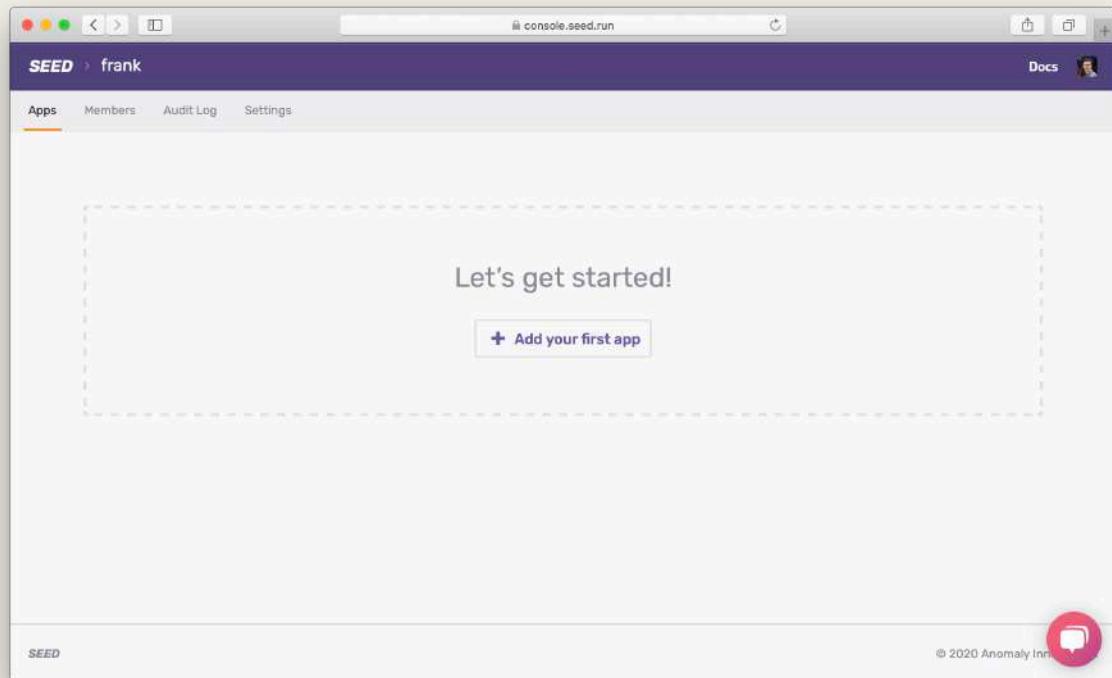
We are going to use [Seed](#) to automate our serverless deployments and manage our environments.

Start by signing up for a free account [here](#).



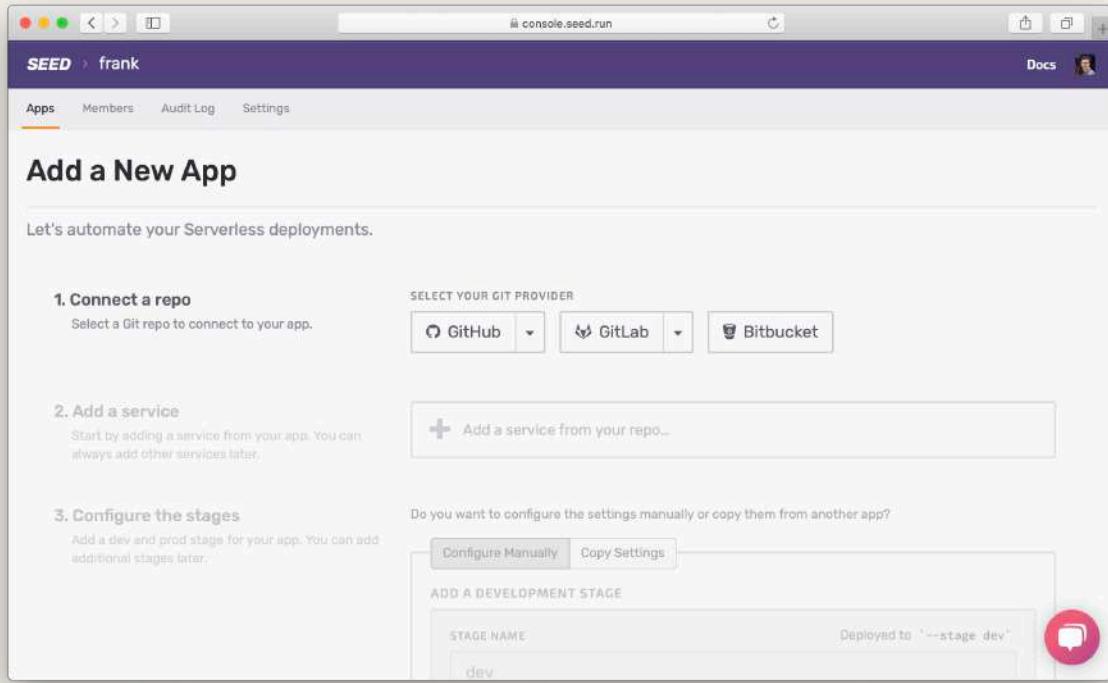
Create new Seed account screenshot

Let's **Add your first app**.



Add your first Seed app screenshot

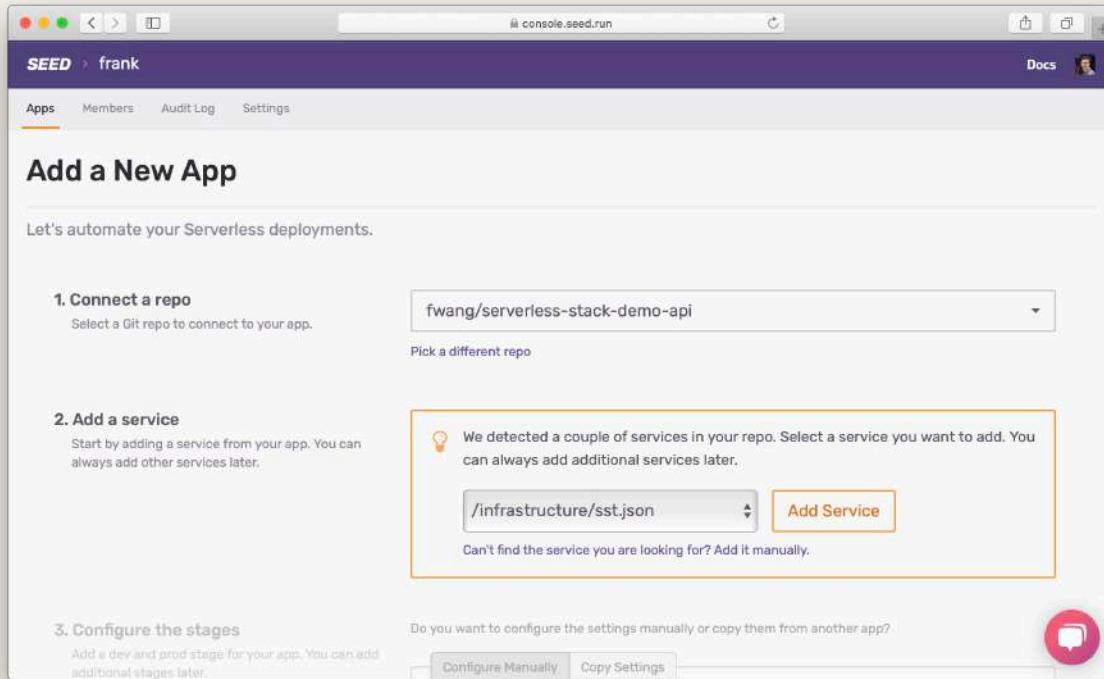
Now to add your project, select **GitHub** as your git provider. You'll be asked to give Seed permission to your GitHub account.



Select Git provider screenshot

Select the repo we've been using so far.

Next, Seed will automatically detect `sst.json` and `serverless.yml` files in your repo. Select the **infrastructure** service. Then click **Add Service**. We'll add our API later.



Serverless.yml detected screenshot

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](#). But for now we'll simply use the one we've used in this tutorial.

◆ CHANGE Run the following command.

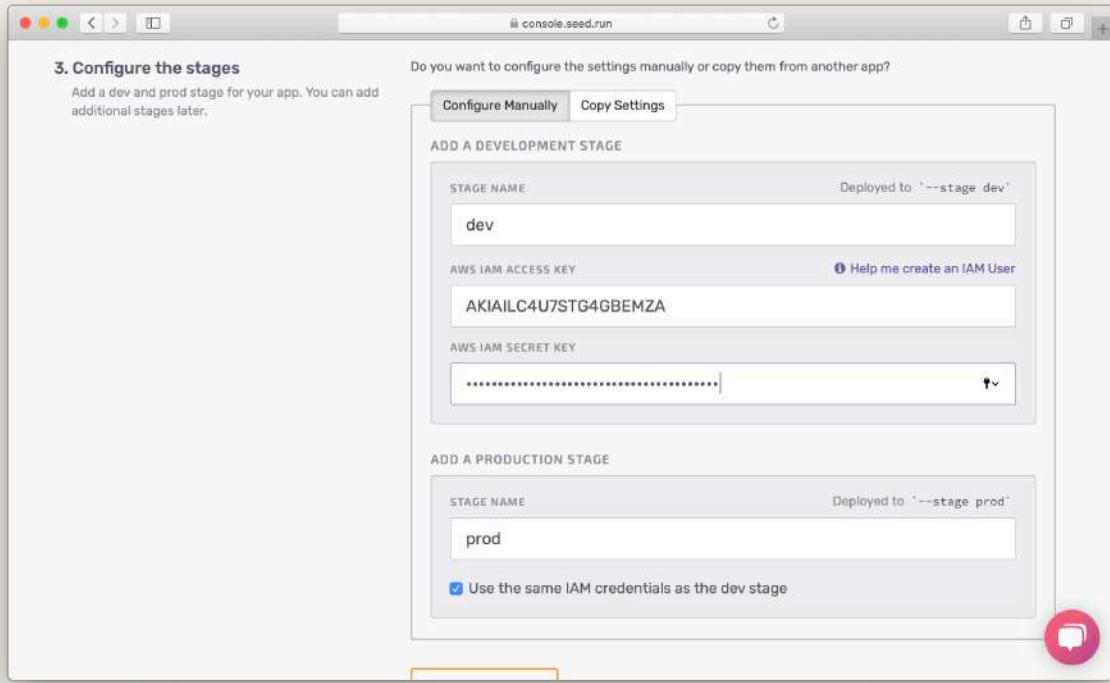
```
$ cat ~/.aws/credentials
```

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.

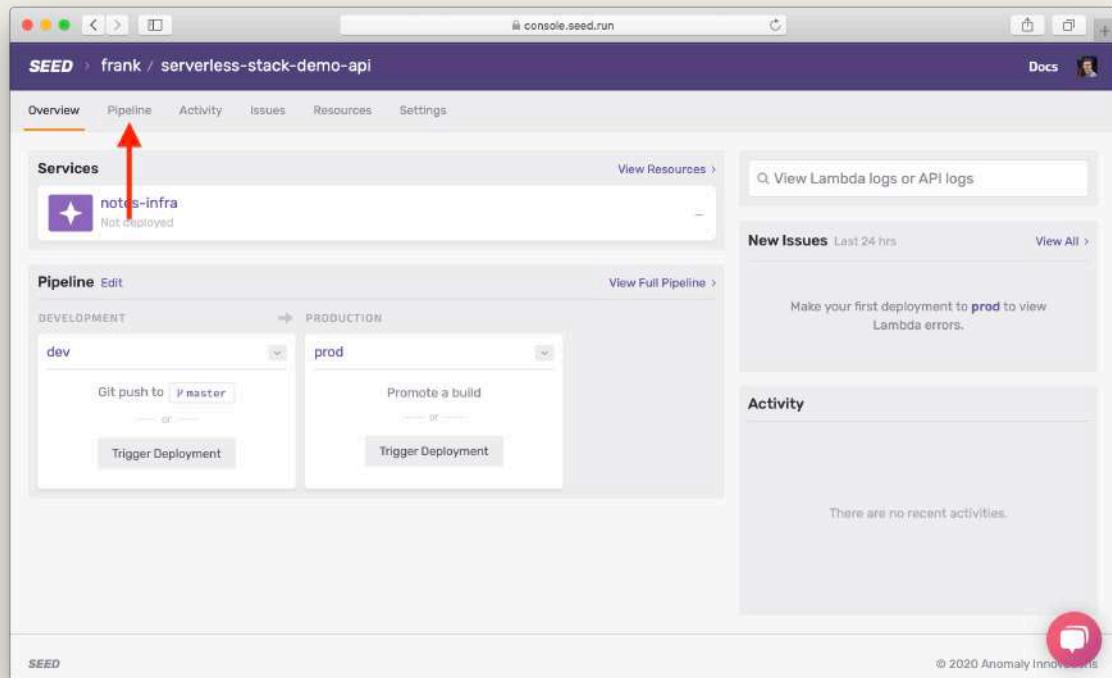
Fill in the credentials and click **Add a New App**.



Add AWS IAM credentials screenshot

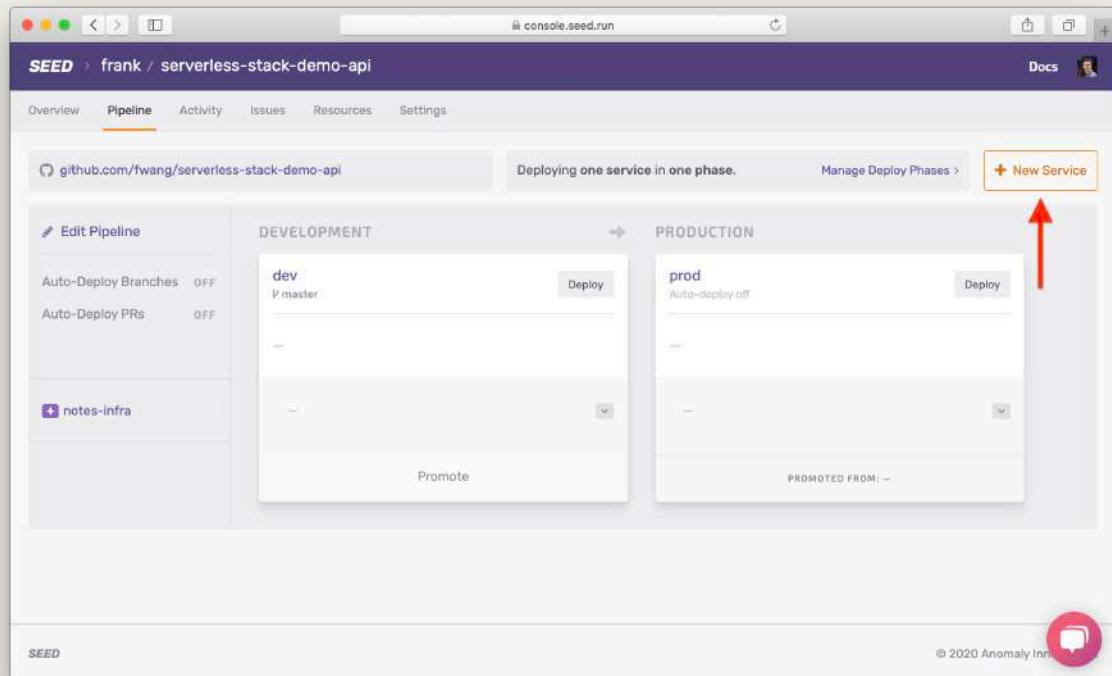
Your new app is created. You'll notice a few things here. First, we have a service called **notes-infra**. It's picking up the service name from our `sst.json`. You can choose to change this by clicking on the service and editing its name. You'll also notice the two stages that have been created.

A Serverless app can have multiple services within it. A service (roughly speaking) is a reference to `asst.json` or `serverless.yml` file. In our case we have two services in our repo. Let's add the API service. Click **Pipeline**.



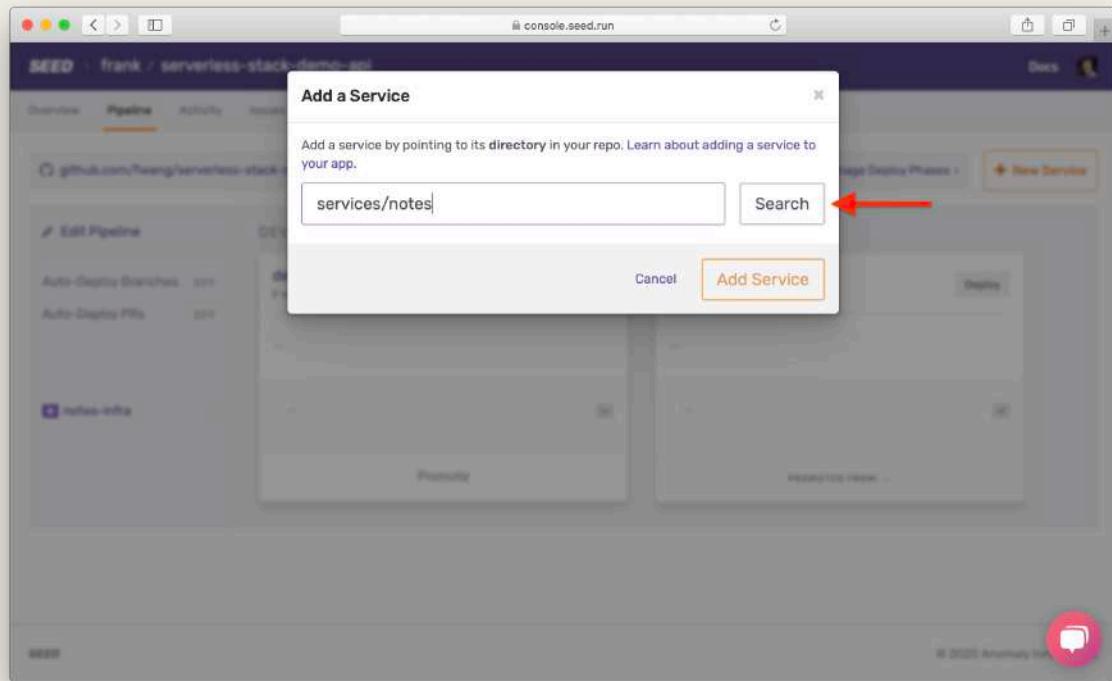
Click pipeline screenshot

Click **New Service**.



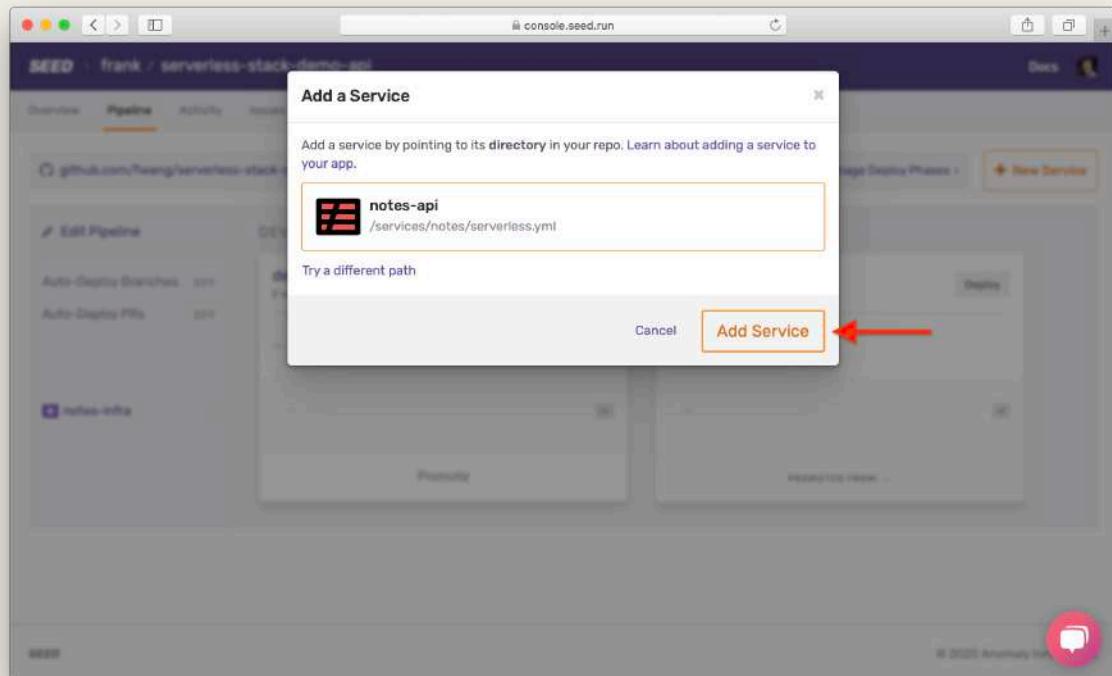
Add new service screenshot

Enter the path to the notes service services/notes. Then hit **Search**.



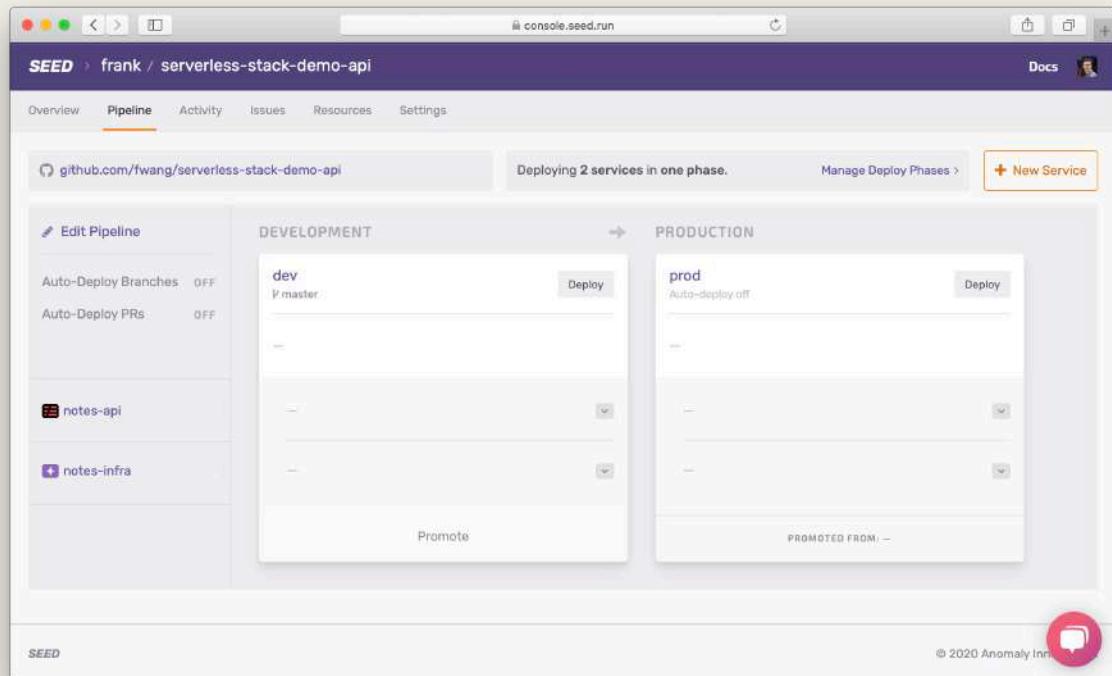
Search new service screenshot

Seed will search for the `serverless.yml` file in the path, to ensure you entered the right path. Hit **Add Service**.



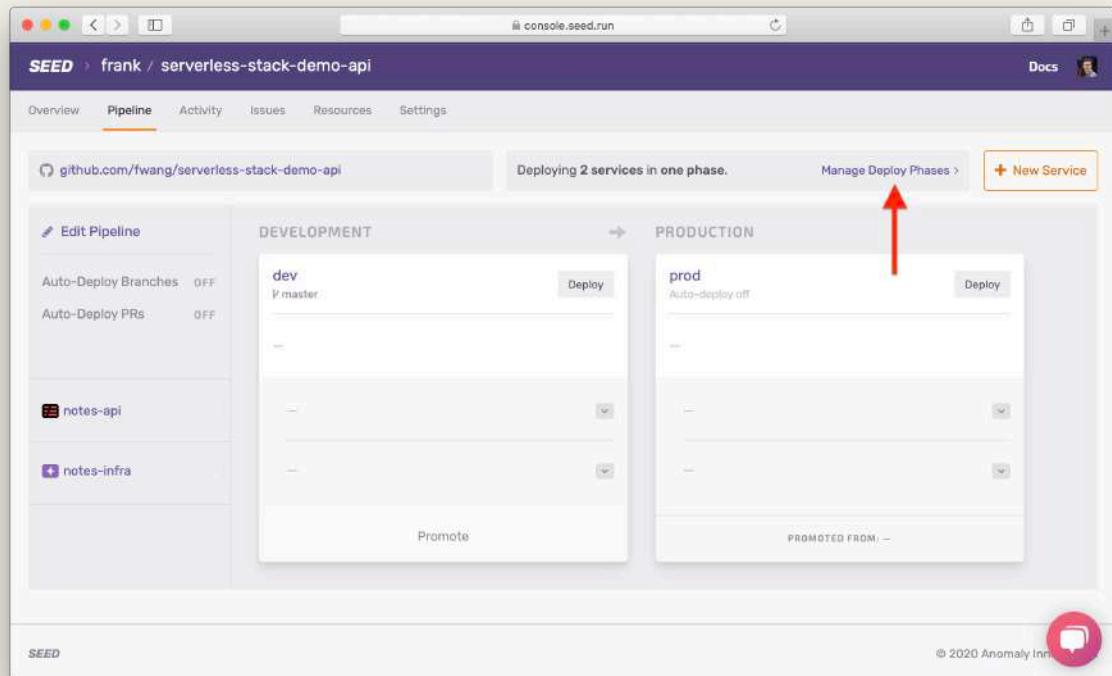
Serverless.yml detected screenshot

Now you have 2 services.



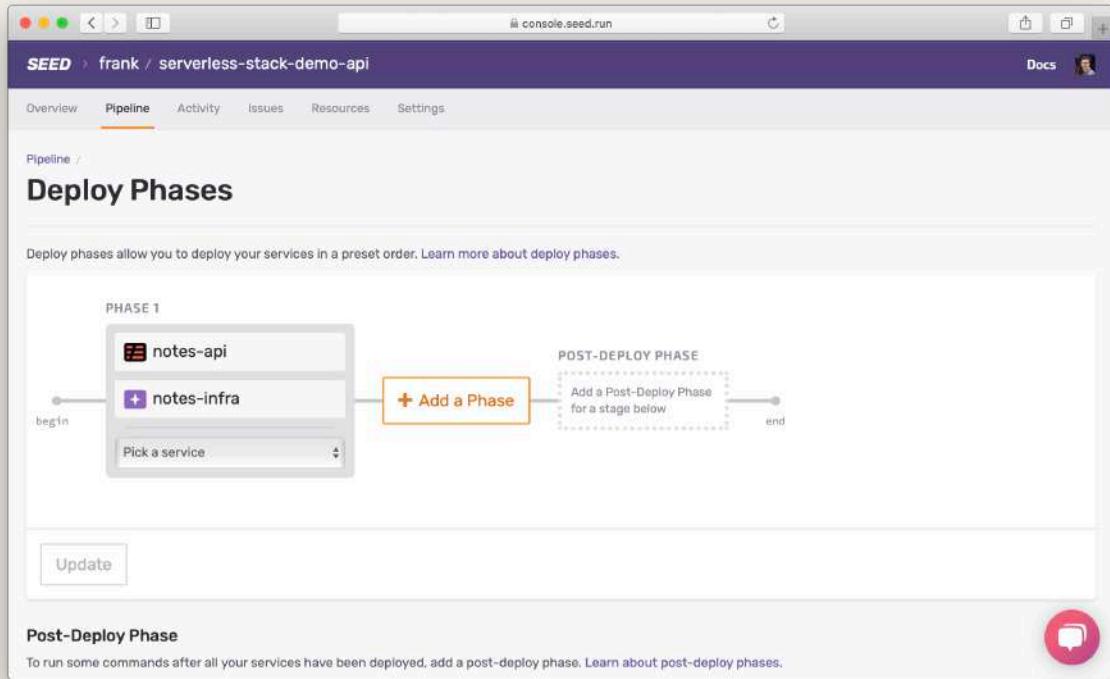
Added notes service screenshot

Before we deploy, let's make sure the services will be deployed in the desired order. To do this click on **Manage Deploy Phases**.



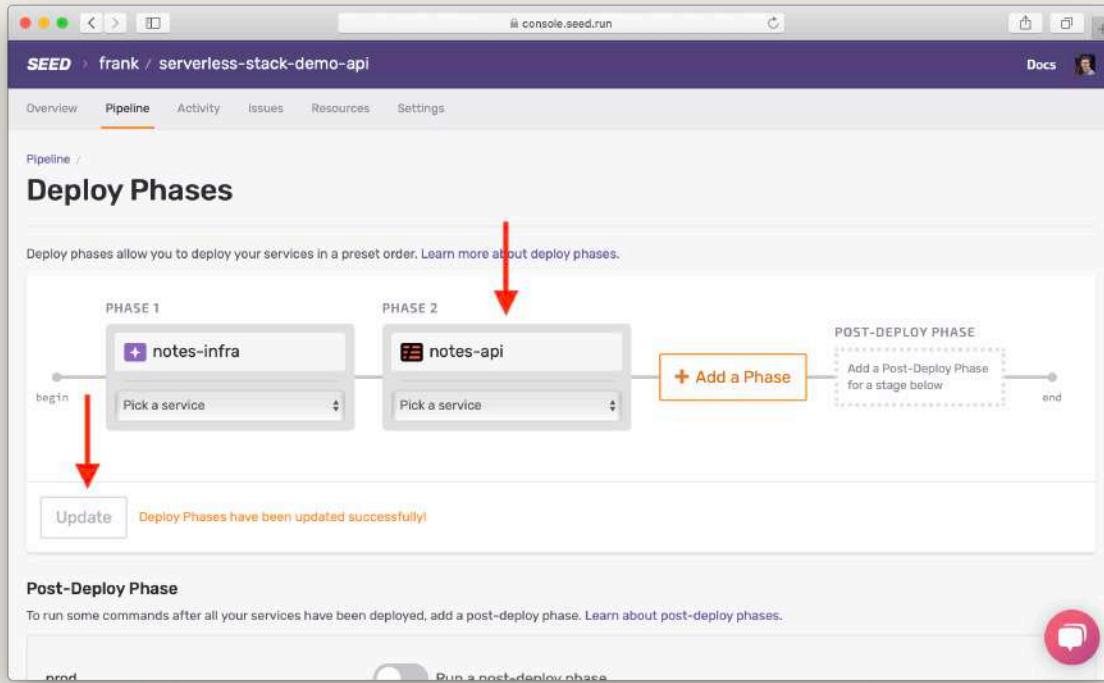
Manage Deploy Phases screenshot

Here you'll notice that by default all the services are deployed concurrently.



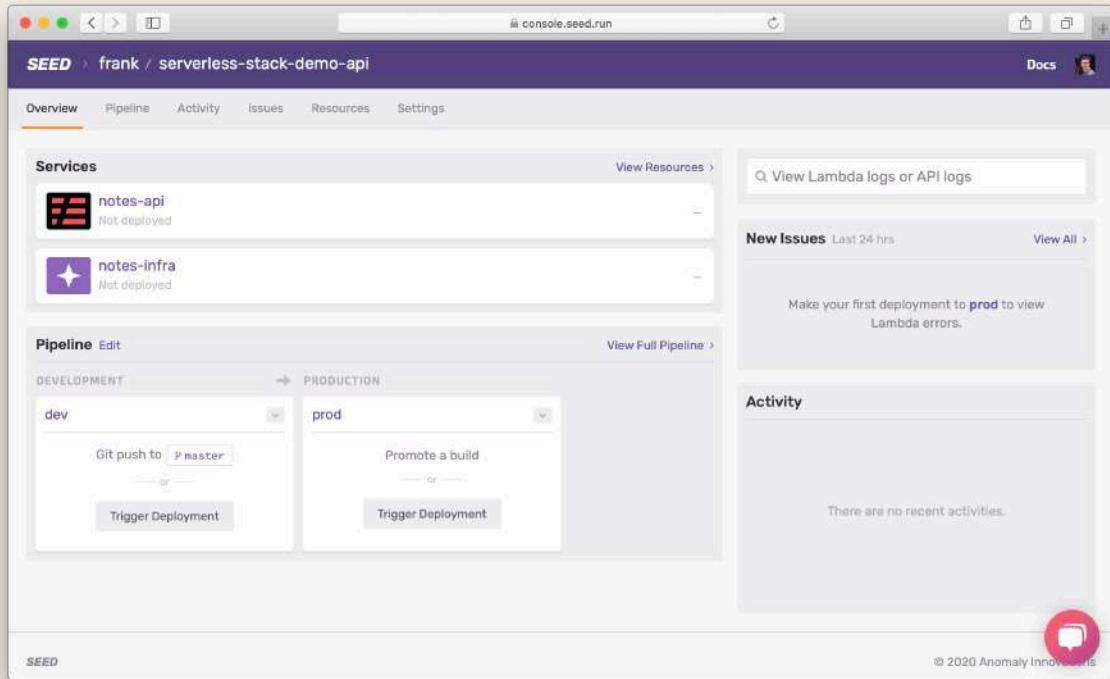
One deploy phase screenshot

Select **Add a phase** and move the API service to **Phase 2**. And hit Update Phases.



Multiple deploy phases screenshot

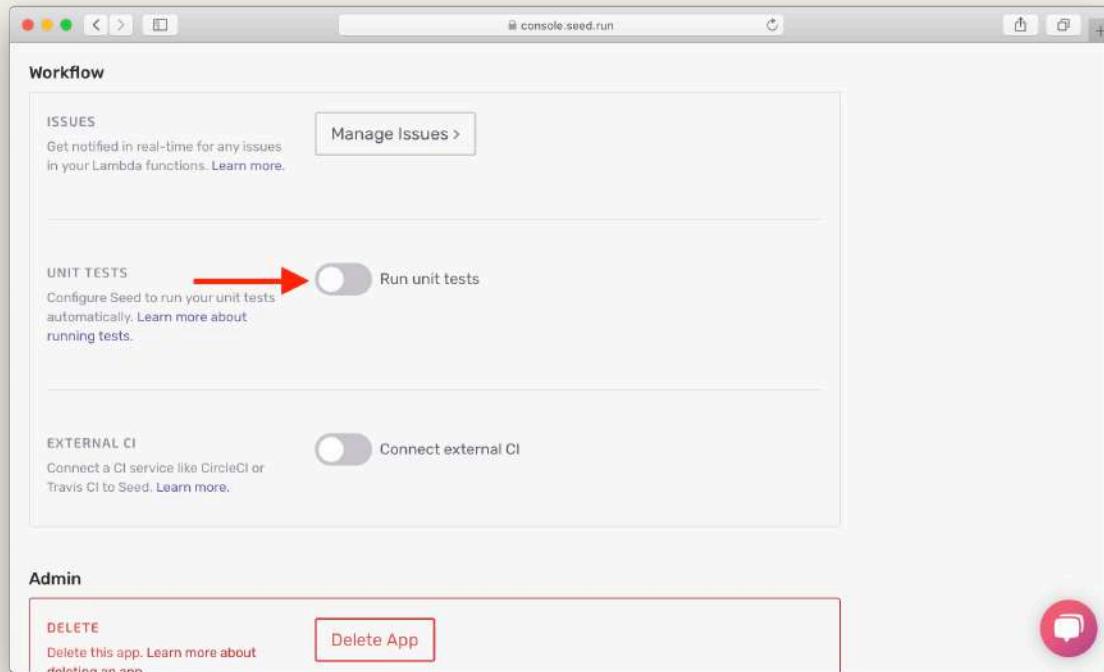
Now, your new app is ready to go!



[View new Seed app screenshot](#)

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](#) chapter. And we want to run those before we deploy our app.

To do this, hit the **Settings** link and click **Enable Unit Tests**.



Click Enable Unit Tests in Seed screenshot

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.

However, before we do that, we'll need to add our secret environment variables.



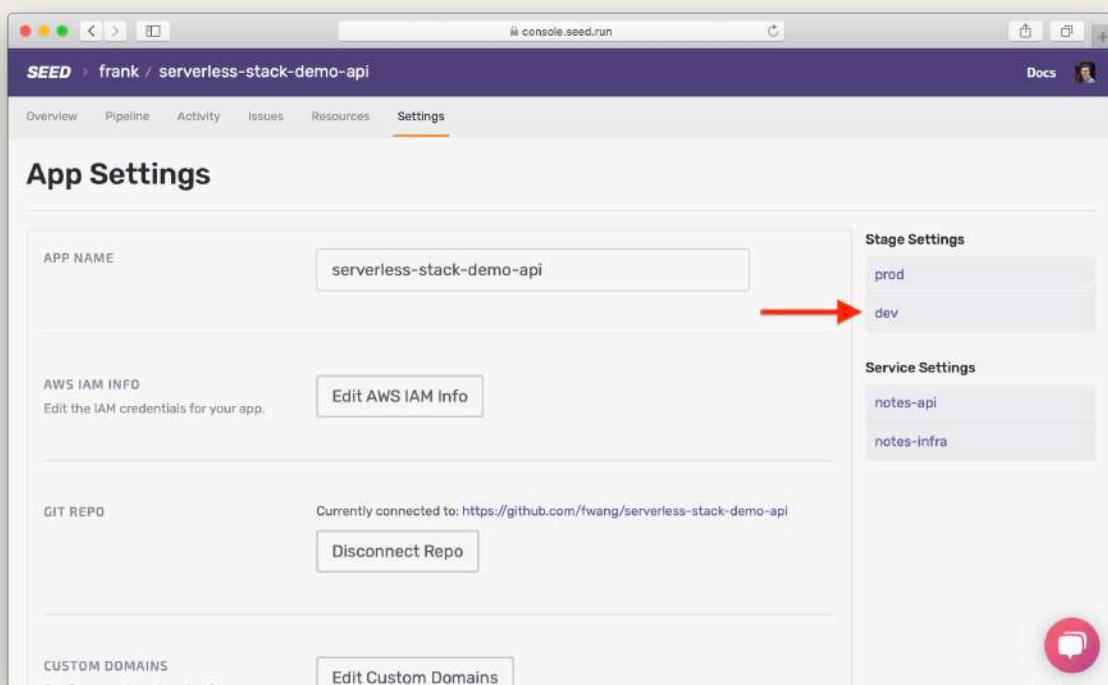
Help and discussion

View the [comments for this chapter on our forums](#)

Configure Secrets in Seed

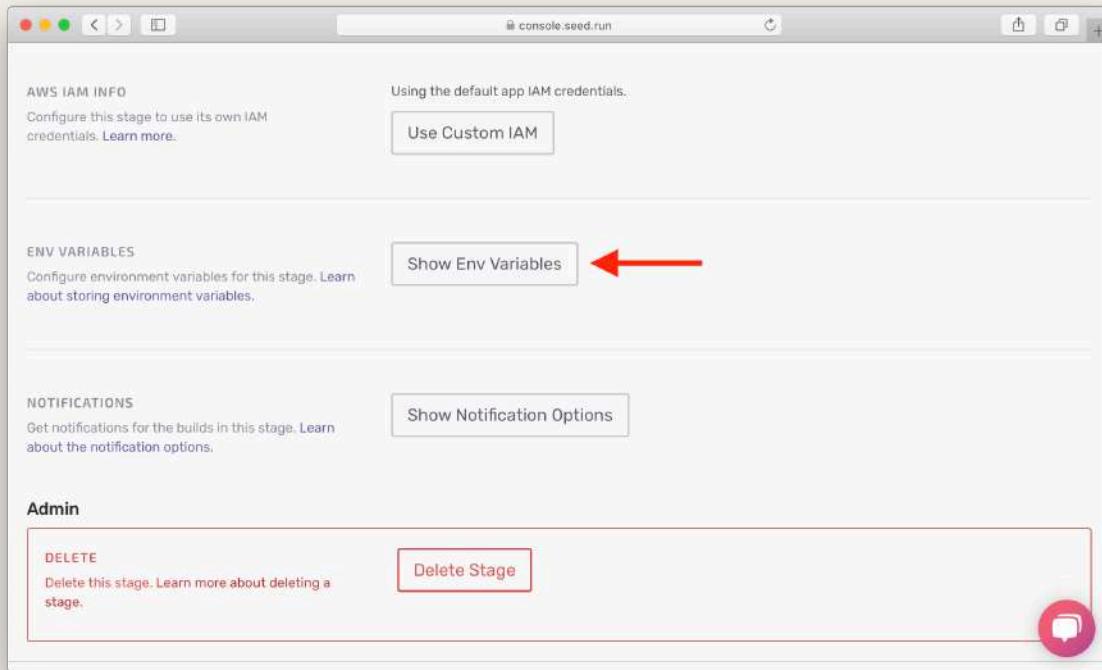
Before we can do our first deployment, we need to make sure to configure our secret environment variables. If you'll recall, we have explicitly not stored these in our code (or in Git). This means that if somebody else on our team needs to deploy, we'll need to pass the .env file around. Instead we'll configure [Seed](#) to deploy with our secrets for us.

To do that, hit **dev** in your app **Settings**.



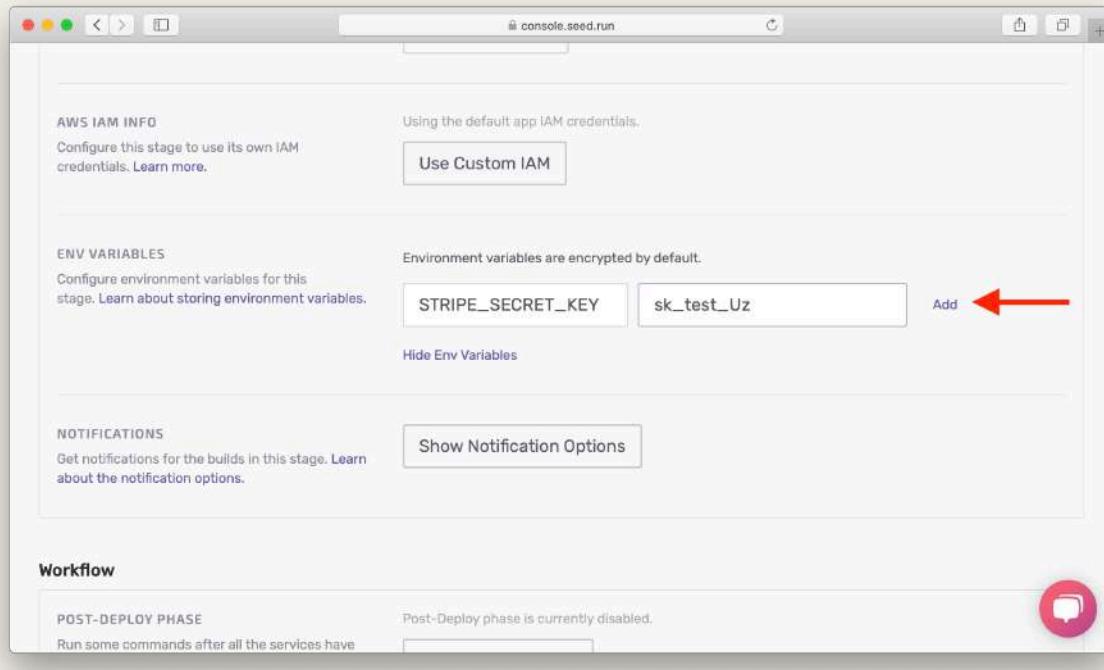
Select dev stage in Settings screenshot

Here click **Show Env Variables**.



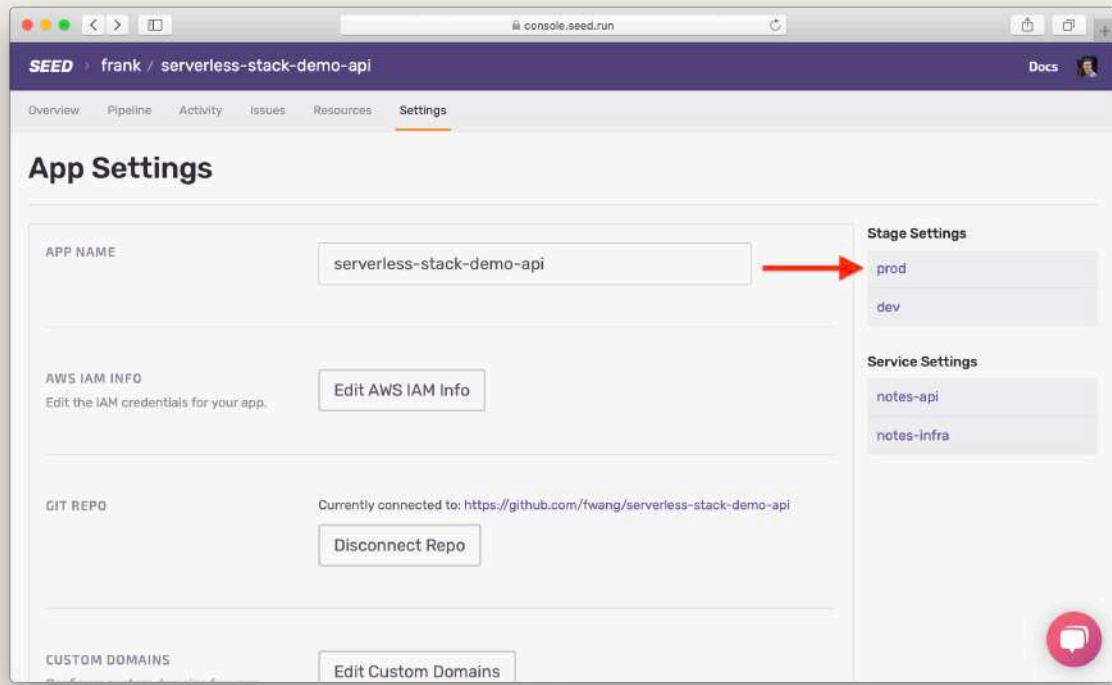
Show dev env variables settings screenshot

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](#) chapter. Hit **Add** to save your secret key.



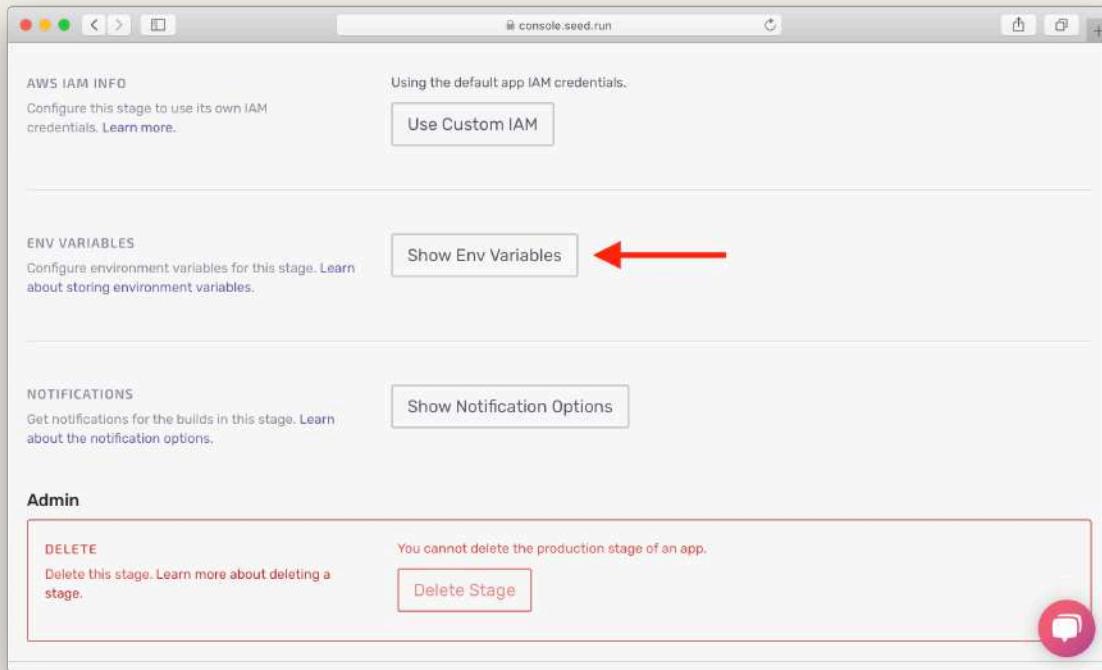
Add secret dev environment variable screenshot

Next we need to configure our secrets for the **prod** stage. Head over to the **prod** stage in your app **Settings**.



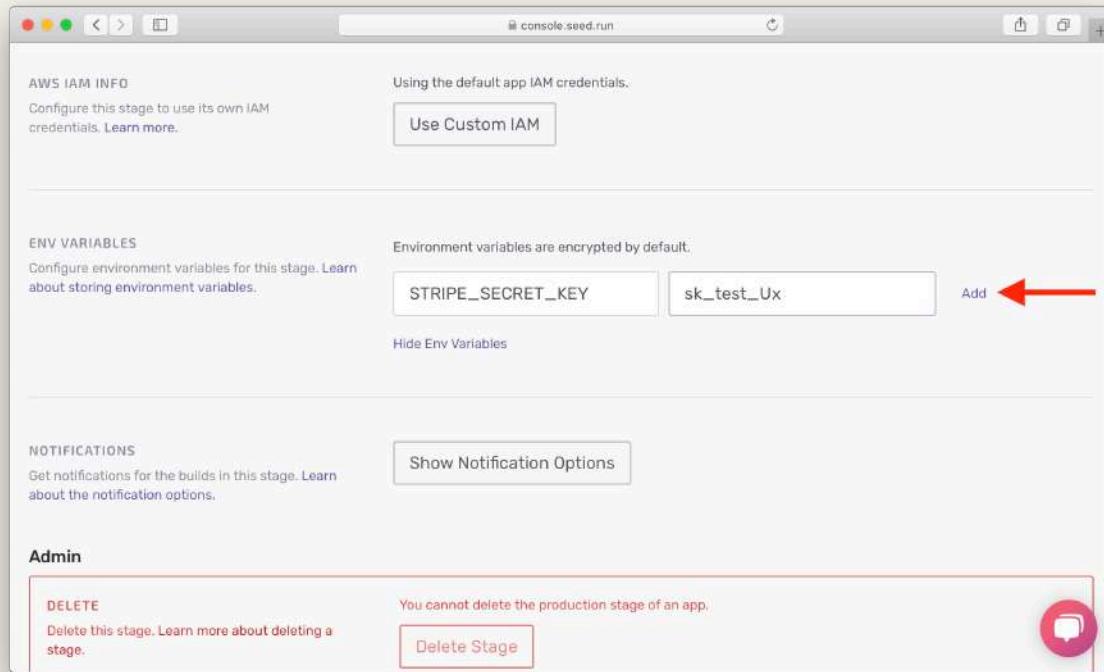
Select prod stage in Settings screenshot

Click **Show Env Variables**.



Show prod env variables settings screenshot

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](#) chapter. Hit **Add** to save your secret key.



Add secret prod environment variable screenshot

Next, we'll trigger our first deployment on Seed.



Help and discussion

View the [comments for this chapter on our forums](#)

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.

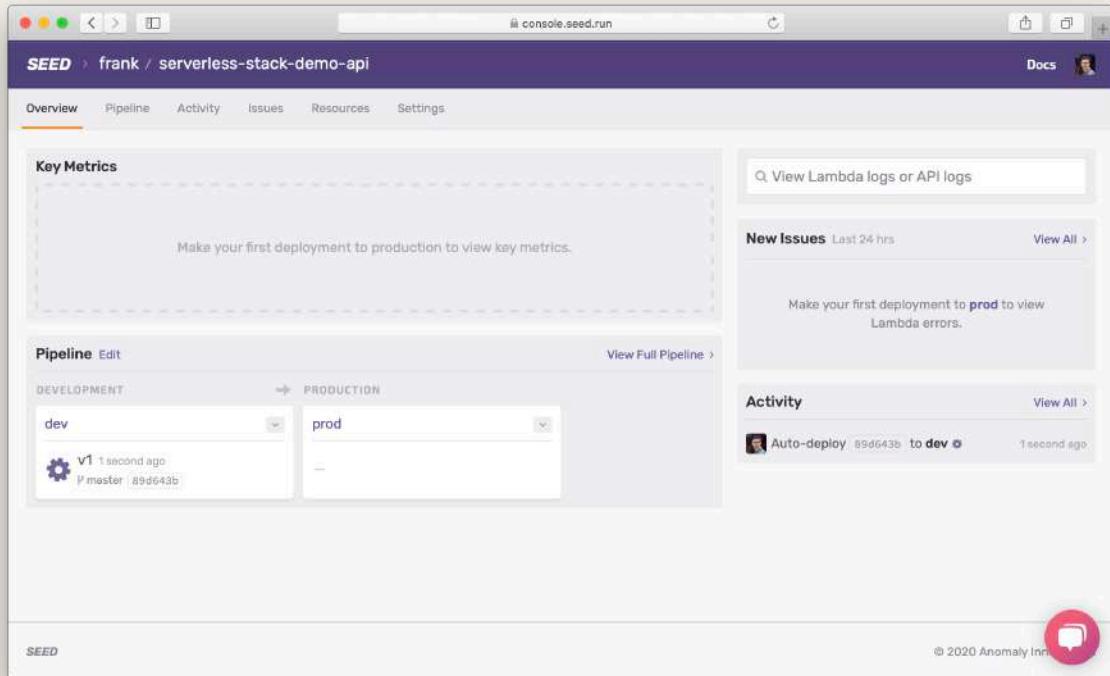
```
$ npm version patch
```

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.

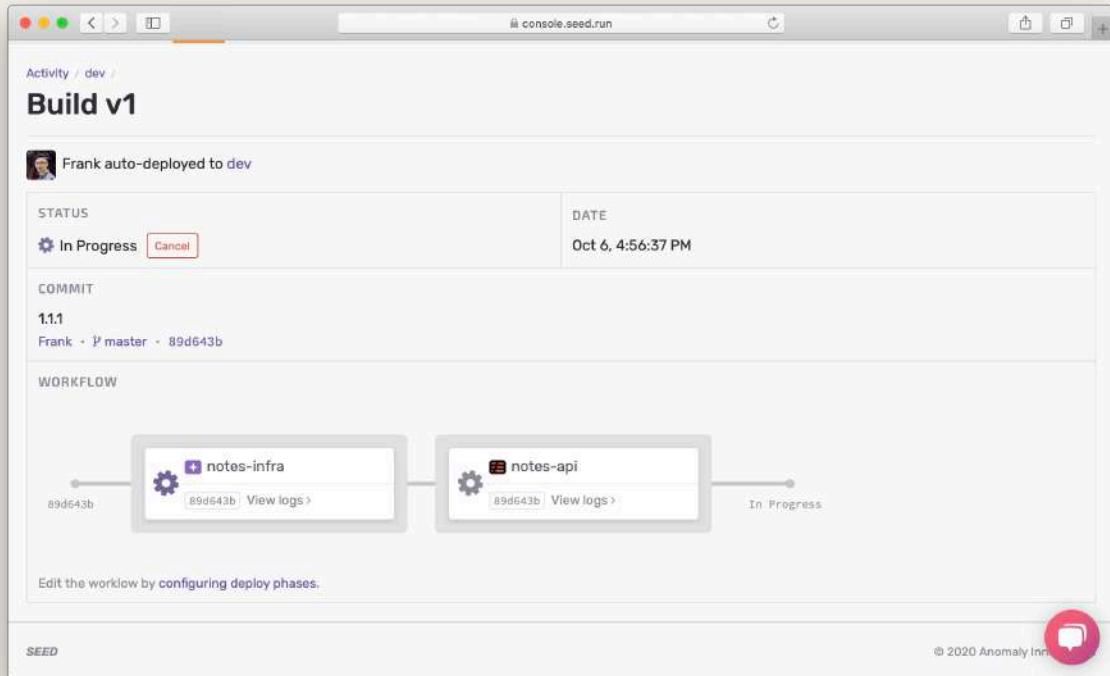
```
$ git push
```

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**.



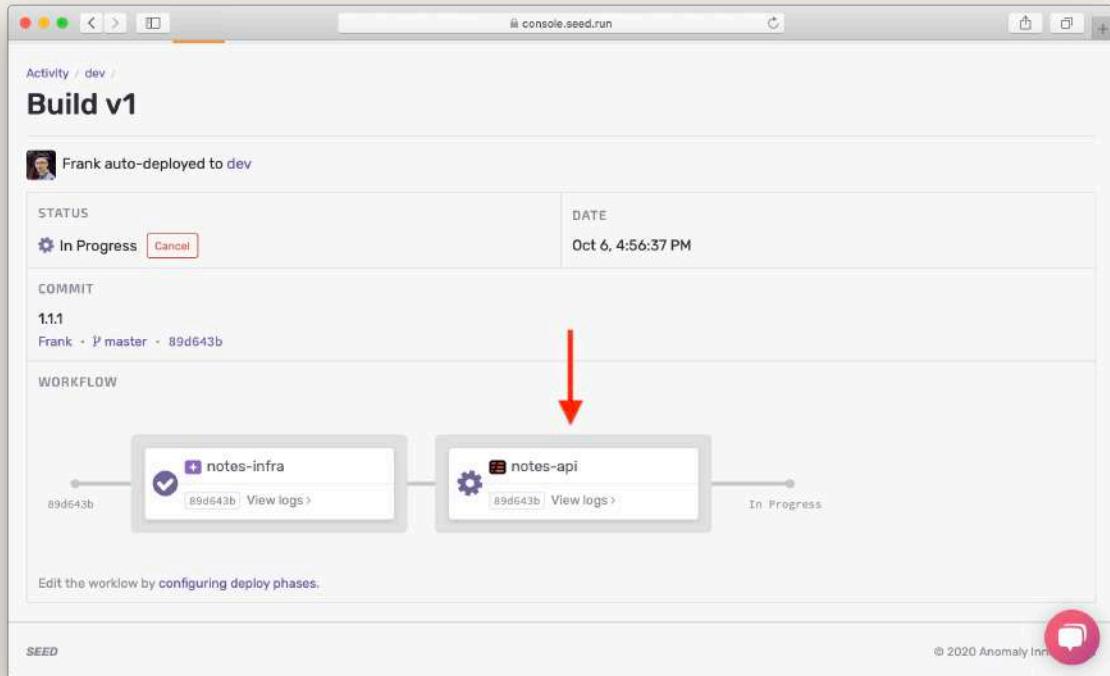
Seed dev build in progress screenshot

Here you'll see the build taking place live. Note that the deployments are carried out in the order specified by the deploy phases.



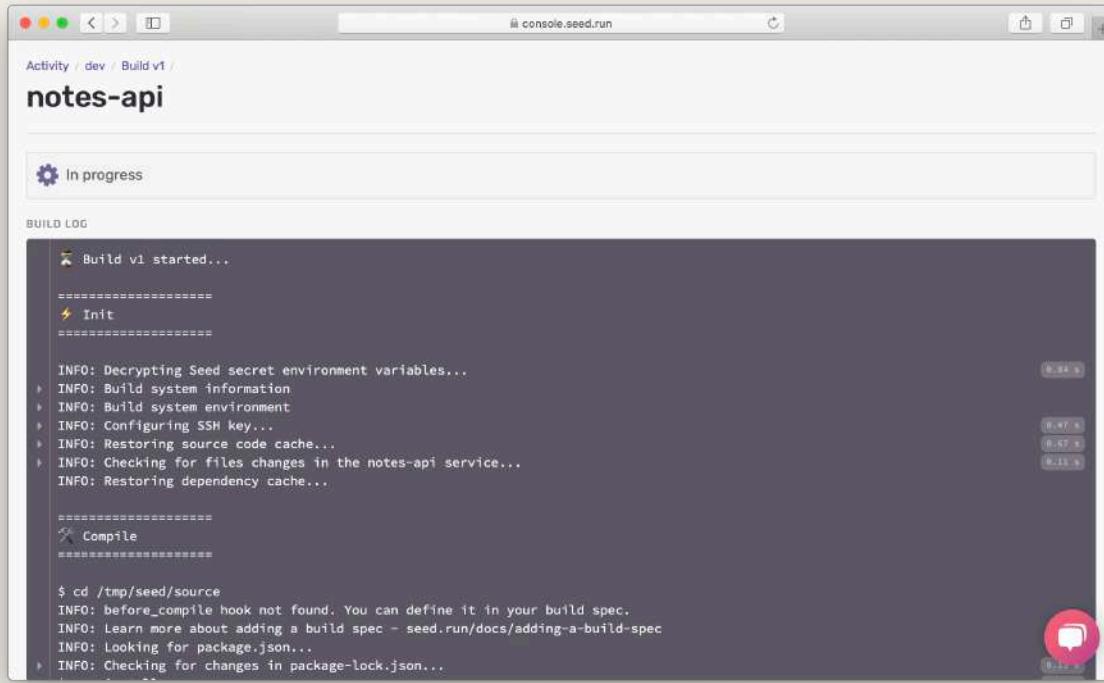
Dev build page phase 1 in progress screenshot

The **notes-api** service will start deploying after the **notes-infra** service has succeeded. Click on the **notes-api** service that is being deployed.



Dev build page phase 2 in progress screenshot

You'll see the build logs for the in progress build here.



Dev build logs in progress screenshot

Notice the tests are being run as a part of the build.

```
INFO: Learn more about adding a build spec - seed.run/docs/adding-a-build-spec
INFO: Looking for package.json...
> $ npm install

=====
Unit Test
=====
$ cd /tmp/seed/source
INFO: Looking for tests...
< $ npm test

> serverless-nodejs-starter@1.1.0 test /tmp/seed/source
> serverless-bundle test

Browserslist: caniuse-lite is outdated. Please run next command 'npm update'
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:        1.552s
Ran all test suites.

=====
Build
=====
$ cd /tmp/seed/source
INFO: before_build hook not found. You can define it in your build spec.
INFO: Learn more about adding a build spec - seed.run/docs/adding-a-build-spec
> $ SLS_DEBUG=* serverless package --stage dev --package s1s-package-output
> INFO: Setting Seed secret environment variables as Lambda environment variables
```

Dev build run tests screenshot

Once the build is complete, take a look at the build log from the **notes-infra** service and make a note of the following:

- Cognito User Pool Id: `UserPoolId`
- Cognito App Client Id: `UserPoolClientId`
- Cognito Identity Pool Id: `IdentityPoolId`
- S3 File Uploads Bucket: `AttachmentsBucketName`

We'll be needing these later in our frontend and when we test our APIs.

```
INFO: Deploying 3 stacks...
+ ✓ dev-notes-infra-dynamodb: Deployed
+ ✓ dev-notes-infra-s3: Deployed
+ ✓ dev-notes-infra-cognito: Deployed

INFO: Stacks output:
+ Stack dev-notes-infra-dynamodb
+ Stack dev-notes-infra-s3
  Status: Deployed
  Outputs:
    AttachmentsBucketName: dev-notes-infra-s3-uploads4f6eb0fd-1w2cue151v4z8
    ExportsOutputFnGetAttUploads4F6EB0FDArn5513CBEA: arn:aws:s3:::dev-notes-infra-s3-uploads4f6eb0fd-1w2cue151v4z8
  Exports:
    dev-notes-infra-s3:ExportsOutputFnGetAttUploads4F6EB0FDArn5513CBEA: arn:aws:s3:::dev-notes-infra-s3-uploads4f6eb0fd-1w2cue151v4z8
  0

+ Stack dev-notes-infra-cognito
  Status: Deployed
  Outputs:
    AuthenticatedRoleName: dev-notes-infra-cognito-CognitoAuthRoleCognitoDefa-187RURA3Q3SW3
    UserPoolClientId: 5bt4nfuqbqu067enlj2aljdfjj
    UserPoolId: us-east-1_9xnomdhwb
    IdentityPoolId: us-east-1:26503ac6-6bla-4b15-978b-4feceb9d484c
  Exports:
    dev-notes-infra-CognitoAuthRole: dev-notes-infra-cognito-CognitoAuthRoleCognitoDefa-187RURA3Q3SW3

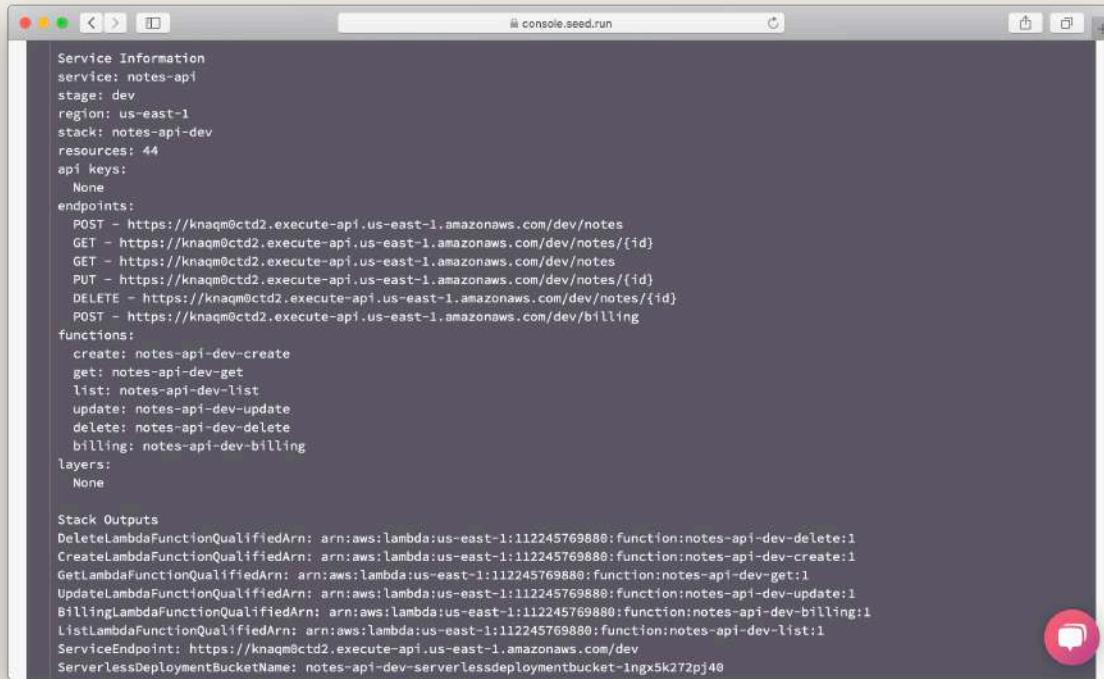
INFO: Build deployed in 2m 13s. No build minutes used while SST is in beta.

=====
Artifact
=====
```

Dev build infrastructure output screenshot

Then, take a look at the build log from the **notes-api** service and make a note of the following:

- Region: `region`
- API Gateway URL: `ServiceEndpoint`



The screenshot shows a browser window titled "console.seed.run". The content is a JSON-like output of a serverless stack. It includes sections for "Service Information" and "Stack Outputs".

```
Service Information
service: notes-api
stage: dev
region: us-east-1
stack: notes-api-dev
resources: 44
api keys:
  None
endpoints:
  POST - https://knaqm0ctd2.execute-api.us-east-1.amazonaws.com/dev/notes
  GET - https://knaqm0ctd2.execute-api.us-east-1.amazonaws.com/dev/notes/{id}
  GET - https://knaqm0ctd2.execute-api.us-east-1.amazonaws.com/dev/notes
  PUT - https://knaqm0ctd2.execute-api.us-east-1.amazonaws.com/dev/notes/{id}
  DELETE - https://knaqm0ctd2.execute-api.us-east-1.amazonaws.com/dev/notes/{id}
  POST - https://knaqm0ctd2.execute-api.us-east-1.amazonaws.com/dev/billing
functions:
  create: notes-api-dev-create
  get: notes-api-dev-get
  list: notes-api-dev-list
  update: notes-api-dev-update
  delete: notes-api-dev-delete
  billing: notes-api-dev-billing
layers:
  None

Stack Outputs
DeleteLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-dev-delete:1
CreateLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-dev-create:1
GetLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-dev-get:1
UpdateLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-dev-update:1
BillingLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-dev-billing:1
ListLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-dev-list:1
ServiceEndpoint: https://knaqm0ctd2.execute-api.us-east-1.amazonaws.com/dev
ServerlessDeploymentBucketName: notes-api-dev-serverlessdeploymentbucket-1ngx5k272pj40
```

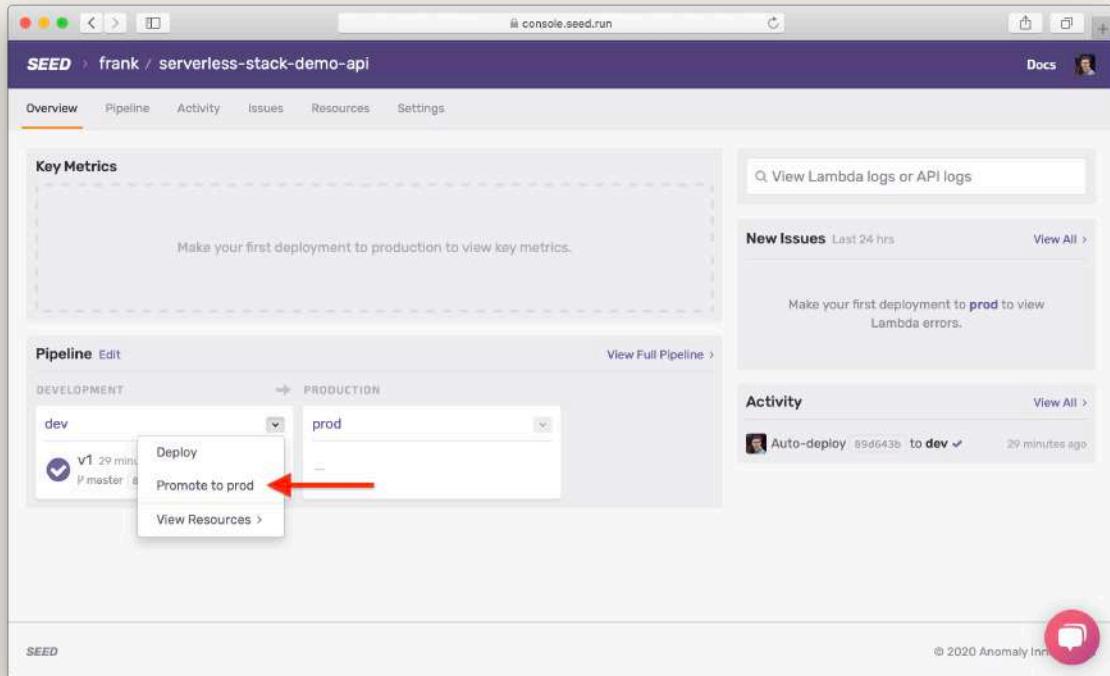
Dev build api stack output screenshot

If you don't see the above info, expand the deploy step in the build log.

Now head over to the app home page. You'll notice that we are ready to promote to production.

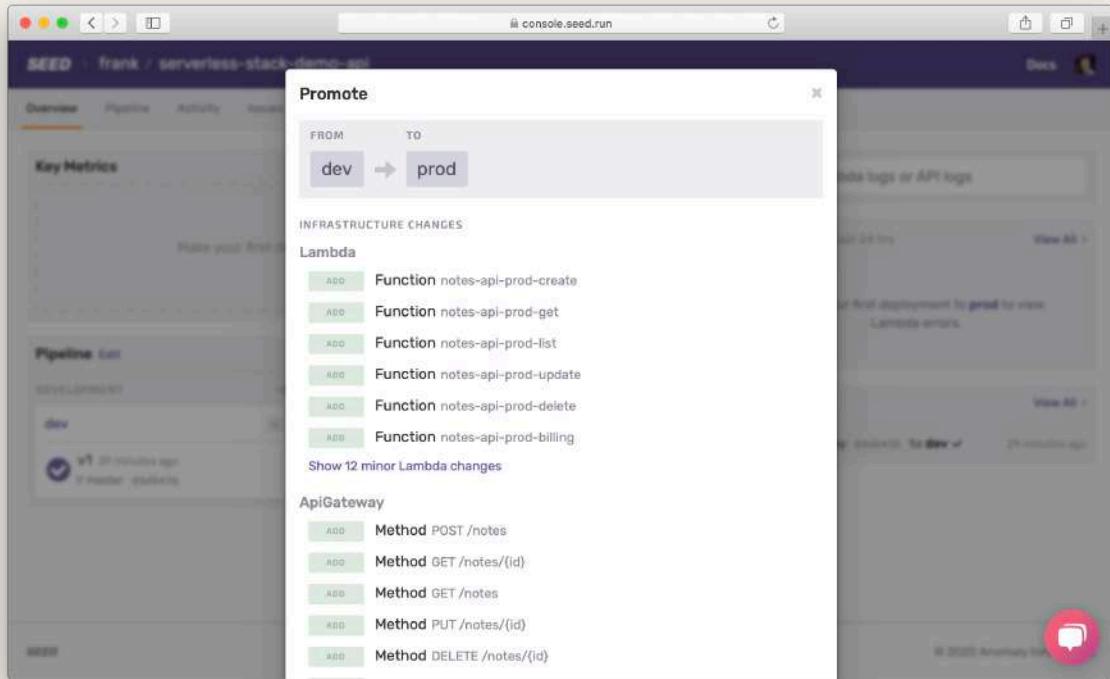
We have a manual promotion step so that you get a chance to review the changes and ensure that you are ready to push to production.

Hit the **Promote** button.



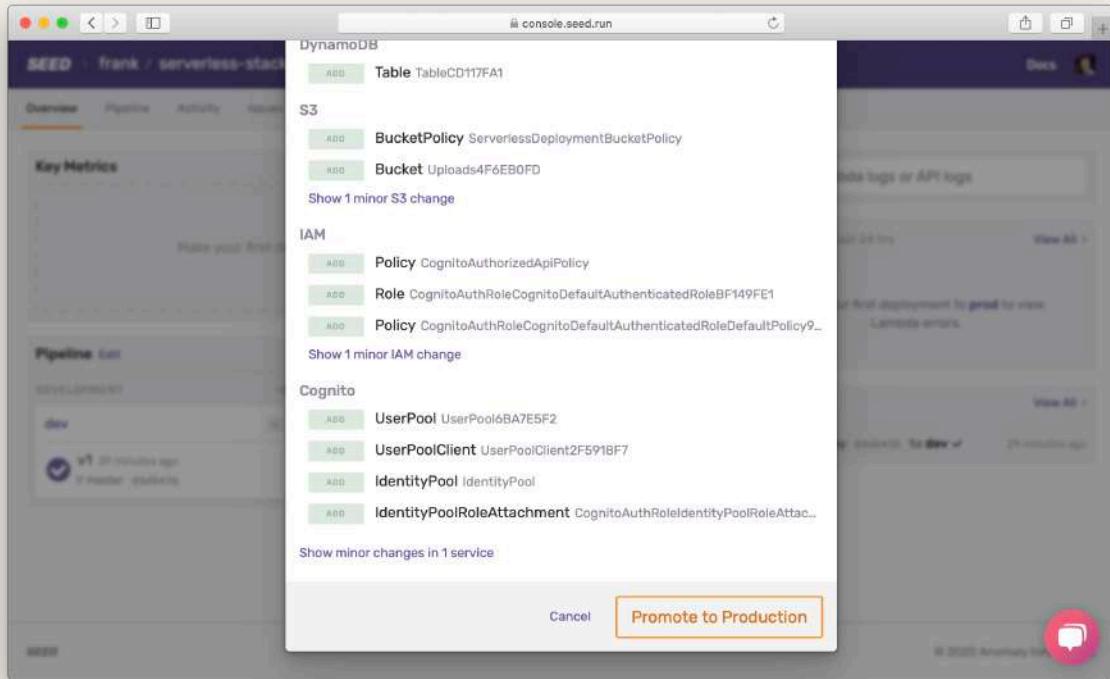
Dev build ready to promote screenshot

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.



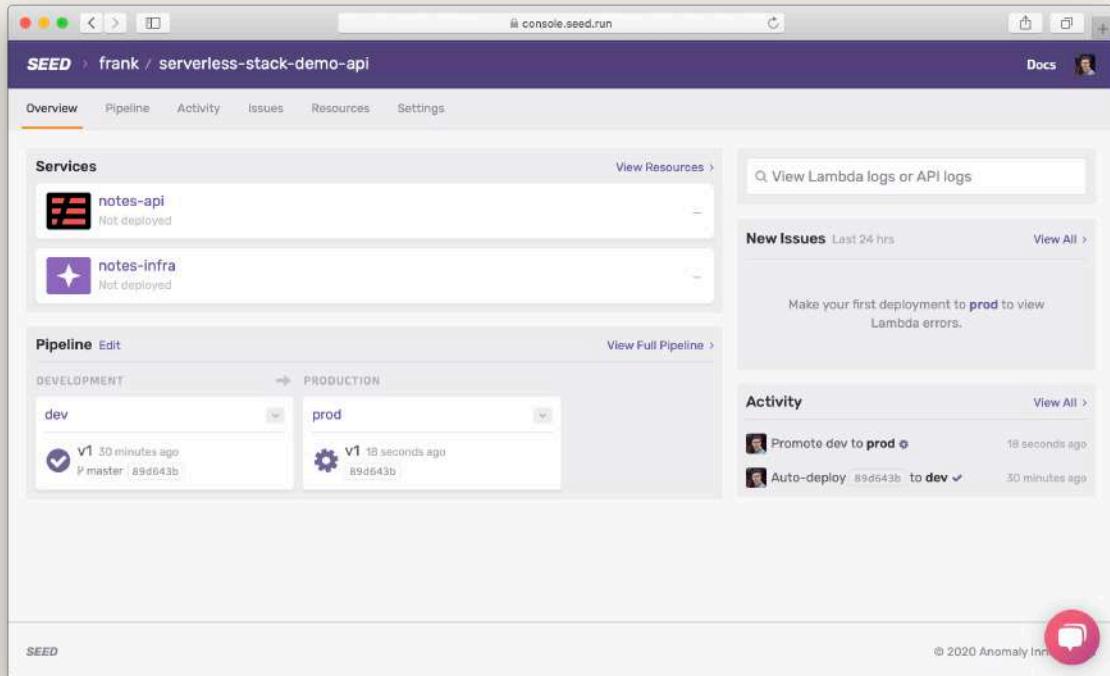
Review promote change set screenshot

Scroll down and hit **Promote to Production**.



Confirm promote dev build screenshot

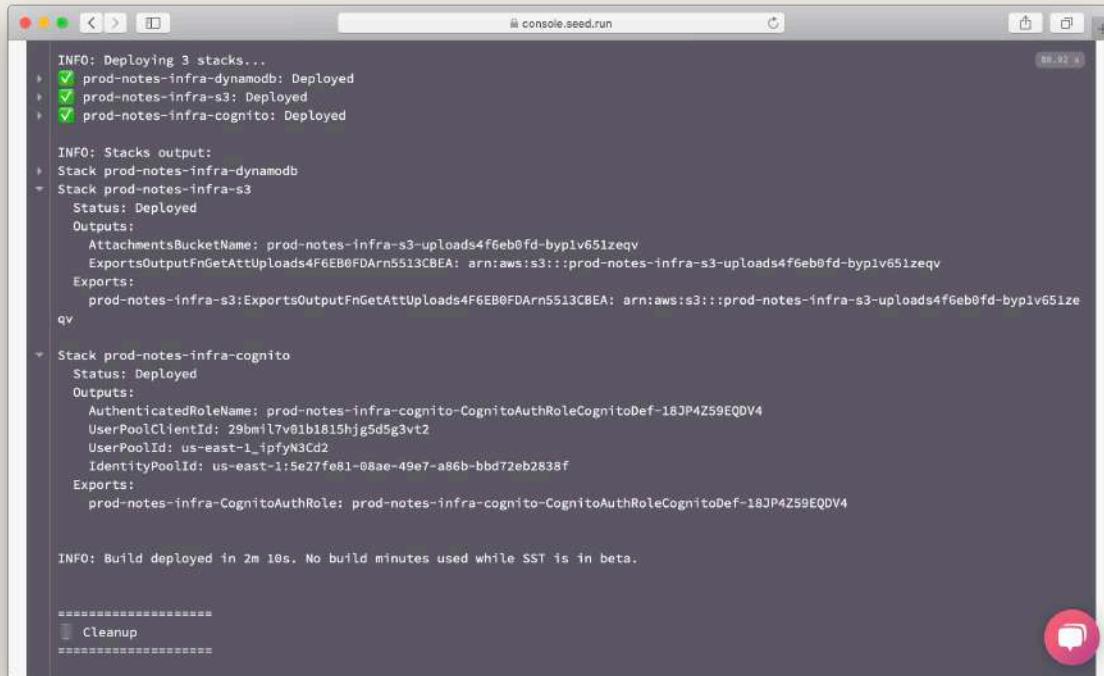
You'll notice that the build is being promoted to the **prod** stage.



prod build in progress screenshot

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 from the **notes-infra** service.

- Cognito User Pool Id: `UserPoolId`
- Cognito App Client Id: `UserPoolClientId`
- Cognito Identity Pool Id: `IdentityPoolId`
- S3 File Uploads Bucket: `AttachmentsBucketName`



```
INFO: Deploying 3 stacks...
❯ ✓ prod-notes-infra-dynamodb: Deployed
❯ ✓ prod-notes-infra-s3: Deployed
❯ ✓ prod-notes-infra-cognito: Deployed

INFO: Stacks output:
❯ Stack prod-notes-infra-dynamodb
❯ Stack prod-notes-infra-s3
  Status: Deployed
  Outputs:
    AttachmentsBucketName: prod-notes-infra-s3-uploads4f6eb0fd-byp1v65izeqv
    ExportsOutputFnGetAttUploads4F6EB0FDArn5513CBEA: arn:aws:s3:::prod-notes-infra-s3-uploads4f6eb0fd-byp1v65izeqv
  Exports:
    prod-notes-infra-s3:ExportsOutputFnGetAttUploads4F6EB0FDArn5513CBEA: arn:aws:s3:::prod-notes-infra-s3-uploads4f6eb0fd-byp1v65izeqv
  qv

❯ Stack prod-notes-infra-cognito
  Status: Deployed
  Outputs:
    AuthenticatedRoleName: prod-notes-infra-cognito-CognitoAuthRoleCognitoDef-18JP4Z59EQDV4
    UserPoolClientId: 29bmil7v81b1815hjg5d5g3vt2
    UserPoolId: us-east-1_ipfyN3cd2
    IdentityPoolId: us-east-1:5e27fe81-08ae-49e7-a86b-bbd72eb2838f
  Exports:
    prod-notes-infra-CognitoAuthRole: prod-notes-infra-cognito-CognitoAuthRoleCognitoDef-18JP4Z59EQDV4

INFO: Build deployed in 2m 10s. No build minutes used while SST is in beta.

=====
  Cleanup
=====
```

Prod build infrastructure output screenshot

And make a note of the following from the **notes-api** service.

- Region: `region`
- API Gateway URL: `ServiceEndpoint`

The screenshot shows a browser window titled "console.seed.run". The content displays service information for a "notes-api" in "prod" stage, located in "us-east-1" region, with 44 resources. It lists endpoints for various HTTP methods (POST, GET, PUT, DELETE) and functions (create, get, list, update, delete, billing). The "Stack Outputs" section provides ARNs for Lambda functions like DeleteLambdaFunctionQualifiedArn, CreateLambdaFunctionQualifiedArn, GetLambdaFunctionQualifiedArn, UpdateLambdaFunctionQualifiedArn, BillingLambdaFunctionQualifiedArn, ListLambdaFunctionQualifiedArn, ServiceEndpoint, and ServerlessDeploymentBucketName.

```
Service Information
service: notes-api
stage: prod
region: us-east-1
stack: notes-api-prod
resources: 44
api keys:
None
endpoints:
POST - https://p4vi2bmkac.execute-api.us-east-1.amazonaws.com/prod/notes
GET - https://p4vi2bmkac.execute-api.us-east-1.amazonaws.com/prod/notes/{id}
GET - https://p4vi2bmkac.execute-api.us-east-1.amazonaws.com/prod/notes
PUT - https://p4vi2bmkac.execute-api.us-east-1.amazonaws.com/prod/notes/{id}
DELETE - https://p4vi2bmkac.execute-api.us-east-1.amazonaws.com/prod/notes/{id}
POST - https://p4vi2bmkac.execute-api.us-east-1.amazonaws.com/prod/billing
functions:
create: notes-api-prod-create
get: notes-api-prod-get
list: notes-api-prod-list
update: notes-api-prod-update
delete: notes-api-prod-delete
billing: notes-api-prod-billing
layers:
None

Stack Outputs
DeleteLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-prod-delete:1
CreateLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-prod-create:1
GetLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-prod-get:1
UpdateLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-prod-update:1
BillingLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-prod-billing:1
ListLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:112245769880:function:notes-api-prod-list:1
ServiceEndpoint: https://p4vi2bmkac.execute-api.us-east-1.amazonaws.com/prod
ServerlessdeploymentBucketName: notes-api-prod-serverlessdeploymentbucket-sfh50rncxd8s
```

Prod build api output screenshot

Next let's configure our serverless API with a custom domain.



Help and discussion

View the [comments](#) for this chapter on our forums

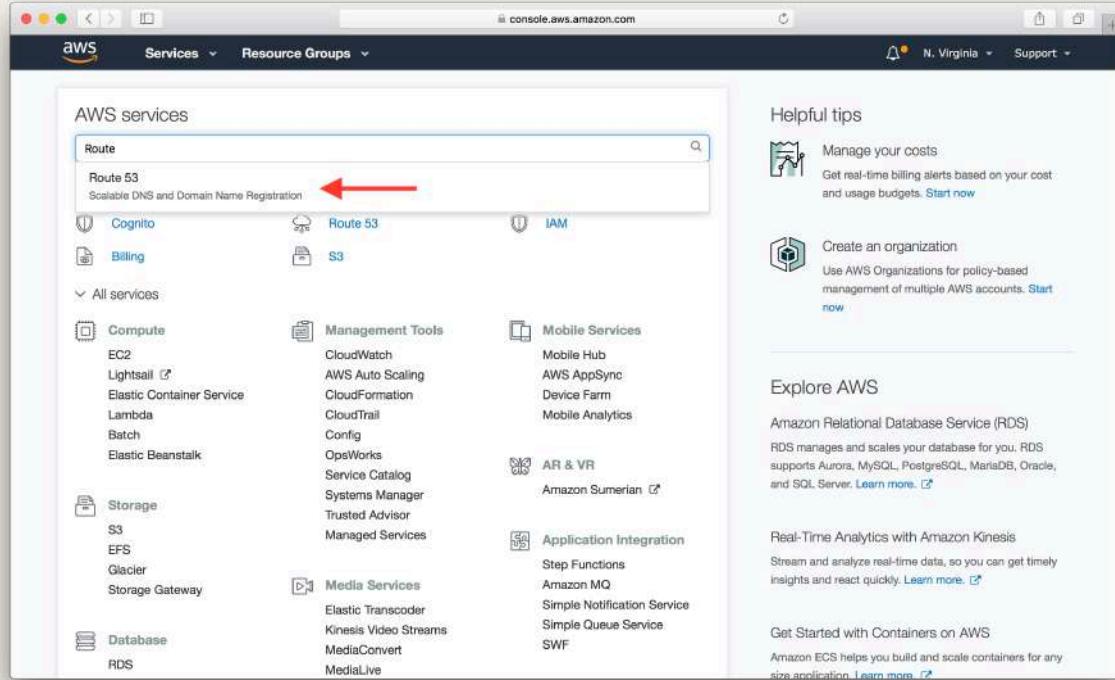
Set Custom Domains Through Seed

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](#).

To start with, we need to purchase a domain on [Amazon Route 53](#). If you have an existing domain not on AWS, follow these docs to [move it over to Route 53](#).

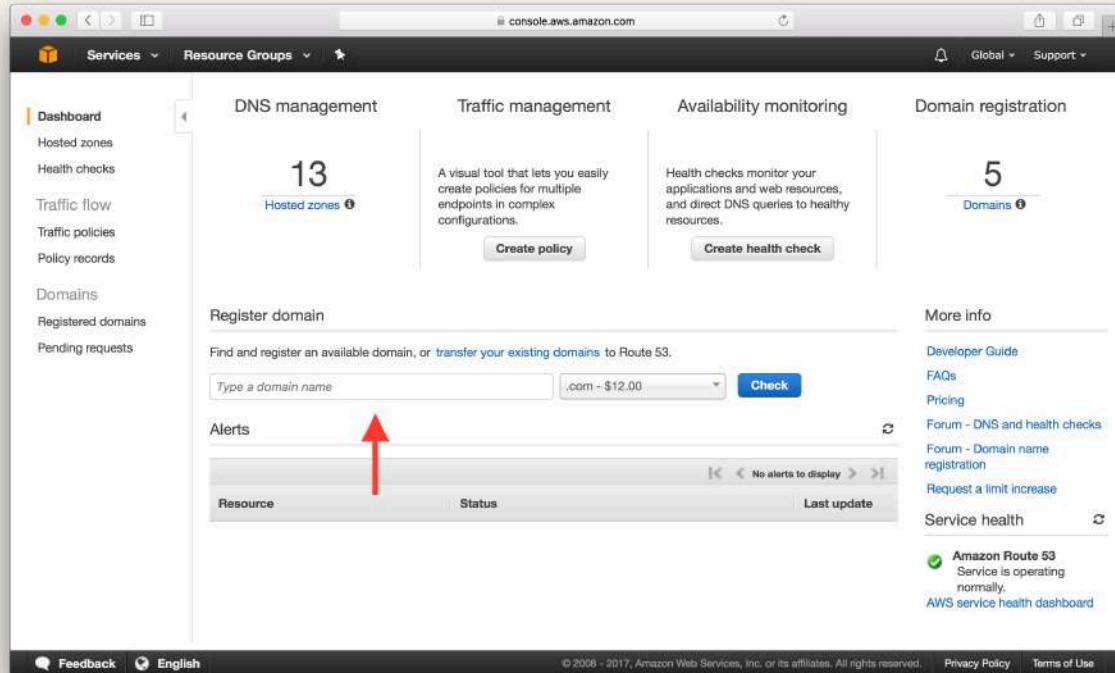
Purchase a Domain with Route 53

Head over to your [AWS Console](#) and select **Route 53** in the list of services.



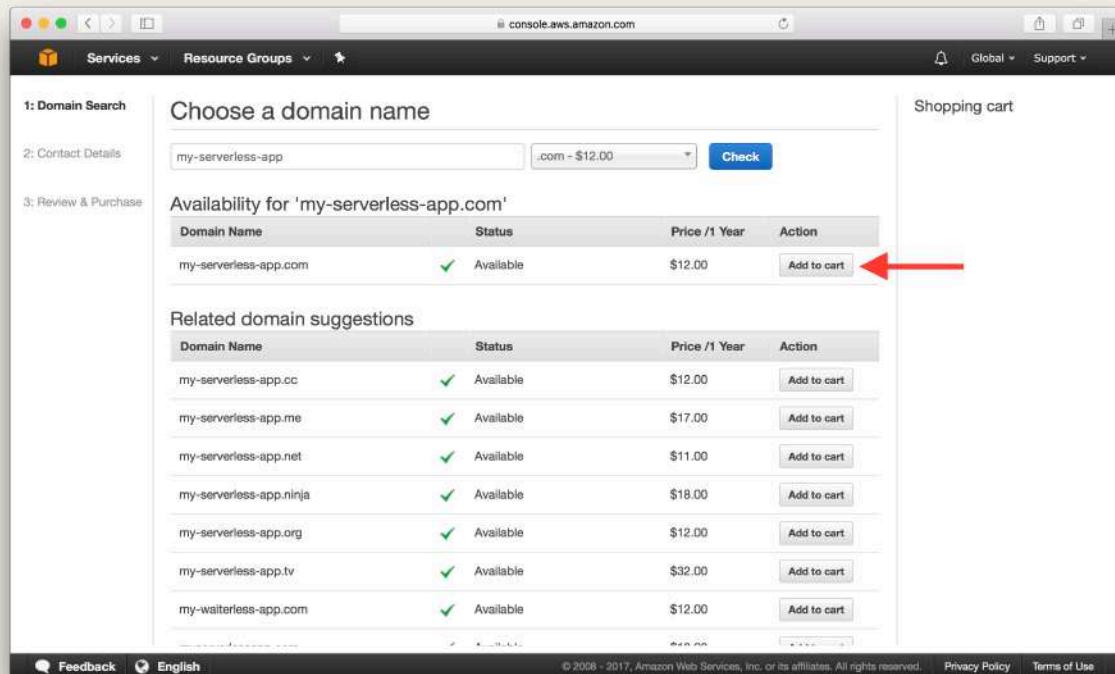
Select Route 53 service screenshot

Type in your domain in the **Register domain** section and click **Check**.



Search available domain screenshot

After checking its availability, click **Add to cart**.



Add domain to cart screenshot

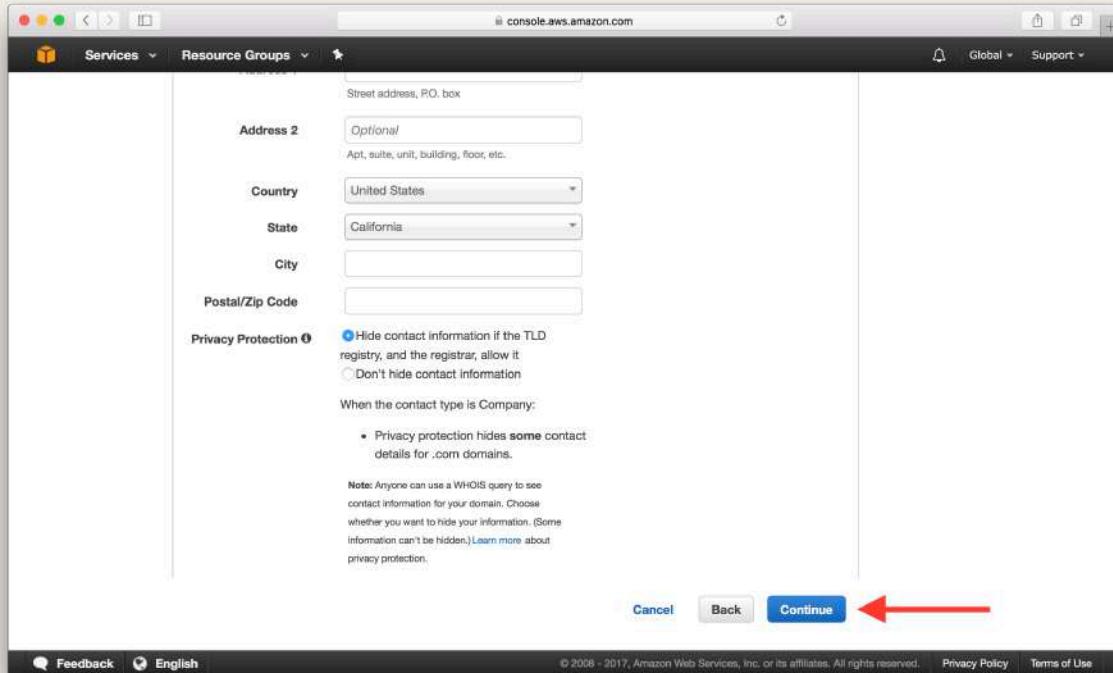
And hit **Continue** at the bottom of the page.

The screenshot shows a list of related domain suggestions for "my-serverless-app". The table includes columns for Domain Name, Status, Price / 1 Year, and Action. Most domains are listed as Available at \$12.00 per year. A red arrow points to the blue "Continue" button at the bottom right of the page.

Domain Name	Status	Price / 1 Year	Action
my-serverless-app.cc	Available	\$12.00	Add to cart
my-serverless-app.me	Available	\$17.00	Add to cart
my-serverless-app.net	Available	\$11.00	Add to cart
my-serverless-app.ninja	Available	\$18.00	Add to cart
my-serverless-app.org	Available	\$12.00	Add to cart
my-serverless-app.tv	Available	\$32.00	Add to cart
my-walterless-app.com	Available	\$12.00	Add to cart
myserverlessapp.com	Available	\$12.00	Add to cart
myserverlesshomepage.com	Available	\$12.00	Add to cart
myserverlessprogram.com	Available	\$12.00	Add to cart
savemyserverlessapp.com	Available	\$12.00	Add to cart
theserverlessapp.com	Available	\$12.00	Add to cart

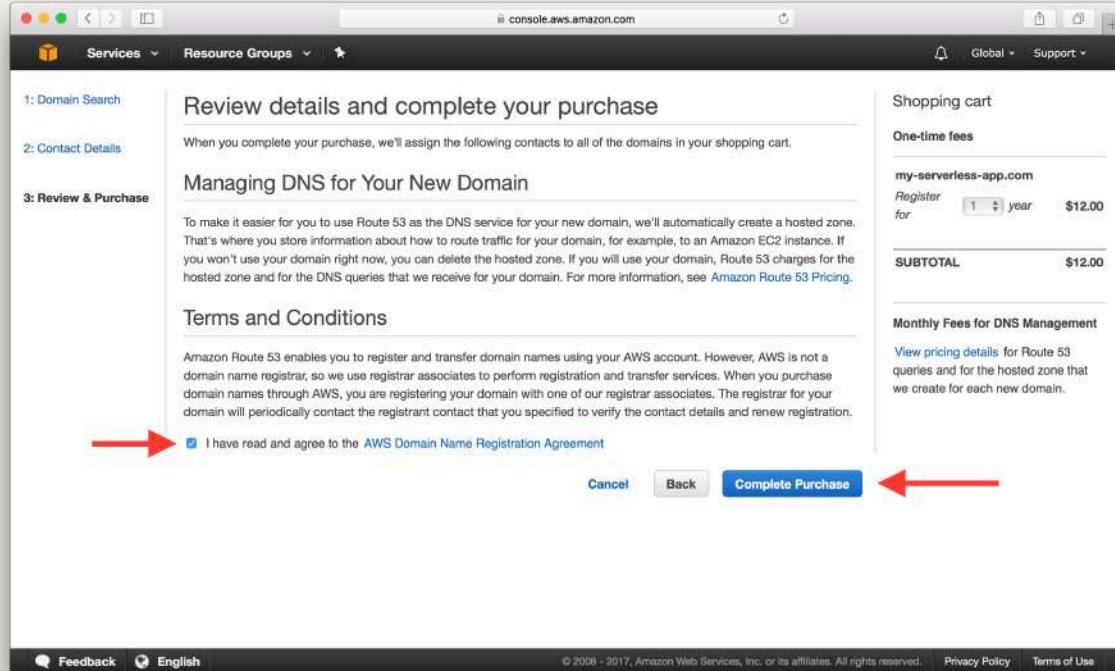
Continue to contact details screenshot

Fill in your contact details and hit **Continue** once again.



Continue to confirm details screenshot

Finally, review your details and confirm the purchase by hitting **Complete Purchase**.

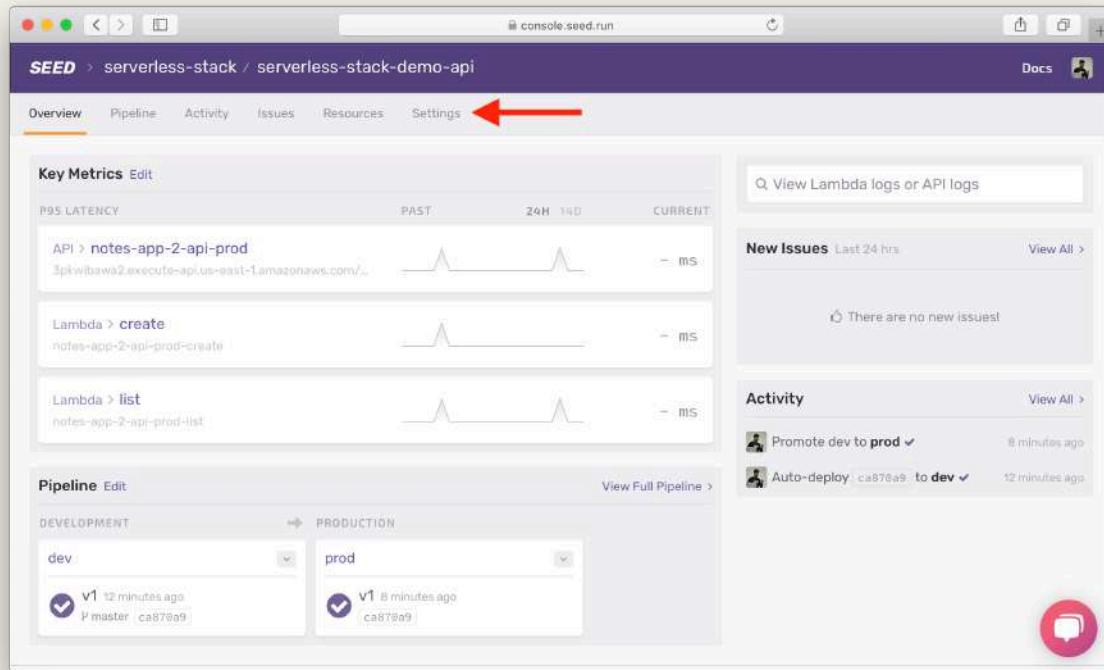


Confirm domain purchase screenshot

Next, let's use this custom domain for our app.

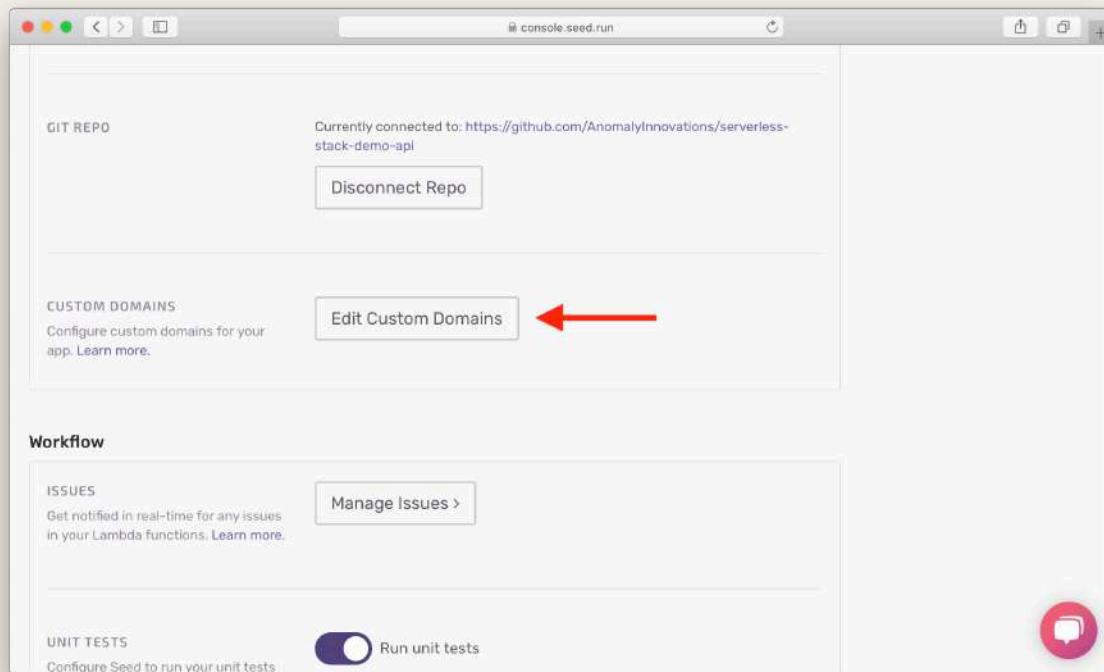
Add Custom Domain on Seed

Head over to our app settings in Seed.



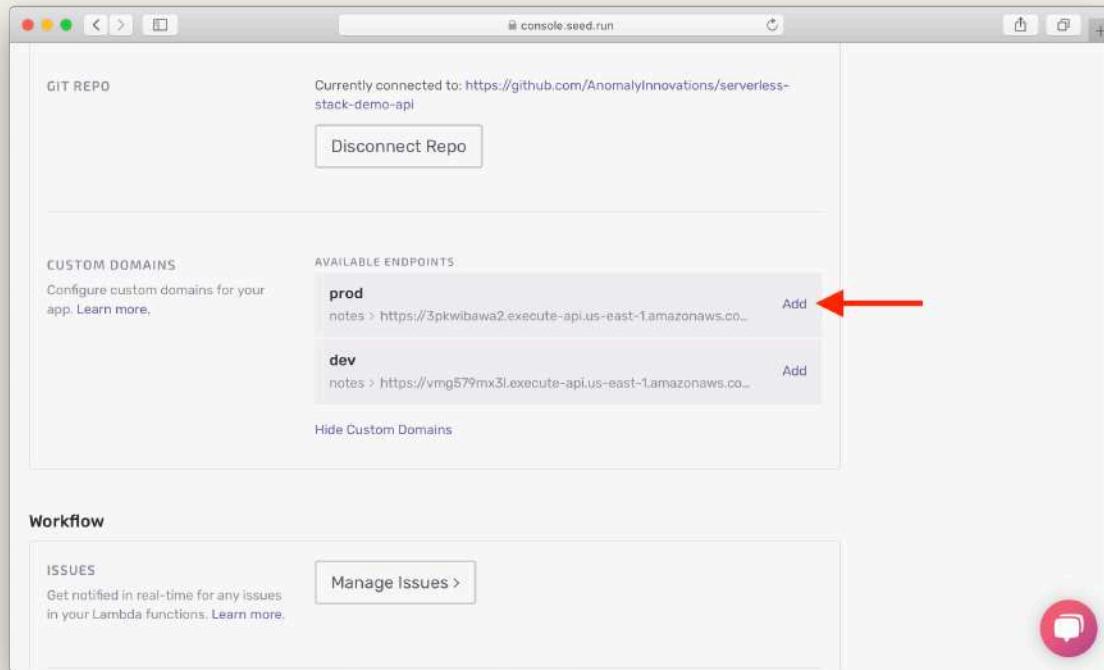
Seed app pipeline screenshot

Here click on **Edit Custom Domains**.



Click Edit Custom Domains in app settings screenshot

And click **Add** for our production endpoint.

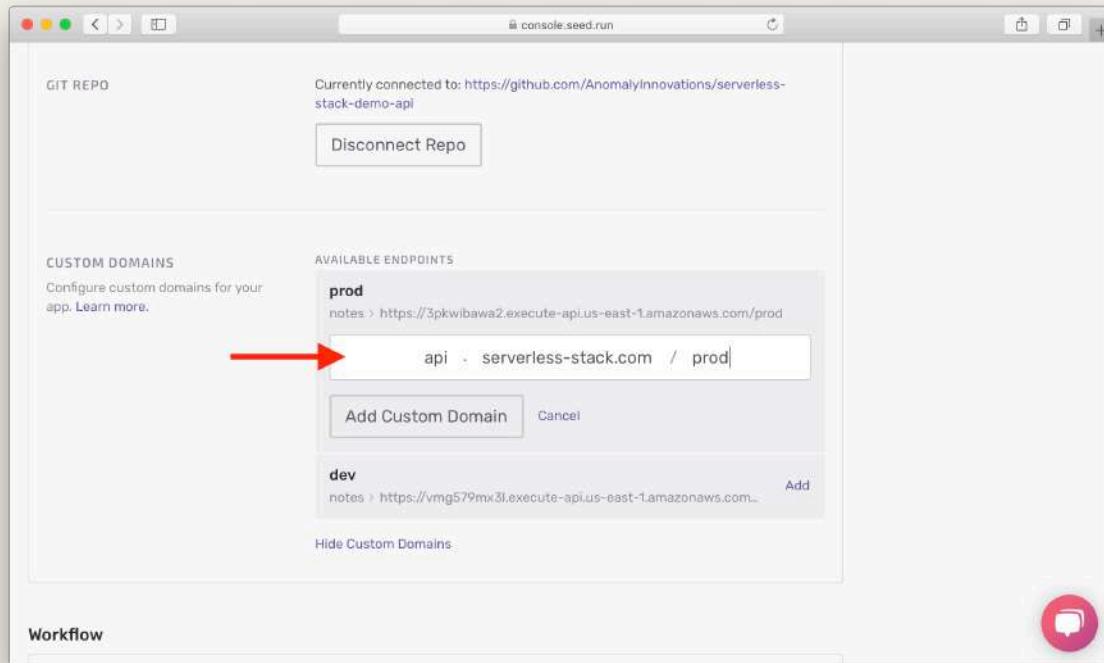


Click Add for production endpoint in custom domain settings

Seed will pull up any domains you have configured in [Route 53](#).

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.

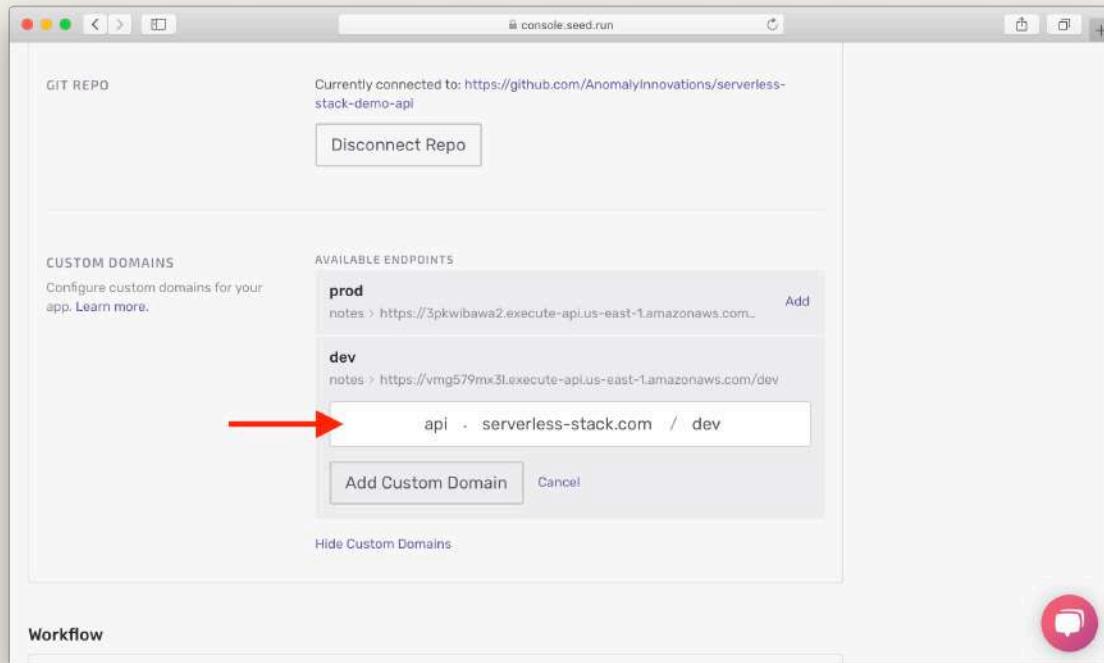
And hit **Add Custom Domain**.



Click Add Custom Domain button for prod endpoint

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`.



Click Add Custom Domain button for dev endpoint

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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](#) chapter, we used a simple utility called [AWS API Gateway Test CLI](#).

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](#) 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](#) 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](#) 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](#) 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 like 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](#) 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.

That's it! Now we are ready to plug our new backend into our React app.



Help and discussion

View the [comments for this chapter on our forums](#)

Deploying the frontend to production

Automating React Deployments

Now that we have our backend deployed to production, we are ready to deploy our frontend to production as well! We'll be using a service called [Netlify](#) 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](#).

Just as in the case with our backend, we could use [Travis CI](#) or [Circle CI](#) 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.



Help and discussion

View the [comments for this chapter on our forums](#)

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 connects 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](#) 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](#) 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](#). This will be fairly similar to what we did for our serverless backend API.



Help and discussion

View the [comments for this chapter on our forums](#)

Create a Build Script

Before we can add our project to [Netlify](#) 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](#) 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.



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.



Help and discussion

View the [comments for this chapter](#) on our forums

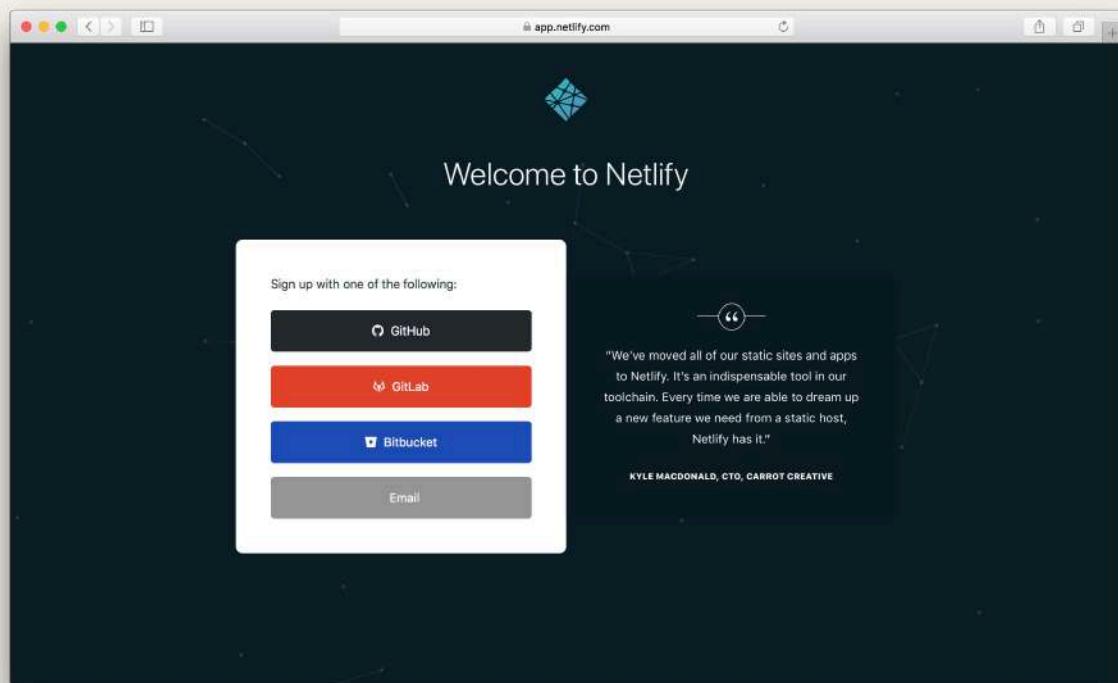


For reference, here is the code we are using

Frontend Source: [create-a-build-script](#)

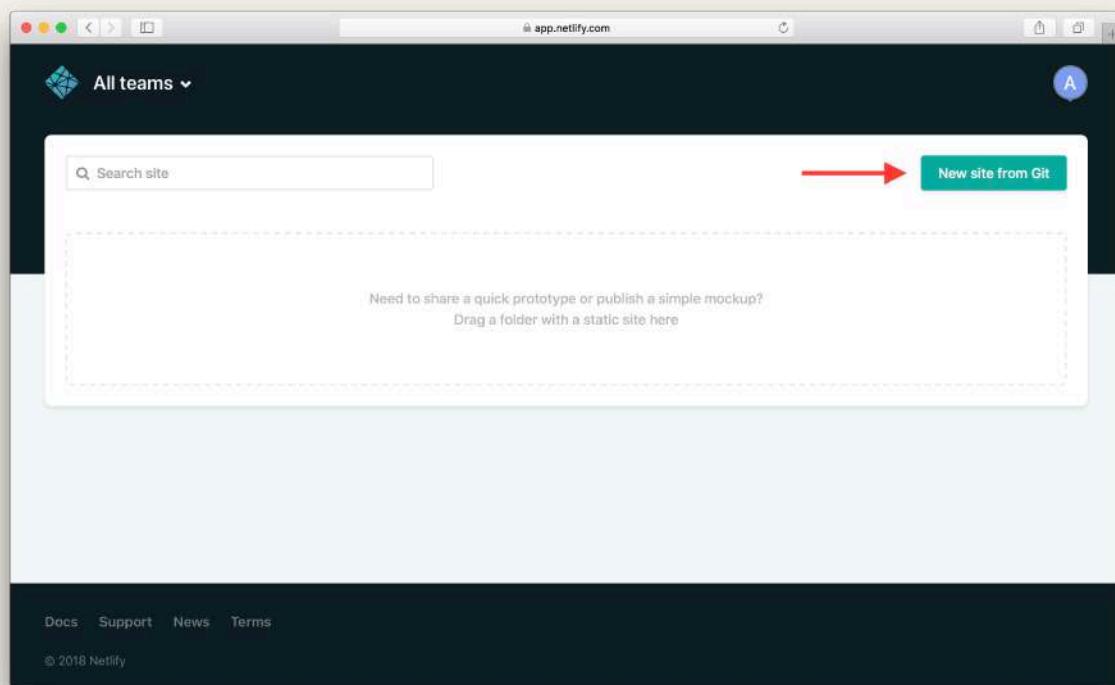
Setting up Your Project on Netlify

Now we are going to set our React app on [Netlify](#). Start by [creating a free account](#).



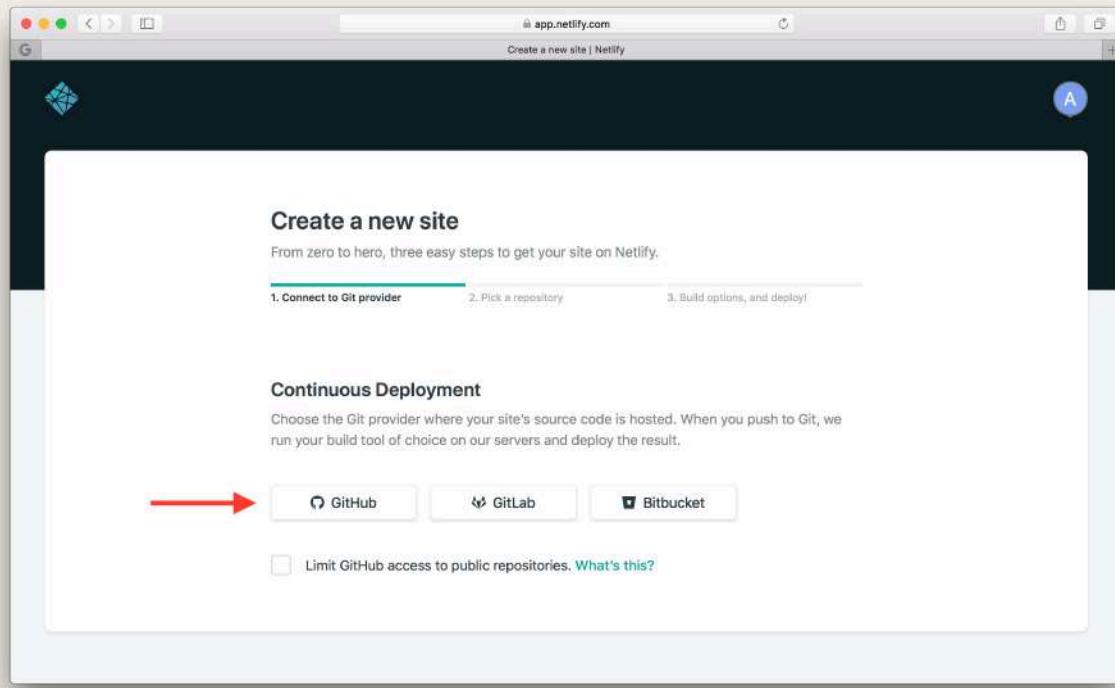
Signup for Netlify screenshot

Next, create a new site by hitting the **New site from Git** button.



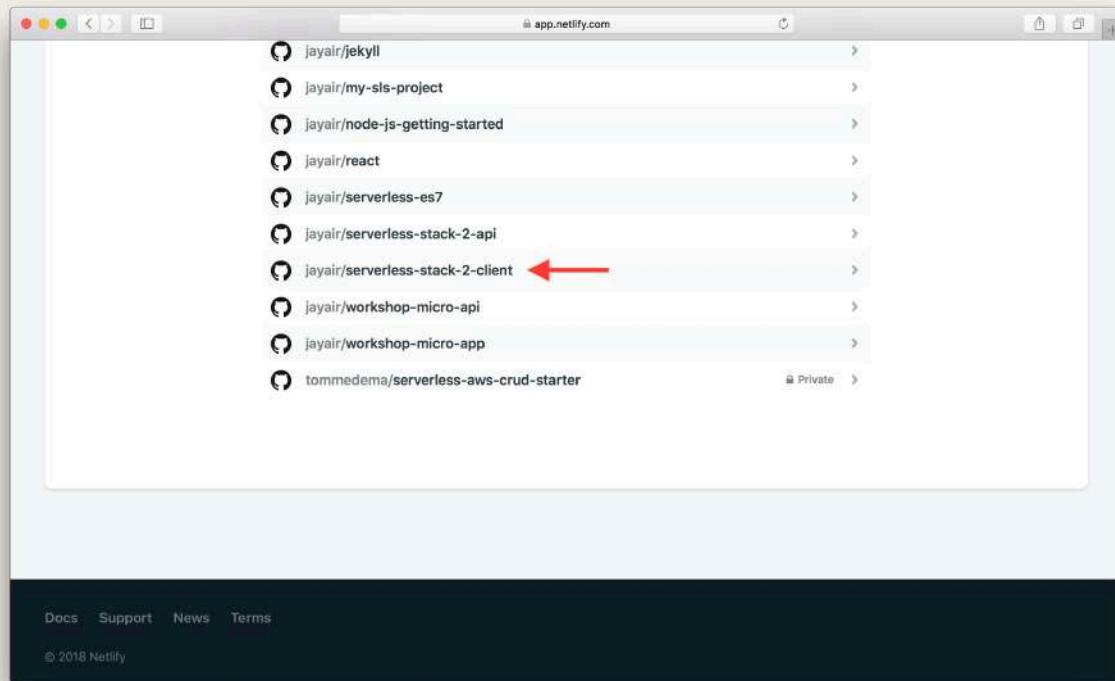
Hit new site from git button screenshot

Pick **GitHub** as your provider.



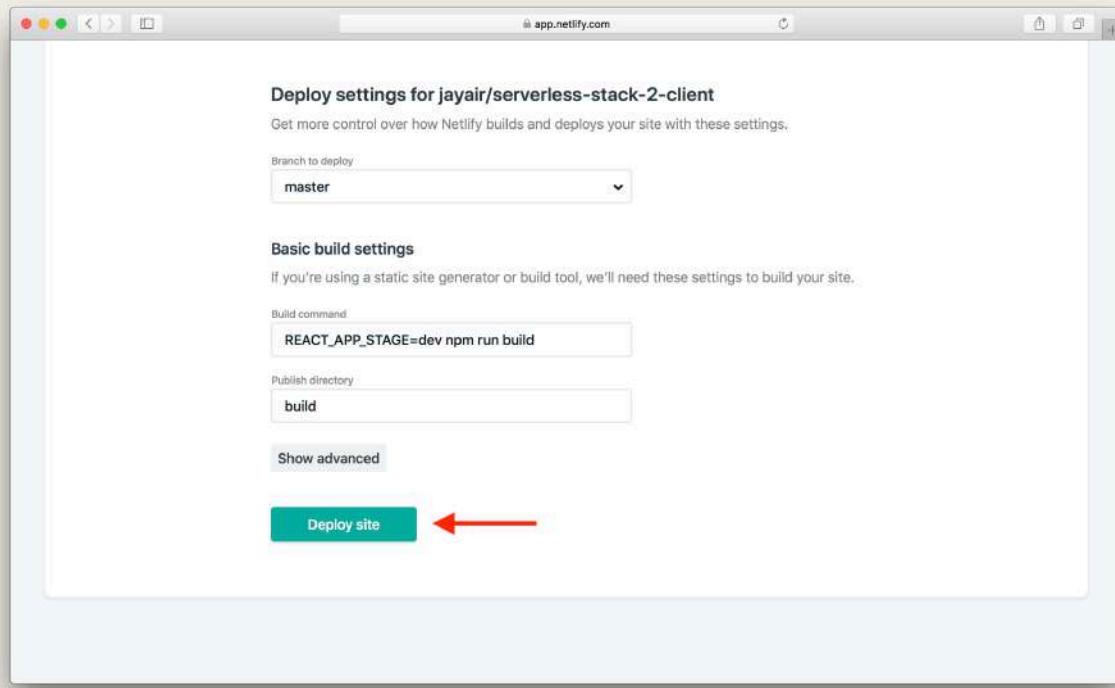
Select GitHub as provider screenshot

Then pick your project from the list.



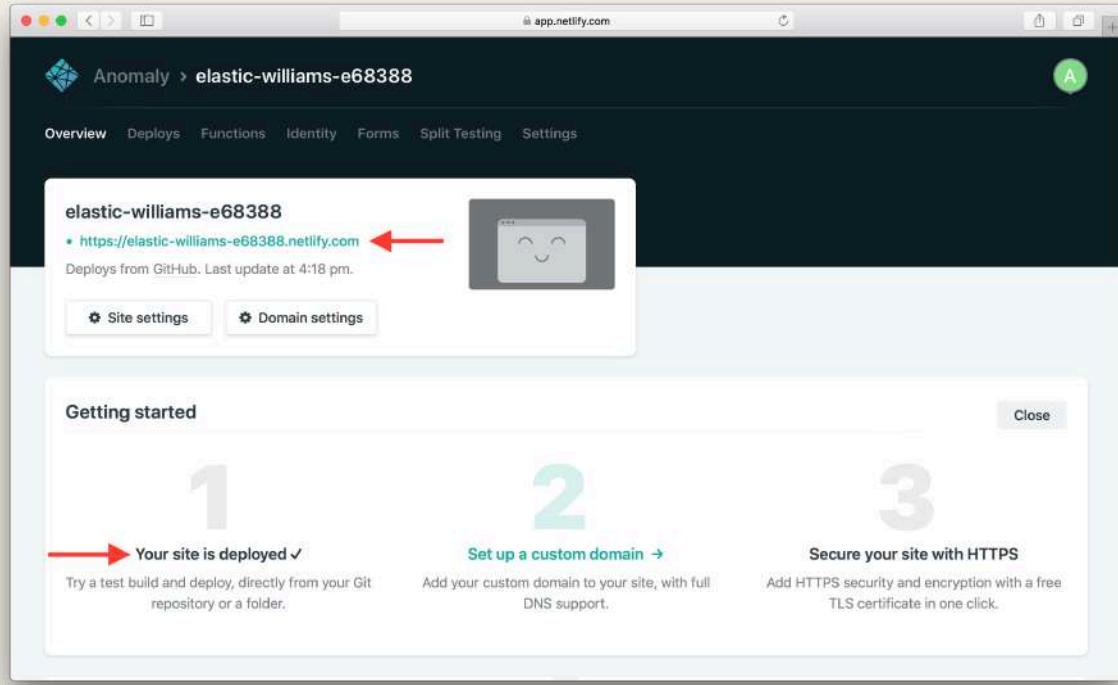
Select GitHub repo from list screenshot

It'll default the branch to master. We can now deploy our app! Hit **Deploy site**.



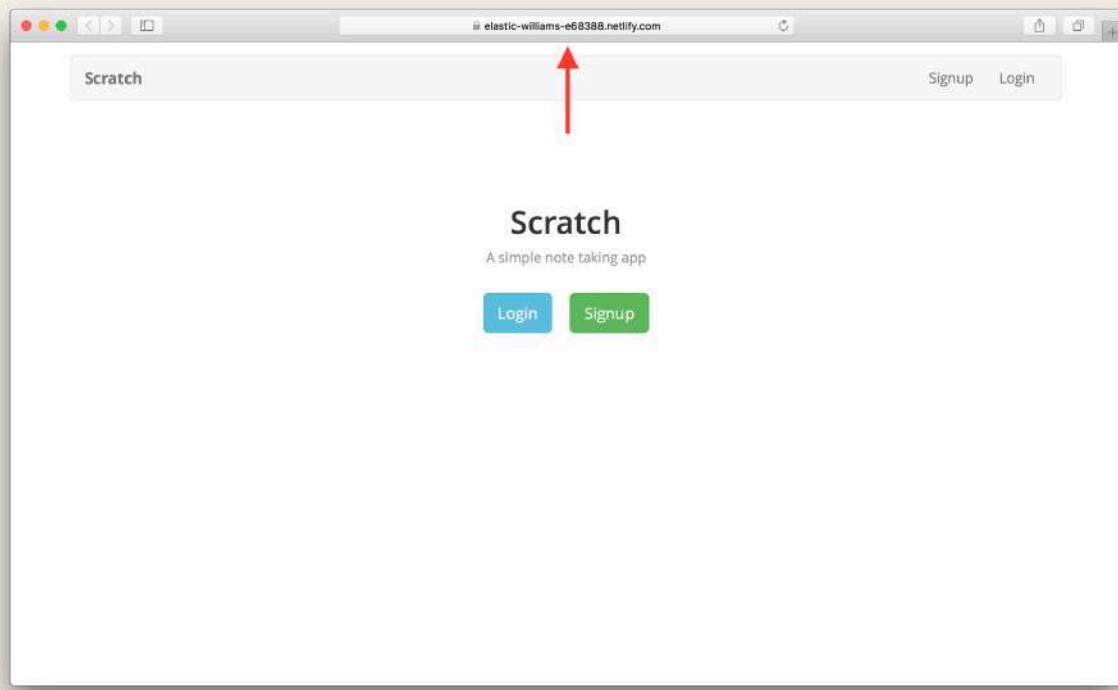
Hit Deploy site screenshot

This should be deploying our app. Once it is done, click on the deployment.



View deployed site screenshot

And you should see your app in action!



Netlify deployed notes app screenshot

Of course, it is hosted on a Netlify URL. We'll change that by configuring custom domains next.



Help and discussion

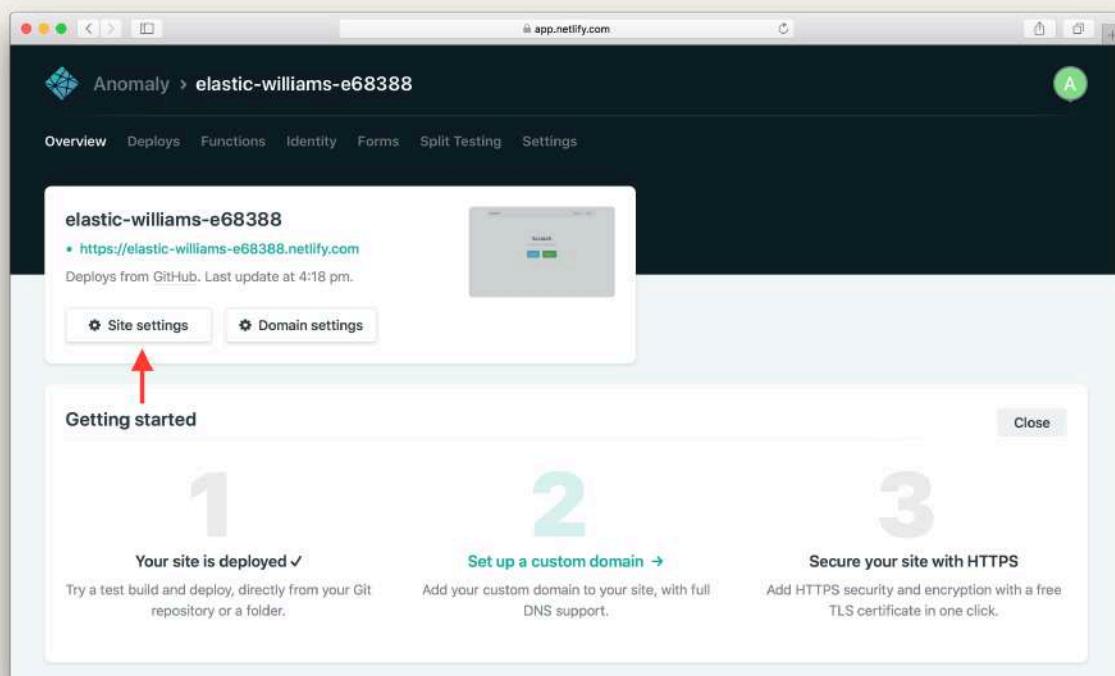
View the [comments for this chapter on our forums](#)

Custom Domains in Netlify

Now that we have our first deployment, let's configure a custom domain for our app through Netlify. Recall that back in the [Set custom domains through Seed](#) chapter we purchased a domain on [Amazon Route 53](#).

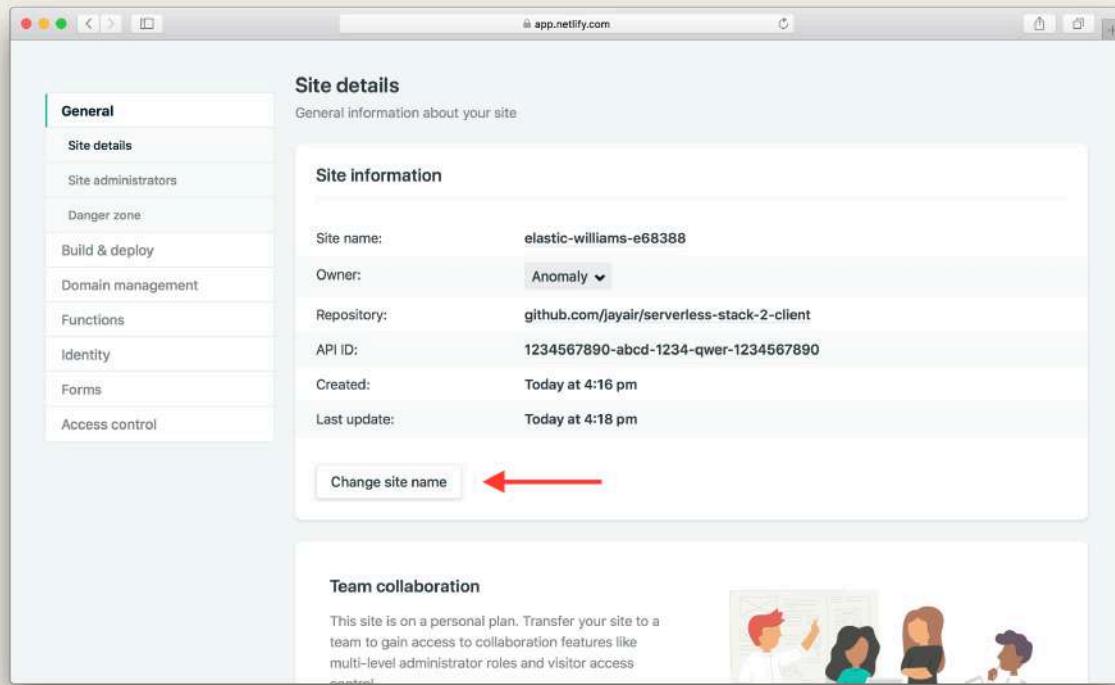
Pick a Netlify Site Name

From the project page in Netlify, hit **Site settings**.



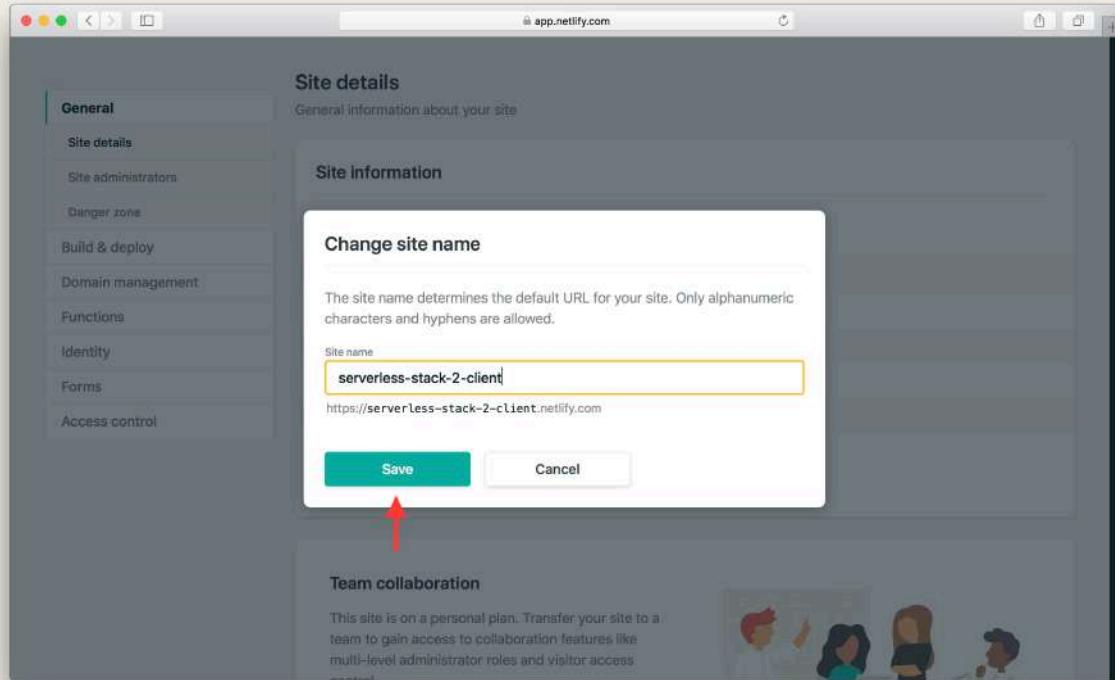
Netlify hit Site settings screenshot

Under **Site information** hit **Change site name**.



Hit Change site name screenshot

The site names are global, so pick a unique one. In our case we are using `serverless-stack-2-client`. And hit **Save**.

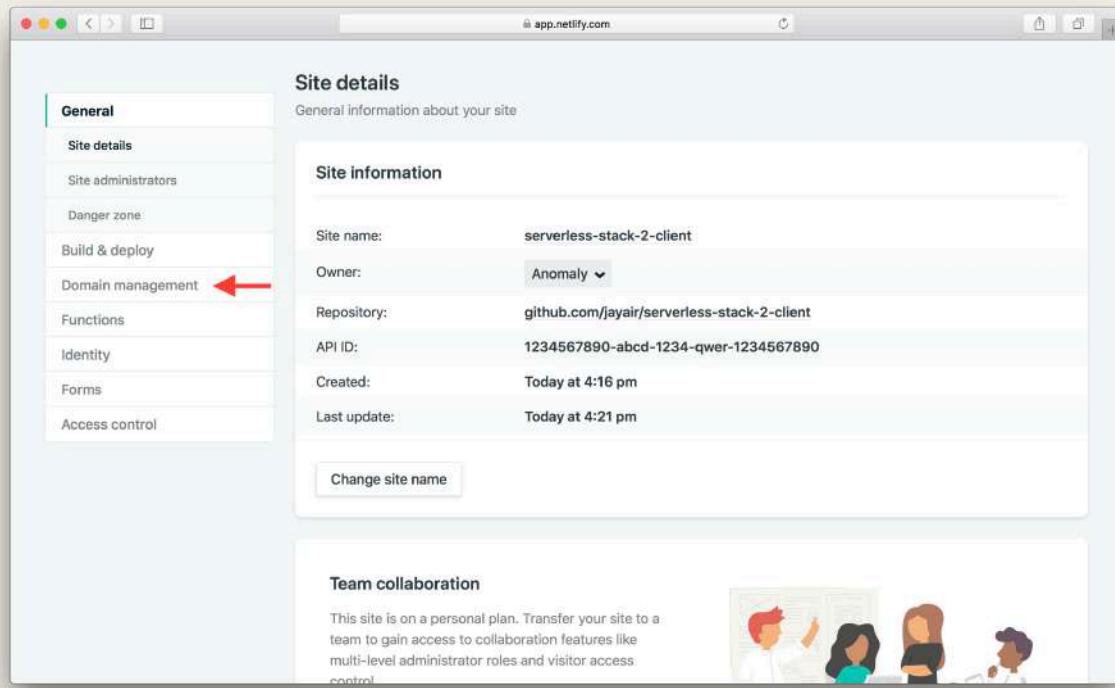


Save change site name screenshot

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.

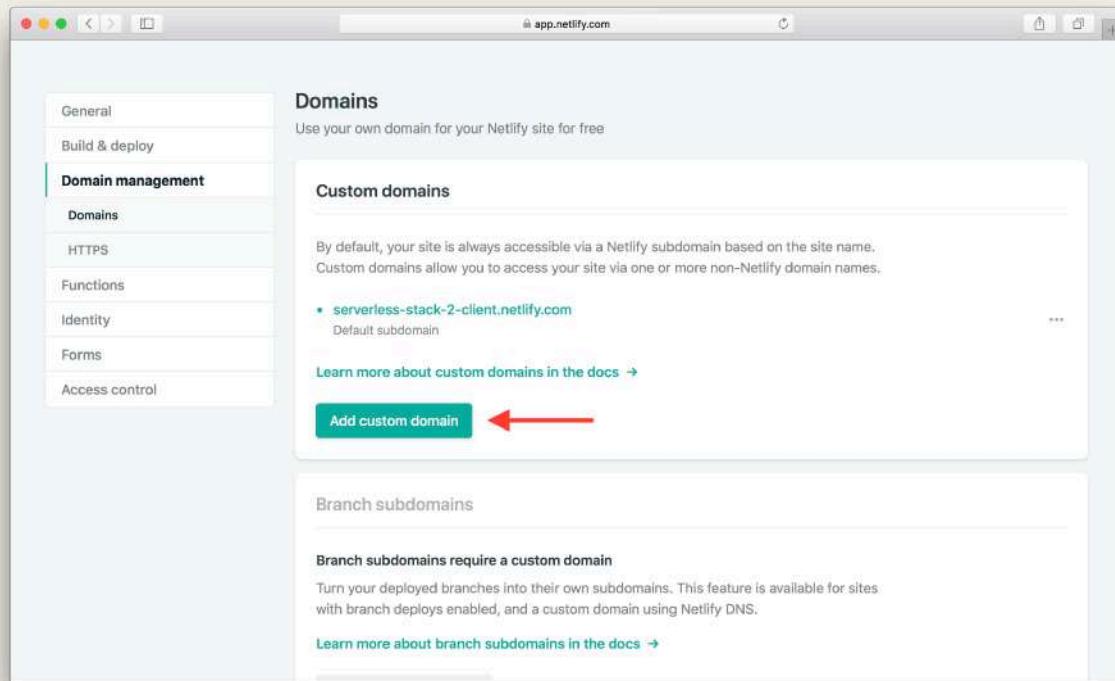
Domain Settings in Netlify

Next hit **Domain management** from the side panel.



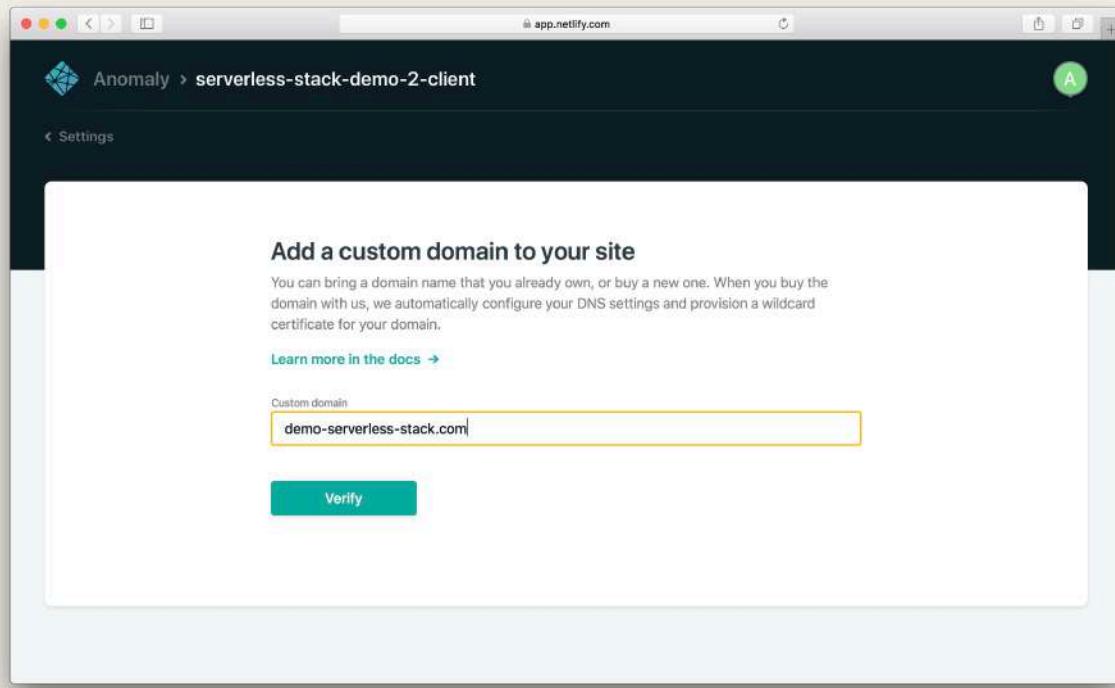
Select Domain management screenshot

And hit **Add custom domain**.



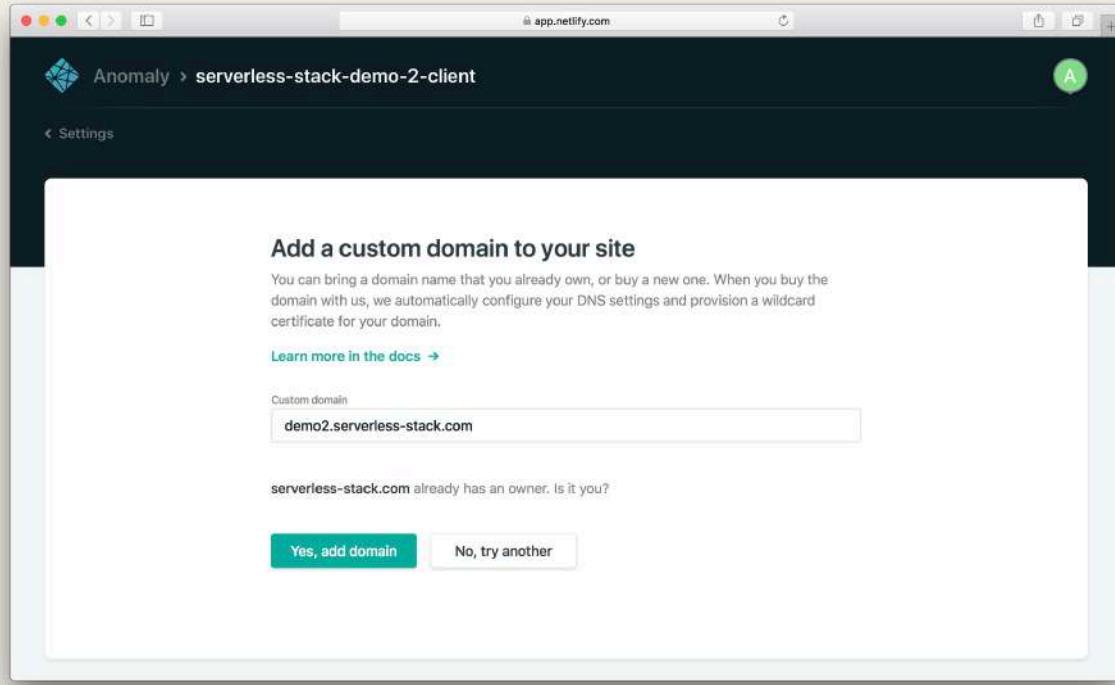
Click Add custom domain screenshot

Type in the name of our domain, for example it might be `demo-serverless-stack.com`. And hit **Save**.



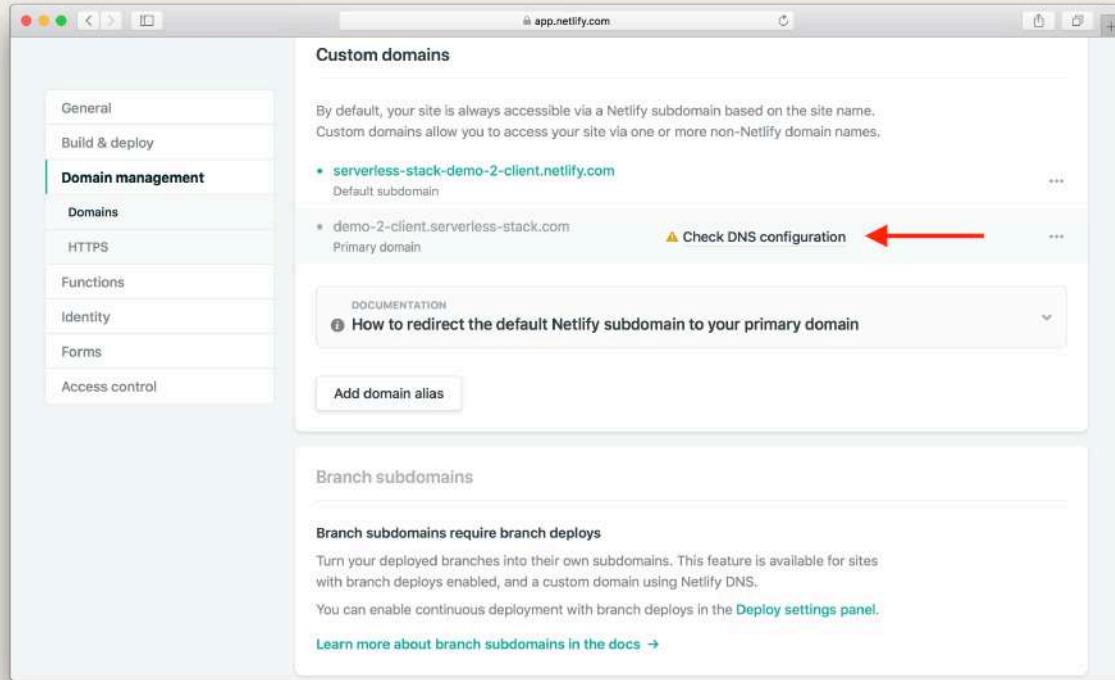
Enter custom domain screenshot

This will ask you to verify that you are the owner of this domain and to add it. Click **Yes, add domain**.



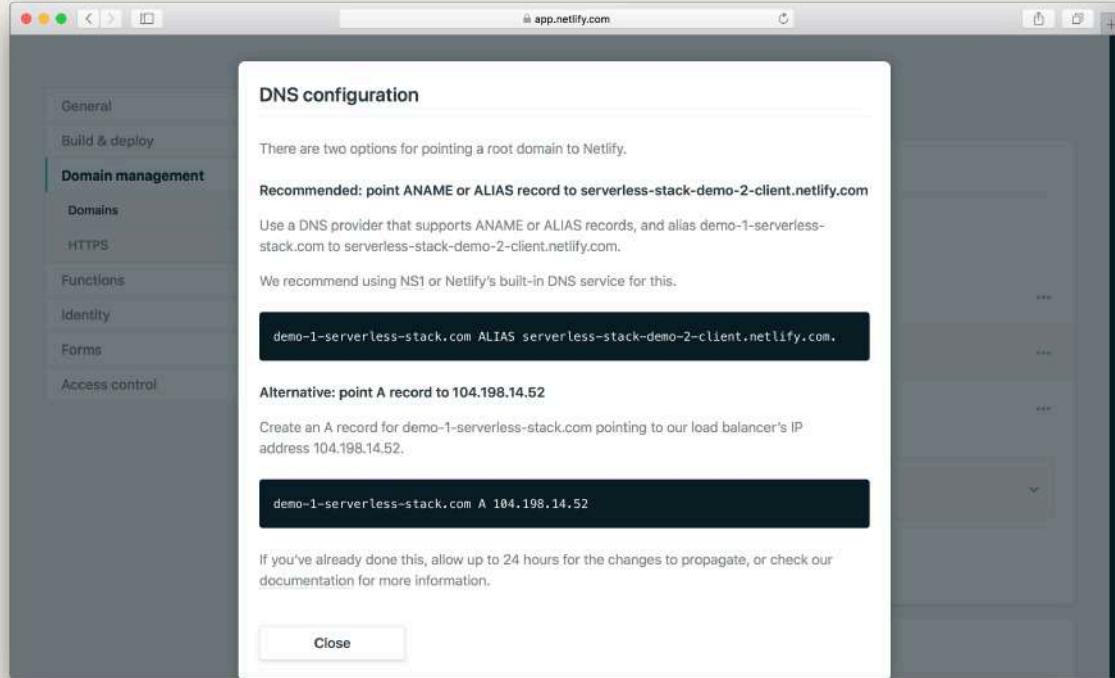
Add root domain screenshot

Next hit **Check DNS configuration**.



Hit check DNS configuration screenshot

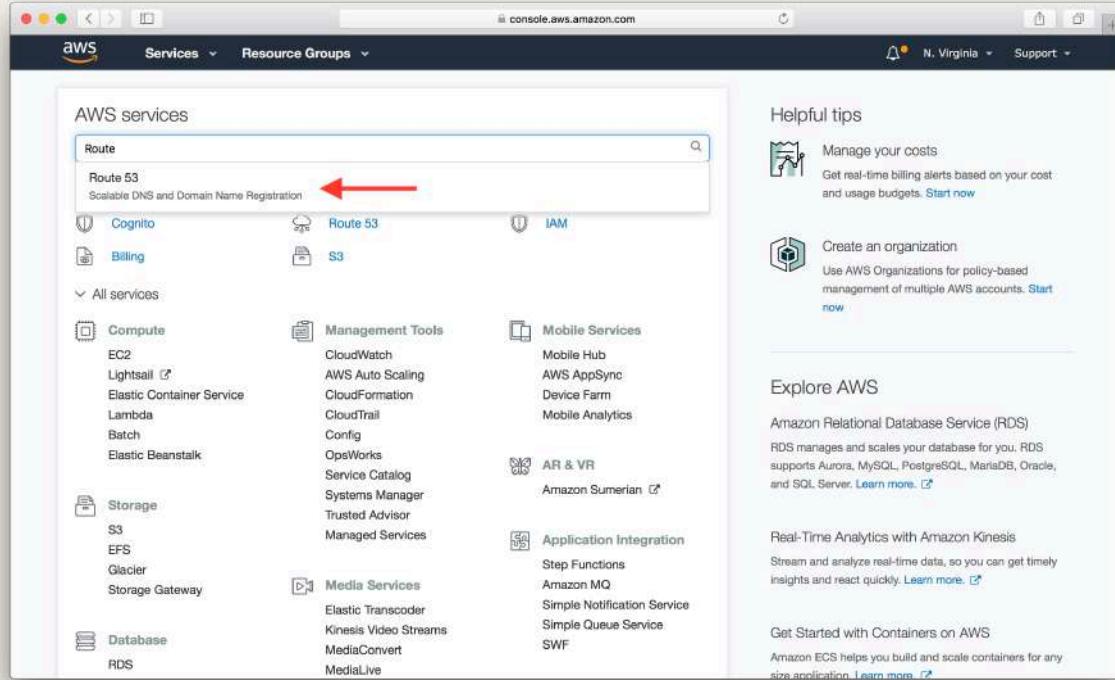
This will show you the instructions for setting up your domain through Route 53.



DNS configuration dialog screenshot

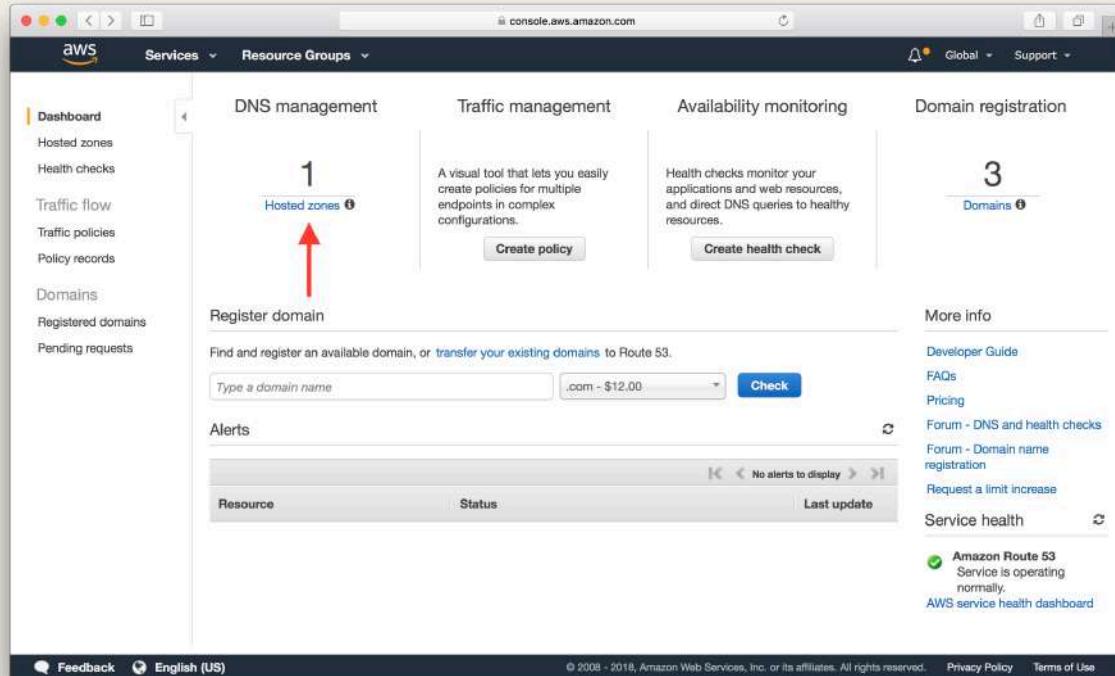
DNS Settings in Route 53

To do this we need to head back to the [AWS Console](#), and search for Route 53 as the service.



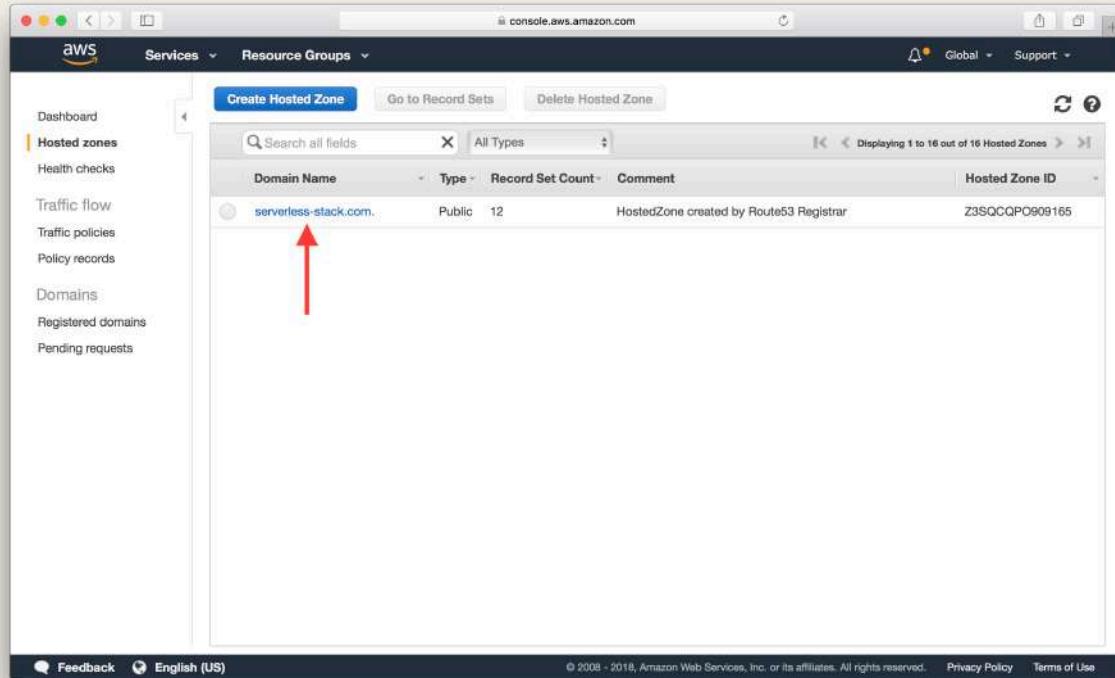
Select Route 53 service screenshot

Click on **Hosted zones**.



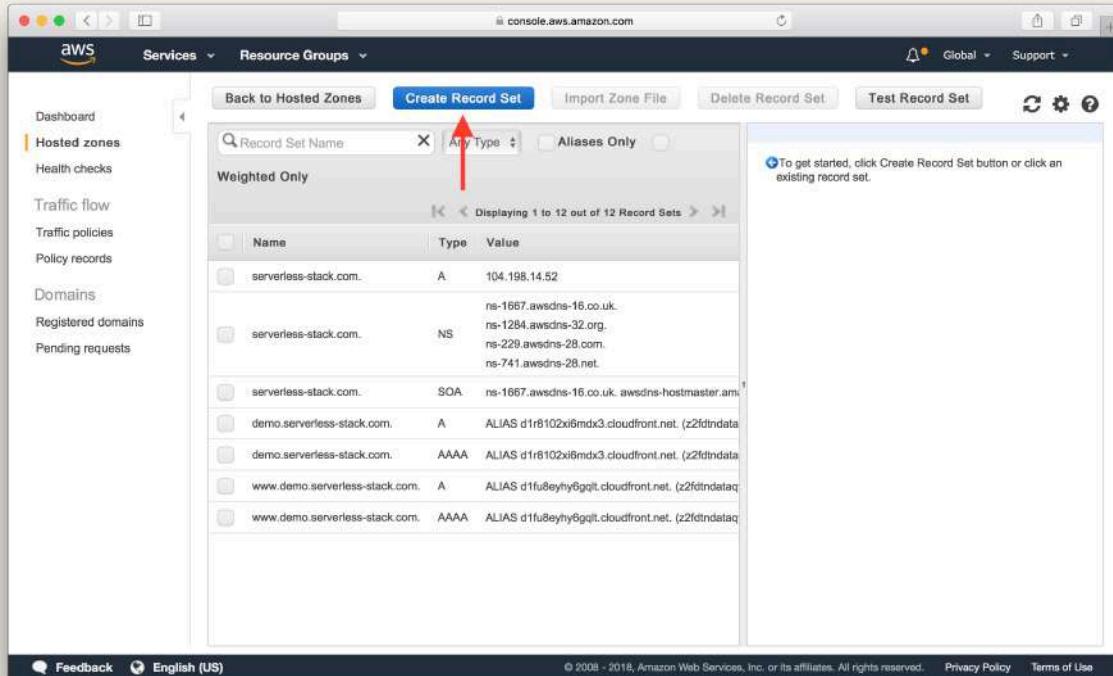
Select Route 53 hosted zones screenshot

And select the domain we want to configure.



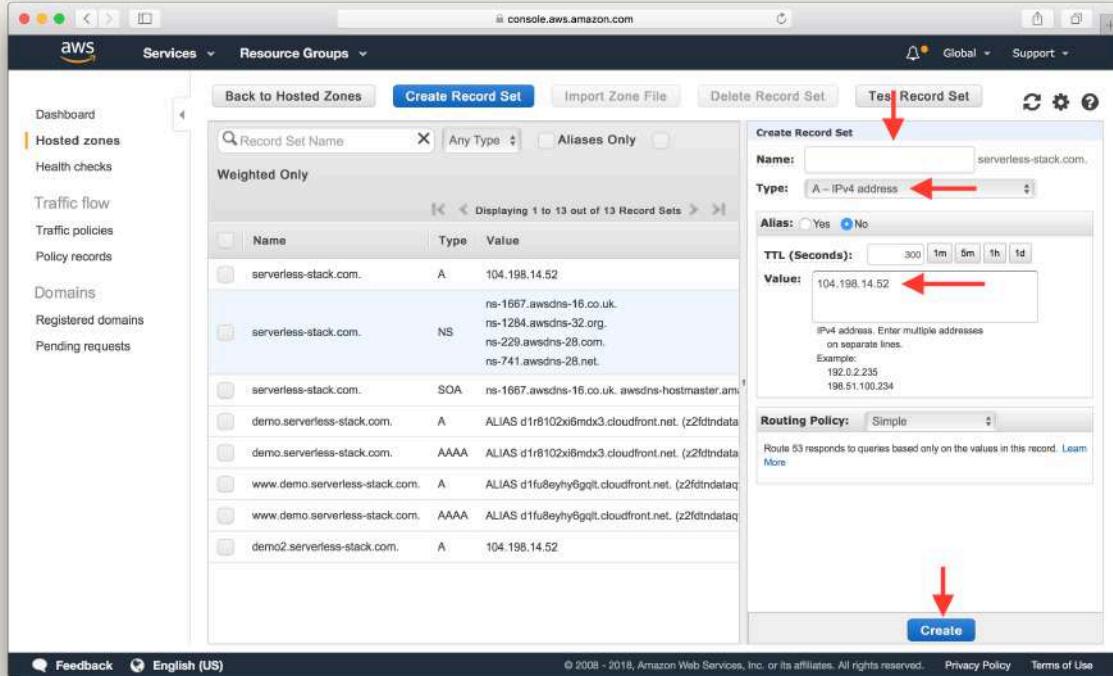
Select Route 53 domain screenshot

Here click on **Create Record Set**.



Create first Route 53 record set screenshot

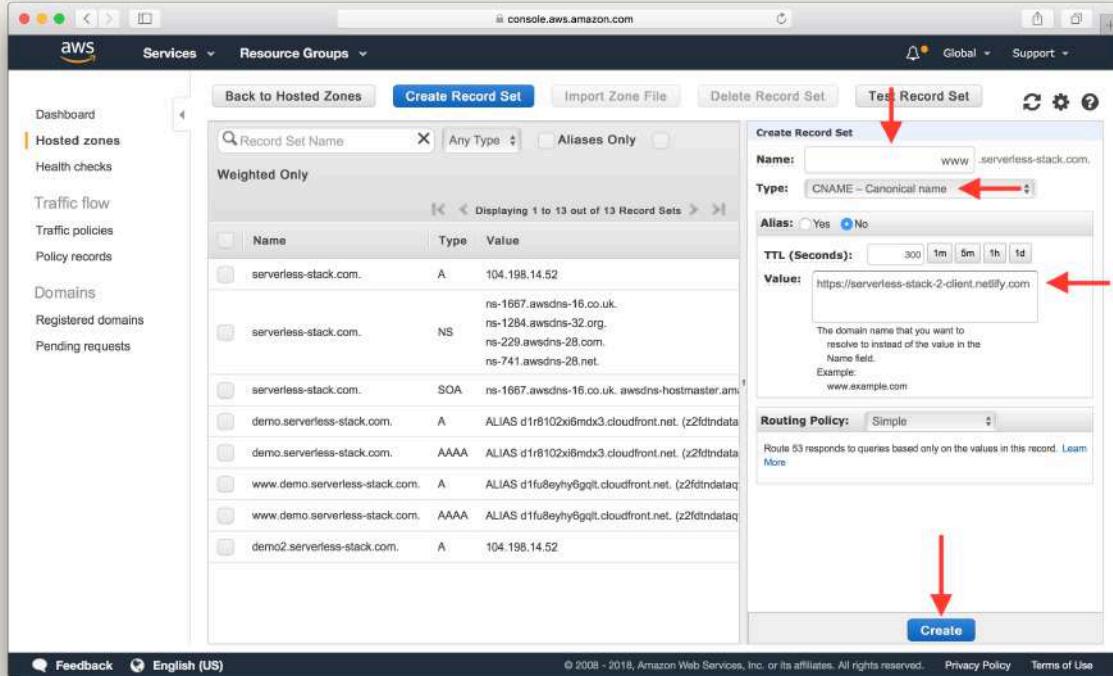
Select **Type** as **A - IPv4 address** and set the **Value** to **104.198.14.52**. And hit **Create**. We get this IP from the [Netlify docs on adding custom domains](#).



Add A record screenshot

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**.

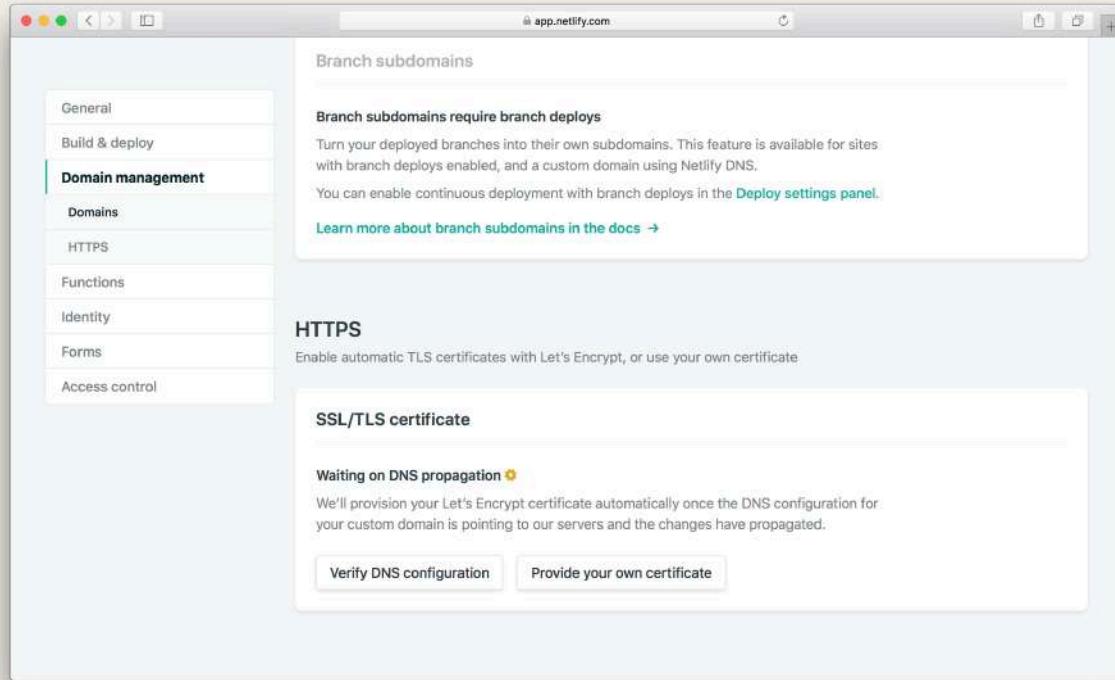


Add CNAME record screenshot

And give the DNS around 30 minutes to update.

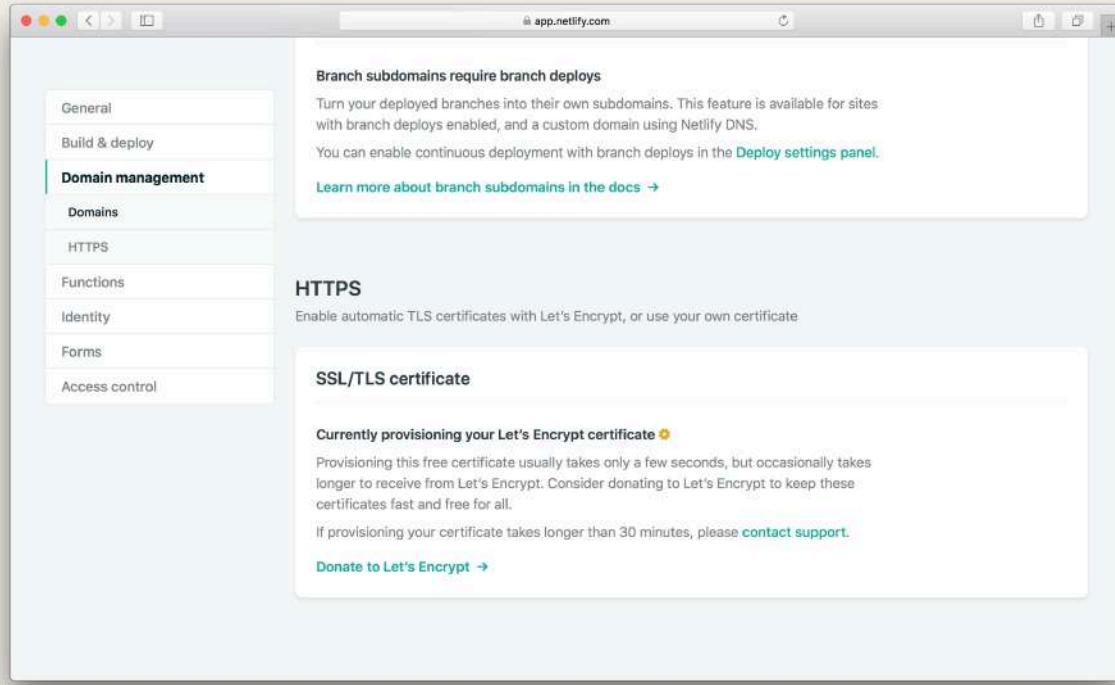
Configure SSL

Back in Netlify, hit **HTTPS** in the side panel. And it should say that it is waiting for the DNS to propagate.



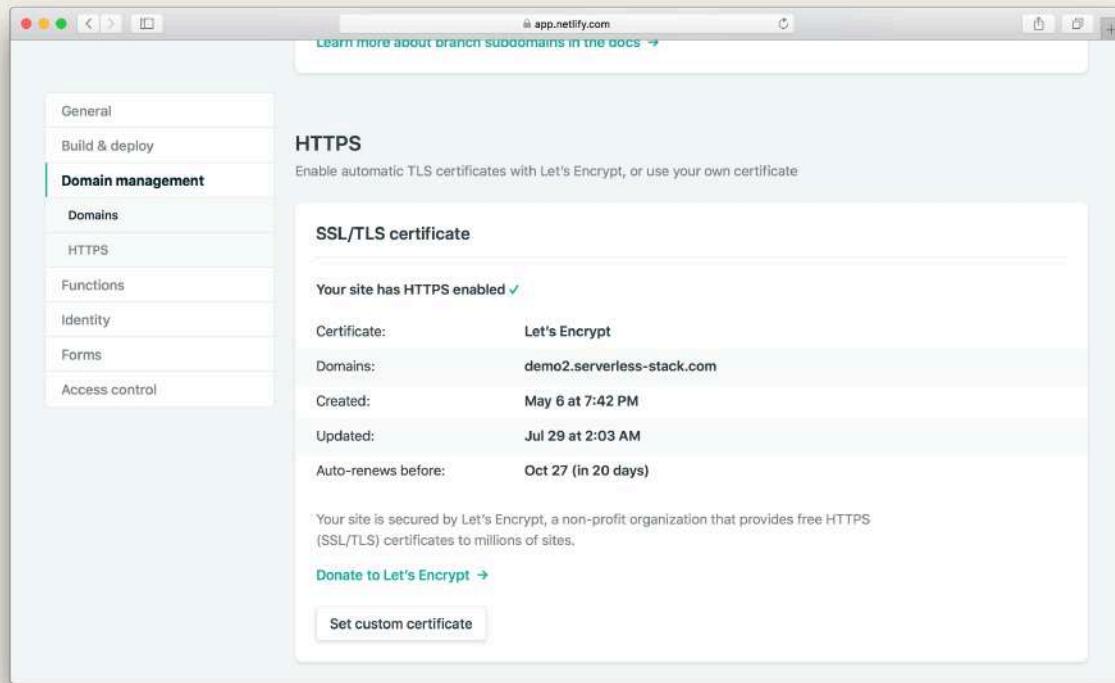
Waiting on DNS propagation screenshot

Once that is complete, Netlify will automatically provision your SSL certificate using Let's Encrypt.



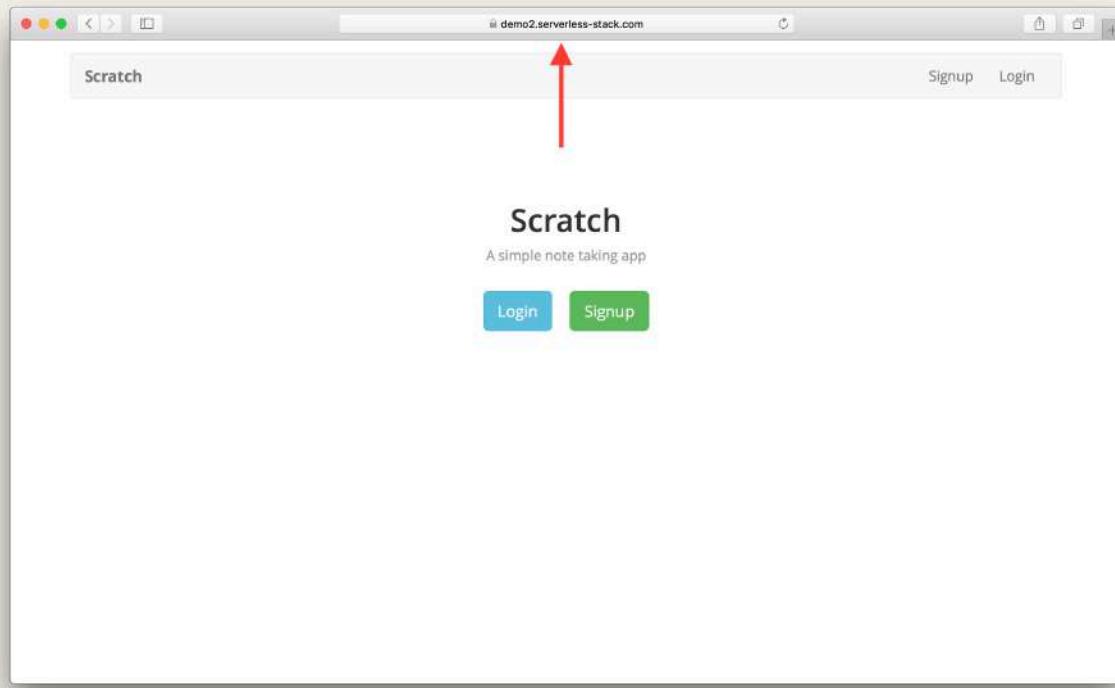
Provisioning Let's Encrypt Certificate screenshot

Wait a few seconds for the certificate to be provisioned.



SSL certificate provisioned screenshot

Now if you head over to your browser and go to your custom domain, your notes app should be up and running!



Notes app on custom domain screenshot

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.



Help and discussion

View the [comments](#) for this chapter on our forums

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>
  )
}
```

```
        </div>
      </div>
    );
}
```

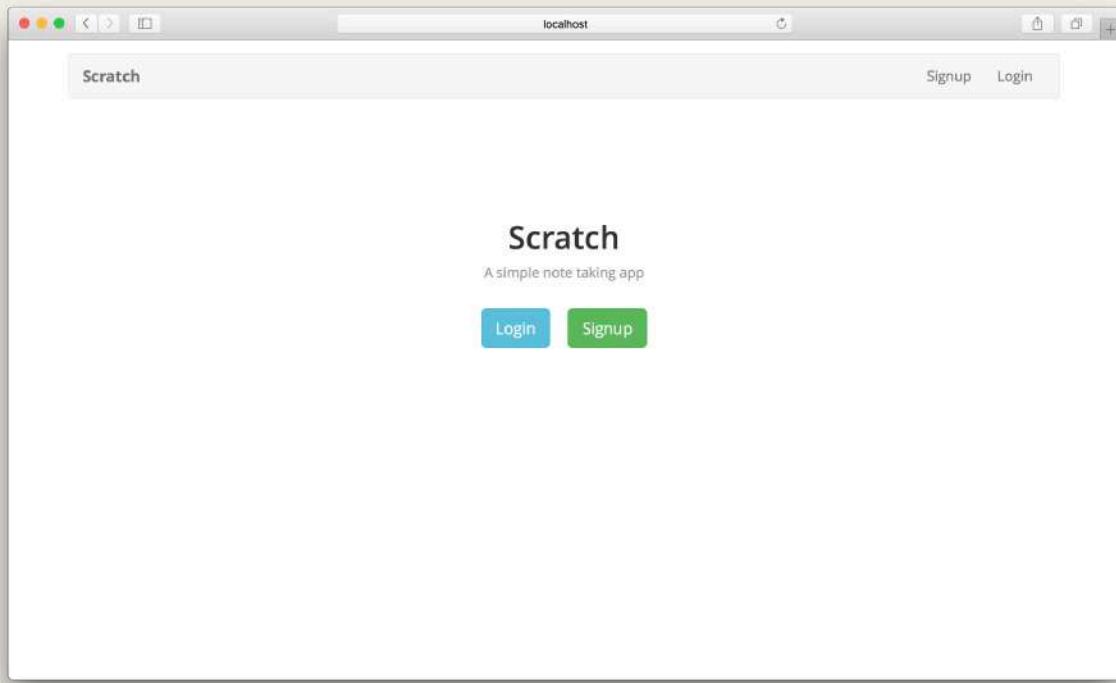
◆ CHANGE And import the Link component from React-Router in the header.

```
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.



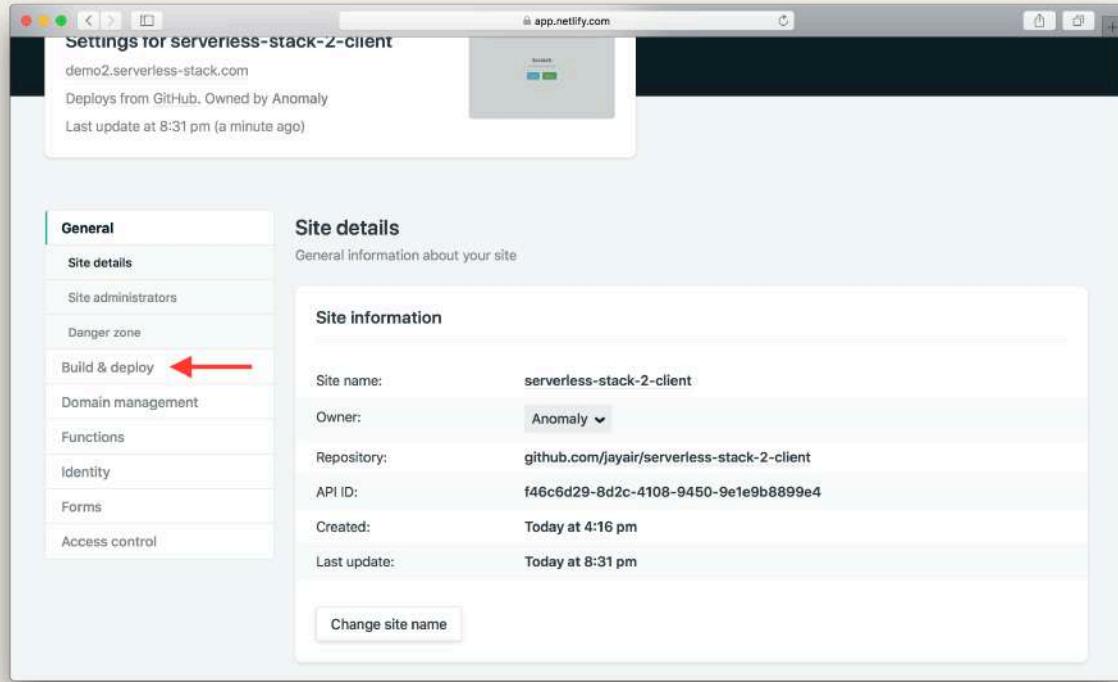
App updated lander screenshot

◆ CHANGE Let's commit these changes to Git.

```
$ git add .  
$ git commit -m "Updating the lander"
```

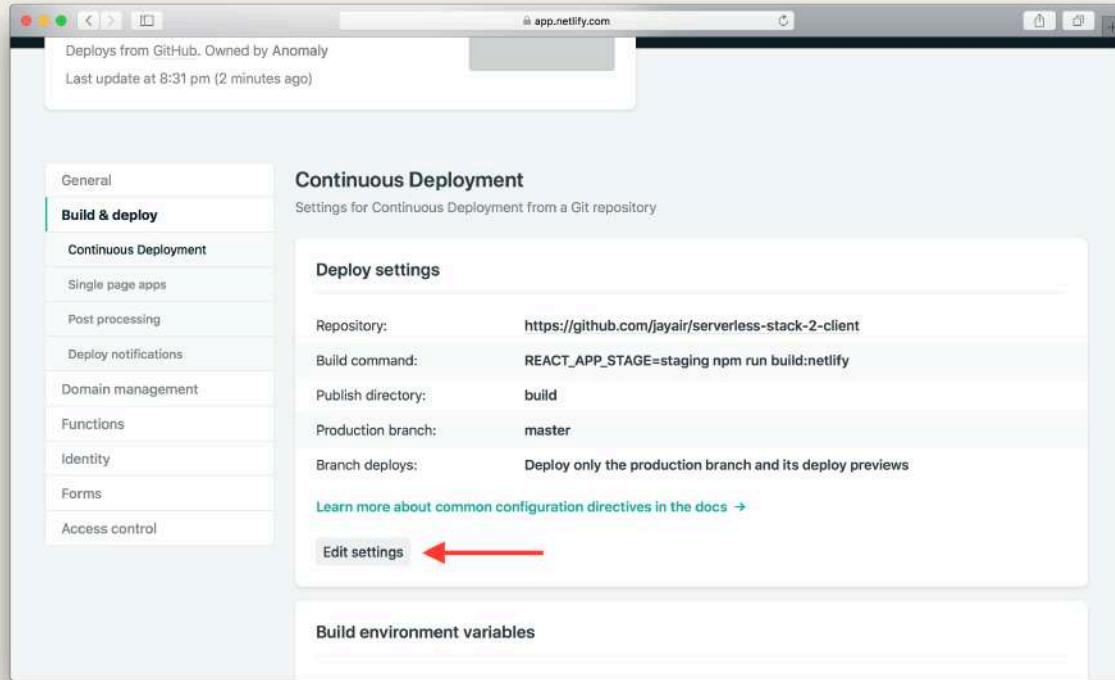
Create a Branch Deployment

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**.



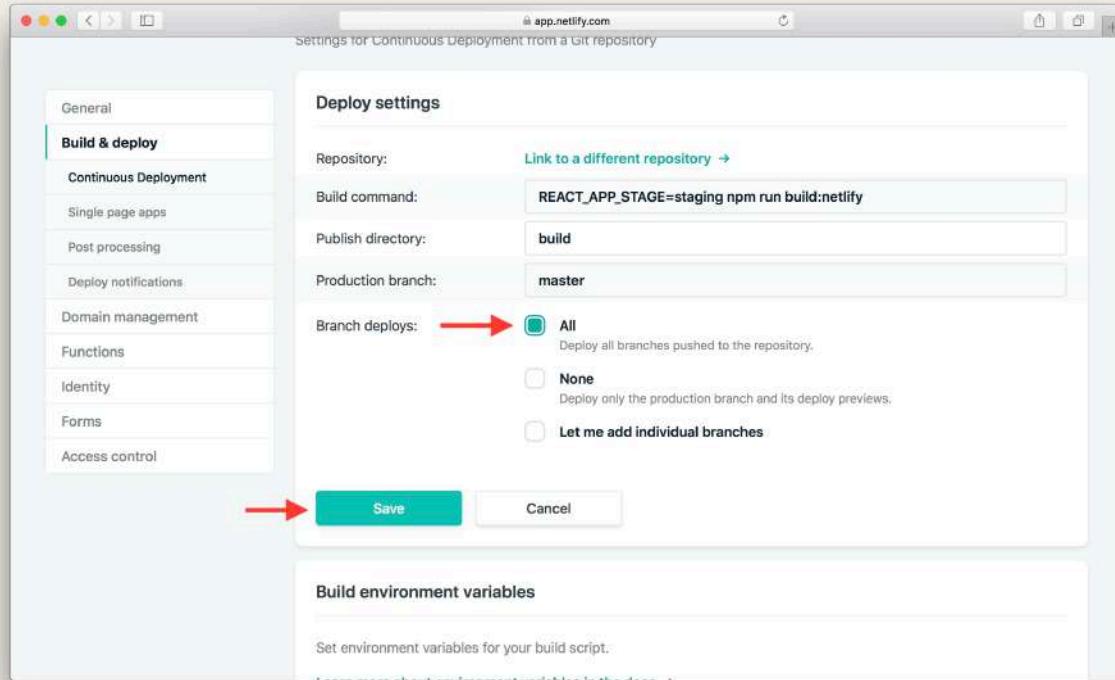
Select Build & deploy screenshot

And hit **Edit settings**.



Edit build settings screenshot

Set **Branch deploys** to **All** and hit **Save**.

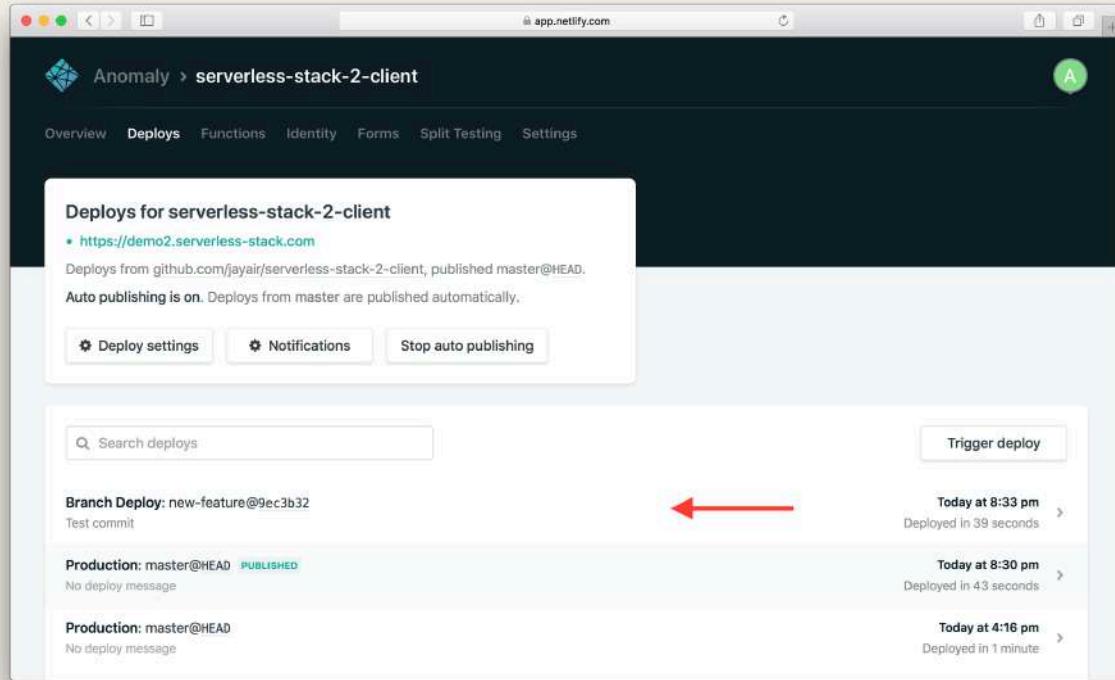


Set branch deploys to all screenshot

◆ 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.

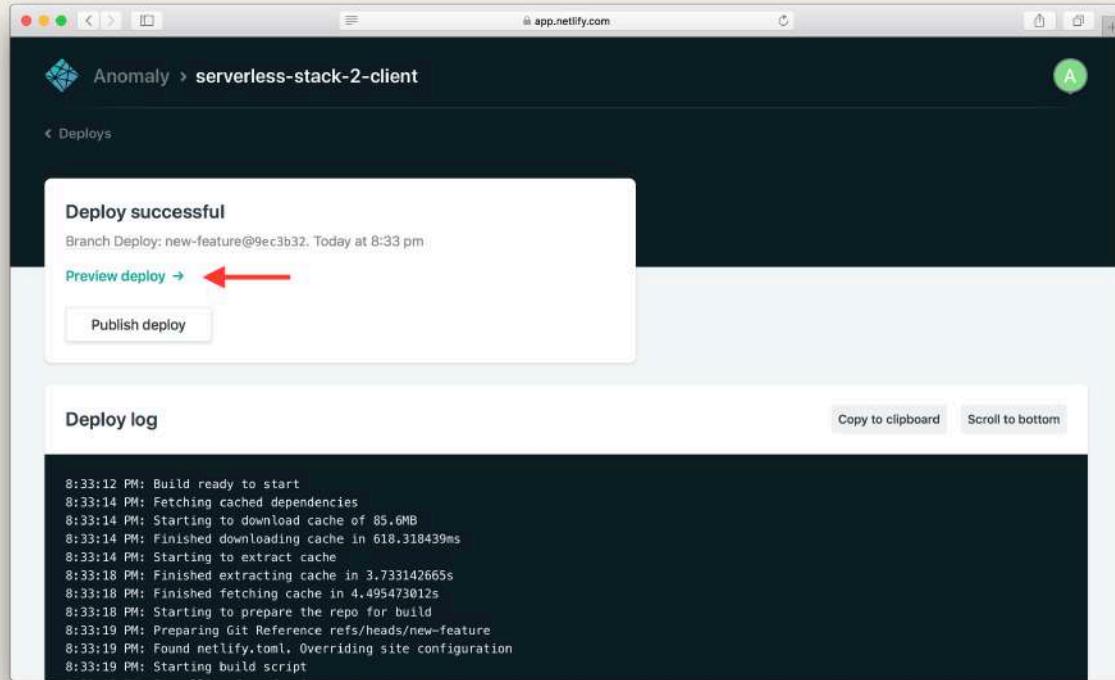
```
$ git push -u origin new-feature
```

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.



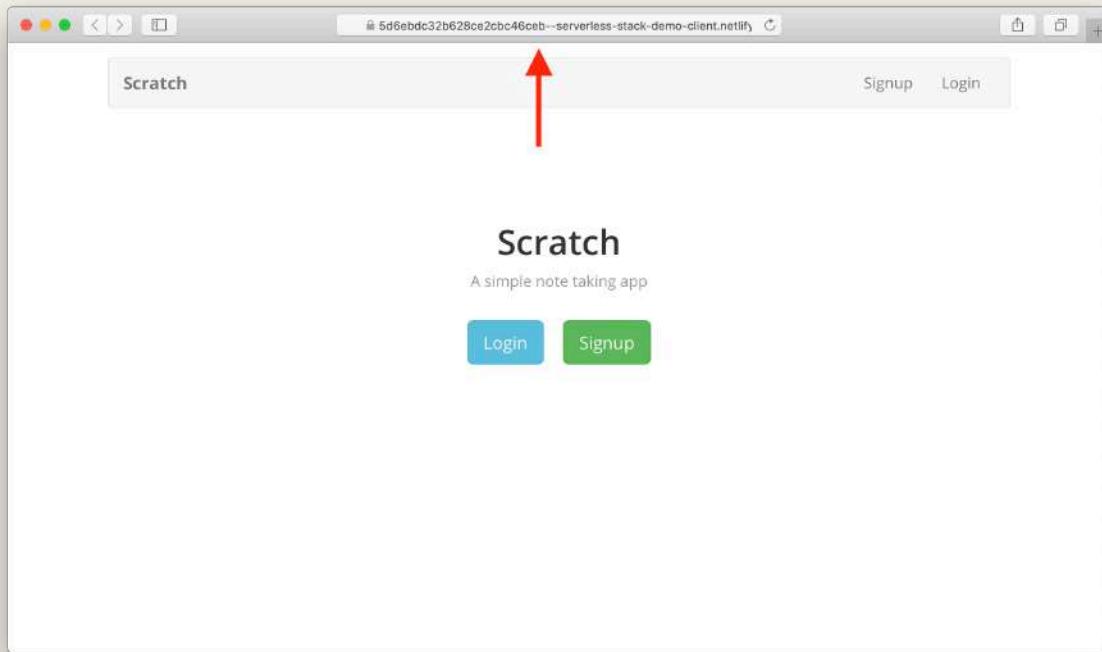
Click on new branch deploy screenshot

Hit **Preview deploy**.



Preview new branch deploy screenshot

And you can see a new version of your app in action!



Preview deploy in action screenshot

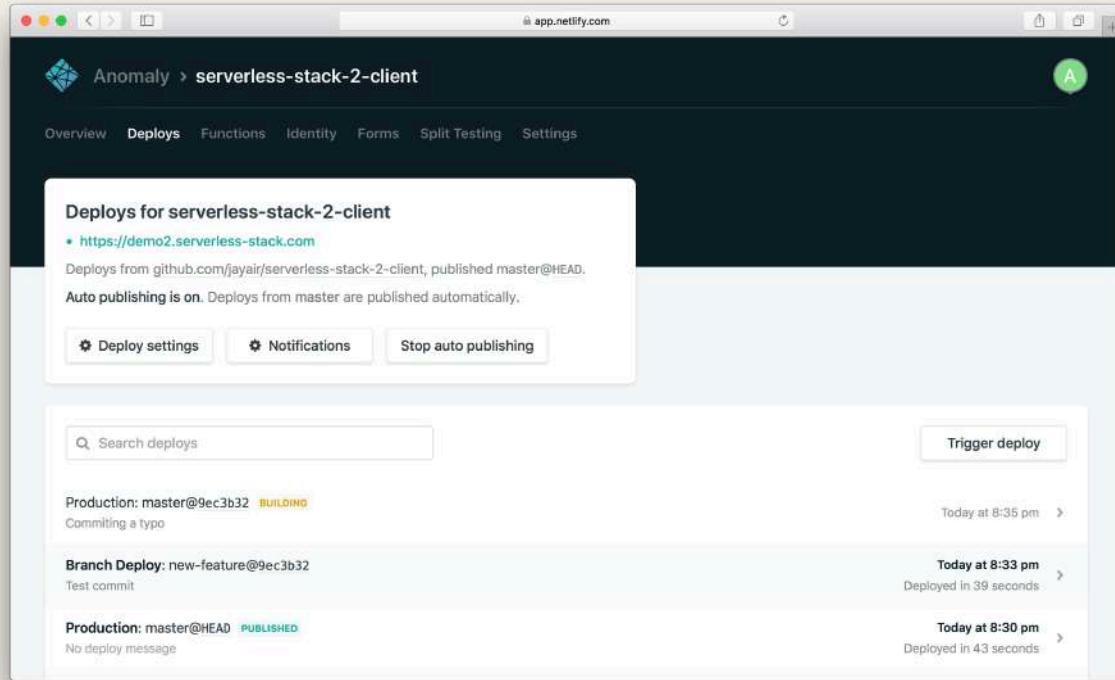
You can test around this version of our frontend app. It is connected to the dev version of our backend API. The idea is that we can test and play around with the changes here without affecting our production users.

Push to Production

◆ CHANGE Now if we feel happy with the changes we can push this to production just by merging to master.

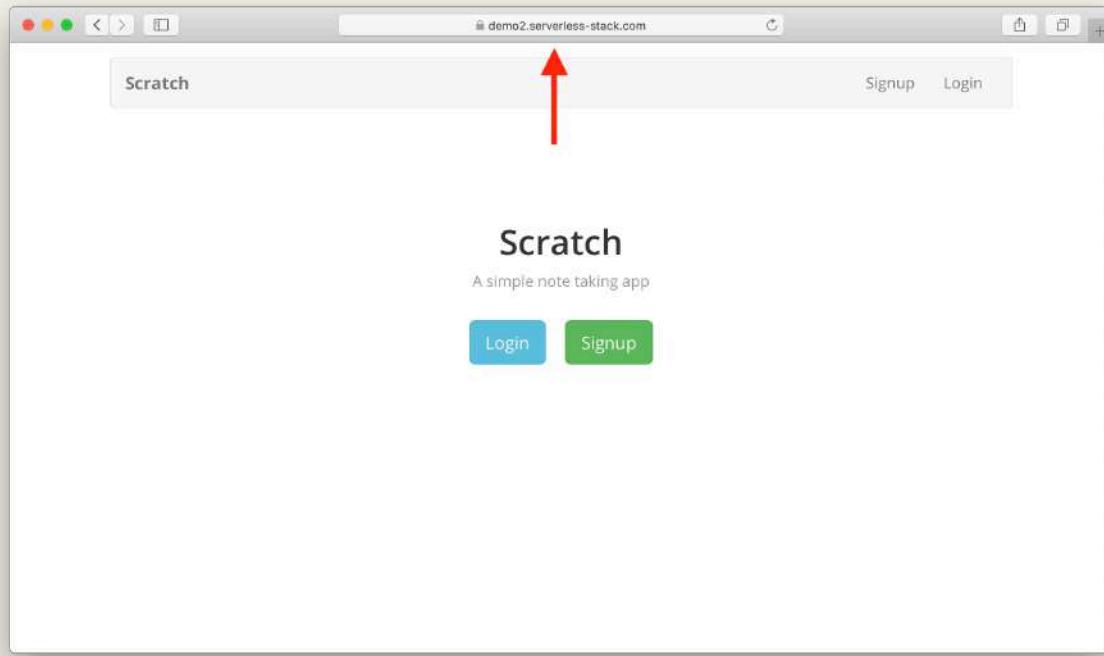
```
$ git checkout master  
$ git merge new-feature  
$ git push
```

You should see this deployment in action in Netlify.



Production deploy after merge screenshot

And once it is done, your changes should be live!

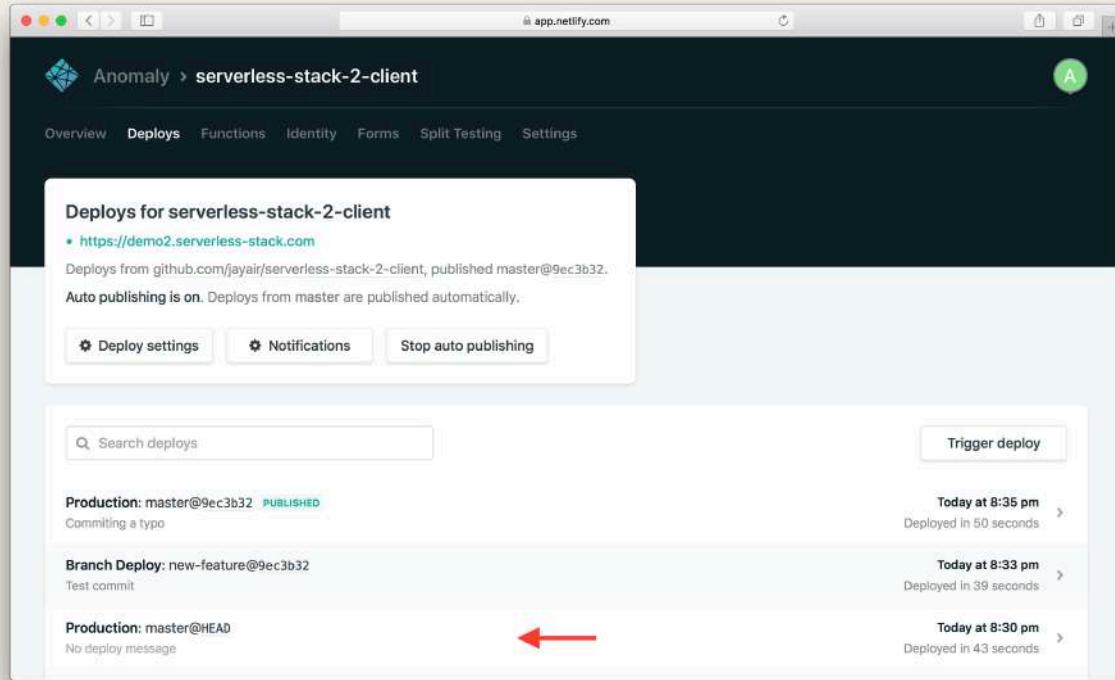


Production deploy is live screenshot

Rolling Back in Production

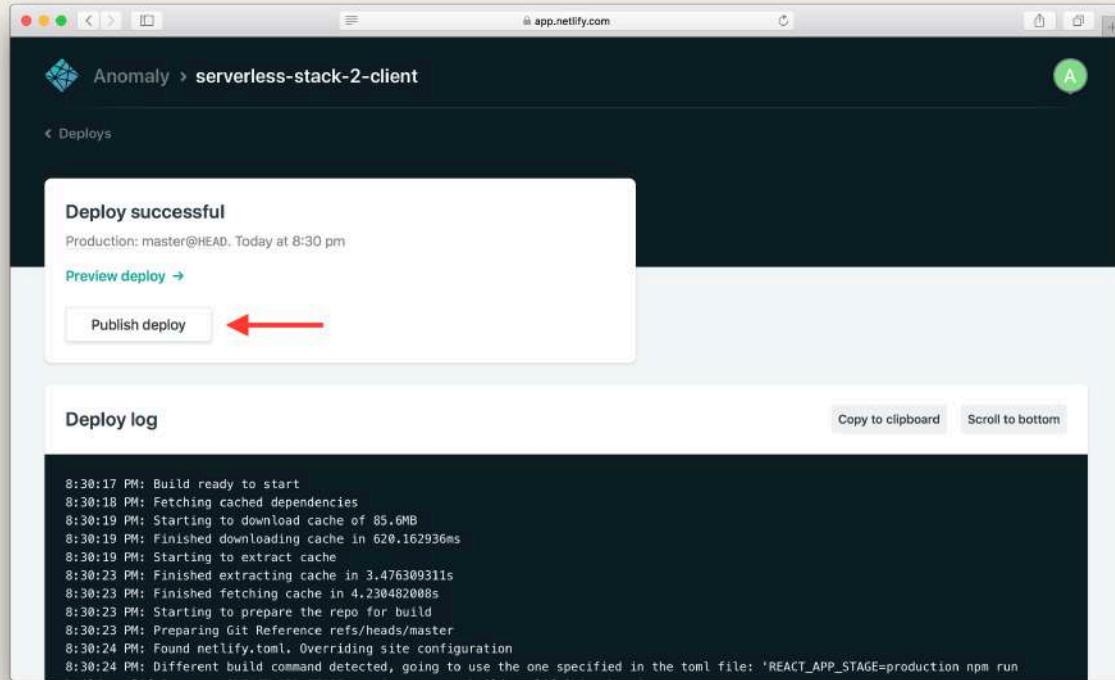
Now for some reason if we aren't happy with the build in production, we can rollback.

Click on the older production deployment.



Click on old production deployment screenshot

And hit **Publish deploy**. This will publish our previous version again.



Publish old production deployment screenshot

And that's it! Now you have an automated workflow for building and deploying your Create React App with serverless.

We are almost ready to wrap things up. But before we do, we want to cover one final really important topic; how to monitor and debug errors when your app is live.



Help and discussion

View the [comments](#) for this chapter on our forums

Monitoring and debugging errors

Debugging Full-Stack Serverless Apps

Now that we are ready to go live with our app, we need to make sure we are setup to monitor and debug errors. This is important because unlike our local environment where we can look at the console (browser or terminal), make changes and fix errors, we cannot do that when our app is live.

Debugging Workflow

We need to make sure we have a couple of things setup before we can confidently ask others to use our app:

- Frontend
 - Be alerted when a user runs into an error.
 - Get all the error details, including the stack trace.
 - In the case of a backend error, get the API that failed.
- Backend
 - Look up the logs for an API endpoint.
 - Get detailed debug logs for all the AWS services.
 - Catch any unexpected errors (out-of-memory, timeouts, etc.).

It's important that we have a good view of our production environments. It allows us to keep track of what our users are experiencing.

Note that, for the frontend the setup is pretty much what you would do for any React application. But we are covering it here because we want to go over the entire debugging workflow. Right from when you are alerted that a user has gotten an error while using your app, all the way till figuring out which Lambda function caused it.

Debugging Setup

Here is what we'll be doing in the next few chapters to help accomplish the above workflow.

- Frontend

On the frontend, we'll be setting up [Sentry](#); a service for monitoring and debugging errors. Sentry has a great free tier that we can use. We'll be integrating it into our React app by reporting any expected errors and unexpected errors. We'll do this by using the [React Error Boundary](#).

- Backend

On the backend, AWS has some great logging and monitoring tools thanks to [CloudWatch](#). We'll be using CloudWatch through the [Seed](#) console. Note that, you can use CloudWatch directly and don't have to rely on Seed for it. We'll also be configuring some debugging helper functions for our backend code.

Looking Ahead

Here's what we'll be going over in the next few chapters:

1. [Setting up error reporting in React](#)
 - [Reporting API errors in React](#)
 - [Reporting unexpected React errors with an Error Boundary](#)
2. [Setting up detailed error reporting in Lambda](#)
3. The debugging workflow for the following Serverless errors:
 - [Logic errors in our Lambda functions](#)
 - [Unexpected errors in our Lambda functions](#)
 - [Errors outside our Lambda functions](#)
 - [Errors in API Gateway](#)

This should give you a good foundation to be able to monitor your app as it goes into production. There are plenty of other great tools out there that can improve on this setup. We want to make sure we cover the basics here. Let's get started!



Help and discussion

View the [comments for this chapter on our forums](#)

Setup Error Reporting in React

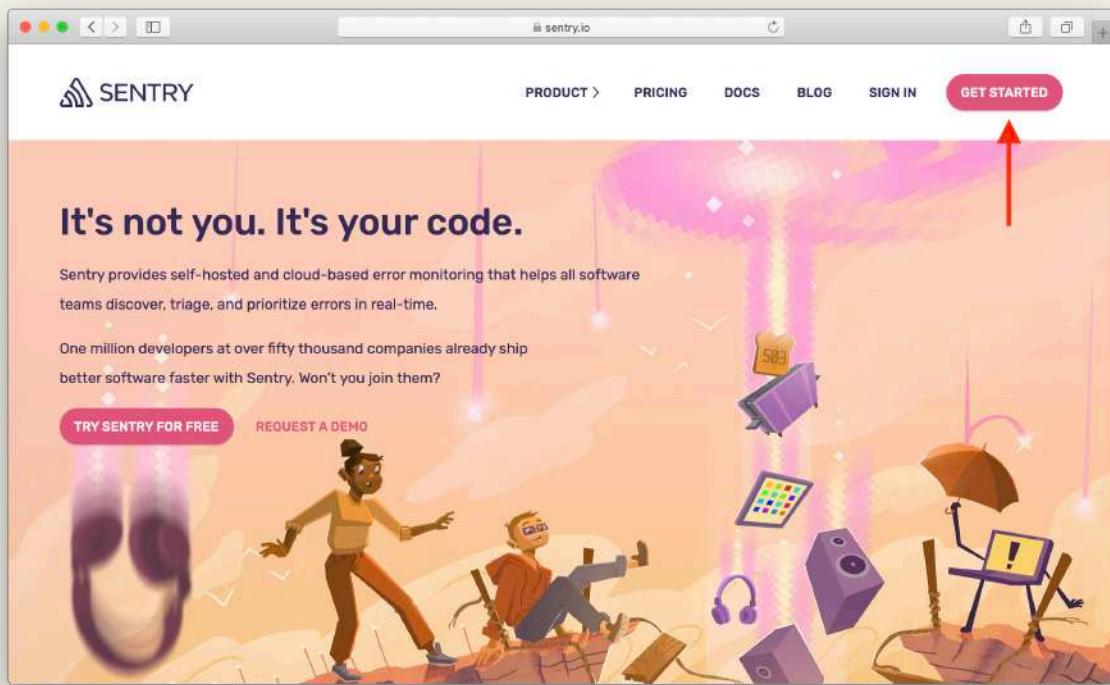
Let's start by setting up error reporting in React. To do so, we'll be using [Sentry](#). Sentry is a great service for reporting and debugging errors. And it comes with a very generous free tier.

In this chapter we'll sign up for a free Sentry account and configure it in our React app. And in the coming chapters we'll be reporting the various frontend errors to it.

Let's get started.

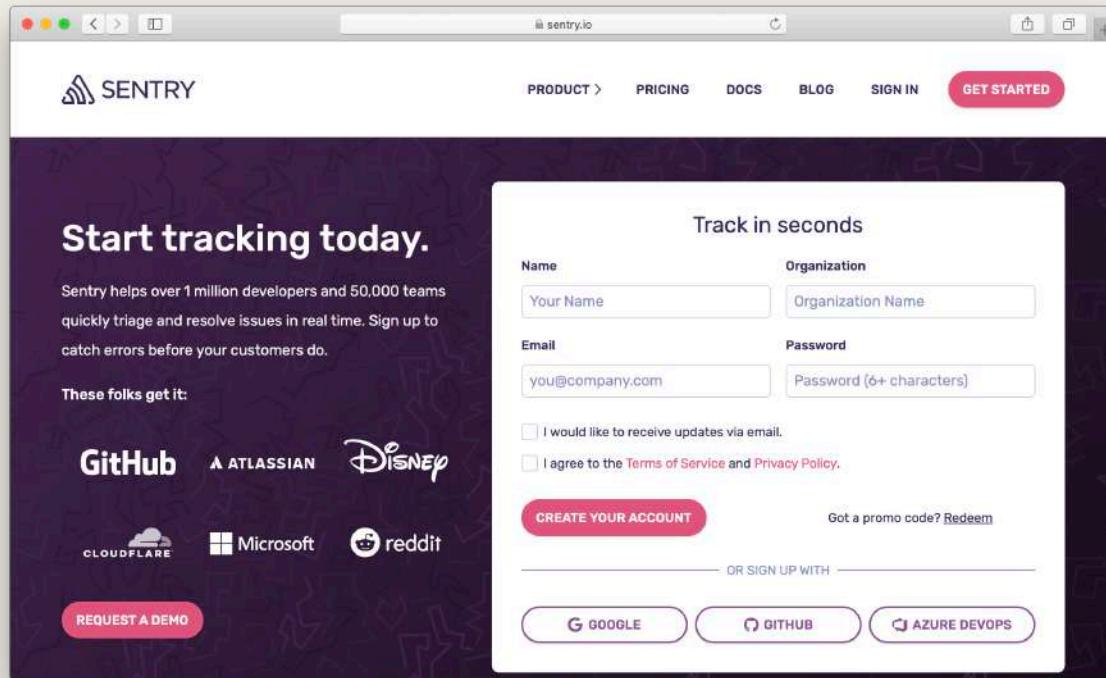
Create a Sentry Account

Head over to [Sentry](#) and hit **Get Started**.



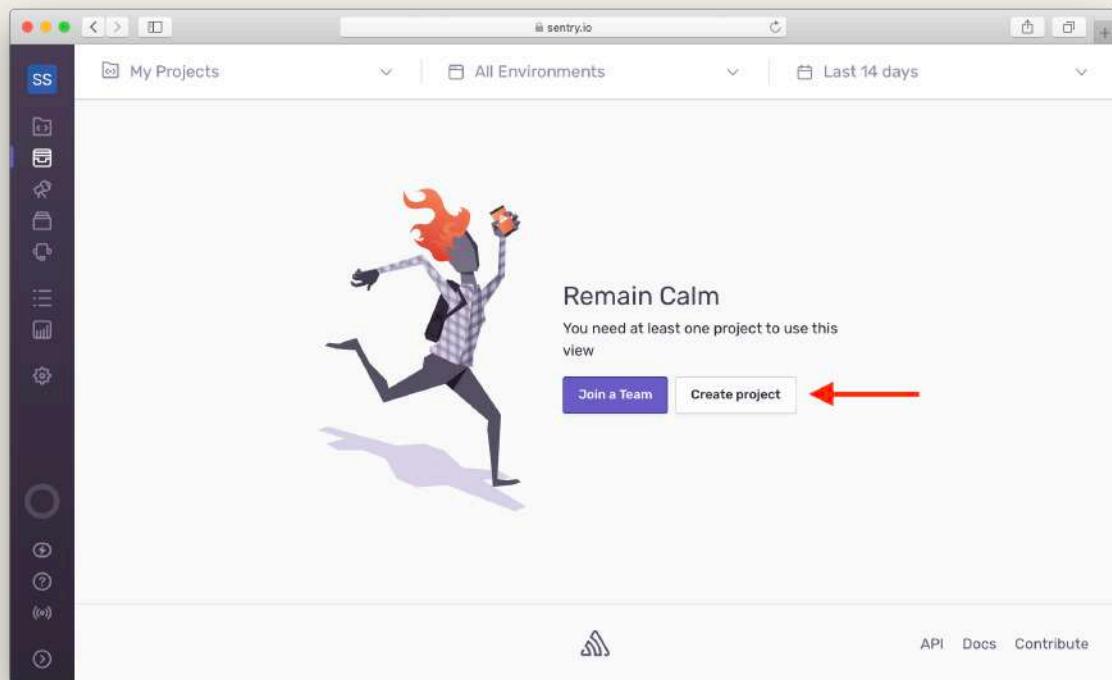
Sentry landing page

Then enter your info and hit **Create Your Account**.



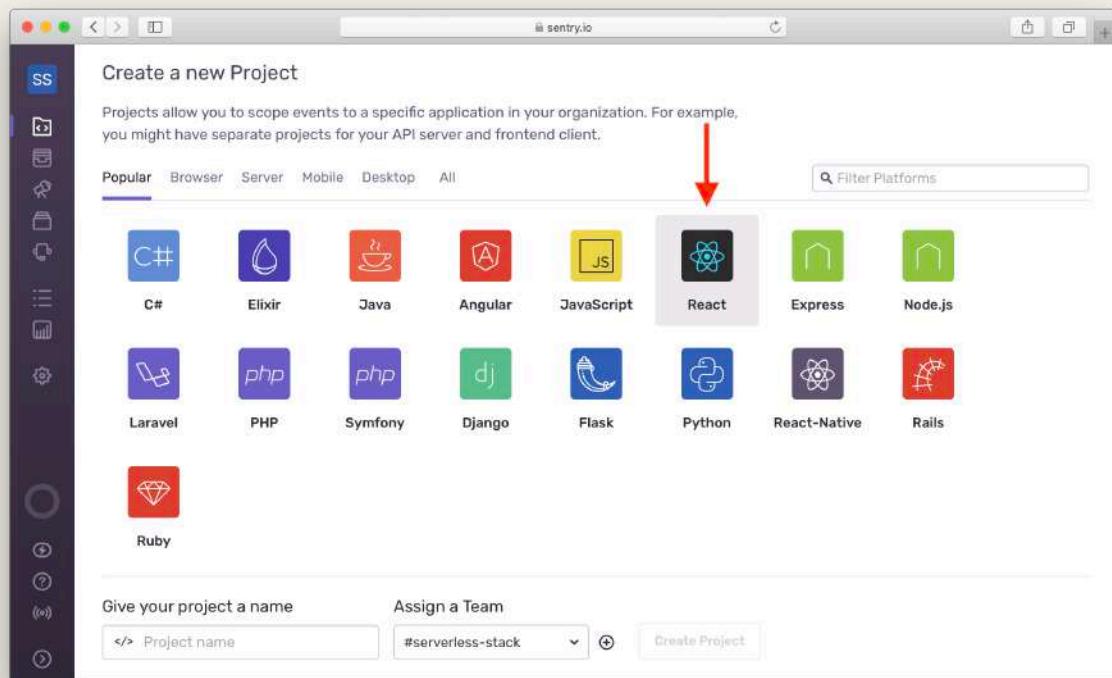
Sentry create an account

Next hit **Create project**.



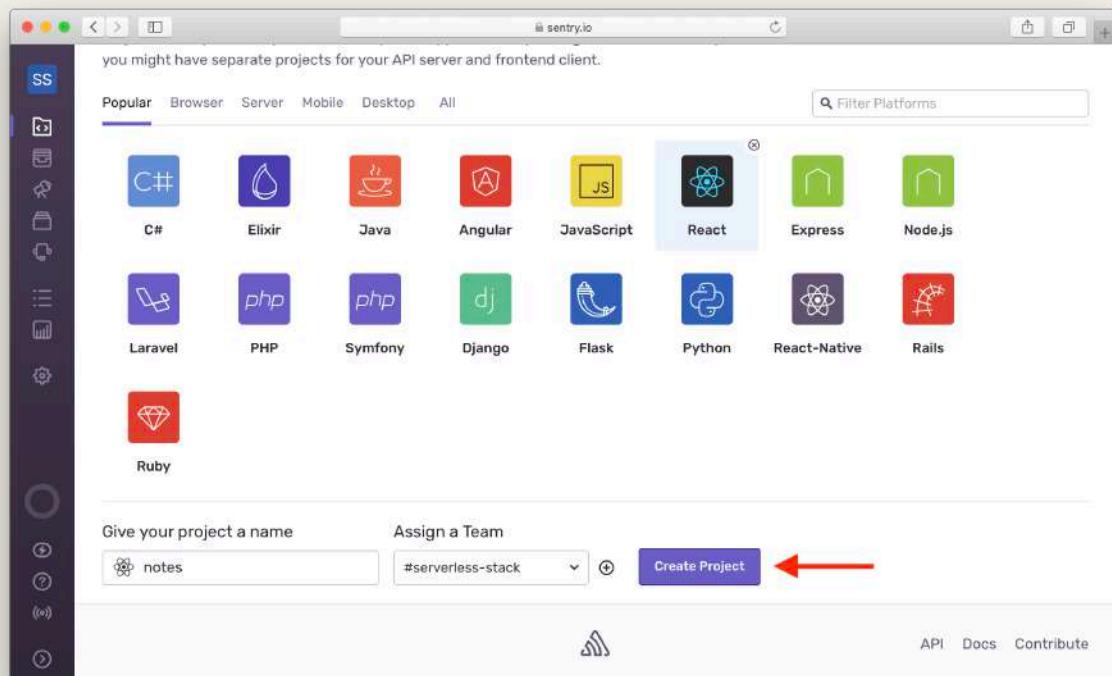
Sentry hit create project

For the type of project, select **React**.



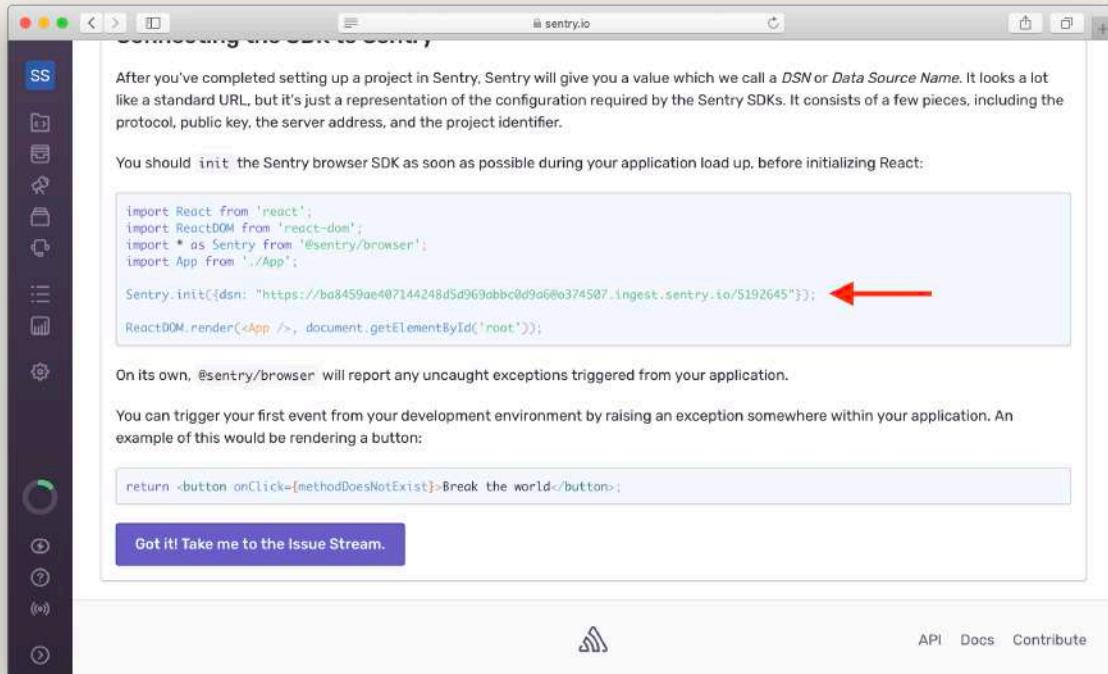
Sentry select React project

Give your project a name.



Sentry name React project

And that's it. Scroll down and copy the `Sentry.init` line.



Sentry init code snippet

Install Sentry

◆ CHANGE Now head over to the project root for your React app and install Sentry.

```
$ npm install @sentry/browser --save
```

We are going to be using Sentry across our app. So it makes sense to keep all the Sentry related code in one place.

◆ CHANGE Add the following to the top of your `src/libs/errorLib.js`.

```
import * as Sentry from "@sentry/browser";  
  
const isLocal = process.env.NODE_ENV === "development";  
  
export function initSentry() {
```

```
if (isLocal) {
  return;
}

Sentry.init({ dsn: "https://your-dsn-id-here@sentry.io/123456" });
}

export function logError(error, errorInfo = null) {
  if (isLocal) {
    return;
  }

  Sentry.withScope((scope) => {
    errorInfo && scope.setExtras(errorInfo);
    Sentry.captureException(error);
  });
}
```

Make sure to replace `Sentry.init({ dsn: "https://your-dsn-id-here@sentry.io/123456" })`; with the line we copied from the Sentry dashboard above.

We are using the `isLocal` flag to conditionally enable Sentry because we don't want to report errors when we are developing locally. Even though we all know that we *rarely* ever make mistakes while developing...

The `logError` method is what we are going to call when we want to report an error to Sentry. It takes:

- An Error object in `error`.
- And, an object with key-value pairs of additional info in `errorInfo`.

Next, let's initialize our app with Sentry.

◆ CHANGE Add the following to the end of the imports in `src/index.js`.

```
import { initSentry } from './libs/errorLib';

initSentry();
```

Now we are ready to start reporting errors in our React app! Let's start with the API errors.

**Help and discussion**

View the [comments for this chapter](#) on our forums

Report API Errors in React

Now that we have our [React app configured with Sentry](#), let's go ahead and start sending it some errors.

So far we've been using the `onError` method in `src/libs/errorLib.js` to handle errors. Recall that it doesn't do a whole lot outside of alerting the error.

```
export function onError(error) {
  let message = error.toString();

  // Auth errors
  if (!(error instanceof Error) && error.message) {
    message = error.message;
  }

  alert(message);
}
```

For most errors we simply alert the error message. But Amplify's Auth package doesn't throw `Error` objects, it throws objects with a couple of properties, including the `message`. So we alert that instead.

For API errors we want to report both the error and the API endpoint that caused the error. On the other hand, for Auth errors we need to create an `Error` object because Sentry needs actual errors sent to it.

◆ **CHANGE** Replace the `onError` method in `src/libs/errorLib.js` with the following:

```
export function onError(error) {
  let errorInfo = {};
  let message = error.toString();

  // Auth errors
```

```
if (!(error instanceof Error) && error.message) {  
    errorInfo = error;  
    message = error.message;  
    error = new Error(message);  
    // API errors  
} else if (error.config && error.config.url) {  
    errorInfo.url = error.config.url;  
}  
  
logError(error, errorInfo);  
  
alert(message);  
}
```

You'll notice that in the case of an Auth error we create an `Error` object and add the object that we get as the `errorInfo`. For API errors, Amplify uses [Axios](#). This has a `config` object that contains the API endpoint that generated the error.

We report this to Sentry by calling `logError(error, errorInfo)` that we added in the [previous chapter](#). And just as before we simply alert the message to the user. It would be a good idea to further customize what you show the user. But we'll leave this as an exercise for you.

This handles all the expected errors in our React app. However, there are a lot of other things that can go wrong while rendering our app. To handle them we are going to setup a [React Error Boundary](#) in the next chapter.



Help and discussion

View the [comments for this chapter on our forums](#)

Setup an Error Boundary in React

In the previous chapter we looked at how to [report API errors to Sentry in our React app](#). Now let's report all those unexpected errors that might happen using a [React Error Boundary](#).

An Error Boundary is a component that allows us to catch any errors that might happen in the child components tree, log those errors, and show a fallback UI.

Create an Error Boundary

It's incredibly straightforward to setup. So let's get started.

◆ **CHANGE** Add the following to `src/components/ErrorBoundary.js`.

```
import React from "react";
import { logError } from "../libs/errorLib";
import "./ErrorBoundary.css";

export default class ErrorBoundary extends React.Component {
  state = { hasError: false };

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    logError(error, errorInfo);
  }

  render() {
    return this.state.hasError ? (
      <div className="ErrorBoundary">
        <span>An error occurred!</span>
      </div>
    ) : this.props.children;
  }
}
```

```
<h3>Sorry there was a problem loading this page</h3>
</div>
) : (
  this.props.children
);
}
}
```

The key part of this component is the `componentDidCatch` and `getDerivedStateFromError` methods. These get triggered when any of the child components have an unhandled error. We set the internal state, `hasError` to true to display our fallback UI. And we report the error to Sentry by calling `logError` with the `error` and `errorInfo` that comes with it.

Let's include some simple styles for this.

◆ CHANGE Create a `src/components/ErrorBoundary.css` file and add:

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

The styles we are using are very similar to our `NotFound` component. We use that when a user navigates to a page that we don't have a route for.

Use the Error Boundary

To use the Error Boundary component that we created, we'll need to add it to our app component.

◆ CHANGE Find the following in `src/App.js`.

```
<ApplicationContext.Provider value={{ isAuthenticated, userHasAuthenticated }}>
  <Routes />
</ApplicationContext.Provider>
```

◆ CHANGE And replace it with:

```
<ErrorBoundary>
  <AppContext.Provider value={{ isAuthenticated, userHasAuthenticated }}>
    <Routes />
  </AppContext.Provider>
</ErrorBoundary>
```

◆ CHANGE Also, make sure to import it in the header of `src/App.js`.

```
import ErrorBoundary from "./components/ErrorBoundary";
```

And that's it! Now an unhandled error in our containers will show a nice error message. While reporting the error to Sentry.

Commit the Changes

◆ CHANGE Let's quickly commit these to Git.

```
$ git add .
$ git commit -m "Adding error reporting"
```

Push the Changes

◆ CHANGE Let's also push these changes to GitHub and deploy our app.

```
$ git push
```

Test the Error Boundary

Before we move on, let's do a quick test.

Replace the following in `src/containers/Home.js`.

```
{isAuthenticated ? renderNotes() : renderLander()}
```

With these faulty lines:

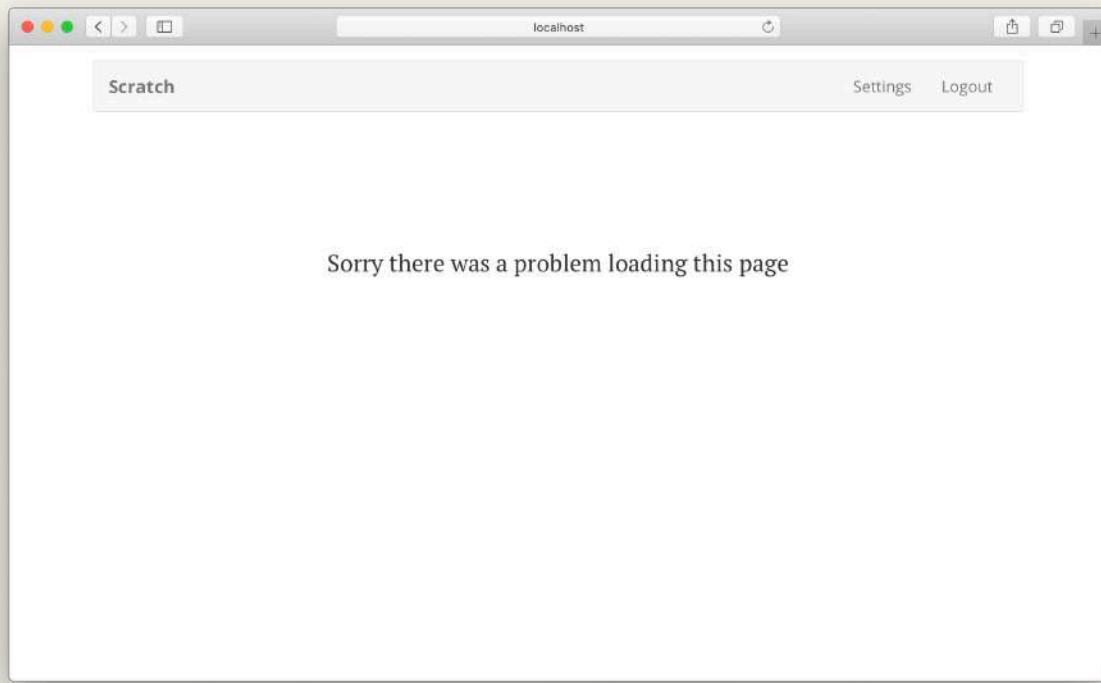
```
{isAuthenticated ? renderNotes() : renderLander()}
{ isAuthenticated.none.no }
```

Now in your browser you should see something like this.



React error message

While developing, React doesn't show your Error Boundary fallback UI by default. To view that, hit the **close** button on the top right.



React Error Boundary fallback UI

Since we are developing locally, we don't report this error to Sentry. But let's do a quick test to make sure it's hooked up properly.

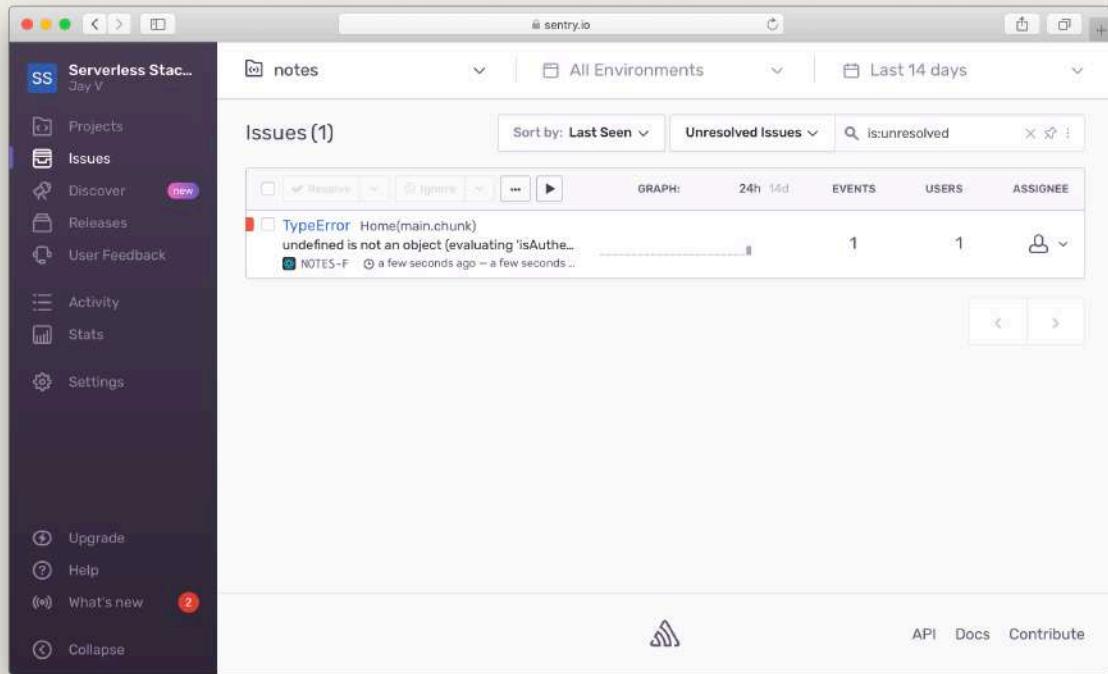
Replace the following from the top of `src/libs/errorLib.js`.

```
const isLocal = process.env.NODE_ENV === "development";
```

With:

```
const isLocal = false;
```

Now if we head over to our browser, we should see the error as before. And we should see the error being reported to Sentry as well! It might take a moment or two before it shows up.



First error in Sentry

And if you click through, you can see the error in detail.

A screenshot of a web browser displaying the Sentry.io interface. The left sidebar shows a project named "Serverless Stac... Jay V" with various sections like Projects, Issues, Discover, Releases, User Feedback, Activity, Stats, and Settings. The main panel has tabs for "BROWSER" and "SDK". Under "BROWSER", it shows "name: Safari" and "version: 13.0.5". Under "SDK", it shows "Name: sentry.javascript.browser" and "Version: 5.15.4". A large box labeled "componentStack" contains a red arrow pointing to a detailed call stack: "in Home (at Routes.js:18), in Route (at Routes.js:18), in Switch (at Routes.js:17), in Routes (at App.js:77), in ErrorBoundary (at App.js:75), in div (at App.js:45), in App (at src/index.js:39), in Router (created by BrowserRouter) in BrowserRouter (at src/index.js:38)". There are "Formatted" and "Raw" buttons at the top right of the componentStack box.

Error details in Sentry

Now our React app is ready to handle the errors that are thrown its way!

Let's cleanup all the testing changes we made above.

```
$ git checkout .
```

Next, let's look at how to handle errors in our Serverless app.



Help and discussion

View the [comments](#) for this chapter on our forums



For reference, here is the complete code for the frontend

[Frontend Source](#)

Setup Error Logging in Serverless

Now that we have our React app configured to report errors, let's move on to our Serverless backend. Our React app is reporting API errors (and other unexpected errors) with the API endpoint that caused the error. We want to use that info to be able to debug on the backend and figure out what's going on.

To do this, we'll setup the error logging in our backend to catch:

- Errors in our code
- Errors while calling AWS services
- Unexpected errors like Lambda functions timing out or running out of memory

We are going to look at how to setup a debugging framework to catch the above errors, and have enough context for us to easily pinpoint and fix the issue. We'll be using [CloudWatch](#) to write our logs, and we'll be using the log viewer in [Seed](#) to view them.

Setup a Debug Lib

Let's start by adding some code to help us with that.

◆ **CHANGE** Create a `services/notes/libs/debug-lib.js` file and add the following to it.

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

let logs;

// Log AWS SDK calls
AWS.config.logger = { log: debug };

export default function debug() {
  logs.push({
```

```
    date: new Date(),
    string: util.format.apply(null, arguments),
})};

export function init(event, context) {
  logs = [];

  // Log API event
  debug("API event", {
    body: event.body,
    pathParameters: event.pathParameters,
    queryStringParameters: event.queryStringParameters,
  });
}

export function flush(e) {
  logs.forEach(({ date, string }) => console.debug(date, string));
  console.error(e);
}
```

We are doing a few things of note in this simple helper.

- **Enable AWS SDK logging**

We start by enabling logging for the AWS SDK. We do so by running `AWS.config.logger = { log: debug }`. This is telling the AWS SDK to log using our logger, the `debug()` method (we'll look at this below). So when you make a call to an AWS service, ie. a query call to the DynamoDB table `dev-notes`, this will log:

```
[AWS dynamodb 200 0.296s 0 retries] query({ TableName: 'dev-notes',
KeyConditionExpression: 'userId = :userId', ExpressionAttributeValues:
{ ':userId': { S: 'USER-SUB-1234' } } }) Note, we only want to log this info when there is an error. We'll look at how we accomplish this below.
```

- **Log API request info**

We initialize our debugger by calling `init()`. We log the API request info, including the path parameters, query string parameters, and request body. We do so using our internal `debug()` method.

- **Log only on error**

We log messages using our special `debug()` method. Debug messages logged using this method only get printed out when we call the `flush()` method. This allows us to log very detailed contextual information about what was being done leading up to the error. We can log:

- Arguments and return values for function calls.
- And, request/response data for HTTP requests made.

We only want to print out debug messages to the console when we run into an error. This helps us reduce clutter in the case of successful requests. And, keeps our CloudWatch costs low!

To do this, we store the log info (when calling `debug()`) in memory inside the `logs` array. And when we call `flush()` (in the case of an error), we `console.debug()` all those stored log messages.

So in our Lambda function code, if we want to log some debug information that only gets printed out if we have an error, we'll do the following:

```
import debug from "../libs/debug-lib";  
  
debug('This stores the message and prints to CloudWatch if Lambda function  
↳ later throws an exception');
```

In contrast, if we always want to log to CloudWatch, we'll:

```
console.log('This prints a message in CloudWatch prefixed with INFO');  
console.warn('This prints a message in CloudWatch prefixed with WARN');  
console.error('This prints a message in CloudWatch prefixed with ERROR');
```

Now let's use the debug library in our Lambda functions.

Setup Handler Lib

You'll recall that all our Lambda functions are wrapped using a `handler()` method. We use this to format what our Lambda functions return as their HTTP response. It also, handles any errors that our Lambda functions throws.

We'll use the debug lib that we added above to improve our error handling.

◆ CHANGE Replace our services/notes/libs/handler-lib.js with the following.

```
import * as debug from "./debug-lib";

export default function handler(lambda) {
  return async function (event, context) {
    let body, statusCode;

    // Start debugger
    debug.init(event, context);

    try {
      // Run the Lambda
      body = await lambda(event, context);
      statusCode = 200;
    } catch (e) {
      // Print debug messages
      debug.flush(e);

      body = { error: e.message };
      statusCode = 500;
    }

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

This should be fairly straightforward:

1. We initialize our debugger by calling `debug.init()`.
2. We run our Lambda function.
3. We format the success response.
4. In the case of an error, we first write out our debug logs by calling `debug.flush(e)`. Where `e` is the error that caused our Lambda function to fail.
5. We format and return our HTTP response.

Using the Error Handler

You might recall the way we are currently using the above error handler in our Lambda functions.

```
import handler from "./libs/handler-lib";  
  
export const main = handler((event, context) => {  
    // Do some work  
    const a = 1 + 1;  
    // Return a result  
    return { result: a };  
});
```

We wrap all of our Lambda functions using the error handler.

Note that, the `handler-lib.js` needs to be **imported before we import anything else**. This is because the `debug-lib.js` that it imports needs to initialize AWS SDK logging before it's used anywhere else.

Commit the Code

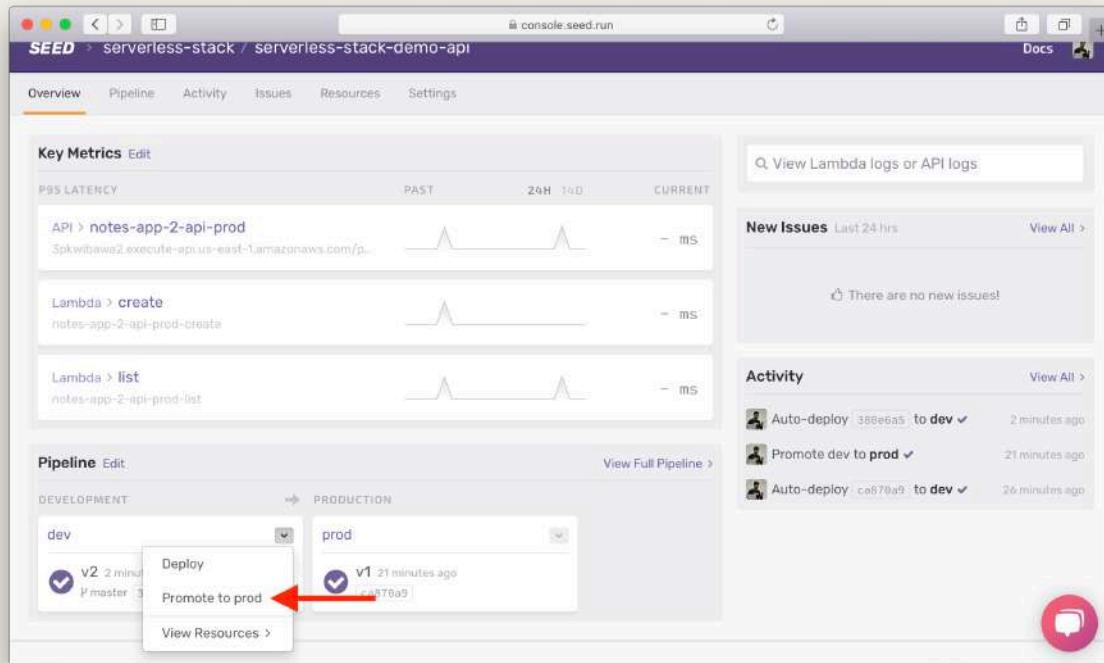
Let's push our changes

◆ CHANGE Let's commit the code we have so far.

```
$ git add .  
$ git commit -m "Adding error logging"  
$ git push
```

And promote the changes to production.

Head over to the Seed console and hit **Promote to prod** once your changes are deployed to dev.

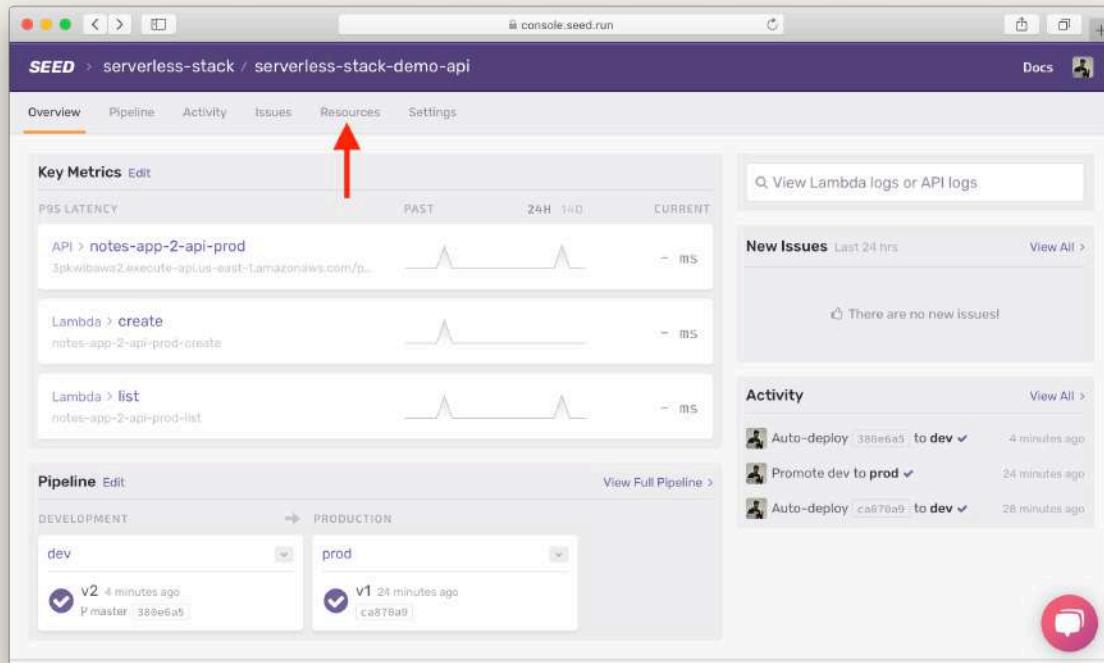


Promote error logging to prod in Seed

Enable Access Logs

The combination of our new error handler and Lambda logs will help us catch most of the errors. However, we can run into errors that don't make it to our Lambda functions. To debug these errors we'll need to look at the API Gateway logs. So let's go head and enable access logs for our API.

From the dashboard for your app on Seed, select the **Resources** tab.



Click Resources tab in Seed dashboard

Then hit **Enable Access Logs** for your API.

The screenshot shows the SEED console interface for a 'serverless-stack-demo-api' service. On the left, there's a sidebar with 'Overview', 'Pipeline', 'Activity', 'Issues', 'Resources' (which is selected), and 'Settings'. Under 'Resources', there are two environments: 'prod' (selected) and 'dev'. In the 'prod' environment, there's a 'notes' service. A note says 'Access logs are not enabled for this service. Enable Access Logs'. To the right of this note is a red arrow pointing to the 'Access Logs' link. Below this, there's a table for 'LAMBDAS' with several API endpoints listed. At the bottom, there's a 'ALL DEPLOYED SERVICES' section and a 'notes' icon.

Enable access logs in Seed

And that's pretty much it! With these simple steps, we are now ready to look at some examples of how to debug our Serverless app.



Help and discussion

View the [comments](#) for this chapter on our forums



For reference, here is the complete code for the backend

[Backend Source](#)

Logic Errors in Lambda Functions

Now that we've [setup error logging for our API](#), we are ready to go over the workflow for debugging the various types of errors we'll run into.

First up, there are errors that can happen in our Lambda function code. Now we all know that we almost never make mistakes in our code. However, it's still worth going over this very "*unlikely*" scenario.

Create a New Branch

Let's start by creating a new branch that we'll use while working through the following examples.

◆ **CHANGE** In the project root for your backend repo, run the following:

```
$ git checkout -b debug
```

Push Some Faulty Code

Let's trigger an error in `get.js` by commenting out the `noteId` field in the DynamoDB call's Key definition. This will cause the DynamoDB call to fail and in turn cause the Lambda function to fail.

◆ **CHANGE** Replace `services/notes/get.js` with the following.

```
import handler from "./libs/handler-lib";
import dynamoDb from "./libs/dynamodb-lib";

export const main = handler(async (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
}
};

const result = await dynamoDb.get(params);
if (!result.Item) {
  throw new Error("Item not found.");
}

// Return the retrieved item
return result.Item;
});
```

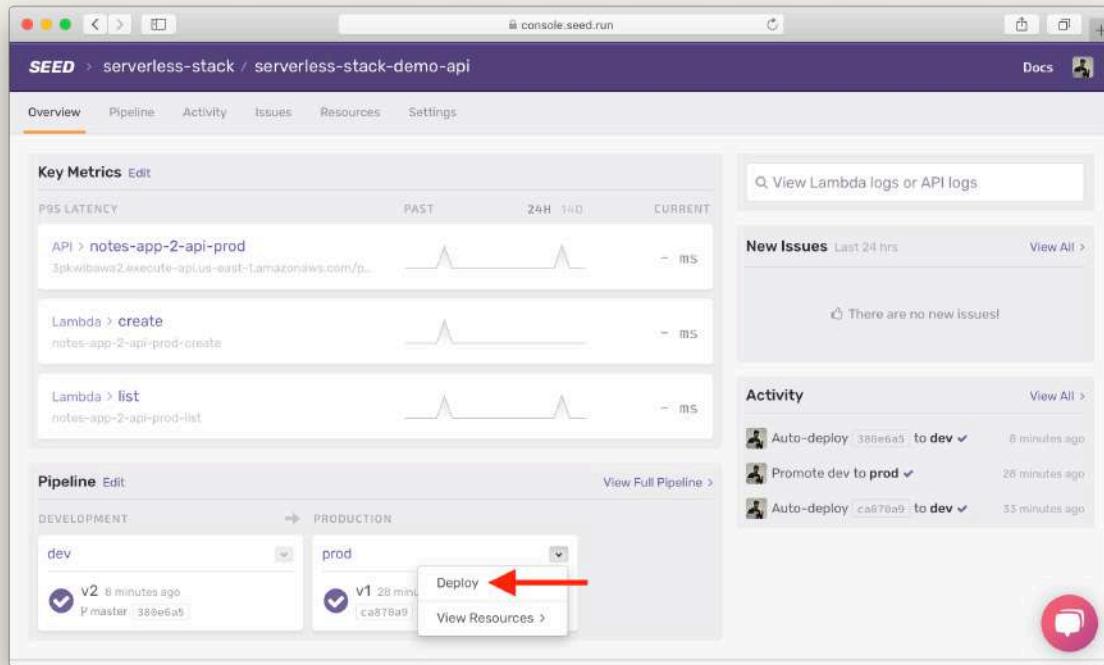
Note the line that we've commented out.

◆ CHANGE Let's commit our changes.

```
$ git add .
$ git commit -m "Adding some faulty code"
$ git push --set-upstream origin debug
```

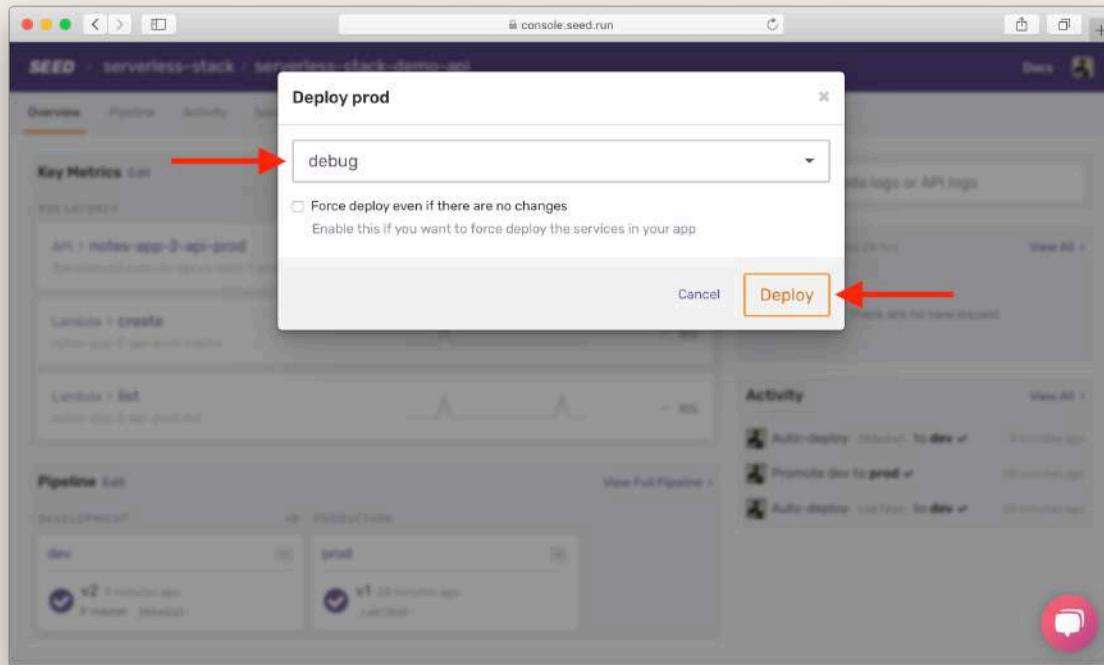
Deploy the Faulty Code

Head over to your Seed dashboard and select the **prod** stage in the pipeline and hit **Deploy**.



Click deploy in Seed pipeline

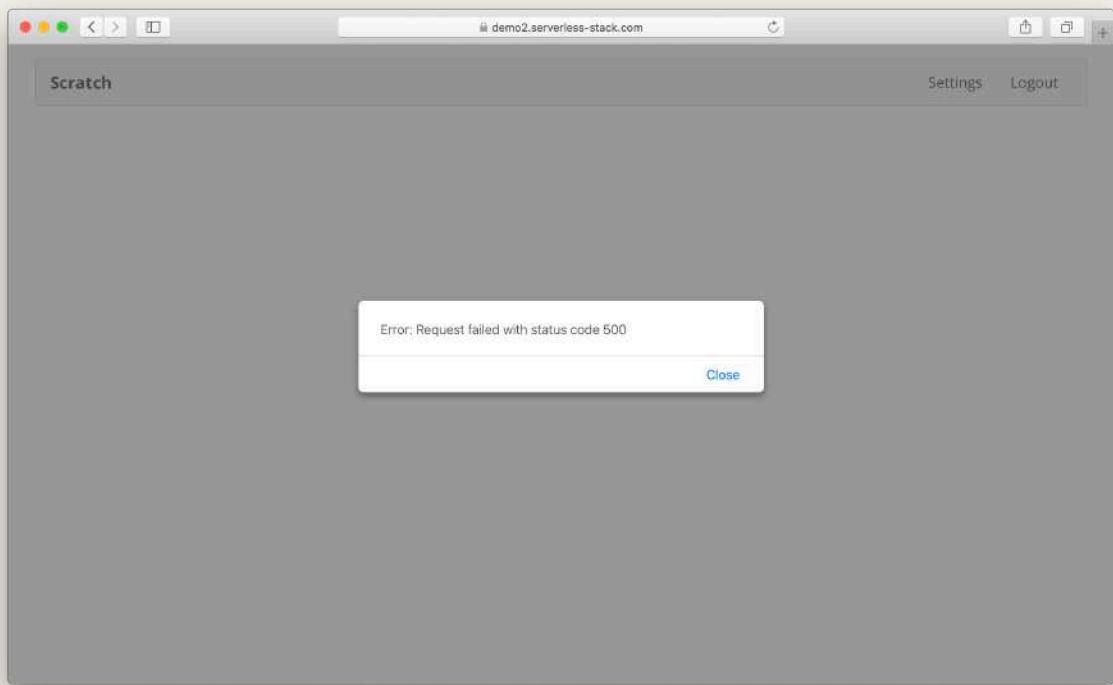
Select the **debug** branch from the dropdown and hit **Deploy**.



Select branch and confirm deploy in Seed

This will deploy our faulty code to production.

Head over on to your notes app, and select a note. You'll notice the page fails to load with an error alert.



Error alert in notes app note page

Debug Logic Errors

To start with, you should get an email from Sentry about this error. Go to Sentry and you should see the error showing at the top. Select the error.

A screenshot of the Sentry.io web interface. The top navigation bar shows the URL 'sentry.io' and the project name 'notes'. Below the navigation is a search bar with filters: 'All Environments' and 'Last 14 days'. The main area is titled 'Issues (1)' and contains a single entry:

Category	Details
Error	r(src/helpers) Request failed with status code 500 NOTES-R a few seconds ago – a few seconds old

A red arrow points upwards from the bottom of the page towards the error entry in the list.

New network error in Sentry

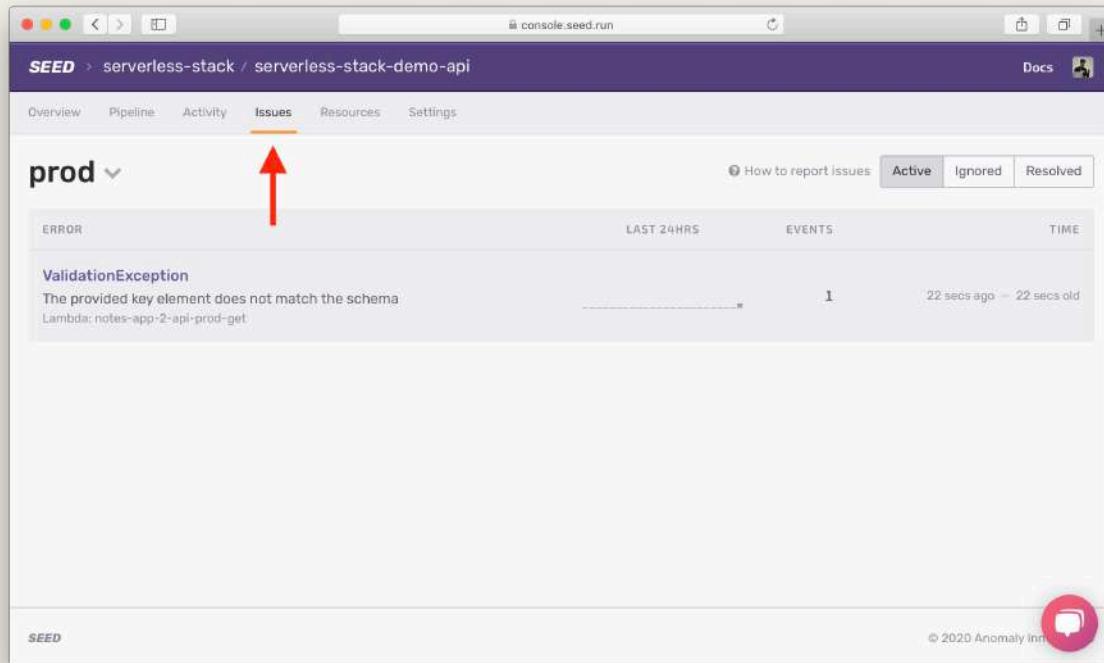
You'll see that our frontend error handler is logging the API endpoint that failed.

The screenshot shows the Sentry interface for a project named "Serverless Stack". On the left, there's a sidebar with options like Projects, Issues (which is selected), Discover, Releases, User Feedback, Activity, Stats, and Settings. The main area displays environment details:

- Mac OS X**:
 - Name: Mac OS X
 - Version: 10.14.6
- BROWSER**:
 - name: Chrome
 - version: 80.0.3987
- ADDITIONAL DATA**:
 - url: <https://api.serverless-stack.seed-demo.club/prod/notes/9c130ef0-11ea-b6f5-7bf68f1b7984> (with a red arrow pointing to it)
 - Formatted Raw
- SDK**:
 - Name: sentry.javascript.browser
 - Version: 5.15.4
- Ownership Rules**:
 - Create Ownership Rule

Error details in Sentry

You'll also get an email from Seed telling you that there was an error in your Lambda functions. If you click on the **Issues** tab you'll see the error at the top.



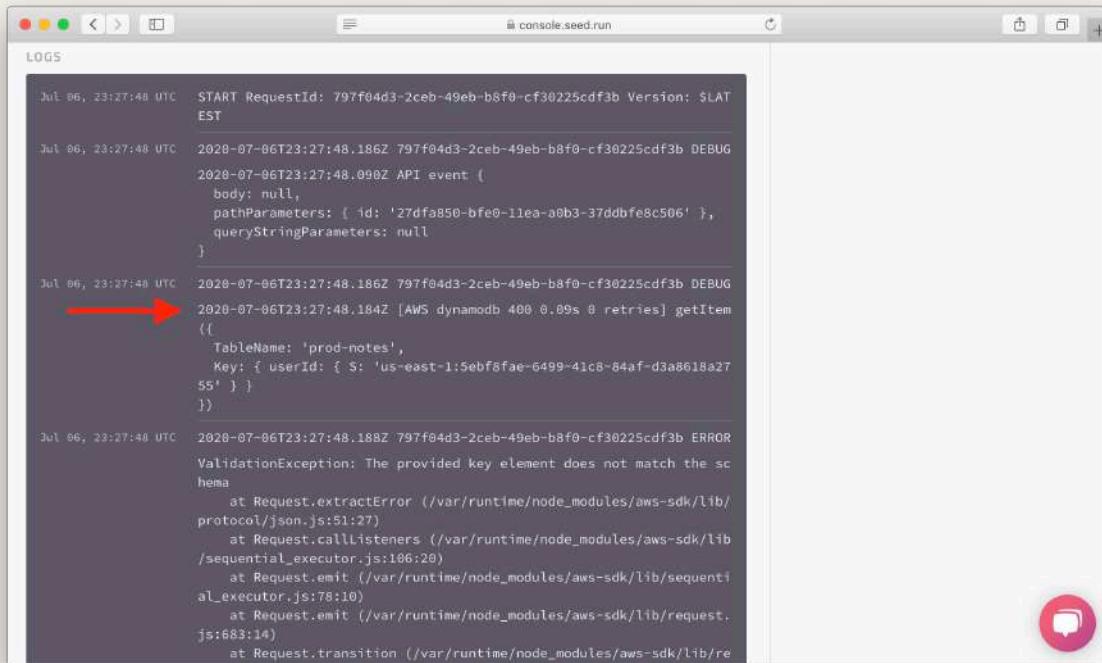
[View Issues in Seed](#)

And if you click on the error, you'll see the error message and stack trace.

The screenshot shows the SEED console interface for a project named 'serverless-stack' under the 'serverless-stack-demo-api' service. The 'Issues' tab is selected, showing a single 'ValidationException' entry. The error message is: 'The provided key element does not match the schema'. The stack trace details the call stack from the AWS SDK's Request class. The right side of the screen displays a timeline of events: 'Jul 6, 2020, 11:27:48 PM UTC' (1 event), 'LAST 24 HOURS', 'LAST 30 DAYS', 'FIRST SEEN' (31 secs ago, Jul 6, 2020, 11:27:48 PM UTC, Build v2), and 'LAST SEEN' (31 secs ago, Jul 6, 2020, 11:27:48 PM UTC). A 'View build info' link and a pink speech bubble icon are also present.

Error details in Seed

If you scroll down a bit further you'll notice the entire request log. Including debug messages from the AWS SDK as it tries to call DynamoDB.



```
LOGS
Jul 06, 23:27:48 UTC START RequestId: 797f04d3-2ceb-49eb-b8f0-cf30225cdf3b Version: $LATEST
Jul 06, 23:27:48 UTC 2020-07-06T23:27:48.186Z 797f04d3-2ceb-49eb-b8f0-cf30225cdf3b DEBUG
2020-07-06T23:27:48.090Z API event {
  body: null,
  pathParameters: { id: '27dfa850-bfe0-11ea-a0b3-37ddbfe8c506' },
  queryStringParameters: null
}
Jul 06, 23:27:48 UTC 2020-07-06T23:27:48.186Z 797f04d3-2ceb-49eb-b8f0-cf30225cdf3b DEBUG
2020-07-06T23:27:48.184Z [AWS dynamodb 400 0.09s 0 retries] getItem
(
  TableName: 'prod-notes',
  Key: { userId: { S: 'us-east-1:5ebf8fae-6499-41c8-84af-d3a8618a2755' } }
)
Jul 06, 23:27:48 UTC 2020-07-06T23:27:48.188Z 797f04d3-2ceb-49eb-b8f0-cf30225cdf3b ERROR
ValidationException: The provided key element does not match the schema
  at Request.extractError (/var/runtime/node_modules/aws-sdk/lib/protocol/json.js:51:27)
  at Request.callListeners (/var/runtime/node_modules/aws-sdk/lib/sequential_executor.js:106:20)
  at Request.emit (/var/runtime/node_modules/aws-sdk/lib/sequential_executor.js:78:10)
  at Request.emit (/var/runtime/node_modules/aws-sdk/lib/request.js:683:14)
  at Request.transition (/var/runtime/node_modules/aws-sdk/lib/re
```

Lambda request log in error details in Seed

The message `The provided key element does not match the schema`, says that there is something wrong with the Key that we passed in. Our debug messages helped guide us to the source of the problem!

Next let's look at how we can debug unexpected errors in our Lambda functions.



Help and discussion

View the [comments](#) for this chapter on our forums

Unexpected Errors in Lambda Functions

Previously, we looked at [how to debug errors in our Lambda function code](#). In this chapter let's look at how to debug some unexpected errors. Starting with the case of a Lambda function timing out.

Debugging Lambda Timeouts

Our Lambda functions often make API requests to interact with other services. In our notes app, we talk to DynamoDB to store and fetch data; and we also talk to Stripe to process payments. When we make an API request, there is the chance the HTTP connection times out or the remote service takes too long to respond. We are going to look at how to detect and debug the issue. The default timeout for Lambda functions are 6 seconds. So let's simulate a timeout using `setTimeout`.

◆ CHANGE Replace our `services/notes/get.js` with the following:

```
import handler from "./libs/handler-lib";
import dynamoDb from "./libs/dynamodb-lib";

export const main = handler(async (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
    }
};
```

```
const result = await dynamoDb.get(params);
if (!result.Item) {
  throw new Error("Item not found.");
}

// Set a timeout
await new Promise(resolve => setTimeout(resolve, 10000));

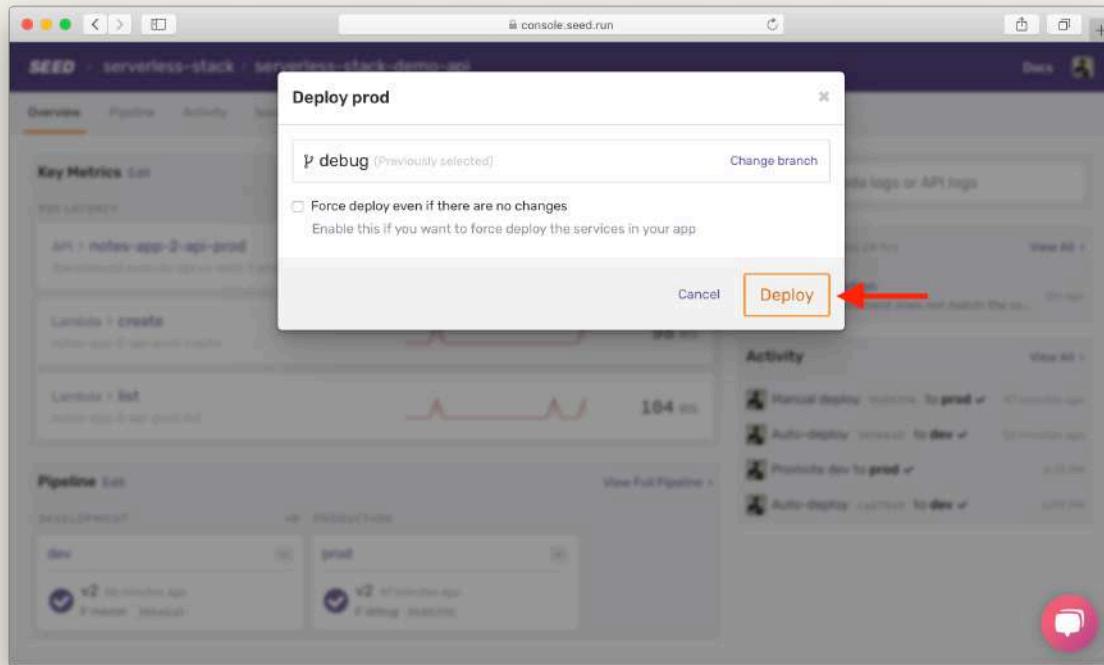
// Return the retrieved item
return result.Item;
});
```



Let's commit this code.

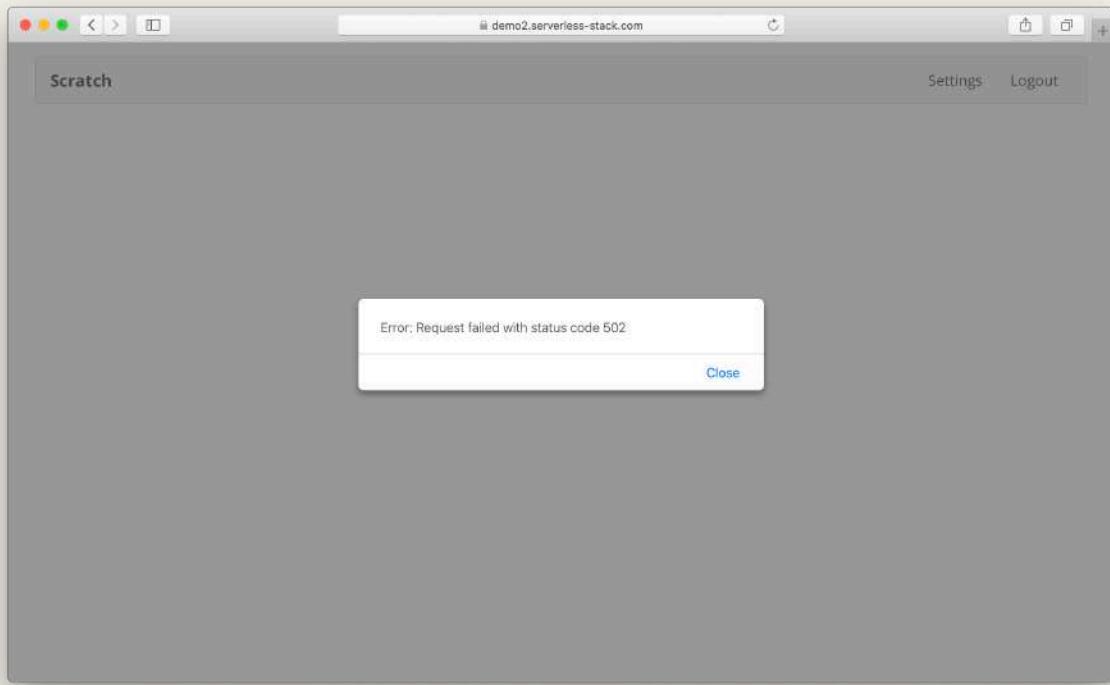
```
$ git add .
$ git commit -m "Adding a timeout"
$ git push
```

Head over to your Seed dashboard, select the **prod** stage in the pipeline and deploy the debug branch.



Deploy debug branch in Seed

On your notes app, try and select a note. You will notice the page tries to load for a couple of seconds, and then fails with an error alert.



Timeout error in notes app note page

You'll get an error alert in Sentry. And if you head over to the **Issues** tab in Seed you'll notice a new error — `Lambda Timeout Error`.

If you click on the new error, you'll notice that the request took `6006.18ms`. And since the Lambda timeout is 6 seconds by default. This means that the function timed out.

The screenshot shows the Seed console interface for a Lambda function named "notes-app-2-api-get". The "ERROR MESSAGE" section displays the text "notes-app-2-api-get timed out after 6.01 seconds". The "TAGS" section includes "lambda", "notes-app-2-api-prod-get", "service", "notes", "stage", and "prod". The "STACK TRACE" section shows "No stack trace available.". The "LOGS" section contains the following entries:

- Jul 06, 23:32:55 UTC START RequestId: 6ce84a25-68ef-4d59-8afe-0e9e49567303 Version: \$LATEST
- Jul 06, 23:33:01 UTC END RequestId: 6ce84a25-68ef-4d59-8afe-0e9e49567303
- Jul 06, 23:33:01 UTC REPORT RequestId: 6ce84a25-68ef-4d59-8afe-0e9e49567303 Duration: 60.628 ms Billed Duration: 6000 ms Memory Size: 1024 MB Max. Memory Used: 85 MB Init Duration: 603.10 ms
- Jul 06, 23:33:01 UTC 2020-07-06T23:33:01.935Z 6ce84a25-68ef-4d59-8afe-0e9e49567303 Task timed out after 6.01 seconds

To the right of the logs, there is a timeline showing "LAST 24 HOURS" and "LAST 30 DAYS". Below the logs, the "FIRST SEEN" section shows a single event from "24 secs ago" on "Jul 6, 2020, 11:33:01 PM UTC" for "Build v3". A "View build info" link is provided. At the bottom right is a pink speech bubble icon with the text "LAMBDA FUNCTIONS".

Timeout error details in Seed

To drill into this issue further, add a `console.log` in your Lambda function. This messages will show in the request log and it'll give you a sense of where the timeout is taking place.

Next let's look at what happens when our Lambda function runs out of memory.

Debugging Out of Memory Errors

By default, a Lambda function has 1024MB of memory. You can assign any amount of memory between 128MB and 3008MB in 64MB increments. So in our code, let's try and allocate more memory till it runs out of memory.

◆ CHANGE Replace your `get.js` with:

```
import handler from "./libs/handler-lib";
import dynamoDb from "./libs/dynamodb-lib";

function allocMem() {
```

```

let bigList = Array(4096000).fill(1);
return bigList.concat(allocMem());
}

export const main = handler(async (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
    }
  };
};

const result = await dynamoDb.get(params);
if (!result.Item) {
  throw new Error("Item not found.");
}

allocMem();

// Return the retrieved item
return result.Item;
});

```

Now we'll set our Lambda function to use the lowest memory allowed and make sure it has time to allocate the memory.

◆ CHANGE Replace the get function block in your services/notes/serverless.yml.

```

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
  memorySize: 128

```

```
timeout: 10
events:
  - http:
    path: notes/{id}
    method: get
    cors: true
    authorizer: aws_iam
```



Let's commit this.

```
$ git add .
$ git commit -m "Adding a memory error"
$ git push
```

Head over to your Seed dashboard and deploy it. Then, in your notes app, try and load a note. It should fail with an error alert.

Just as before, you'll see the error in Sentry. And head over to new issue in Seed.

The screenshot shows the Seed dashboard interface. At the top, there's a navigation bar with tabs like 'prod', 'staging', and 'development'. Below the navigation, a modal window is open titled 'Lambda Runtime Error'.

ERROR MESSAGE: notes-app-2-api-get exited with error: signal: killed
Runtime.ExitError

TAGS: lambda, notes-app-2-api-prod-get, service, notes, stage, prod

STACK TRACE: No stack trace available.

LOGS:

```
Jul 06, 23:37:24 UTC START RequestId: cfc52191-e73d-4152-b61c-84d07c6e3038 Version: $LATEST
Jul 06, 23:37:31 UTC END RequestId: cfc52191-e73d-4152-b61c-84d07c6e3038
Jul 06, 23:37:31 UTC REPORT RequestId: cfc52191-e73d-4152-b61c-84d07c6e3038 Duration: 6843.70 ms Billed Duration: 6900 ms Memory Size: 128 MB Max Memory Used: 128 MB Init Duration: 342.02 ms
Jul 06, 23:37:31 UTC RequestId: cfc52191-e73d-4152-b61c-84d07c6e3038 Error: Runtime exited with error: signal: killed
```

LAST 24 HOURS and **LAST 30 DAYS** buttons are visible on the right.

FIRST SEEN: 8 secs ago, Jul 6, 2020, 11:37:31 PM UTC, Build v4, View build info >

LAST SEEN: 8 secs ago, Jul 6, 2020, 11:37:31 PM UTC, Build v4, View build info >

LAMBDA FUNCTIONS: notes-app-2-api-prod-get (1)

Memory error details in Seed

Note the request took all of 128MB of memory. Click to expand the request.

You'll see exited with error: signal: killed Runtime.ExitError. This is printed out by Lambda runtime indicating the runtime was killed. This means that you should give your function more memory or that your code is leaking memory.

Next, we'll look at how to debug errors that happen outside your Lambda function handler code.



Help and discussion

View the [comments for this chapter on our forums](#)

Errors Outside Lambda Functions

We've covered debugging [errors in our code](#) and [unexpected errors](#) in Lambda functions. Now let's look at how to debug errors that happen outside our Lambda functions.

Initialization Errors

Lambda functions could fail not because of an error inside your handler code, but because of an error outside it. In this case, your Lambda function won't be invoked. Let's add some faulty code outside our handler function.

◆ **CHANGE** Replace our `services/notes/get.js` with the following.

```
import handler from "./libs/handler-lib";
import dynamoDb from "./libs/dynamodb-lib";

// Some faulty code
dynamoDb.notExist();

export const main = handler(async (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
    }
  };

  const result = await dynamoDb.get(params);
```

```
if (!result.Item) {
  throw new Error("Item not found.");
}

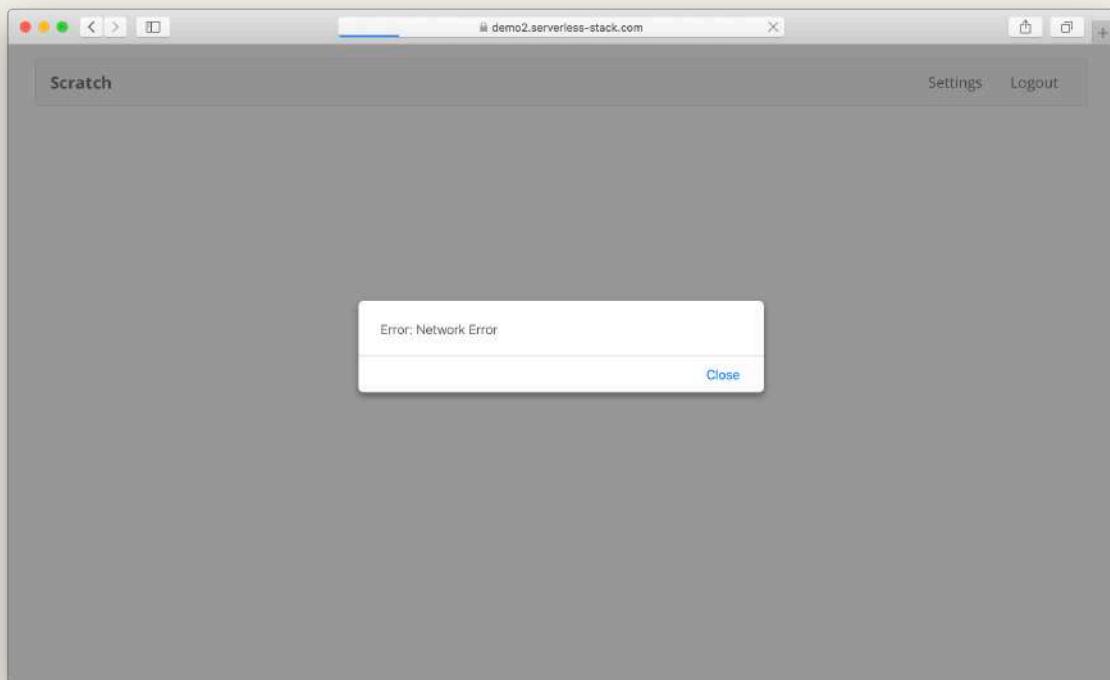
// Return the retrieved item
return result.Item;
});
```

◆ CHANGE Commit this code.

```
$ git add .
$ git commit -m "Adding an init error"
$ git push
```

Head over to your Seed dashboard, and deploy it.

Now if you select a note in your notes app, you'll notice that it fails with an error.



Init error in notes app note page

You should see an error in Sentry. And if you head over to the Issues in Seed and click on the new error.

The screenshot shows the Sentry interface for a 'prod' environment. A new error titled 'TypeError' has been identified. The error message is 'dynamodb_lib.notExist is not a function'. The stack trace details the execution path through various modules like webpack, loader.js, and helpers.js. The logs section shows a single log entry from July 6, 2020, at 11:41:40 PM UTC, indicating an 'Uncaught Exception'. On the right side, a timeline shows the error was first seen 1 minute ago and last seen 1 minute ago. It also lists 3 events related to this issue. A sidebar indicates there are 3 Lambda functions associated with this error.

Init error details in Seed

You'll notice the error message `dynamodb_lib.notExist is not a function`.

Note that, you might see there are 3 events for this error. This is because the Lambda runtime prints out the error message multiple times.

Handler Function Errors

Another error that can happen outside a Lambda function is when the handler has been misnamed.

◆ CHANGE Replace our `get.js` with the following.

```
import handler from "./libs/handler-lib";
import * as dynamoDbLib from "./libs/dynamodb-lib";
```

```
// Wrong handler function name
export const main2 = handler(async (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
    }
  };

  const result = await dynamoDbLib.call("get", params);
  if (!result.Item) {
    throw new Error("Item not found.");
  }

  // Return the retrieved item
  return result.Item;
});
```

◆ CHANGE Let's commit this.

```
$ git add .
$ git commit -m "Adding a handler error"
$ git push
```

Head over to your Seed dashboard and deploy it. Then, in your notes app, try and load a note. It should fail with an error alert.

Just as before, you'll see the error in Sentry. Head over to the new error in Seed.

The screenshot shows the Seed console interface for a 'prod' environment. A modal window is open for a Lambda function named 'notes-app-2-api-prod-get'. The title of the modal is 'Runtime.HandlerNotFound'. The 'ERROR MESSAGE' section contains the text 'get.main is undefined or not exported'. The 'STACK TRACE' section shows a detailed stack trace starting with 'Runtime.HandlerNotFound: get.main is undefined or not exported' and listing several internal module loading steps. The 'LOGS' section shows a single log entry from July 6, 2020, at 23:46:24 UTC, which includes the error message: 'Jul 06, 23:46:24 UTC 2020-07-06T23:46:24.522Z undefined ERROR Uncaught Exception { "errorType": "Runtime.HandlerNotFound", "errorMessage": "get main is undefined or not exported" }'. To the right of the modal, there's a timeline showing '3 events' from 'Jul 6, 2020, 11:46:24 PM UTC' to 'Jul 6, 2020, 11:46:19 PM UTC'. Below the timeline, sections for 'LAST 24 HOURS', 'LAST 30 DAYS', and 'FIRST SEEN' are visible, along with a 'LAMBDA FUNCTIONS' summary for 'notes-app-2-api-prod-get'.

Handler error details in Seed

You should see the error `Runtime.HandlerNotFound`, along with message `get.main is undefined or not exported`.

And that about covers the main Lambda function errors. So the next time you see one of the above error messages, you'll know what's going on.

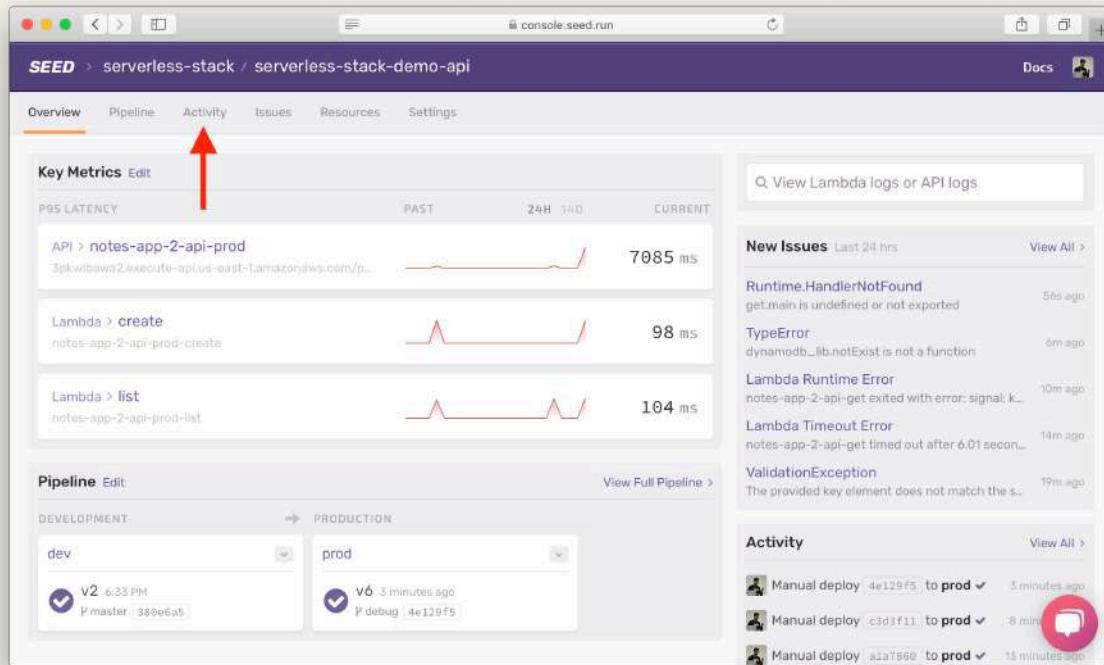
Rollback the Changes



Let's revert all the faulty code that we created.

```
$ git checkout master  
$ git branch -D debug
```

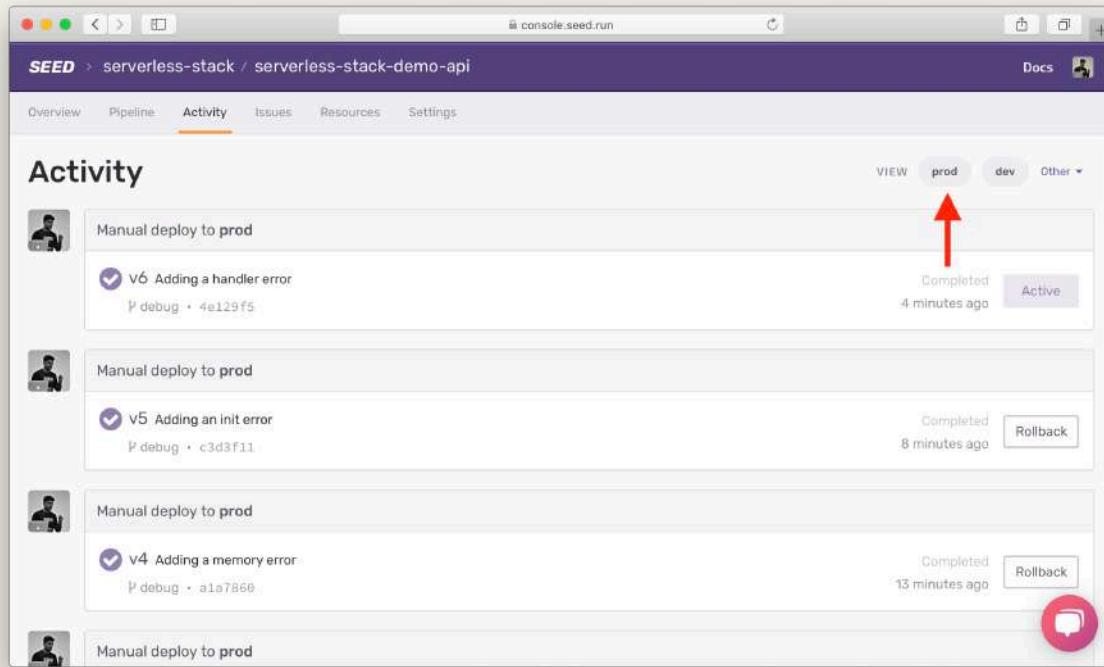
And rollback the prod build in Seed. Click on **Activity** in the Seed dashboard.



A screenshot of the SEED (Serverless Error Detection and Diagnosis) web interface. The page title is "SEED > serverless-stack / serverless-stack-demo-api". The navigation bar includes links for Overview, Pipeline, Activity (which is highlighted with a red arrow), Issues, Resources, and Settings. The main content area is divided into several sections: "Key Metrics" (P95 LATENCY for API and Lambda operations), "Pipeline Edit" (showing a flow from DEVELOPMENT to PRODUCTION with stages dev and prod), and "Activity" (listing recent manual deployments). On the right side, there's a sidebar with a search bar ("View Lambda logs or API logs") and a "New Issues" section listing errors like "Runtime.HandlerNotFound" and "Lambda Runtime Error".

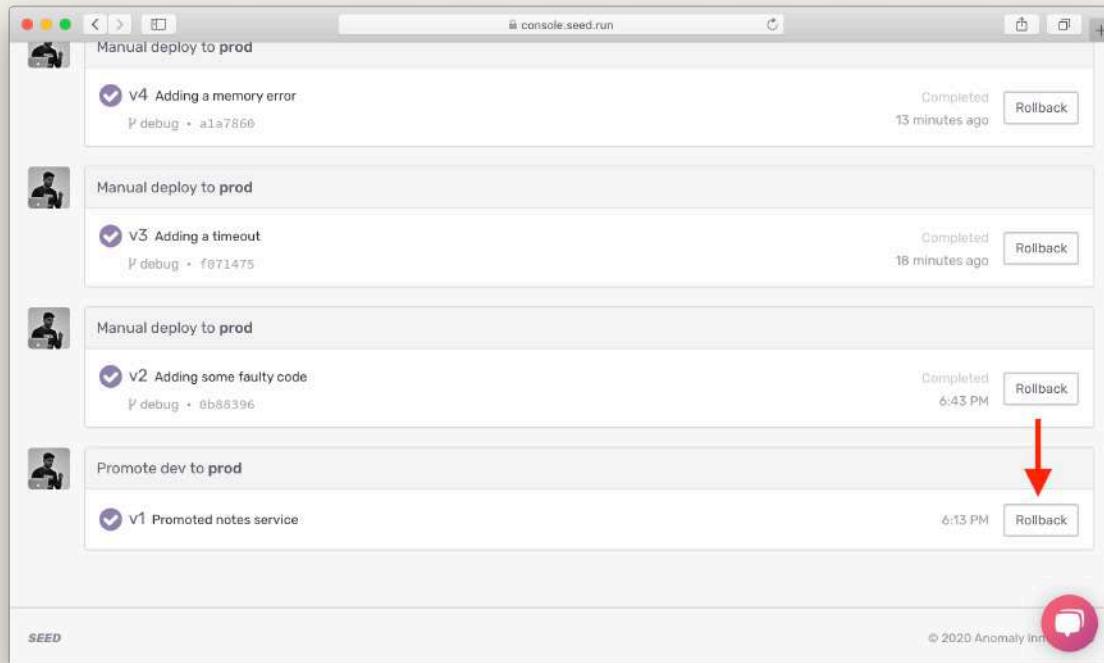
Click activity in Seed

Then click on **prod** over on the right. This shows us all the deployments made to our prod stage.



Click on prod activity in Seed

Scroll down to the last deployment from the `master` branch, past all the ones made from the `debug` branch. Hit **Rollback**.



Rollback on prod build in Seed

This will rollback our app to the state it was in before we deployed all of our faulty code.

Now let's move on to debugging API Gateway errors.



Help and discussion

View the [comments](#) for this chapter on our forums

Errors in API Gateway

In the past few chapters we looked at how to debug errors in our Lambda functions. However, our APIs can fail before our Lambda function has been invoked. In these cases, we won't be able to debug using the Lambda logs. Since there won't be any requests made to our Lambda functions.

The two common causes for these errors are:

1. Invalid API path
2. Invalid API method

Let's look at how to debug these.

Invalid API Path

Head over to the frontend repo.

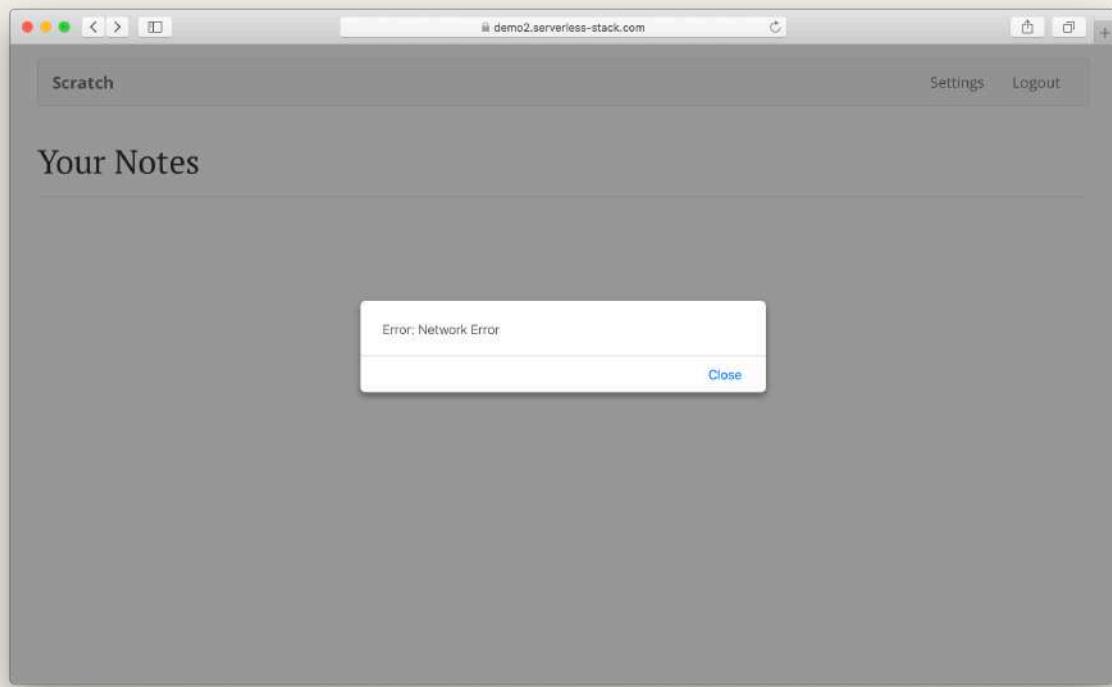
◆ CHANGE Open `src/containers/Home.js`, and replace the `loadNotes()` function with:

```
function loadNotes() {  
  return API.get("notes", "/invalid_path");  
}
```

◆ CHANGE Let's commit this and deploy it.

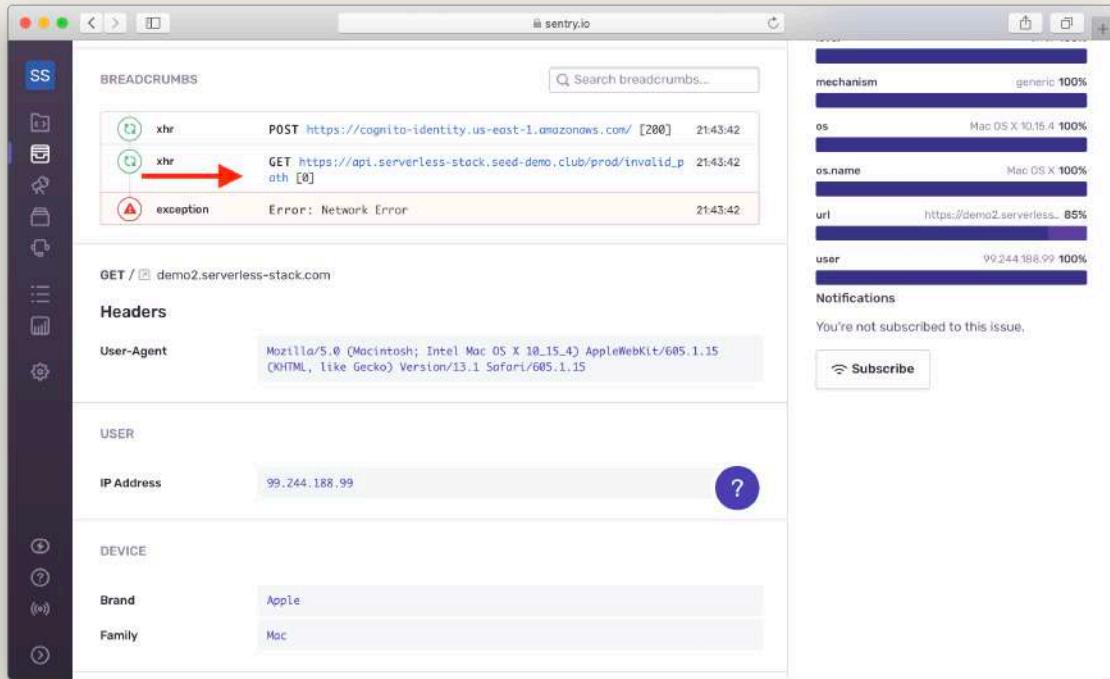
```
$ git add .  
$ git commit -m "Adding fault paths"  
$ git push
```

Head over to your notes app, and load the home page. You'll notice the page fails with an error alert saying Network Alert.



Invalid path error in notes app

On Sentry, the error will show that a GET request failed with status code 0.



Invalid path error in Sentry

What happens here is that: - The browser first makes an OPTIONS request to /invalid_path. - API Gateway returns a 403 response. - The browser throws an error and does not continue to make the GET request.

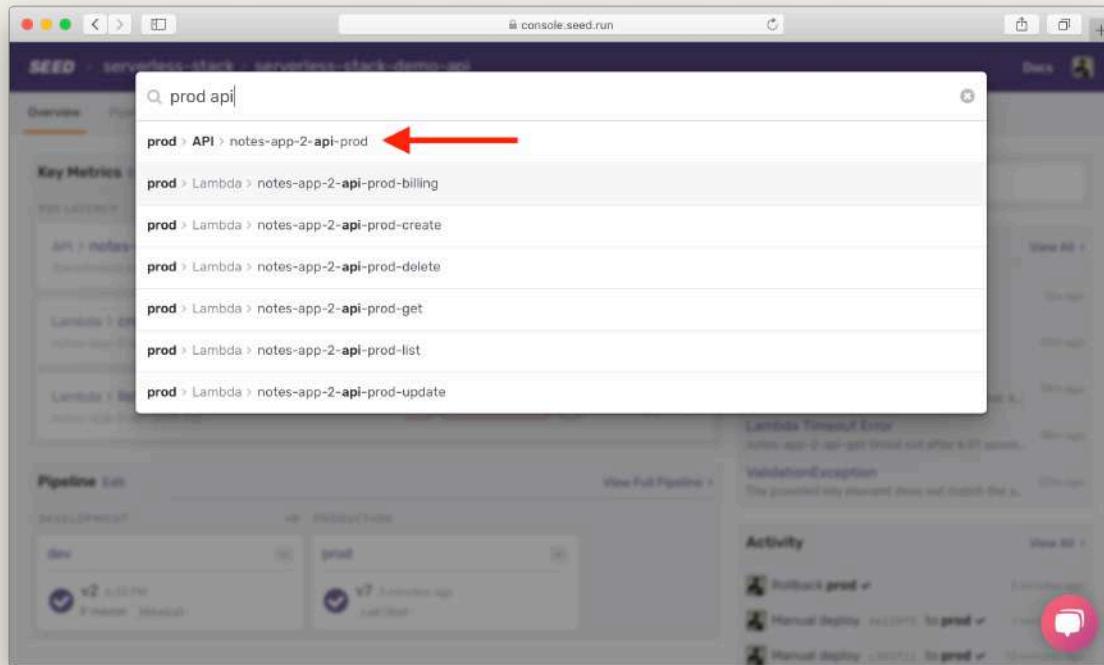
This means that our Lambda function was not invoked. So we'll need to check our API access logs instead.

Click on **View Lambda logs or API logs** in your Seed dashboard.

The screenshot shows the SEED dashboard for a 'serverless-stack-demo-api' project. The 'Overview' tab is selected. In the 'Key Metrics' section, there are three cards: 'API > notes-app-2-api-prod' (P95 LATENCY: 695 ms), 'Lambda > create' (notes-app-2-api-prod-create), and 'Lambda > list' (notes-app-2-api-prod-list, P95 LATENCY: 111 ms). The 'Pipeline' section shows a flow from 'DEVELOPMENT' (dev) to 'PRODUCTION' (prod). The 'prod' stage has a deployment labeled 'v7' (50 minutes ago, ca878a9). The 'Activity' section lists recent events: 'Rollback prod' (50 minutes ago), 'Manual deploy 4e129f5 to prod' (54 min), and 'Manual deploy c3d3f11 to prod' (58 minutes ago). A red arrow points to the search bar at the top right, which contains the placeholder 'Q, View Lambda logs or API logs'.

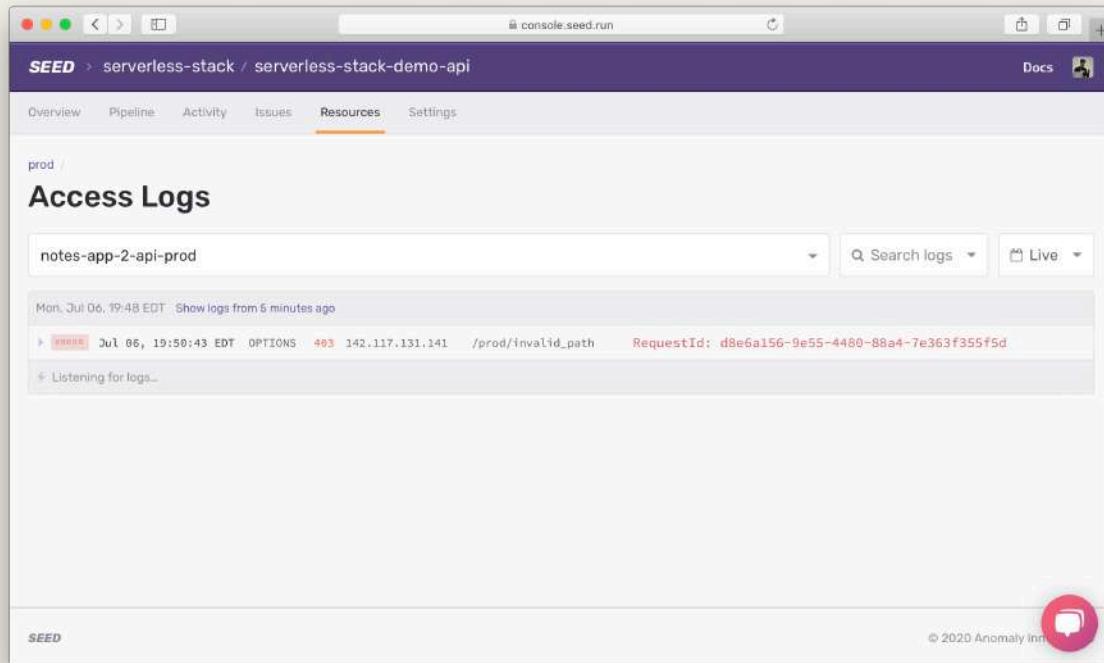
Click API logs search in Seed dashboard

Search `prod api` and select the API access log.



Search for API log in Seed dashboard

You should see an OPTIONS request with path /prod/invalid_path. You'll notice the request failed with a 403 status code.



Invalid API path request error in Seed

This will tell you that for some reason our frontend is making a request to an invalid API path. We can use the error details in Sentry to figure out where that request is being made.

Invalid API method

Now let's look at what happens when we use an invalid HTTP method for our API requests. Instead of a GET request we are going to make a PUT request.

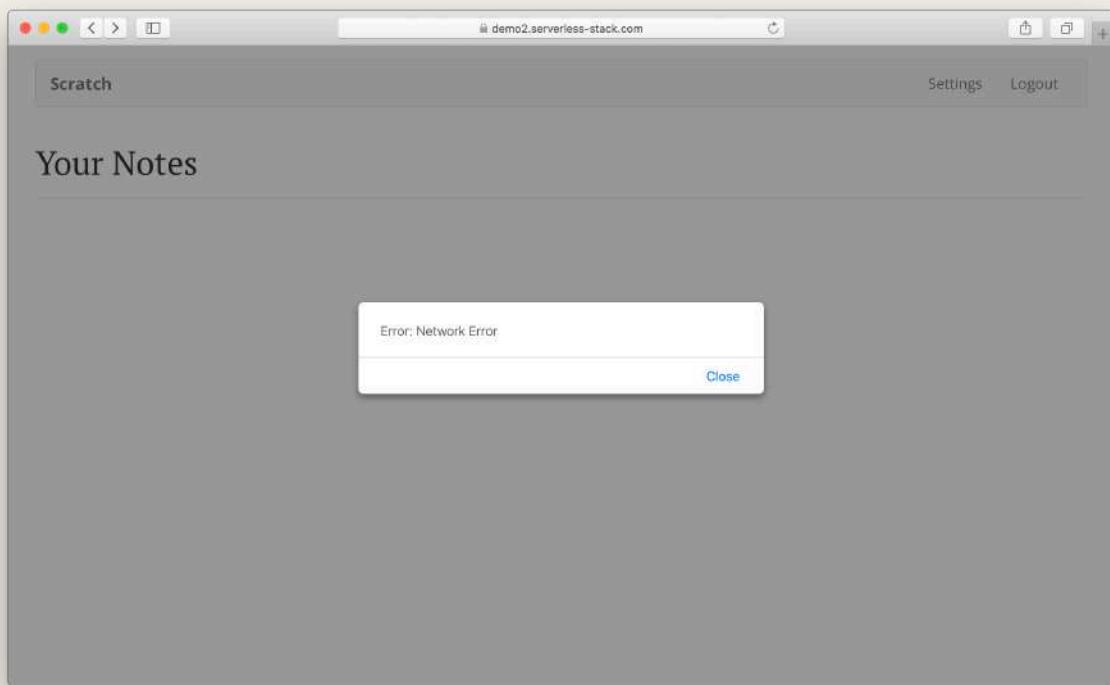
◆ CHANGE In `src/containers/Home.js` replace the `loadNotes()` function with:

```
function loadNotes() {  
  return API.put("notes", "/notes");  
}
```

◆ CHANGE Let's deploy our code.

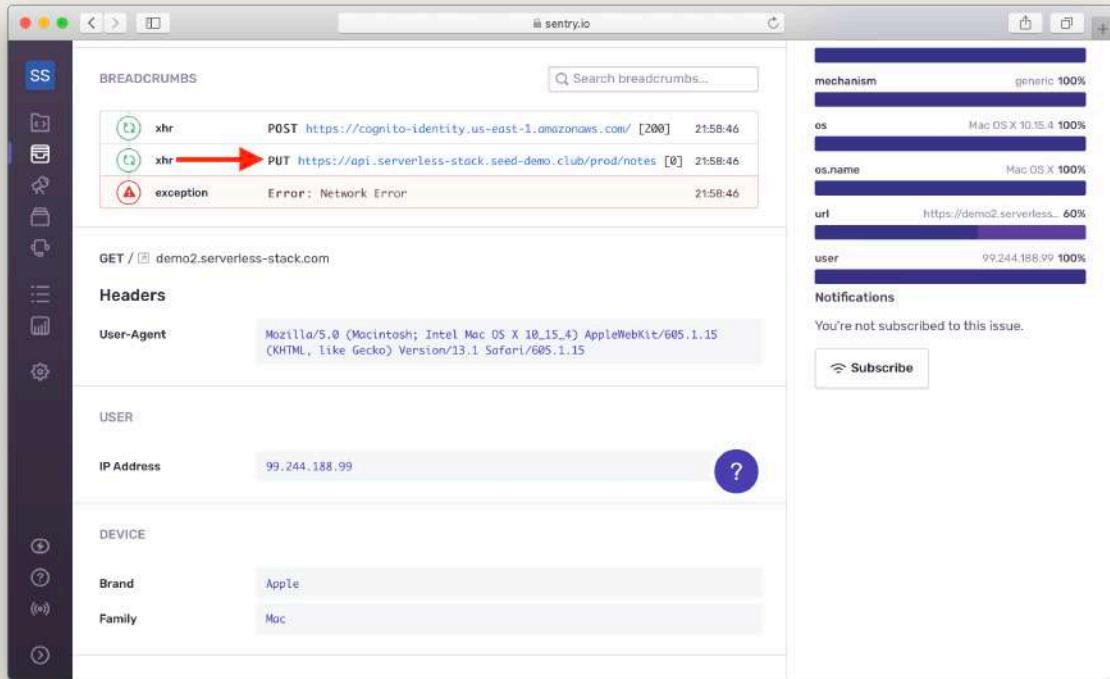
```
$ git add .
$ git commit -m "Adding invalid method"
$ git push
```

Our notes app should fail to load the home page.



Invalid method error in notes app

You should see a similar Network Error as the one above in Sentry. Select the error and you will see that the PUT request failed with 0 status code.

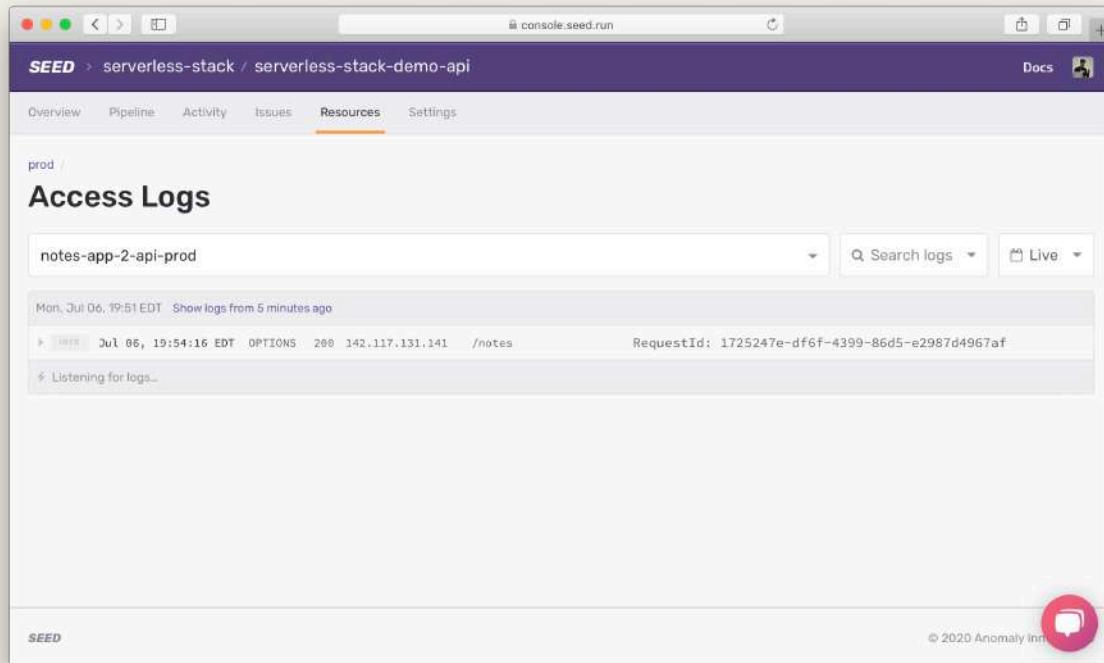


Invalid method error in Sentry

Here's what's going on behind the scenes:

- The browser first makes an OPTIONS request to /notes.
- API Gateway returns a successful 200 response with the HTTP methods allowed for the path.
- The allowed HTTP methods are GET and POST. This is because we defined:
 - GET request on /notes to list all the notes
 - POST request on /notes to create a new note
- The browser reports the error because the request method PUT is not allowed.

So in this case over on Seed, you'll only see an OPTIONS request in your access log, and not the PUT request.



Invalid API method request error in Seed

The access log combined with the Sentry error details should tell us what we need to do to fix the error.

And that covers all the major types of Serverless errors and how to debug them.

Remove the Faulty Code

Let's cleanup all the faulty code.

◆ CHANGE In `src/containers/Home.js` replace the `loadNotes()` function with the original:

```
function loadNotes() {  
  return API.get("notes", "/notes");  
}
```

Deploy the code.

```
$ git add .  
$ git commit -m "Reverting faulty code"  
$ git push
```

Now you are all set to go live with your brand new Serverless app!

Let's wrap things up next.



Help and discussion

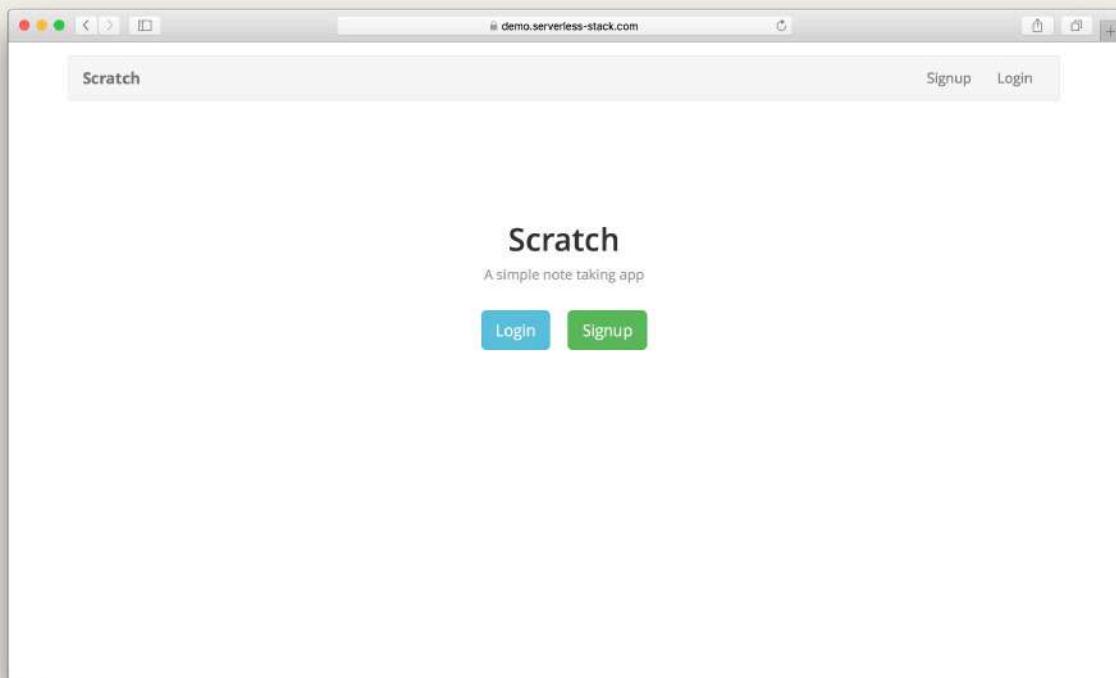
View the [comments](#) for this chapter on our forums

Conclusion

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.



App update live screenshot

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](#). 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 [fill out our survey](#) or send us any comments or feedback you might have, via [email](#).

[Fill out our survey](#)

Also, we'd love to feature what you've built with Serverless Stack, please [send us a URL and brief description](#).

Finally, if you've found this guide helpful, [please consider sponsoring us on GitHub](#).

Thank you and we hope you found this guide helpful!



Help and discussion

[View the comments for this chapter on our forums](#)

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](#): Documentation for the Serverless Framework
- [DynamoDB, explained](#): A Primer on the DynamoDB NoSQL database
- [React JS Docs](#): The official React docs
- [JSX In Depth](#): Learn JSX in a bit more detail
- [Create React App User Guide](#): The really comprehensive Create React App user guide
- [React-Bootstrap Docs](#): The official React-Bootstrap docs
- [Bootstrap v3 Docs](#): The Bootstrap v3 docs that React-Bootstrap is based on
- [React Router Docs](#): The official React Router v4 docs
- [AWS Amplify Developer Guide](#): The AWS Amplify developer guide
- [AWS Amplify API Reference](#): The AWS Amplify API reference
- [AWS CloudFormation Docs](#): The AWS user guide for CloudFormation
- [Jest Unit Test Docs](#): The official Jest docs
- [Seed Docs](#): The official Seed docs
- [Netlify 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.

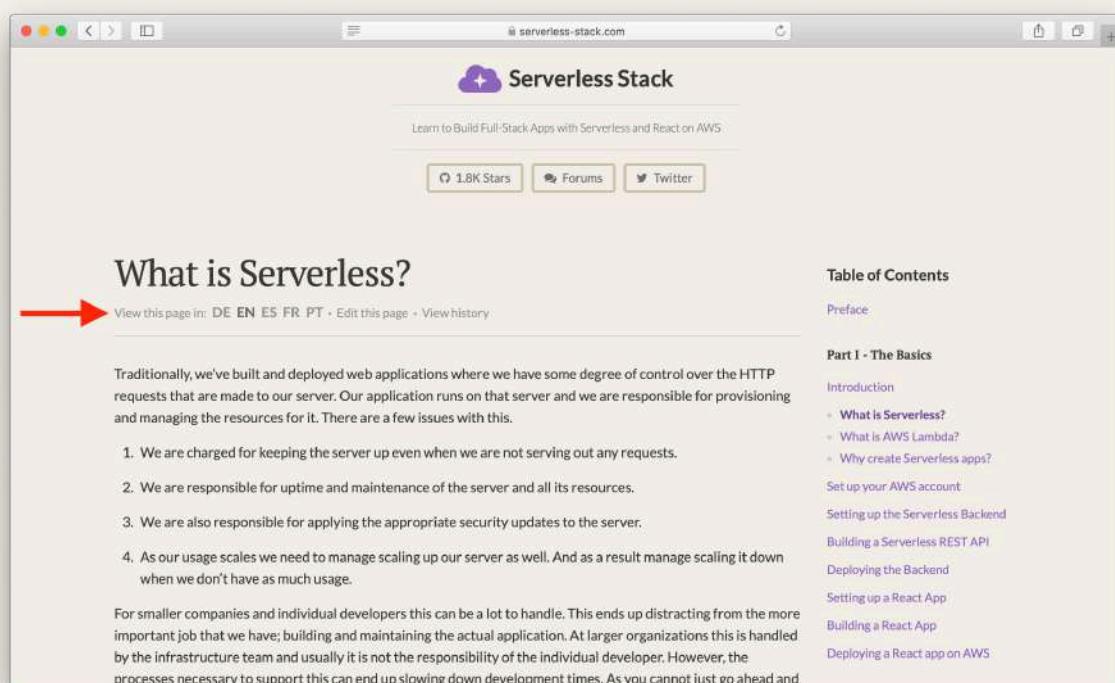


Help and discussion

View the [comments for this chapter on our forums](#)

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.



Chapter translation links Screenshot

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](#).

```
{% for page in site.chapters %} {% if page.lang == "en" and page.ref %} {% assign pages = site.chapters | where:"ref", page.ref | where_exp:"page", "page.lang != 'en'" | sort: 'lang' %} {% if pages.size > 0 %}{{ page.title }}
```

```
{% for page in pages %}  
{{ page.lang }}: {{ page.title }}  
{% endfor %}  
  
{% endif %}  
{% endif %}  
  
{% endfor %}
```

A big thanks to our contributors for helping make Serverless Stack more accessible!

- Bernardo Bugmann
- Sebastian Gutierrez
- Vincent Oliveira
- Leonardo Gonzalez
- Vieko Franetovic
- Christian Kaindl
- Jae Chul Kim



Help and discussion

View the [comments for this chapter on our forums](#)

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](#).

- **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](#) to fix any typos or errors you might find.

- **Helping others in the comments**

If you've found yourself using the [Discourse comments](#) 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](#).

- **Help translate the guide**

Our incredible readers are helping translate Serverless Stack into multiple languages. You can check out [our progress here](#). If you would like to help with our translation efforts, [leave us a comment here](#).

- **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](#).

- **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](#).

- **Give us a Star on GitHub**

We rely on our GitHub repo for everything from hosting this site to code samples and comments. [Starring our repo](#) helps us get the word out.

- **Sharing this guide**

Share this guide via Twitter or Facebook 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](#).



Help and discussion

View the [comments for this chapter](#) on our forums

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](#) of our [GitHub repo](#).

You can get these updates emailed to you via our [newsletter](#).

Changes

v5.0.1: Updating to new eBook format (Current)

Oct 21, 2020: Generating new eBook using Pandoc.

- [Tutorial changes](#)

v5.0: Using CDK to configure infrastructure resources

Oct 7, 2020: Moving from CloudFormation to AWS CDK to configure infrastructure resources. And using Serverless Stack Toolkit (SST) to deploy CDK alongside Serverless Framework.

- [Tutorial changes](#)
- [API](#)

v4.1: Adding new monitoring and debugging section

Apr 8, 2020: Adding a new section on monitoring and debugging full-stack Serverless apps. Updating React Router. Using React Context to manage app state.

- [Tutorial changes](#)
- [API](#)
- [Client](#)

v4.0: New edition of Serverless Stack

Oct 8, 2019: Adding a new section for Serverless best practices. Updating to React Hooks. Reorganizing chapters. Updating backend to Node 10.

- [Tutorial changes](#)
- [API](#)
- [Client](#)

v3.4: Updating to serverless-bundle and on-demand DynamoDB

Jul 18, 2019: Updating to serverless-bundle plugin and On-Demand Capacity for DynamoDB.

- [Tutorial changes](#)
- [API](#)

v3.3.3: Handling API Gateway CORS errors

Jan 27, 2019: Adding CORS headers to API Gateway 4xx and 5xx errors.

- [Tutorial changes](#)
- [API](#)

v3.3.2: Refactoring async Lambda functions

Nov 1, 2018: Refactoring async Lambda functions to return instead of using the callback.

- [Tutorial changes](#)
- [API](#)

v3.3.1: Updated to Create React App v2

Oct 5, 2018: Updated the frontend React app to use Create React App v2.

- [Tutorial changes](#)
- [Client](#)

v3.3: Added new chapters

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](#)
- [Facebook Login Client](#)
- [User Management Client](#)

v3.2: Added section on Serverless architecture

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](#)
- [Monorepo API](#)

v3.1: Update to use UsernameAttributes

May 24, 2018: CloudFormation now supports UsernameAttributes. This means that we don't need the email as alias work around.

- [Tutorial changes](#)
- [API](#)
- [Client](#)

v3.0: Adding Part II

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](#).

- [Tutorial changes](#)
- [API](#)
- [Client](#)

v2.2: Updating to user Node.js starter and v8.10

Apr 11, 2018: Updating the backend to use Node.js starter and Lambda Node v8.10. [Discussion on the update](#).

- [Tutorial changes](#)
- [API](#)

v2.1: Updating to Webpack 4

Mar 21, 2018: Updating the backend to use Webpack 4 and serverless-webpack 5.

- [Tutorial changes](#)
- [API](#)

v2.0: AWS Amplify update

Updating frontend to use AWS Amplify. Verifying SSL certificate now uses DNS validation. [Discussion on the update](#).

- [Tutorial changes](#)
- [Client](#)

v1.2.5: Using specific Bootstrap CSS version

Feb 5, 2018: Using specific Bootstrap CSS version since `latest` now points to Bootstrap v4. But React-Bootstrap uses v3.

- [Tutorial changes](#)
- [Client](#)

v1.2.4: Updating to React 16

Dec 31, 2017: Updated to React 16 and fixed `sigv4Client.js` [IE11 issue](#).

- [Tutorial changes](#)
- [Client](#)

v1.2.3: Updating to babel-preset-env

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](#)
- [API](#)

v1.2.2: Adding new chapters

Dec 1, 2017: Added the following *Extra Credit* chapters.

1. Customize the Serverless IAM Policy
 2. Environments in Create React App
- [Tutorial changes](#)

v1.2.1: Adding new chapters

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](#)

v1.2: Upgrade to Serverless Webpack v3

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.](#)

- [Tutorial changes](#)
- [API](#)

v1.1: Improved Session Handling

Aug 30, 2017: Fixing some issues with session handling in the React app. A few minor updates bundled together. [Discussion on the update.](#)

- [Tutorial changes](#)
- [Client](#)

v1.0: IAM as authorizer

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.](#)

- [Tutorial changes](#)
- [API](#)
- [Client](#)

v0.9: Cognito User Pool as authorizer

- [API](#)
- [Client](#)



Help and discussion

View the [comments](#) for this chapter on our forums

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. 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 1000 page eBook

You can also follow us on Twitter.

[Subscribe](#)



Help and discussion

View the [comments for this chapter on our forums](#)

Introduction

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](#) 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 or her 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](#) 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
 - Sharing resources using cross-stack references
 - Sharing code between services
 - Sharing API endpoints across services
- Configuring environments
 - Using separate AWS accounts to manage environments
 - Parameterizing resource names
 - Managing environment specific configs

- Best practices for handling secrets
- Sharing domains across environments
- Development lifecycle
 - Working locally
 - Creating feature environments
 - Creating pull request environments
 - Promoting to production
 - Handling rollbacks
- Using AWS X-Ray to trace Lambda functions

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

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. We'll be using [AWS CDK](#) with [SST](#).

2. Serverless Services

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 be using [Serverless Framework](#) for our services.

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](#) 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.



Help and discussion

View the [comments](#) for this chapter on our forums

Organize a Serverless app

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.

- **Stack**

A stack is what CloudFormation stack. In our case it is defined using [CDK](#).

- **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 on the other hand is created by the following stacks in CDK:

- **CognitoStack**: Defines a Cognito User and Identity pool used to store user data.

- **DynamoDBStack:** Defines a DynamoDB table called notes used to store notes data.
- **S3Stack:** 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/
|   --- billing-api/
|   --- notes-api/
|   --- notify-job/
| - infrastructure/
| - 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 infrastructure/ directory is a CDK app that is made up of multiple stacks.
5. 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.
6. The libs/ dir is just to illustrate that any common code that might be used across all services can be placed in here.
7. To deploy this application you are going to need to run serverless deploy separately in each of the services.

8. Environments (or stages) 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 of these issues. Services like [IOpipe](#), [Epsagon](#), and [Dashbird](#) help you with observability of your Lambda functions. And our own [Seed](#) 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](#). 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`
2. `serverless-stack-demo-ext-api`

In `serverless-stack-demo-ext-resources`, you have:

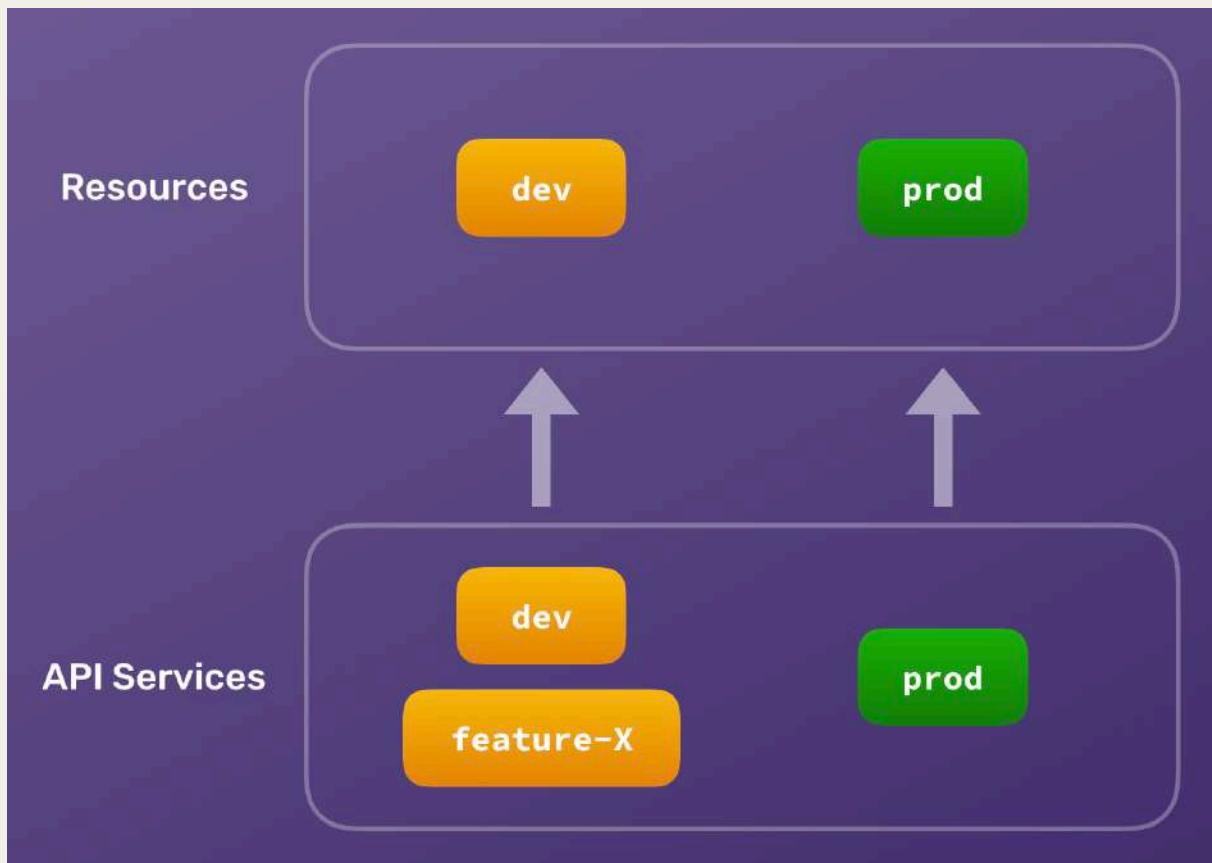
```
/lib/
  CognitoStack.js
  DynamoDBStack.js
  S3Stack.js
```

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**.



Organize serverless projects in an app

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.

Note that, we build on this monorepo setup further by using [Lerna](#) and [Yarn Workspaces](#) in our [Using Lerna and Yarn Workspaces with Serverless](#) extra credit chapter.

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.



Help and discussion

View the [comments for this chapter](#) on our forums

Cross-Stack References in Serverless

In the previous chapter we looked at [some of the most common patterns for organizing your Serverless applications](#). 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](#) 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.

You also might be defining your AWS infrastructure using [AWS CDK](#). And you want to make sure your Serverless API is connected to those resources.

For example, you might have your DynamoDB tables created in CDK and your APIs (as a Serverless service) need to refer to them. Of course you don't want to hard code this. To do this we are going to be using cross-stack references.

A cross-stack reference is a way for one CloudFormation template to refer to the resource in another CloudFormation template.

- Cross-stack references have a name and value.
- Cross-stack references only apply within the same region.
- The name needs to be unique for a given region in an AWS account.

A reference is created when one stack creates a CloudFormation export and another imports it. So for example, our `DynamoDBStack.js` is exporting the name of our DynamoDB table, and our `notes-api` service is importing it. Once the reference has been created, you cannot remove the DynamoDB stack without first removing the stack that is referencing it (the `notes-api`).

The above relationship between two stacks means that they need to be deployed and removed in a specific order. We'll be looking at this later.

CloudFormation Export in CDK

To create a cross-stack reference, we first create a CloudFormation export.

```
new CfnOutput(this, "TableName", {  
    value: table.tableName,  
    exportName: app.logicalPrefixedName("ExtTableName"),  
});
```

The above is a CDK example from our `DynamoDBStack.js`.

Here the `exportName` is the name of the CloudFormation export. We use a convenience method from `SST` called `app.logicalPrefixedName` that prefixes our export name with the name of the stage we are deploying to, and the name of our SST app. This ensures that our export name is unique when we deploy our stack across multiple environments.

CloudFormation Export in Serverless Framework

Similarly, we can create a CloudFormation export in Serverless Framework by adding the following

```
resources:  
  Outputs:  
    NotePurchasedTopicArn:  
      Value:  
        Ref: NotePurchasedTopic  
      Export:  
        Name: ${self:custom.stage}-ExtNotePurchasedTopicArn
```

The above is an example from the `serverless.yml` of our `billing-api`. We can add a `resources:` section to our `serverless.yml` and the `Outputs:` allows us to add CloudFormation exports.

Just as above we need to name our CloudFormation export. We do it using the `Name:` property. Here we are prefixing our export name with the stage name (`/${self:custom.stage}`) to make it unique across environments.

The `/${self:custom.stage}` is a custom variable that we define at the top of our `serverless.yml`.

```
# Our stage is based on what is passed in when running serverless
# commands. Or falls back to what we have set in the provider section.
stage: ${opt:stage, self:provider.stage}
```

Importing a CloudFormation Export

Now once we've created a CloudFormation export, we need to import it in our `serverless.yml`. To do so, we'll use the `Fn::ImportValue` CloudFormation function.

For example, in our `notes-api/serverless.yml`.

```
provider:
  environment:
    tableName: !ImportValue ${self:custom.sstApp}-ExtTableName
```

We import the name of the DynamoDB table that we created and exported in `DynamoDBStack.js`.

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 how we share code between our services.



Help and discussion

View the [comments for this chapter on our forums](#)

Share Code Between Services

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](#). 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](#) 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](#) does this automatically for you.

You can also use [Yarn Workspaces](#) (and [Lerna](#)) to manage the dependencies for your monorepo setup. We cover this setup in a separate chapter — [Using Lerna and Yarn Workspaces with Serverless](#).

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](#)'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](#) 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 falls back to what we have set in the provider section.
  stage: ${opt:stage, self:provider.stage}
sstAppMapping:
  prod: prod
```

```
dev: dev
sstApp: ${self:custom.sstAppMapping.${self:custom.stage}},
  ↵ self:custom.sstAppMapping.dev}-notes-ext-infra

lambdaPolicyXRay:
  Effect: Allow
  Action:
    - xray:PutTraceSegments
    - xray:PutTelemetryRecords
  Resource: "*"
```

And in each of our services, 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.

For simplifying our `serverless.yml` config within a service, we split it up further. In our `services/notes-api/serverless.yml` in our [sample repo](#) you'll notice the following:

```
resources:
  # API Gateway Errors
  - ${file(resources/api-gateway-errors.yml)}
  # Cognito Identity Pool Policy
  - ${file(resources/cognito-policy.yml)}
```

The `api-gateway-errors.yml` adds the headers for 4xx and 5xx API errors. While the `cognito-policy.yml` adds the IAM policy for allowing our Cognito authenticated users to access the Notes API.

Statement:

```
- Effect: 'Allow'  
  Action:  
    - 'execute-api:Invoke'  
  Resource:  
    !Sub 'arn:aws:execute-  
→ api:${AWS::Region}:${AWS::AccountId}: ${ApiGatewayRestApi}/*'
```

Next, let's look at what happens when multiple API services need to share the same API endpoint.

**Help and discussion**

View the [comments for this chapter](#) on our forums

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://z6pv8ao4l.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` 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: !ImportValue ${self:custom.stage}-ExtApiGatewayRestApiId
    restApiRootResourceId: !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           !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           !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.

**Help and discussion**

View the [comments](#) for this chapter on our forums

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](#) we are using SST to deploy our CDK app. CDK internally keeps track of the dependencies between stacks.

Our `lib/index.js` looks like this.

```
export default function main(app) {
  new DynamoDBStack(app, "dynamodb");

  const s3 = new S3Stack(app, "s3");

  new CognitoStack(app, "cognito", { bucketArn: s3.bucket.bucketArn });
}
```

Here CDK knows that the `CognitoStack` depends on the `S3Stack`. And it needs to wait for the `S3Stack` to complete first.

SST will deploy the stacks in our CDK app concurrently while ensuring that the dependencies are respected.

Next for the [API repo](#) 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 dev-ExtNotePurchasedTopicArn.
- 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 dev-ExtNotePurchasedTopicArn 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](#), we handle this using a concept of [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.



Help and discussion

View the [comments for this chapter on our forums](#)

Manage environments

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](#) 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](#).

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](#) and for [PRs on PR creation](#). 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!



Help and discussion

View the [comments for this chapter on our forums](#)

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](#) 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 *write* 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

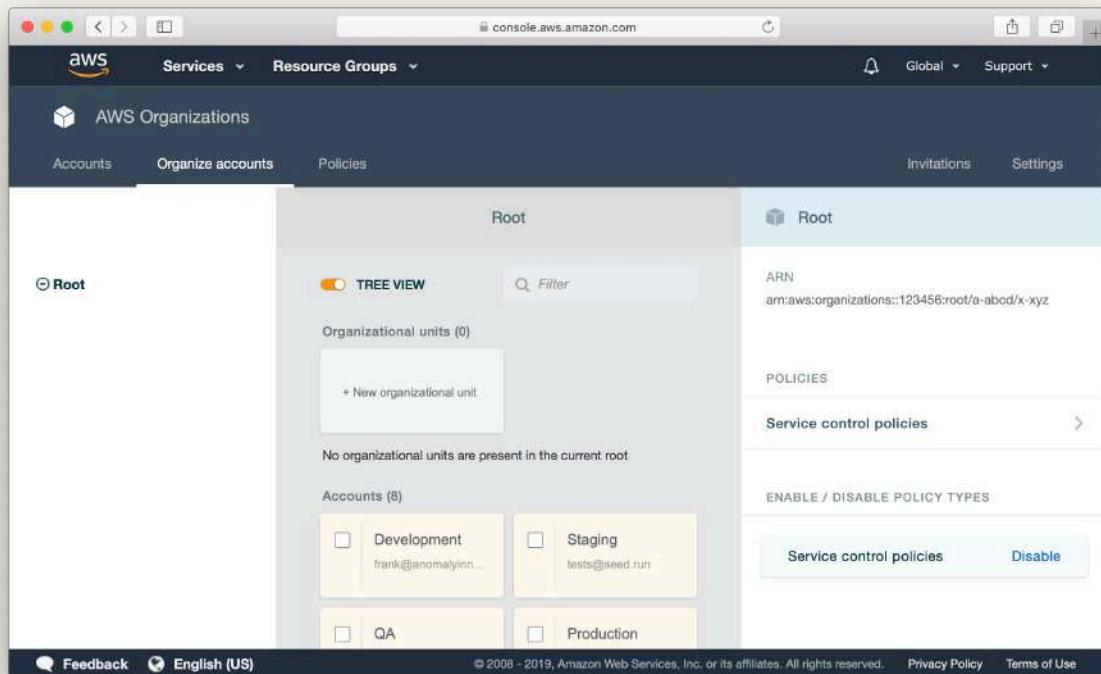
Another benefit of separating environments across AWS accounts is helping you work with AWS' service limits. You might be familiar with the various hard and soft limits of the AWS services you use. Like the [Lambda's 75 GB code storage limit](#) or the [total number of S3 buckets per account limit](#). 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.



Accounts in AWS Organization console

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.



Help and discussion

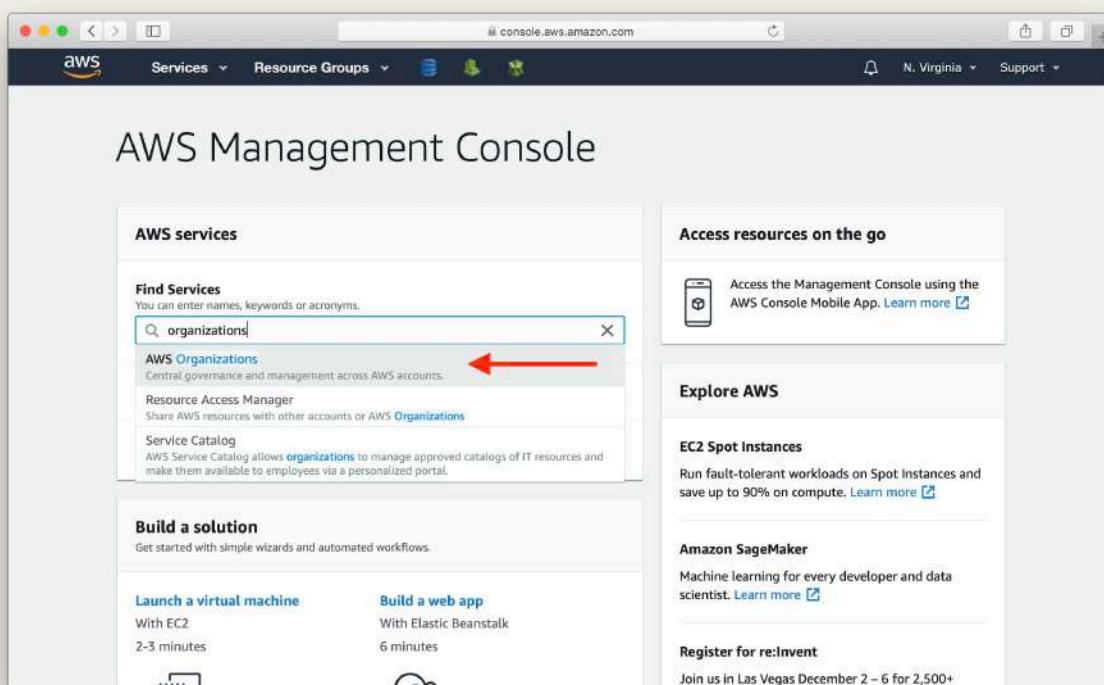
View the [comments for this chapter on our forums](#)

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

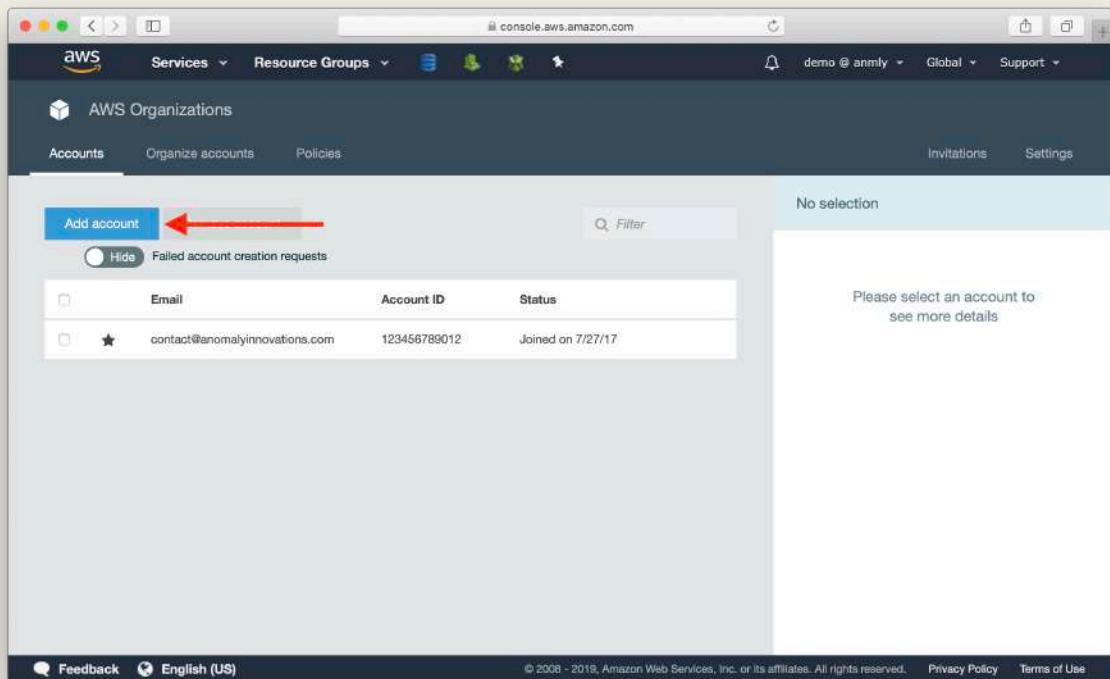
Go to the AWS Organizations console.



Select AWS Organizations service

The account labeled with the star is your **master** AWS account. This account cannot be removed from the organization.

Select **Add account**.

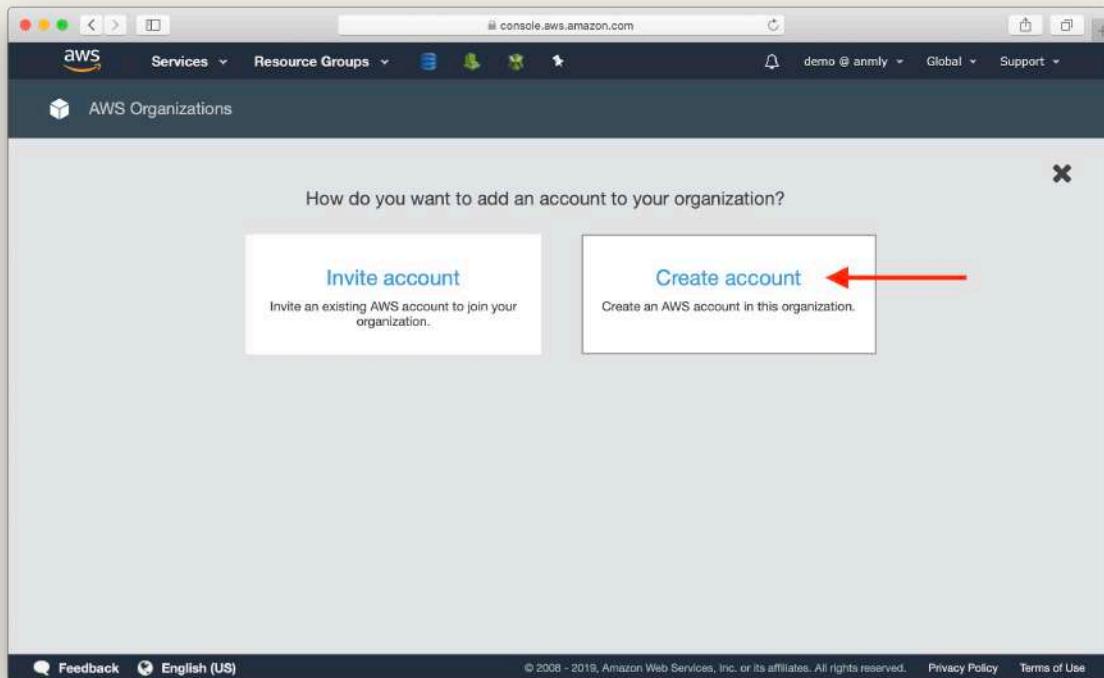


A screenshot of the AWS Organizations console in a web browser. The URL is `console.aws.amazon.com`. The top navigation bar includes the AWS logo, Services dropdown, Resource Groups dropdown, and various icons. The user is signed in as `demo @ anmly`. The main menu has options for Accounts, Organize accounts, Policies, Invitations, and Settings. The 'Accounts' tab is selected. On the left, there's a table with columns: (checkbox), Email, Account ID, and Status. A single row is shown: (checkbox), **★ contact@anomalyinnovations.com**, 123456789012, Joined on 7/27/17. Above the table, there's a blue button labeled 'Add account' with a red arrow pointing to it. Below the table, there's a link 'Failed account creation requests'. To the right of the table, a sidebar says 'No selection' and 'Please select an account to see more details'. At the bottom, there are links for Feedback, English (US), and footer links: © 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved., Privacy Policy, and Terms of Use.

Add account in AWS Organizations

You can either create a new AWS account or if you already have multiple standalone AWS accounts, you can add them into your organization.

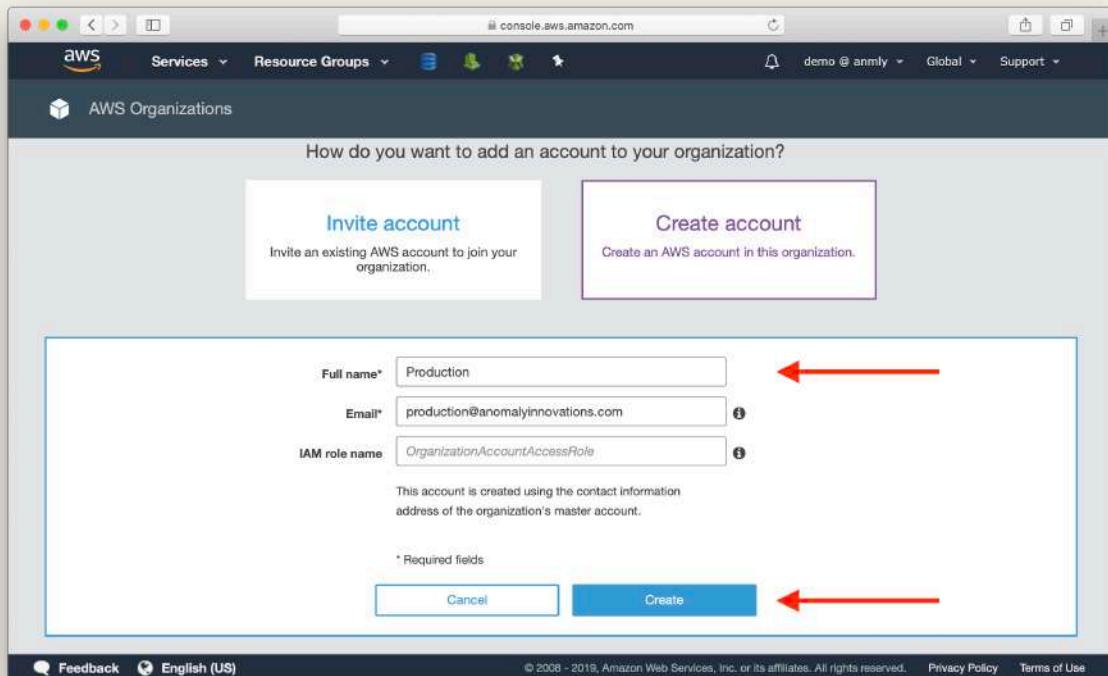
Select **Create account**.



Create account in AWS Organizations

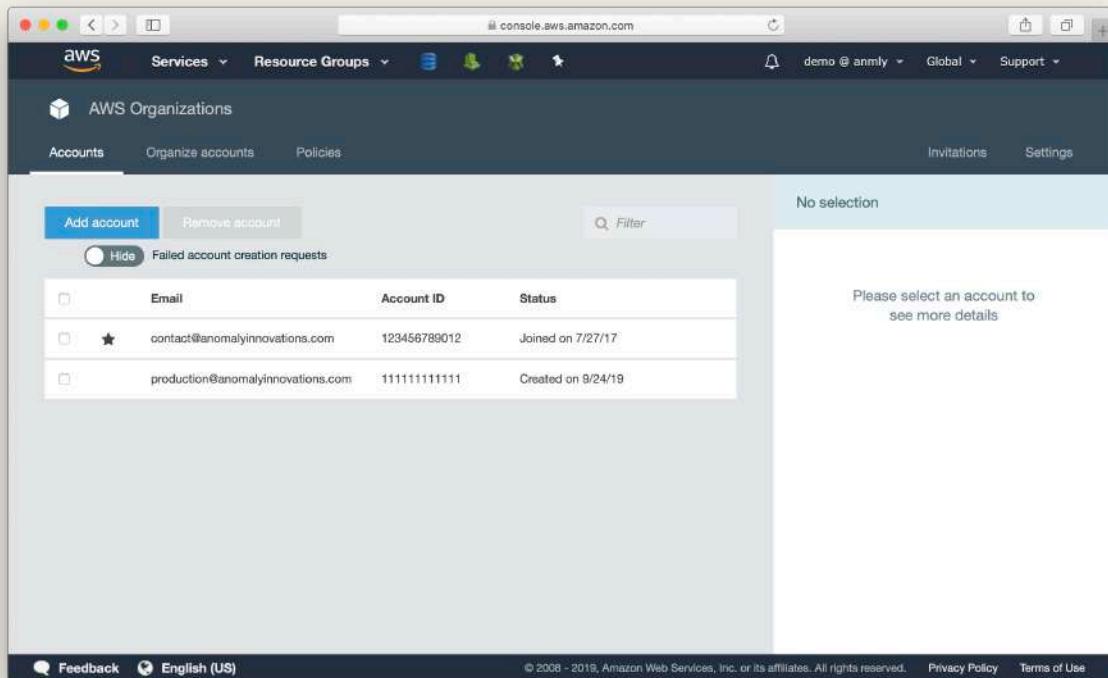
Let's create our Production account first. Fill out the following:

- **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.



Set Production account detail

Now, you have 2 AWS accounts in your organization.



The screenshot shows the AWS Organizations console interface. At the top, there's a navigation bar with links for Services, Resource Groups, and various user and account settings. Below the navigation is the 'AWS Organizations' header. Underneath the header, there are tabs for 'Accounts', 'Organize accounts', and 'Policies'. On the right side of the main content area, there are buttons for 'Invitations' and 'Settings'. The central part of the screen displays a table of accounts. The columns are 'Email', 'Account ID', and 'Status'. There are two rows of data:

Email	Account ID	Status
contact@anomalyinnovations.com	123456789012	Joined on 7/27/17
production@anomalyinnovations.com	111111111111	Created on 9/24/19

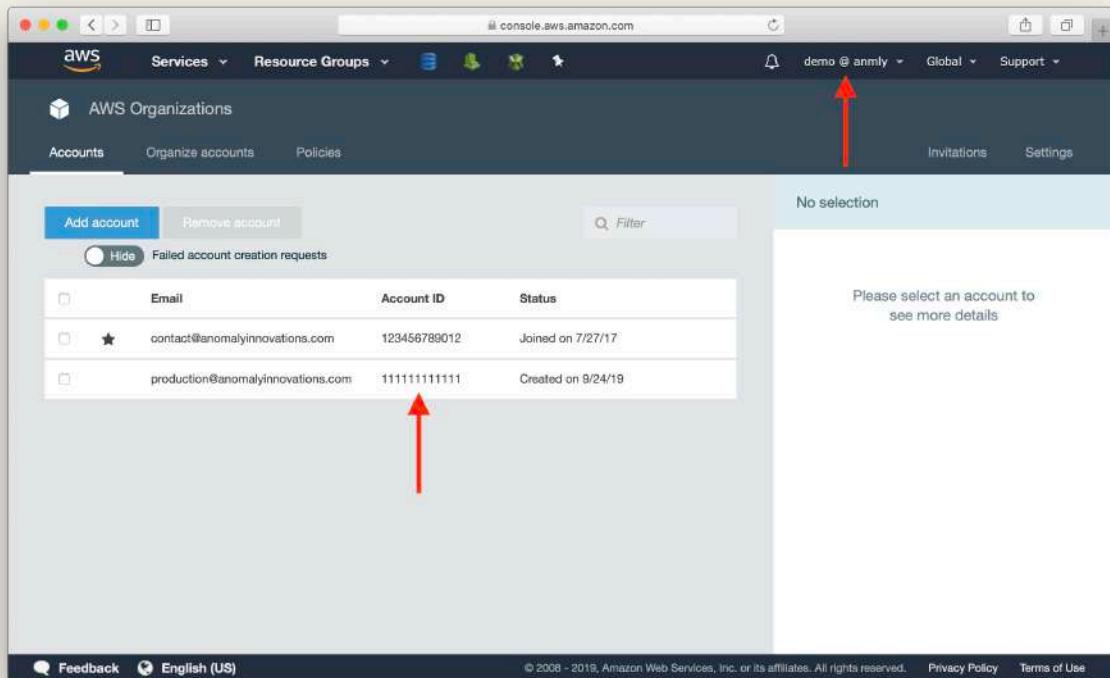
To the right of the table, a message says 'Please select an account to see more details'. At the bottom of the page, there are links for Feedback, English (US), and legal notices like Privacy Policy and Terms of Use.

Production account created in AWS Organizations

Access AWS accounts

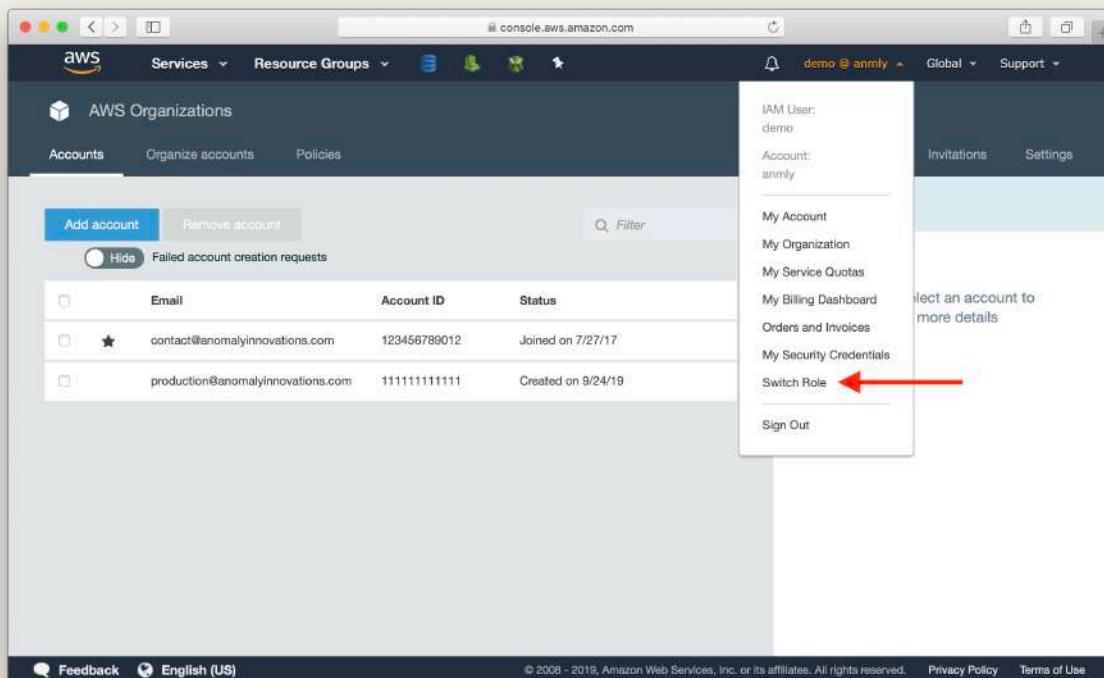
Next, let's try switch into the Production account. First, take a note of the newly created **Account ID**. We need this number in the next step.

Then, select the account picker at the top.



Select account picker in AWS console

Select **Switch Role**.



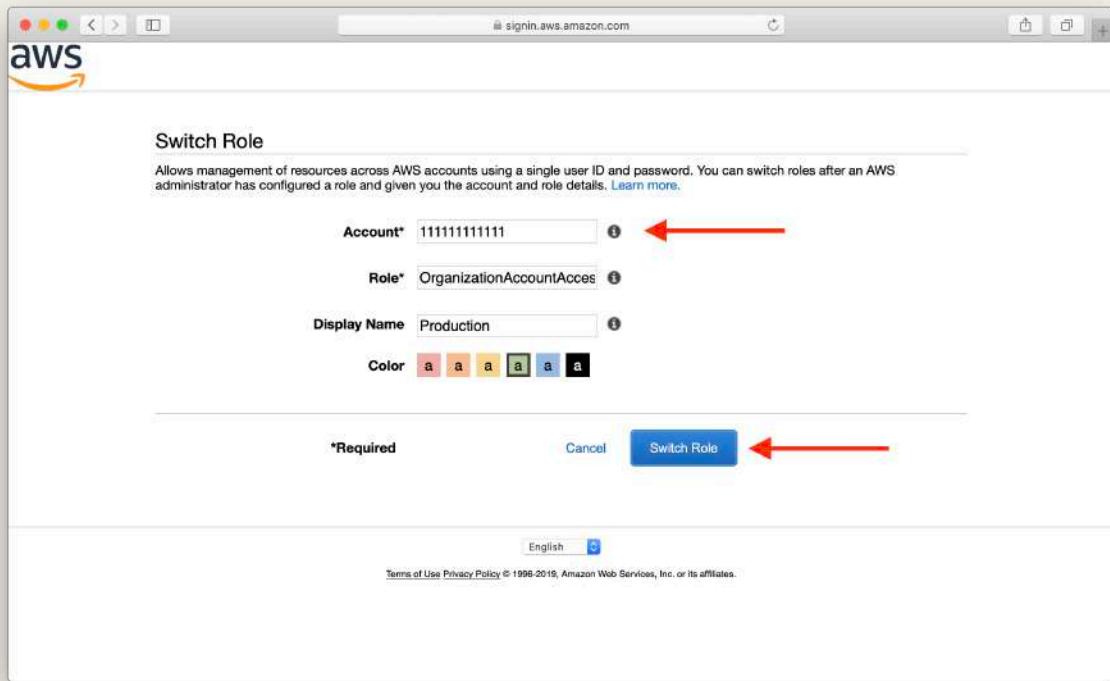
Select switch role in AWS console

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.

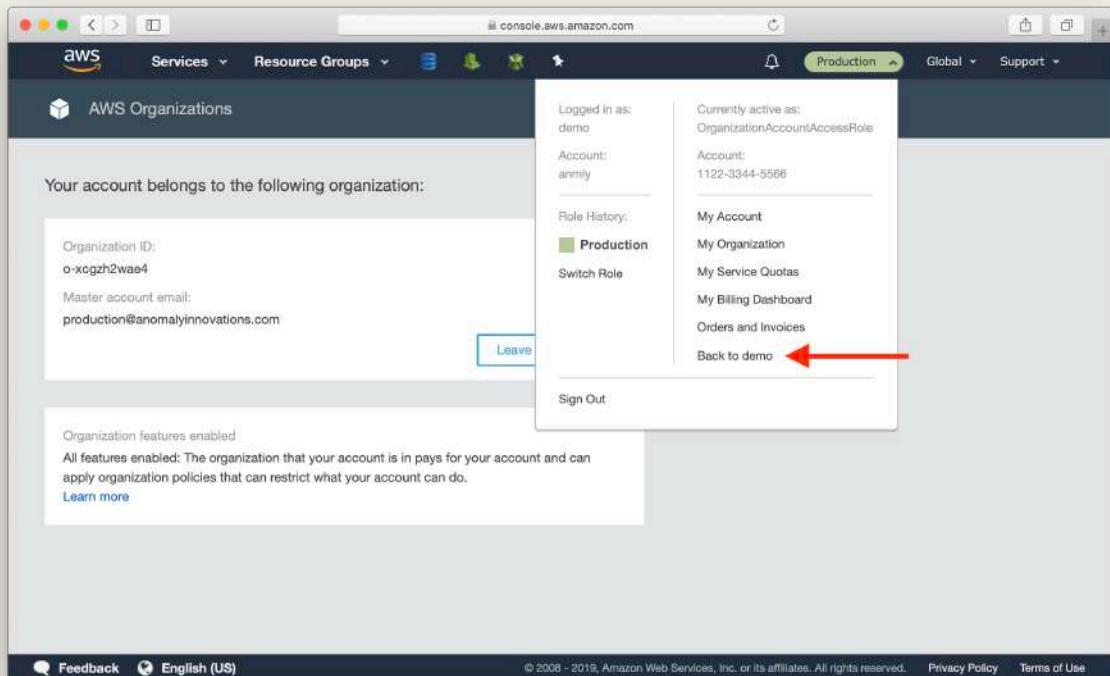
Then select **Switch Role**.



Assume role in Production account

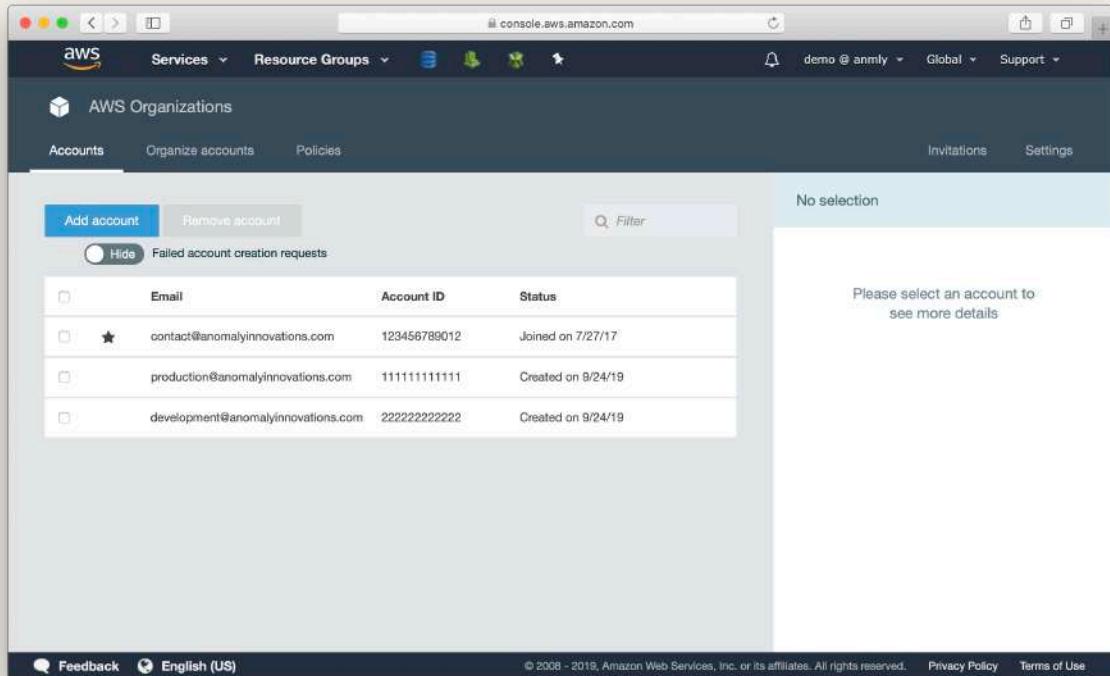
Now, you are in the **Prod** account. You can check which account you are currently assumed into by looking at the top bar.

You can switch back to the master account by clicking on the account picker and selecting **Back to master**.



Switch back to master account

Next, repeat the above steps to create the Development account.



The screenshot shows the AWS Organizations console interface. At the top, there are tabs for 'Accounts', 'Organize accounts', and 'Policies'. Below these are buttons for 'Add account' (highlighted in blue), 'Remove account', and a 'Filter' search bar. A message 'No selection' is displayed above a table. The table has columns for 'Email', 'Account ID', and 'Status'. It lists three accounts:

Email	Account ID	Status
contact@anomalyinnovations.com	123456789012	Joined on 7/27/17
production@anomalyinnovations.com	111111111111	Created on 9/24/19
development@anomalyinnovations.com	222222222222	Created on 9/24/19

A message 'Please select an account to see more details' is shown on the right side of the table area. At the bottom of the page, there are links for 'Feedback', 'English (US)', and copyright information.

Create Development account in AWS Organizations

Now we have our AWS accounts created. Let's make sure we are using these environments correctly in the configuration of our app.



Help and discussion

View the [comments for this chapter on our forums](#)

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	<code>\$serviceName-\$stage-\$ functionName</code>	<code>notes-app-ext-notes-api-dev-get</code>
API Gateway project	<code>\$stage-\$serviceName</code>	<code>dev-notes-app-ext-notes-api</code>
CloudWatch log groups	<code>/aws/lambda/\$serviceName-\$stage-\$ functionName</code>	<code>/aws/lambda/notes-app-ext-notes-api-dev-get</code>
IAM roles	<code>\$serviceName-\$stage-\$region-lambdaRole</code>	<code>notes-app-ext-notes-api-dev-us-east-1-lambdaRole</code>

Resource	Scheme	Example
S3 bucket	\$stackName-\$resourceName-\$hash	notes-app-ext-notes-api-serverlessdeploymentbucket-19fhidI3prw0m

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: ${self:custom.stage}-note-purchased
```

Parameterize Resources in CDK With SST

For CDK on the other hand we use [SST](#) to automatically parameterize our stack names. And use a helper method to parameterize specific resource names.

So for example in the `lib/index.js` file in our [resources repo](#).

```
export default function main(app) {
  new DynamoDBStack(app, "dynamodb");

  const s3 = new S3Stack(app, "s3");

  new CognitoStack(app, "cognito", { bucketArn: s3.bucket.bucketArn });
}
```

Our stack names are called dynamodb, s3, and cognito. But when these are deployed, they are deployed as:

```
dev-notes-ext-infra-dynamodb
dev-notes-ext-infra-s3
dev-notes-ext-infra-cognito
```

Where dev is the stage we are deploying to and notes-ext-infra is the name of our SST app, as specified in our `sst.json`.

For specific resources, such as CloudFormation exports, we use the `app.logicalPrefixedName` helper method. Here's an example from `lib/DynamoDBStack.js`.

```
new CfnOutput(this, "TableName", {
  value: table.tableName,
  exportName: app.logicalPrefixedName("ExtTableName"),
});
```

The `app.logicalPrefixedName` prefixes our export name with the name of the stage and the app.

Parameterizing your resources allows your app to be deployed to multiple environments without naming conflicts. Next, let's deploy our app!



Help and discussion

View the [comments for this chapter on our forums](#)

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](#) 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](#).

Configure AWS Profiles

Follow the [setup up IAM users](#) 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.

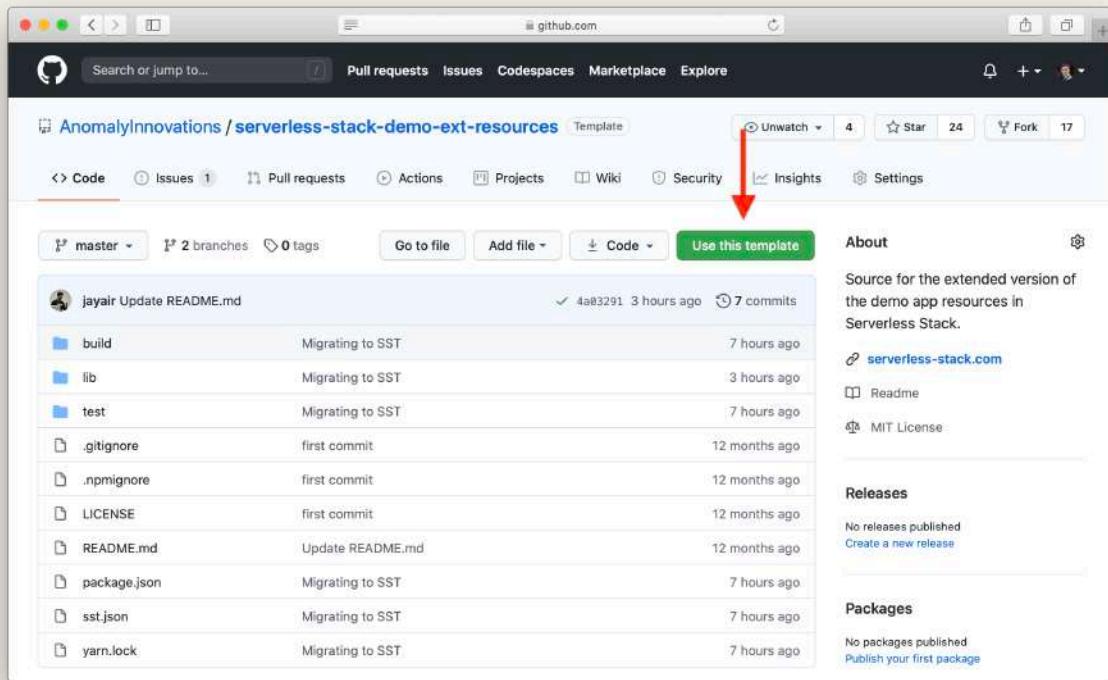
Production deployments should always go through our CI/CD pipeline.

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.

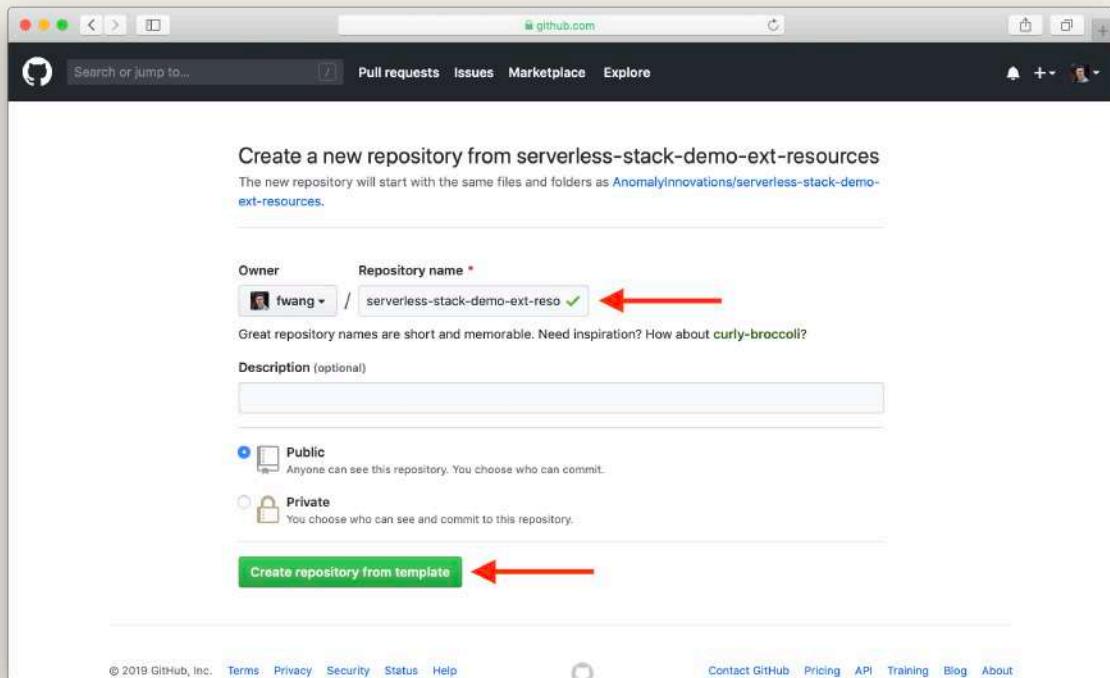
Create demo repos

Let's first create [the resources repo](#). Click **Use this template**.



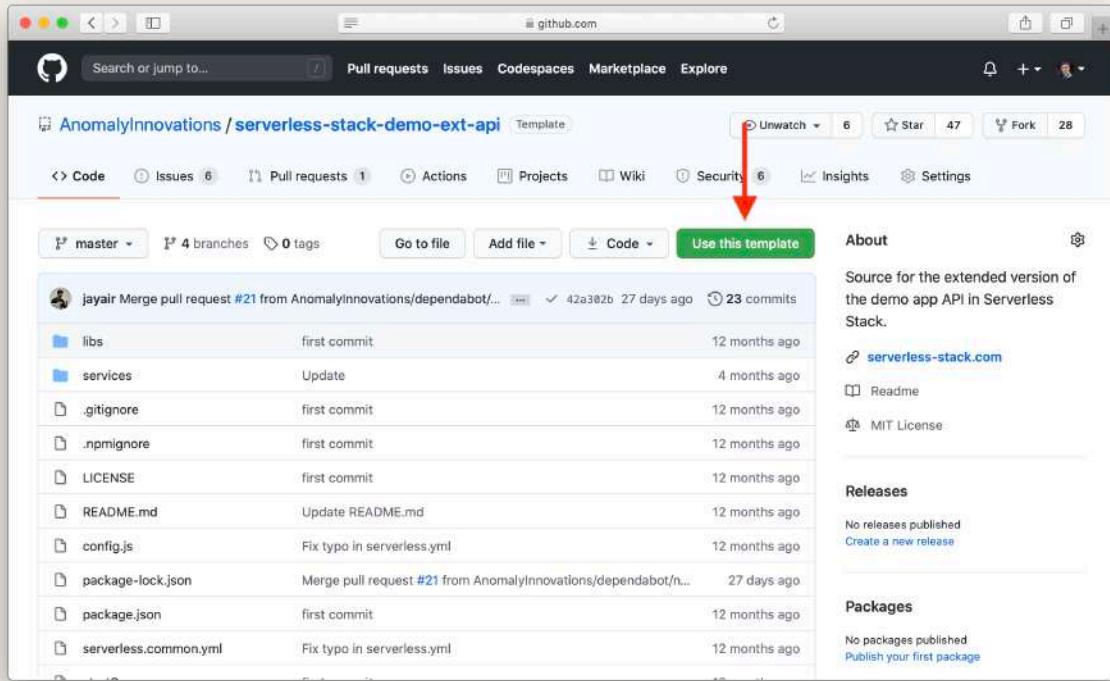
Use demo resources repo template

Enter Repository name **serverless-stack-demo-ext-resources** and click **Create repository from template**.



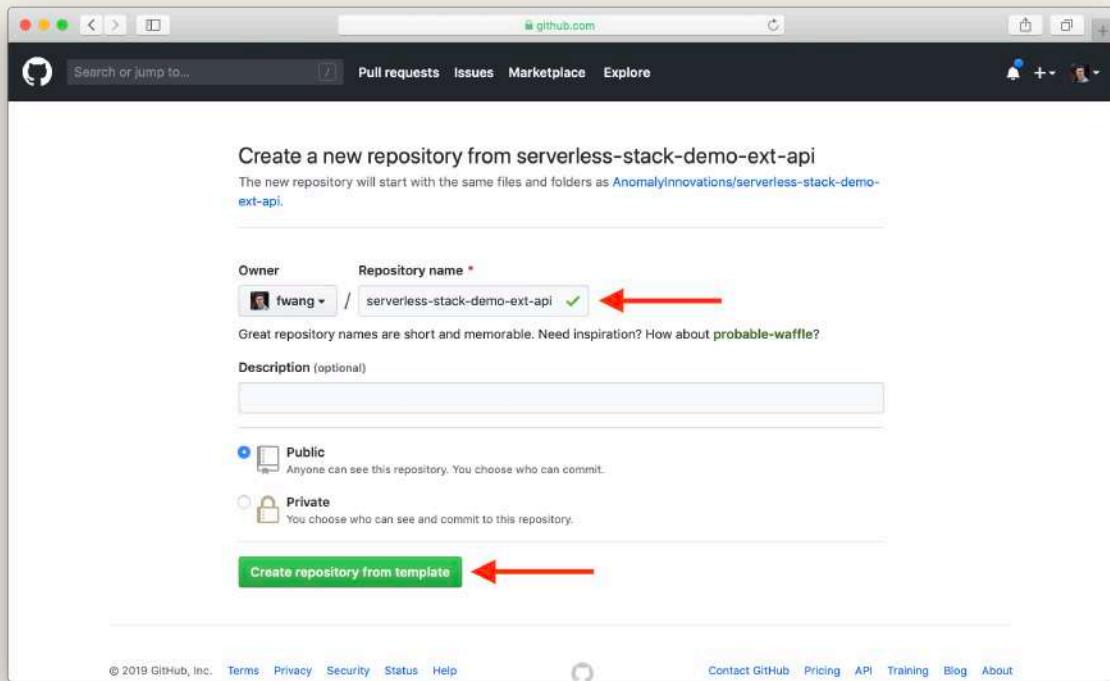
Create demo resources repo on GitHub

And do the same for the API services repo.



Create demo API services repo template

Enter Repository name **serverless-stack-demo-ext-api** and click **Create repository from template**.



Create demo API services repo on GitHub

Now that we've forked these repos, let's deploy them to our environments. We are going to use [Seed](#) to do this but you can set this up later with your favorite CI/CD tool.



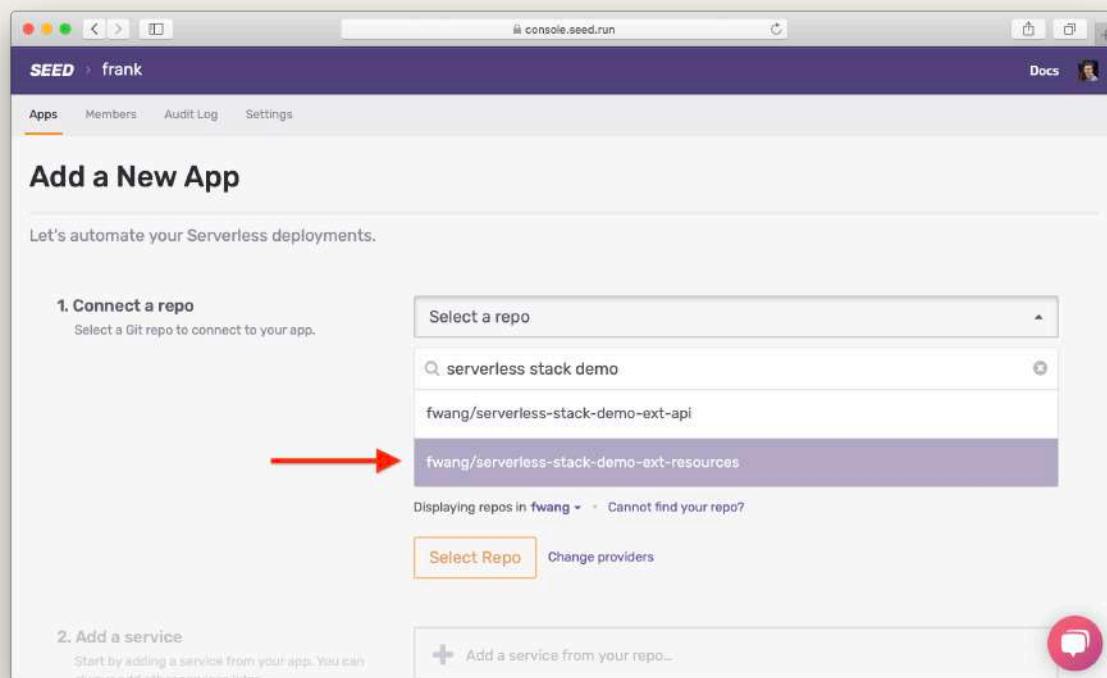
Help and discussion

View the [comments for this chapter on our forums](#)

Deploy the Resources Repo

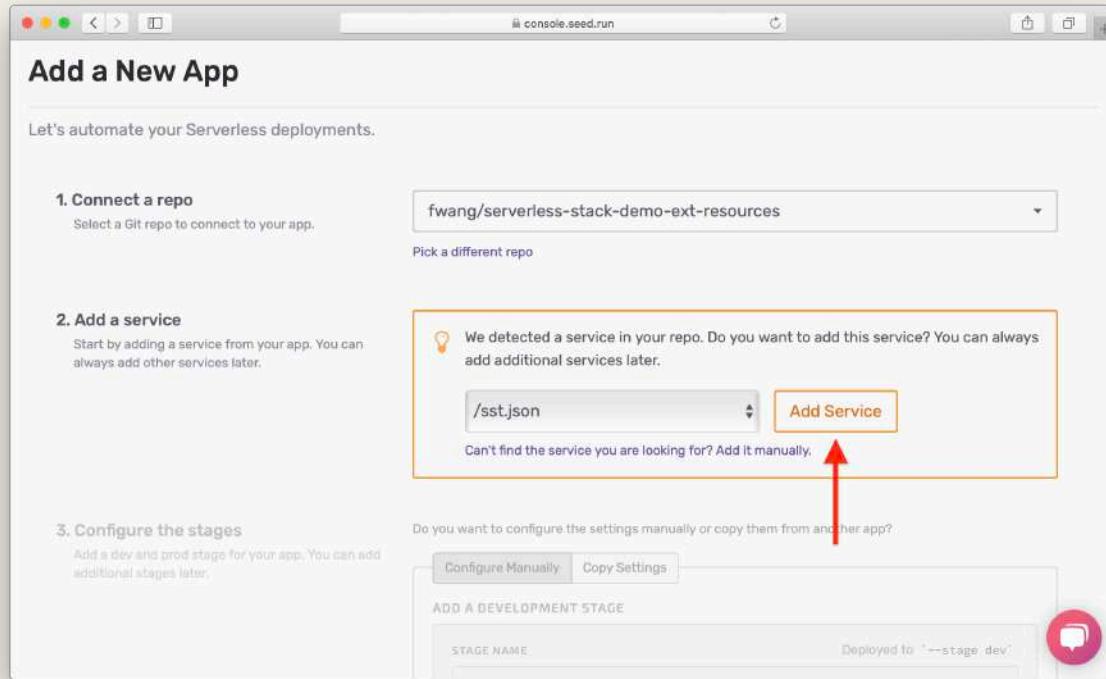
First, add the resources repo on Seed. If you haven't yet, you can create a free account [here](#).

Go in to your [Seed account](#), add a new app, authenticate with GitHub, search for the resources repo, and select it.



Search for Git repository

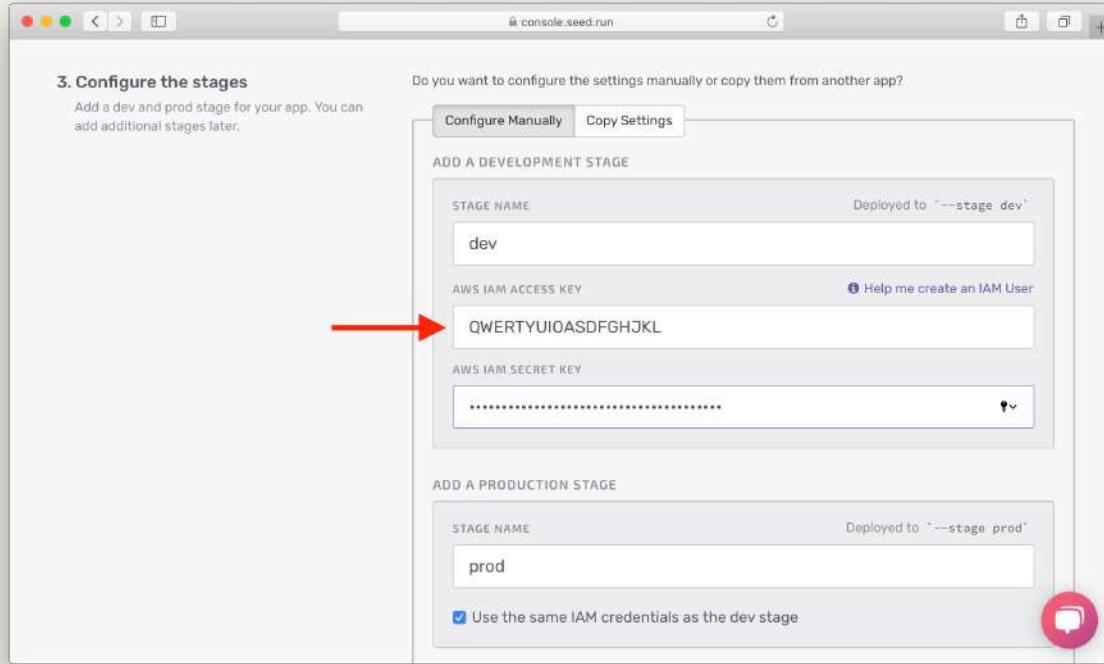
Seed will now automatically detect the SST service in the repo. After detection, select **Add Service**.



Select Serverless service to add

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.

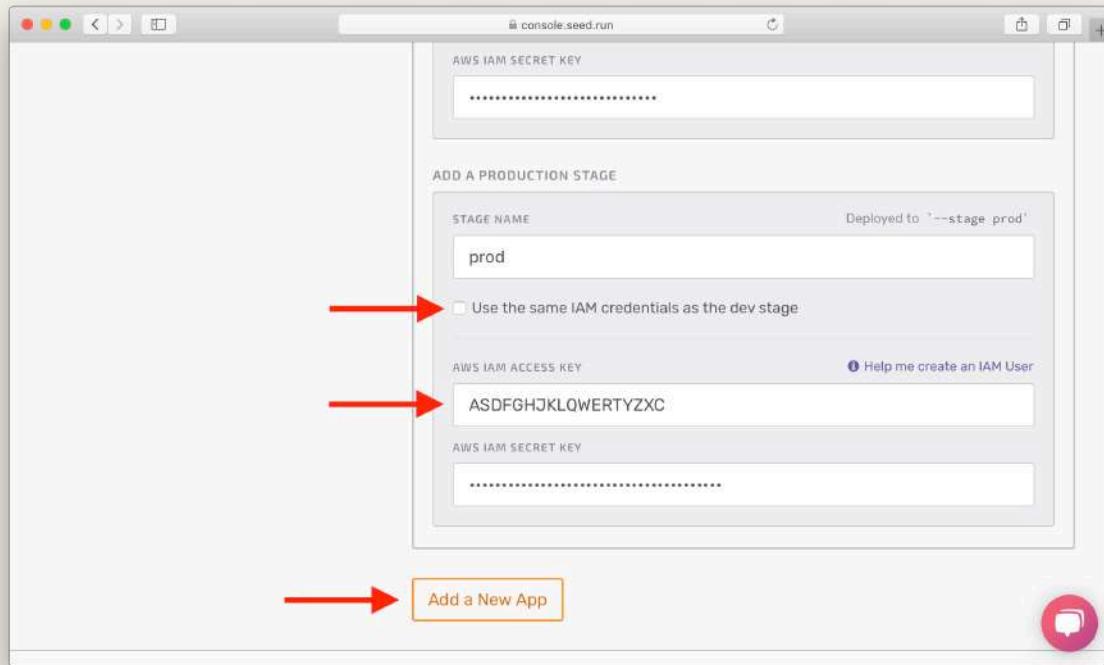
Let's first configure the **Development** stage. Enter: - **Stage Name:** dev - **AWS IAM Access Key** and **AWS IAM Secret Key:** the IAM credentials of the IAM user you created in your **Development** AWS account above.



Set dev stage IAM credentials

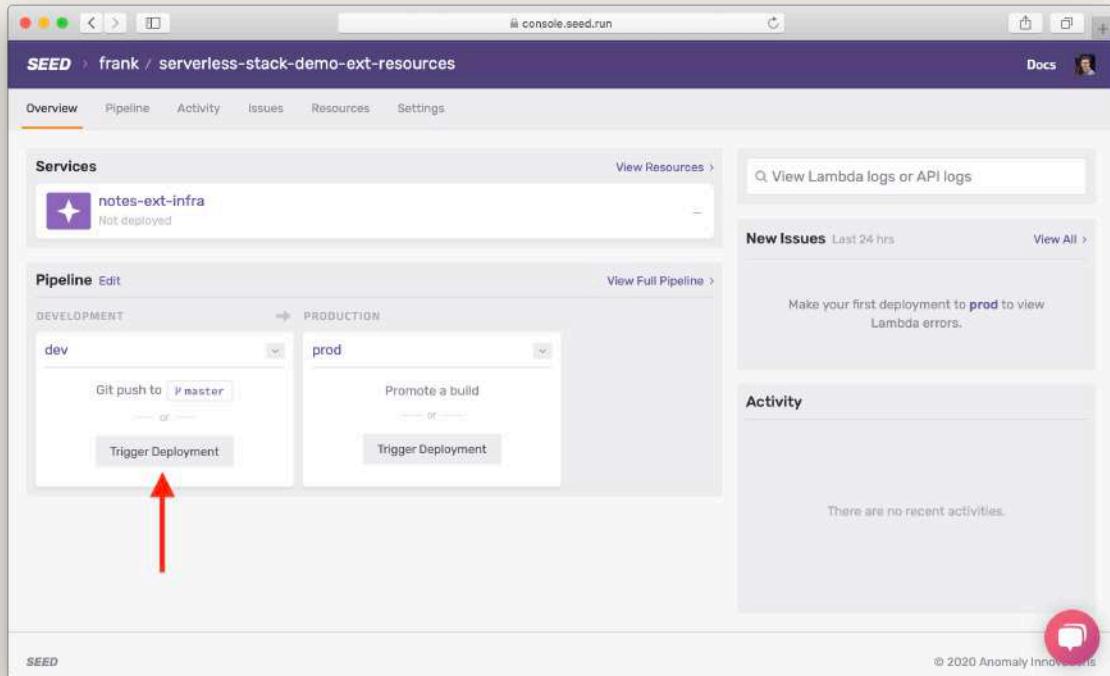
Next, let's configure the **Production** stage. Uncheck **Use the same IAM credentials as the dev stage** checkbox since we want to use a different AWS account for **Production**. Then enter: - **Stage Name:** prod - **AWS IAM Access Key** and **AWS IAM Secret Key:** the IAM credentials of the IAM user you created in your **Production** AWS account above.

Finally hit **Add a New App**.



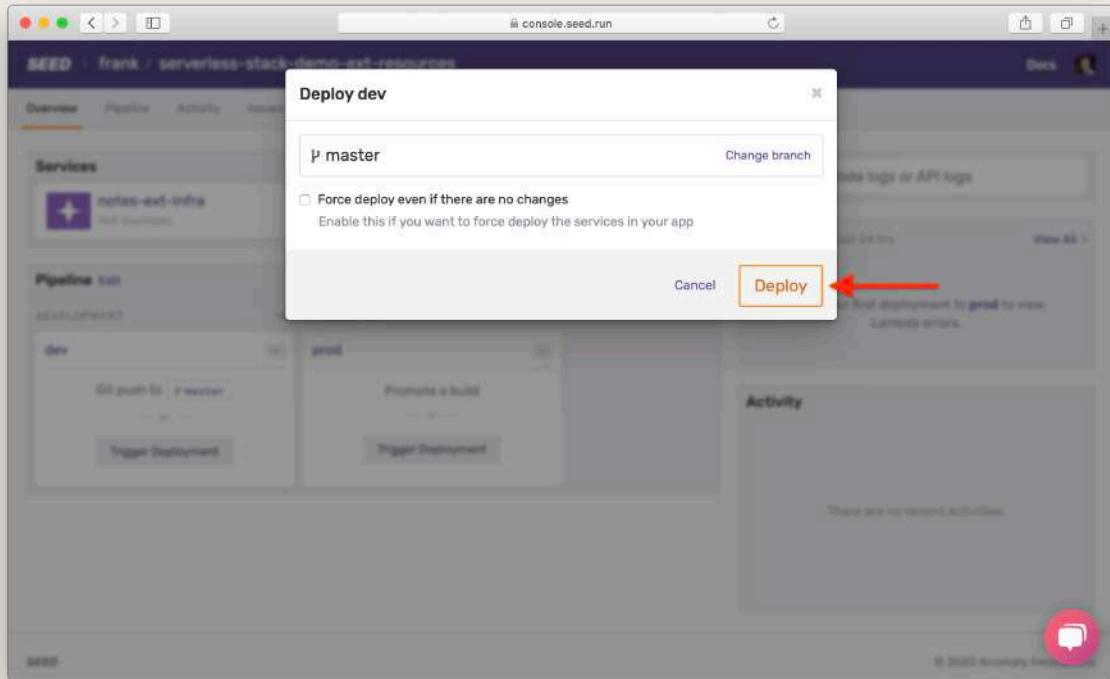
Create an App in Seed

Now let's make our first deployment. Click **Trigger Deployment** under the **dev** stage.



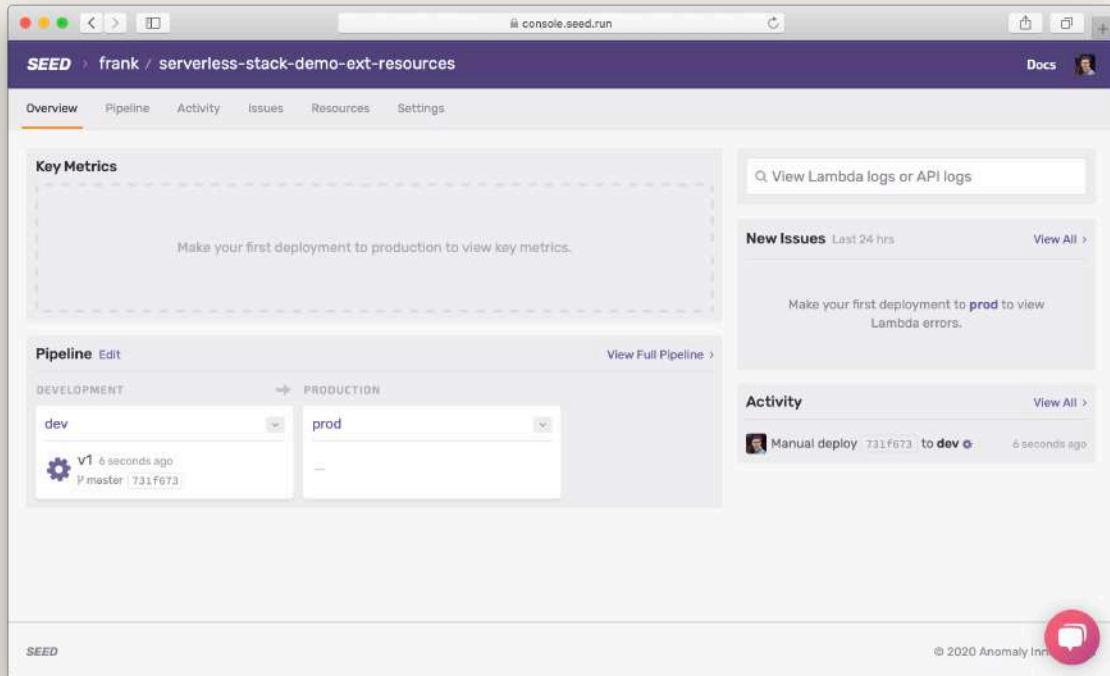
Select Deploy in dev stage

We are deploying the `master` branch here. Confirm this by clicking **Deploy**.



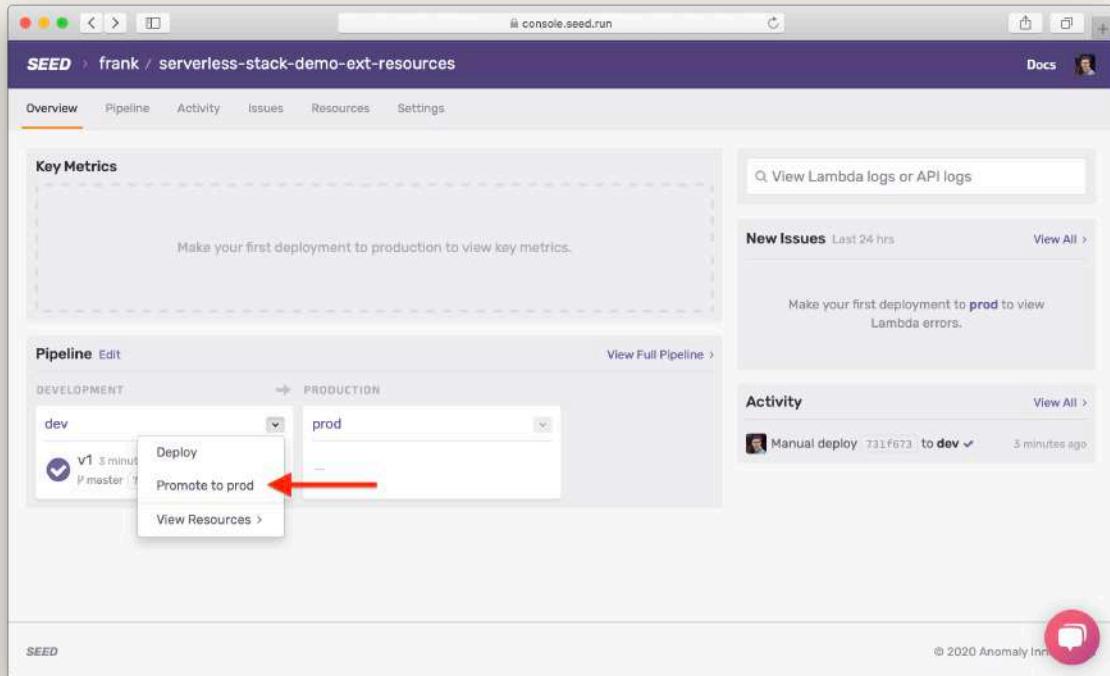
Select master branch to deploy

You'll notice that the service is being deployed.



Show service is deploying in dev stage

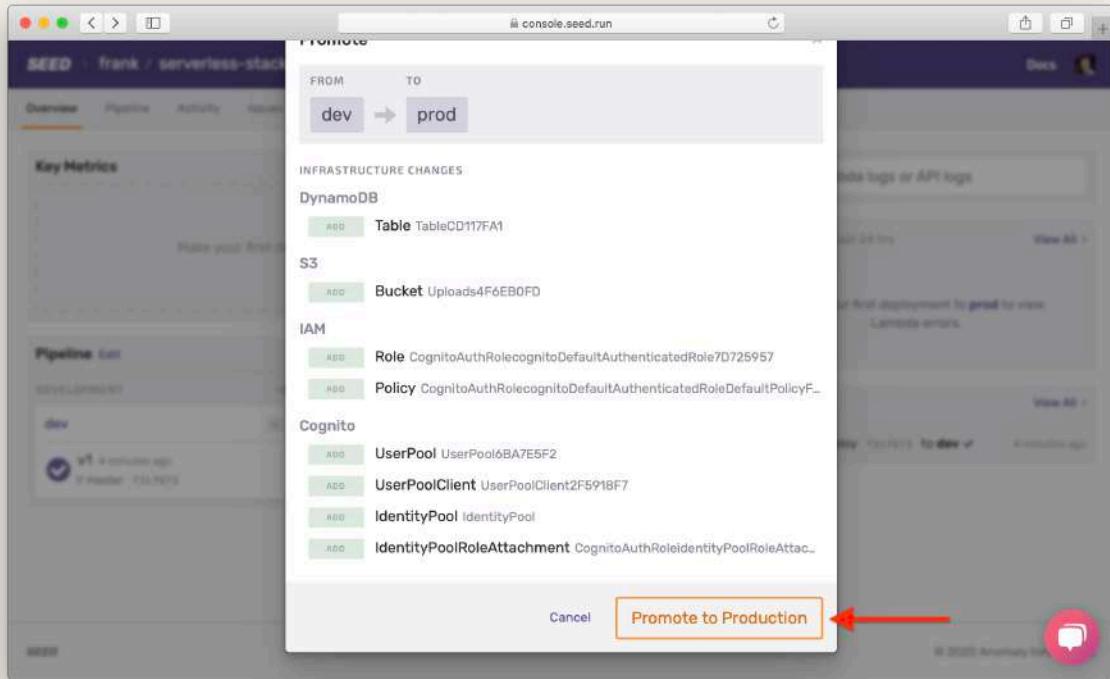
After the service is successfully deployed. Click **Promote** to deploy this to the **prod** stage.



Select Promote in dev stage

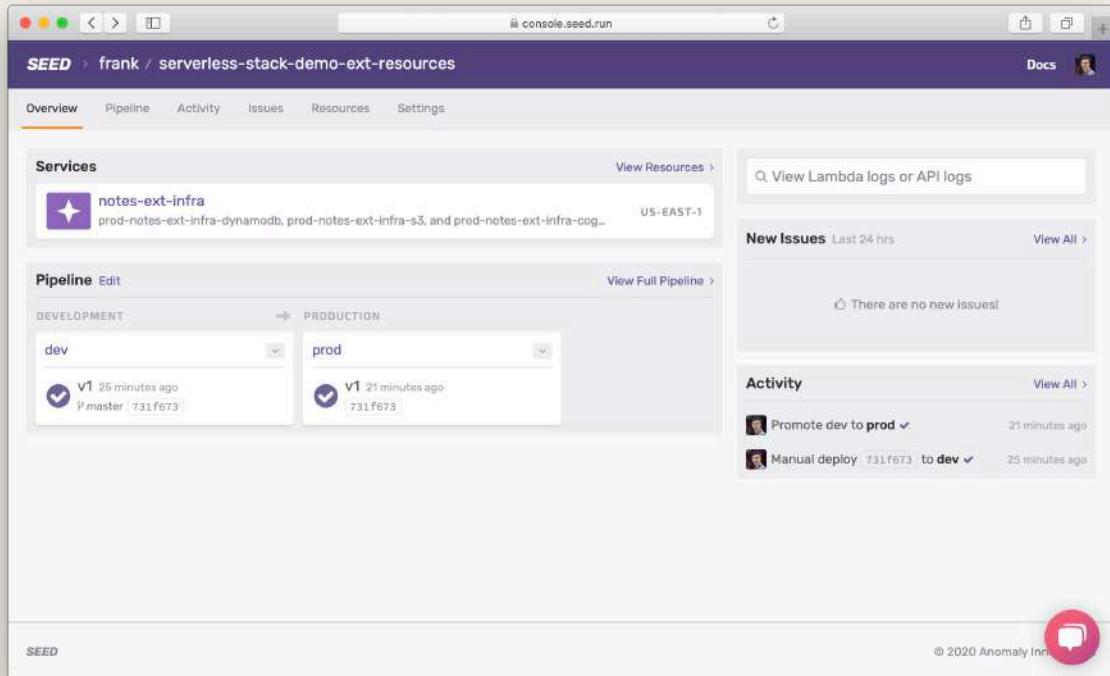
You will see a list of changes in resources. Since this is the first time we are deploying to the prod stage, the change list shows all the resources that will be created. We'll take a look at this in detail later in the [Promoting to production](#) chapter.

Click **Promote to Production**.



Promote dev stage to prod stage

Now our resources have been deployed to both **dev** and **prod**.



Show service is deployed in prod stage

Next, let's deploy our API services repo.



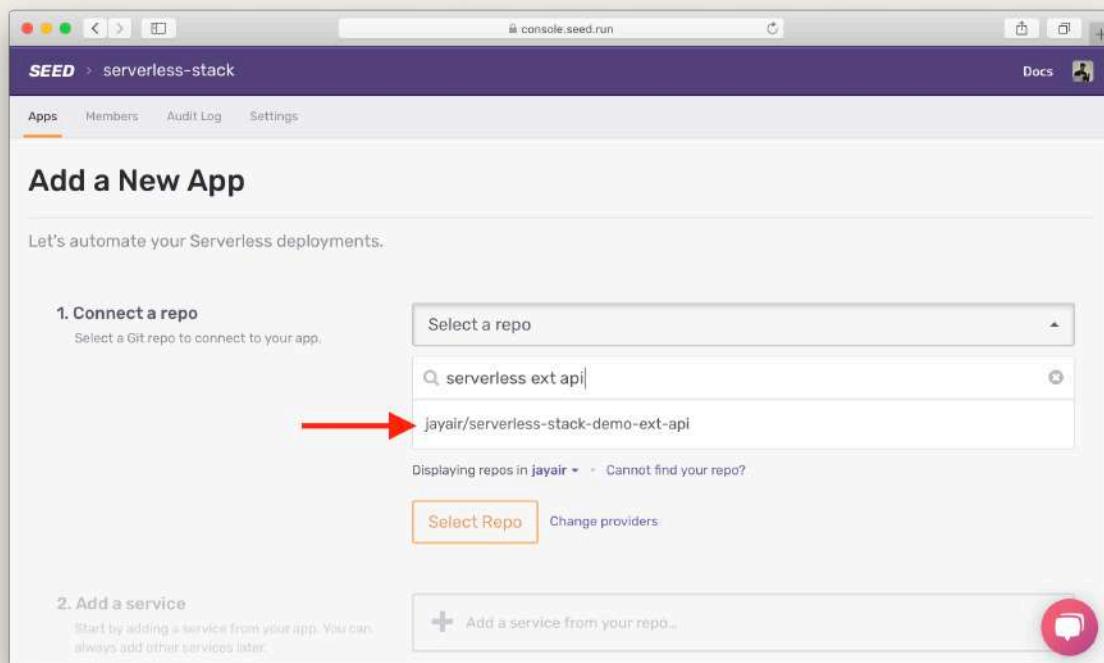
Help and discussion

View the [comments for this chapter on our forums](#)

Deploy the API Services Repo

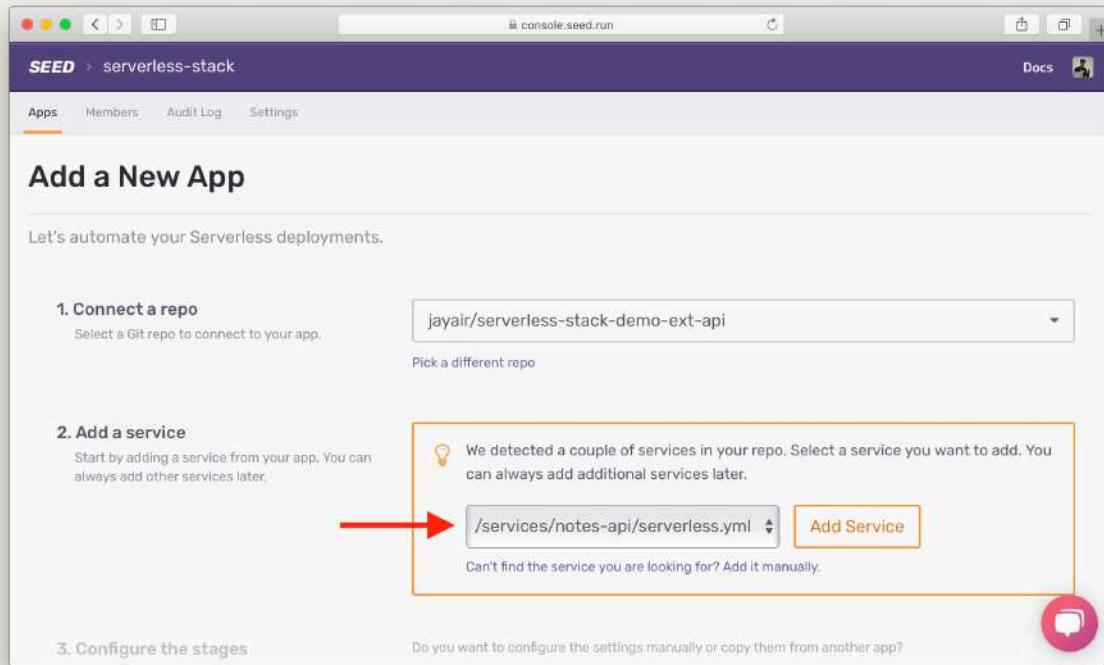
Just as the previous chapter we'll add the API repo on Seed and deploy it to our environments.

Click **Add an App** again, and select your Git provider. This time, select the API repo.



Select Add an App in Seed

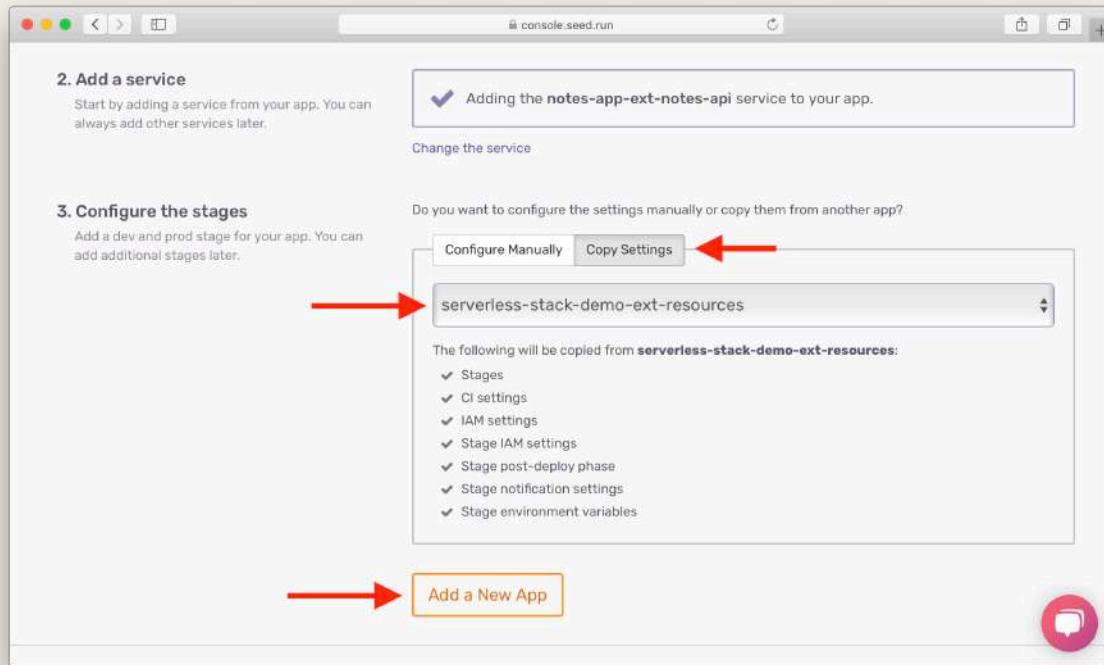
Select the **notes-api** service from the list of services.



Select Serverless service to add

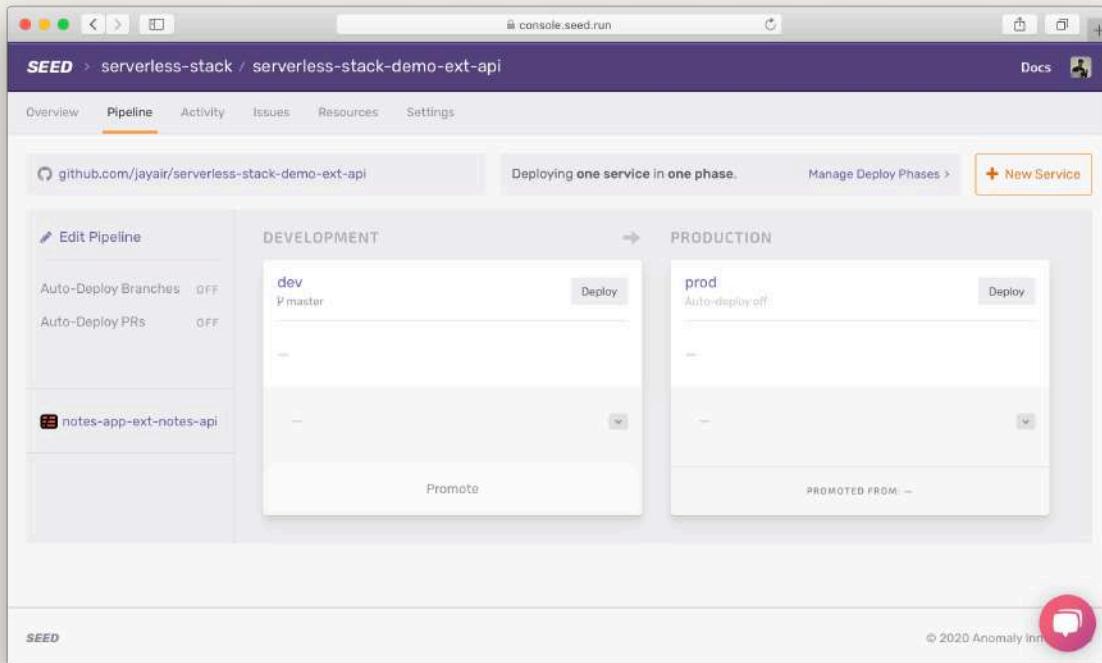
The environments for our API repo are identical to our resources repo. So instead of manually configuring them, we'll copy the settings.

Select **Copy Settings** tab, and select the resources app. Then hit **Add a New App**.



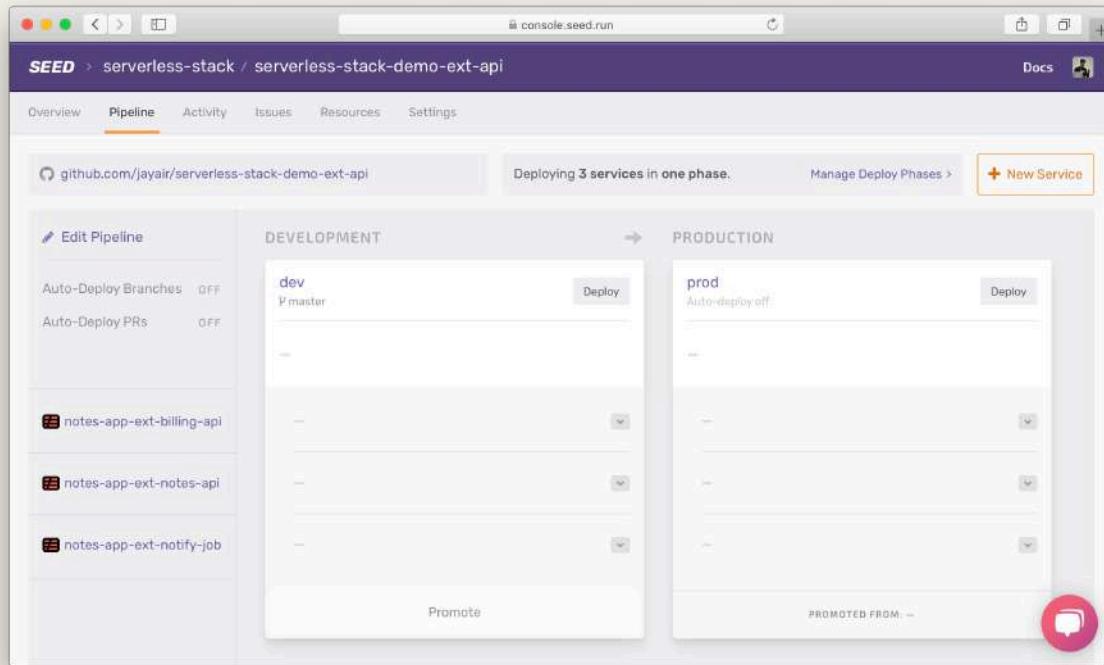
Set app settings from resources

The API app has been created. Now, let's add the other services. Head over to the **Pipeline** tab.



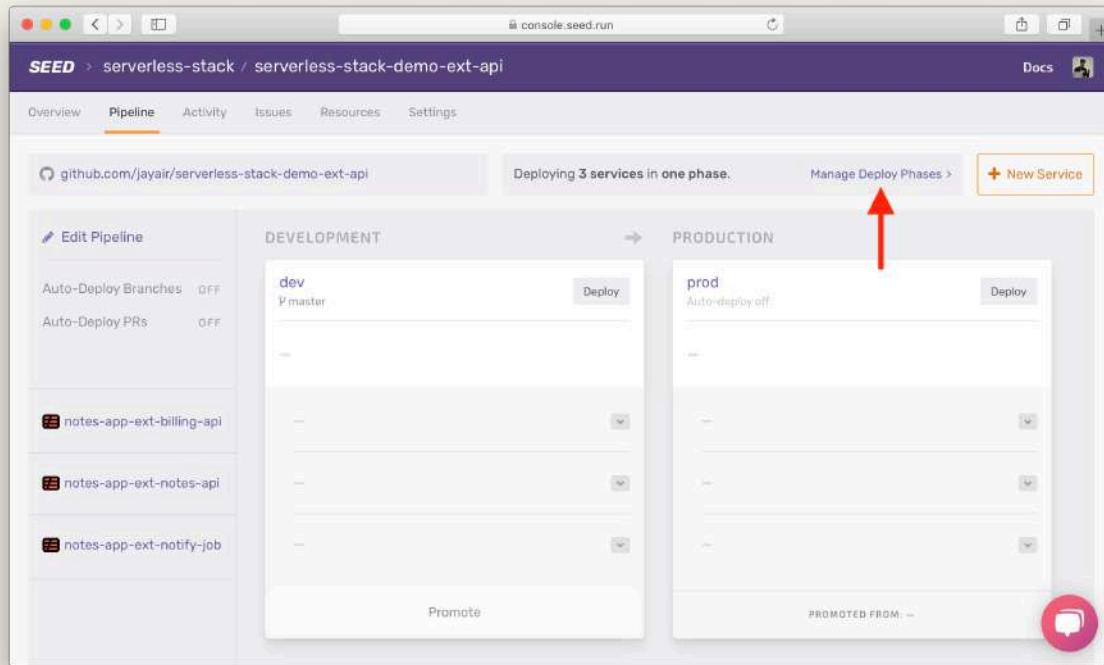
Create an App in Seed

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.



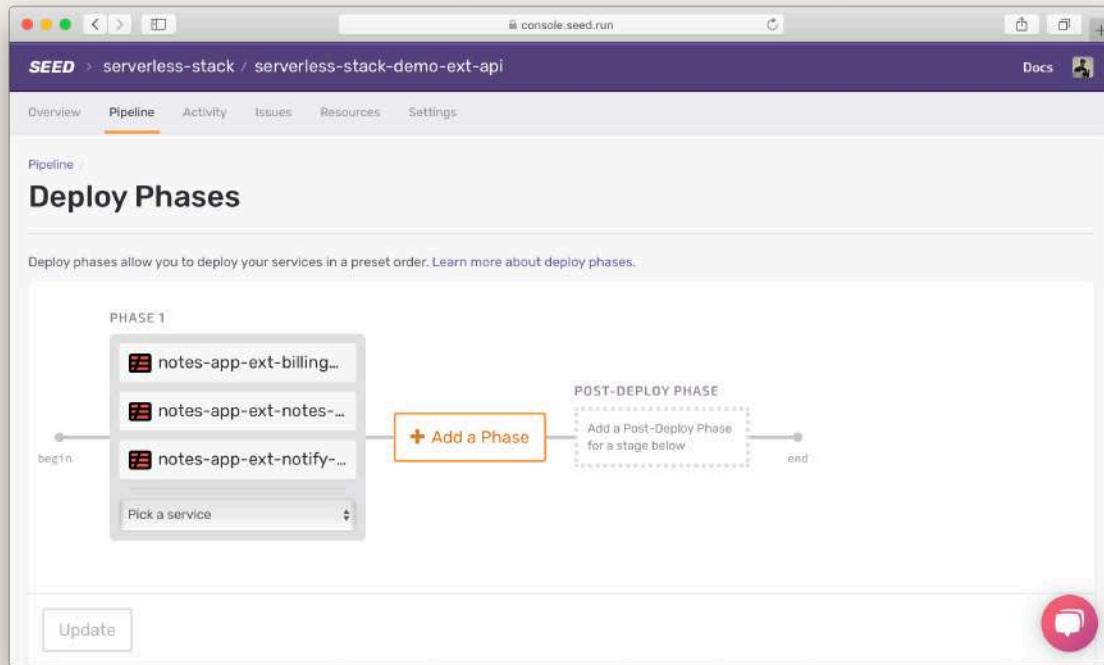
Added all services in Seed

Next, click on **Manage Deploy Phases**.



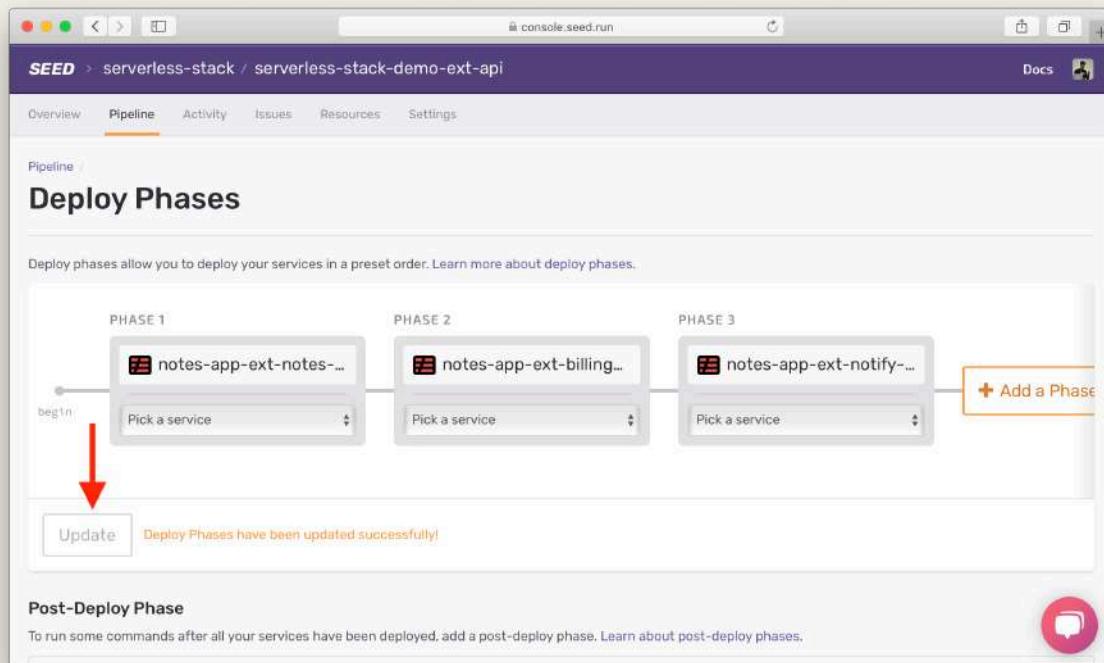
Hit Manage Deploy Phases screenshot

Again you'll notice that by default all the services are deployed concurrently.



Default Deploy Phase screenshot

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 to add 2 phases. And move the **billing-api** service to **Phase 2**, and the **notify-job** service to **Phase 3**. Finally, click **Update Phases**.



Edit Deploy Phase screenshot

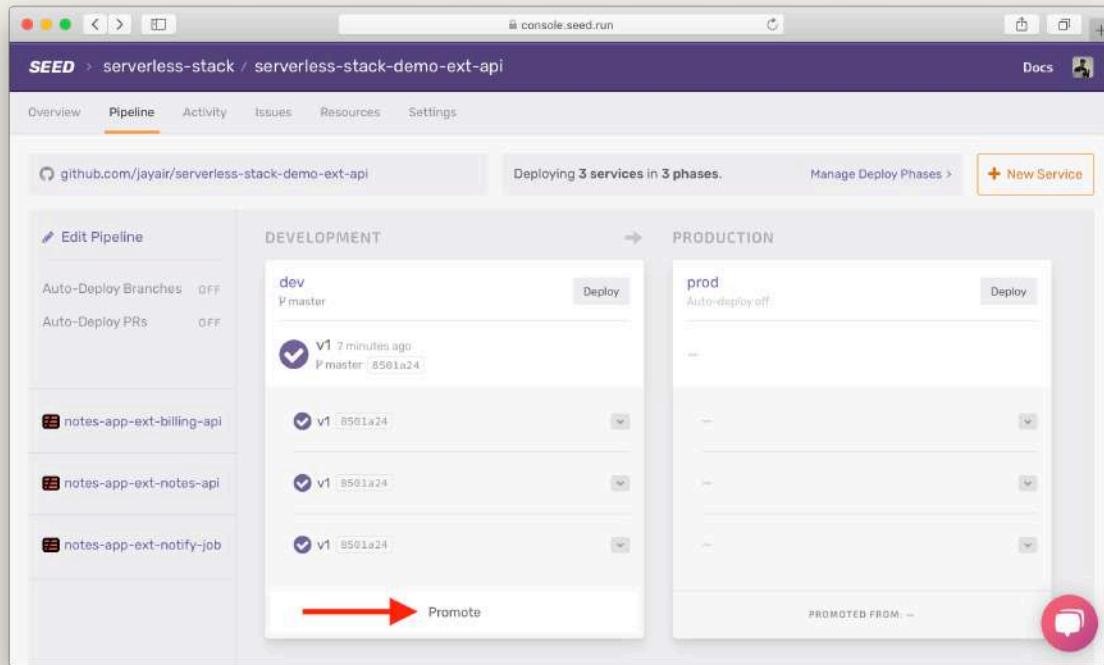
Now let's make our first deployment.

The screenshot shows a browser window titled "console.seed.run" displaying a deployment summary for "Build v1" in the "dev" stage. At the top, it says "Jay manually deployed to dev". Below that is a table with two columns: "STATUS" and "DATE". The "STATUS" column shows a green checkmark and "Completed", and the "DATE" column shows "Jul 7, 1:32:08 PM". Under the "COMMIT" section, it lists "Initial commit" by "Jay" with a commit hash "8501a24". The "WORKFLOW" section shows three sequential steps: "notes-app-ext-note", "notes-app-ext-billing", and "notes-app-ext-notif". Each step has a green checkmark, a timestamp (e.g., "1m 23s"), and a "View logs" button. A message bubble icon is visible in the bottom right corner.

Show services are deploying in dev stage

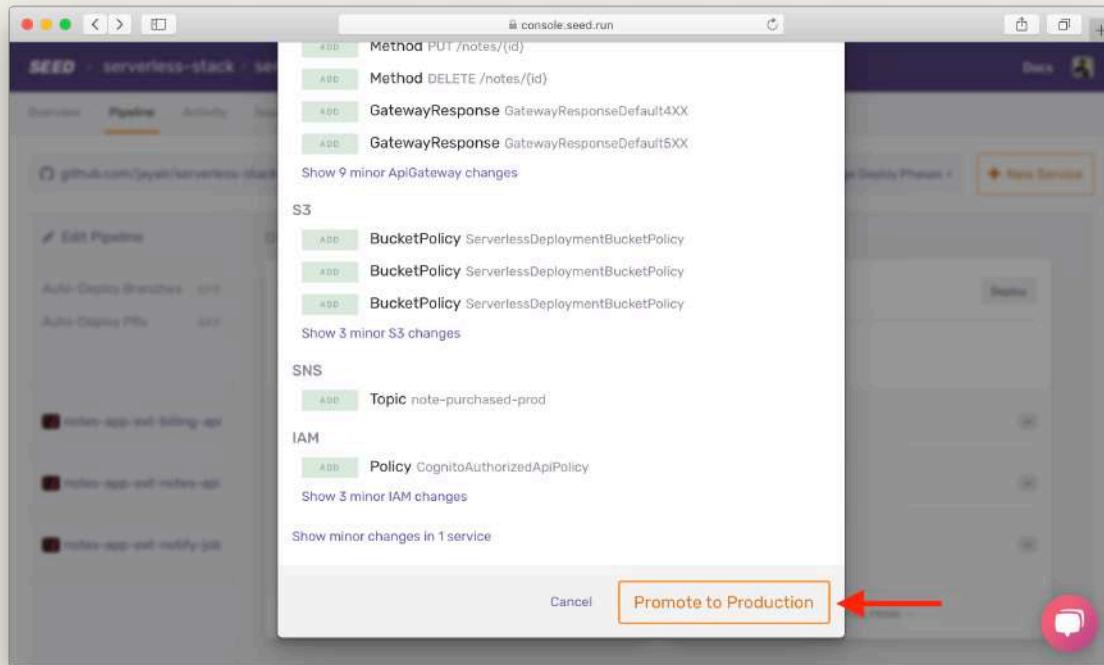
You can see the deployments were carried out according to the specified deploy phases.

Just as before, promote **dev** to **prod**.



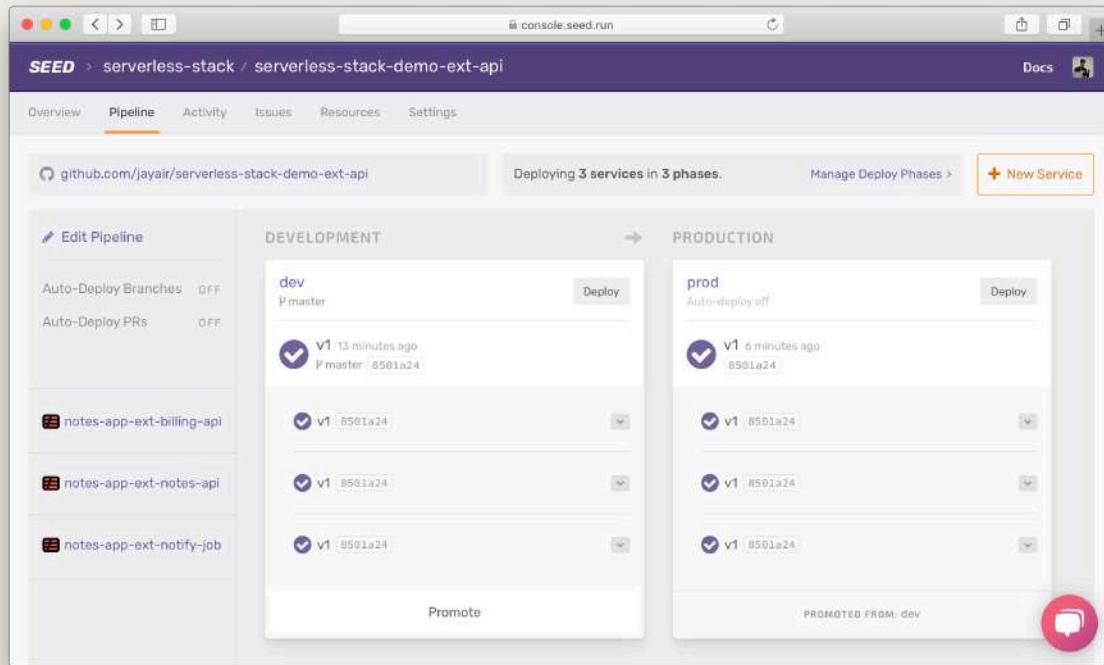
Select Promote in dev stage

Hit **Promote to Production**.



Promote dev stage to prod stage

Now we have the API deployed to both **dev** and **prod**.



Show services are deployed in prod stage

Now that our entire app has been deployed, let's look at how we are sharing environment specific configs across our services.



Help and discussion

View the [comments](#) for this chapter on our forums

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](#).

1. We have two repos — `serverless-stack-demo-ext-resources` and `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.

We need our infrastructure resources and API services environments to be mapped according to this scheme.

API	Resources
prod	prod
dev	dev
featureX	dev
pr#12	dev
etc...	dev

So we want all of the ephemeral stages in our API services to share the `dev` version of the resources.

Let's look at how to do that.

Link The Environments Across Apps

In our `serverless.common.yml` (the part of the config that is shared across all our Serverless services), we are going to link to our resources app. You'll recall that our resources are configured using CDK and deployed using SST.

First we start by defining the stage that our Serverless services are going to be deployed to.

```
custom:  
  # Our stage is based on what is passed in when running serverless  
  # commands. Or falls back to what we have set in the provider section.  
  stage: ${opt:stage, self:provider.stage}
```

Next we create a simple mapping of the dev and prod stage names between our Serverless services and SST app.

```
sstAppMapping:  
  prod: prod  
  dev: dev
```

This seems a bit redundant because the stage names we are using across the two repos are the same. But you might choose to call it dev in your Serverless services. And call it development in your SST app.

Next, we create the reference to our SST app.

```
sstApp: ${self:custom.sstAppMapping.${self:custom.stage}},  
  ↵  self:custom.sstAppMapping.dev}-notes-ext-infra
```

Let's look at this in detail.

- First let's understand the basic format, `${VARIABLE}-notes-ext-infra`.
- The `notes-ext-infra` is hardcoded to the name of our SST app. As listed in the `sst.json` in our [resources repo](#).
- The `${VARIABLE}` format allows us to also specify a fallback. So in the case of `${VARIABLE_1, VARIABLE_2}`, it'll first try `VARIABLE_1`. If it doesn't resolve then it'll try `VARIABLE_2`.

- So Serverless will first try to resolve, `self:custom.sstAppMapping.${self:custom.stage}`. It'll check if the stage we are currently deploying to (`self:custom.stage`) has a mapping set in `self:custom.sstAppMapping`. If it does, then it uses it. In other words, if we are currently deploying to dev or prod, then use the corresponding stage in our SST app.
 - If the stage we are currently deploying to does not have a corresponding stage in our SST app (not dev or prod), then we fallback to `self:custom.sstAppMapping.dev`. As in, we fallback to using the dev stage of our SST app.

This allows us to map our environments correctly across our Serverless services and SST apps.

For reference, here's what the top of our `serverless.common.yml` looks like:

```
custom:  
  # Our stage is based on what is passed in when running serverless  
  # commands. Or falls back to what we have set in the provider section.  
  stage: ${opt:stage, self:provider.stage}  
  
sstAppMapping:  
  prod: prod  
  dev: dev  
  
sstApp: ${self:custom.sstAppMapping.${self:custom.stage}},  
  ↵  self:custom.sstAppMapping.dev}-notes-ext-infra
```

Now we are going to use the resources based on the `sstApp`. Open up the `serverless.yml` file in the `notes-api` service.

•

```
custom: ${file(..../serverless.common.yml):custom}
```

```
provider:  
  environment:  
    stage: ${self:custom.stage}  
    tableName: !ImportValue ${self:custom.sstApp}-ExtTableName  
...  
...
```

The `!ImportValue ${self:custom.sstApp}-ExtTableName` line allows us to import the CloudFormation export from the appropriate stage of our SST app. In this case we are importing the name of the DynamoDB table that's been created.

The `provider:` and `environment:` options allow us to add environment variables to our Lambda functions. Recall that we can access this via the `process.env.tableName` variable at runtime.

So in our `list.js`, we'll read the `tableName` from the `environment` variable `process.env.tableName`.

```
const params = {
  TableName: process.env.tableName,
  KeyConditionExpression: "userId = :userId",
  ExpressionAttributeValues: {
    ":userId": event.requestContext.identity.cognitoIdentityId
  }
};
```

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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 lay out the best practice for storing secrets and managing them across multiple environments.

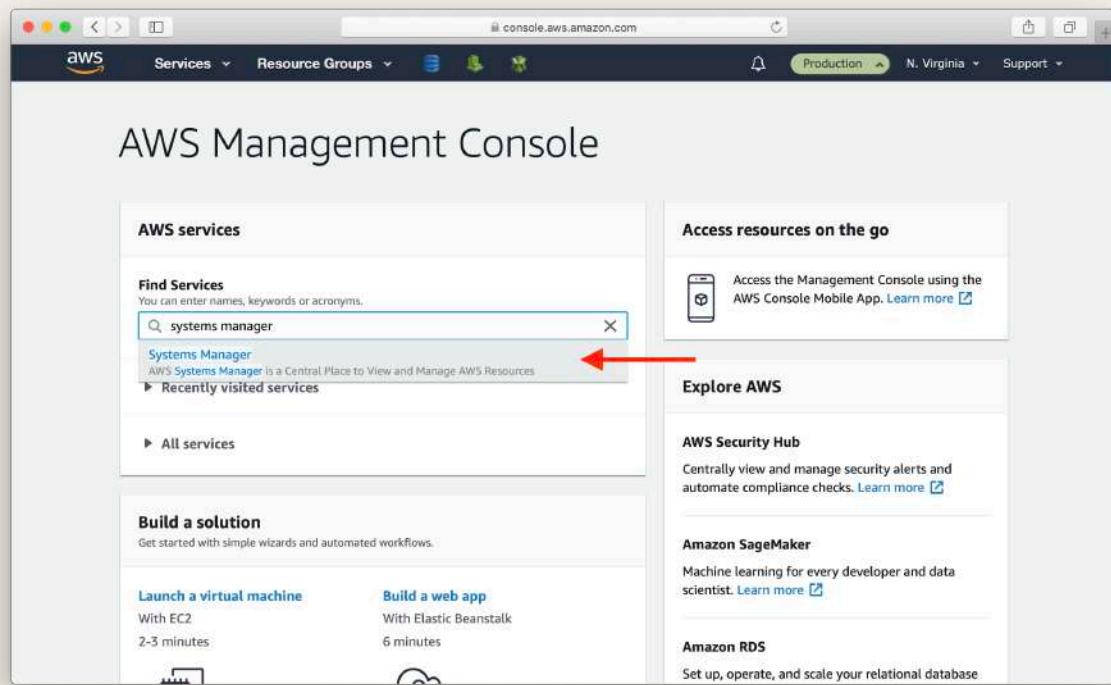
Store Secrets in AWS Parameter Store

[AWS Systems Manager Parameter Store](#) (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](#), 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:

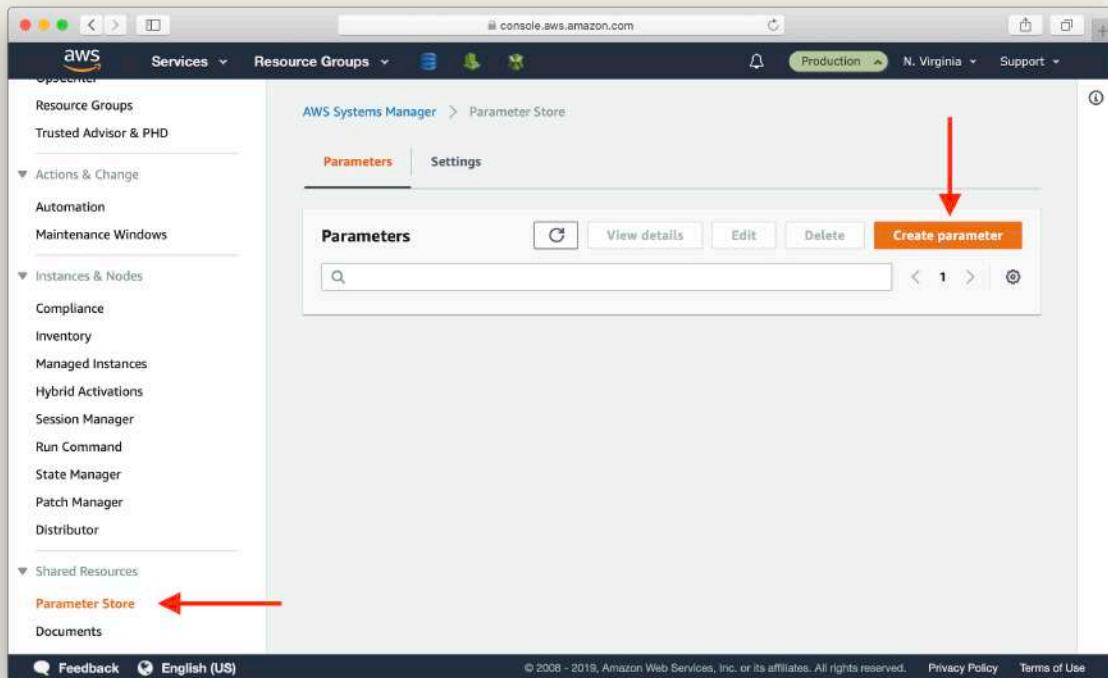
- The **live** key in the Production account's SSM console.
- The **test** key in the Development account's SSM console.

First go in to your **Production** account, and go to your Systems Manager console.



Select Systems Manager service

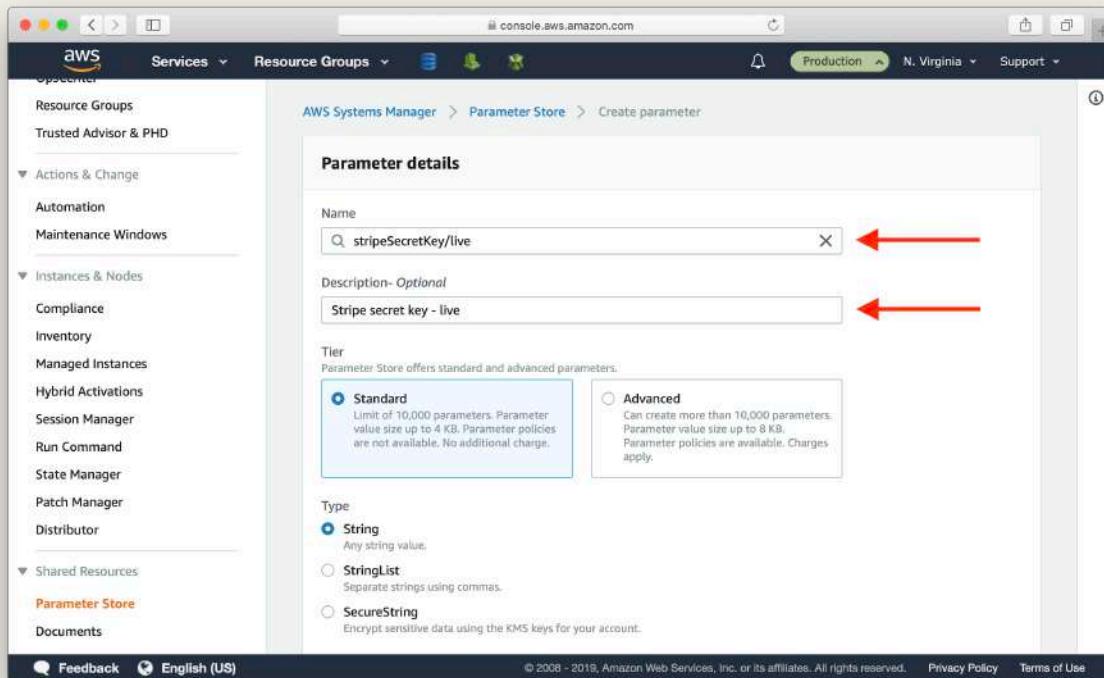
Select **Parameter Store** from the left menu, and select **Create parameter**.



Select Create parameter in Parameter Store

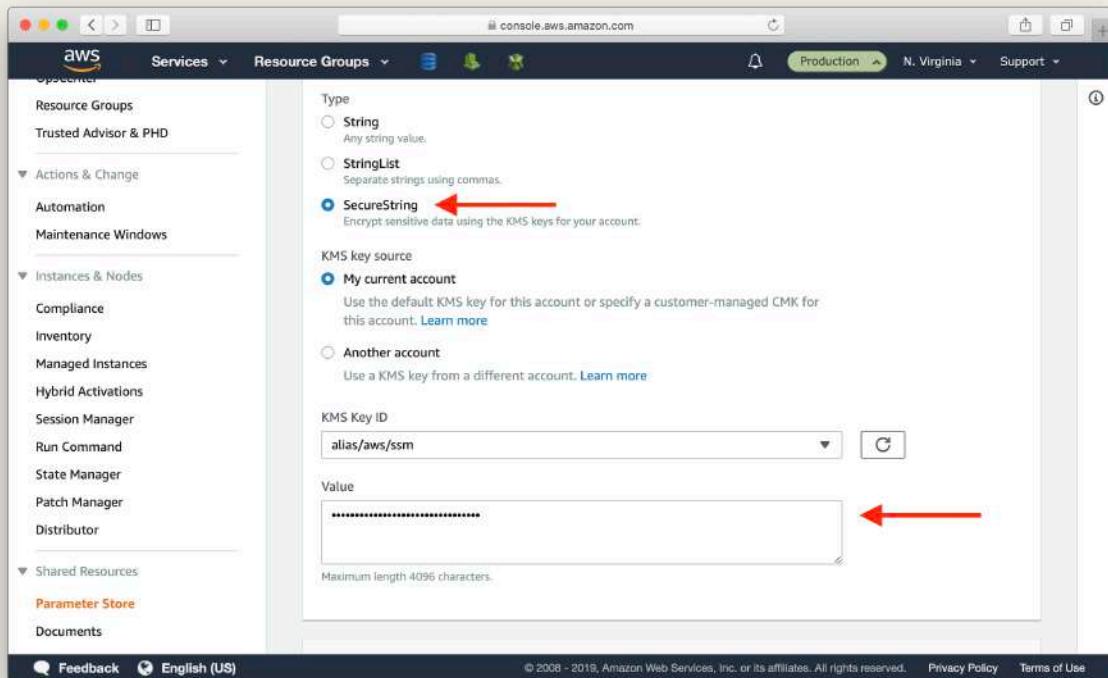
Fill in:

- **Name:** /stripeSecretKey/live
- **Description:** Stripe secret key - live



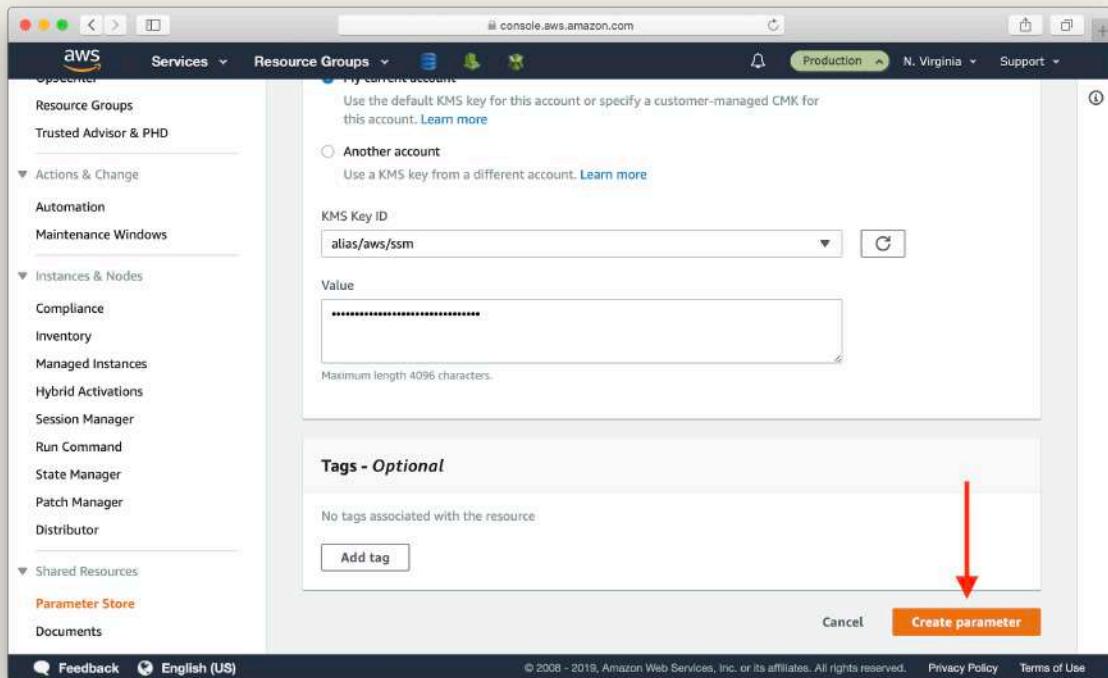
Set parameter details in Parameter Store

Select **SecureString**, and paste your live Stripe key in **Value**.



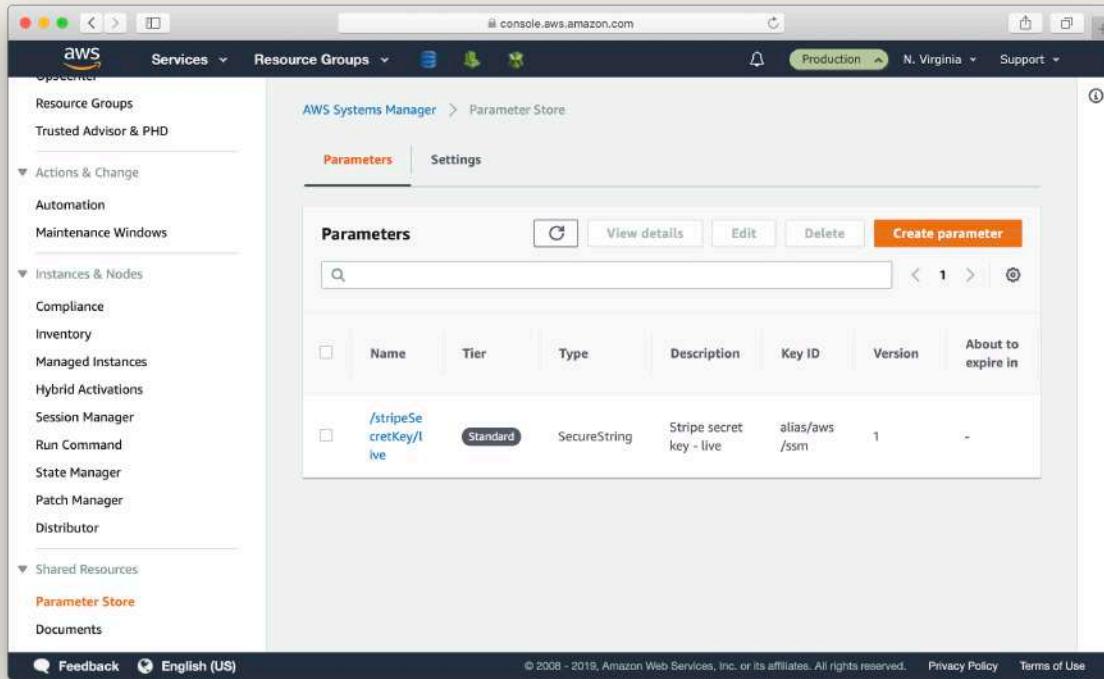
Select SecureString parameter type

Scroll to the bottom and hit **Create parameter**.



Create parameter in Parameter Store

The key is added.



The screenshot shows the AWS Systems Manager Parameter Store interface. The left sidebar lists various AWS services like Resource Groups, Trusted Advisor & PHD, Actions & Change, Instances & Nodes, and Shared Resources. Under Shared Resources, 'Parameter Store' is selected. The main area displays a table titled 'Parameters' with one row. The row contains the following data:

Name	Tier	Type	Description	Key ID	Version	About to expire in
/stripeSecretKey/live	Standard	SecureString	Stripe secret key - live	alias/aws/ssm	1	-

Show parameter created screenshot

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

The screenshot shows the AWS Systems Manager Parameter Store interface. On the left, there's a navigation sidebar with sections like Resource Groups, Trusted Advisor & PHD, Actions & Change, Instances & Nodes, Shared Resources, and Parameter Store. The Parameter Store section is currently selected. The main area is titled "Parameters" and shows a table with one row. The table columns are Name, Tier, Type, Description, Key ID, Version, and About to expire in. The single row contains the value "/stripeSecretKey/test" for the Name column, "Standard" for Tier, "SecureString" for Type, "Stripe secret key - test" for Description, "alias/aws/ssm" for Key ID, "1" for Version, and no expiration information.

Create parameter in Development account

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:
  - ${file(../../serverless.common.yml):lambdaPolicyXRay}
  - Effect: Allow
    Action:
      - ssm:GetParameter
    Resource:
      !Sub
        'arn:aws:ssm:${AWS::Region}:${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 adminPhoneNumber = "+14151234567";

const stageConfigs = {
  dev: {
    stripeKeyName: "/stripeSecretKey/test"
  },
  prod: {
    stripeKeyName: "/stripeSecretKey/live"
  }
};

const config = stageConfigs[stage] || stageConfigs.dev;

export default {
  stage,
  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](#).

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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:

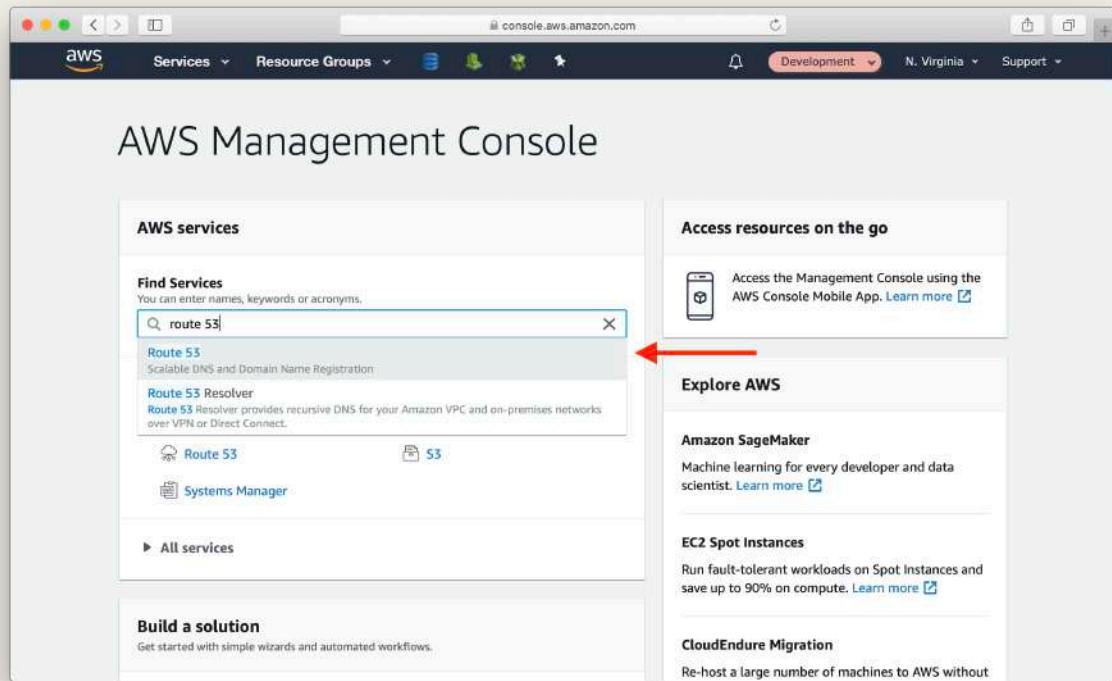
- prod → ext-api.serverless-stack.com
- dev → dev.ext-api.serverless-stack.com

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

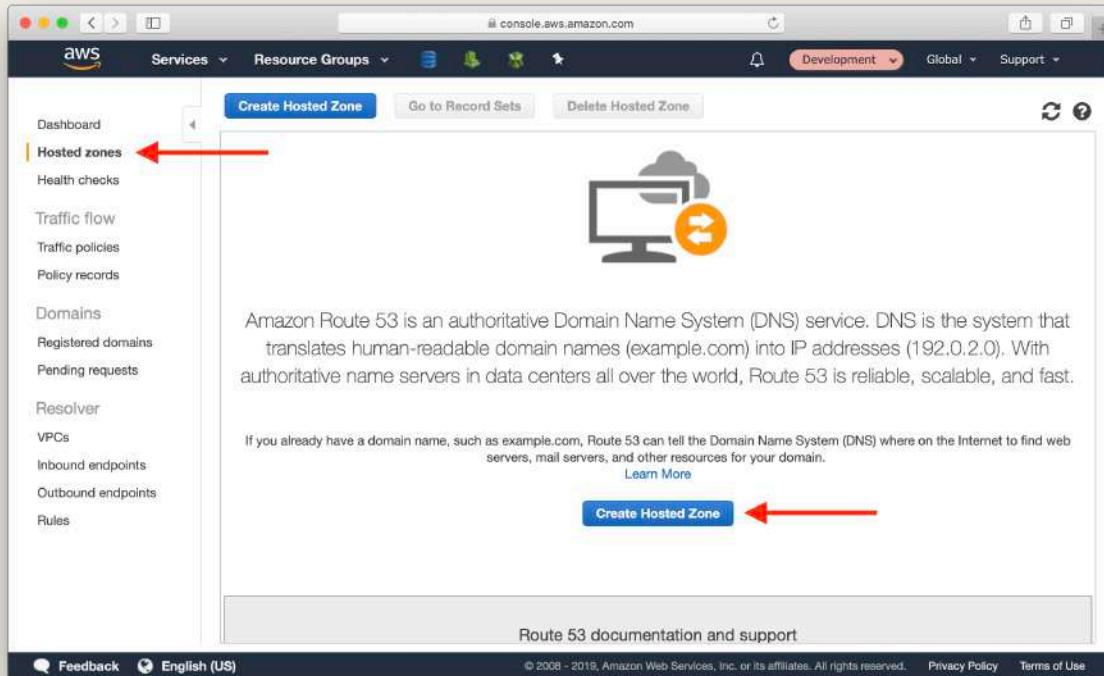
We are going to have to delegate the subdomain dev.ext-api.serverless-stack.com to be hosted in the Development AWS account. Just a quick note, as you follow these steps, pay attention to the account name shown at the top right corner of the screenshot. It'll tell you which account we are working with.

First, go into your Route 53 console in your Development account.



Select Route 53 service

Click **Hosted zones** in the left menu. Then select **Create Hosted Zone**.

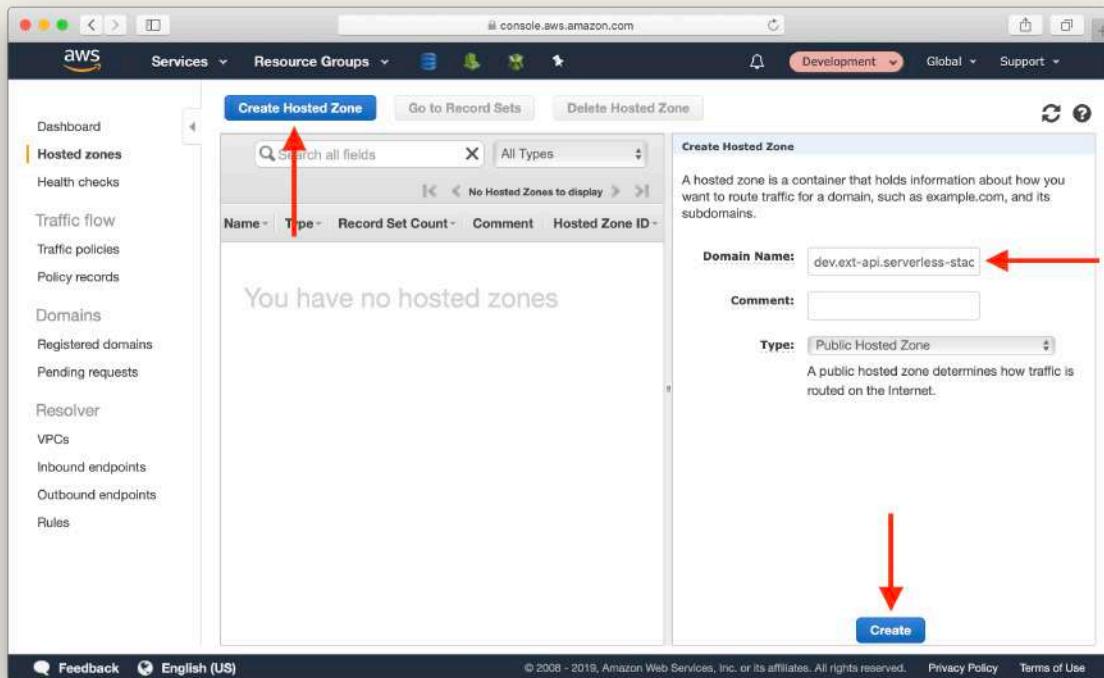


Select Create Hosted Zone

Select **Create Hosted Zone** at the top. Enter:

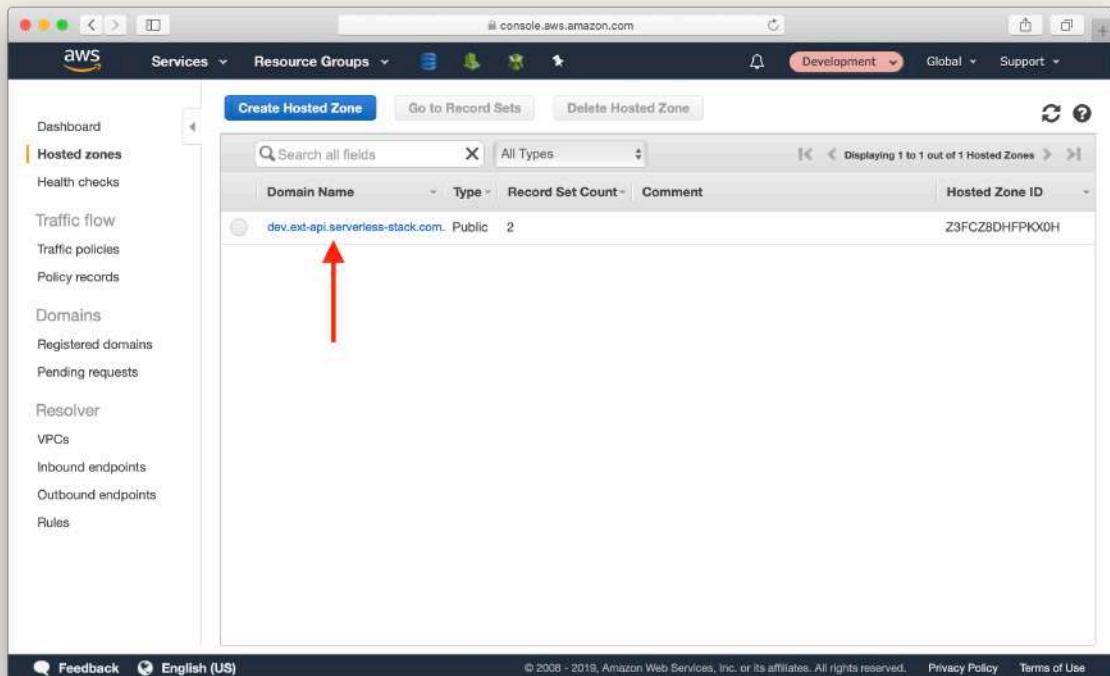
- **Domain Name:** dev.ext-api.serverless-stack.com

Then click **Create**.



Created Hosted Zone in Route 53

Select the zone you just created.



The screenshot shows the AWS Route 53 service in the AWS Management Console. The left sidebar is collapsed, and the main area displays a table of hosted zones. One row is visible:

Domain Name	Type	Record Set Count	Comment	Hosted Zone ID
dev.ext-api.serverless-stack.com.	Public	2		Z3FCZ8DHFPKX0H

Select Hosted Zone in Route 53

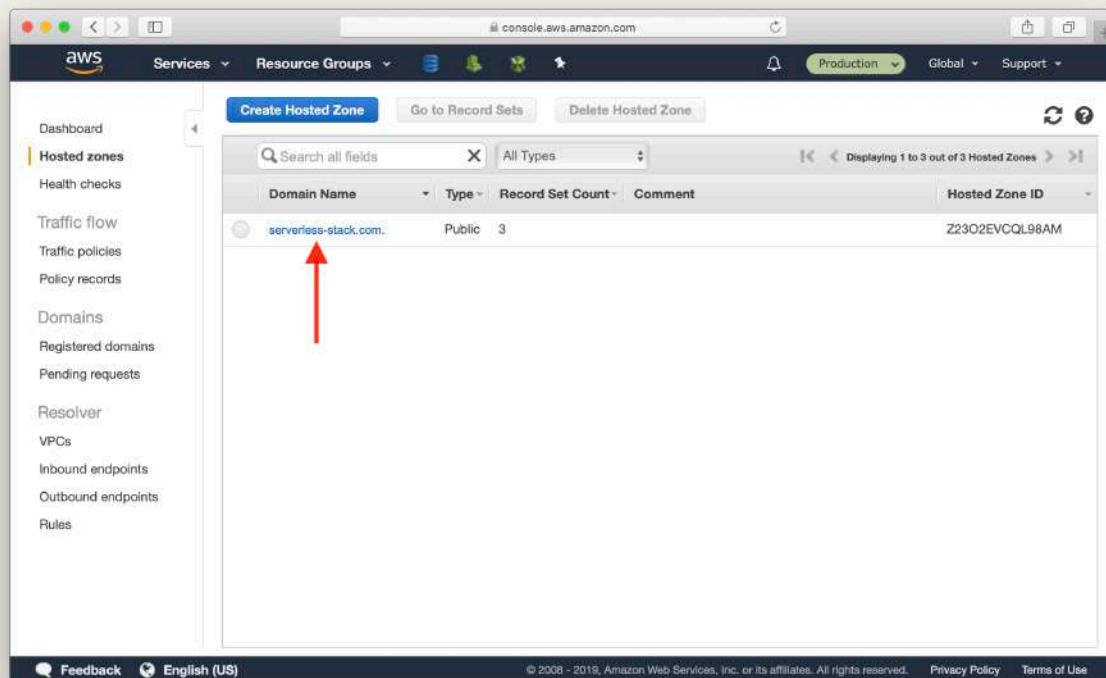
Click on the row with **NS** type. And copy the 4 lines in the **Value** field. We need this in the steps after.

The screenshot shows the AWS Route 53 service in the AWS Management Console. The left sidebar lists various options like Dashboard, Hosted zones, Traffic flow, and Domains. The main area shows a table of record sets for the domain 'dev.ext-api.serverless-stack.com'. One record set is selected, showing its details. The 'Type' is listed as 'NS - Name server'. The 'Value' field contains four name servers: 'ns-849.awsdns-42.net.', 'ns-477.awsdns-59.com.', 'ns-1876.awsdns-42.co.uk.', and 'ns-1103.awsdns-09.org.'. A red arrow points to the 'Value' input field. At the bottom right of the record set details, there is a 'Save Record Set' button.

Show NS Record Set in Route 53

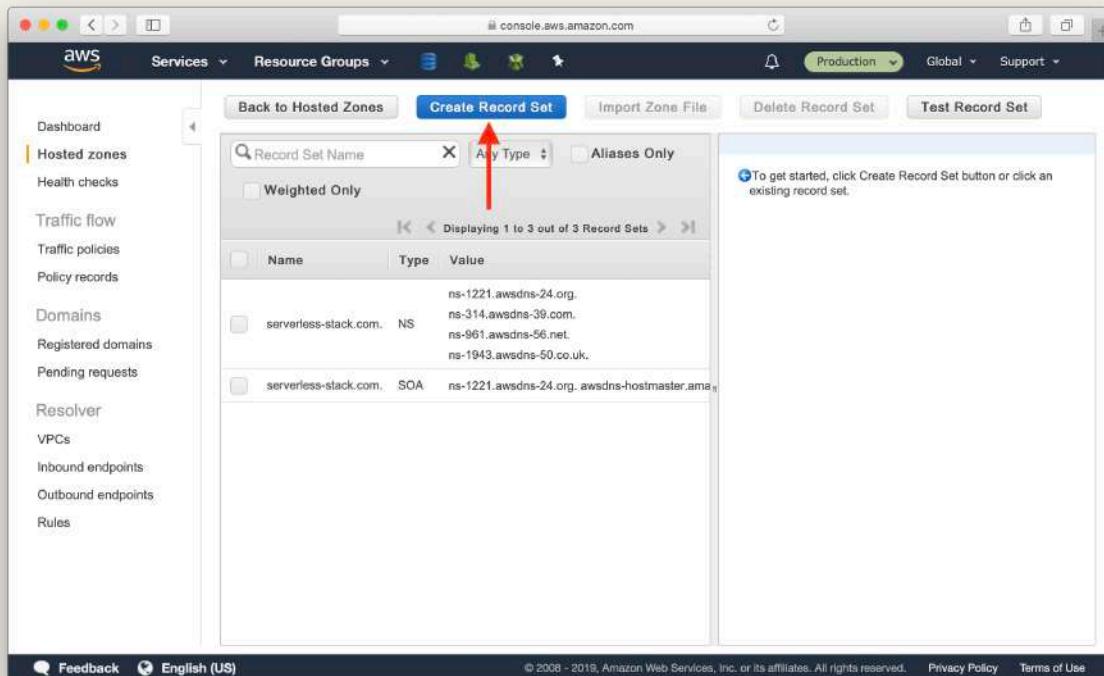
Now, switch to the Production account where the domain is hosted. And go into Route 53 console.

Select the domain.



Select Route 53 in Production account

Click **Create Record Set**.



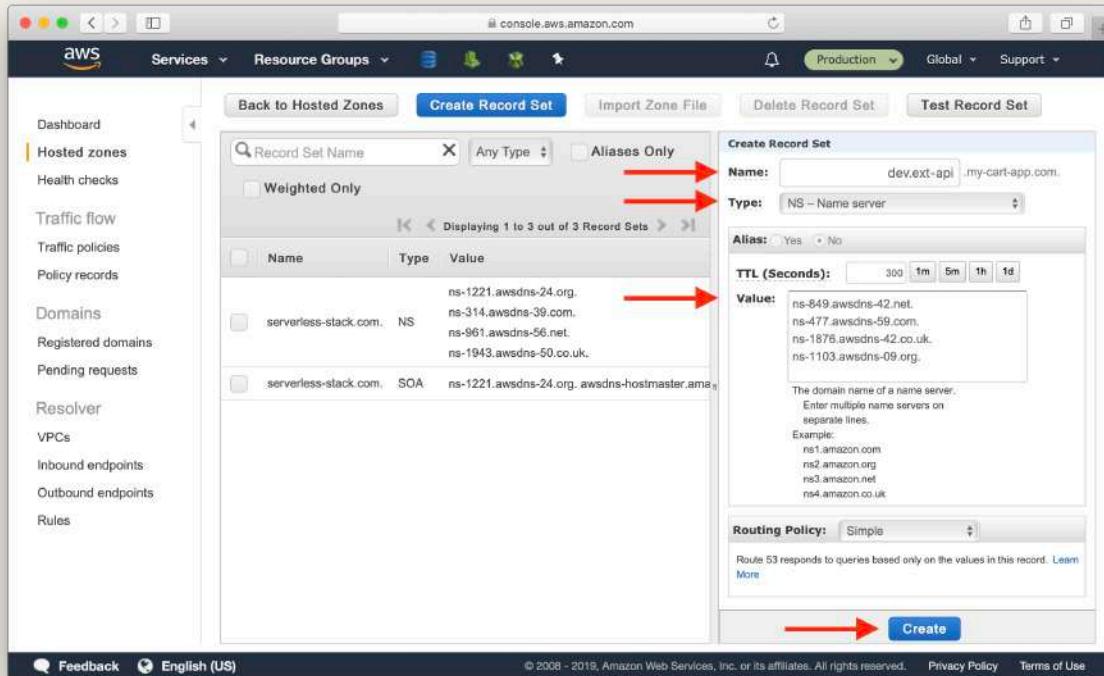
Select Create Record Set

Fill in:

- **Name:** dev.ext-api
- **Type:** NS - Name server

And paste the 4 lines from above in the **Value** field.

Click **Create**.



Created Record Set in Route 53

You should see a new `dev.ext-api.serverless-stack.com` row in the table.

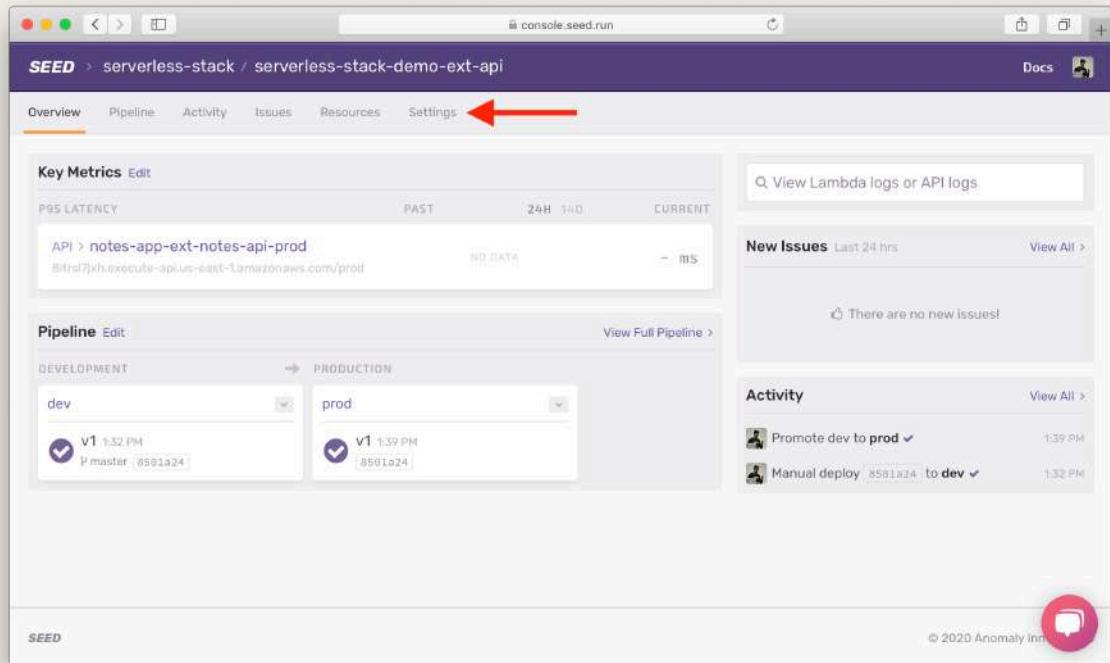
The screenshot shows the AWS Route 53 console with the 'Hosted zones' section selected. A list of record sets is displayed, including:

Name	Type	Value
serverless-stack.com.	NS	ns-1221.awsdns-24.org. ns-314.awsdns-39.com. ns-961.awsdns-56.net. ns-1943.awsdns-50.co.uk.
serverless-stack.com.	SOA	ns-1221.awsdns-24.org. awsdns-hostmaster.root-3143.awsdns-39.com. ns-849.awsdns-42.net. ns-477.awsdns-59.com. ns-1876.awsdns-42.co.uk. ns-1103.awsdns-09.org.
dev.ext-api.serverless-stack.com.	NS	

Show subdomain delegated to Development account

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](#) for the dev stage.

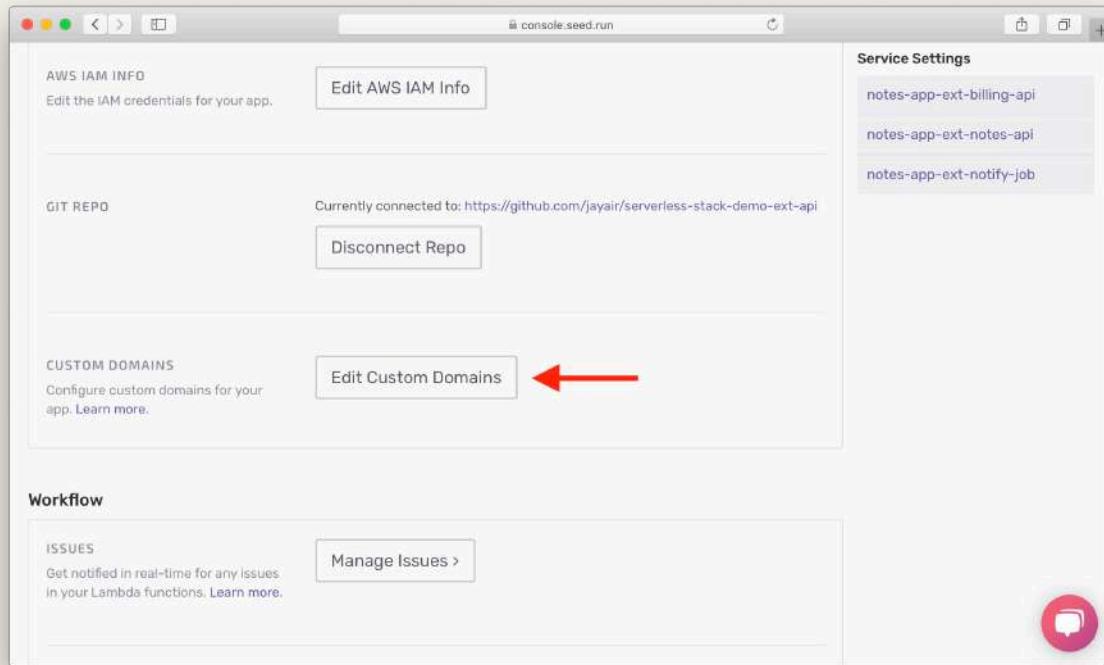
Go to the API app, and head into app settings.



The screenshot shows the SEED console interface for a project named 'serverless-stack-demo-ext-api'. The 'Overview' tab is active. At the top right, there is a 'Settings' tab with a red arrow pointing to it. The main area displays 'Key Metrics' (P95 LATENCY: NO DATA), a 'Pipeline' section showing a flow from 'DEVELOPMENT' (dev) to 'PRODUCTION' (prod) with two deployments ('v1 1:32 PM' and 'v1 1:39 PM'), and an 'Activity' section listing recent events like 'Promote dev to prod' and 'Manual deploy'. A search bar and a 'New Issues' section are also visible.

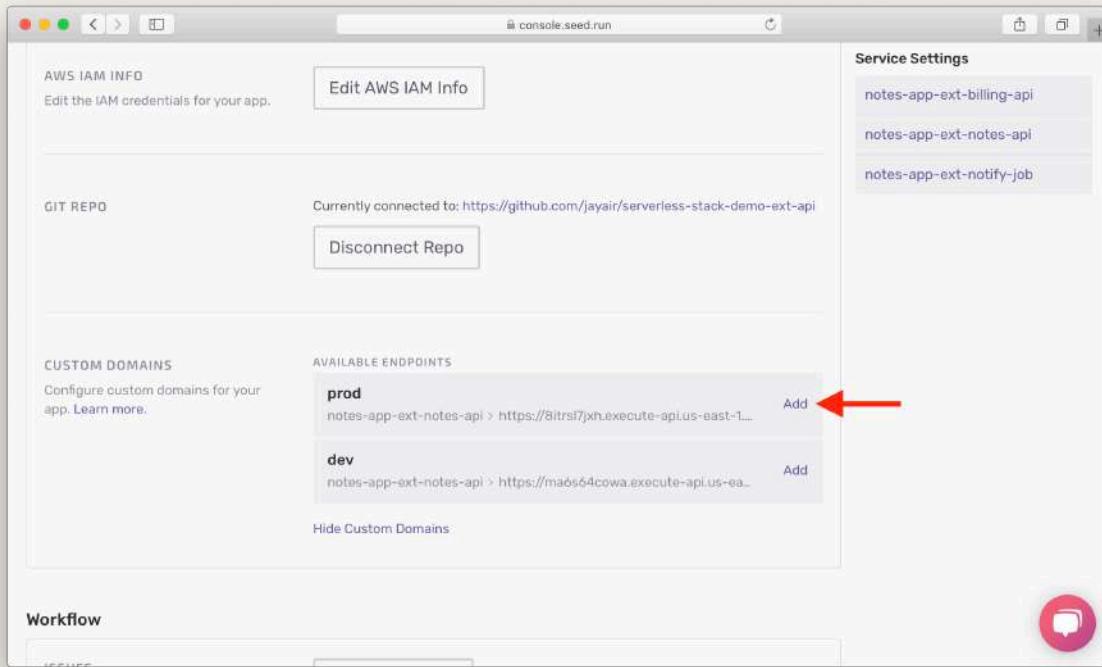
Select app settings in Seed

Select **Edit Custom Domains**.



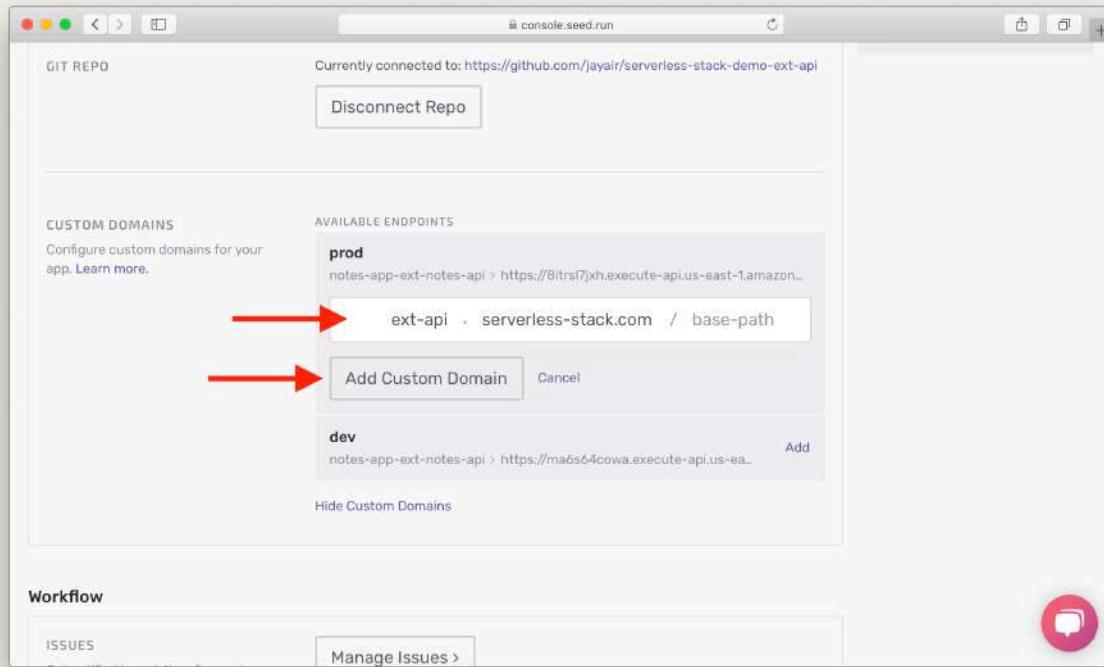
Select Edit Custom Domains

Both **dev** and **prod** endpoints are listed. Select **Add** on the **prod** endpoint.



Select Add domain for prod stage

Select the domain **serverless-stack.com** and enter the subdomain **ext-api**. Then select **Add Custom Domain**.

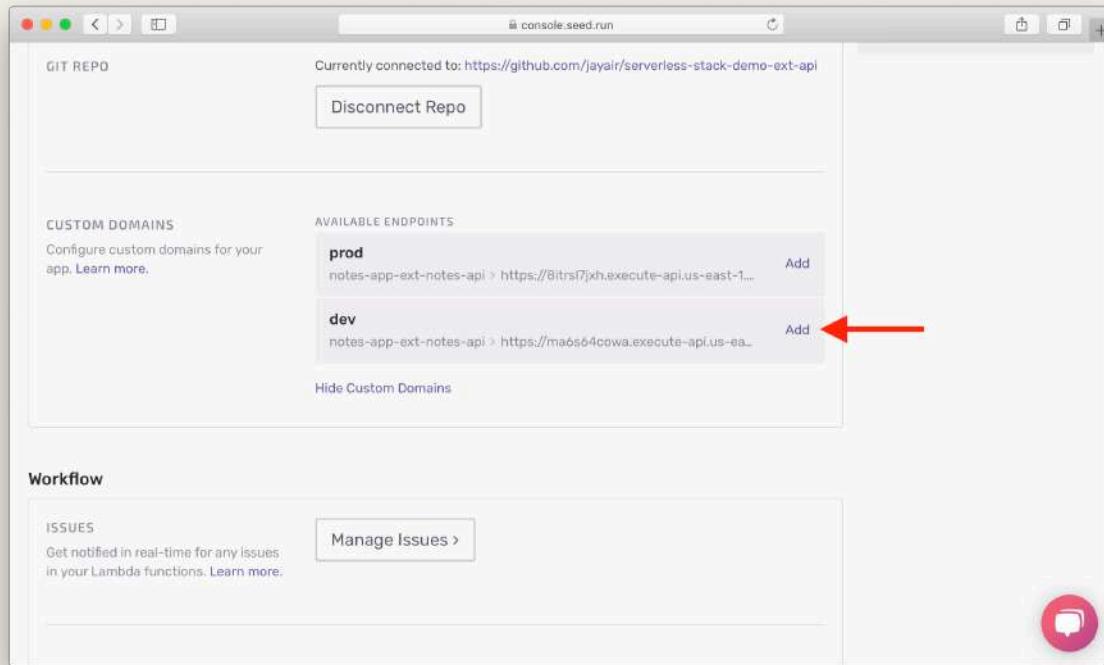


Select base domain and subdomain

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.

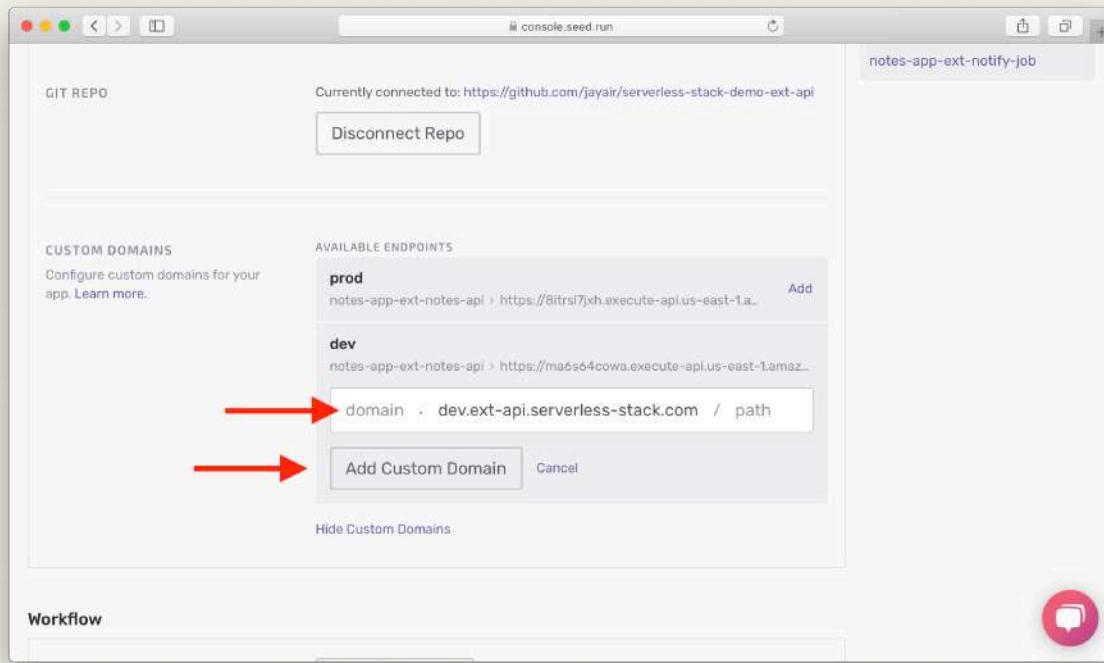
The last step is update the CloudFront distribution, which can take up to 40 minutes.

Let's setup the domain for our **dev** api. Select **Add**.



Select Add domain for dev stage

Select the domain **dev.ext-api.serverless-stack.com** and leave the subdomain empty. Then select **Add Custom Domain**.



Select base domain for dev stage

Similarly, you might have to wait for up to 40 minutes.

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](#).

Next, let's quickly look at how you'll be managing the cost and usage for your two AWS accounts.



Help and discussion

View the [comments](#) for this chapter on our forums

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.

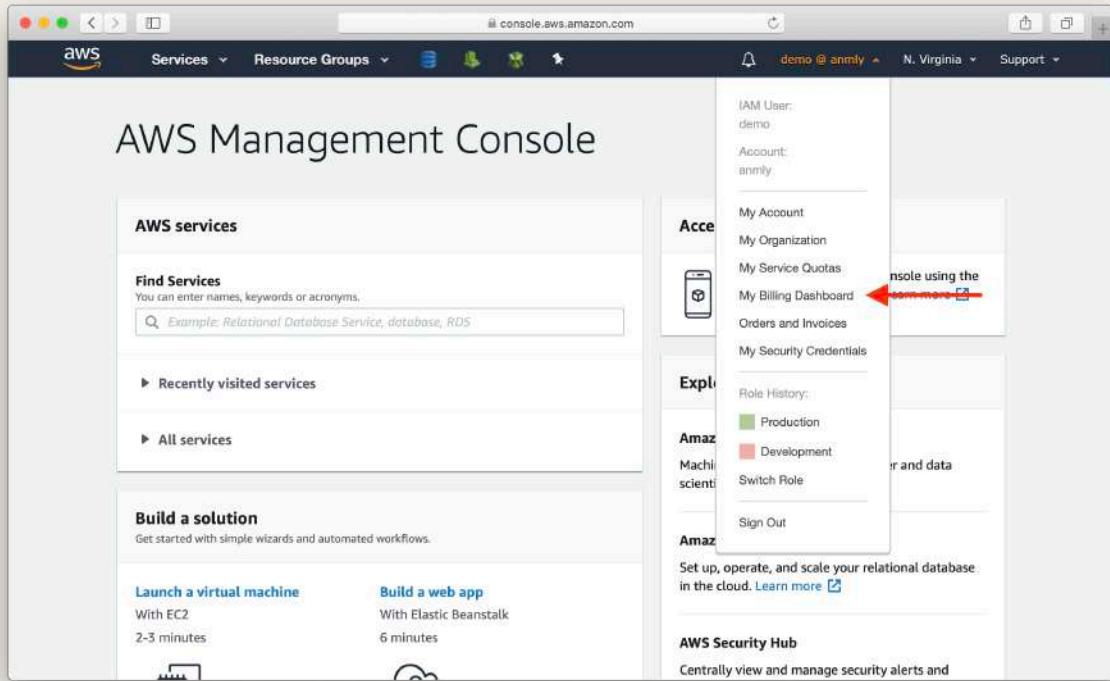
Free Tier

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**.

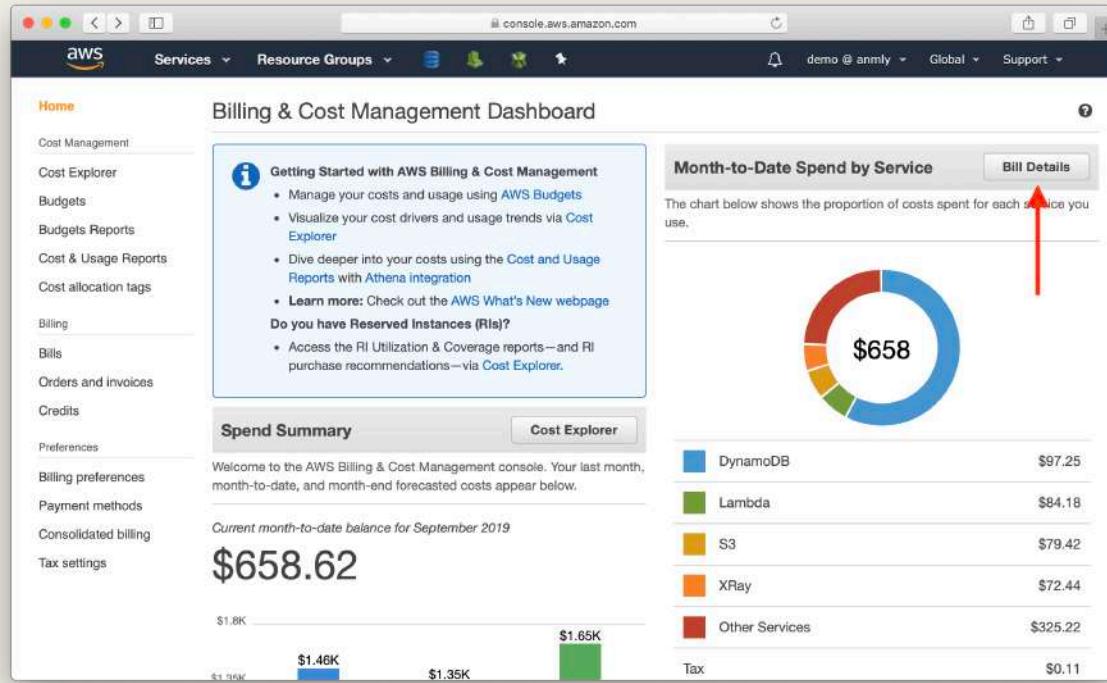


Select My Billing Dashboard

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**.



Select Bill Details screenshot

And click **Bill details by account**. Here you can see the cost allocation for each account.

The screenshot shows the AWS Bills page for September 2019. The sidebar on the left has 'Bills' selected. The main area displays the 'Estimated Total' bill of \$658.62. A red arrow points from the text 'Select Bill details by account screenshot' to the 'Bill details by account' button, which is highlighted with a yellow border. Below this, the 'Details By Account' section lists three environments: Demo (\$0.00), Development (\$1.35), and Production (\$633.27). A small note at the bottom explains that usage and recurring charges for the statement period will be charged on your next billing date.

	Total
▶ Demo	\$0.00
▶ Development	\$1.35
▶ Production	\$633.27

Select Bill details by account screenshot

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!



Help and discussion

View the [comments for this chapter on our forums](#)

Development lifecycle

Working on Serverless Apps

So to quickly recap, we've split our real world Serverless app into two repos, [one creates our infrastructure resources](#) and the [second creates our API services](#).

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](#)
- [Invoking API Gateway endpoints locally](#)
- [Creating and working on feature environments](#)
- [Creating a pull request environment](#)
- [Promoting dev to production](#)
- [Rolling back](#)

Let's start with how you work locally on your Lambda functions.



Help and discussion

View the [comments for this chapter](#) on our forums

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 handler from "../../libs/handler-lib";  
import dynamoDb from "../../libs/dynamodb-lib";  
  
export const main = handler(async (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
  }
};

const result = await dynamoDb.get(params);
if (!result.Item) {
  throw new Error("Item not found.");
}

// Return the retrieved item
return result.Item;
});
```

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. Serverless Framework sets the `IS_LOCAL` environment variable when it is run locally.

So in your code, you can check this 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.

**Help and discussion**

View the [comments for this chapter on our forums](#)

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](#) 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 included as `startServer` in the [sample repo](#). 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.



Help and discussion

View the [comments for this chapter on our forums](#)

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`, there are plugins like `serverless-dynamodb-local` and `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: nodejs12.x
  stage: dev
  region: us-east-1
  tracing:
    lambda: true

apiGateway:
  restApiId: !ImportValue ${self:custom.stage}-ExtApiGatewayRestApiId
  restApiRootResourceId: !ImportValue
  ${self:custom.stage}-ExtApiGatewayRestApiRootResourceId
  restApiResources:
    /notes/{id}: !ImportValue
  ${self:custom.stage}-ExtApiGatewayResourceNotesIdVarId

environment:
  stage: ${self:custom.stage}

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.

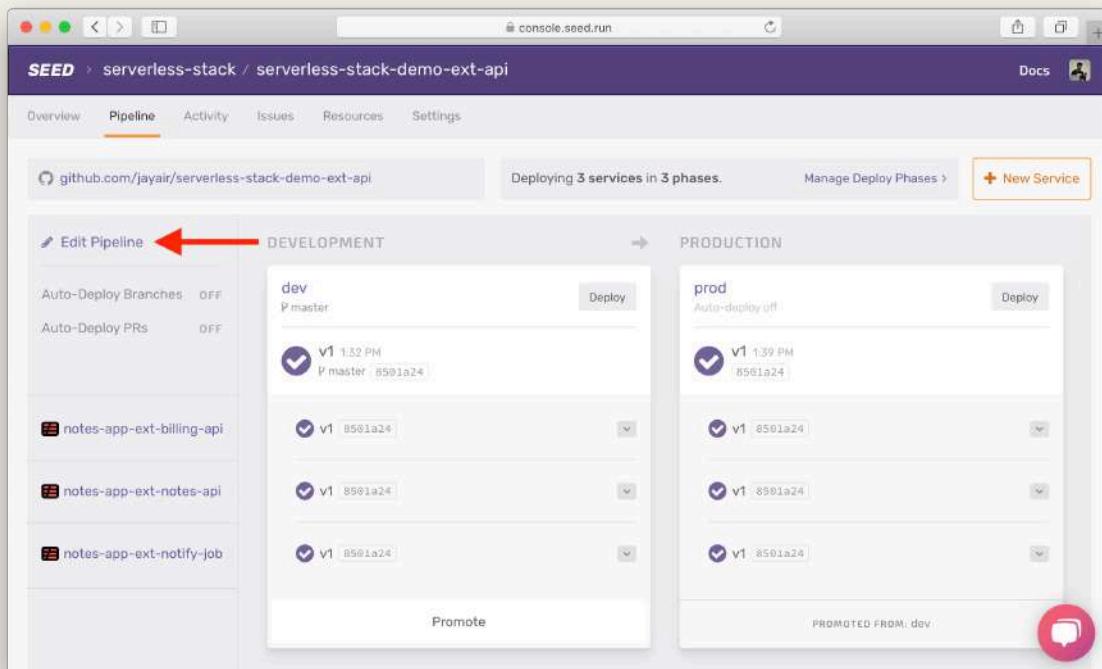
Add the handler file `like.js`.

```
import { success } from "../../libs/response-lib";  
  
export async function main(event, context) {  
    // Business logic code for liking a post  
  
    return success({ status: true });  
}
```

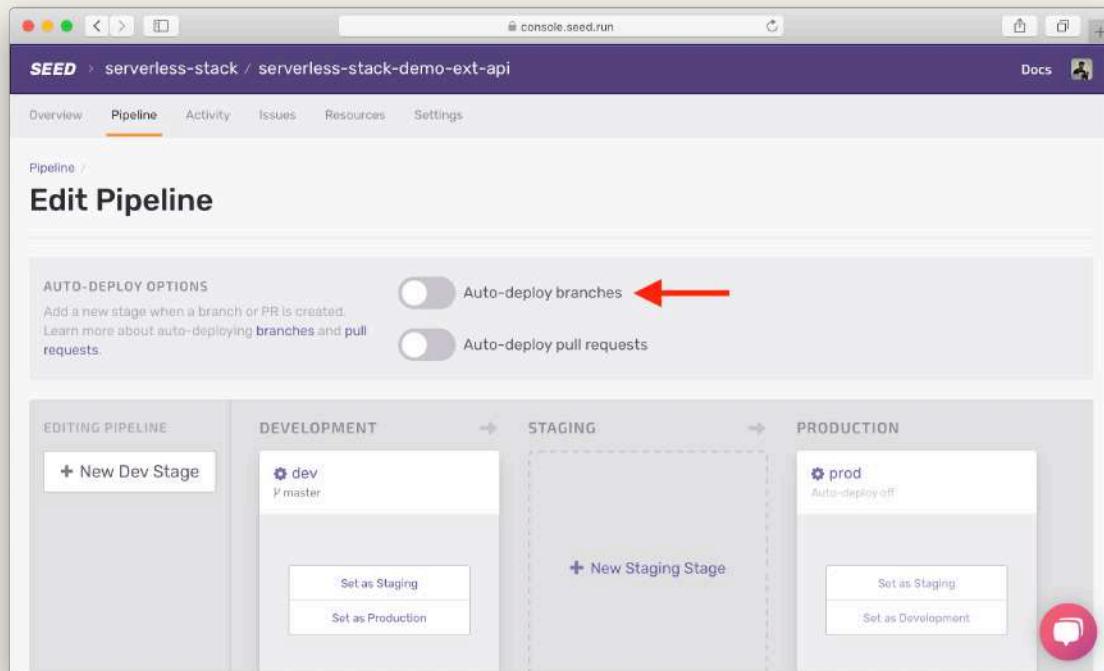
Now before we push our Git branch, let's enable the branch workflow in Seed.

Enable branch workflow in Seed

Go to your app on Seed and head over to the **Pipeline** tab and hit **Edit Pipeline**.

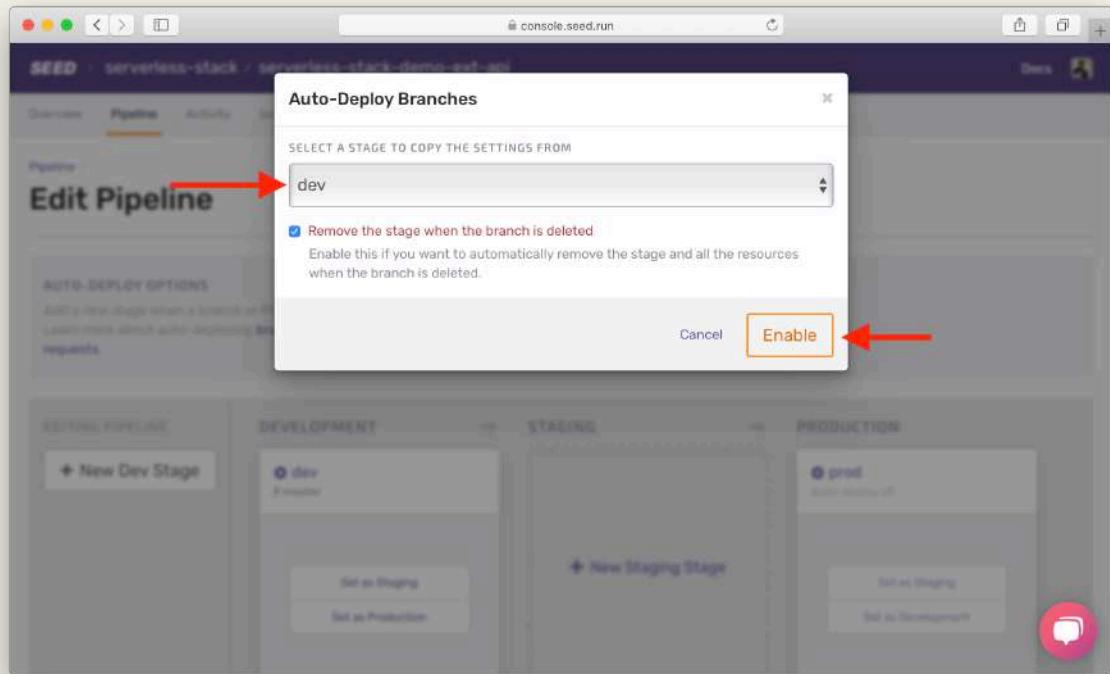


Enable **Auto-deploy branches**.



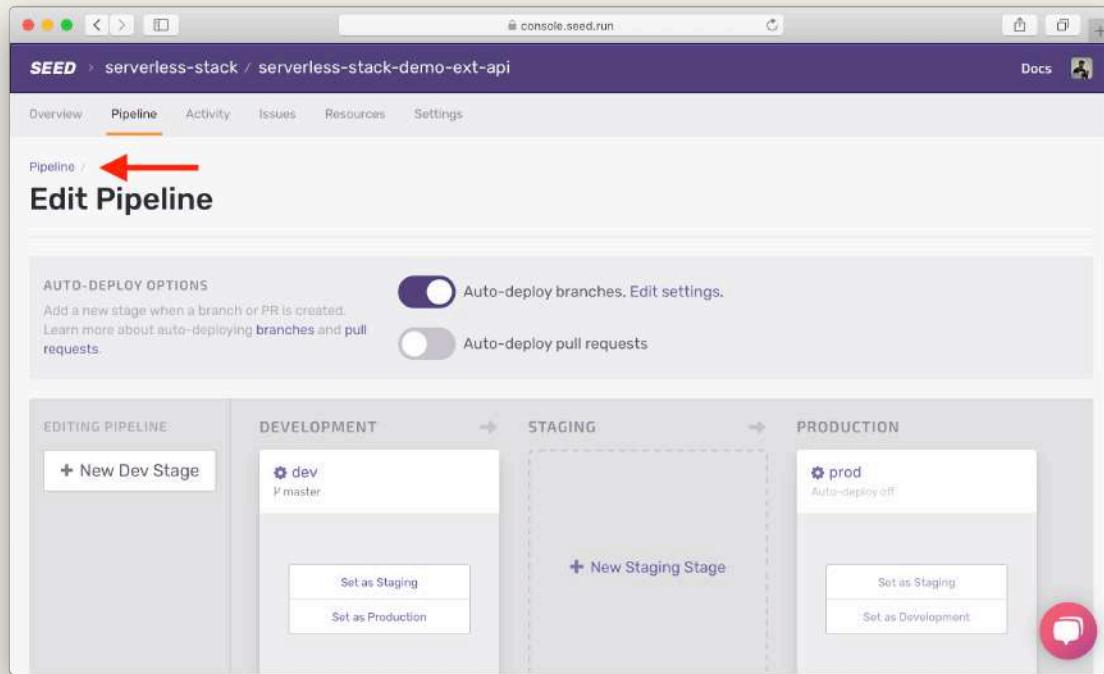
Select Enable Auto-Deploy Branches

Select the **dev** stage, since we want the stage to be deployed into the **Development AWS account**. Click **Enable**.



Select Enable Auto-Deploy

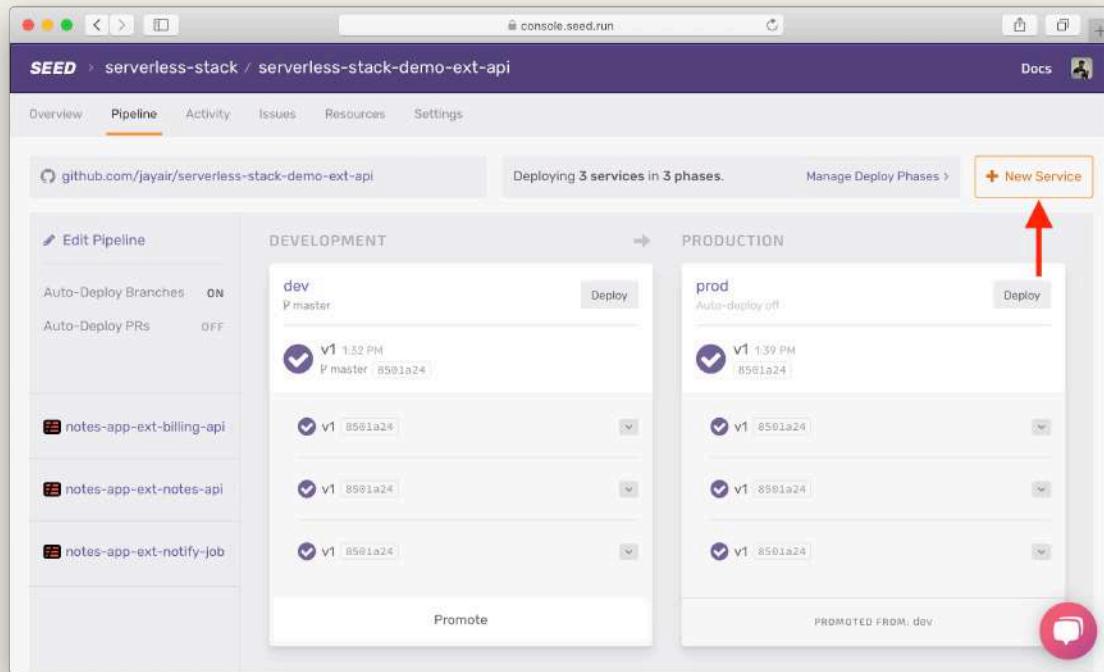
Click **Pipeline** to head back.



Head back to pipeline

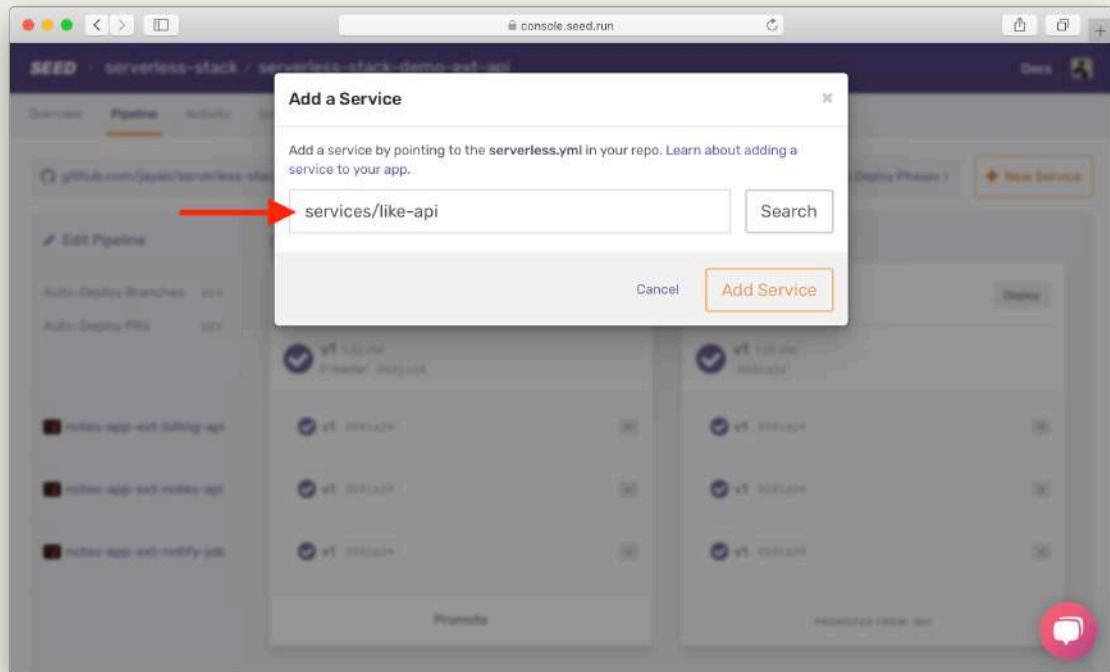
Add the new service to Seed

Click on **Add a Service**.



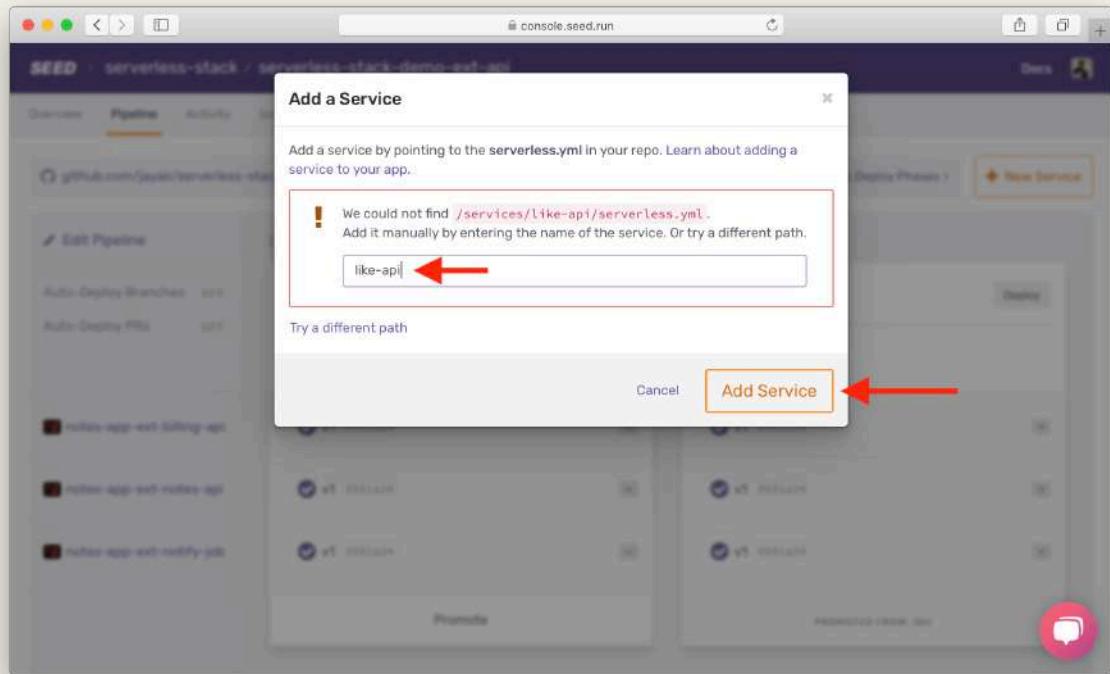
Select Add a service

Enter the path to the service `services/like-api` and click **Search**.



Select search new service path

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**.



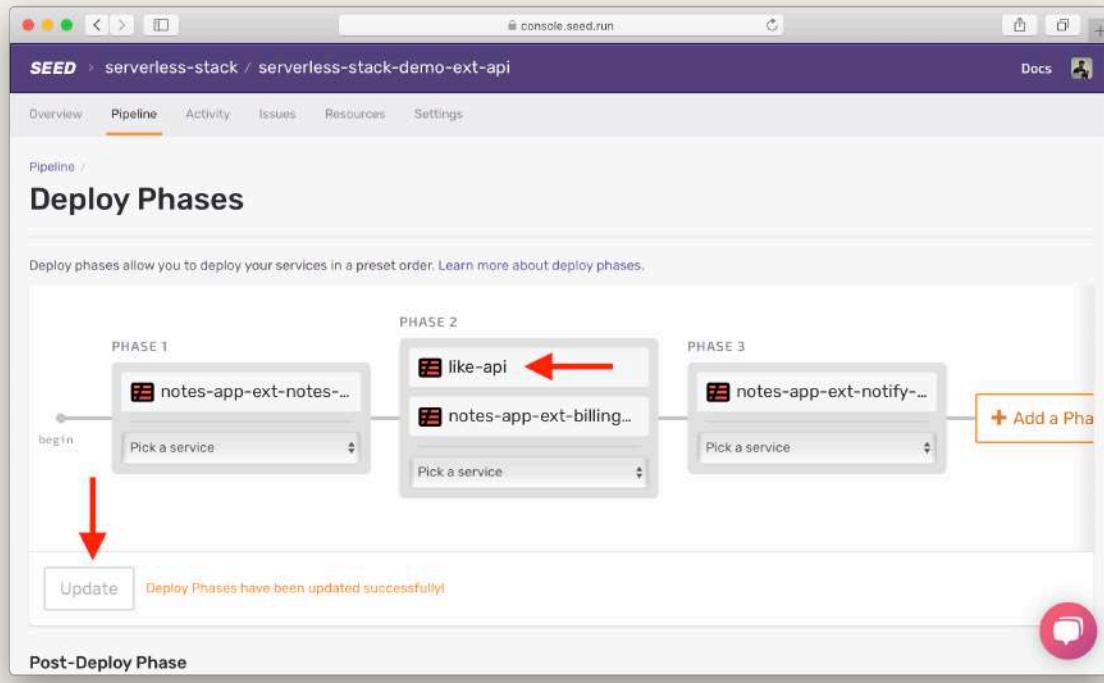
Set new service name

This should add the new service across all your stages.

The screenshot shows the SEED console interface for a project named 'serverless-stack-demo-ext-api'. The pipeline is currently set to 'OFF'. It displays two stages: 'DEVELOPMENT' and 'PRODUCTION'. In the 'DEVELOPMENT' stage, there are four services: 'like-api', 'notes-app-ext-billing-api', 'notes-app-ext-notes-api', and 'notes-app-ext-notify-job'. Each service has a deployment history with multiple versions (v1) and their respective commit IDs. In the 'PRODUCTION' stage, there is one service named 'prod' which also has a deployment history with multiple versions (v1) and their respective commit IDs. A red speech bubble icon is visible in the bottom right corner of the production section.

Added new service in Seed

By default, the new service is added to the last deploy phase. Let's click on **Manage Deploy Phases**, and move it to Phase 2. This is because it's dependent on the API Gateway resources exported by notes-api.



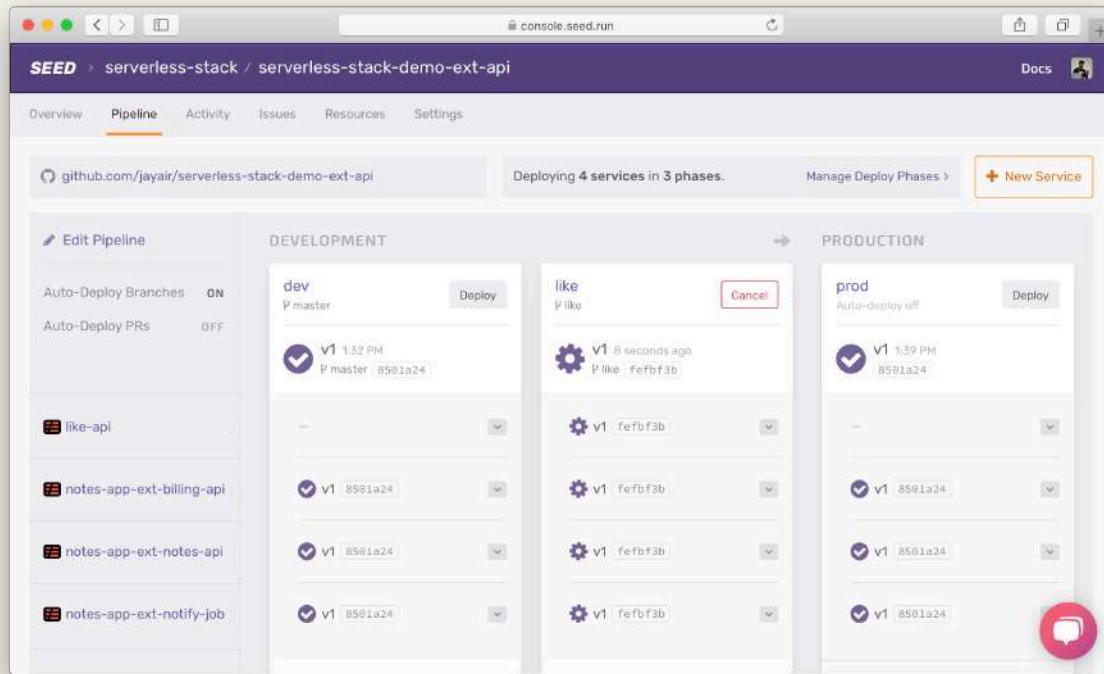
Show default Deploy Phase

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.

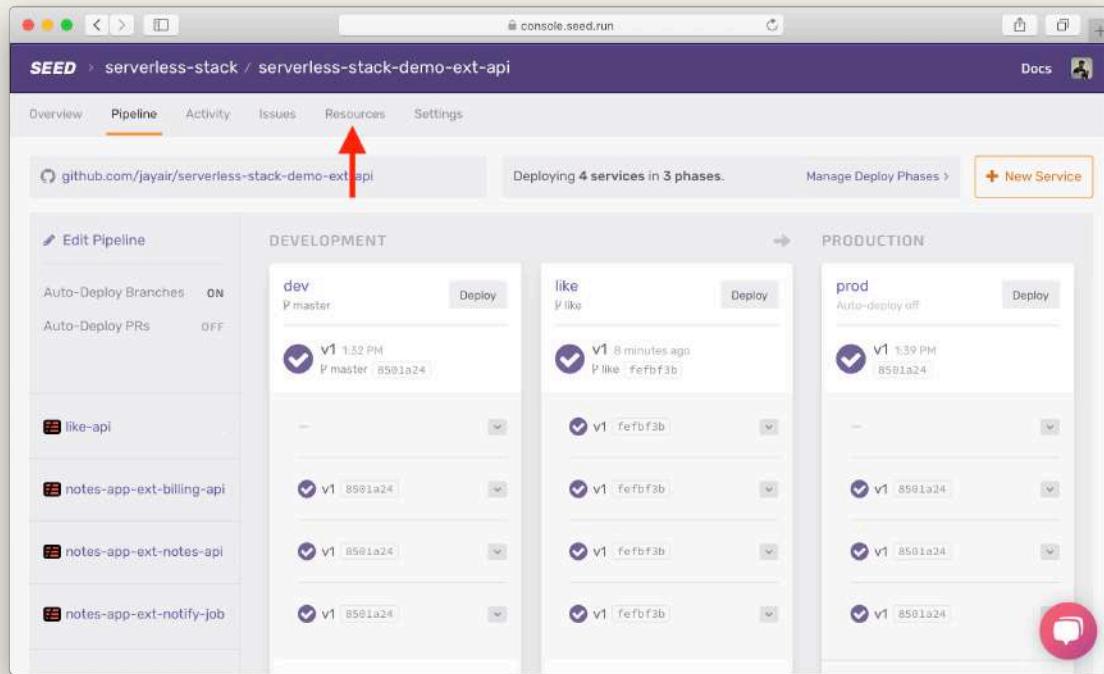
```
$ git add .
$ git commit -m "Add like API"
$ git push --set-upstream origin like
```

Back in Seed, a new stage called `like` is created and is being deployed automatically.



Show new feature stage created

After the new stage successfully deploys, you can get the API endpoint in the stage's resources page. Head over to the **Resources** tab.



The screenshot shows the SEED (Serverless Environment Deployment) interface. At the top, there's a navigation bar with tabs: Overview, Pipeline (which is highlighted in orange), Activity, Issues, and Resources. Below the navigation bar, the URL is shown as `github.com/jayair/serverless-stack-demo-ext-api`. To the right of the URL, it says "Deploying 4 services in 3 phases." and there's a "Manage Deploy Phases" button. Further to the right is a "New Service" button. The main area is divided into three columns: DEVELOPMENT, like, and PRODUCTION. The DEVELOPMENT column contains four services: like-api, notes-app-ext-billing-api, notes-app-ext-notes-api, and notes-app-ext-notify-job. The like column contains two services: like and notes-app-ext-notify-job. The PRODUCTION column contains one service: notes-app-ext-notify-job. Each service has a "Deploy" button. A red arrow points from the text above to the "Resources" tab in the navigation bar.

Select Resources tab in Seed

And select the **like** stage.

The screenshot shows the SEED console interface for the 'notes-app-ext-notes-api' service. On the left, there's a sidebar with stages: 'prod' (selected), 'dev', and 'like'. A red arrow points to the 'like' stage. The main area displays API endpoints for the 'notes' resource across six stages. The 'like' stage is highlighted in blue. The table below lists the API paths and their corresponding Lambda functions:

Stage	Method	Path	Function	Logs	Metrics
prod	POST	/billing	notes-app-ext-billing-api-prod-billing	Logs	Metrics
prod	GET	/notes	notes-app-ext-notes-api-prod-list	Logs	Metrics
prod	POST	/notes	notes-app-ext-notes-api-prod-create	Logs	Metrics
prod	PUT	/notes/{id}	notes-app-ext-notes-api-prod-get	Logs	Metrics
prod	DELETE	/notes/{id}	notes-app-ext-notes-api-prod-update	Logs	Metrics
prod			notes-app-ext-notes-api-prod-delete	Logs	Metrics

Select feature stage

You will see the API Gateway endpoint for the **like** stage and the API path for the **like** handler.

The screenshot shows the SEED console interface for the 'serverless-stack-demo-ext-api' service. The 'like' stage is selected. The 'LAMBDAS' section lists several endpoints:

Method	Path	Handler	Logs	Metrics
POST	/billing	notes-app-ext-billing-api-like-billing	Logs	Metrics
GET	/notes	notes-app-ext-notes-api-like-list	Logs	Metrics
POST	/notes	notes-app-ext-notes-api-like-create	Logs	Metrics
GET	/notes/{id}	notes-app-ext-notes-api-like-get	Logs	Metrics
PUT	/notes/{id}	notes-app-ext-notes-api-like-update	Logs	Metrics
DELETE	/notes/{id}	notes-app-ext-notes-api-like-delete	Logs	Metrics
POST	/notes/{id}/like	notes-app-ext-like-api-like-like	Logs	Metrics

Show API Gateway endpoint in feature 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.



Help and discussion

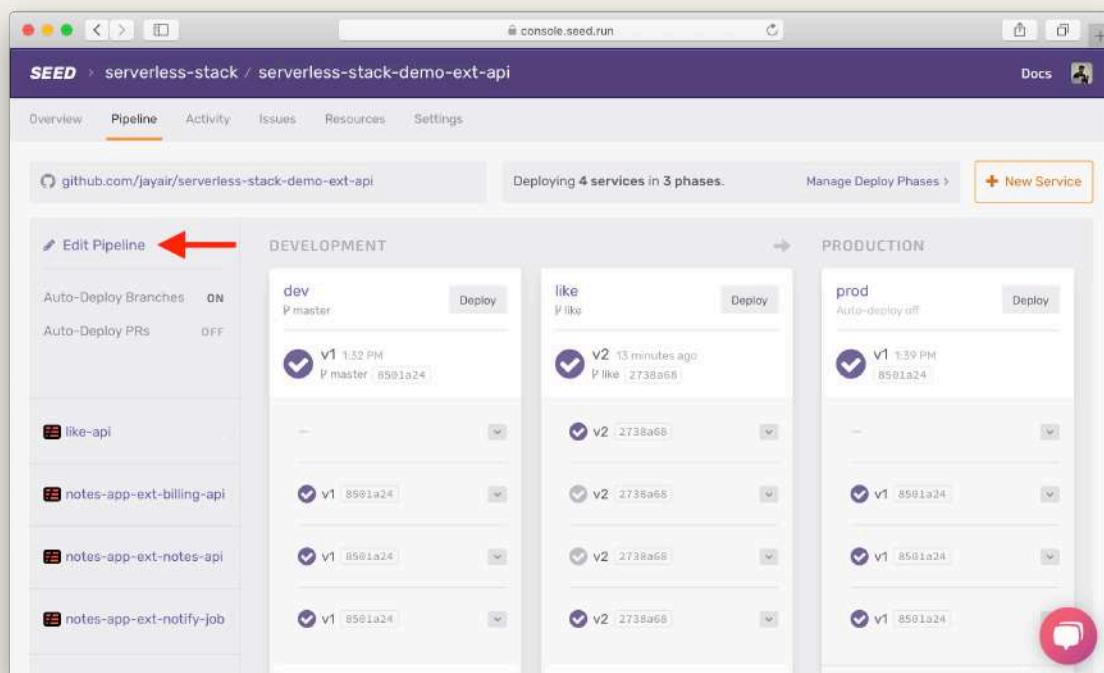
View the [comments for this chapter on our forums](#)

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.

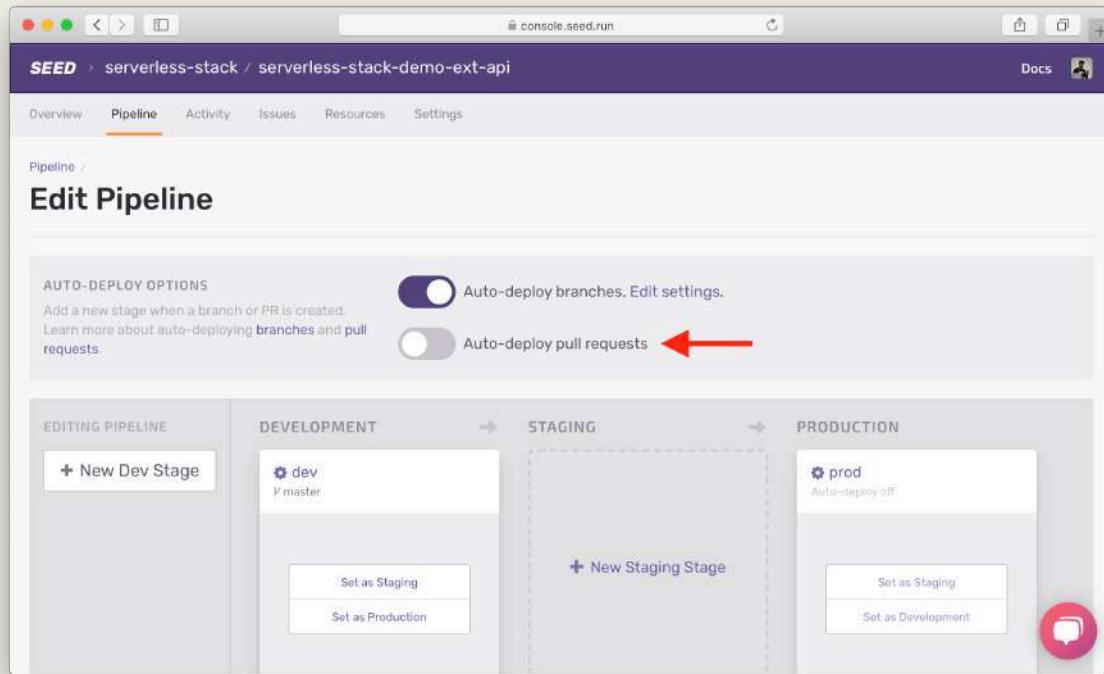
Enable pull request workflow on Seed

To enable auto-deploying pull requests, head over to your app on Seed. Click **Settings**.



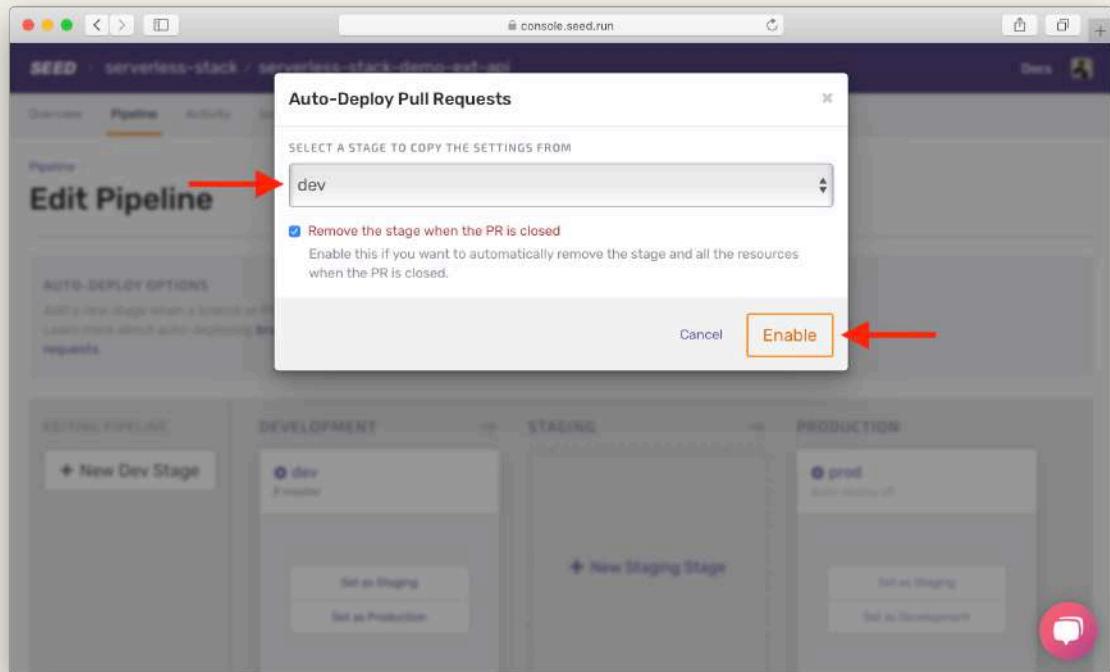
Select edit pipeline in Seed

And **Enable auto-deploy pull requests**.



Select Enable Auto-Deploy PRs

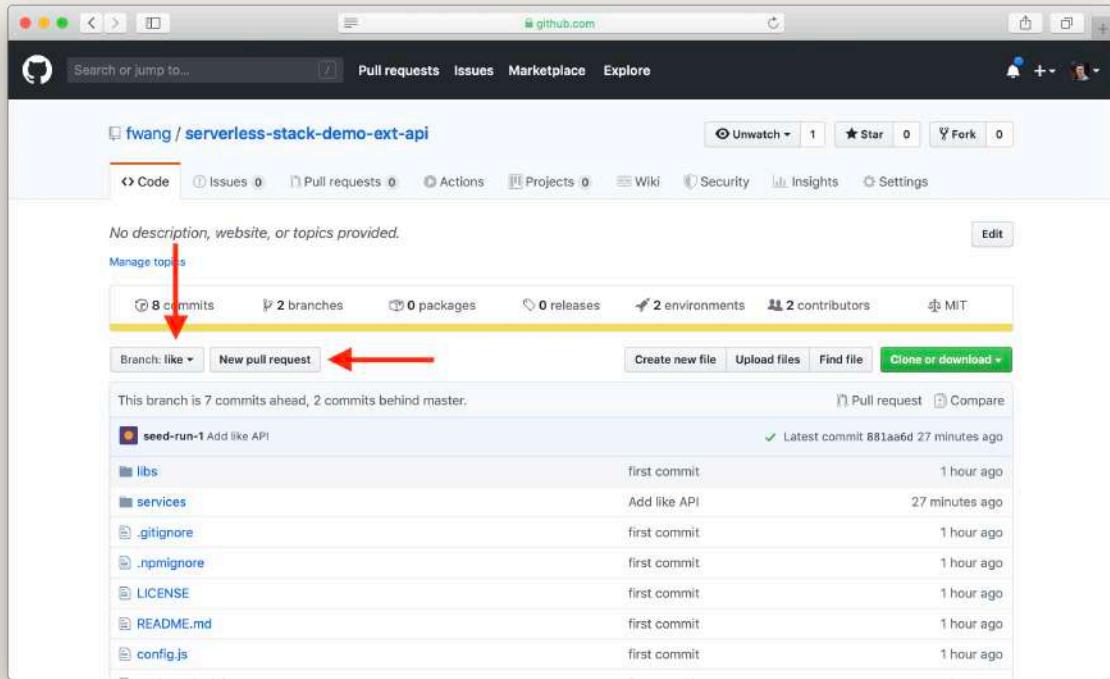
Select the **dev** stage, since we want the stage to be deployed into the **Development AWS account**. Click **Enable**.



Select Enable Auto-Deploy

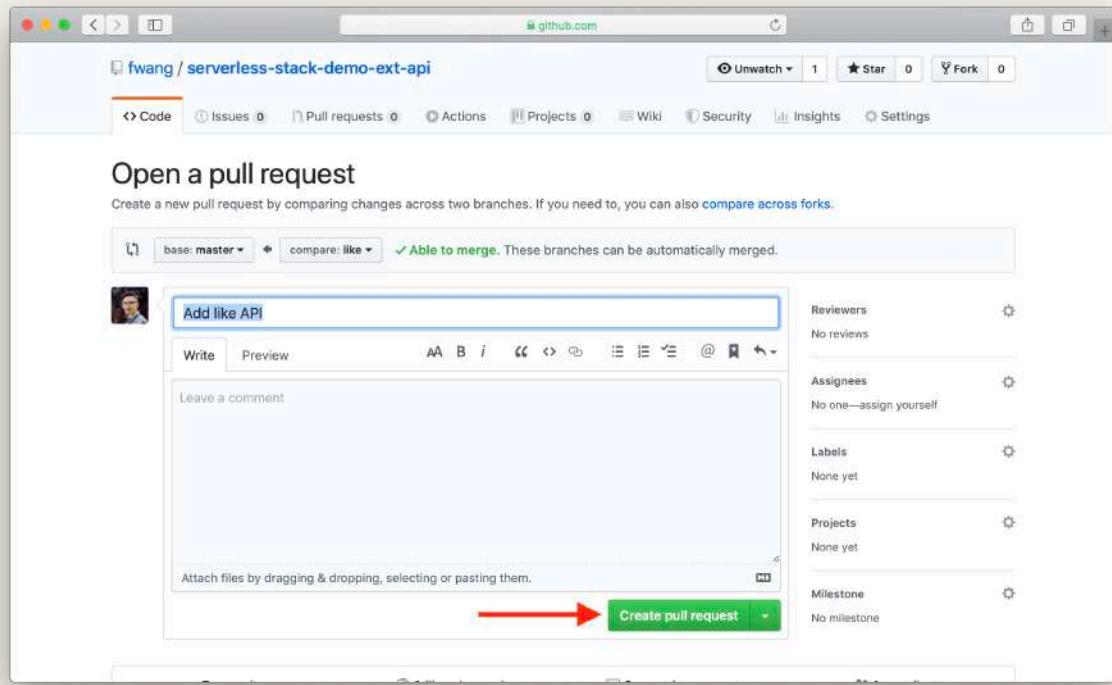
Create a pull request

Go to GitHub, and select the **like** branch. Then hit **New pull request**.



Select New pull requests in GitHub

Click **Create pull request**.



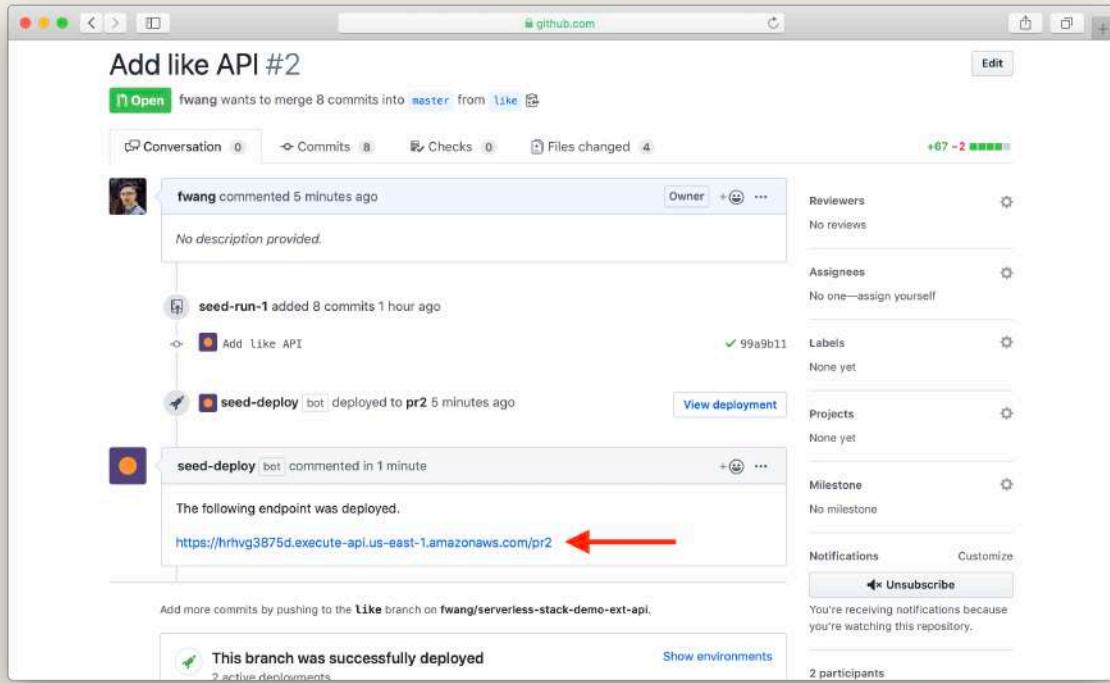
Select Create pull request in GitHub

Now back in Seed, a new stage (in this case **pr2**) should be created and is being deployed automatically.

The screenshot shows the SEED console interface for managing a serverless application. The pipeline is named 'serverless-stack-demo-ext-api'. It includes four services: like-api, notes-app-ext-billing-api, notes-app-ext-notes-api, and notes-app-ext-notify-job. The pipeline consists of three stages: 'dev', 'like', and 'pr2'. The 'pr2' stage is currently active, indicated by a red arrow pointing to its name. Within the 'like' stage, there are two versions: v1 and v2. In the 'pr2' stage, there are three versions: v1, v2, and v3. A pink speech bubble icon is located in the bottom right corner of the interface.

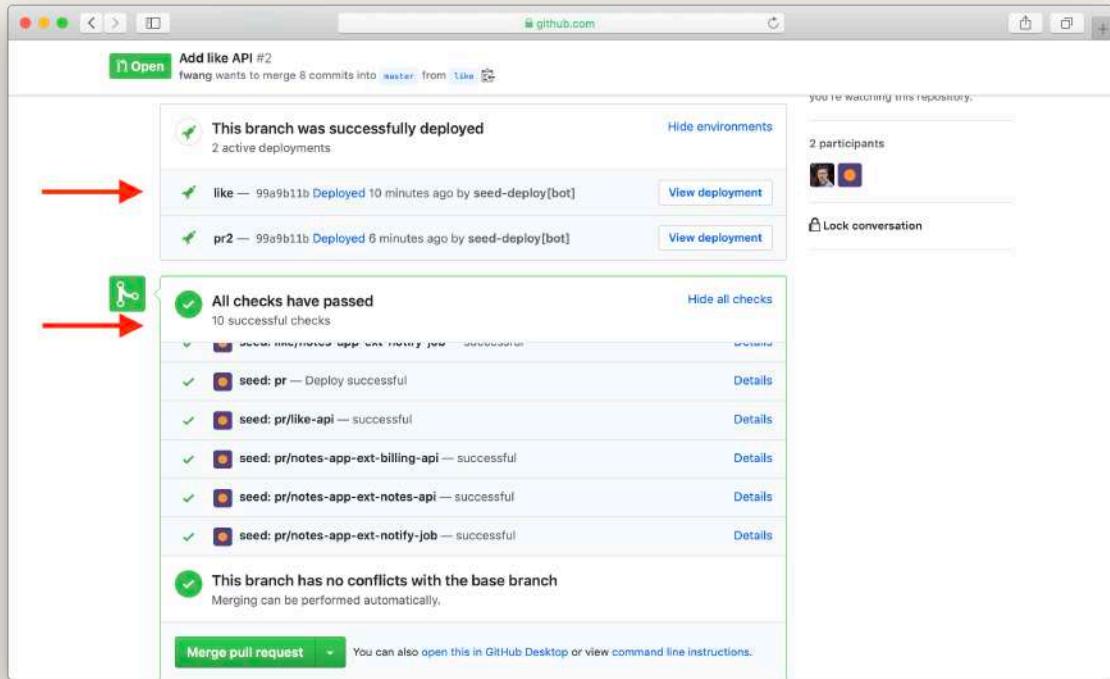
Show pull request stage created

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.



Show API endpoint in GitHub PR page

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.

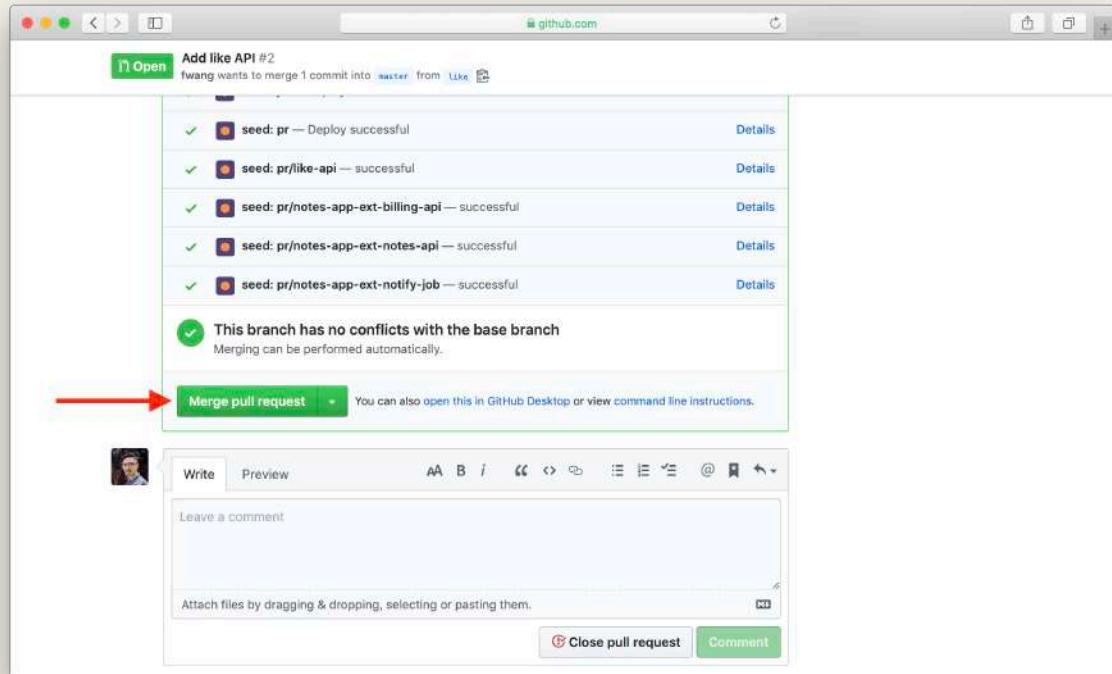


Show pull request checks in GitHub

Now that our new feature has been reviewed, we are ready to merge it to master.

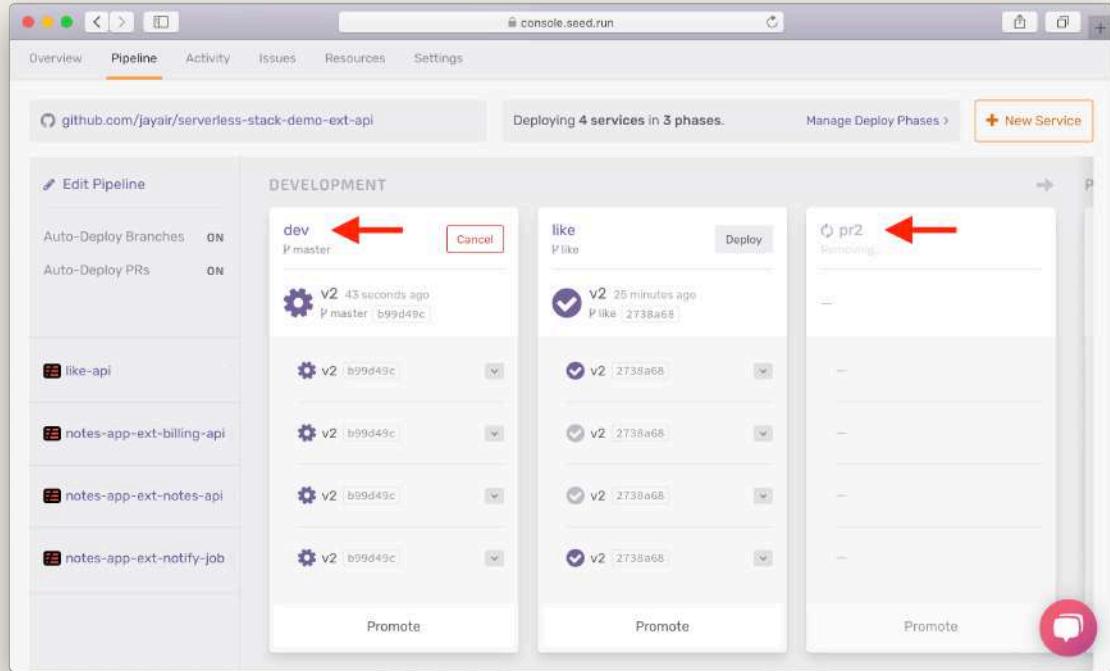
Merge to master

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**.



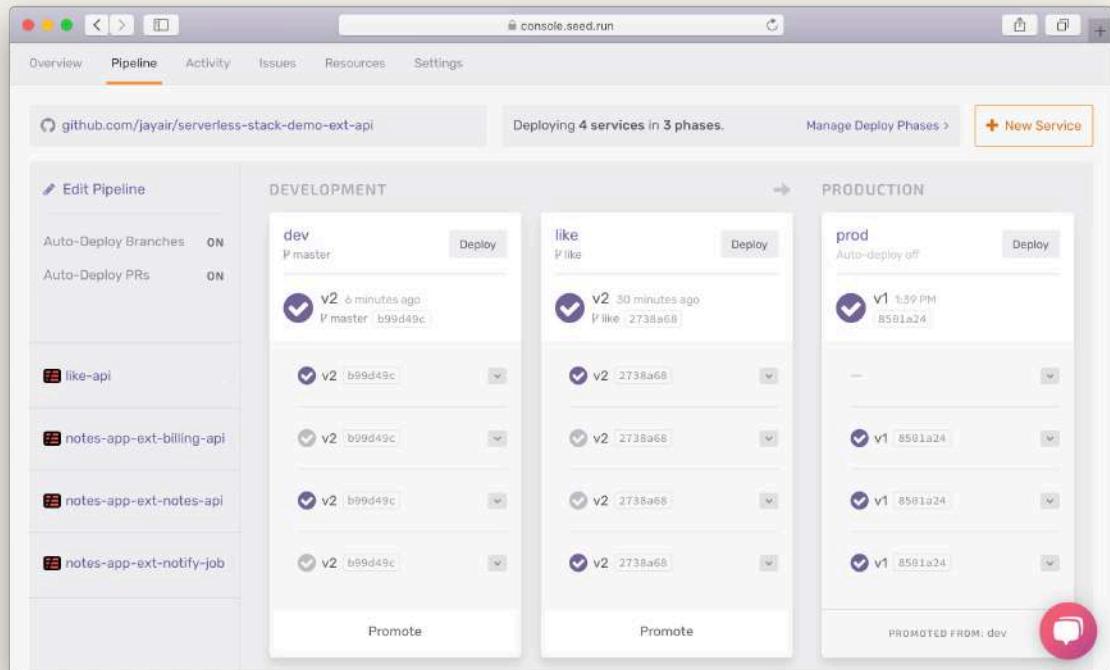
Select Merge pull request

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.



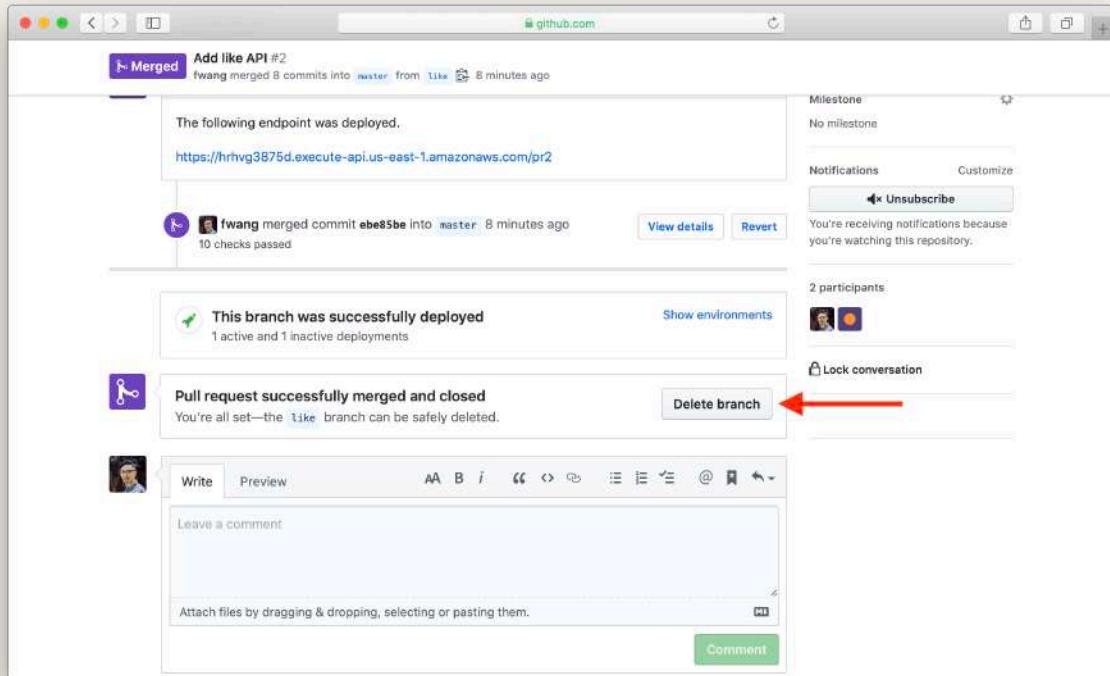
Show dev stage auto deploying

After the deployment completes and the **pr2** stage is removed, this is what your pipeline should look like:



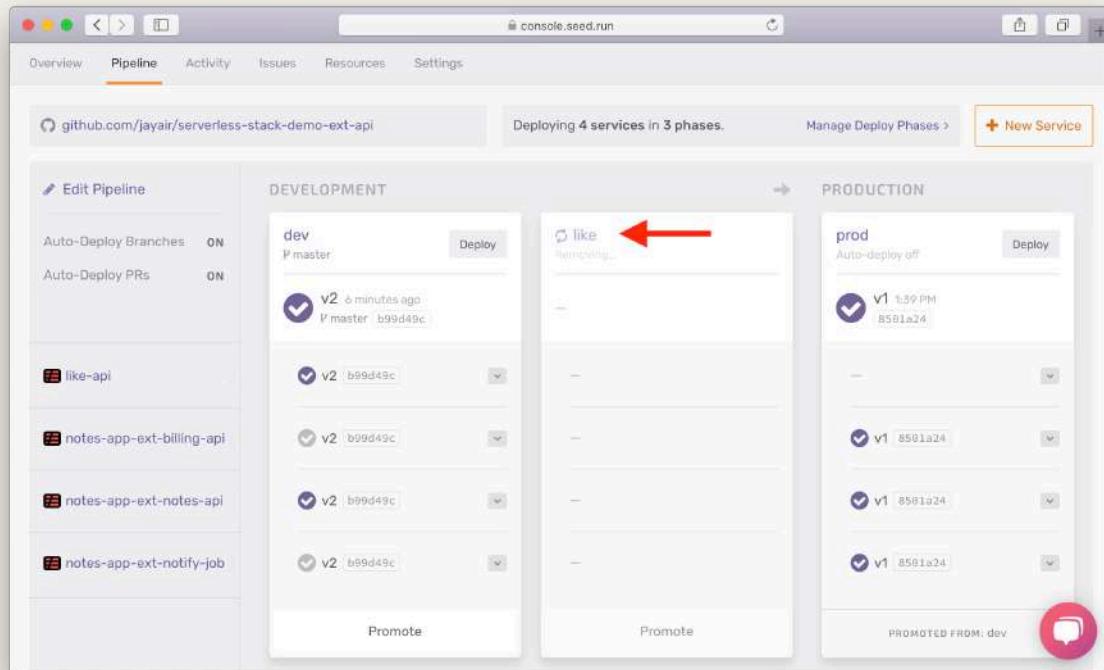
Show pull request stage removed

From GitHub's pull request screen, we can remove the **like** branch.



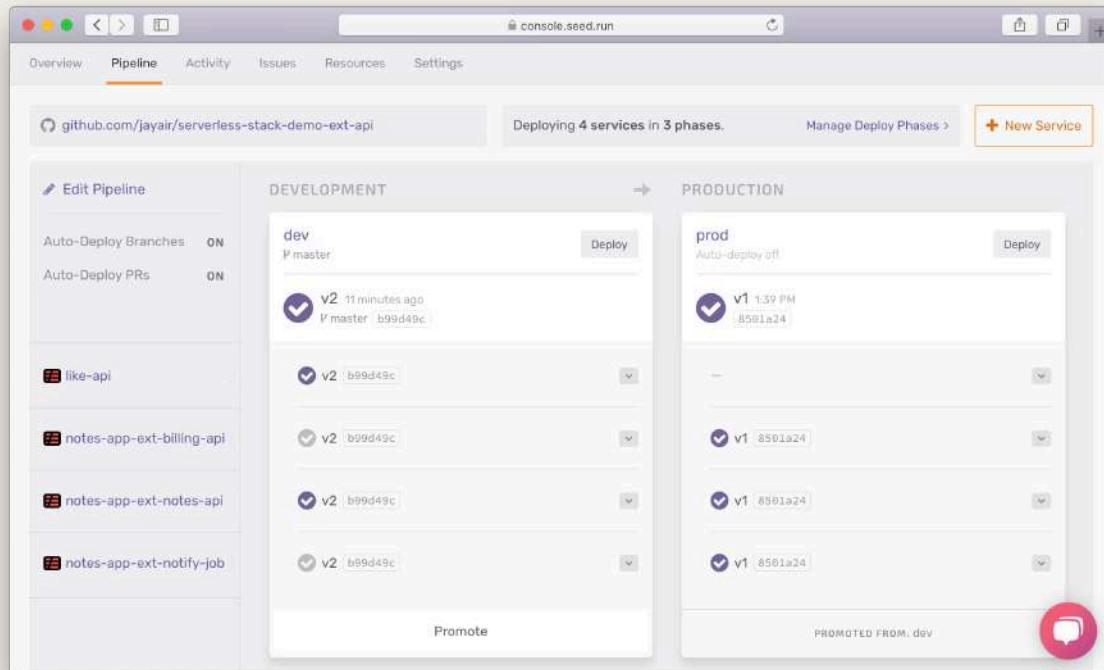
Select remove branch in GitHub

Back in Seed, this will trigger the **like** stage to be automatically removed.



Show branch stage removed

After the removal is completed, your pipeline should now look like this.



Show feature merged in dev stage

Next, we are ready to promote our new feature to production.



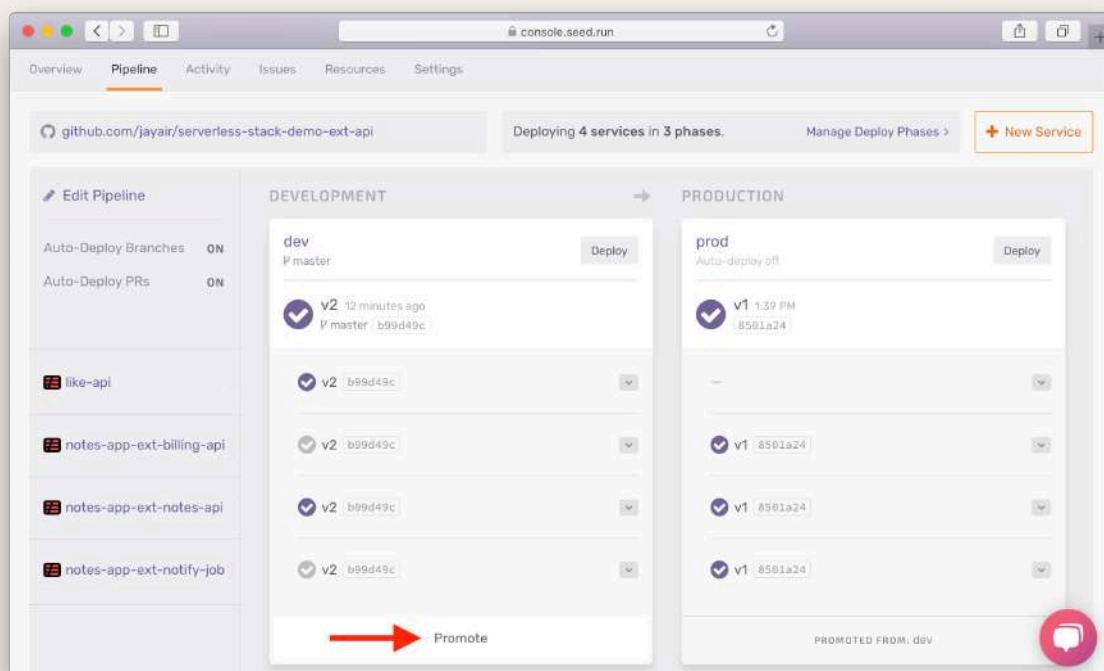
Help and discussion

View the [comments for this chapter on our forums](#)

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.

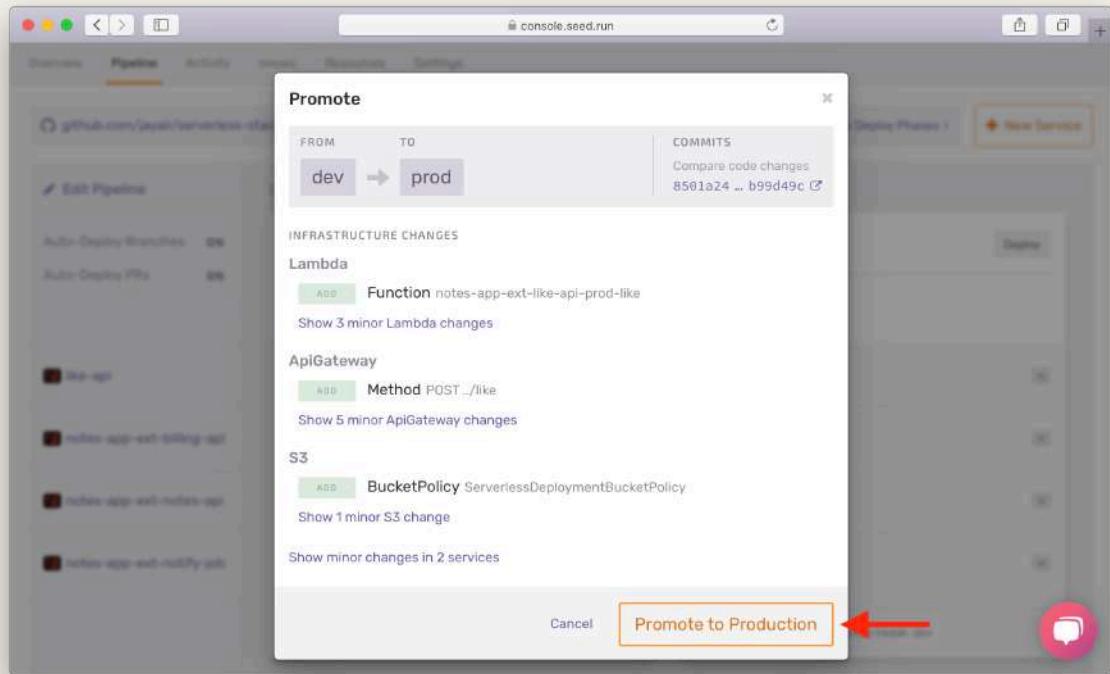
Head over to Seed. And then hit **Promote** at the bottom of the dev stage.



Select Promote in dev stage

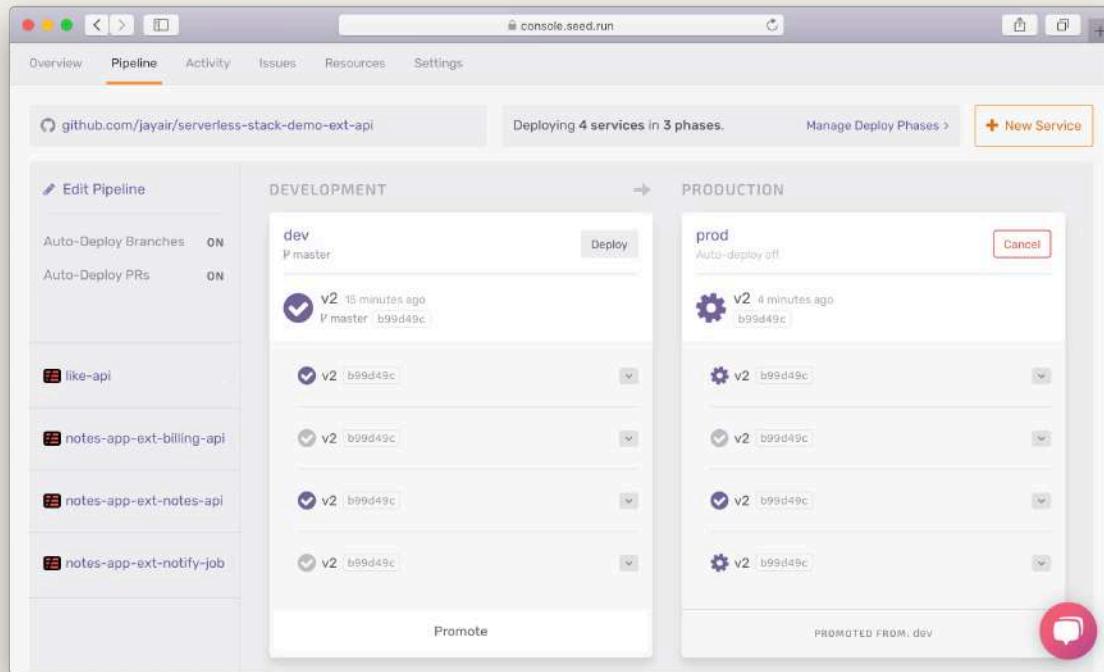
You will see a list of changes. Note, only the major changes are shown here. The change list shows that we added a Lambda functions and an API Gateway method. A couple of other minor resources like the Lambda execution IAM role and Lambda's CloudWatch log group were also added but those are hidden by default.

Hit **Promote to Production**.



Select Promote to Production

This will trigger the prod stage to start building.



Show deploying in prod stage

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](#), 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](#). 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.



Help and discussion

View the [comments for this chapter on our forums](#)

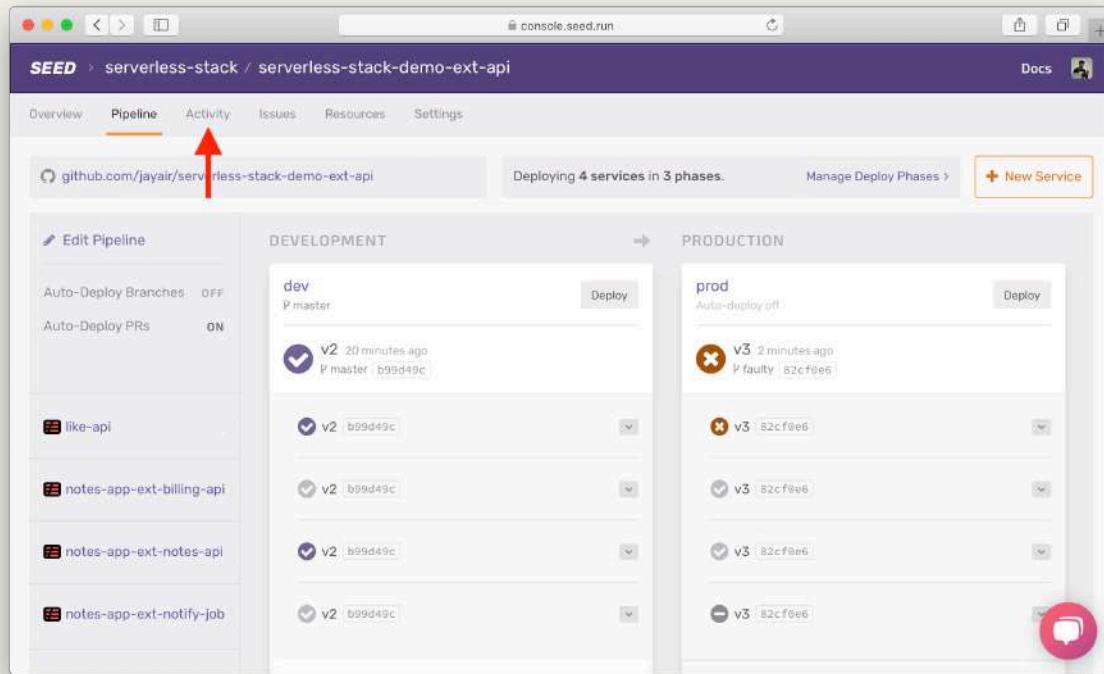
Rollback Changes

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. Head over to the **Activity** tab to see a list of historical builds.



Select prod stage in Seed

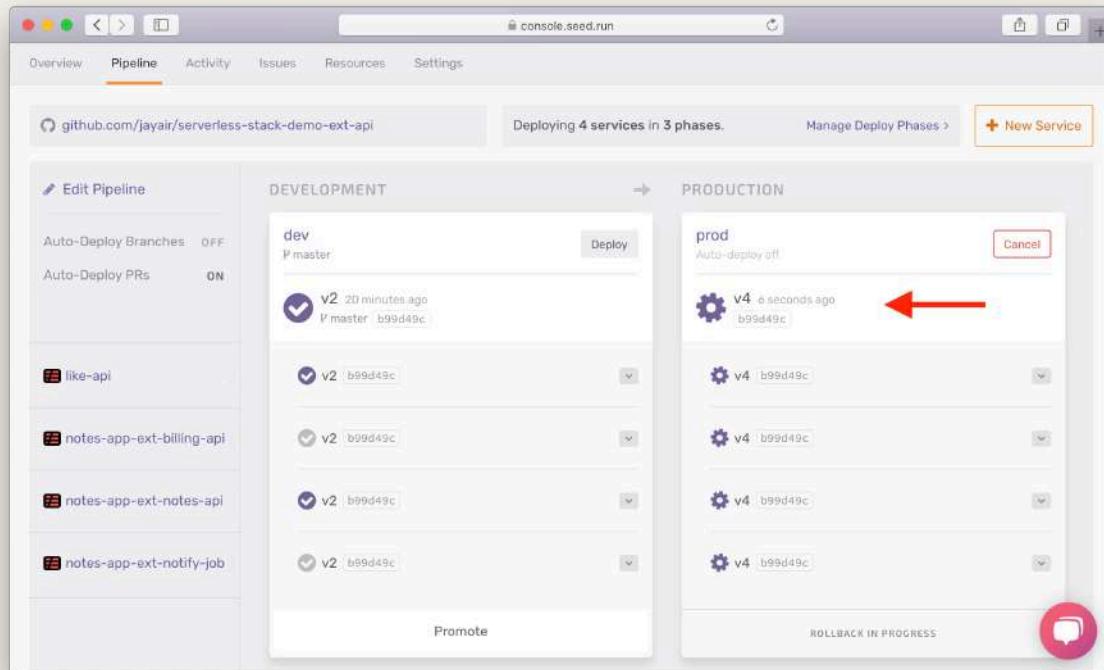
Pick an older successful build and hit **Rollback**.

The screenshot shows the SEED console interface for the 'serverless-stack-demo-ext-api' stack. The 'Activity' tab is selected. The prod stage has a failed deployment (v3 Adding faulty code) and an active rollback button. A red arrow points to the 'Rollback' button for the prod stage.

Stage	Action	Status	Time	Buttons
prod	v3 Adding faulty code	Failed	2 minutes ago	Active
prod	Promote dev to prod	In Progress	8 minutes ago	Rollback
dev	v2 Promoted all services	Completed	20 minutes ago	Active
dev	Auto-deploy to dev	Completed	20 minutes ago	Active
pr1	Auto-deploy to pr1	In Progress	8 minutes ago	Comment

Select Rollback in prod stage

Notice a new build is triggered for the prod stage.



Show rolling back in 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](#). 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: ${self:custom.stage}-ExtNotePurchasedTopicArn`

And the notify-job service imports the topic and uses it to trigger the notify function:

```
functions:  
  notify:  
    handler: notify.main  
    events:  
      - sns:  
          arn: !ImportValue ${self:custom.stage}-ExtNotePurchasedTopicArn  
          topicName: ${self:custom.stage}-note-purchased
```

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.



Help and discussion

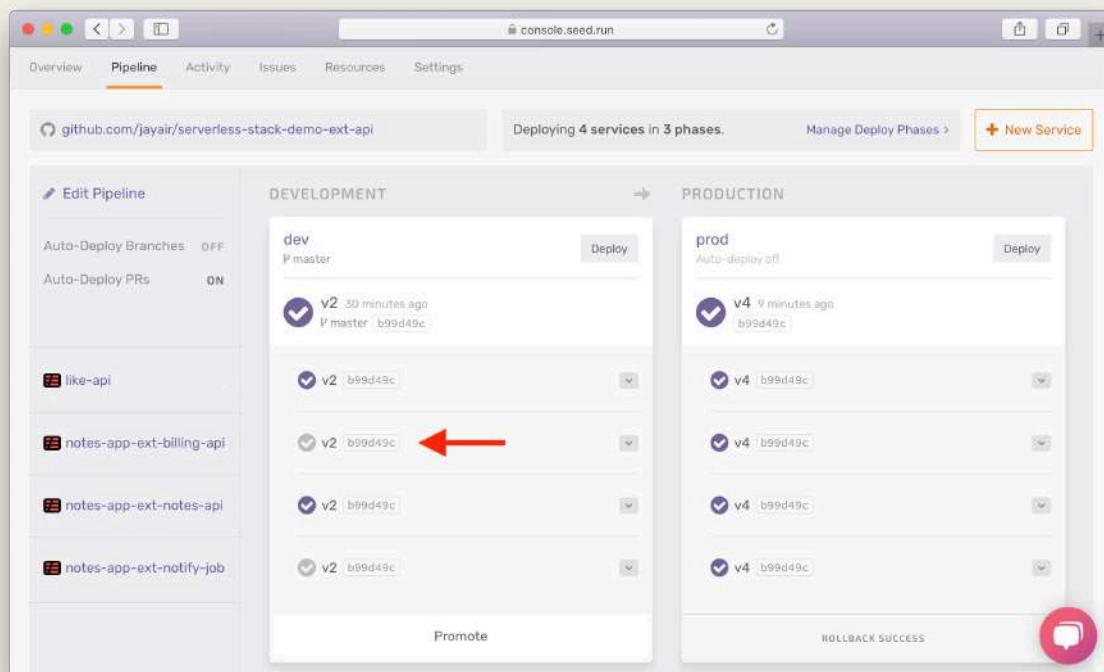
View the [comments for this chapter on our forums](#)

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.



Show deployment skipped in Seed

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](#) services. It's not uncommon for teams to have apps with over 40 services in a single repo on [Seed](#). 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](#), 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.



Help and discussion

View the [comments for this chapter on our forums](#)

Observability

Tracing Serverless Apps with X-Ray

This chapter is based on a blog post over on the [Seed blog](#) – www.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](#) 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
```

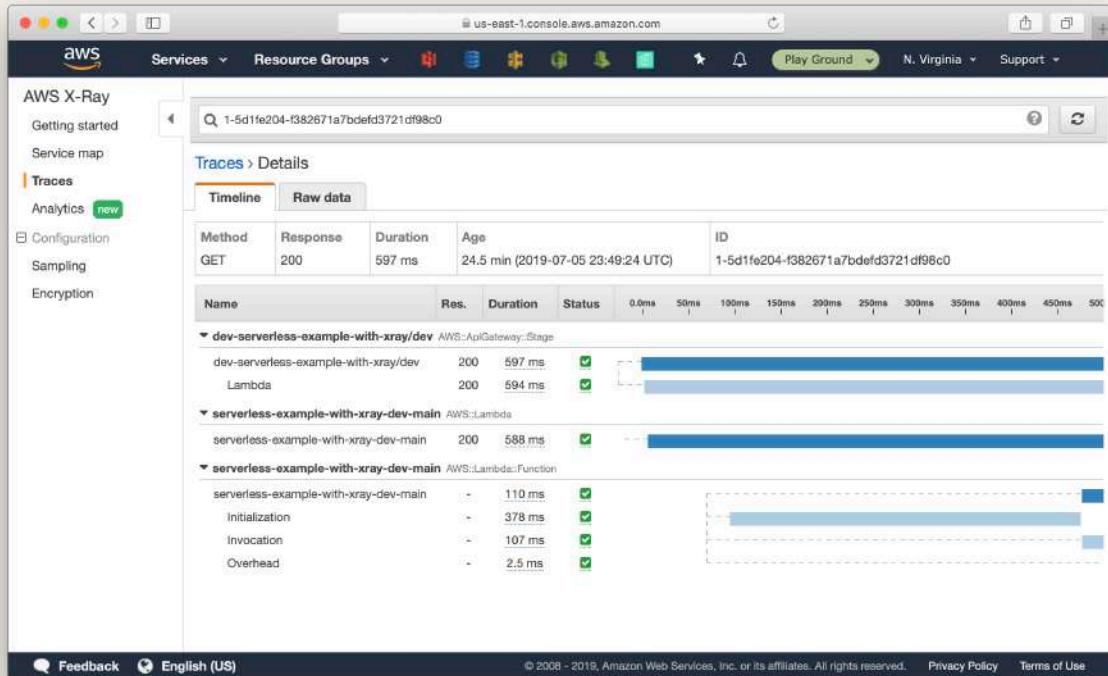
Head over to your AWS X-Ray console, and select **Traces** from the left menu.

The screenshot shows the AWS X-Ray Traces console. On the left, there's a navigation sidebar with options like 'Getting started', 'Service map', 'Traces' (which is selected and highlighted in orange), 'Analytics', 'Configuration', 'Sampling', and 'Encryption'. The main area has two sections: 'Trace overview' at the top and 'Trace list' below it. In the 'Trace overview' section, there's a search bar, a dropdown for 'Last 30 minutes', and a summary table with columns for 'URL', 'Avg response time', '% of Traces', and 'Response'. The table shows one trace for the URL 'https://wkd2y6o35.execute-api.us-east-1.amazonaws.com/dev/'. In the 'Trace list' section, there's a table with columns for 'ID', 'Age', 'Method', 'Response', 'Response time', 'URL', and 'Client IP'. It lists two traces: one with ID '...21df98c0' and another with ID '...55e9e773'. Both traces show a GET method, a 200 response, and a response time between 58.0 ms and 597 ms. The Client IP for both is 142.1.7.162. At the bottom of the page, there are links for 'Feedback', 'English (US)', and 'Privacy Policy'.

Select Traces from AWS X-Ray console

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.



Select a trace in AWS X-Ray console

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**:

The screenshot shows the AWS X-Ray console interface. On the left, there's a sidebar with options like Getting started, Service map, Traces (which is selected), Analytics, Configuration, Sampling, and Encryption. The main area has a search bar at the top with placeholder text "Enter service name, annotation, trace ID. Or click the Help icon for additional details." Below it is a "Trace overview" section with a table showing one trace entry:

URL	Avg response time	% of Traces	Response
https://wkd2y6o35.execute-api.us-east-1.amazonaws.com/dev/	533 ms	100.00%	3 OK, 0 Throttles

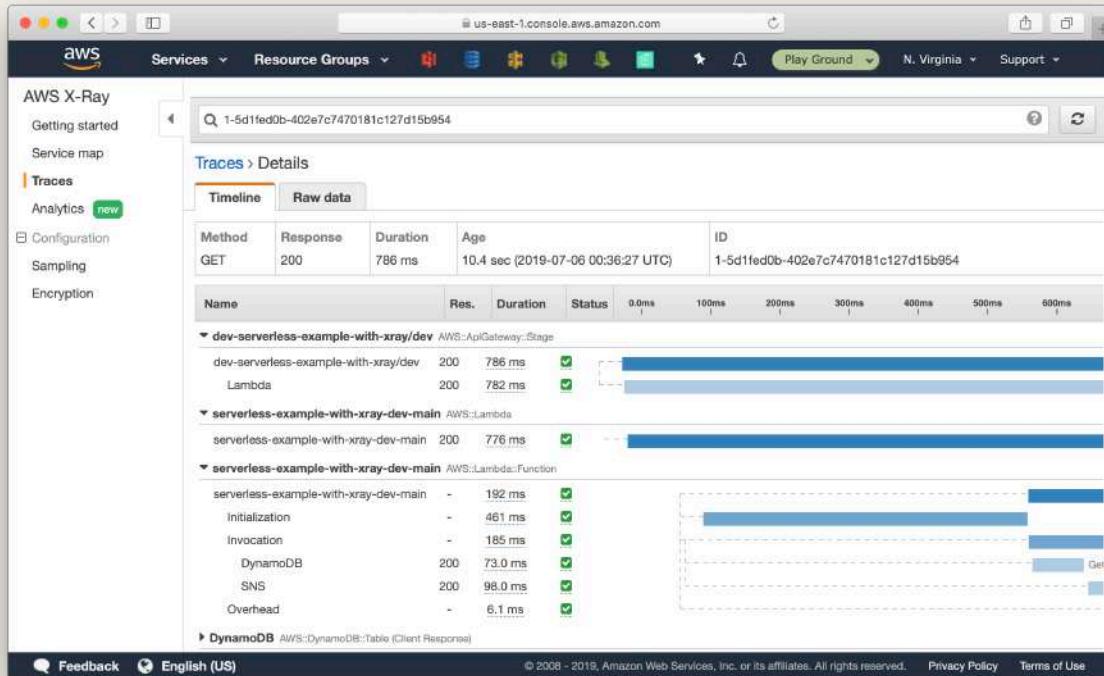
Below this is a "Trace list" section with a table showing four trace entries:

ID	Age	Method	Response	Response time	URL	Client IP
...2a069340	19.6 sec	GET	502	731 ms	https://wkd2y6o...	142.1.7.162
...d75a8f64	6.2 min	GET	200	744 ms	https://wkd2y6o...	142.1.7.162
...21df98c0	45.4 min	GET	200	597 ms	https://wkd2y6o...	142.1.7.162
...55e9e773	44.4 min	GET	200	58.0 ms	https://wkd2y6o...	142.1.7.162

At the bottom of the page, there are links for Feedback, English (US), Privacy Policy, and Terms of Use.

Select recent trace in AWS X-Ray console

Select the new trace.



[View updated trace in AWS X-Ray console](#)

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](#) has AWS X-Ray enabled by default so you can play around with the concepts we talked about in this chapter.



Help and discussion

[View the comments for this chapter on our forums](#)

Conclusion

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 [fill out our survey](#) or send us any comments or feedback you might have, via [email](#).

Also, if there are any other topics you'd like us to cover, please leave a comment in the discussion thread below!

Finally, if you've found this guide helpful, [please consider sponsoring us on GitHub](#).

Thank you and we hope you found this guide helpful!



Help and discussion

View the [comments for this chapter on our forums](#)

Serverless

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](#) 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:

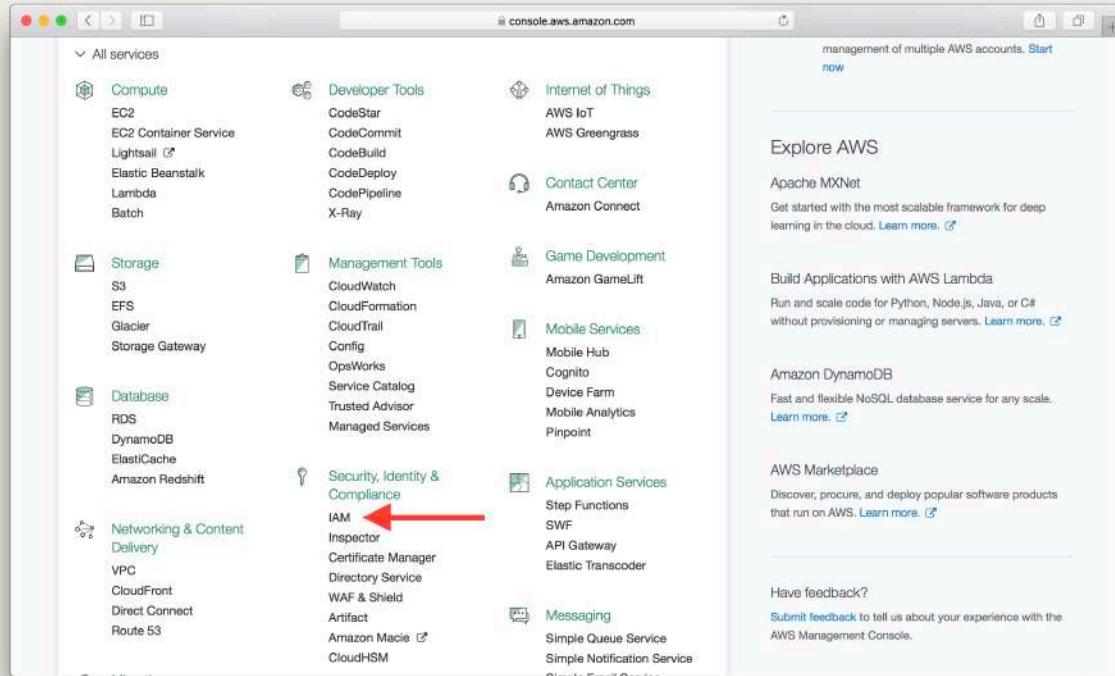
- [Enable API Gateway CloudWatch Logs](#)
- [Enable Lambda CloudWatch Logs](#)
- [Viewing API Gateway CloudWatch Logs](#)
- [Viewing Lambda CloudWatch Logs](#)

Let's get started.

Enable API Gateway CloudWatch Logs

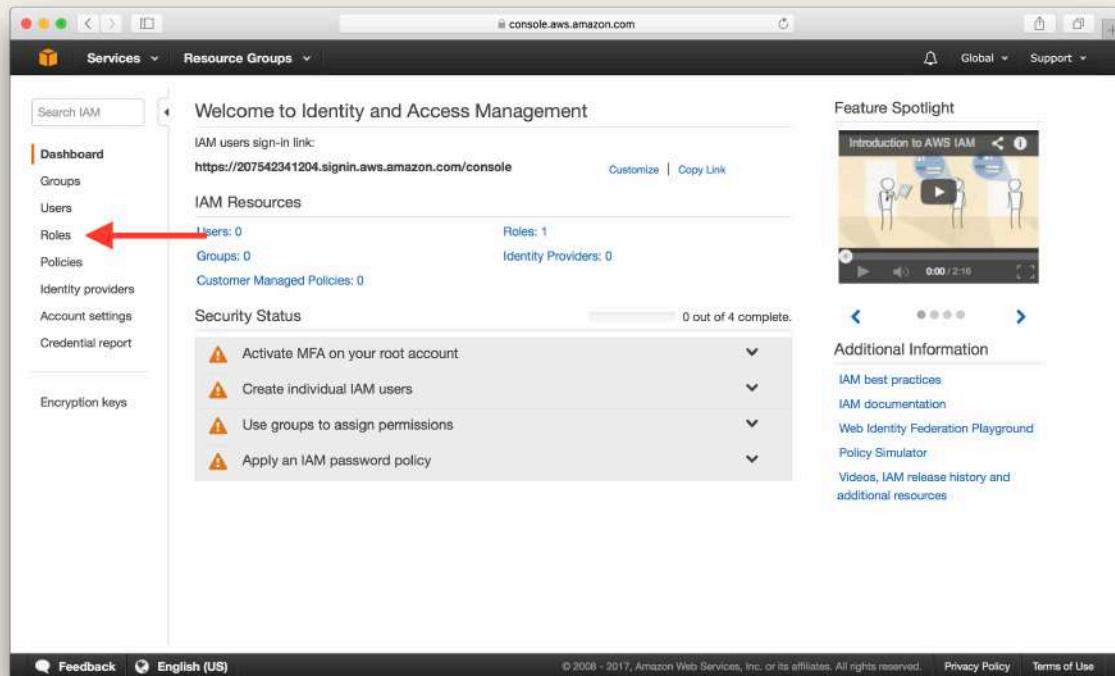
This is a two step process. First, we need to create an IAM role that allows API Gateway to write logs in CloudWatch. Then we need to turn on logging for our API project.

First, log in to your [AWS Console](#) and select IAM from the list of services.



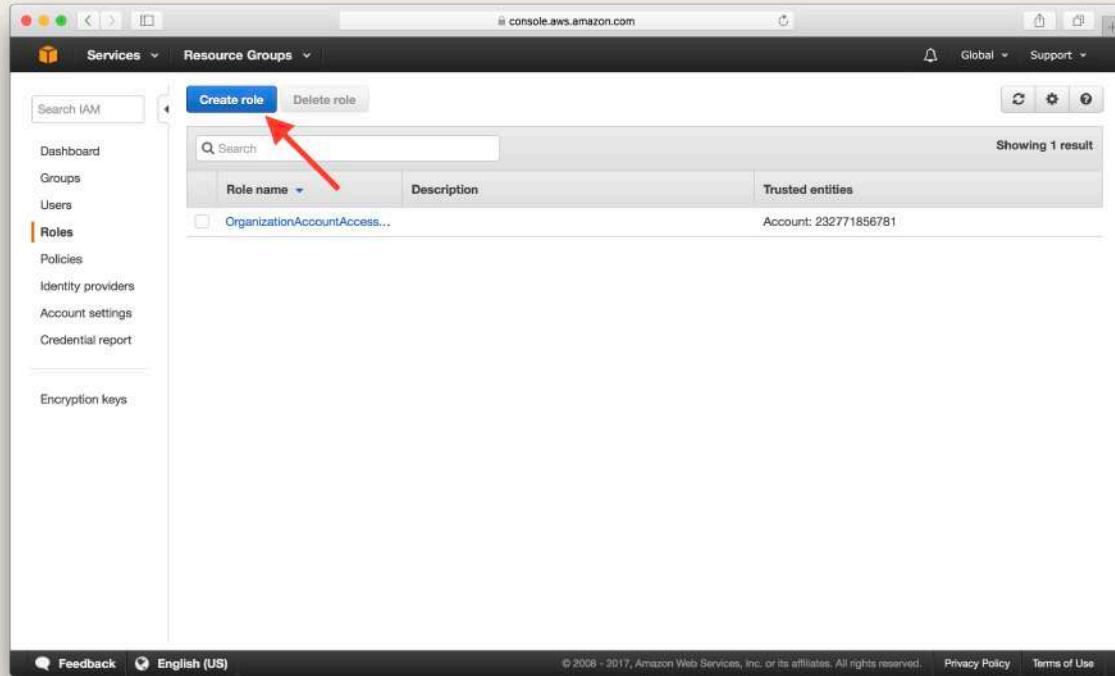
Select IAM Service Screenshot

Select **Roles** on the left menu.



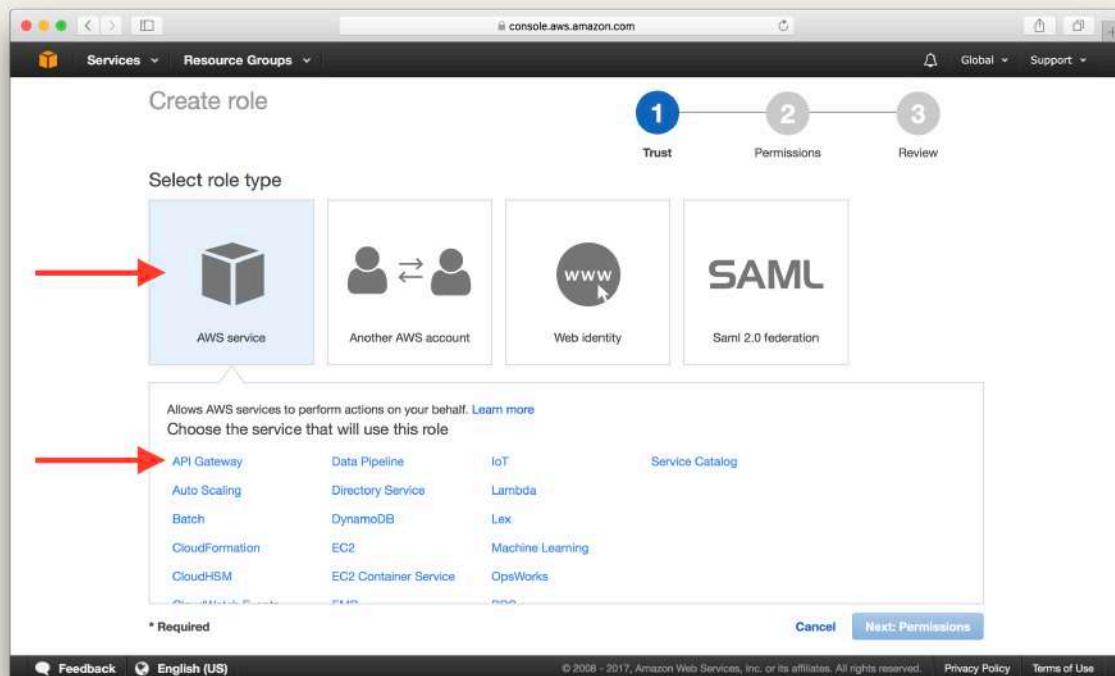
Select IAM Roles Screenshot

Select **Create Role**.



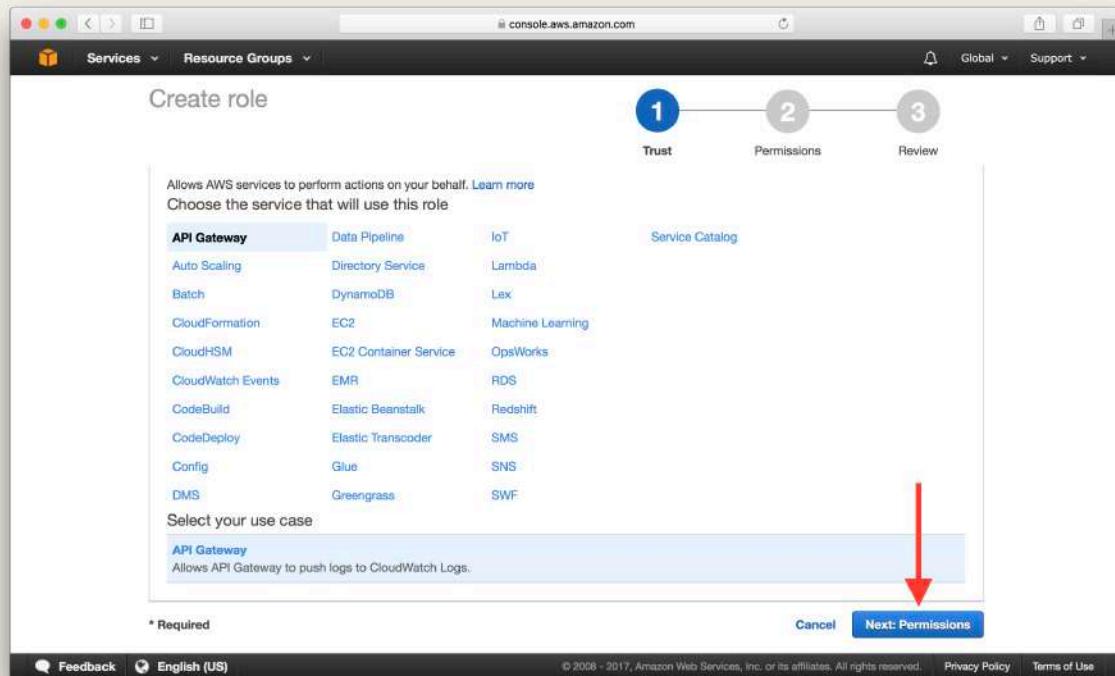
Select Create IAM Role Screenshot

Under **AWS service**, select **API Gateway**.



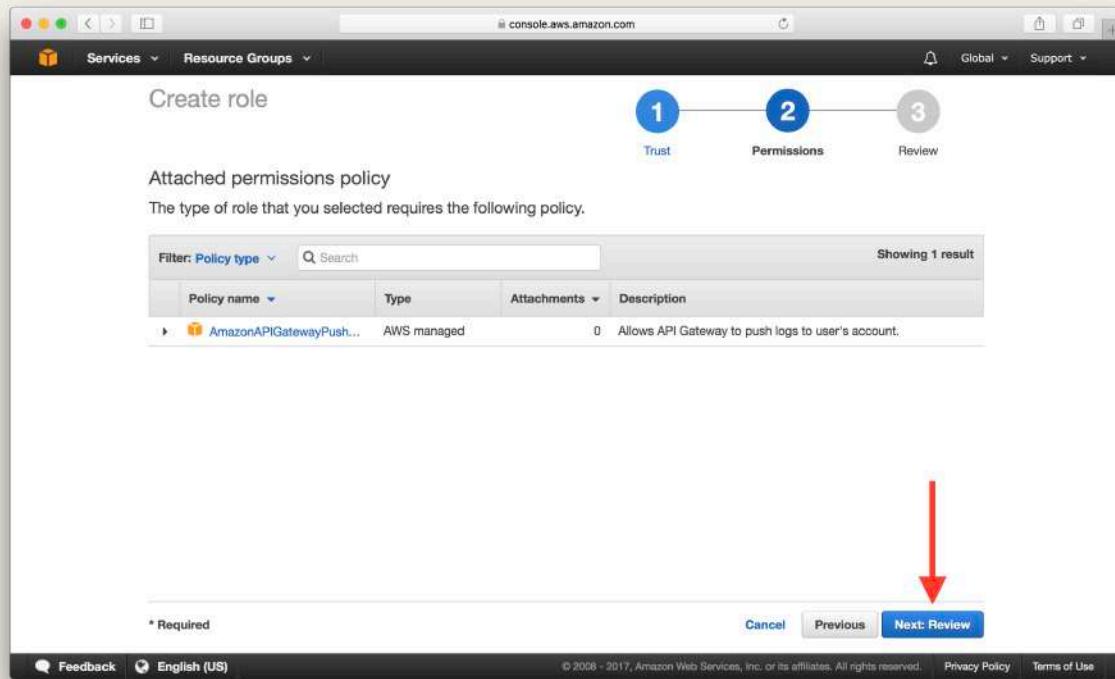
Select API Gateway IAM Role Screenshot

Click **Next: Permissions**.



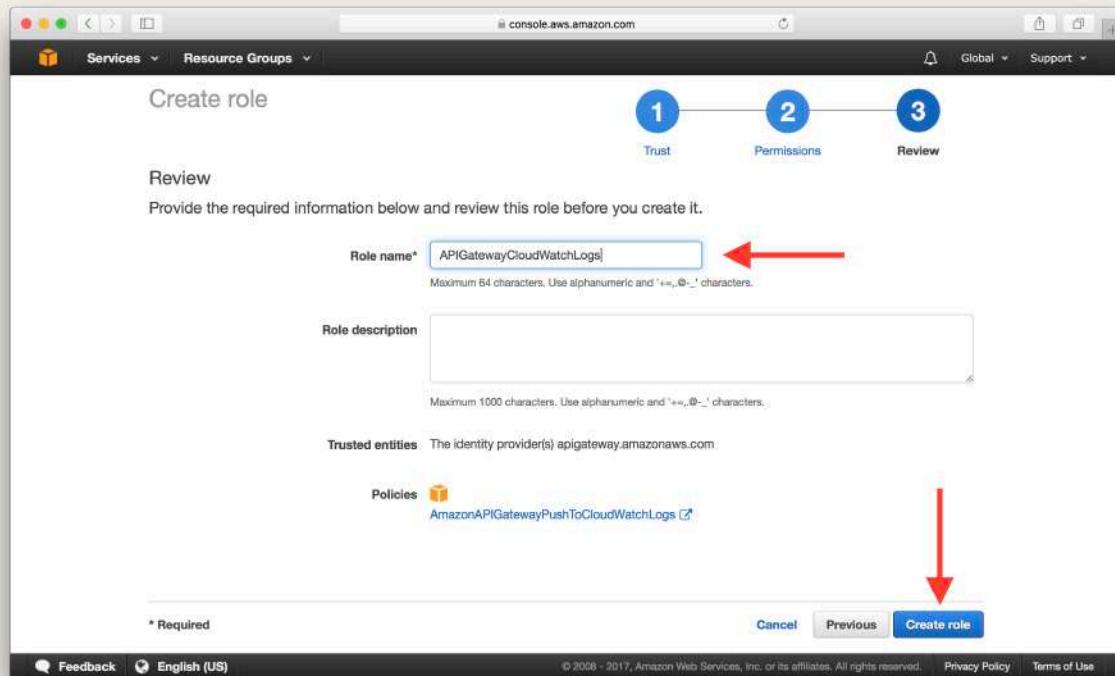
Select IAM Role Attach Permissions Screenshot

Click **Next: Review**.



Select Review IAM Role Screenshot

Enter a **Role name** and select **Create role**. In our case, we called our role `APIGatewayCloudWatchLogs`.



Fill in IAM Role Info Screenshot

Click on the role we just created.

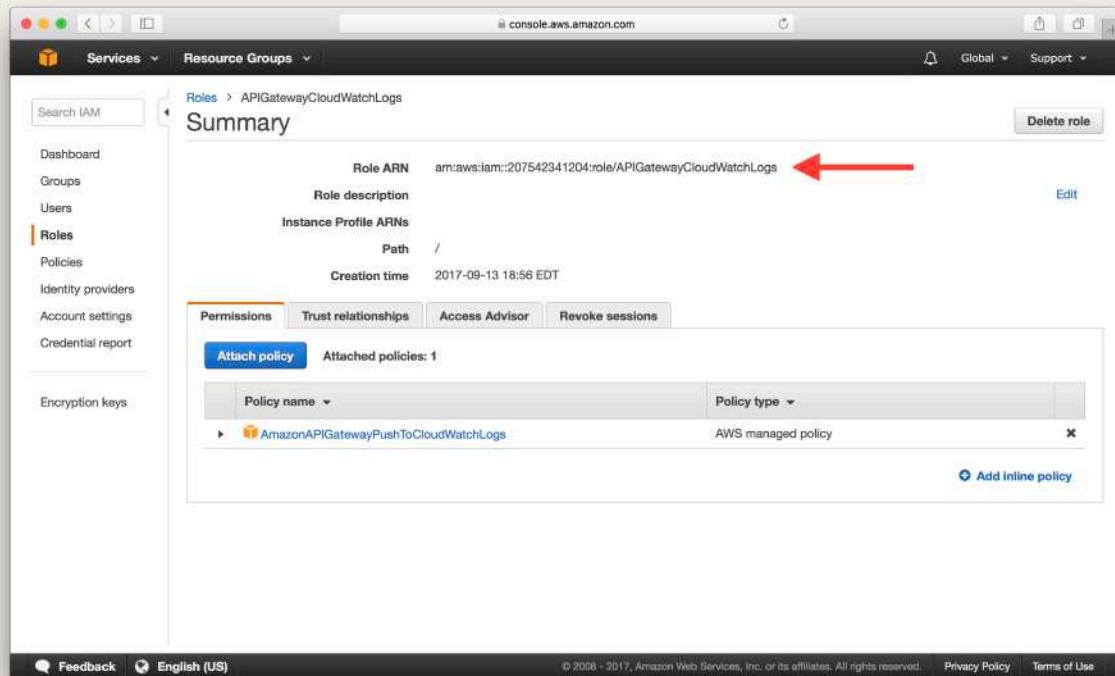
The screenshot shows the AWS IAM Roles page. On the left, there's a sidebar with options like Dashboard, Groups, Users, Roles (which is selected), Policies, Identity providers, Account settings, Credential report, and Encryption keys. The main area has a search bar and buttons for 'Create role' and 'Delete role'. A table lists two roles:

Role name	Description	Trusted entities
APIGatewayCloudWatchLogs	AWS service: apigateway	
OrganizationAccountAccess...	Account: 232771856781	

A red arrow points to the first row, highlighting the 'APIGatewayCloudWatchLogs' role.

Select Created API Gateway IAM Role Screenshot

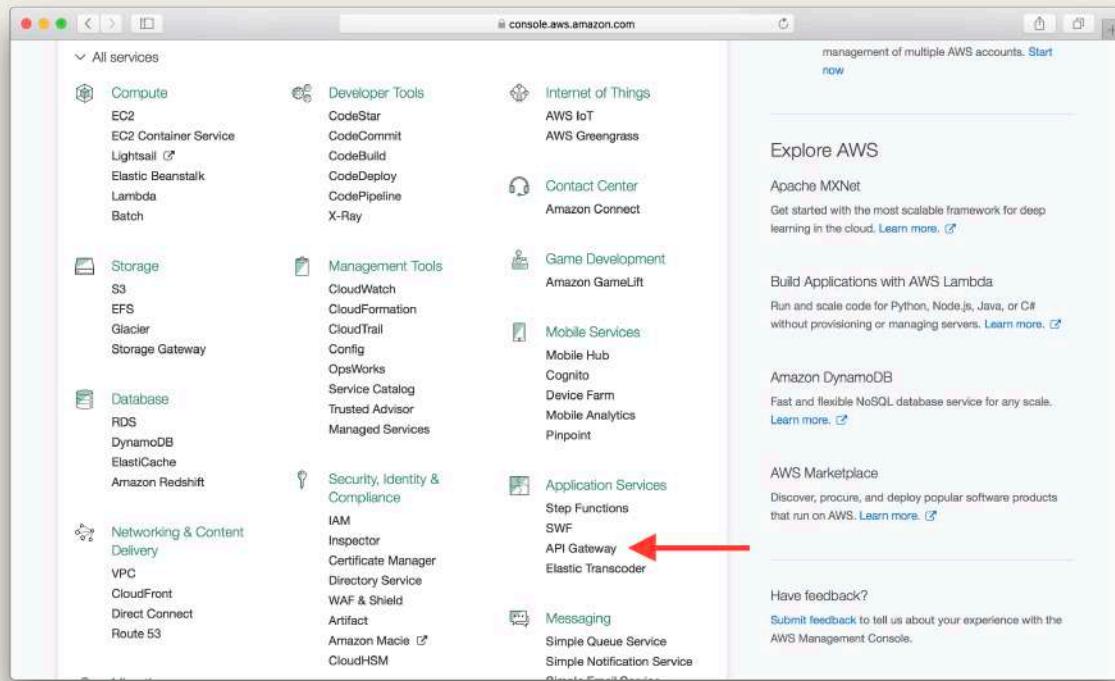
Take a note of the **Role ARN**. We will be needing this soon.



IAM Role ARN Screenshot

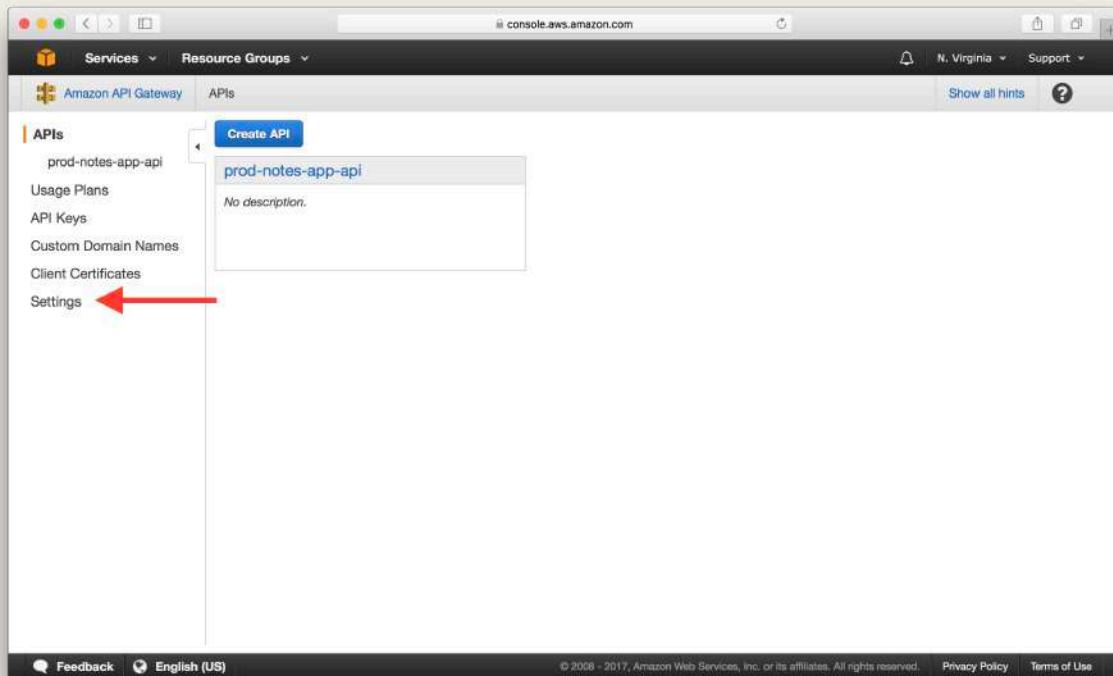
Now that we have created our IAM role, let's turn on logging for our API Gateway project.

Go back to your [AWS Console](#) and select API Gateway from the list of services.



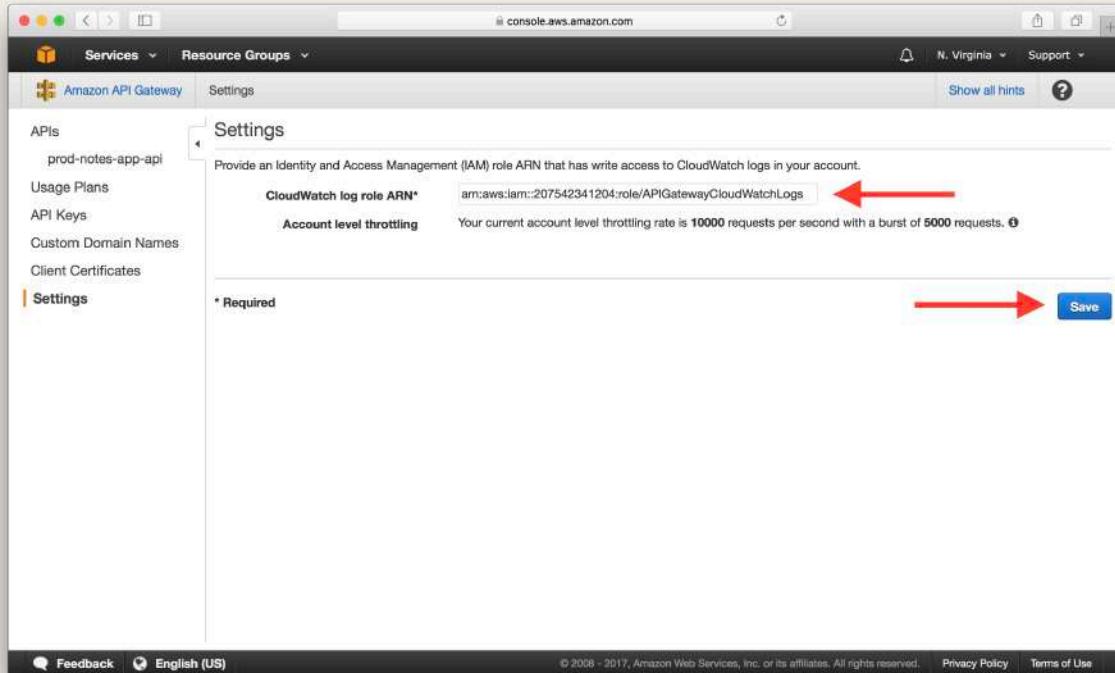
Select API Gateway Service Screenshot

Select **Settings** from the left panel.



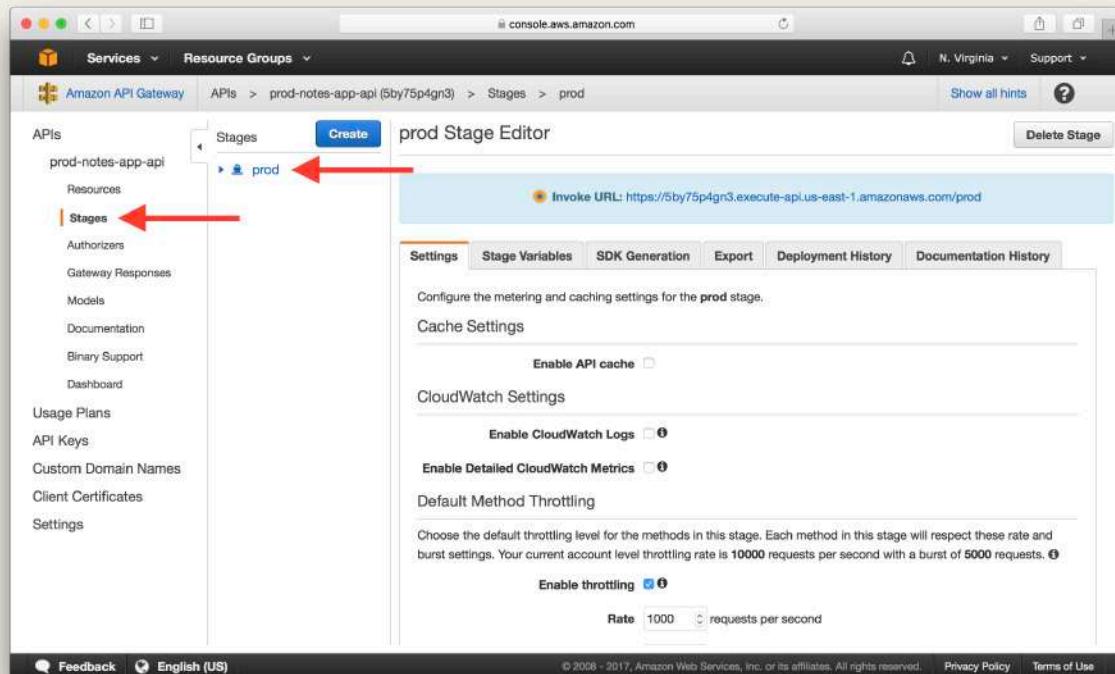
Select API Gateway Settings Screenshot

Enter the ARN of the IAM role we just created in the **CloudWatch log role ARN** field and hit **Save**.



Fill in API Gateway CloudWatch Info Screenshot

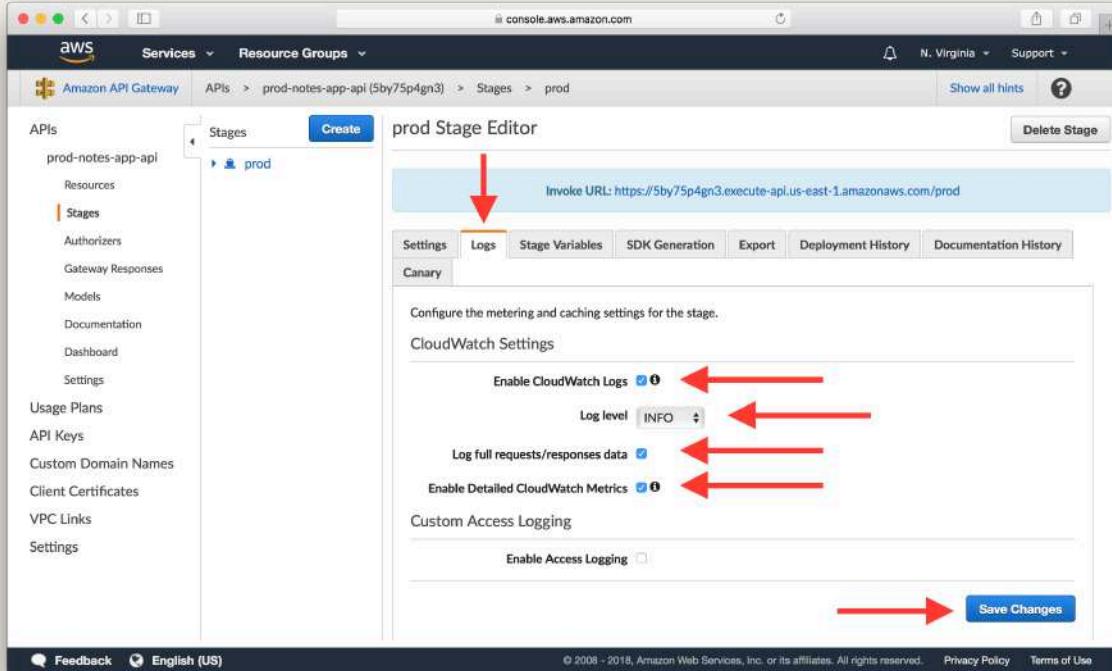
Select your API project from the left panel, select **Stages**, then pick the stage you want to enable logging for. For the case of our [Notes App API](#), we deployed to the prod stage.



Select API Gateway Stage Screenshot

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.



Fill in API Gateway Logging Info Screenshot

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 recommend leaving on. Here is [how to enable access logs for your API Gateway project](#).

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:

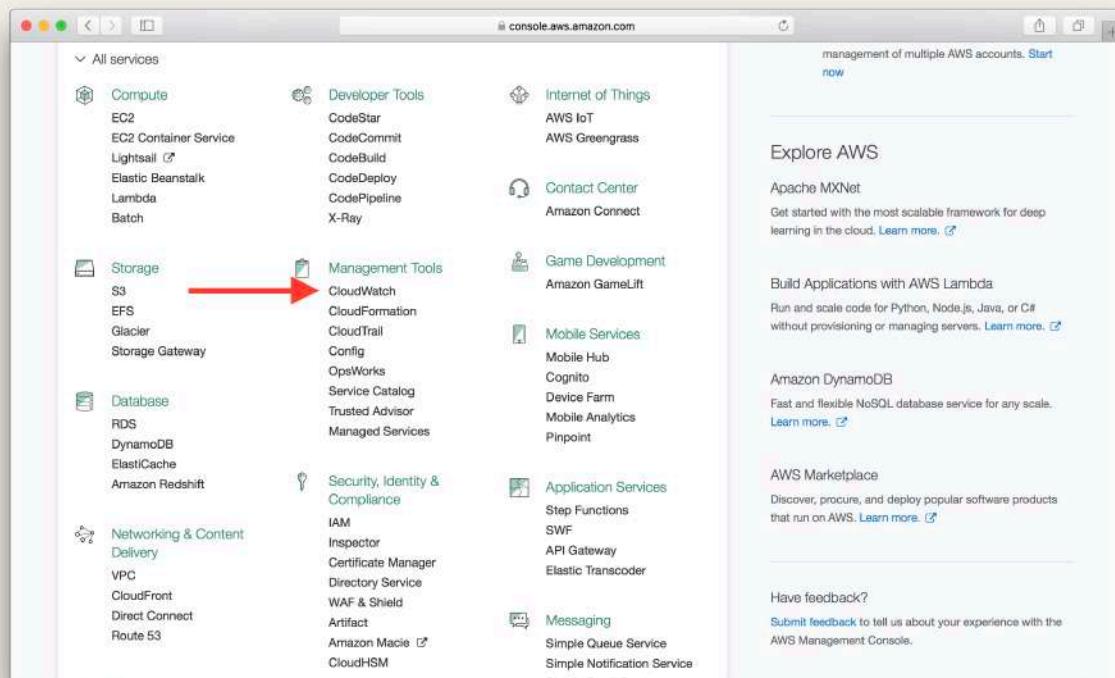
```
export function main(event, context, callback) {
  console.log('Hello world');
```

```
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](#) and select CloudWatch from the list of services.



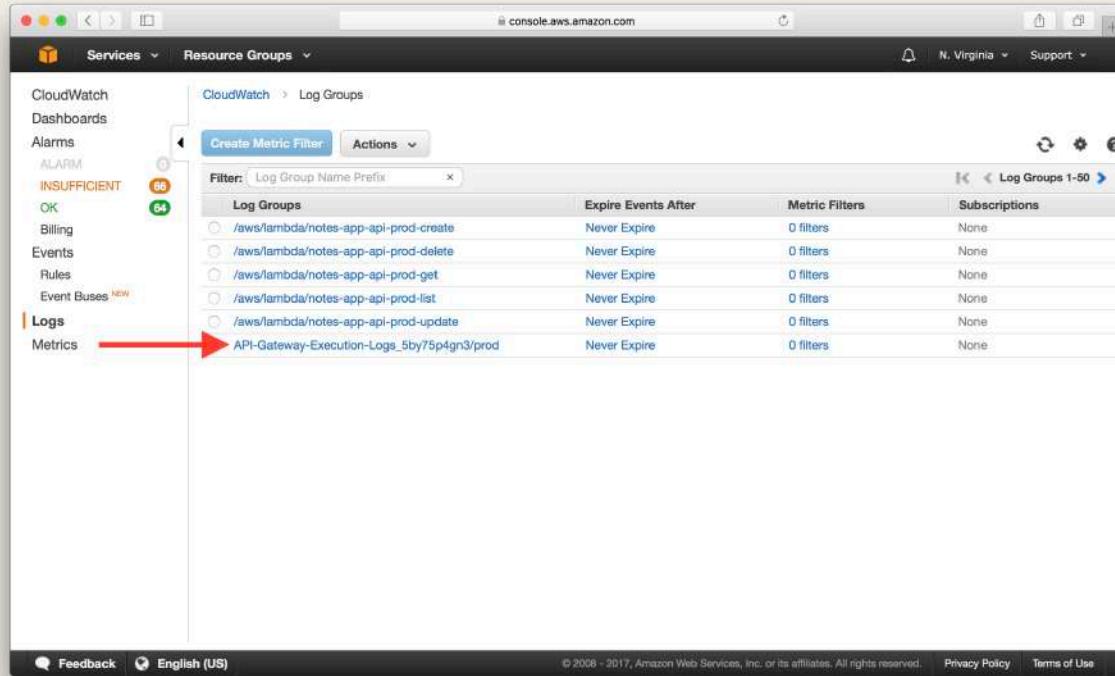
Select CloudWatch Service Screenshot

Select **Logs** from the left panel.

The screenshot shows the AWS CloudWatch Metrics Summary page. The left sidebar has 'Metrics' selected, indicated by a red arrow. The main content area displays a 'Metric Summary' section with a search bar and a 'Browse Metrics' button. Below it is an 'Alarm Summary' section showing 66 alarms in the 'INSUFFICIENT' state. At the bottom, there are four line charts for 'TargetTracking-table/prod-beta...', each showing consumption metrics over time.

Select CloudWatch Logs Screenshot

Select the log group prefixed with **API-Gateway-Execution-Logs_** followed by the API Gateway id.



The screenshot shows the AWS CloudWatch Log Groups interface. The left sidebar has 'Logs' selected, indicated by a red arrow. The main area displays a table of log groups with the following columns: Log Groups, Expire Events After, Metric Filters, and Subscriptions. The table lists several log groups, including '/aws/lambda/notes-app-api-prod-create', '/aws/lambda/notes-app-api-prod-delete', '/aws/lambda/notes-app-api-prod-get', '/aws/lambda/notes-app-api-prod-list', and '/aws/lambda/notes-app-api-prod-update'. The last row, '/aws/lambda/notes-app-api-prod-update', is highlighted with a pink background. The status bar at the bottom indicates 'Feedback', 'English (US)', and copyright information: '© 2006 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.' followed by 'Privacy Policy' and 'Terms of Use'.

Select CloudWatch API Gateway Log Group Screenshot

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.

The screenshot shows the AWS CloudWatch Logs interface. On the left, there's a navigation sidebar with 'Services' (CloudWatch, Dashboards, Alarms), 'Logs' (selected, with 66 items), and 'Metrics'. The main area shows a list of 'Log Streams' under the heading 'Streams for API-Gateway-Execution-Logs...'. A red arrow points to the first item in the list: 'ec8956637a99787bd197eacd77acce5e'. The list includes various log stream names and their last event times. At the bottom of the page, there are links for 'Feedback', 'English (US)', and legal notices.

Select CloudWatch API Gateway Log Stream Screenshot

This shows you the log entries grouped by request.

The screenshot shows the AWS CloudWatch Log Groups interface. On the left sidebar, under the 'Logs' section, there are two red arrows pointing to specific log entries. The first arrow points to a group of log entries for September 5, 2017, at 19:18:00. The second arrow points to a group of log entries for September 13, 2017, at 15:08:28. Both groups contain multiple log entries with detailed message logs.

Time (UTC +00:00)	Message
2017-09-05	(eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method request path: {id=de1b2bf0-926e-11e7-9c19-35de5ccc401ec} (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method request query string: {} (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method request headers: {x-AMZ-Date=20170905T191756Z, Accept=application/json, X-Amzn-Trace-Id=Root=00000000000000000000000000000000} (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method request body: (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Endpoint request URI: https://lambda.us-east-1.amazonaws.com/2015-03-31/functions/i... (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Endpoint request headers: {x-amzn-lambda-integration-tag=eb3ad80-926e-11e7-ac8d-b7ef47cf7d48} (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Endpoint request body: (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Endpoint response body before transformations: {"resource":"/notes/{id}", "path":"/notes/de1b2bf0-926e-11e7-9c19-35de5ccc401ec"} (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Endpoint response body: {"statusCode":200, "headers":{"Access-Control-Allow-Origin":"*"}, "body":{}}, {"x-amzn-Remapped-Content-Length":0, x-amzn-RequestId": "00000000000000000000000000000000"} (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method response body after transformations: {"attachment": "https://notes-app-uploads.s3.amazonaws.com/11e7-9c19-35de5ccc401ec/1505545355444.png"} (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method response headers: {"Access-Control-Allow-Origin": "https://notes-app-uploads.s3.amazonaws.com", "Access-Control-Allow-Credentials": "true"} (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Successfully completed execution (eb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method completed with status: 200
2017-09-13	(eb22b95c-9895-11e7-87c5-772559cc6127) Verifying Usage Plan for request: 8622b95c-9895-11e7-87c5-772559cc6127. API Key: A (eb22b95c-9895-11e7-87c5-772559cc6127) API Key authorized because method 'GET /notes' does not require API Key. Request will be charged against the usage plan. (eb22b95c-9895-11e7-87c5-772559cc6127) Usage Plan check succeeded for API Key and API Stage 5by75p4gn3/prod (eb22b95c-9895-11e7-87c5-772559cc6127) Starting execution for request: 8622b95c-9895-11e7-87c5-772559cc6127 (eb22b95c-9895-11e7-87c5-772559cc6127) HTTP Method: GET, Resource Path: /notes (eb22b95c-9895-11e7-87c5-772559cc6127) Method request path: [] (eb22b95c-9895-11e7-87c5-772559cc6127) Method request query string: {}

CloudWatch API Gateway Log Entries Screenshot

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.

The screenshot shows the AWS CloudWatch Log Groups interface. On the left, a navigation menu includes 'Services' (selected), 'Resource Groups', 'CloudWatch' (selected), 'Dashboards', 'Alarms' (with 66 notifications), 'Billing' (with 64 notifications), 'Events', 'Rules', 'Event Buses', 'Logs' (selected), and 'Metrics'. The main area displays a table of log groups with the following columns: 'Log Groups', 'Expire Events After', 'Metric Filters', and 'Subscriptions'. A red arrow points to the first log group in the list, which is '/aws/lambda/notes-app-api-prod-create'. The table data is as follows:

Log Groups	Expire Events After	Metric Filters	Subscriptions
/aws/lambda/notes-app-api-prod-create	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-delete	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-get	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-list	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-update	Never Expire	0 filters	None
API-Gateway-Execution-Logs_5by75p4gn3/prod	Never Expire	0 filters	None

Select CloudWatch Lambda Log Group Screenshot

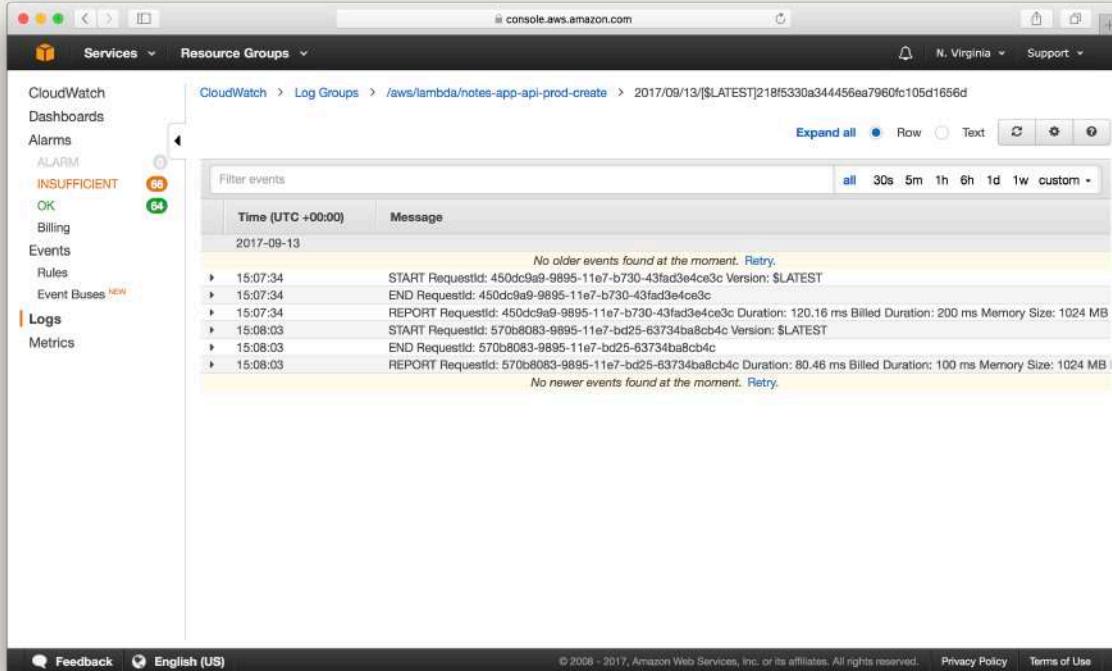
Select the first stream.

The screenshot shows the AWS CloudWatch Logs interface for a Lambda function. The left sidebar shows various AWS services like CloudWatch, Dashboards, Alarms, Billing, Events, Rules, Event Buses, and Logs. Under Logs, there are 66 ALARM and 64 INSUFFICIENT items. The main area displays a list of log streams with their last event times. A red arrow points to the first log stream in the list.

Log Stream	Last Event Time
2017/09/13[\$LATEST]218f5330a344456ea7960fc105d1656d	2017-09-13 11:08 UTC-4
2017/09/13[\$LATEST]12e001232056404ba058fe25b004d35d	2017-09-13 05:15 UTC-4
2017/09/13[\$LATEST]d13860998d74404c8799025a704e4fae	2017-09-13 01:21 UTC-4
2017/09/13[\$LATEST]fe3067e7fee94dab0bb006462816118	2017-09-13 00:05 UTC-4
2017/09/12[\$LATEST]7bf801a83ff4825b31bddbd160c4c1b	2017-09-12 12:55 UTC-4
2017/09/12[\$LATEST]fe5405a8595144ad8a4f38c02e022414	2017-09-12 10:04 UTC-4
2017/09/12[\$LATEST]85bfafab9d40b649ac8f003263c3ca5ae5	2017-09-12 09:12 UTC-4
2017/09/12[\$LATEST]1184b9d215cf49b4a3b07070f4a8d471	2017-09-12 05:13 UTC-4
2017/09/12[\$LATEST]aac137430ea3442d9f5beef045a9464	2017-09-12 04:04 UTC-4
2017/09/12[\$LATEST]8195aa2531b84ab79fbf39e9759ed98a	2017-09-12 03:26 UTC-4
2017/09/12[\$LATEST]9861df5d8f4741159a4c9369bd24d2e98	2017-09-11 23:28 UTC-4
2017/09/11[\$LATEST]755739a02ddc409fb029dbd9bf5ee15b	2017-09-11 16:49 UTC-4
2017/09/11[\$LATEST]75b5e5dd8eddf74cfca402a64fd54e01e7	2017-09-11 15:10 UTC-4
2017/09/11[\$LATEST]12ecfd2d8e0744da8e944641228e1c35	2017-09-11 09:57 UTC-4
2017/09/11[\$LATEST]8f63fbe058b14d5b8ds10539e344a33d	2017-09-11 08:27 UTC-4
2017/09/11[\$LATEST]d13d2a109a5e469fbbeefadcb12cfra0d	2017-09-11 20:36 UTC-4
2017/09/11[\$LATEST]00bfa776c4044ab3b7228bfe684344cb	2017-09-10 20:13 UTC-4
2017/09/10[\$LATEST]2ed0cc4f8b277da98e16e31ecab275d8	2017-09-10 09:06 UTC-4

Select CloudWatch Lambda Log Stream Screenshot

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.



CloudWatch Lambda Log Entries Screenshot

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.



Help and discussion

View the [comments](#) for this chapter on our forums

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](#). Check out [Common 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](#)
- [Missing IAM Policy](#)
- [Lambda Function Error](#)
- [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](#).

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.

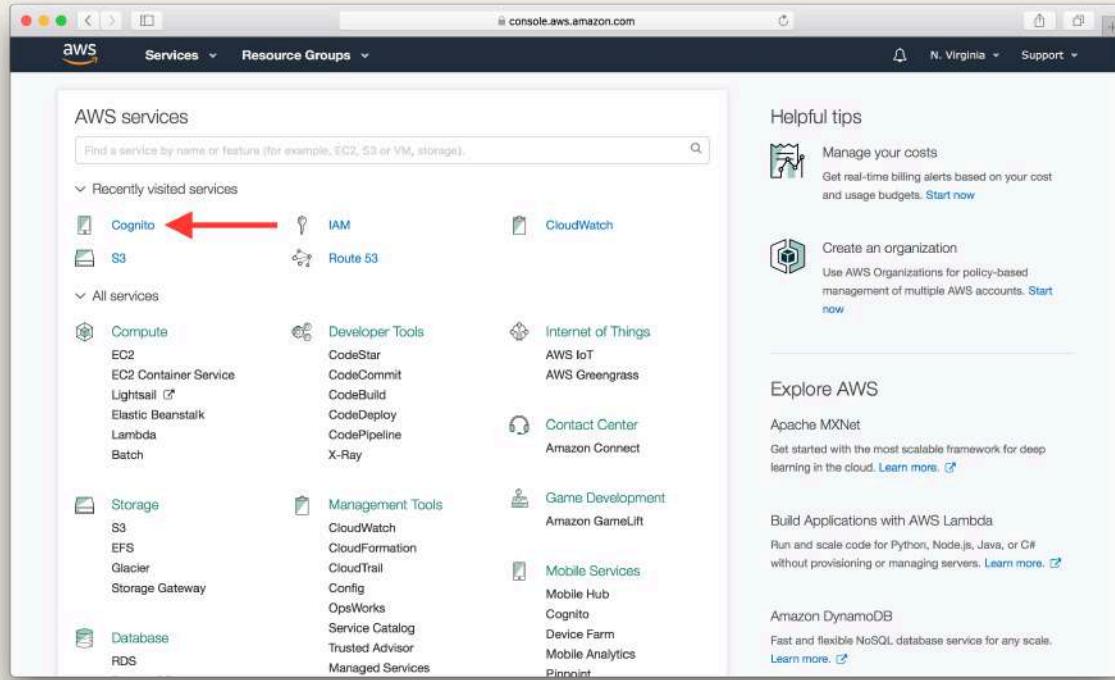
Missing IAM Policy

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](#).

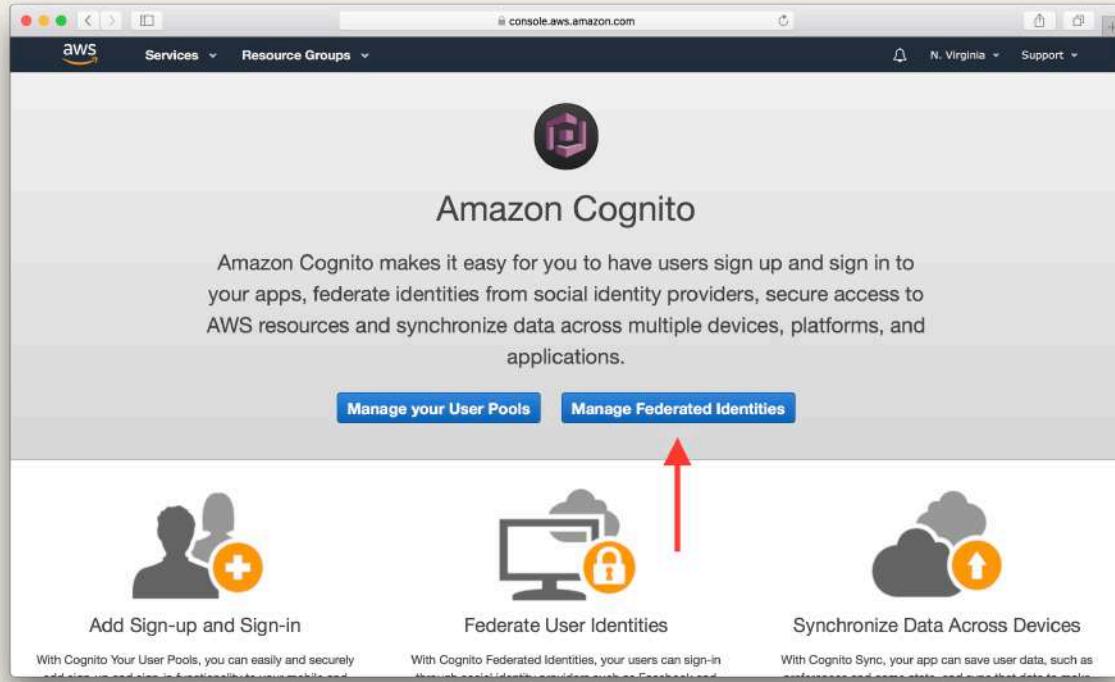
Before we can use the simulator we first need to find out the name of the IAM role that we are using to connect to API Gateway. We had created this role back in the [Create a Cognito identity pool](#) chapter.

Select **Cognito** from your [AWS Console](#).



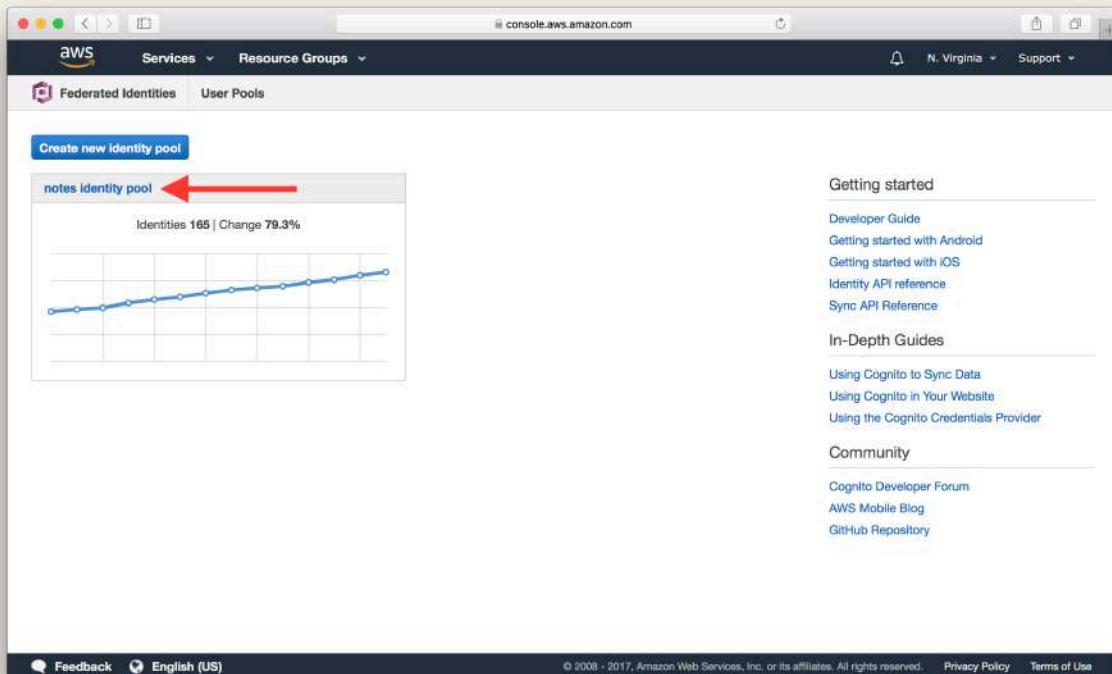
Select Cognito Service Screenshot

Next hit the **Manage Federated Identities** button.



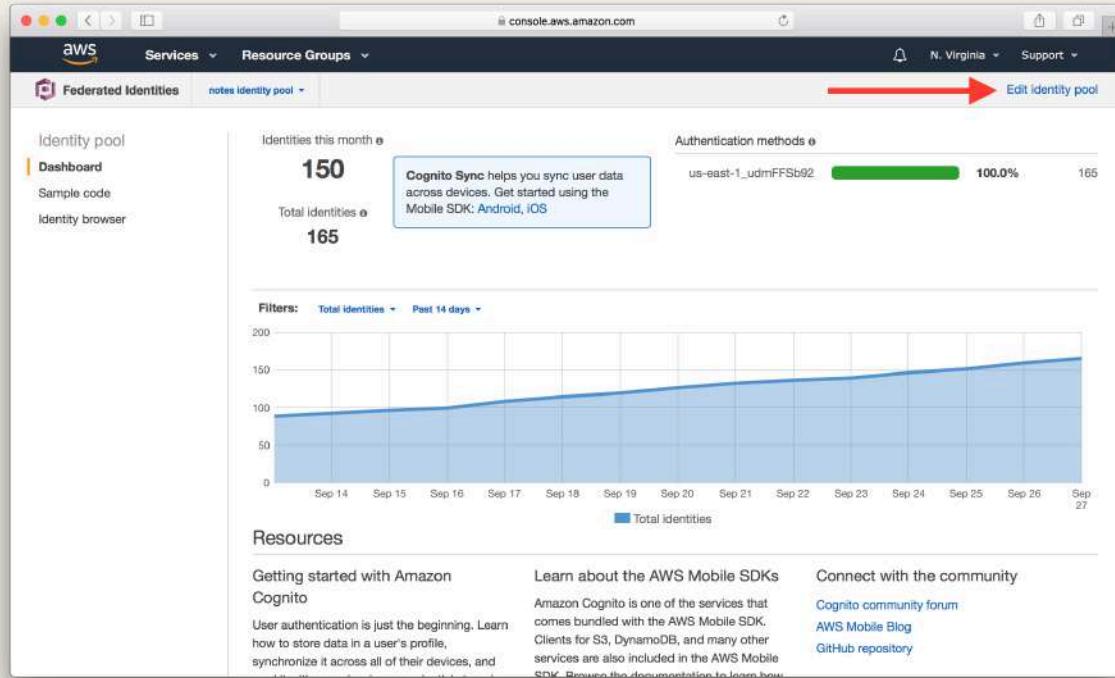
Click Manage Federated Identities Screenshot

And select your Identity Pool. In our case it's called `notes identity pool`.



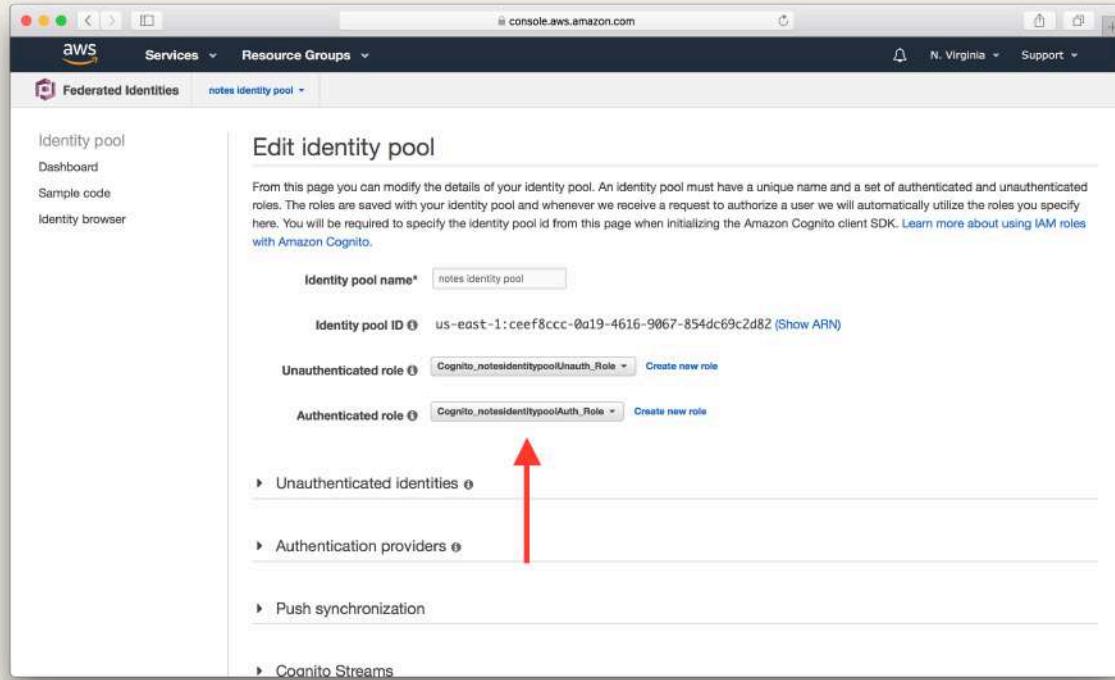
Select identity pool Screenshot

Click **Edit identity pool** at the top right.



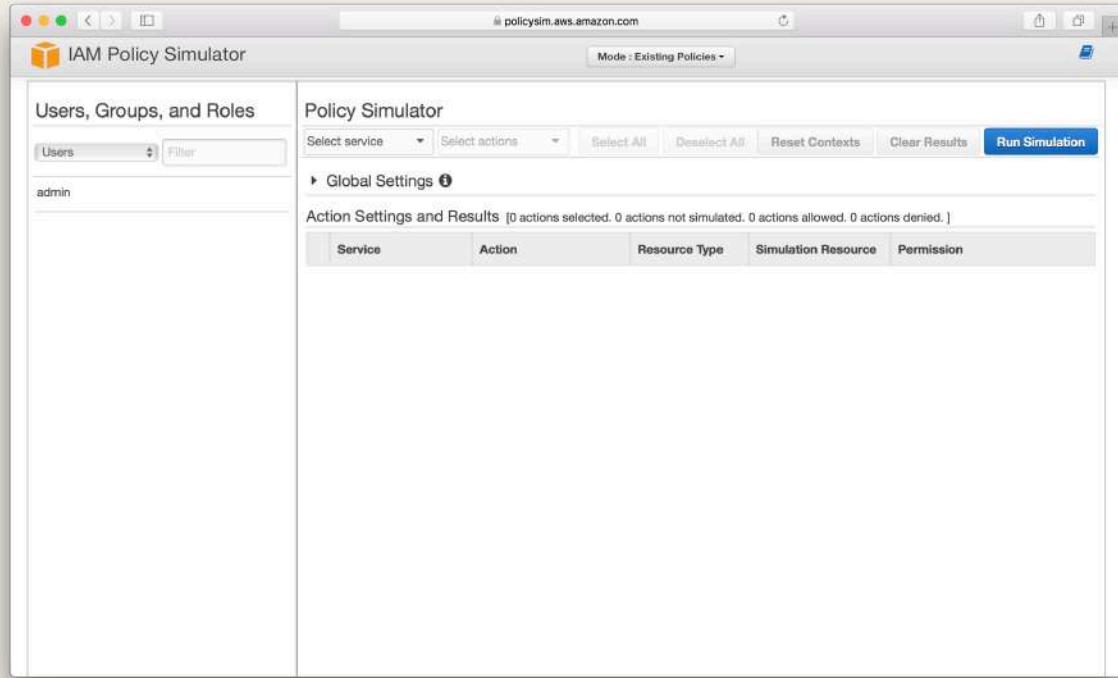
Click Edit identity pool Screenshot

Here make a note of the name of the **Authenticated role**. In our case it is `Cognito_notesidentitypoolAuth_Role`.



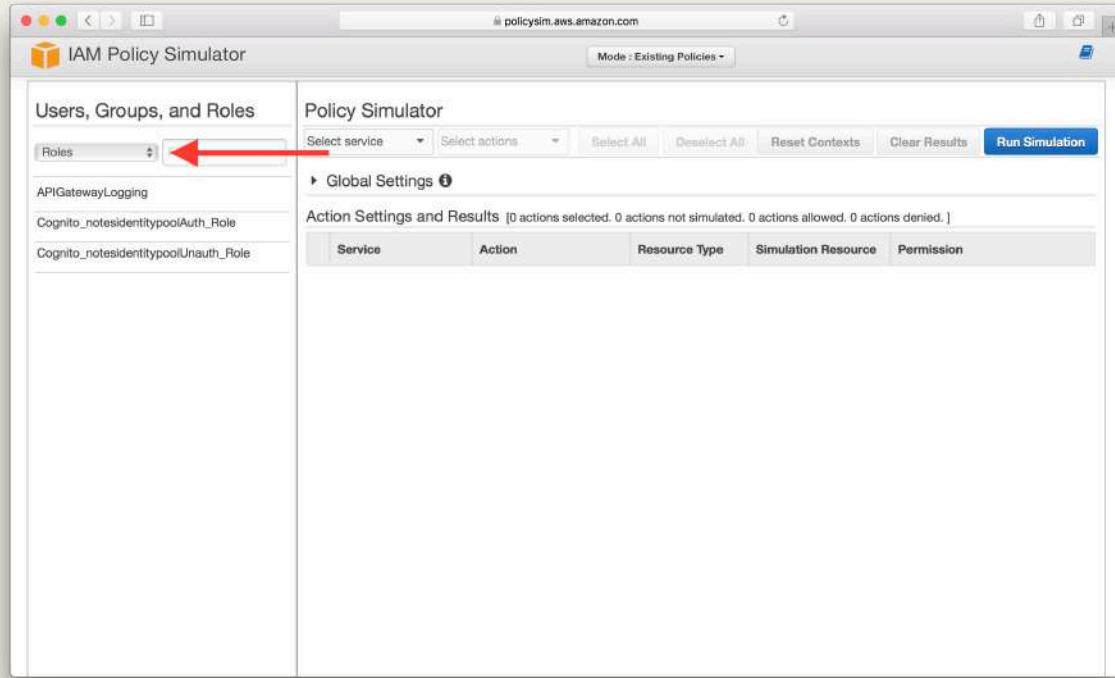
Identity Pool Auth Role Screenshot

Now that we know the IAM role we are testing, let's open up the [IAM Policy Simulator](#).



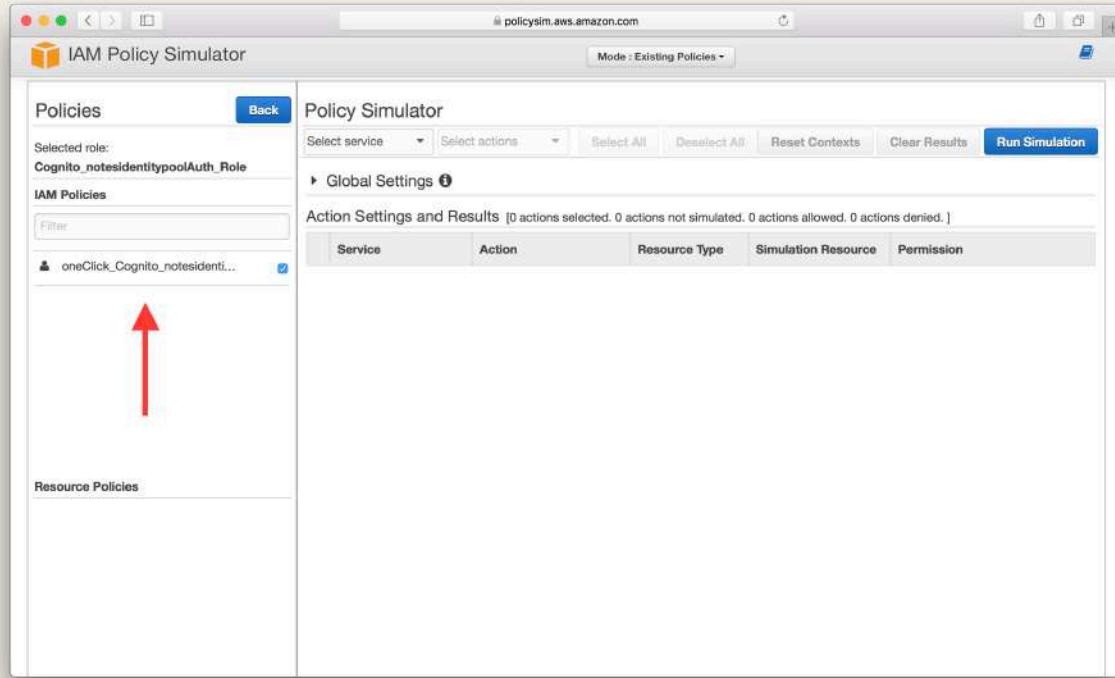
Open IAM Policy Simulator

Select **Roles**.



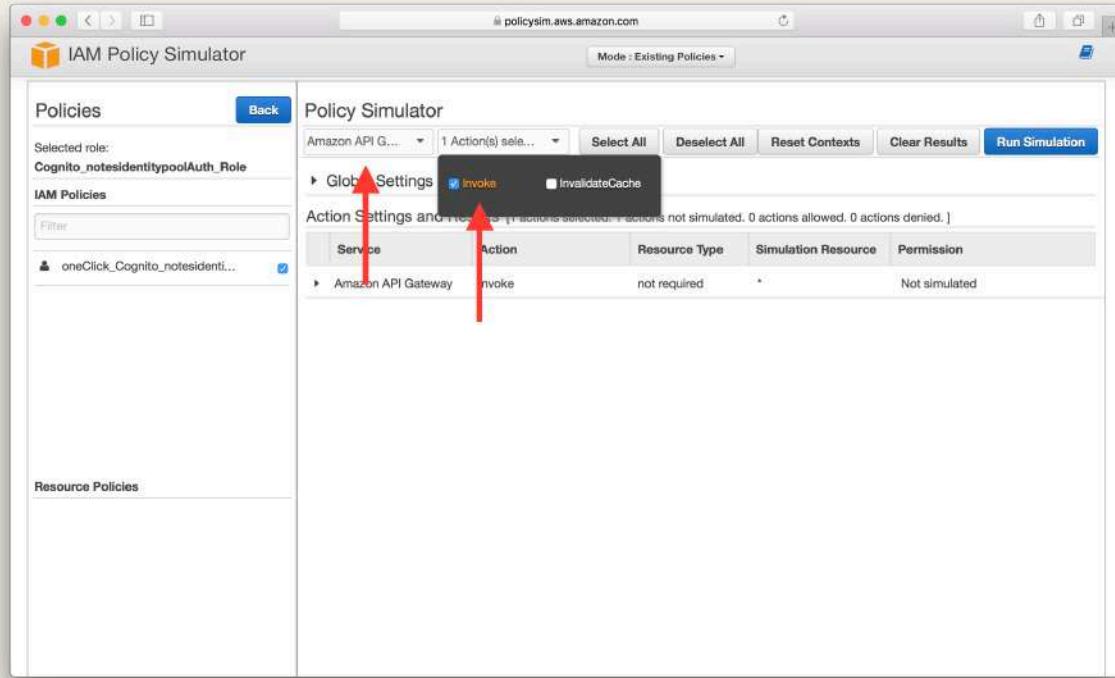
Select IAM Service Simulator Roles

Select the IAM role that we made a note of in the steps above. In our case it is Cognito_notesidentitypoolAuth_Role.



Select IAM Service Simulator Role

Select **API Gateway** as the service and select the **Invoke** action.



Select IAM Service Simulator 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](#) 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/*`.

The screenshot shows the IAM Policy Simulator interface. On the left, there's a sidebar with 'Policies' and 'Resource Policies'. The main area is titled 'Policy Simulator' and shows a table of results. A red arrow points to the 'Run Simulation' button at the top right of the table header. Another red arrow points to the 'Resource' input field where 'arn:aws:execute-api:us-east-1:ly55wbovq4/*' is entered.

Service	Action	Resource Type	Simulation Resource	Permission
Amazon API Gateway	Invoke	execute-api-general	execute-api-general	Not simulated

Resource You can specify the resource and context keys used to simulate this action. By default the simulation resource is **.
execute-api-general arn:aws:execute-api:us-east-1:ly55wbovq4/*

Enter API Gateway Endpoint ARN

If your IAM role is configured properly you should see **allowed** under **Permission**.

The screenshot shows the IAM Policy Simulator interface. On the left, there's a sidebar with 'Policies' and 'Resource Policies'. The main area is titled 'Policy Simulator' and shows 'Amazon API G...' selected. A table lists a single action: 'Amazon API Gateway | Invoke' under 'Service', 'execute-api-general' under 'Action', 'execute-api-general' under 'Resource Type', and 'execute-api-general' under 'Simulation Resource'. The 'Permission' column shows 'allowed' with '1 matching statements.' A red arrow points upwards from the bottom right towards the 'allowed' status.

IAM Service Simulator Permission Allowed

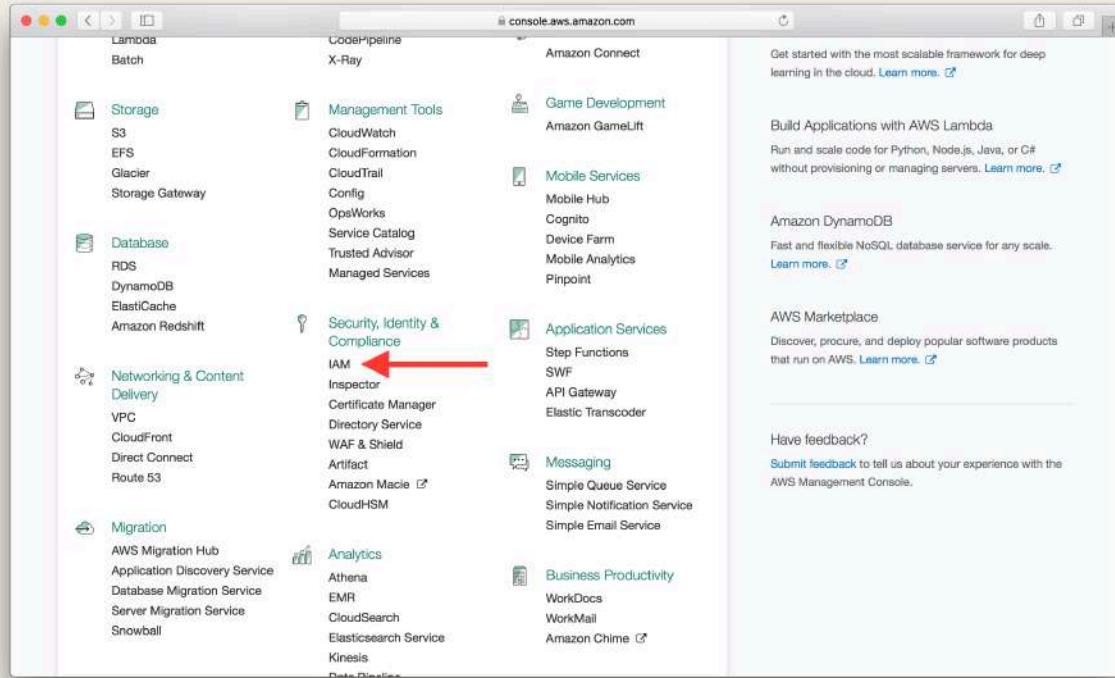
But if something is off, you'll see **denied**.

The screenshot shows the IAM Policy Simulator interface. On the left, there's a sidebar with 'Policies' and 'Resource Policies'. The main area is titled 'Policy Simulator' and shows 'Amazon API G...' selected. A table lists actions: 'Service' (Amazon API Gateway), 'Action' (Invoke), 'Resource Type' (execute-api-general), 'Simulation Resource' (execute-api-general), and 'Permission' (denied). A red arrow points to the 'denied' status in the table.

Service	Action	Resource Type	Simulation Resource	Permission
Amazon API Gateway	Invoke	execute-api-general	execute-api-general	denied Implicitly denied (no mat...)

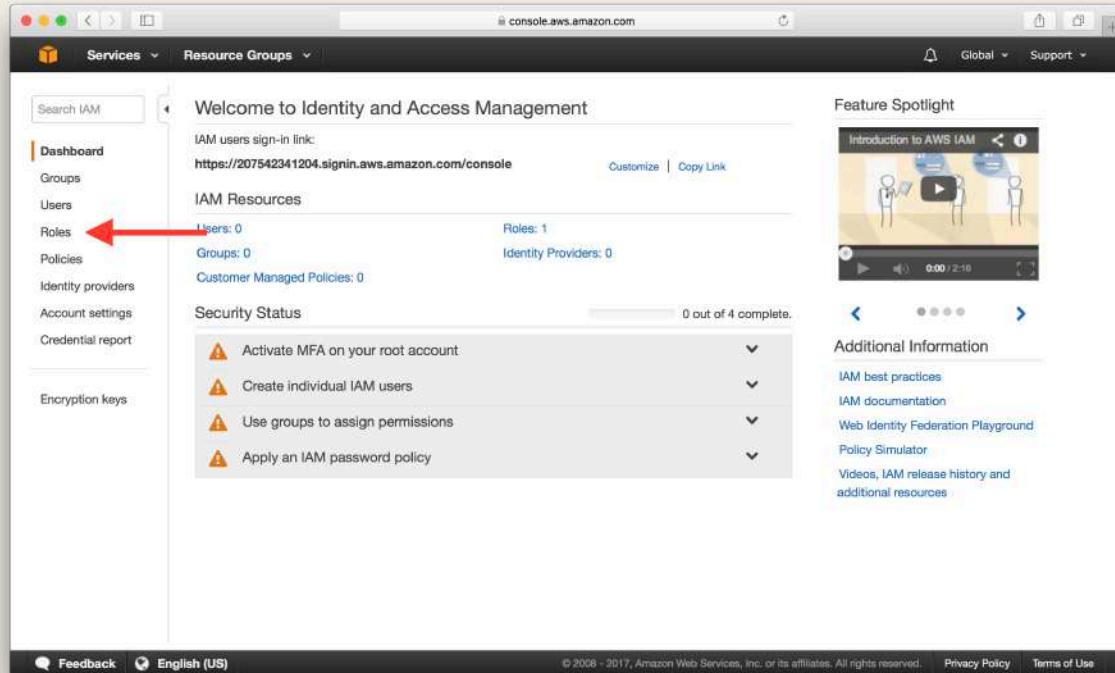
IAM Service Simulator Permission Denied

To fix this and edit the role we need to go back to the [AWS Console](#) and select IAM from the list of services.



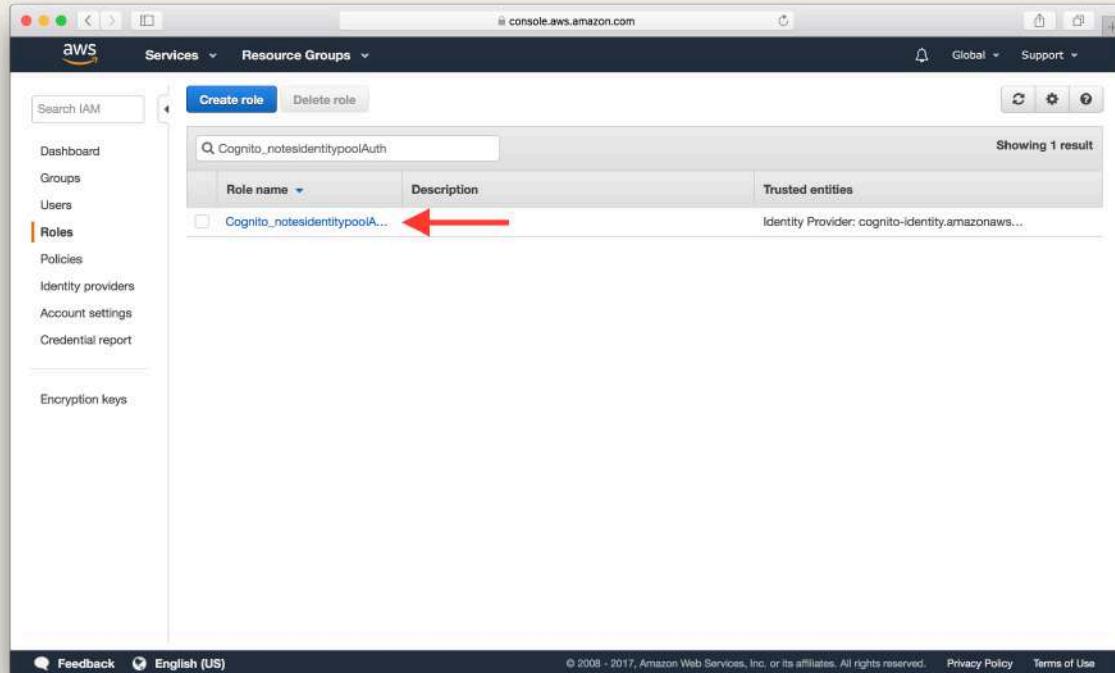
Select IAM Service Screenshot

Select **Roles** on the left menu.



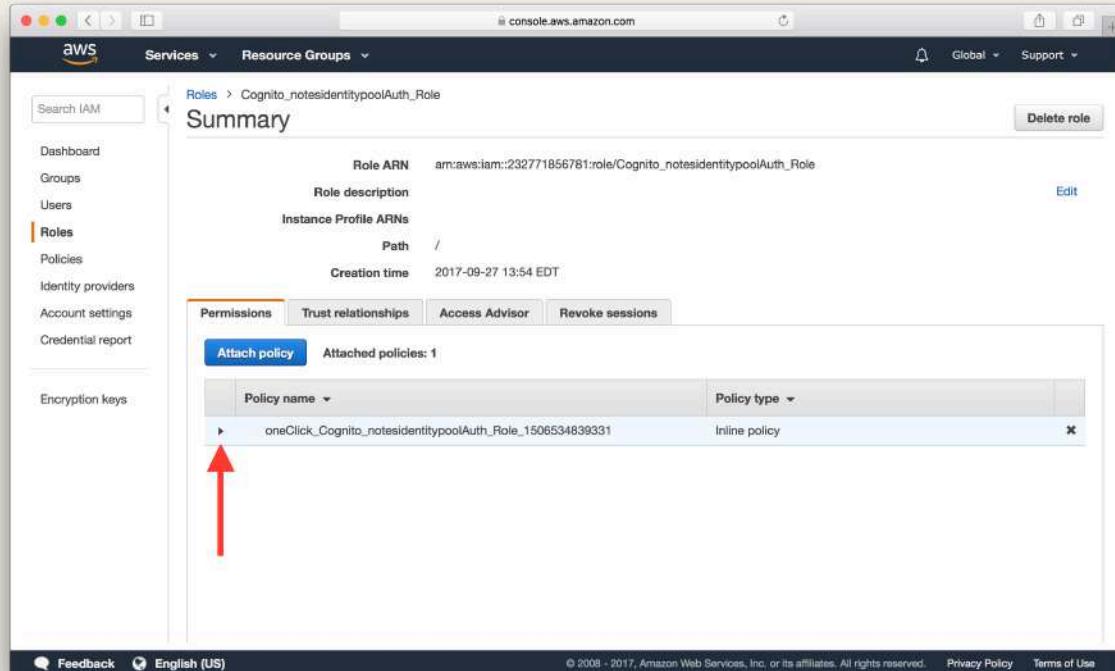
Select IAM Roles Screenshot

And select the IAM role that our Identity Pool is using. In our case it's called Cognito_notesidentitypoolAuth_Role.



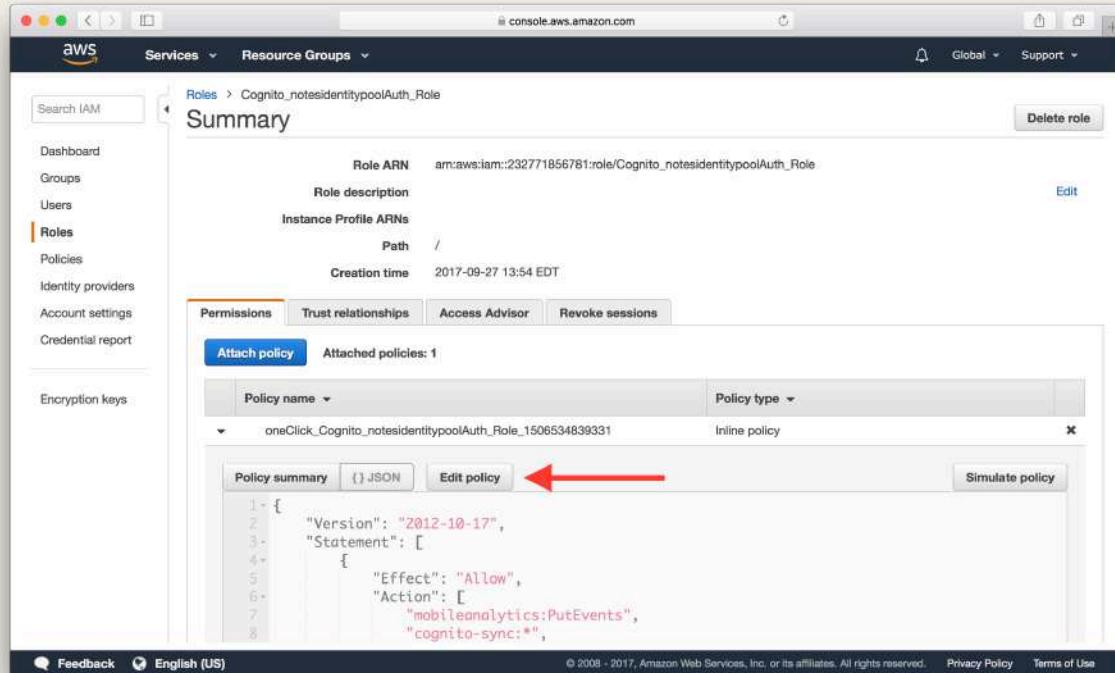
Select notes identity pool auth role Screenshot

Expand the policy under the list of policies.



Expand auth role policy Screenshot

Click **Edit policy**.

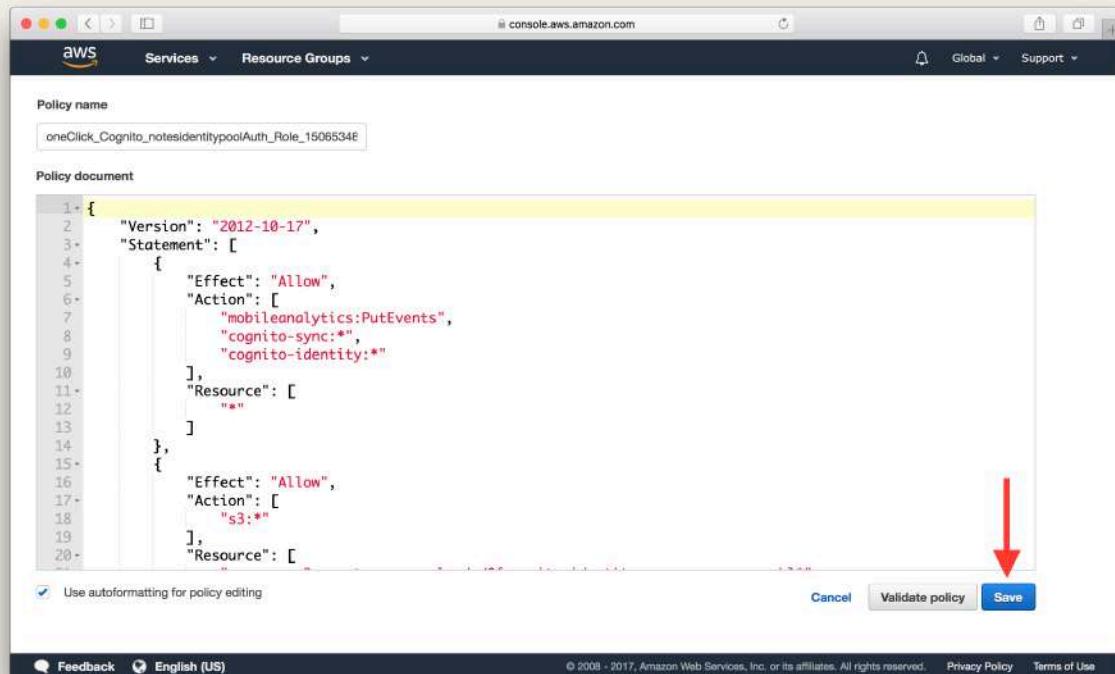


Edit auth role policy Screenshot

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.



Save auth role policy Screenshot

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.



Help and discussion

View the [comments for this chapter on our forums](#)

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](#).

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](#).



Help and discussion

View the [comments for this chapter](#) on our forums

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](#) 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](#). 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.



Help and discussion

View the [comments](#) for this chapter on our forums

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](#), 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.

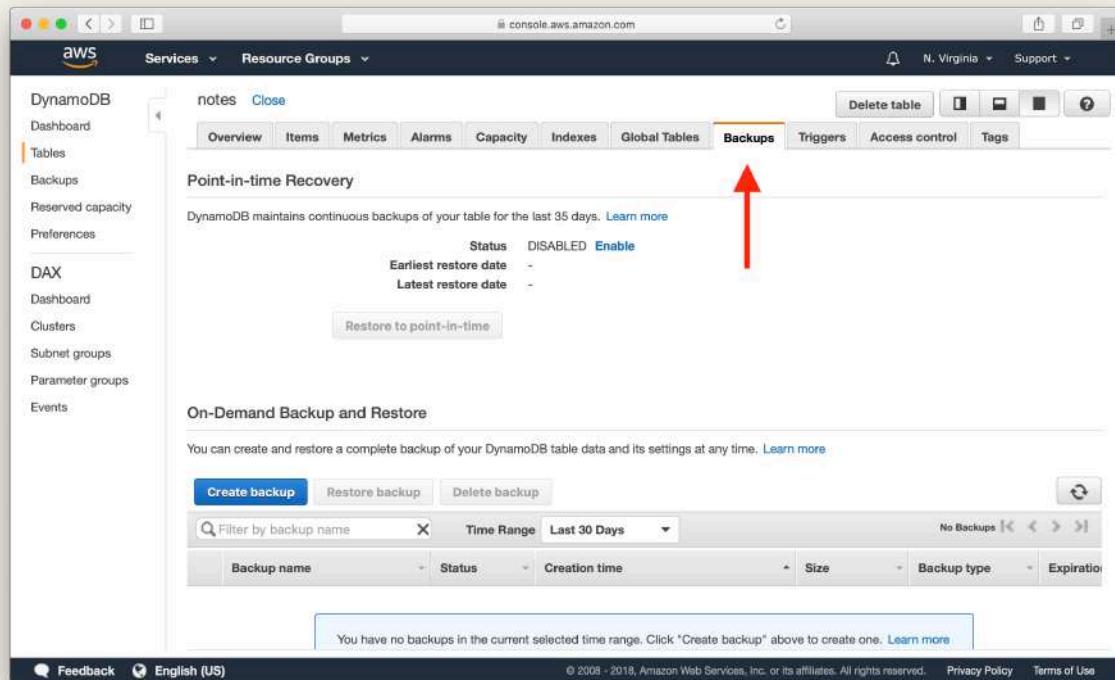
2. Point-in-time recovery

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.

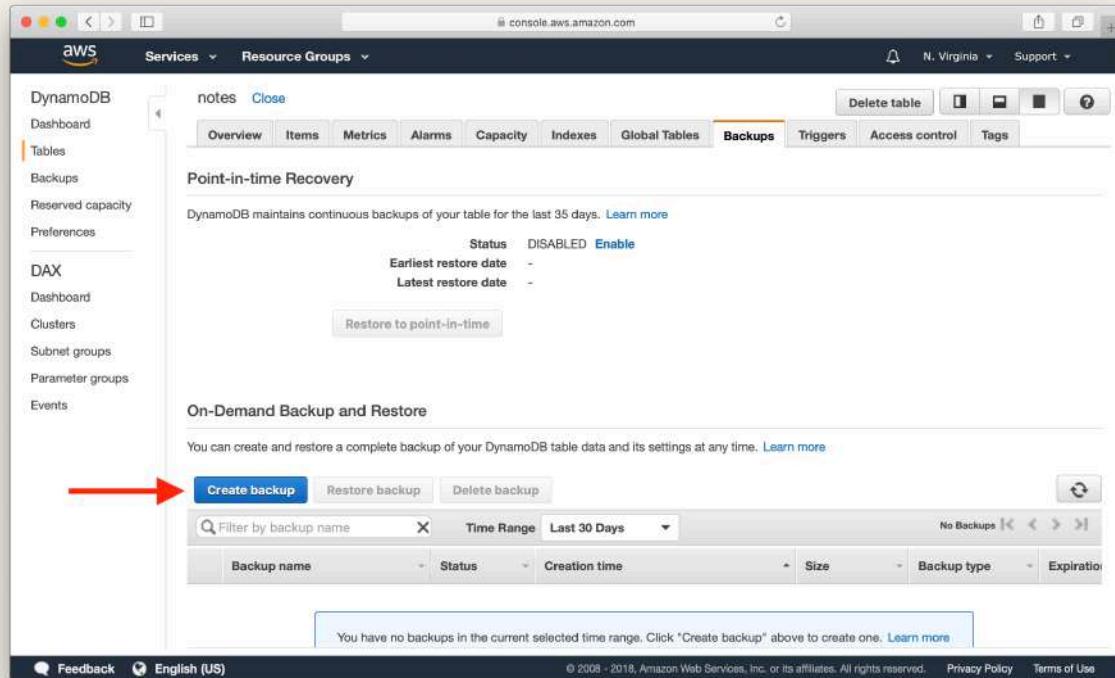
On-Demand Backup

Head over to your table and click on the **Backups** tab.



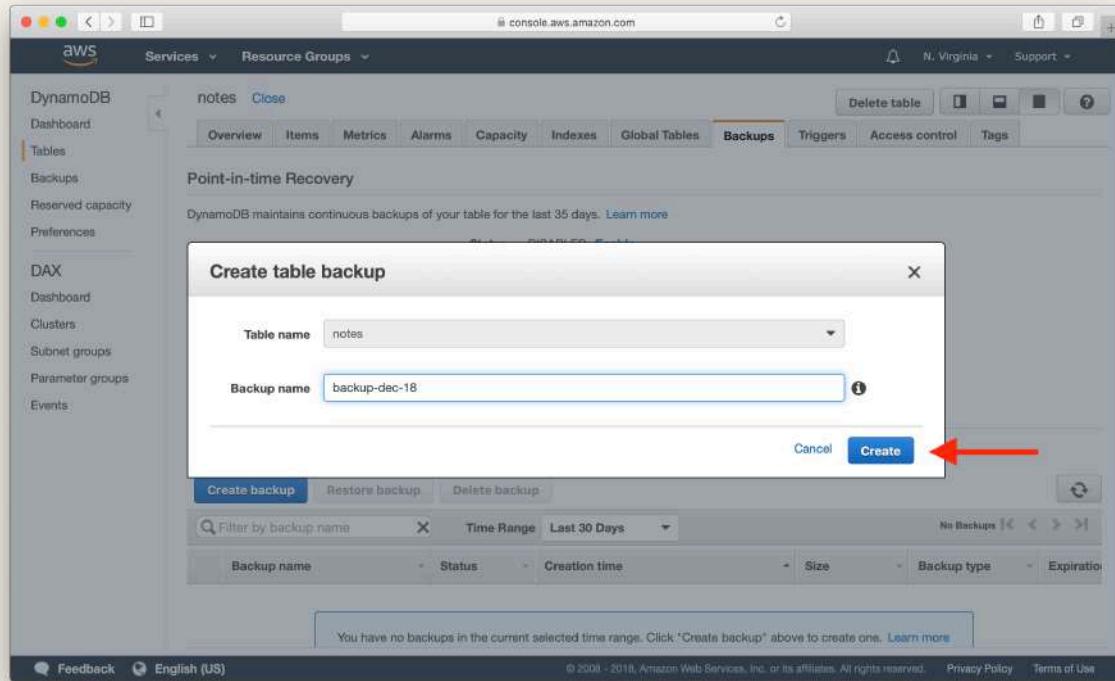
Click on Backups tab screenshot

And just hit **Create backup**.



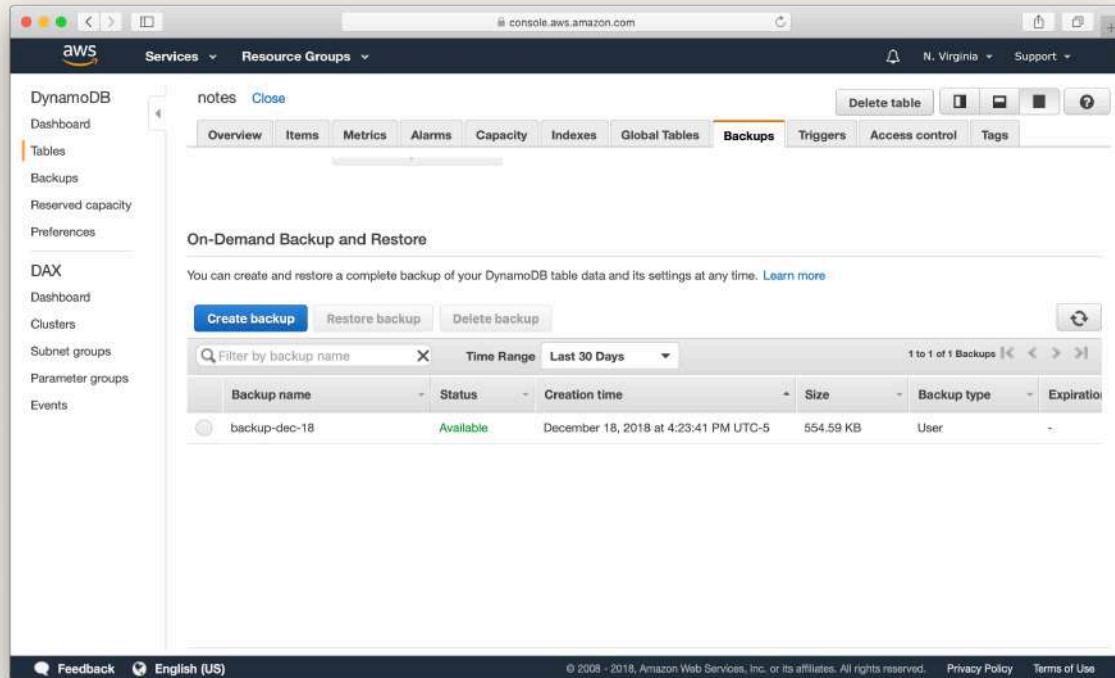
Create DynamoDB table backup screenshot

Give your backup a name and hit **Create**.



Name DynamoDB table backup screenshot

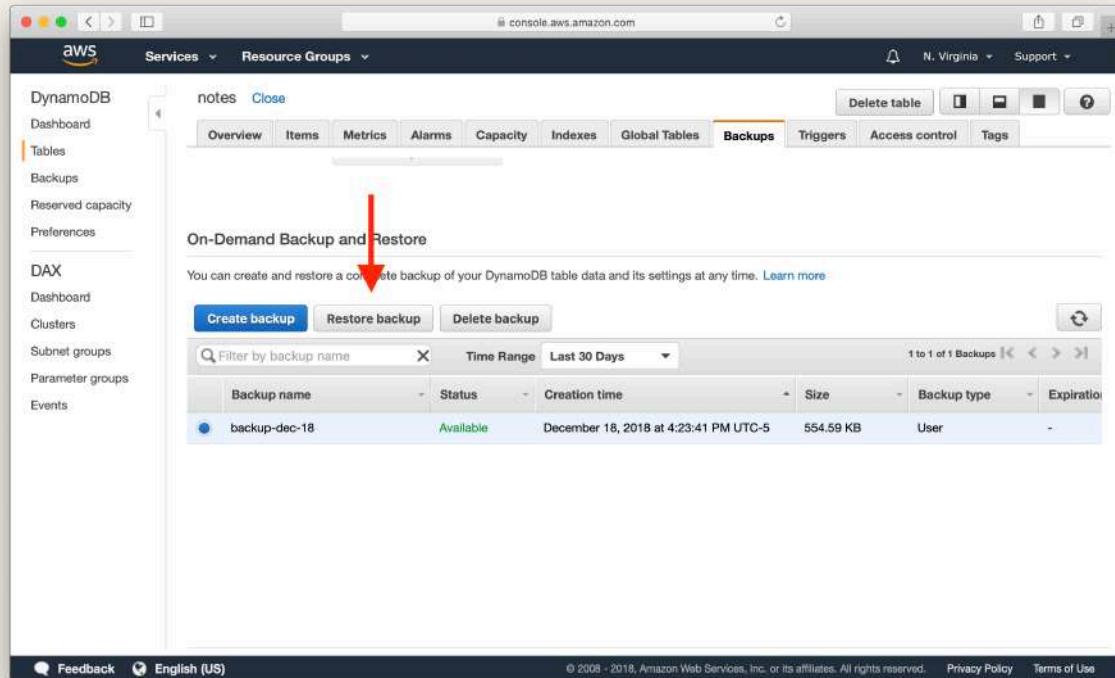
You should now be able to see your newly created backup.



New DynamoDB table backup screenshot

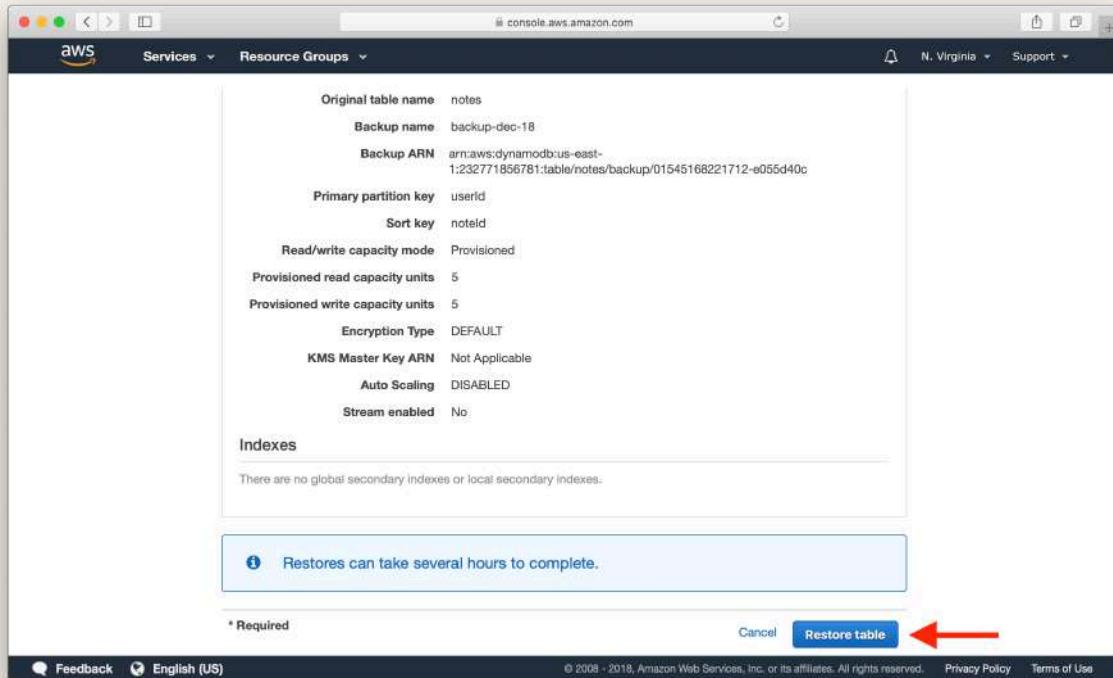
Restore Backup

Now to restore your backup, simply select the backup and hit **Restore backup**.



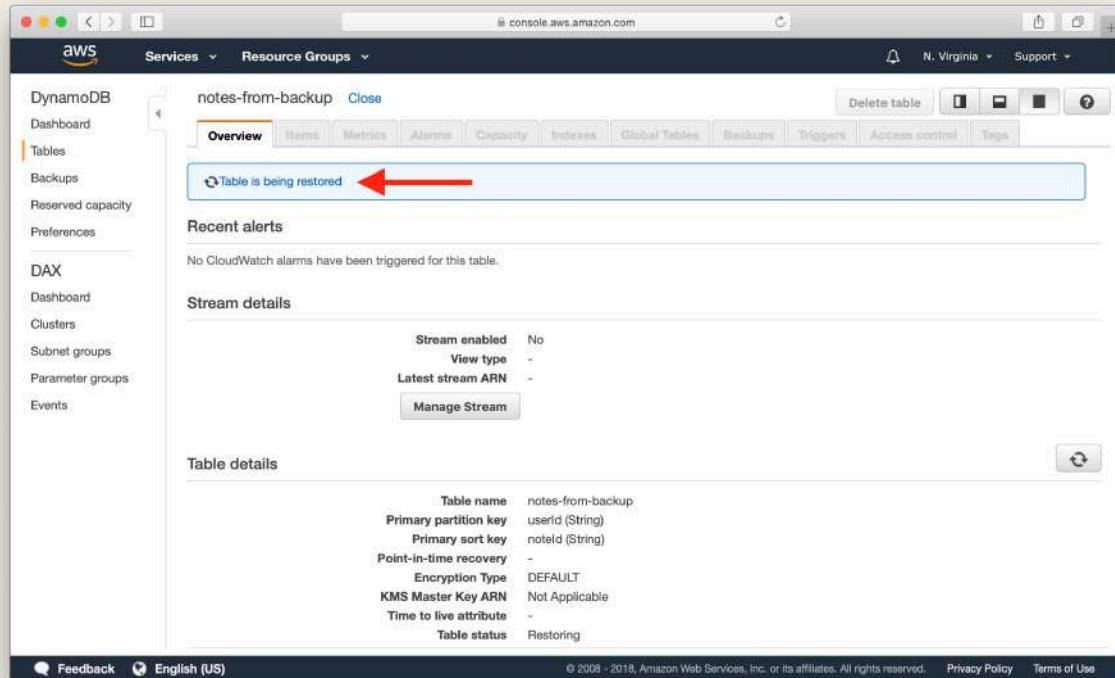
Select Restore DynamoDB table backup screenshot

Here you can type in the name of the new table you want to restore to and hit **Restore table**.



Restore DynamoDB table backup screenshot

Depending on the size of the table, this might take some time. But you should notice a new table being created from the backup.

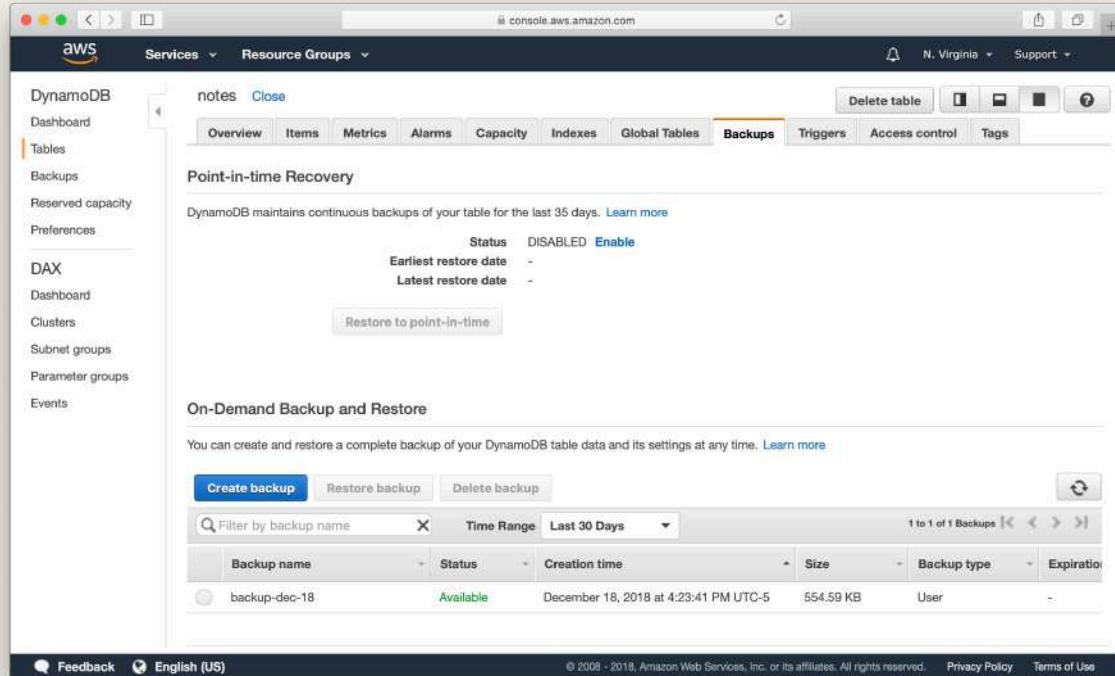


New DynamoDB table from backup screenshot

DynamoDB makes it easy to create and restore on-demand backups. You can also read more about [on-demand backups here](#).

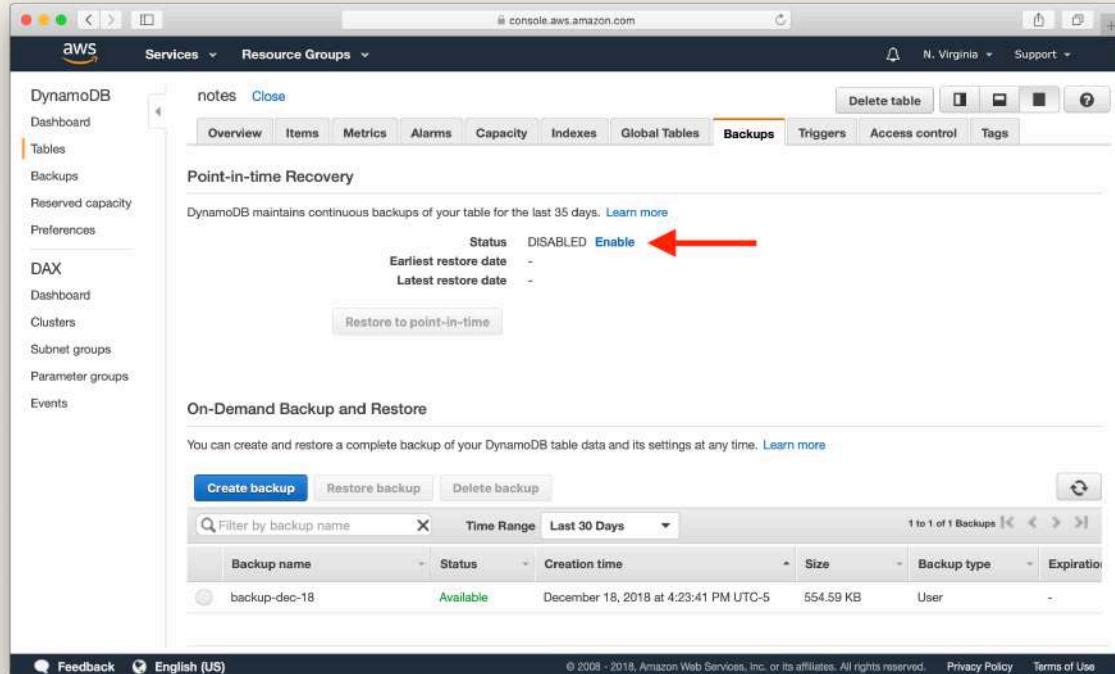
Point-in-Time Recovery

To enable Point-in-time Recovery once again head over to the **Backups** tab.



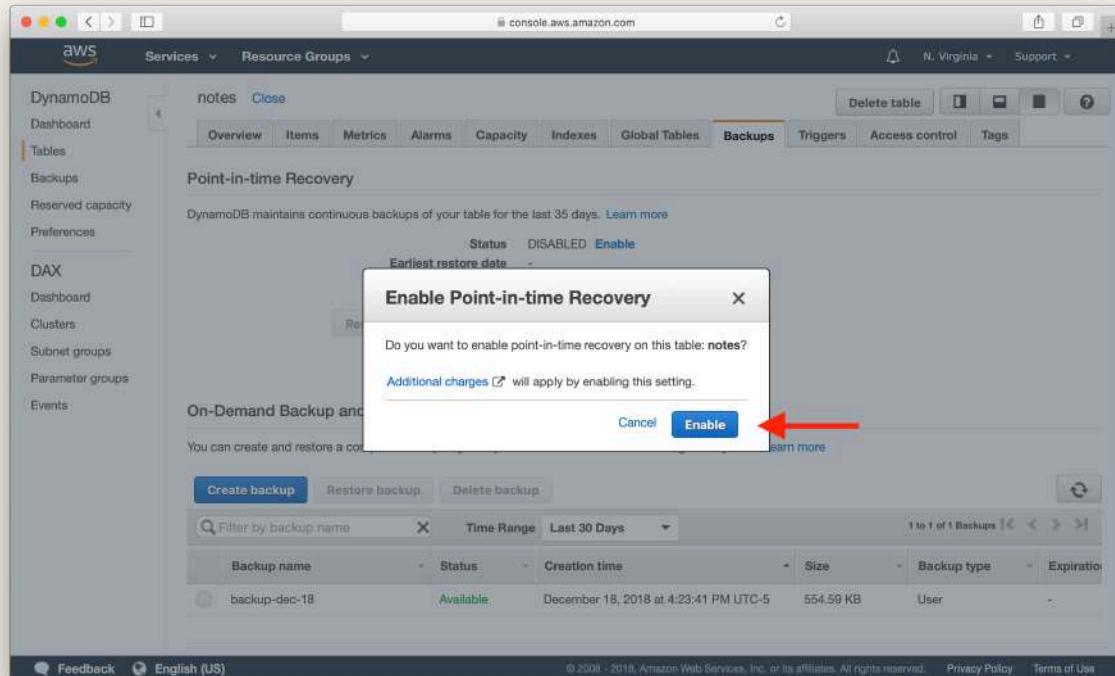
Head to Backups tab screenshot

And hit **Enable** in the Point-in-time Recovery section.



Hit Enable DynamoDB Point-in-time Recovery screenshot

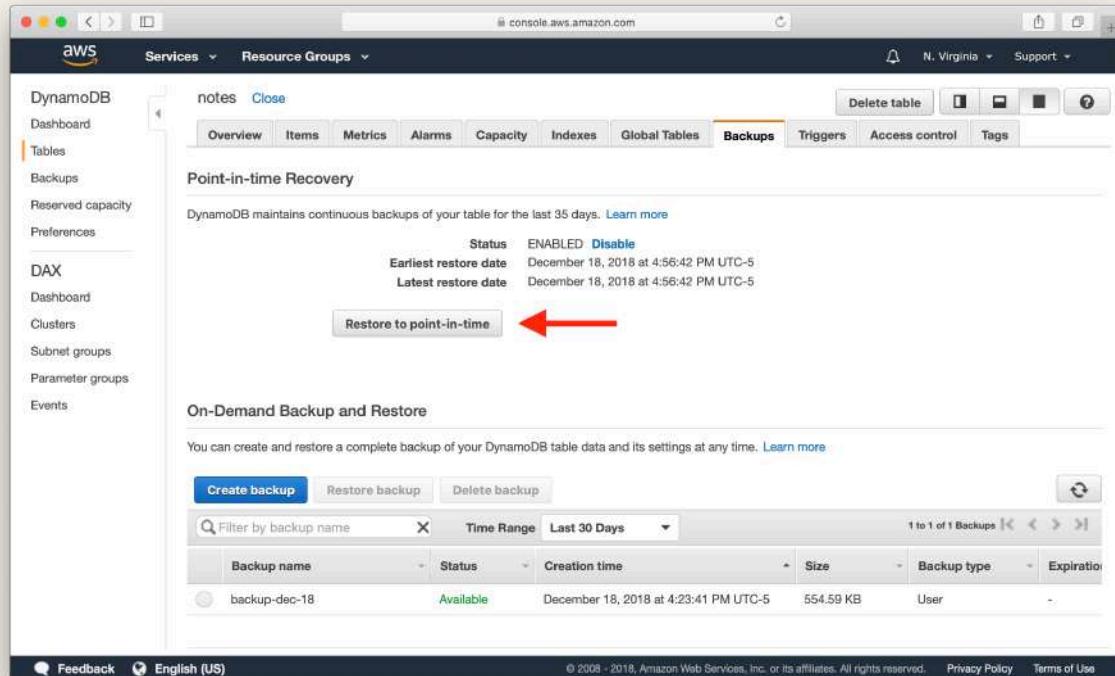
This will notify you that additional charges will apply for this setting. Click **Enable** to confirm.



Confirm Enable DynamoDB Point-in-time Recovery screenshot

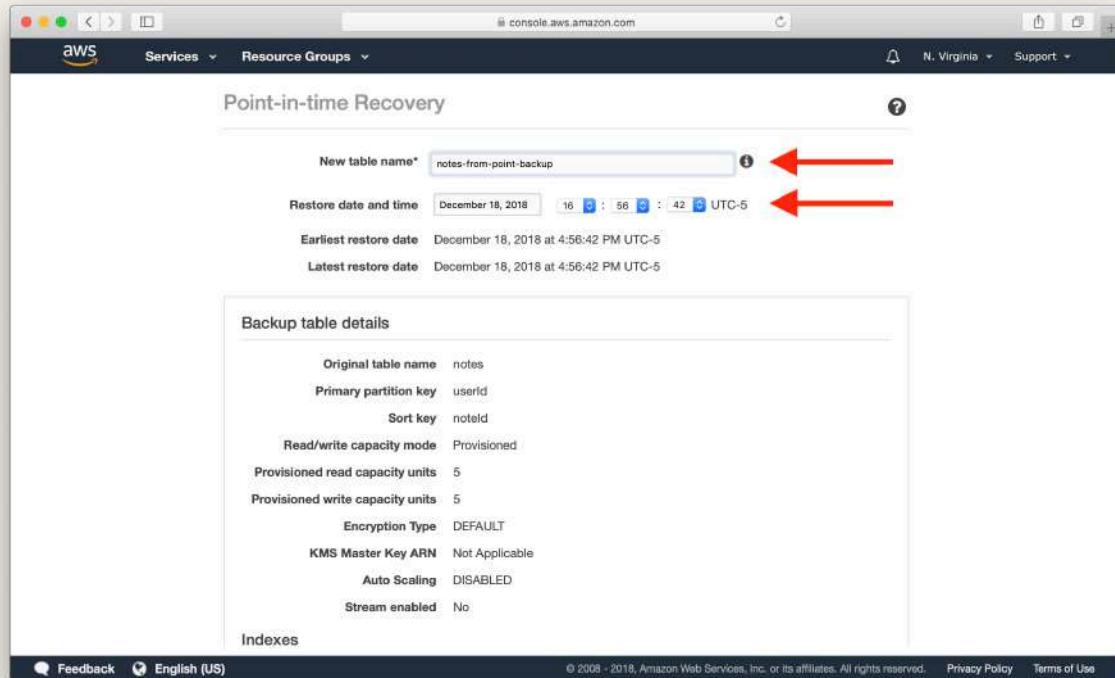
Restore to Point-in-Time

Once enabled, you can click **Restore to point-in-time** to restore to an older point.



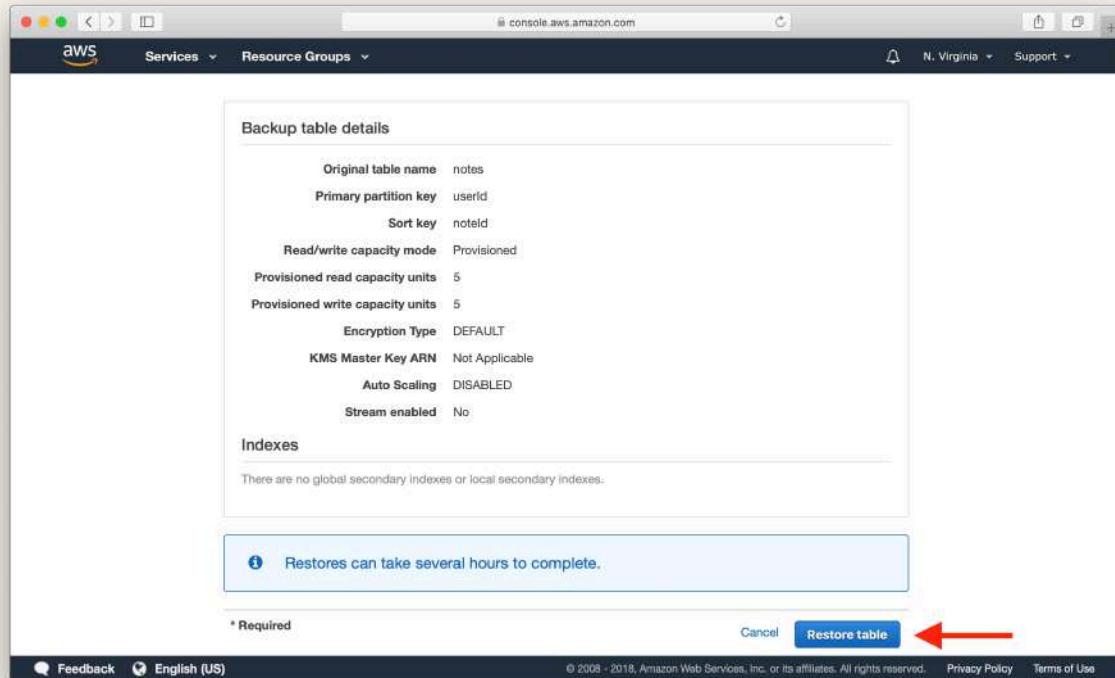
Restore DynamoDB to Point-in-time screenshot

Here you can type in the name of the new table to be restored to and select the time you want to recover to.



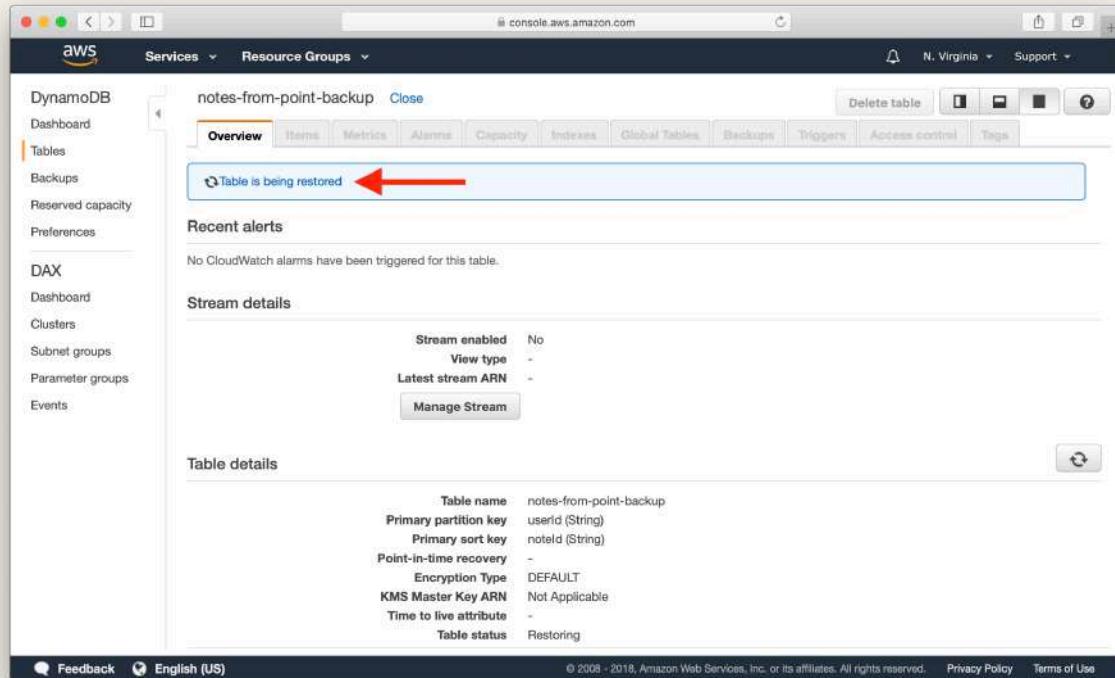
Pick Restore DynamoDB to Point-in-time screenshot

And hit **Restore table**.



Select Restore DynamoDB table to Point-in-time screenshot

You should see your new table being restored.



New restored DynamoDB table Point-in-time screenshot

You can read more about the details of [Point-in-time Recovery here](#).

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.

**Help and discussion**

View the [comments](#) for this chapter on our forums

Configure Multiple AWS Profiles

When we configured our AWS CLI in the [Configure the AWS CLI](#) 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](#) 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](#).

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](#) 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](#).

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.

**Help and discussion**

View the [comments](#) for this chapter on our forums

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](#) 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

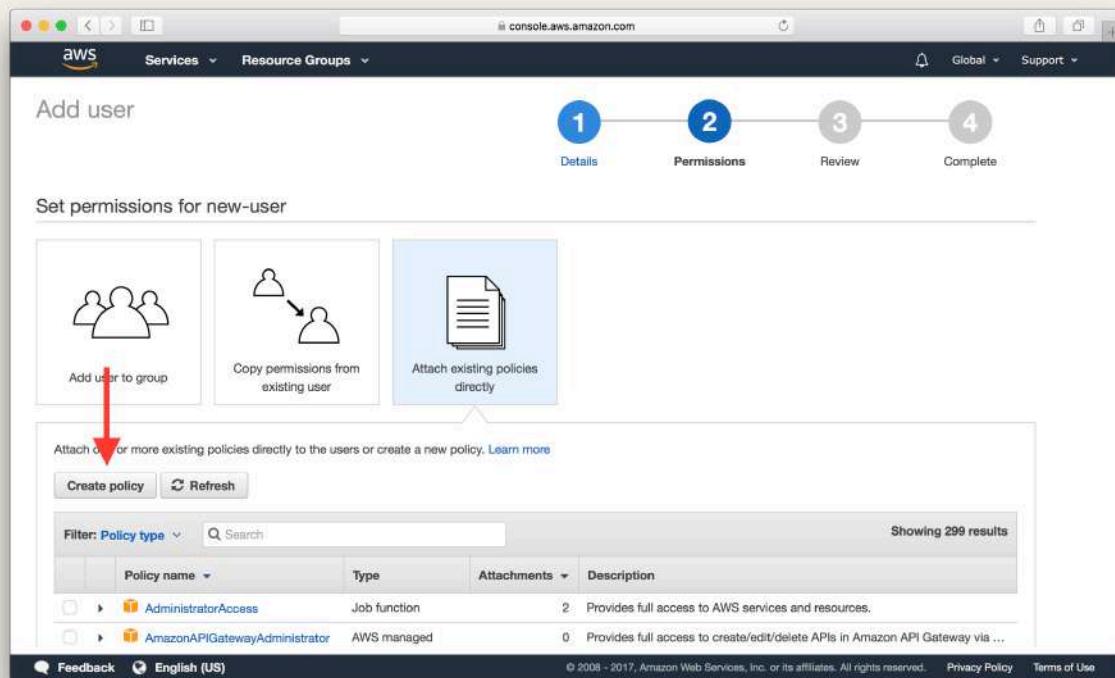
A simple IAM Policy template

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": [  
        "*"  
      ]  
    }  
  ]  
}
```

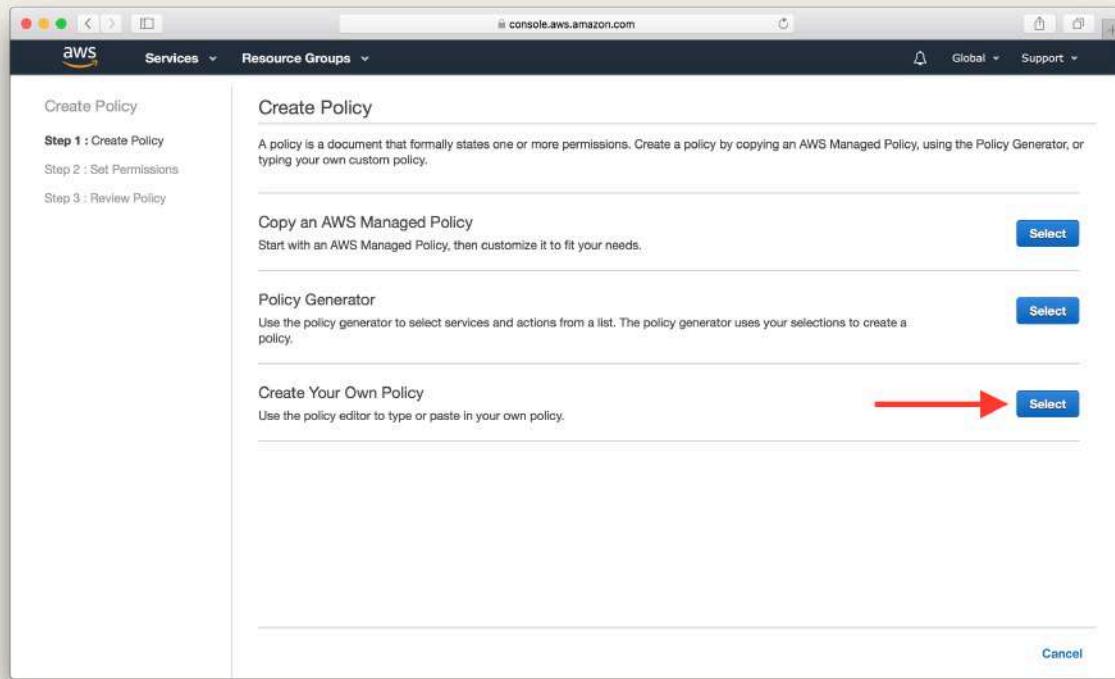
We can attach this policy to the IAM user we are creating by continuing from the **Attach existing policies directly** step in the [Create an IAM User](#) chapter.

Hit the **Create policy** button.



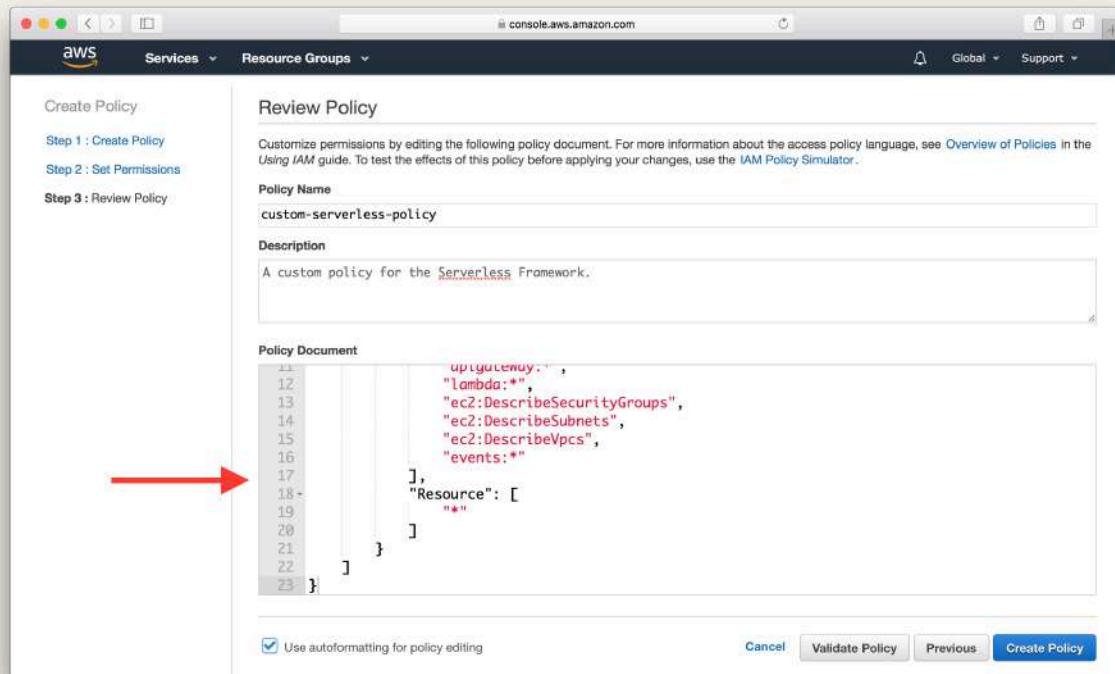
Select Create IAM Policy Screenshot

And hit **Select** in the **Create Your Own Policy** section.



Select Create your own IAM Policy Screenshot

Here pick a name for your new policy and paste the policy created above in the **Policy Document** field.



Create your own IAM Policy Screenshot

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:::<service_name>*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "s3:*
```

```
    "arn:aws:s3:::<service_name>*/*"
  ],
},
{
  "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:PATCH",
    "apigateway:POST",
    "apigateway:PUT",
    "apigateway:DELETE"
  ],
  "Resource": [
    "arn:aws:apigateway:<region>:::/restapis"
  ],
},
{
  "Effect": "Allow",
  "Action": [
    "apigateway:GET",
    "apigateway:PATCH",
    "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](#) 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](#) required for a Serverless project.



Help and discussion

View the [comments for this chapter on our forums](#)

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](#)), 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](#) 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.cognitoAuthenticationProvider;
  // Cognito authentication provider looks like:
  // cognito-idp.us-east-1.amazonaws.com/us-east-1xxxxxxxxx,cognito-idp.us-
  ↵ east-1.amazonaws.com/us-east-1aaaaaaaa:CognitoSignIn:qqqqqqqq-1111-
  ↵ 2222-3333-rrrrrrrrrrrr
  // Where us-east-1aaaaaaaa is the User Pool id
  // And qqqqqqqq-1111-2222-3333-rrrrrrrrrrrr 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-1xxxxxxxxx,cognito-idp.us-east-
  ↵ 1.amazonaws.com/us-east-1aaaaaaaa:CognitoSignIn:qqqqqqqq-1111-2222-
  ↵ 3333-rrrrrrrrrrrr

```

Where `us-east-1aaaaaaaa` is the User Pool id and `qqqqqqqq-1111-2222-3333-rrrrrrrrrrrr` 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.



Help and discussion

[View the comments for this chapter on our forums](#)

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](#). You can use:

- [Generated API Gateway SDK](#)
- [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](#) chapter.

However if you are looking to simply connect to API Gateway using the AWS JS SDK, we've created a standalone [sigV4Client.js](#) 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 `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](#).

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` 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.

→(<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.



Help and discussion

View the [comments for this chapter on our forums](#)

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](#). If you are using TypeScript, we have a starter for you as well, [Serverless TypeScript Starter](#). We also have a Python version called [Serverless Python Starter](#). Our starter projects also work really well with [Seed](#); a fully-configured CI/CD pipeline for Serverless Framework.

Serverless Node.js Starter uses the `serverless-bundle` plugin (an extension of the `serverless-webpack` plugin) and the `serverless-offline` plugin. It supports:

- **Using ES6 or TypeScript in your Lambda function code**
- **Generating optimized packages with Webpack**
- **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>.

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
- 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](#)

```
$ 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](#).

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`.
2. Add environment variables for your local stage to `.env`.

3. Uncomment environment: block in the `serverless.yml` and reference the environment variable as `${env:MY_ENV_VAR}`. Where `MY_ENV_VAR` is added to your `.env` file.
4. Make sure to not commit your `.env`.

So give it a try and send us an [email](#) if you have any questions or open a [new issue](#) if you've found a bug.



Help and discussion

View the [comments for this chapter on our forums](#)

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](#) 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](#) plugin to apply [Webpack's tree shaking algorithm](#) to only include the relevant bits of code needed for your Lambda function.

ES6 and TypeScript

AWS Lambda supports Node.js 10.x and 12.x. However most modern JavaScript projects rely on ES6 features (like `import/export`) and TypeScript. To support ES6 and TypeScript, you can use [Babel](#) and [TypeScript](#) to transpile your Lambda functions.

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. This can be really hard to do across multiple projects.

We created a plugin to solve all of these issues.

Only One Dependency

Enter [serverless-bundle](#); a plugin that will generate an optimized Lambda function package for your ES6 or TypeScript Lambda functions without you having to manage any Webpack, Babel, or ESLint configs!

- `"eslint"`
 - `"webpack"`
 - `"ts-loader"`
 - `"typescript"`
 - `"css-loader"`
 - `"graphql-tag"`
 - `"@babel/core"`
 - `"babel-eslint"`
 - `"babel-loader"`
 - `"eslint-loader"`
 - `"@babel/runtime"`
 - `"@babel/preset-env"`
 - `"serverless-webpack"`
 - `"source-map-support"`
 - `"webpack-node-externals"`
 - `"eslint-config-strongloop"`
 - `"tsconfig-paths-webpack-plugin"`
 - `"fork-ts-checker-webpack-plugin"`
-

```
- "@babel/plugin-transform-runtime"
- "babel-plugin-source-map-support"

+ "serverless-bundle"
```

serverless-bundle has a few key advantages:

- Only one dependency
- Supports ES6 and TypeScript
- Generates optimized packages
- Linting Lambda functions using [ESLint](#)
- Supports transpiling unit tests with [babel-jest](#)
- Source map support for proper error messages

Getting Started

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](#).

Our ever popular [Serverless Node.js Starter](#) has now been updated to use the serverless-bundle plugin. And we also have a TypeScript version of our starter – [Serverless TypeScript Starter](#)



Help and discussion

View the [comments](#) for this chapter on our forums

Using Lerna and Yarn Workspaces with Serverless

In the [Organizing Serverless Projects](#) chapter we covered the standard monorepo setup. This included [how to share code between your services](#) and [how to deploy a Serverless app with interdependent services](#).

This setup works pretty well but as your team and project grows, you run into a new issue. You have some common code libraries that are used across multiple services. An update to these libraries would redeploy all your services. If your services were managed by separate folks on your team or by separate teams, this poses a problem. For any change made to the common code, would require all the other folks on your team to test or update their services.

Here it makes sense to manage your common code libraries as packages. So your services could potentially be using different version of the same package. This will allow your team to update to the newer version of the package when it works best for them. Avoiding the scenario where a small change to some common code breaks all the services that depend on it.

However, managing these packages in the same repo can be really challenging. To tackle this issue we are going to use:

- [Yarn Workspaces](#)

This optimizes our repo by hoisting all of our separate node_modules/ to the root level. So that a single `yarn install` command installs the NPM modules for all our services and packages.

- [Lerna](#)

This helps us manage our packages, publish them, and keeps track of the dependencies between them.

Lerna and Yarn Workspaces together helps create a monorepo setup that allows our Serverless project to scale as it grows.

To help get you started with this, we created a starter project — [Serverless Lerna + Yarn Workspaces Monorepo Starter](#)

- Designed to scale for larger projects
- Maintains internal dependencies as packages
- Uses Lerna to figure out which services have been updated
- Supports publishing dependencies as private NPM packages
- Uses `serverless-bundle` to generate optimized Lambda packages
- Uses Yarn Workspaces to hoist packages to the root `node_modules` / directory

This will help get you started with this setup. But if you are not familiar with Lerna or Yarn Workspaces, make sure to check out their docs.

Installation

To create a new Serverless project

```
$ serverless install --url
↳ https://github.com/AnomalyInnovations/serverless-lerna-yarn-starter
↳ --name my-project
```

Enter the new directory

```
$ cd my-project
```

Install NPM packages for the entire project

```
$ yarn
```

How It Works

The directory structure roughly looks like:

```
package.json
/libs
/packages
```

```
/sample-package
  index.js
  package.json
/services
  /service1
    handler.js
    package.json
    serverless.yml
  /service2
    handler.js
    package.json
    serverless.yml
```

This repo is split into 3 directories. Each with a different purpose:

- packages

These are internal packages that are used in our services. Each contains a `package.json` and can be optionally published to NPM. Any changes to a package should only deploy the service that depends on it.

- services

These are Serverless services that are deployed. Has a `package.json` and `serverless.yml`. There are two sample services.

1. `service1`: Depends on the `sample-package`. This means that if it changes, we want to deploy `service1`.
2. `service2`: Does not depend on any internal packages.

More on deployments below.

- libs

Any common code that you might not want to maintain as a package. Does NOT have a `package.json`. Any changes here should redeploy all our services.

The `packages/` and `services/` directories are Yarn Workspaces.

Services

The Serverless services are meant to be managed on their own. Each service is based on our [Serverless Node.js Starter](#). It uses the `serverless-bundle` plugin (based on [Webpack](#)) to create

optimized Lambda packages.

This is good for keeping your Lambda packages small. But it also ensures that you can have Yarn hoist all your NPM packages to the project root. Without Webpack, you'll need to disable hoisting since Serverless Framework does not package the dependencies of a service correctly on its own.

Install an NPM package inside a service.

```
$ yarn add some-npm-package
```

Run a function locally.

```
$ serverless invoke local -f get
```

Run tests in a service.

```
$ yarn test
```

Deploy the service.

```
$ serverless deploy
```

Deploy a single function.

```
$ serverless deploy function -f get
```

To add a new service.

```
$ cd services/
$ serverless install --url
  ↳ https://github.com/AnomalyInnovations/serverless-nodejs-starter --name
  ↳ new-service
$ cd new-service
$ yarn
```

Packages

Since each package has its own package.json, you can manage it just like you would any other NPM package.

To add a new package.

```
$ mkdir packages/new-package  
$ yarn init
```

Packages can also be optionally published to NPM.

Libs

If you need to add any other common code in your repo that won't be maintained as a package, add it to the `libs/` directory. It does not contain a package.json. This means that you'll need to install any NPM packages as dependencies in the root.

To install an NPM package at the root.

```
$ yarn add -W some-npm-package
```

Deployment

We want to ensure that only the services that have been updated get deployed. This means that, if a change is made to:

- services

Only the service that has been changed should be deployed. For ex, if you change any code in `service1`, then `service2` should not be deployed.

- packages

If a package is changed, then only the service that depends on this package should be deployed. For ex, if `sample-package` is changed, then `service1` should be deployed.

- libs

If any of the libs are changed, then all services will get deployed.

Deployment Algorithm

To implement the above, use the following algorithm in your CI:

1. Run `lerna ls --since ${prevCommitSHA} -all` to list all packages that have changed since the last successful deployment. If this list includes one of the services, then deploy it.
2. Run `git diff --name-only ${prevCommitSHA} ${currentCommitSHA}` to get a list of all the updated files. If they don't belong to any of your Lerna packages (`lerna ls -all`), deploy all the services.
3. Otherwise skip the deployment.

Deploying Through Seed

[Seed](#) supports deploying Serverless monorepo projects that use Lerna and Yarn Workspaces. To enable it, add the following to the `seed.yml` in your repo root:

```
check_code_change: lerna
```

To test this:

Add the App

1. Fork this repo and add it to [your Seed account](#).
2. Add both of the services.
3. Deploy your app once.

Update a Service

- Make a change in `services/service2/handler.js` and `git push`.
- Notice that `service2` has been deployed while `service1` was skipped.

Update a Package

- Make a change in `packages/sample-package/index.js` and `git push`.
- Notice that `service1` should be deployed while `service2` will have been skipped.

Update a Lib

- Finally, make a change in `libs/index.js` and `git push`.
- Both `service1` and `service2` should've been deployed.

This starter should give you a great template to build your next monorepo Serverless project. So give it a try and let us know what you think.

**Help and discussion**

View the [comments](#) for this chapter on our forums

React

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](#). 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

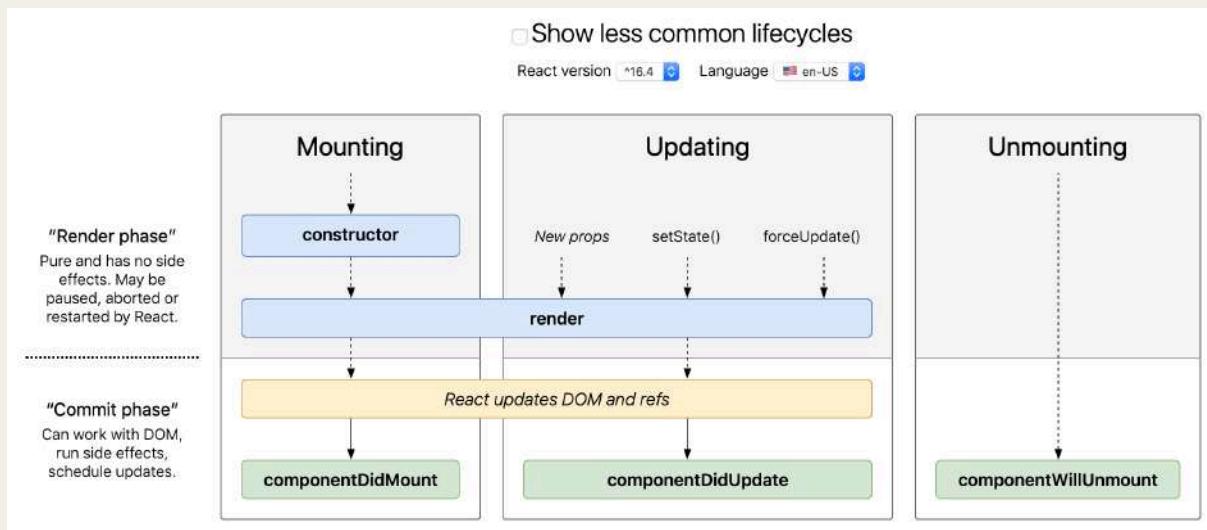
Note that, this chapter does not cover specific React Hooks in detail, [the React Docs are a great place for that](#).

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.



React Class lifecycle flowchart

You can [view the above in detail here](#).

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 with 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](#).
- 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 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 on 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 only on 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](#).

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 every 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 have to 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](#), Function components have a more straightforward 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](#), title “[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 transition 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](#) 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 happens thanks to [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.



Help and discussion

View the [comments for this chapter on our forums](#)

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](#).

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) (<https://twitter.com/wSokra/status/866703557323632640>) and [@dan_abramov](https://twitter.com/dan_abramov/status/866646657437491201) (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.

```
ServerlessStackDemoClient — ServerlessStackDemoClient — -bash — 90x25
> react-scripts build

Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

 314.66 KB  build/static/js/main.92eaa171.js
  2.98 KB   build/static/js/0.501c9213.chunk.js
  2.67 KB   build/static/js/2.69156576.chunk.js
  2.56 KB   build/static/js/1.424b6dfc.chunk.js
  2.46 KB   build/static/js/3.919af17b.chunk.js
  2.02 KB   build/static/js/4.e243d17a.chunk.js
  547 B     build/static/js/5.de81e9a1.chunk.js
  374 B     build/static/css/main.dc12301d.css

The project was built assuming it is hosted at the server root.
To override this, specify the homepage in your package.json.
For example, add this to build it for GitHub Pages:

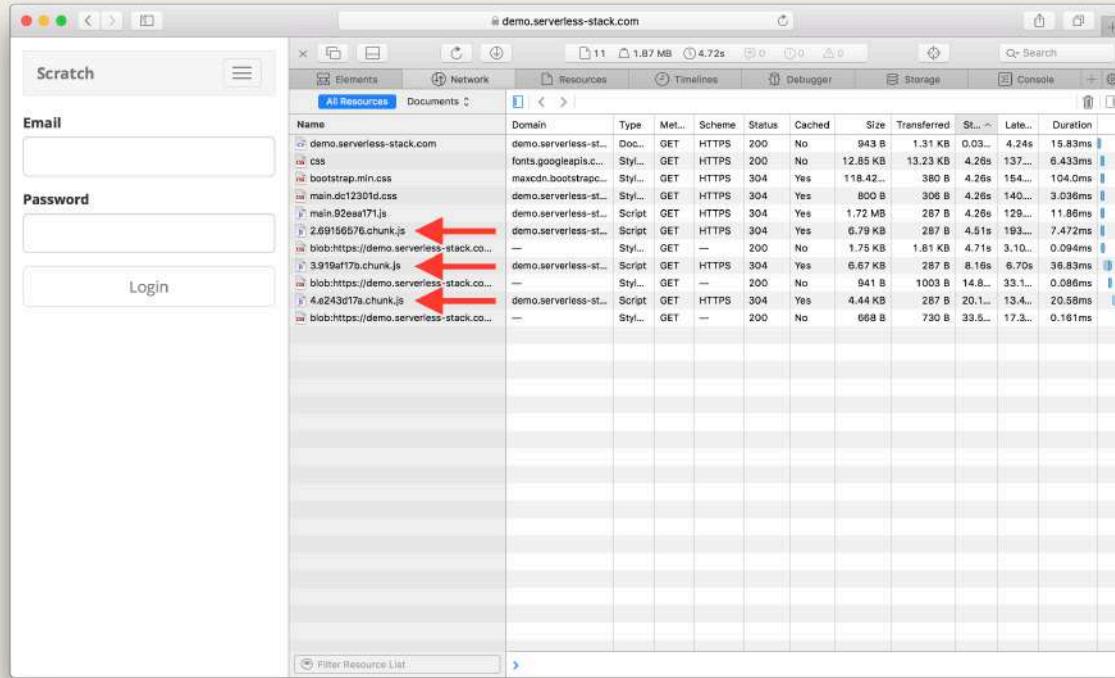
  "homepage" : "http://myname.github.io/myapp",

The build folder is ready to be deployed.
You may serve it with a static server:
```

Create React App Code Splitting build screenshot

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](#).



Create React App loading Code Splitting screenshot

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](#).

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` has. And have fun code splitting!



Help and discussion

View the [comments for this chapter on our forums](#)



For reference, here is the code we are using

Frontend Source: [code-splitting-in-create-react-app](#)

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](#).

Custom Environment Variables

[Create React App](#) 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](#) chapter. And `YOUR_CF_DISTRIBUTION_ID` and `YOUR_WWW_CF_DISTRIBUTION_ID` are the CloudFront Distributions for the `apex` and `www` 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](#).



Help and discussion

View the [comments for this chapter on our forums](#)

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](#) to host our assets, [CloudFront](#) to serve it, [Route 53](#) to manage our domain, and [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.



Help and discussion

View the [comments for this chapter](#) on our forums

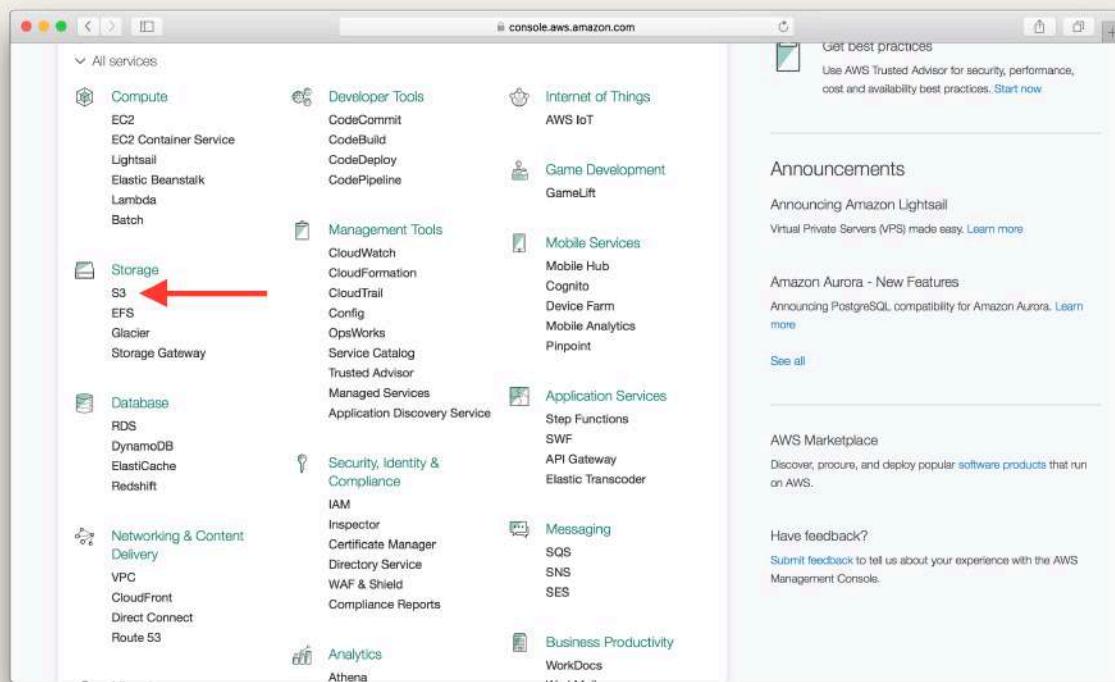
Create an S3 Bucket

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.

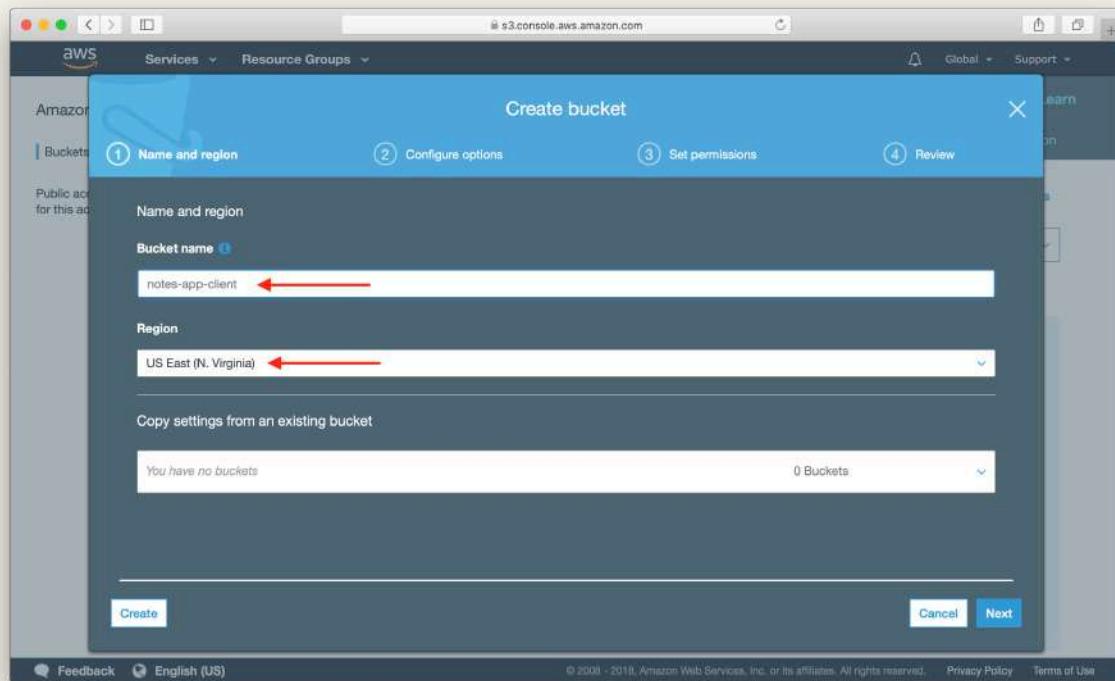
Create the Bucket

First, log in to your [AWS Console](#) and select S3 from the list of services.



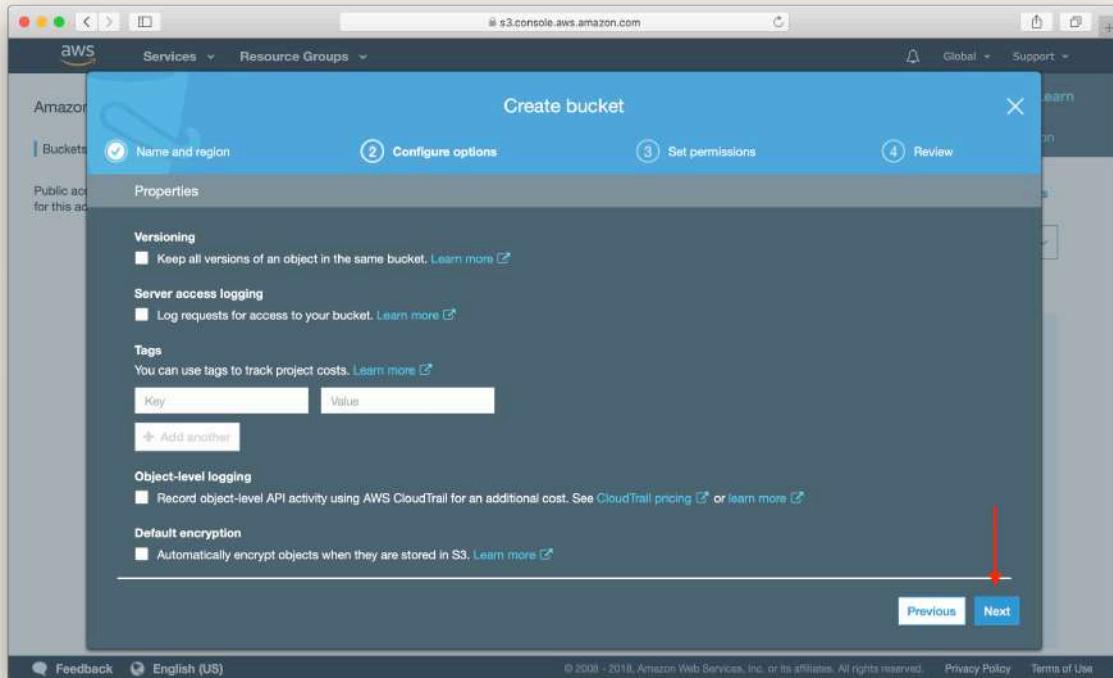
Select S3 Service screenshot

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.



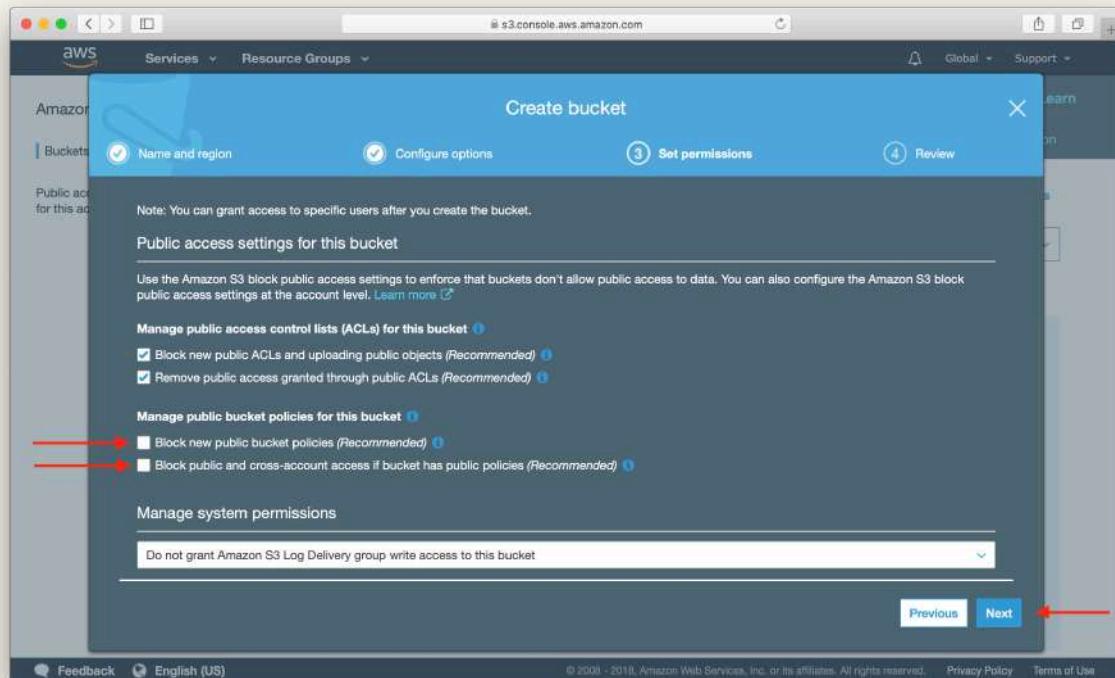
Create S3 static website Bucket screenshot

Click **Next** through the configure options step.



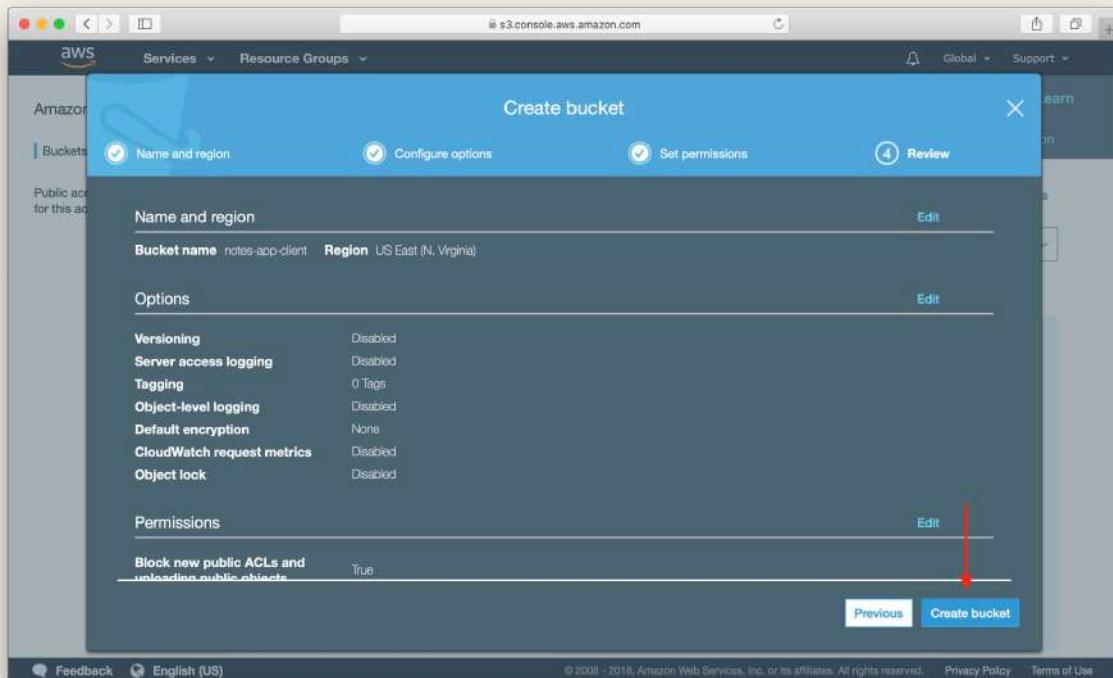
Create S3 static website Bucket next configure options screenshot

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.



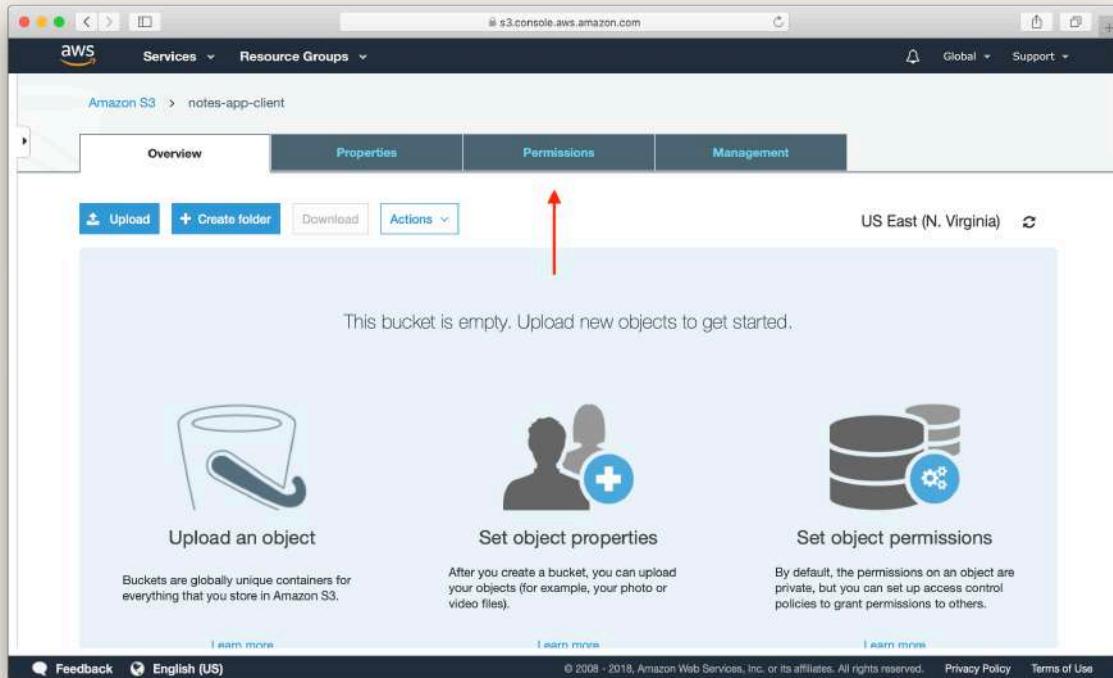
Create S3 static website Bucket next permissions screenshot

Click **Create bucket** on the review page to create the bucket.



Create S3 static website Bucket next review screenshot

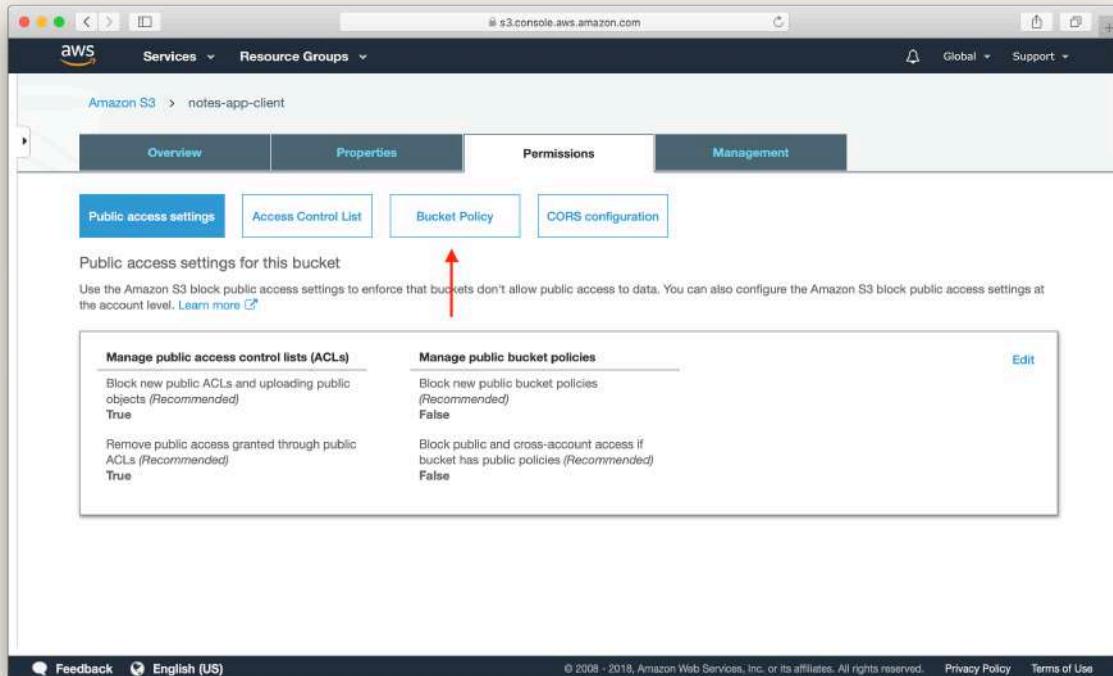
Now click on your newly created bucket from the list and navigate to its permissions panel by clicking **Permissions**.



Select AWS S3 static website Bucket permissions screenshot

Add Permissions

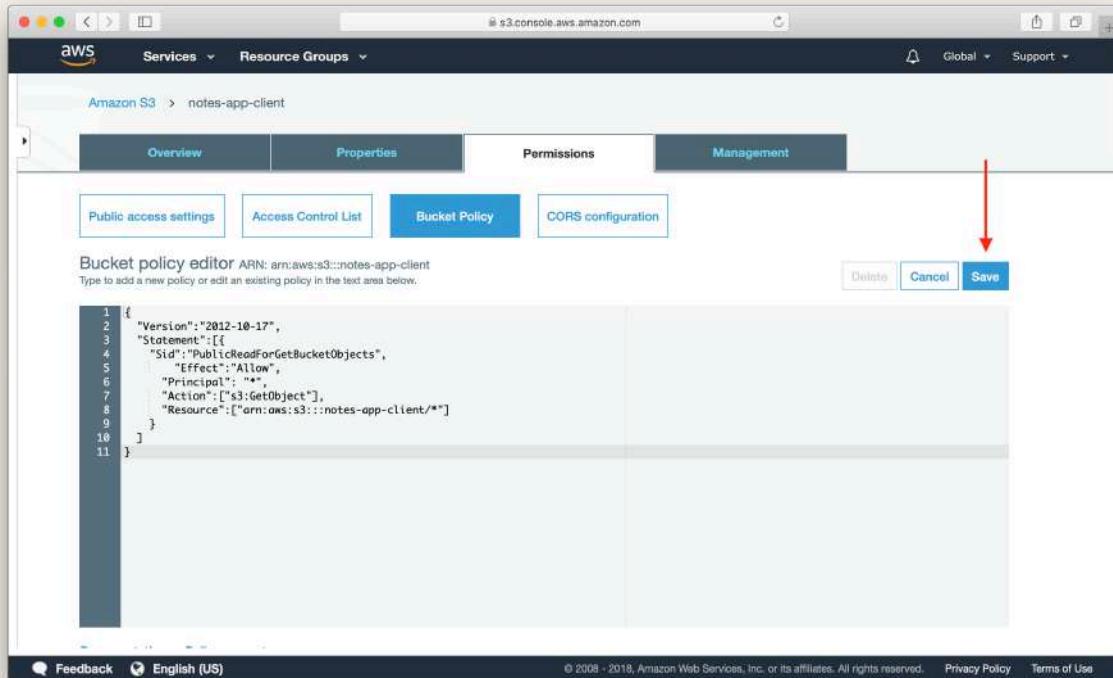
Buckets by default are not publicly accessible, so we need to change the S3 Bucket Permission. Select the **Bucket Policy** from the permissions panel.



Add AWS S3 Bucket permission screenshot

◆ 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/*"]  
  }]  
}
```

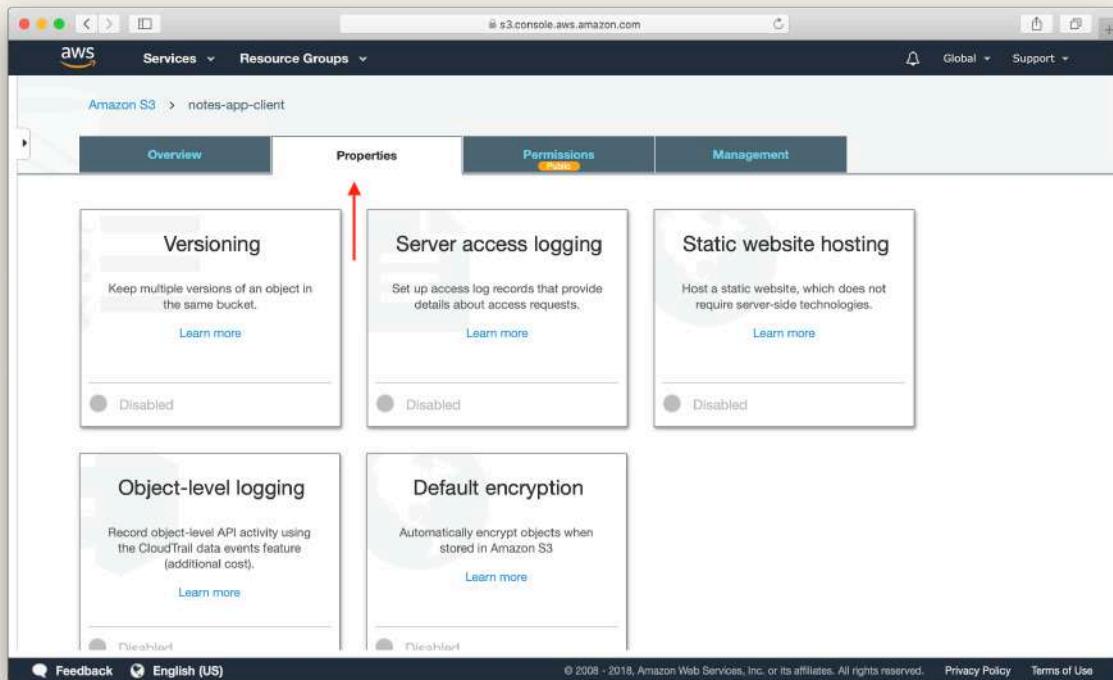


Save bucket policy screenshot

And hit **Save**.

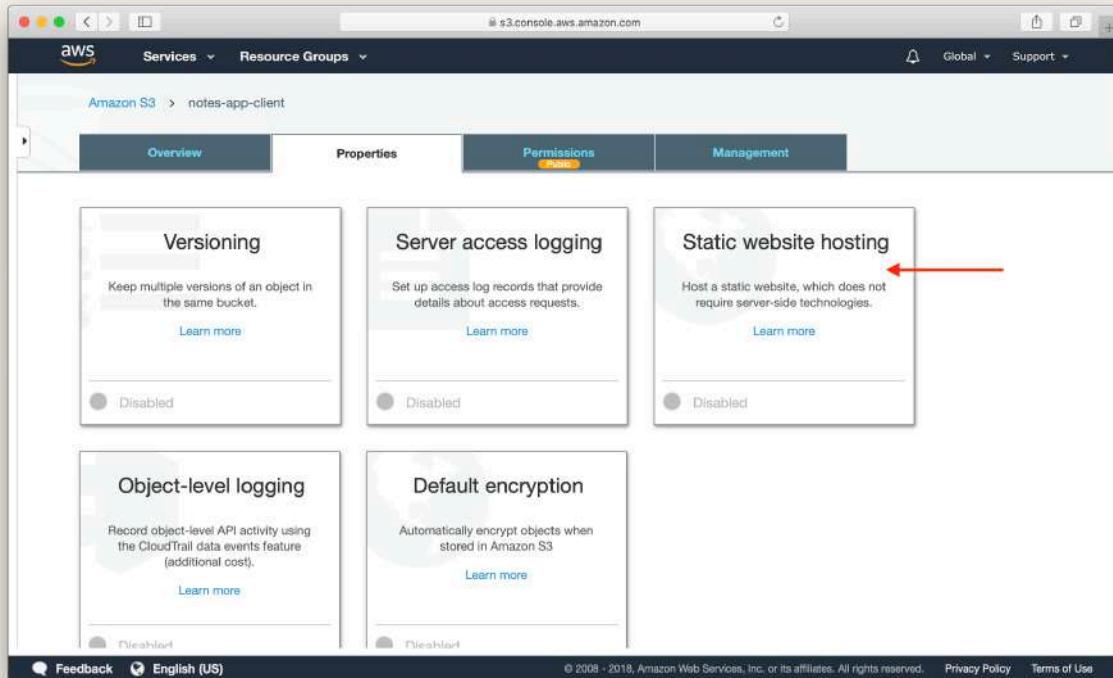
Enable Static Web Hosting

And finally we need to turn our bucket into a static website. Select the **Properties** tab from the top panel.



Select properties tab screenshot

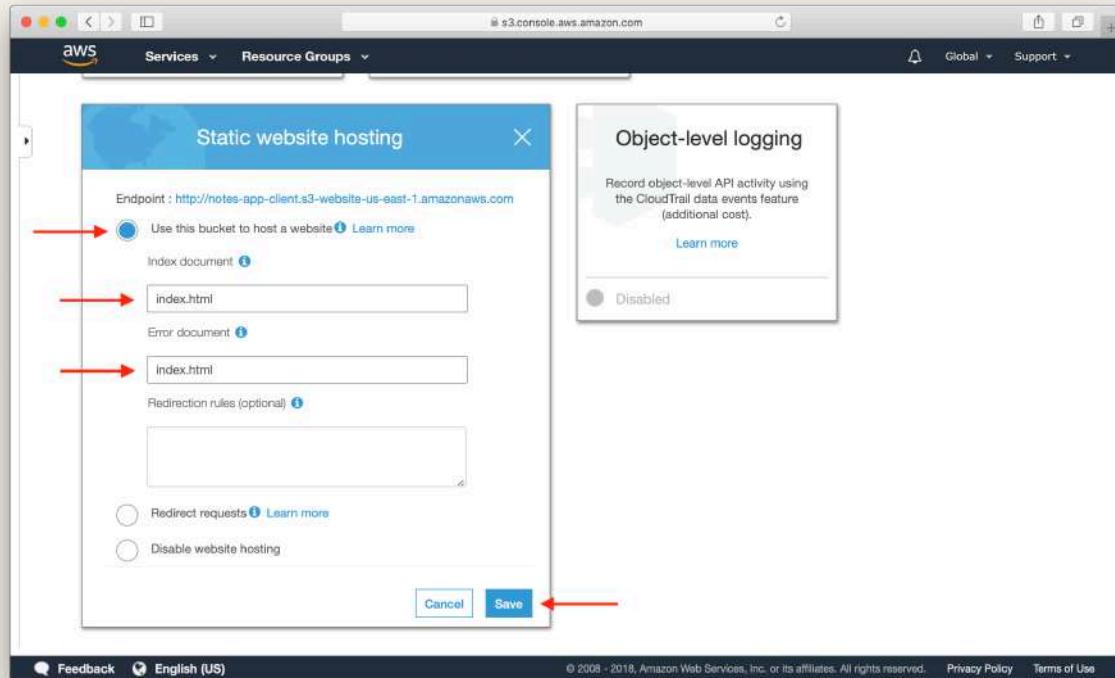
Select **Static website hosting**.



Select static website hosting screenshot

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`.



Edit static website hosting properties screenshot

Now that our bucket is all set up and ready, let's go ahead and upload our assets to it.



Help and discussion

View the [comments for this chapter on our forums](#)

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
```

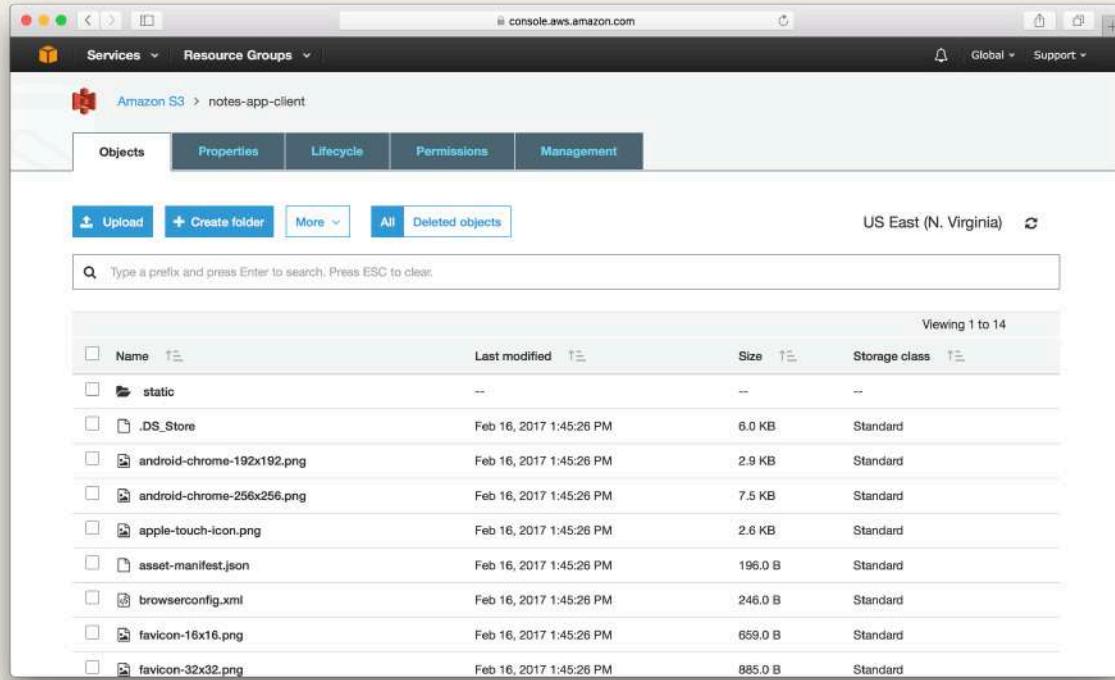
This packages all of our assets and places them in the `build/` directory.

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](#) 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](#) and check if your bucket has the files we just uploaded.

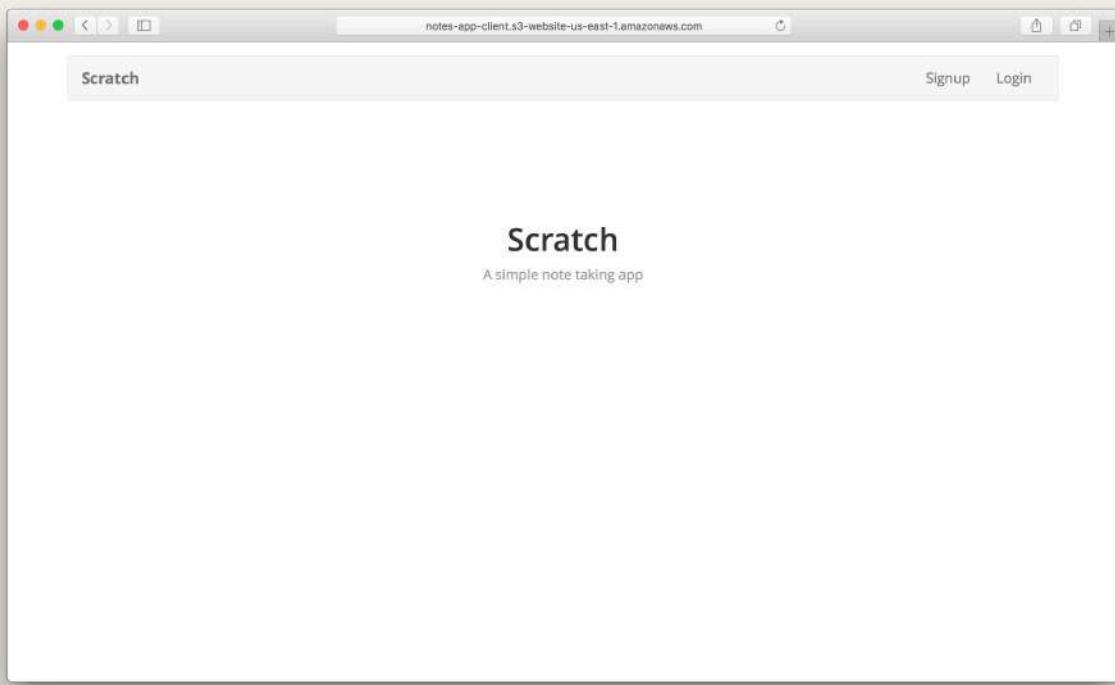


The screenshot shows the AWS S3 console interface. At the top, there's a navigation bar with 'Services', 'Resource Groups', and other global settings. Below that, the 'Amazon S3' section is selected, showing the path 'notes-app-client'. The main area has tabs for 'Objects', 'Properties', 'Lifecycle', 'Permissions', and 'Management'. Under 'Objects', there are buttons for 'Upload', '+ Create folder', 'More', 'All', and 'Deleted objects'. A search bar below these buttons contains the placeholder 'Type a prefix and press Enter to search, Press ESC to clear.' To the right of the search bar, it says 'US East (N. Virginia)' with a refresh icon. The main content area displays a table titled 'Viewing 1 to 14' with columns: Name, Last modified, Size, and Storage class. The table lists various files and folders uploaded to the bucket:

Name	Last modified	Size	Storage class
static	--	--	Standard
.DS_Store	Feb 16, 2017 1:45:26 PM	6.0 KB	Standard
android-chrome-192x192.png	Feb 16, 2017 1:45:26 PM	2.9 KB	Standard
android-chrome-256x256.png	Feb 16, 2017 1:45:26 PM	7.5 KB	Standard
apple-touch-icon.png	Feb 16, 2017 1:45:26 PM	2.6 KB	Standard
asset-manifest.json	Feb 16, 2017 1:45:26 PM	196.0 B	Standard
browserconfig.xml	Feb 16, 2017 1:45:26 PM	246.0 B	Standard
favicon-16x16.png	Feb 16, 2017 1:45:26 PM	659.0 B	Standard
favicon-32x32.png	Feb 16, 2017 1:45:26 PM	885.0 B	Standard

Uploaded to S3 screenshot

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>), you should see it live.



App live on S3 screenshot

Next we'll configure CloudFront to serve our app out globally.



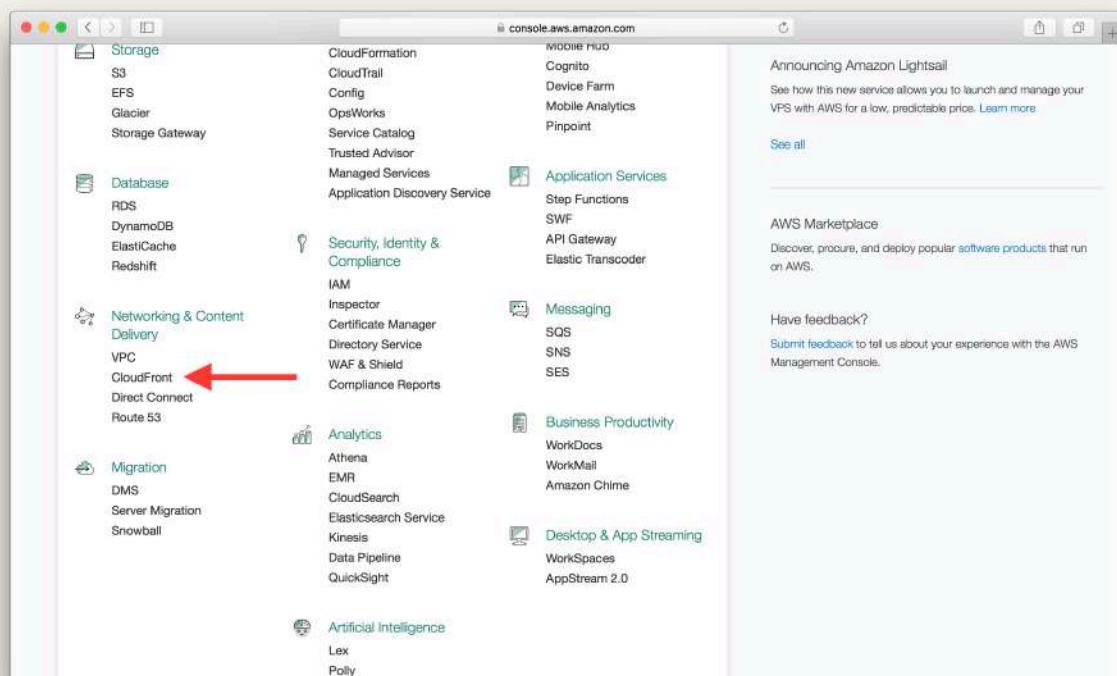
Help and discussion

View the [comments for this chapter on our forums](#)

Create a CloudFront Distribution

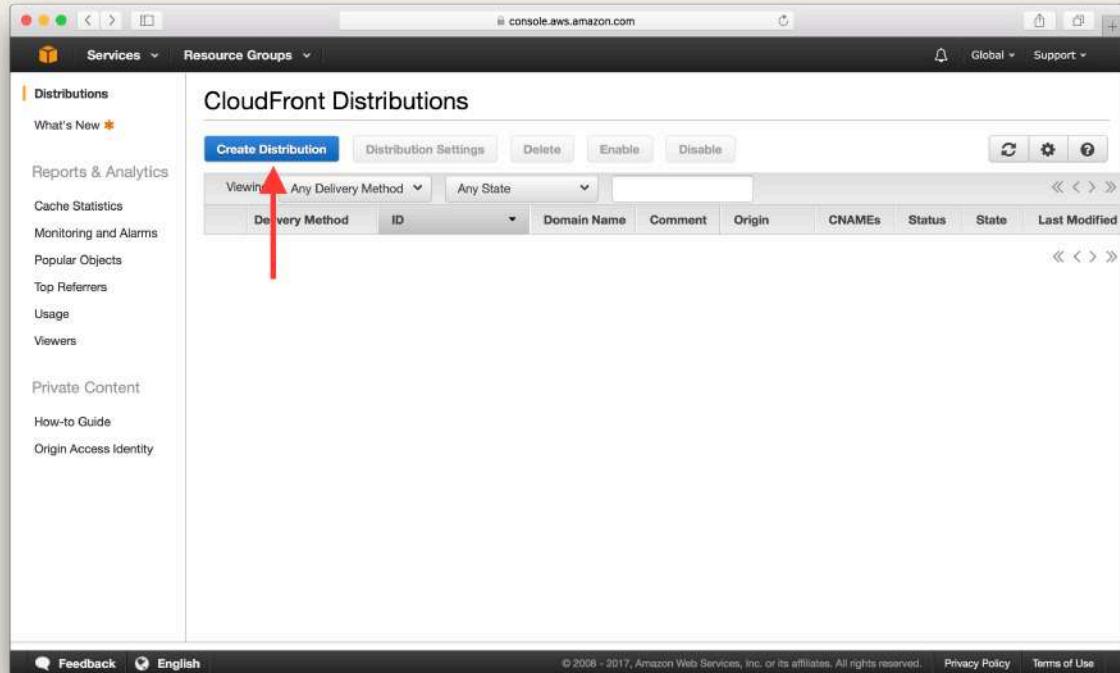
Now that we have our app up and running on S3, let's serve it out globally through CloudFront. To do this we need to create an AWS CloudFront Distribution.

Select CloudFront from the list of services in your [AWS Console](#).



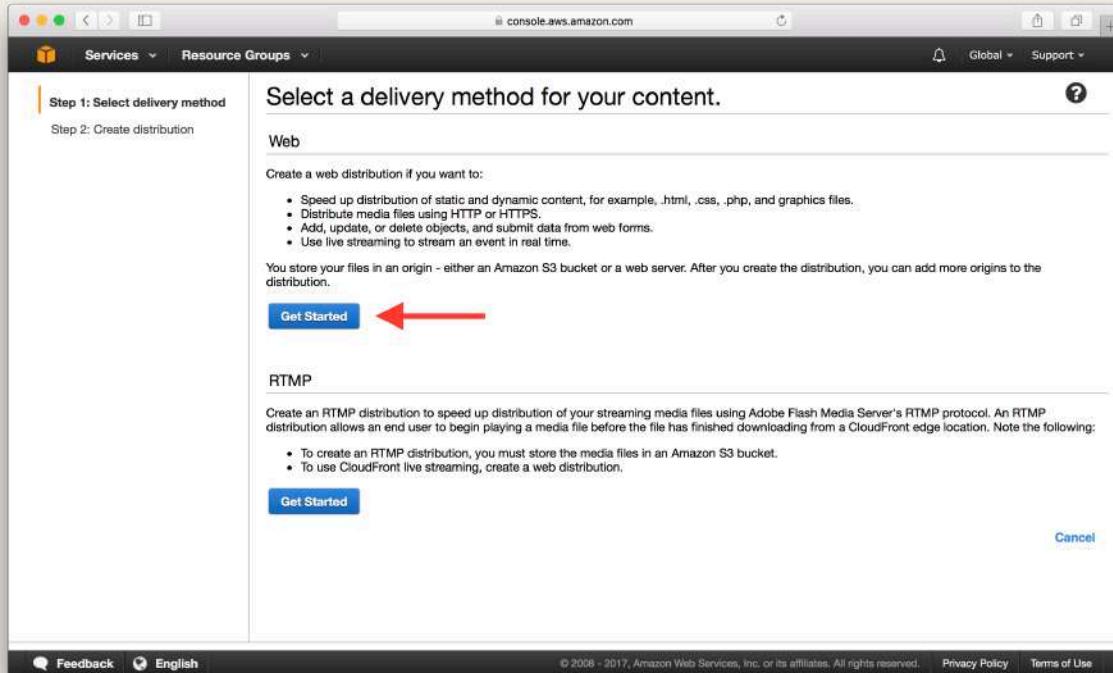
Select AWS CloudFront service screenshot

Then select **Create Distribution**.



Create AWS CloudFront Distribution screenshot

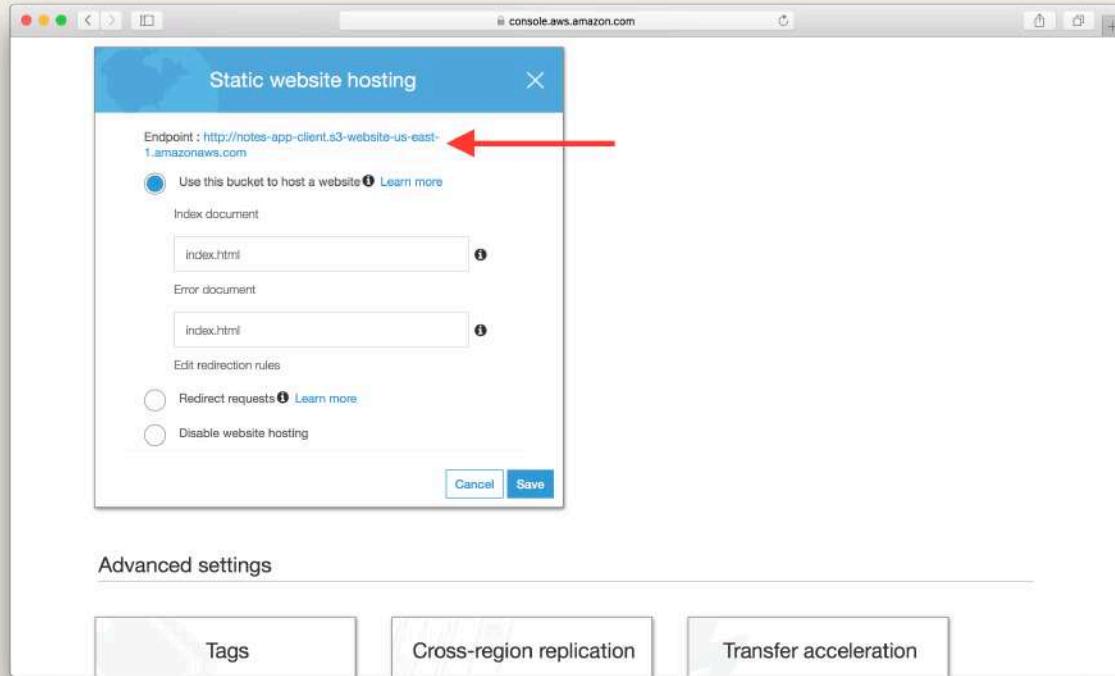
And then in the **Web** section select **Get Started**.



Select get started web screenshot

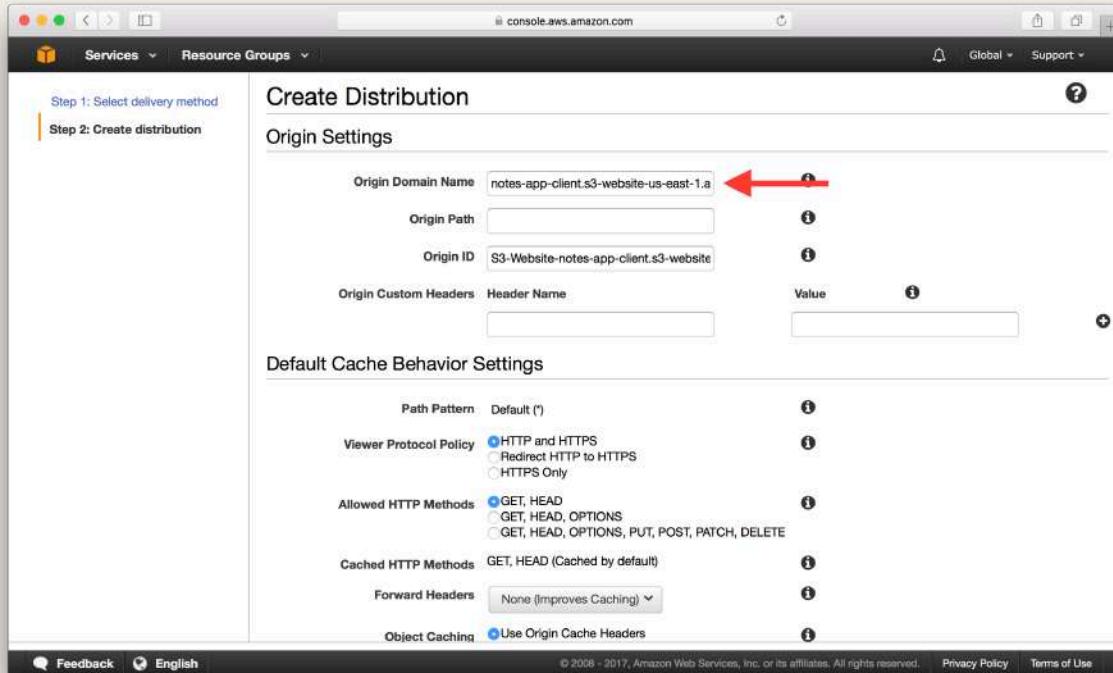
In the Create Distribution form we need to start by specifying the Origin Domain Name for our Web CloudFront Distribution. This field is pre-filled with a few options including the S3 bucket we created. But we are **not** going to select on the options in the dropdown. This is because the options here 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.



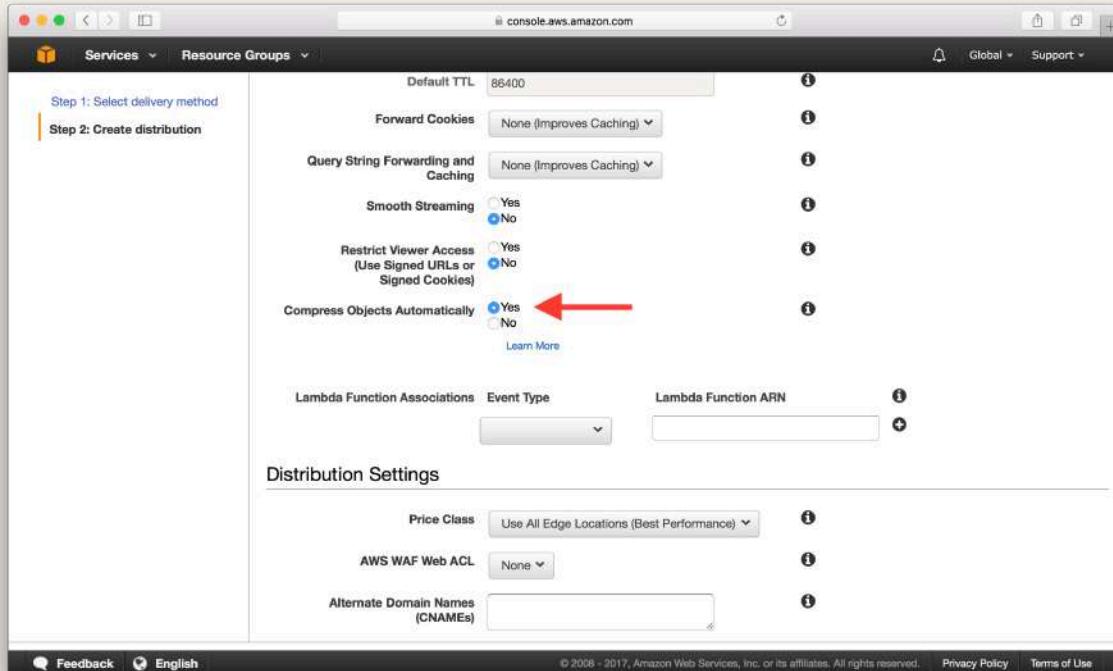
S3 static website domain screenshot

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`.



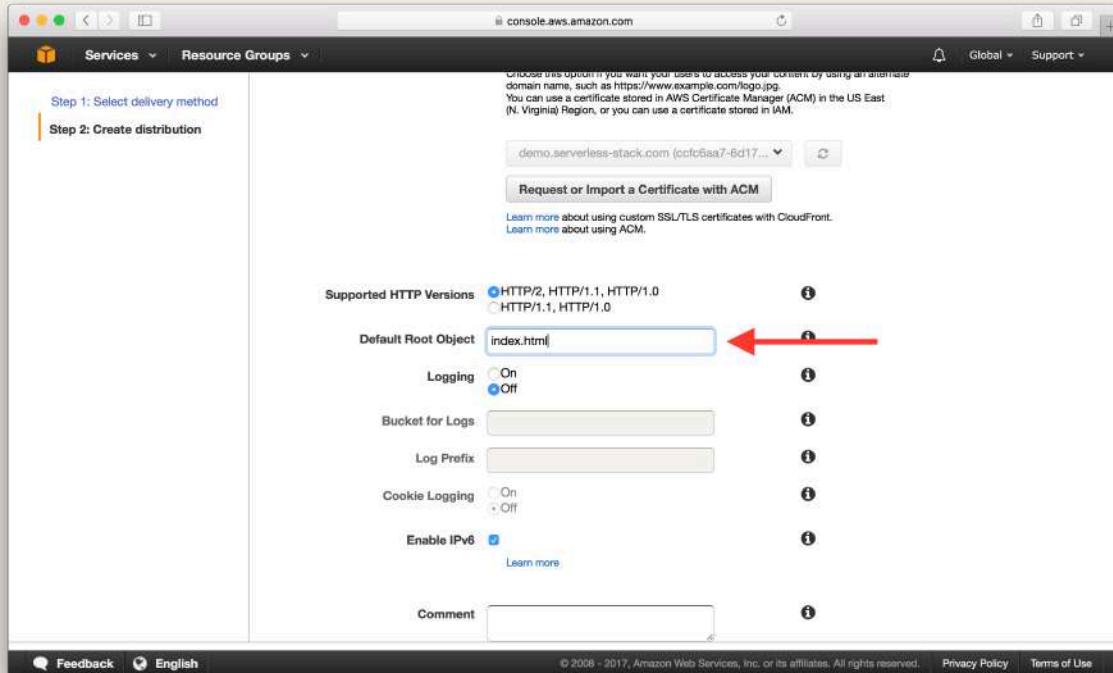
Fill origin domain name field screenshot

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.



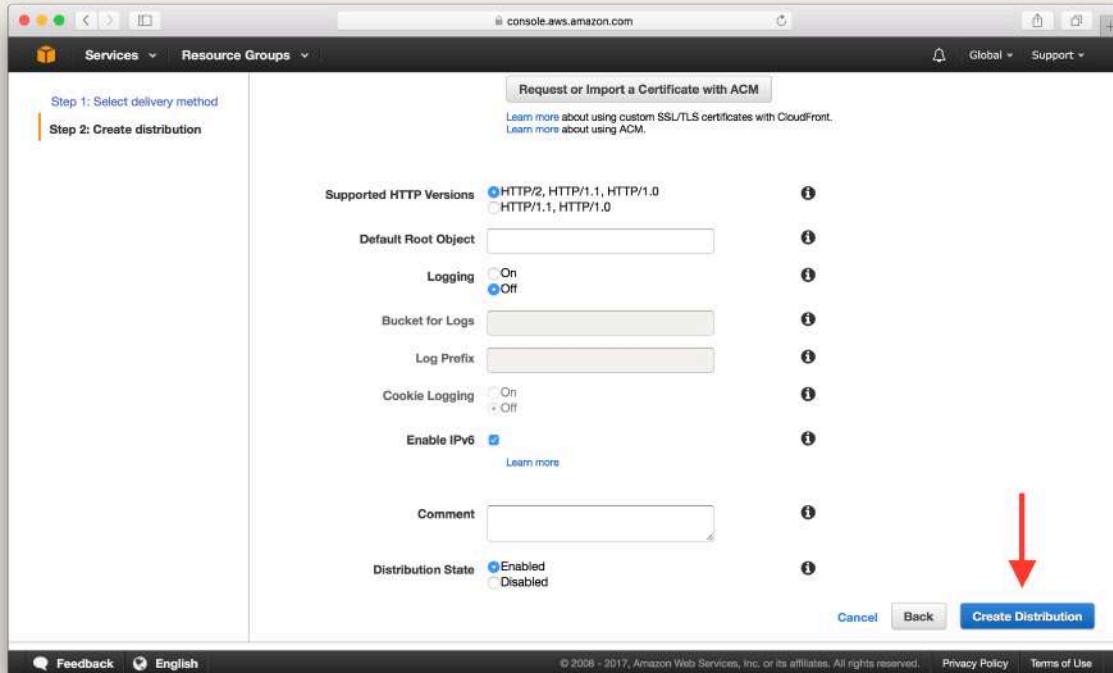
Select compress objects automatically screenshot

Next, scroll down a bit further to set the **Default Root Object** to `index.html`.



Set default root object screenshot

And finally, hit **Create Distribution**.



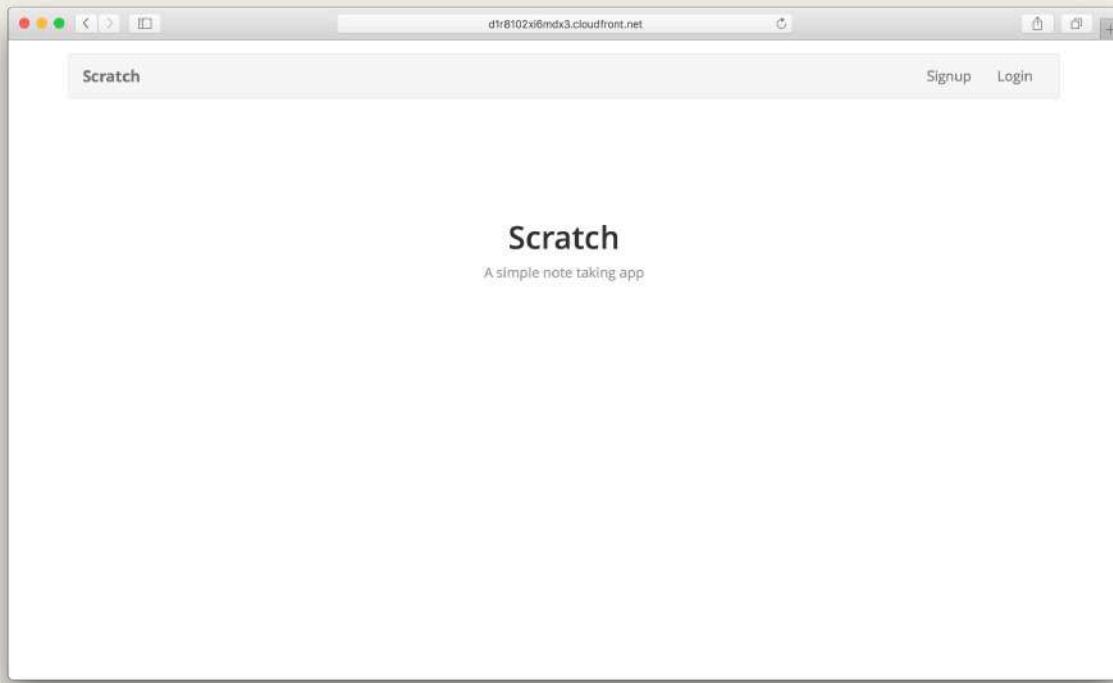
Hit create distribution screenshot

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.

The screenshot shows the AWS CloudFront Distribution configuration page for a distribution with ID E1KTCKT9SOAHBW. The 'General' tab is selected. In the 'Alternate Domain Names (CNAMEs)' section, the 'Domain Name' field contains 'd1r8102xi6ndx3.cloudfront.net'. A red arrow points to this field. Other fields in the same section include 'SSL Certificate' (Default CloudFront Certificate (*.cloudfront.net)) and 'Custom SSL Client Support' (disabled). The page also displays other distribution settings like ARN, Log Prefix, Delivery Method, and Price Class.

AWS CloudFront Distribution doamin name screenshot

And if you navigate over to that in your browser, you should see your app live.



App live on CloudFront screenshot

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.

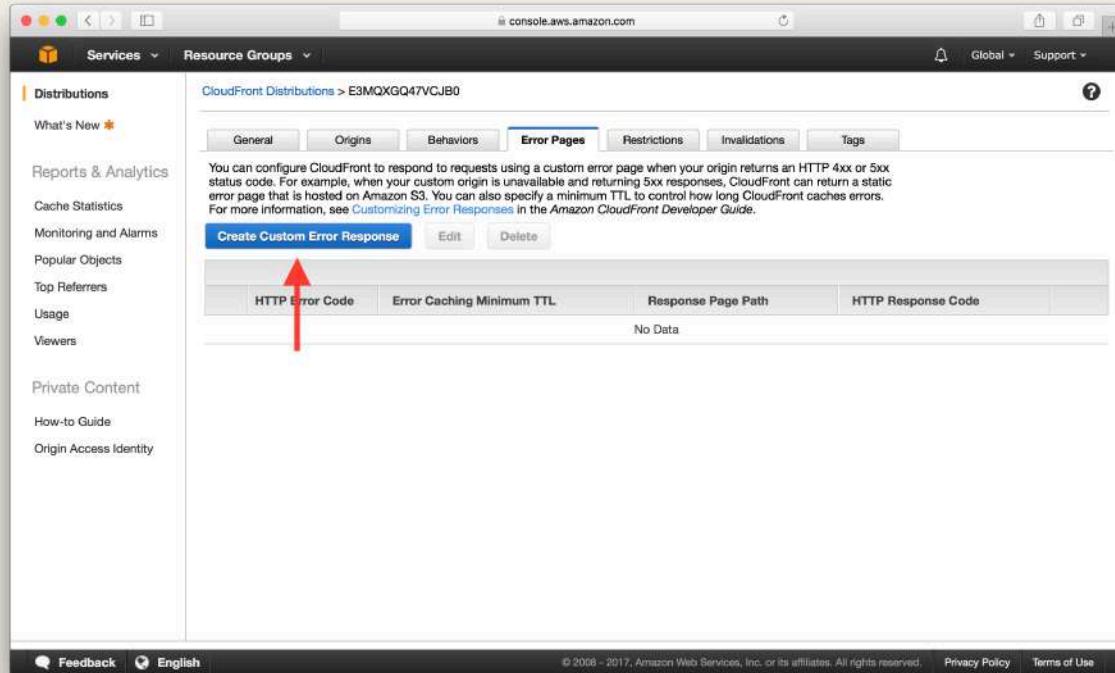
To set up a custom error response, head over to the **Error Pages** tab in our Distribution.

The screenshot shows the AWS CloudFront Distribution configuration page for a distribution with ID E1KTCKT9SOAHBW. The 'Error Pages' tab is highlighted with a red arrow. The page displays various distribution settings such as ARN, Log Prefix, Delivery Method, and SSL Certificate.

Setting	Value
Distribution ID	E1KTCKT9SOAHBW
ARN	arn:aws:cloudfront::232771856781:distribution/E1KTCKT9SOAHBW
Log Prefix	-
Delivery Method	Web
Cookie Logging	Off
Distribution Status	Deployed
Comment	-
Price Class	Use All Edge Locations (Best Performance)
AWS WAF Web ACL	-
State	Enabled
Alternate Domain Names (CNAMEs)	demo.serverless-stack.com
SSL Certificate	demo.serverless-stack.com (93691abe-f8be-4020-905e-af94bc00913)
Domain Name	d1r6102x16mdx3.cloudfront.net
Custom SSL Client Support	Only Clients that Support Server Name Indication (SNI)
Supported HTTP Versions	HTTP/2, HTTP/1.1, HTTP/1.0
IPv6	Enabled
Default Root Object	-
Last Modified	2017-06-08 15:36 UTC+1
Log Bucket	-

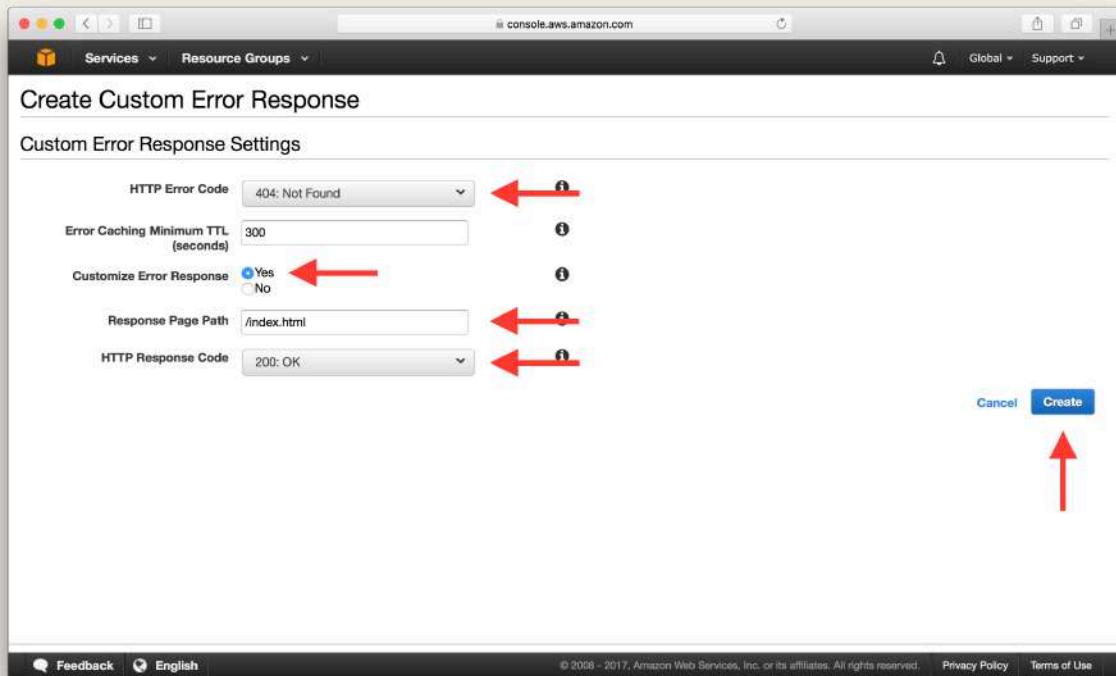
Error Pages in CloudFront screenshot

And select **Create Custom Error Response**.



Select Create Custom Error Response in CloudFront screenshot

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**.



Create Custom Error Response screenshot

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.

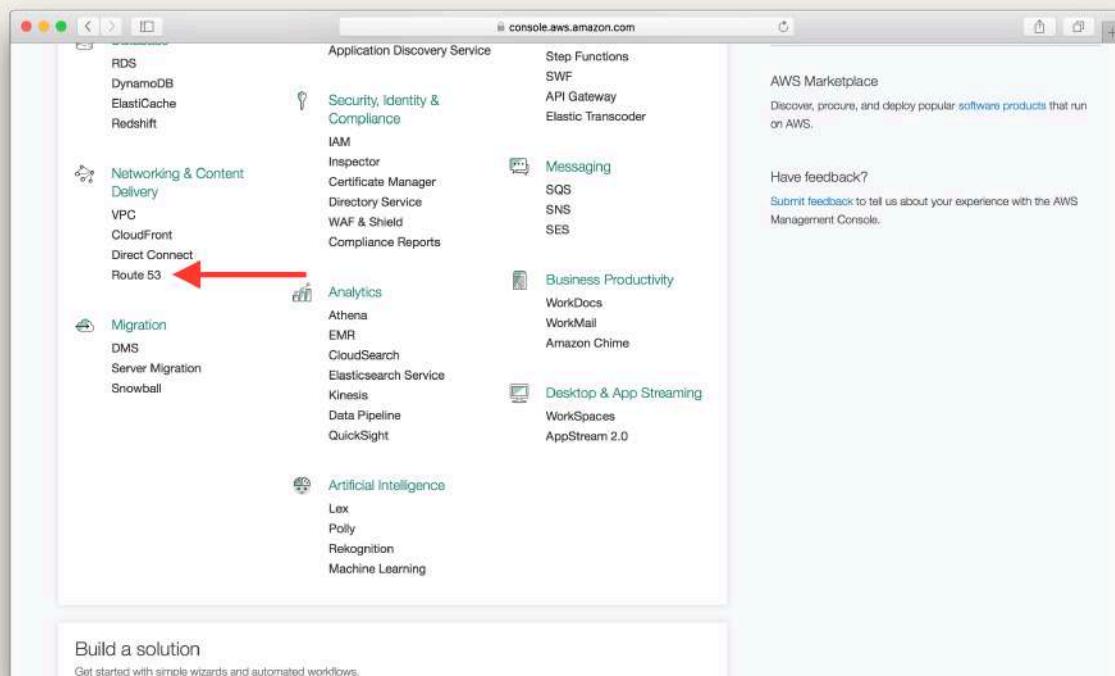


Help and discussion

View the [comments](#) for this chapter on our forums

Purchase a Domain with Route 53

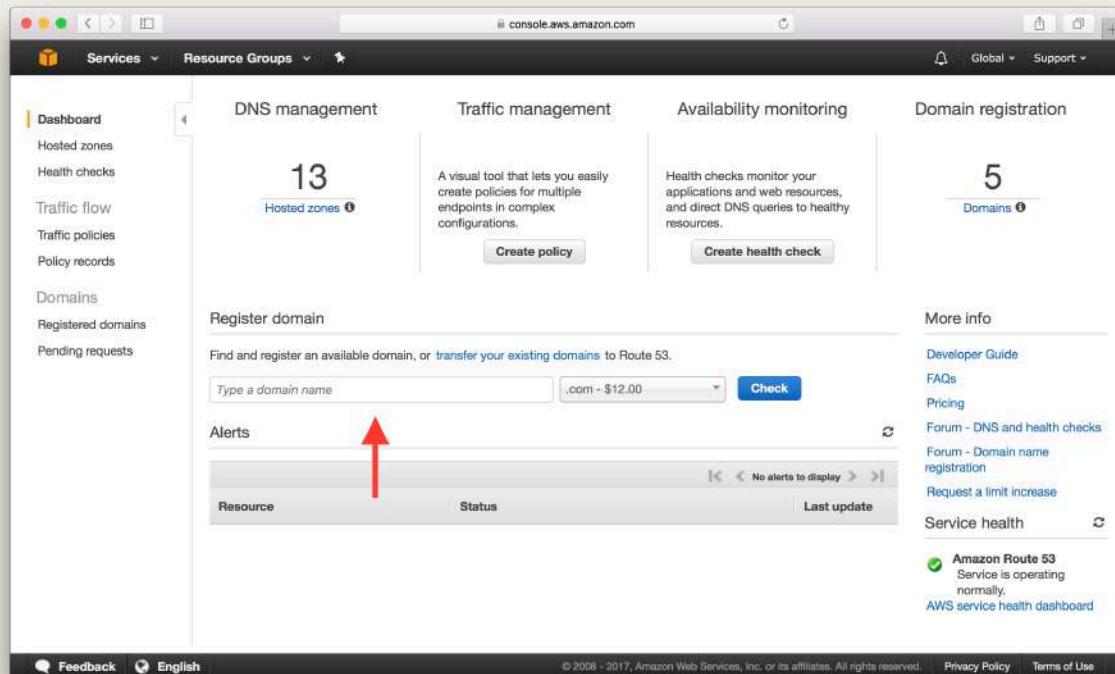
Now that we have our CloudFront distribution live, let's set up our domain with it. You can purchase a domain right from the [AWS Console](#) by heading to the Route 53 section in the list of services.



Select Route 53 service screenshot

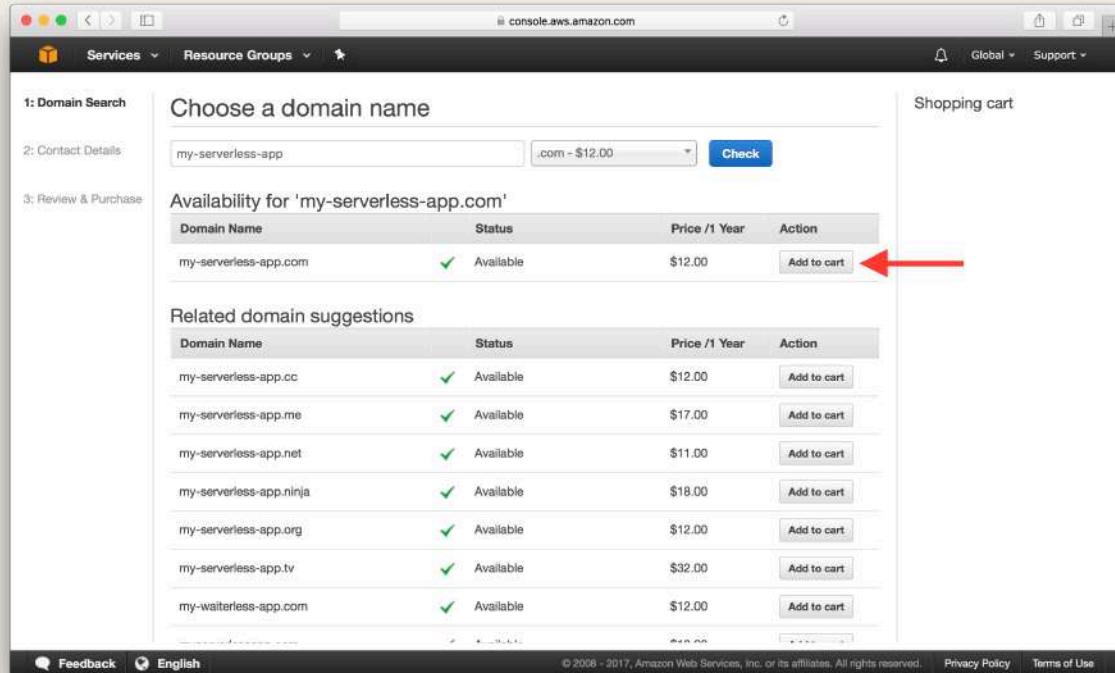
Purchase a Domain with Route 53

Type in your domain in the **Register domain** section and click **Check**.



Search available domain screenshot

After checking its availability, click **Add to cart**.



Add domain to cart screenshot

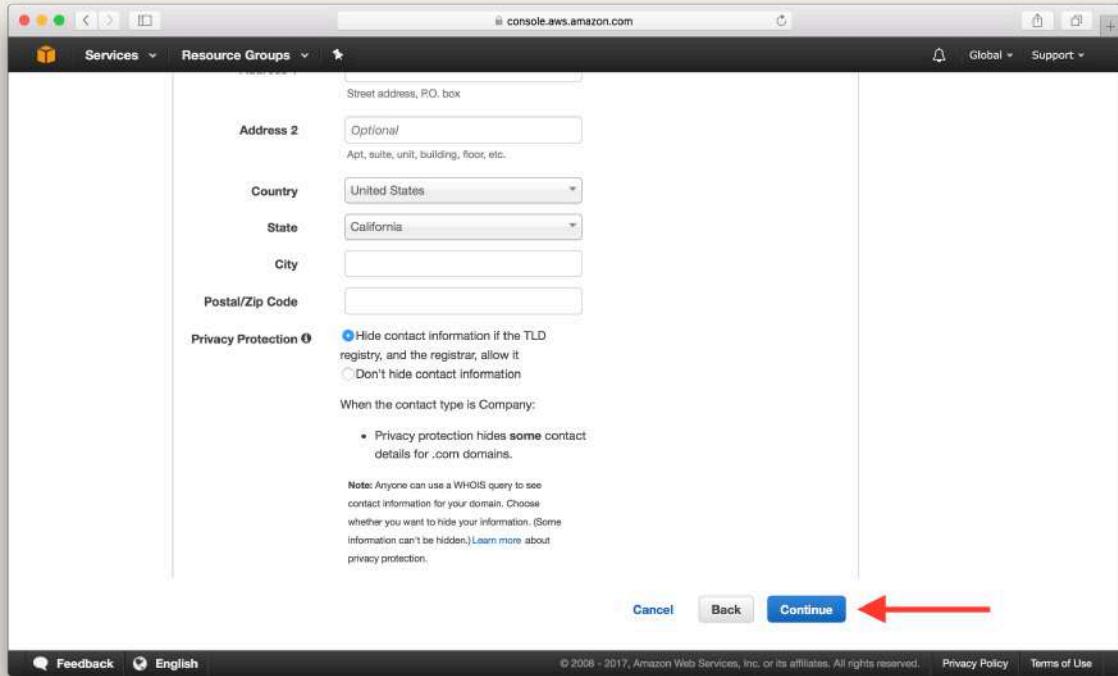
And hit **Continue** at the bottom of the page.

The screenshot shows a list of related domain suggestions for "my-serverless-app". Each suggestion includes the domain name, status (Available), price (\$12.00 to \$32.00 per year), and an "Add to cart" button. To the right of the table, there is a sidebar titled "Monthly Fees for DNS Management" with a link to view pricing details. At the bottom right of the main content area, there are "Cancel" and "Continue" buttons, with a red arrow pointing to the "Continue" button.

Domain Name	Status	Price / 1 Year	Action
my-serverless-app.cc	✓ Available	\$12.00	Add to cart
my-serverless-app.me	✓ Available	\$17.00	Add to cart
my-serverless-app.net	✓ Available	\$11.00	Add to cart
my-serverless-app.ninja	✓ Available	\$18.00	Add to cart
my-serverless-app.org	✓ Available	\$12.00	Add to cart
my-serverless-app.tv	✓ Available	\$32.00	Add to cart
my-waiterless-app.com	✓ Available	\$12.00	Add to cart
myserverlessapp.com	✓ Available	\$12.00	Add to cart
myserverlesshomepage.com	✓ Available	\$12.00	Add to cart
myserverlessprogram.com	✓ Available	\$12.00	Add to cart
savemyserverlessapp.com	✓ Available	\$12.00	Add to cart
theserverlessapp.com	✓ Available	\$12.00	Add to cart

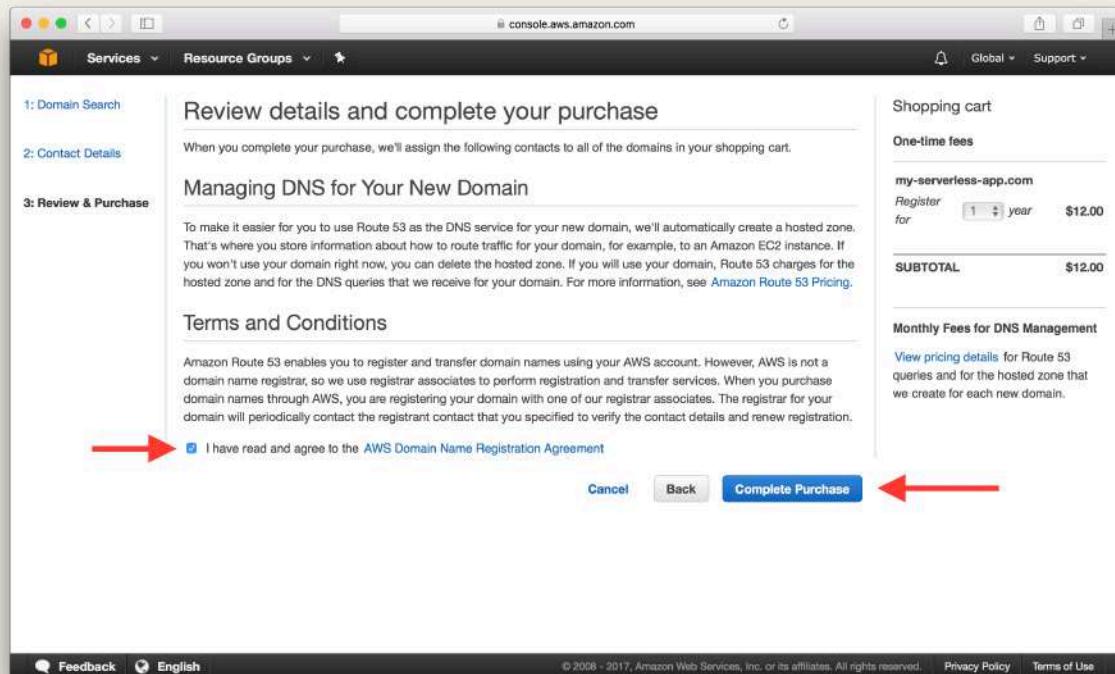
Continue to contact details screenshot

Fill in your contact details and hit **Continue** once again.



Continue to confirm details screenshot

Finally, review your details and confirm the purchase by hitting **Complete Purchase**.



Confirm domain purchase screenshot

Next up, we'll set up SSL to make sure we can use HTTPS with our domain.



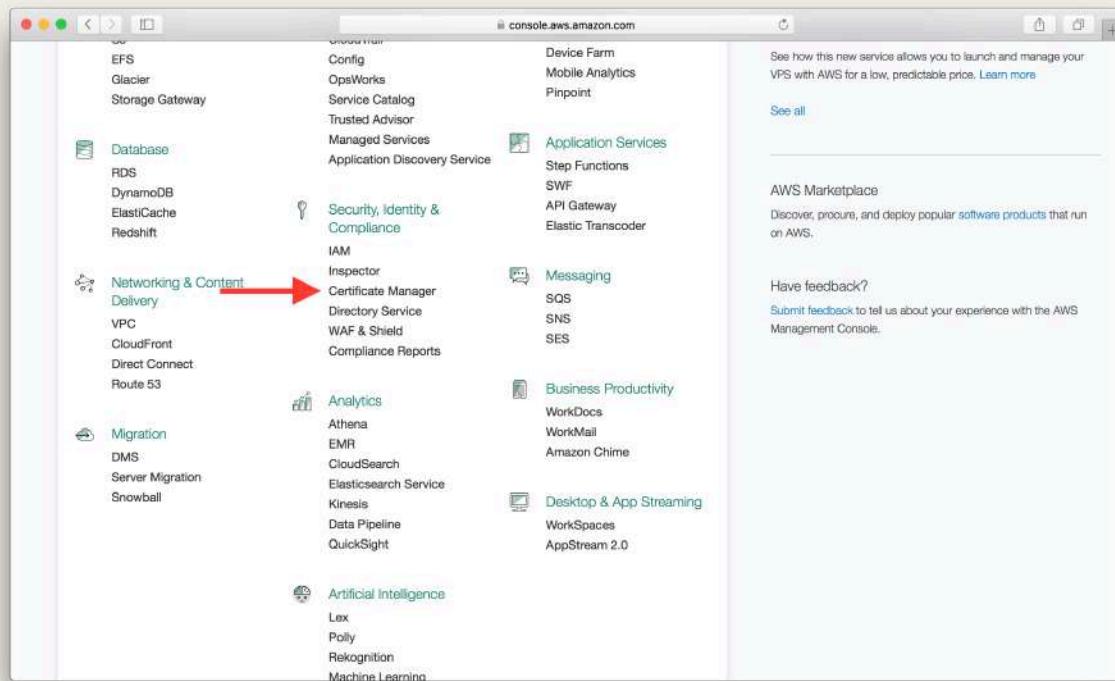
Help and discussion

View the [comments for this chapter on our forums](#)

Set up SSL

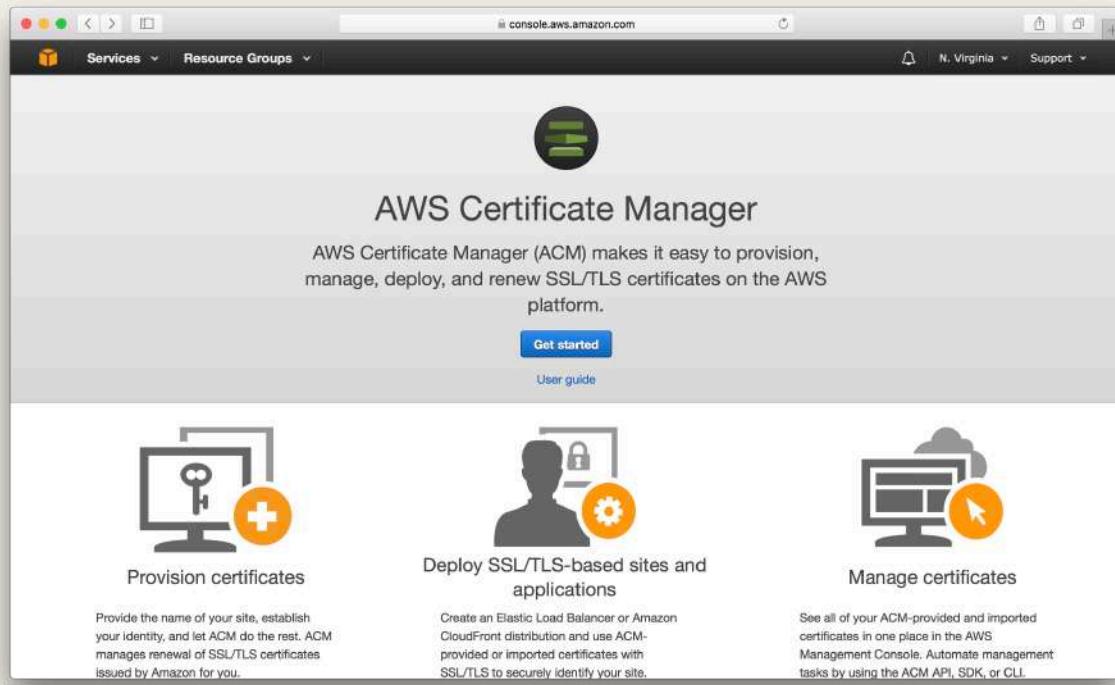
Now that we have our domain, request a certificate to enable us to use SSL or HTTPS with our domain. AWS makes this fairly easy to do, thanks to Certificate Manager.

Select **Certificate Manager** from the list of services in your [AWS Console](#). 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.



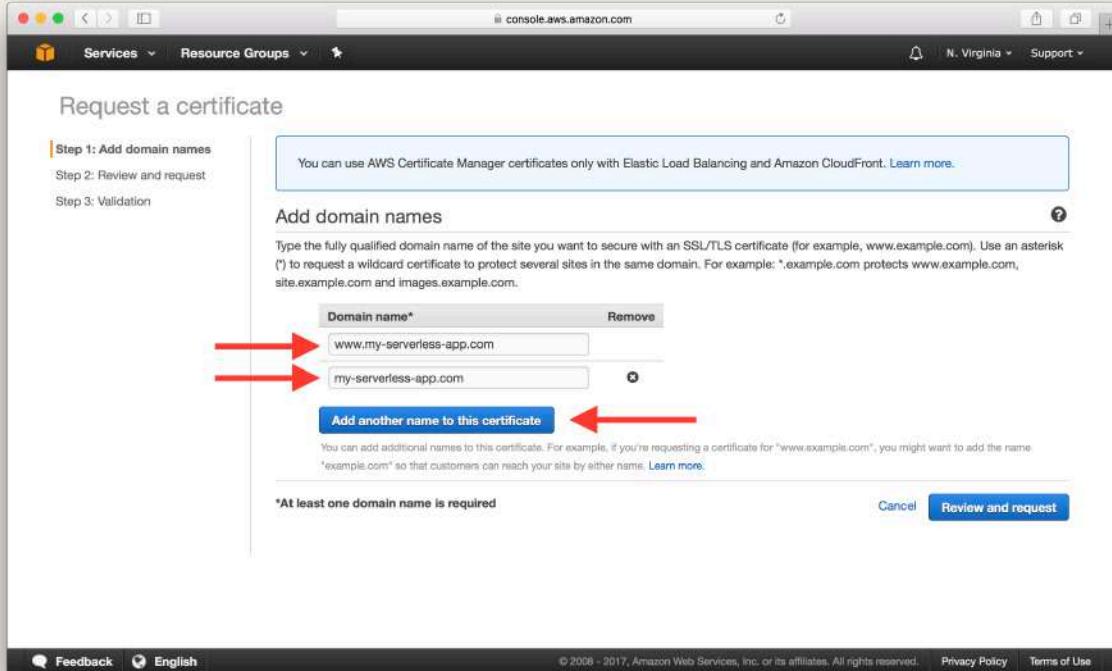
Select Certificate Manager service screenshot

If this is your first certificate, you'll need to hit **Get started**. If not then hit **Request a certificate** from the top.



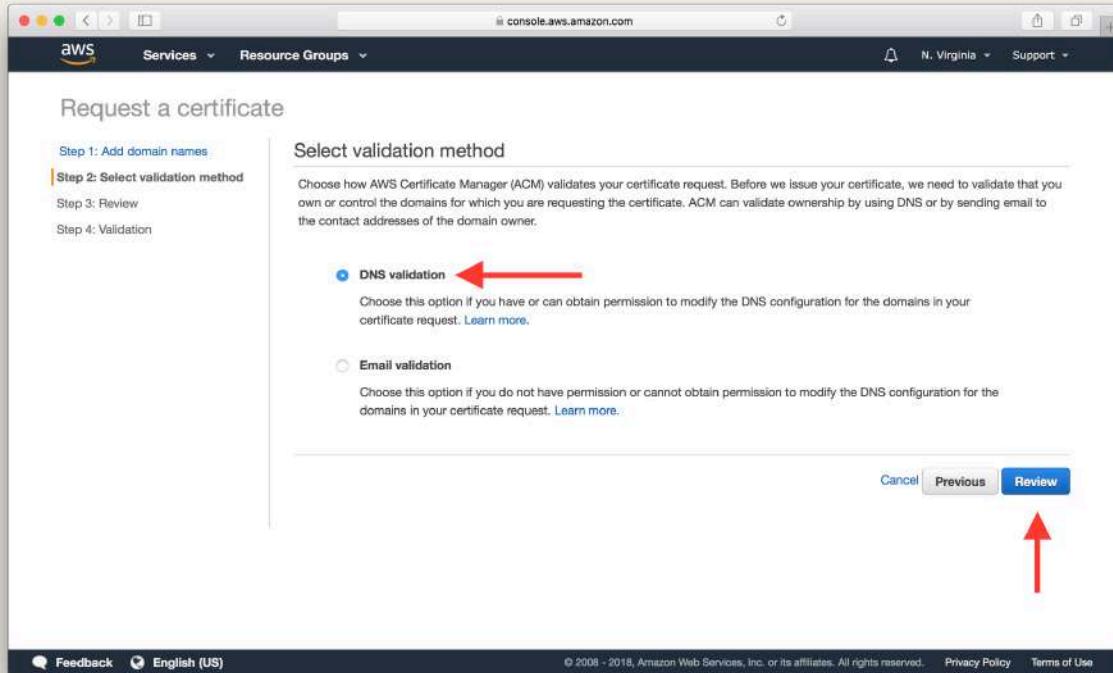
Get started with Certificate Manager screenshot

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.



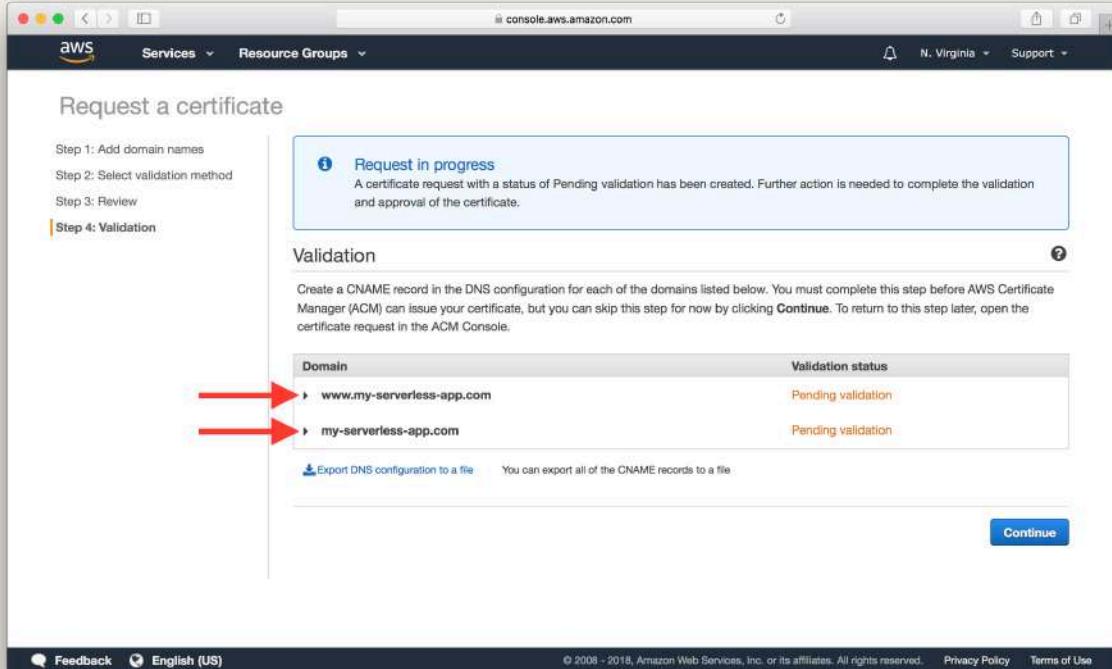
Add domain names to certificate screenshot

Now to confirm that we control the domain, select the **DNS validation** method and hit **Review**.



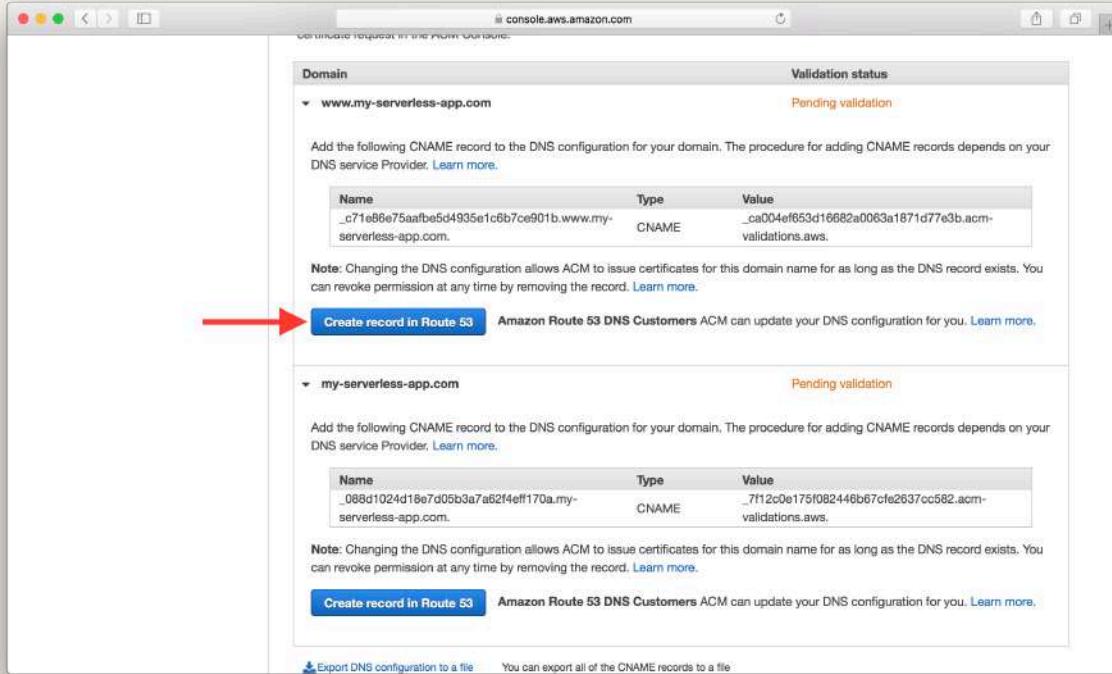
Select dns validation for certificate screenshot

On the validation screen expand the two domains we are trying to validate.



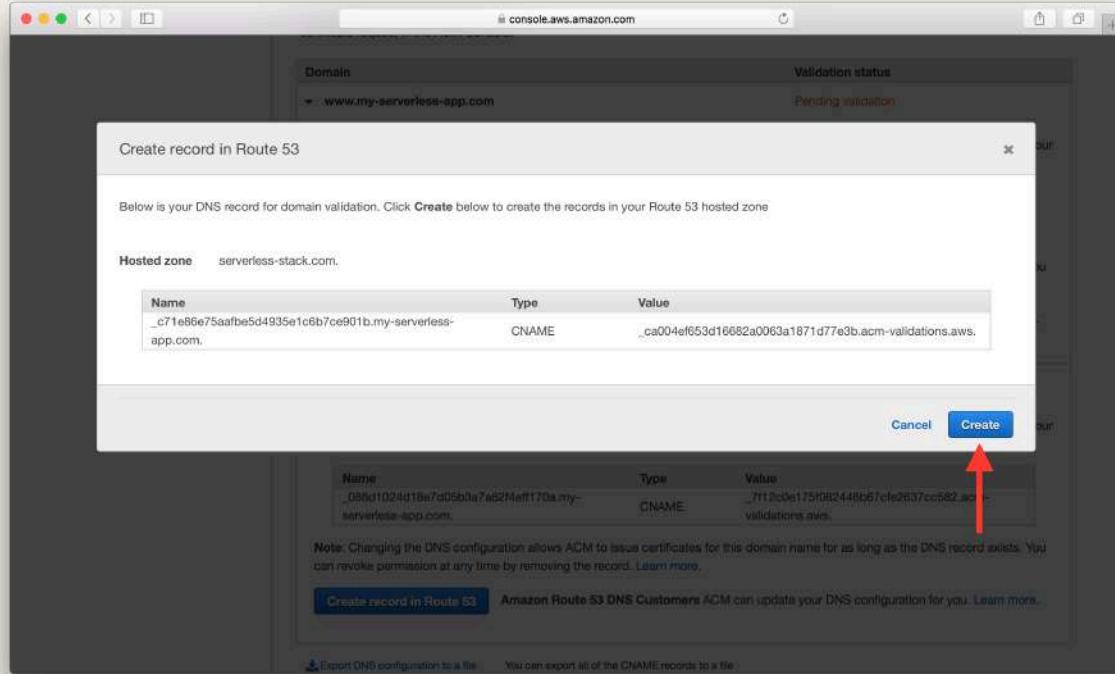
Expand dns validation details screenshot

Since we control the domain through Route 53, we can directly create the DNS record through here by hitting **Create record in Route 53**.



Create Route 53 dns record screenshot

And confirm that you want the record to be created by hitting **Create**.



Confirm Route 53 dns record screenshot

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 up, we'll associate our domain and its certificate with our CloudFront Distribution.



Help and discussion

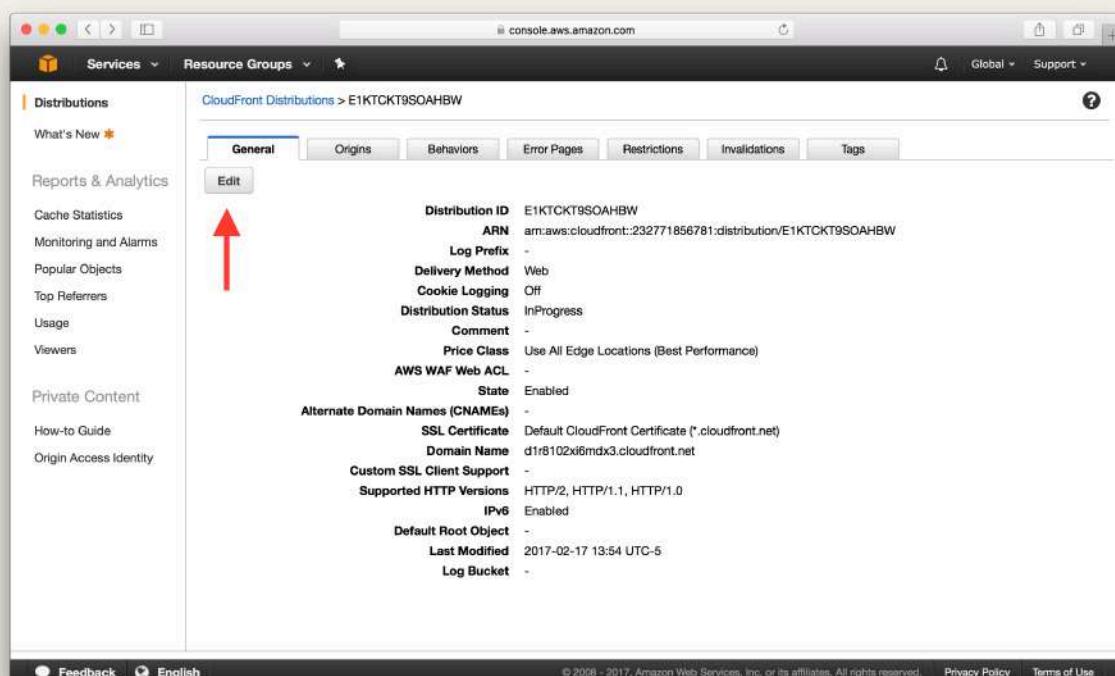
View the [comments for this chapter on our forums](#)

Set up Your Domain with CloudFront

Now that we have our domain and a certificate to serve it over HTTPS, let's associate these with our CloudFront Distribution

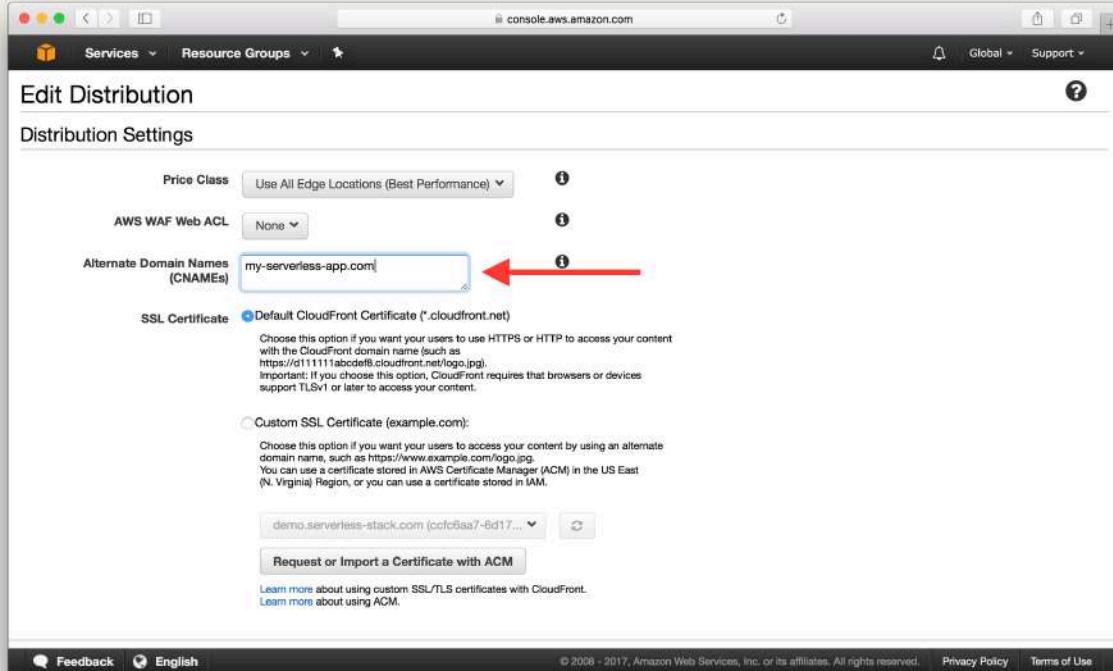
Add Alternate Domain for CloudFront Distribution

Head over to the details of your CloudFront Distribution and hit **Edit**.



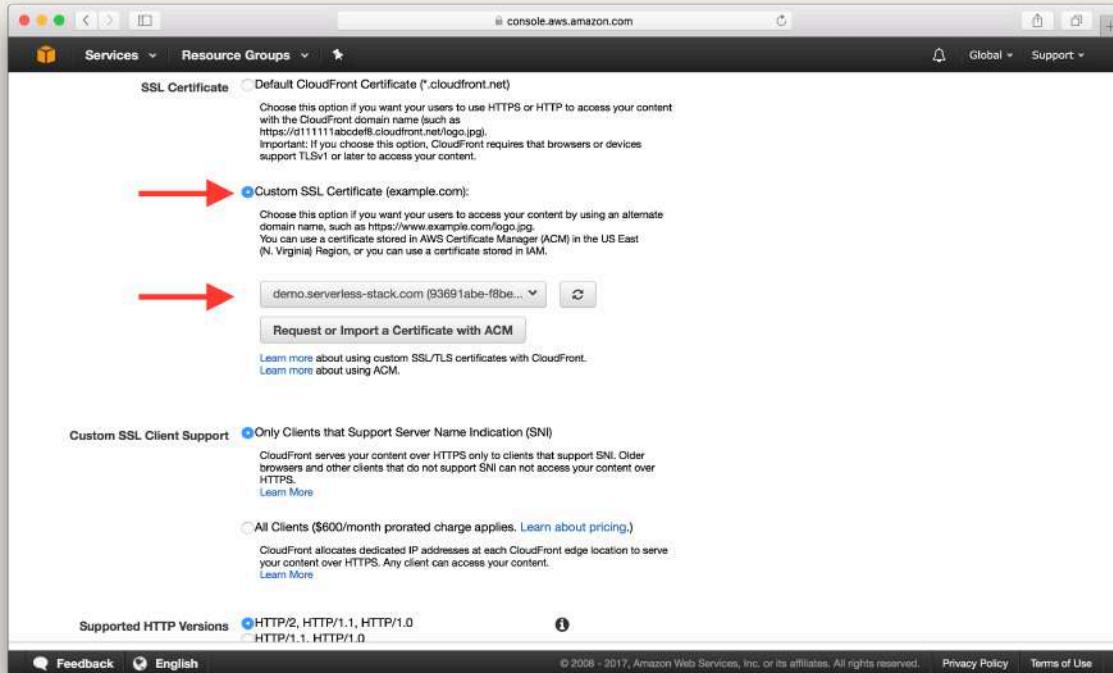
Edit CloudFront Distribution screenshot

And type in your new domain name in the **Alternate Domain Names (CNAMEs)** field.



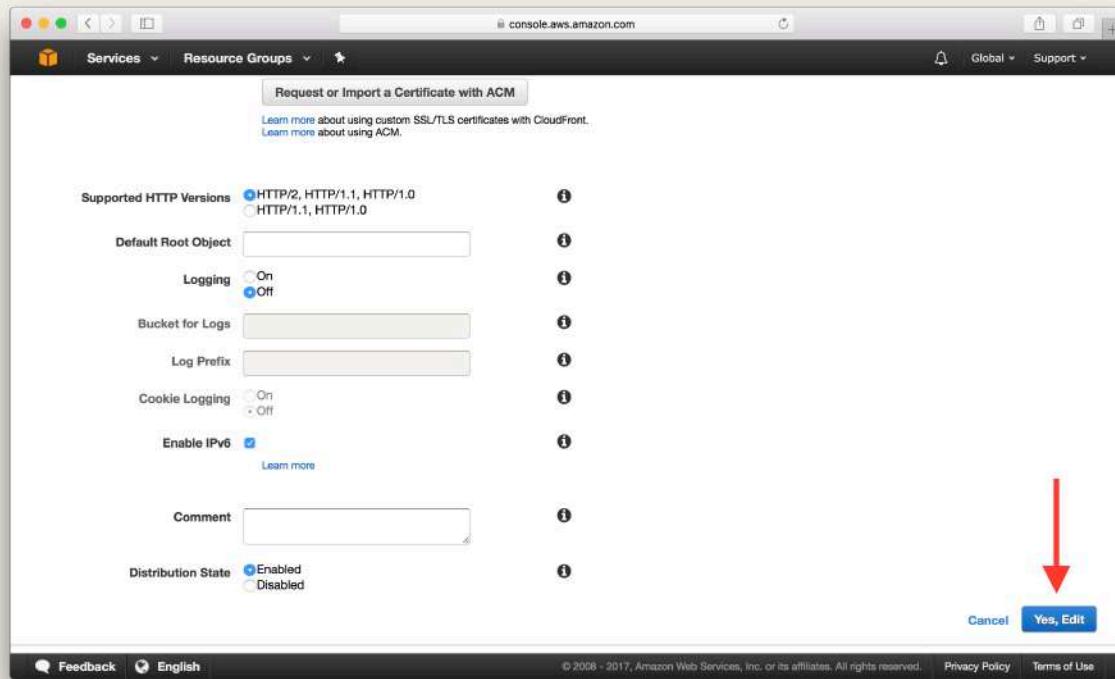
Set alternate domain name screenshot

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**.



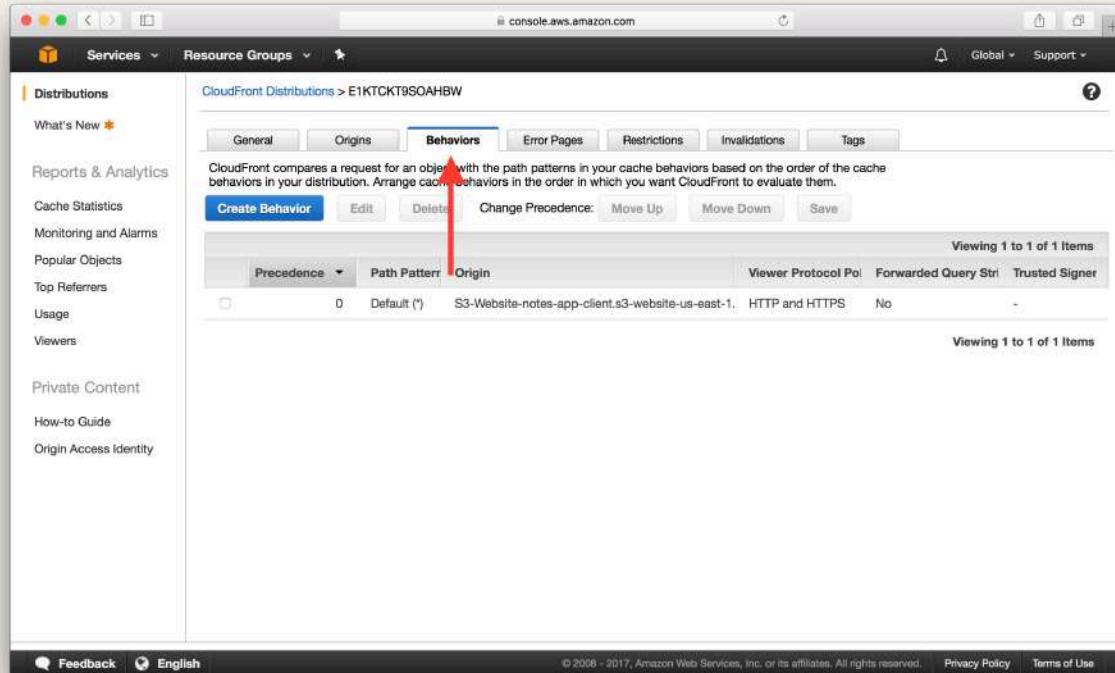
Select custom SSL Certificate screenshot

Scroll down and hit **Yes, Edit** to save the changes.



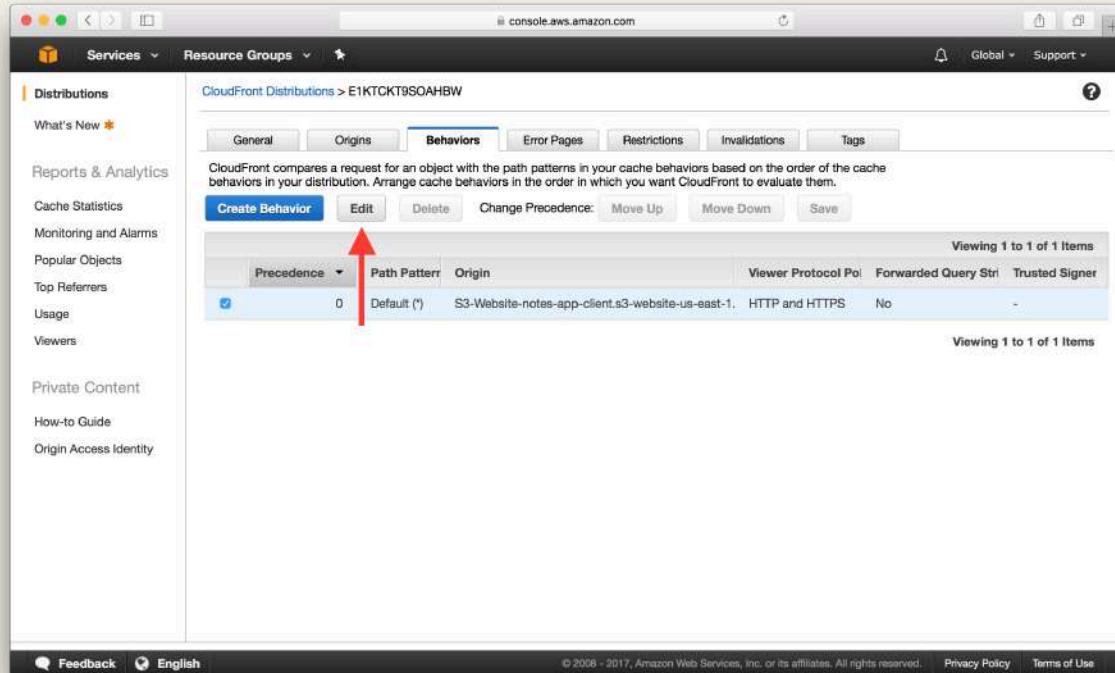
Yes edit CloudFront changes screenshot

Next, head over to the **Behaviors** tab from the top.



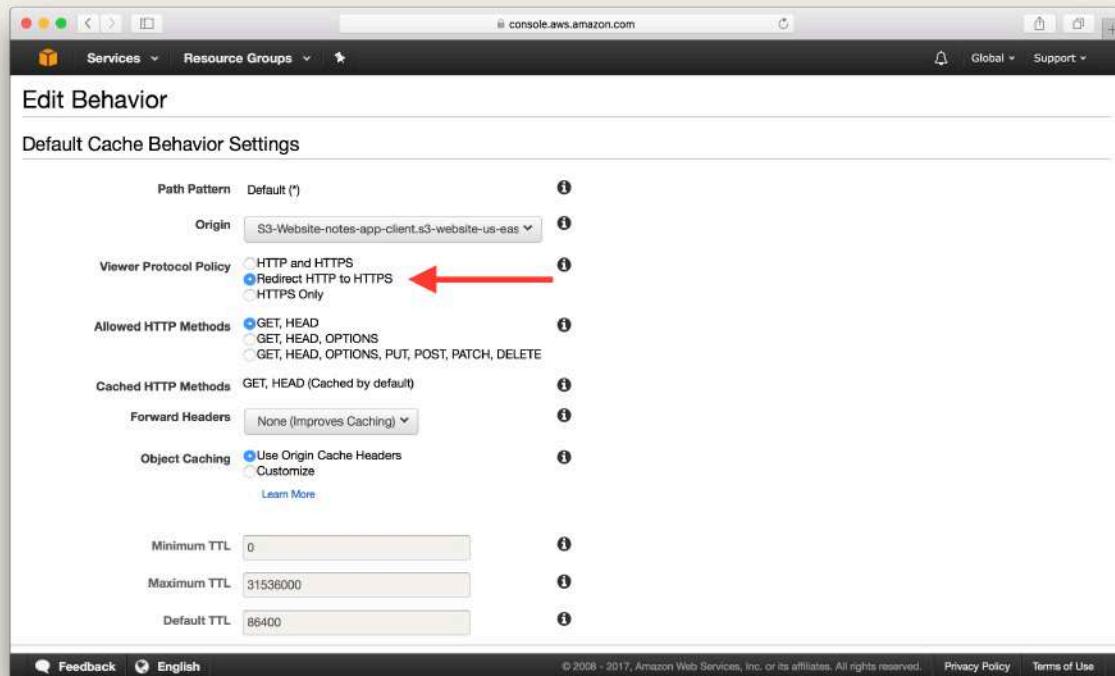
Select Behaviors tab screenshot

And select the only one we have and hit **Edit**.



Edit Distribution Behavior screenshot

Then switch the **Viewer Protocol Policy** to **Redirect HTTP to HTTPS**. And scroll down to the bottom and hit **Yes, Edit**.

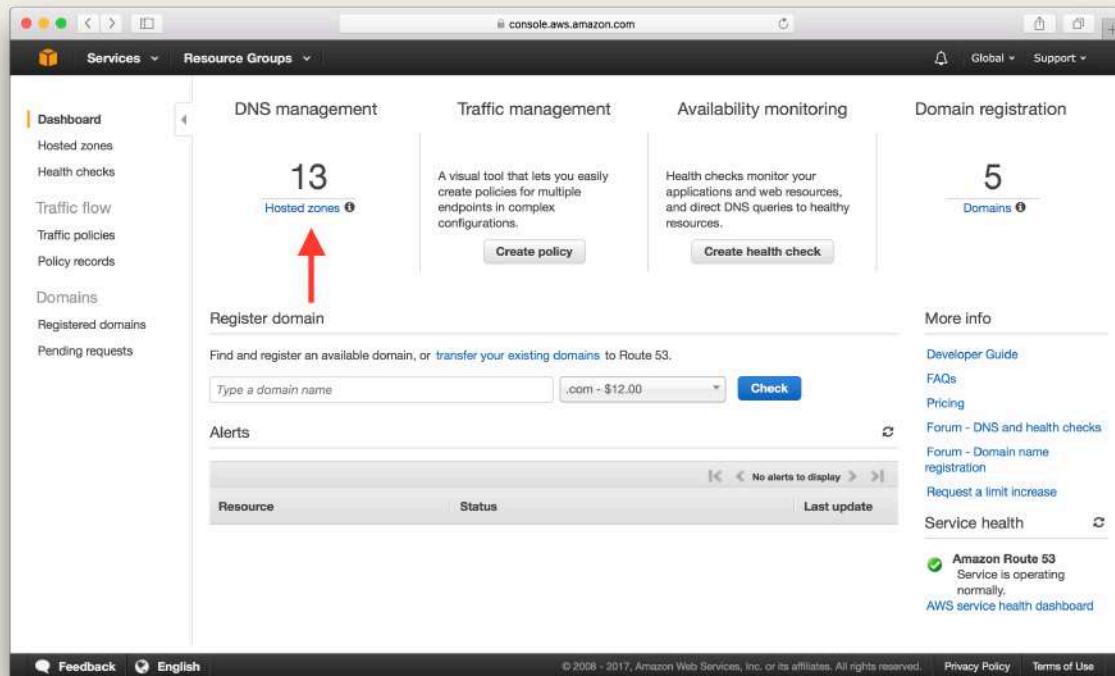


Switch Viewer Protocol Policy screenshot

Next, let's point our domain to the CloudFront Distribution.

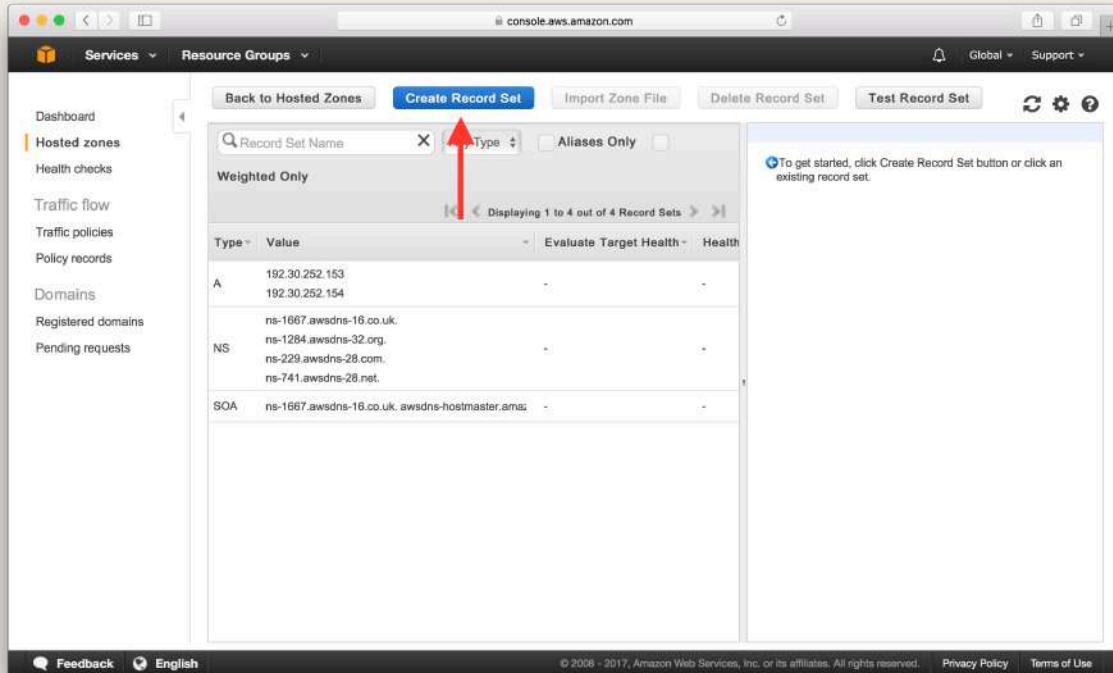
Point Domain to CloudFront Distribution

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**.



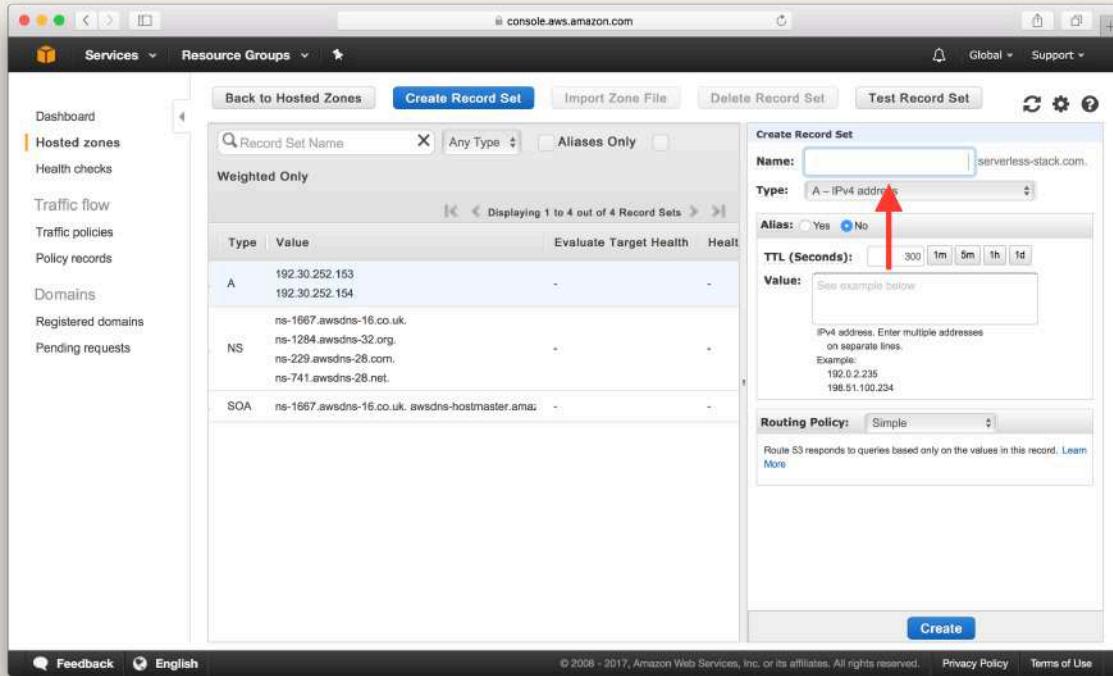
Select Route 53 hosted zones screenshot

Select your domain from the list and hit **Create Record Set** in the details screen.



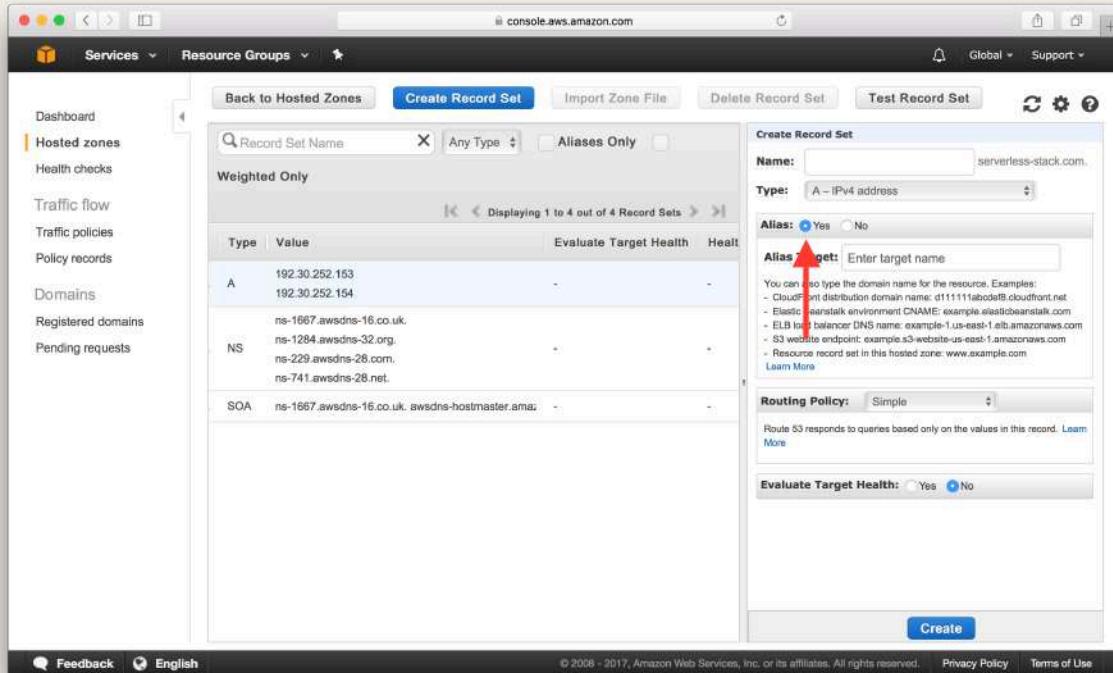
Select create record set screenshot

Leave the **Name** field empty since we are going to point our bare domain (without the www.) to our CloudFront Distribution.



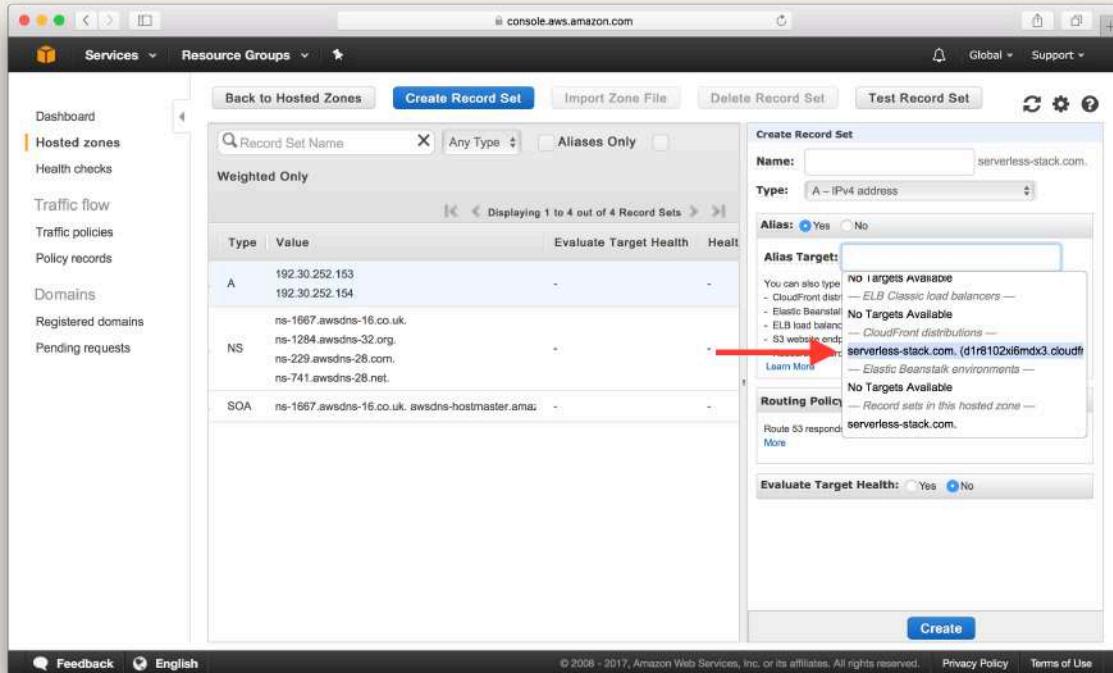
Leave name field empty screenshot

And select **Alias** as **Yes** since we are going to simply point this to our CloudFront domain.



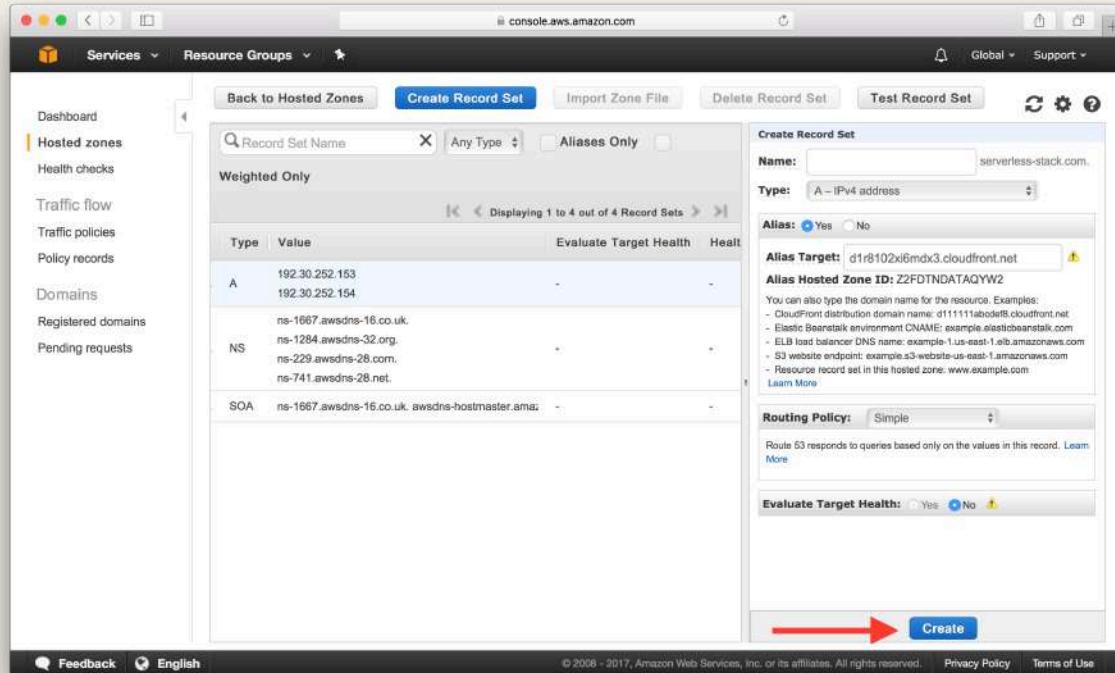
Set Alias to yes screenshot

In the **Alias Target** dropdown, select your CloudFront Distribution.



Select your CloudFront Distribution screenshot

Finally, hit **Create** to add your new record set.

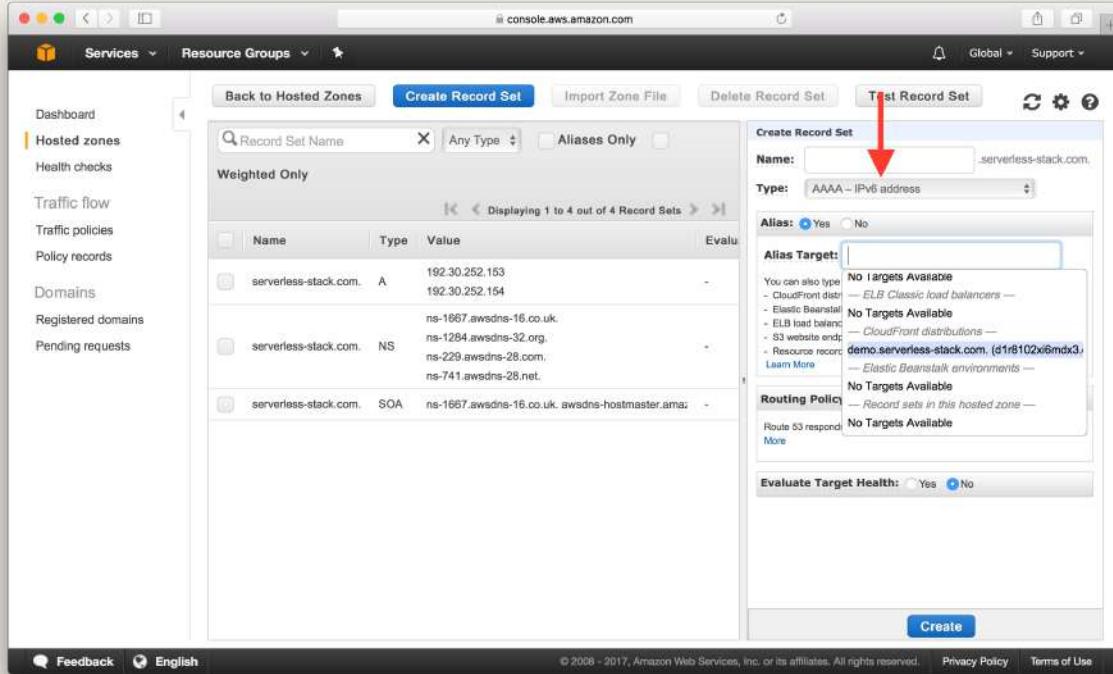


Select create to add record set screenshot

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.

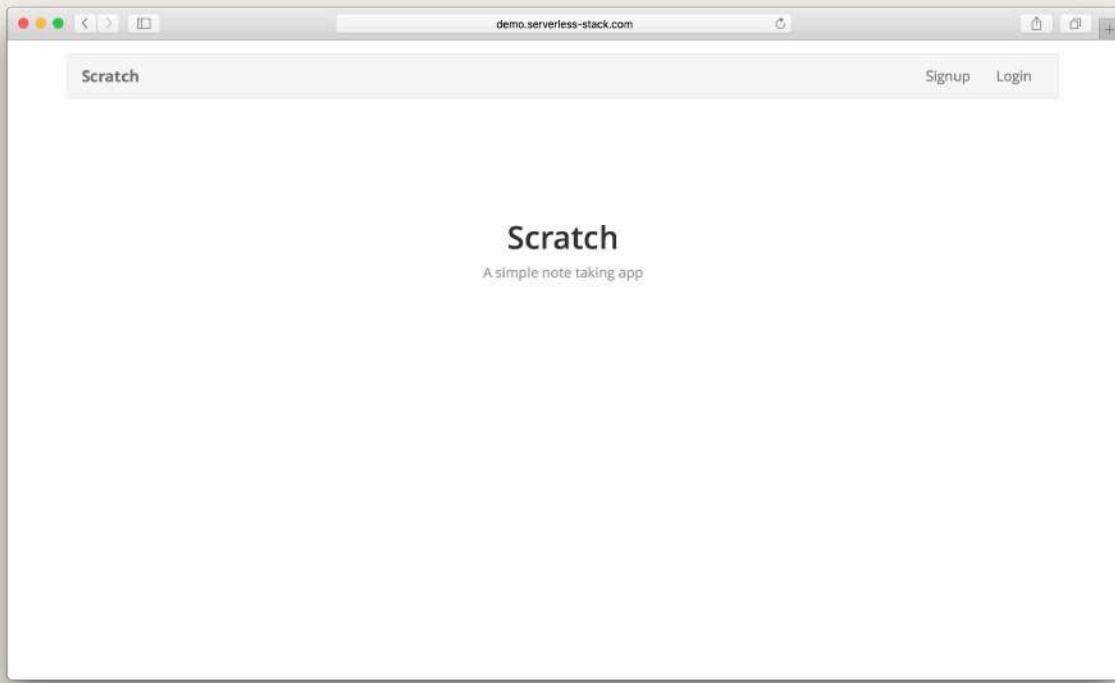
Create a new Record Set with the exact settings as before, except make sure to pick **AAAA - IPv6 address** as the **Type**.



Select AAAA IPv6 record set screenshot

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.



App live on new domain screenshot

Next up, we'll take a quick look at ensuring that our www. domain also directs to our app.



Help and discussion

View the [comments for this chapter on our forums](#)

Set up WWW Domain Redirect

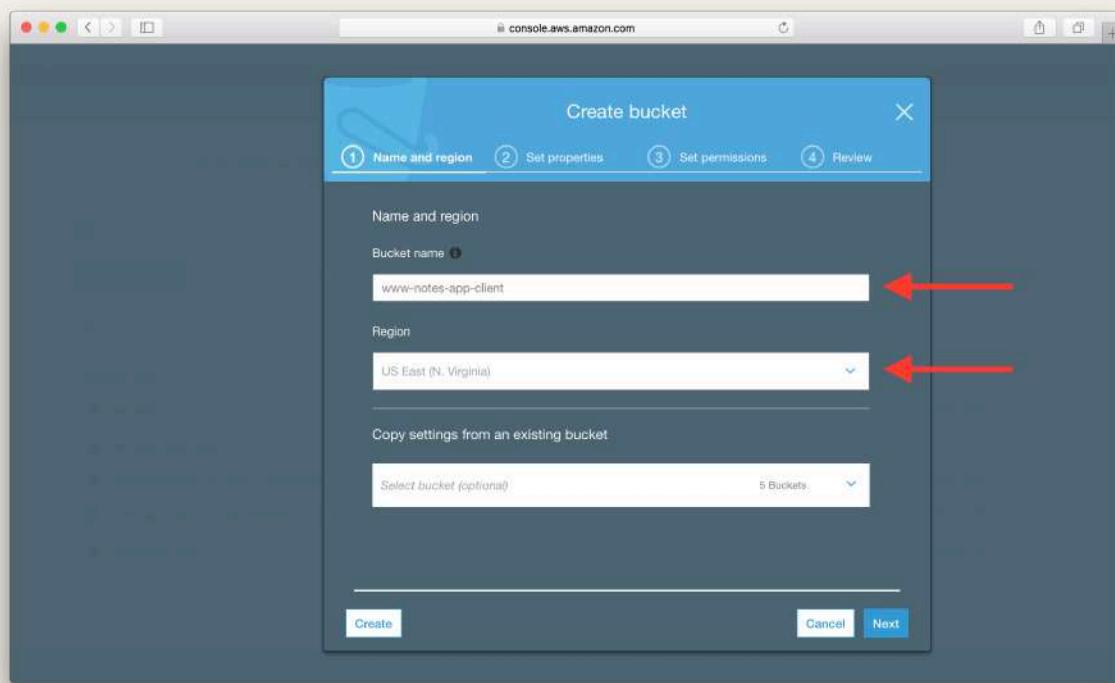
There's plenty of debate over the www vs non-www domains and while both sides have merit; we'll go over how to set up another domain (in this case the www) and redirect it to our original. The reason we do a redirect is to tell the search engines that we only want one version of our domain to appear 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.

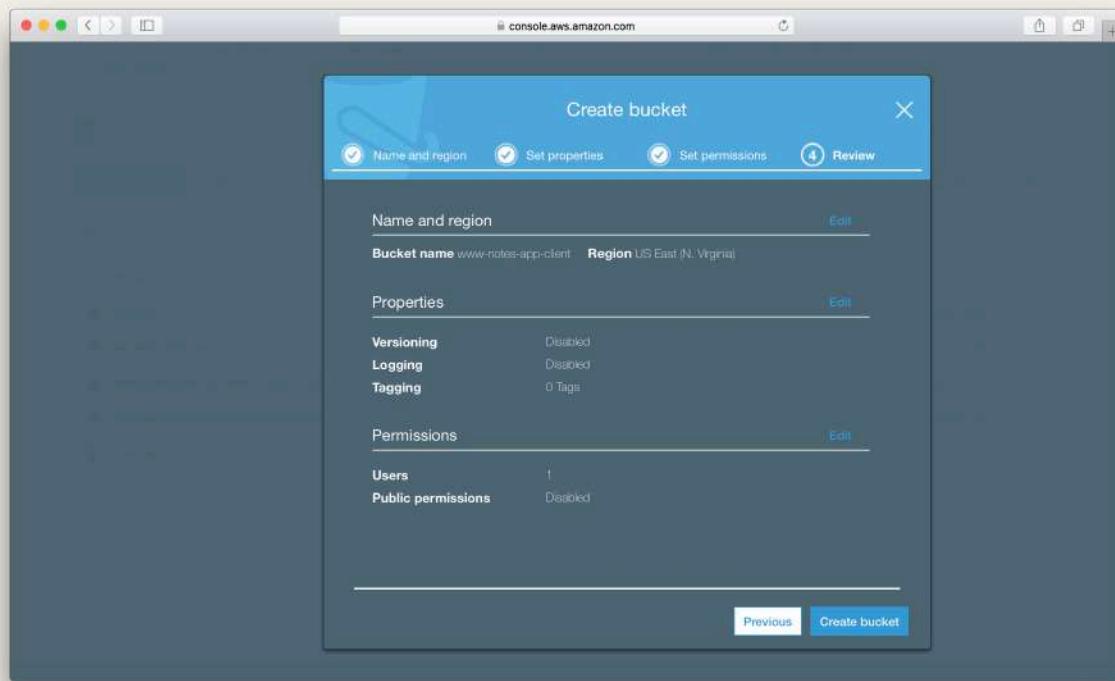
Create S3 Redirect Bucket

Create a **new S3 Bucket** through the [AWS Console](#). 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.



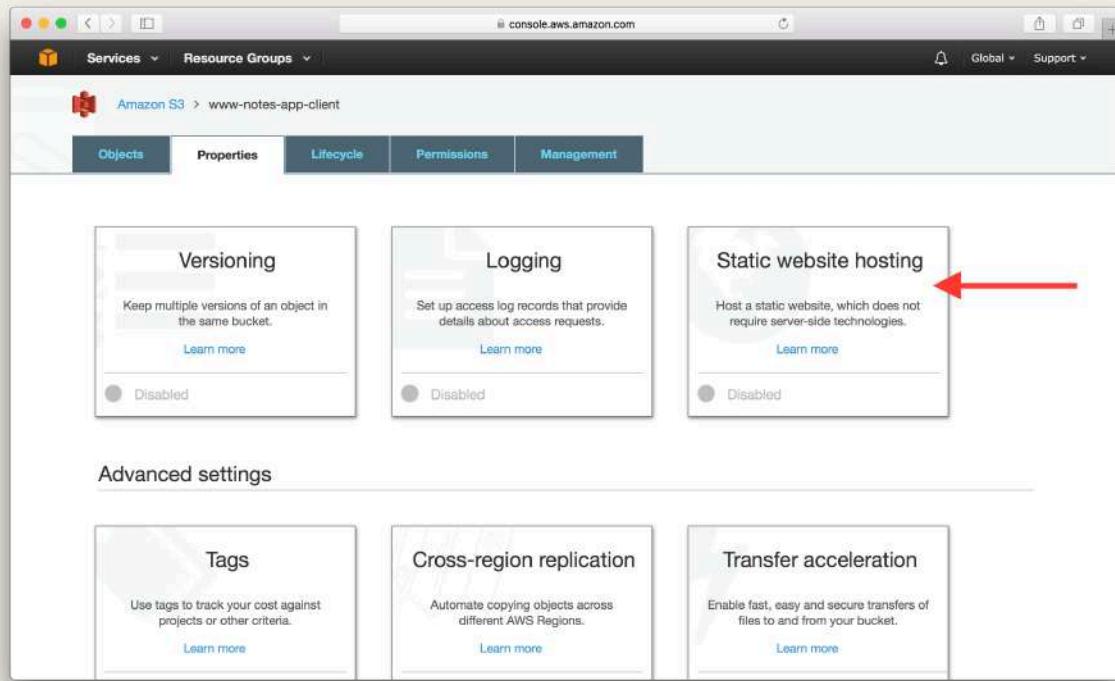
Create S3 Redirect Bucket screenshot

Next just follow through the steps and leave the defaults intact.



Use defaults to create S3 redirect bucket screenshot

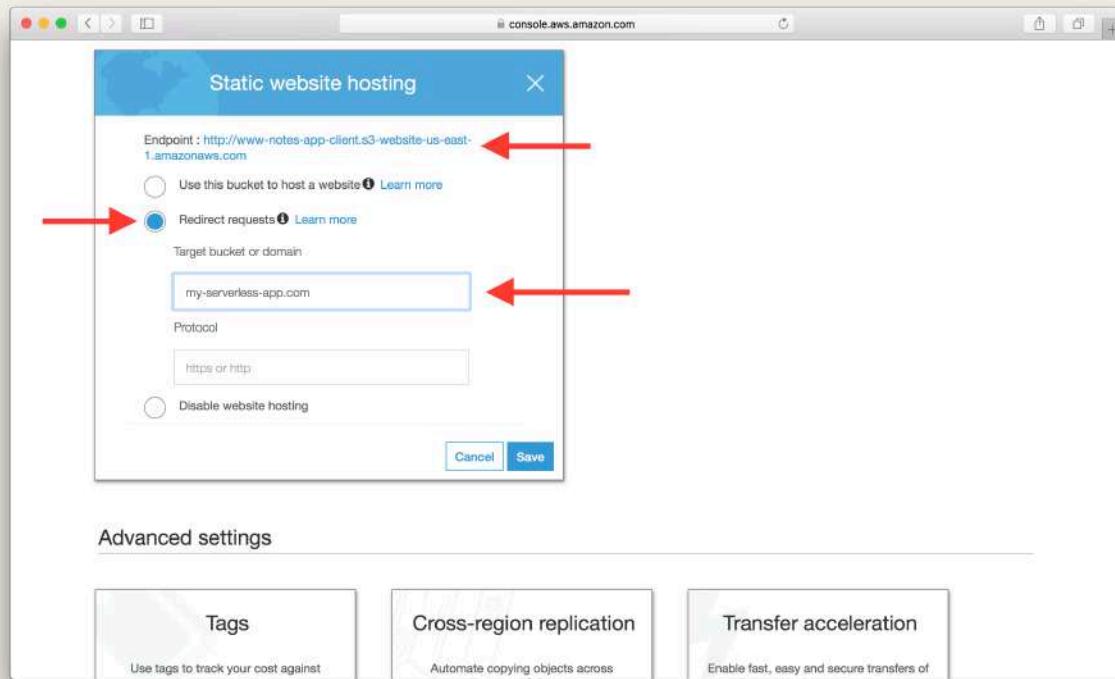
Now go into the **Properties** of the new bucket and click on the **Static website hosting**.



Select static website hosting screenshot

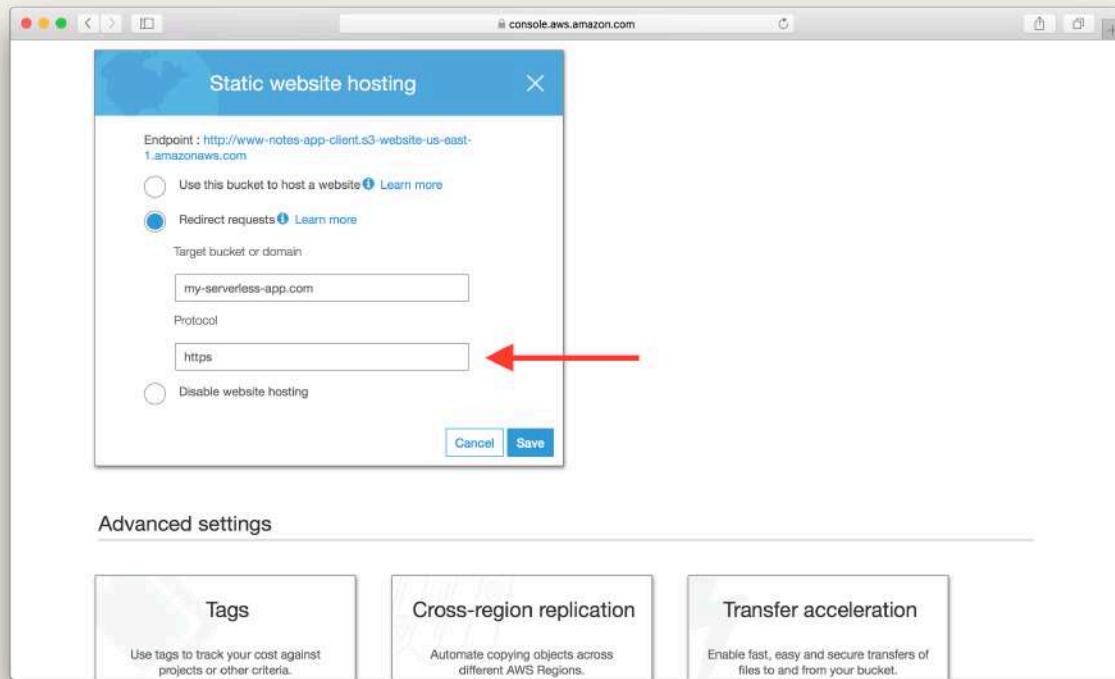
But unlike last time we are going to select the **Redirect requests** option and fill in the domain we are going to be redirecting towards. This is the domain that we set up in our last chapter.

Also, make sure to copy the **Endpoint** as we'll be needing this later.



Select redirect requests screenshot

Change the **Protocol** to **https** and hit **Save**.

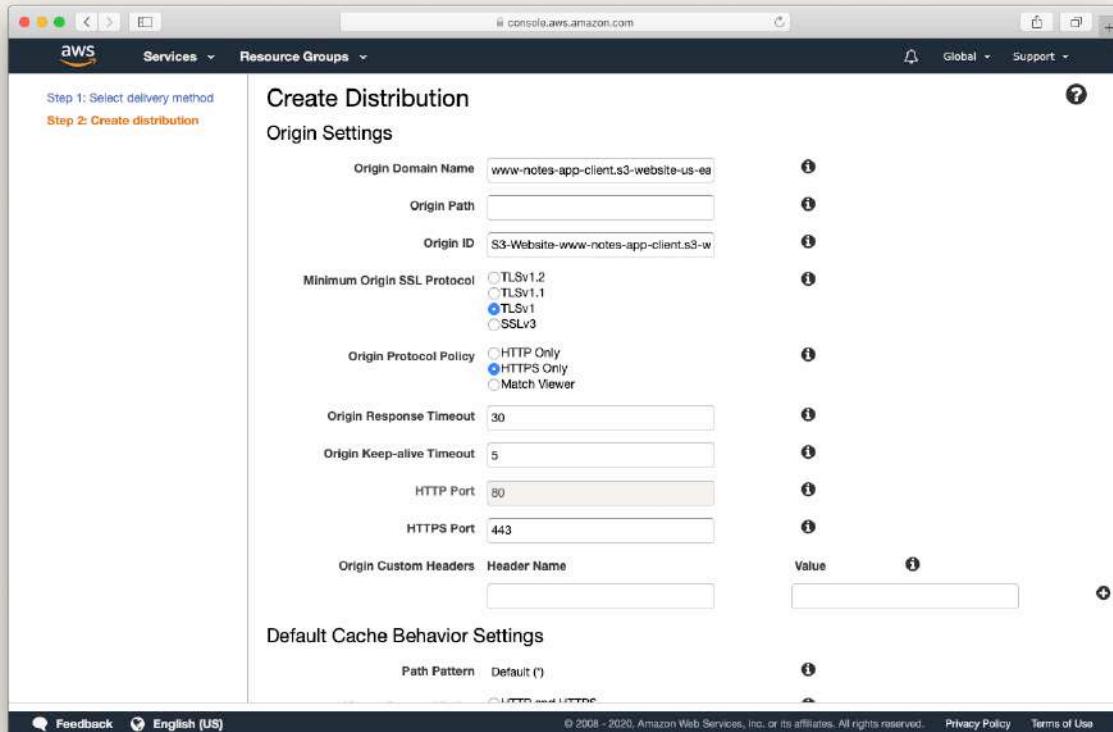


Change S3 Redirect to HTTPS screenshot

And hit **Save** to make the changes. Next we'll create a CloudFront Distribution to point to this S3 redirect Bucket.

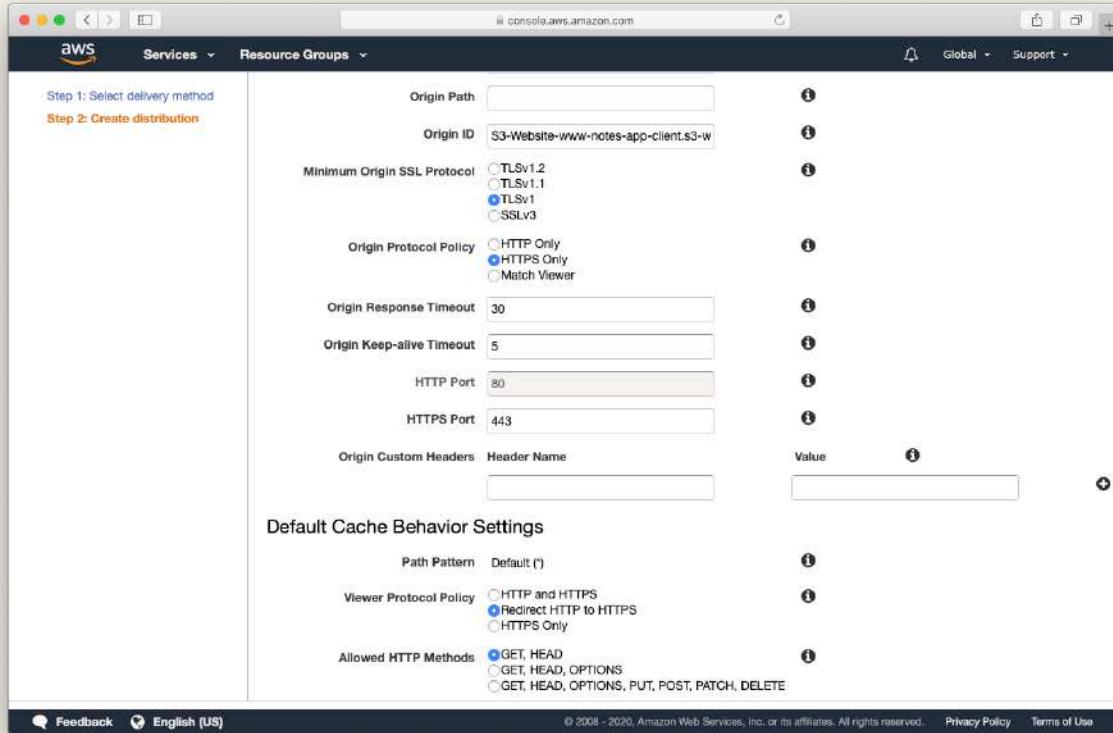
Create a CloudFront Distribution

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`. In addition, select **HTTPS Only** as the Protocol Policy.



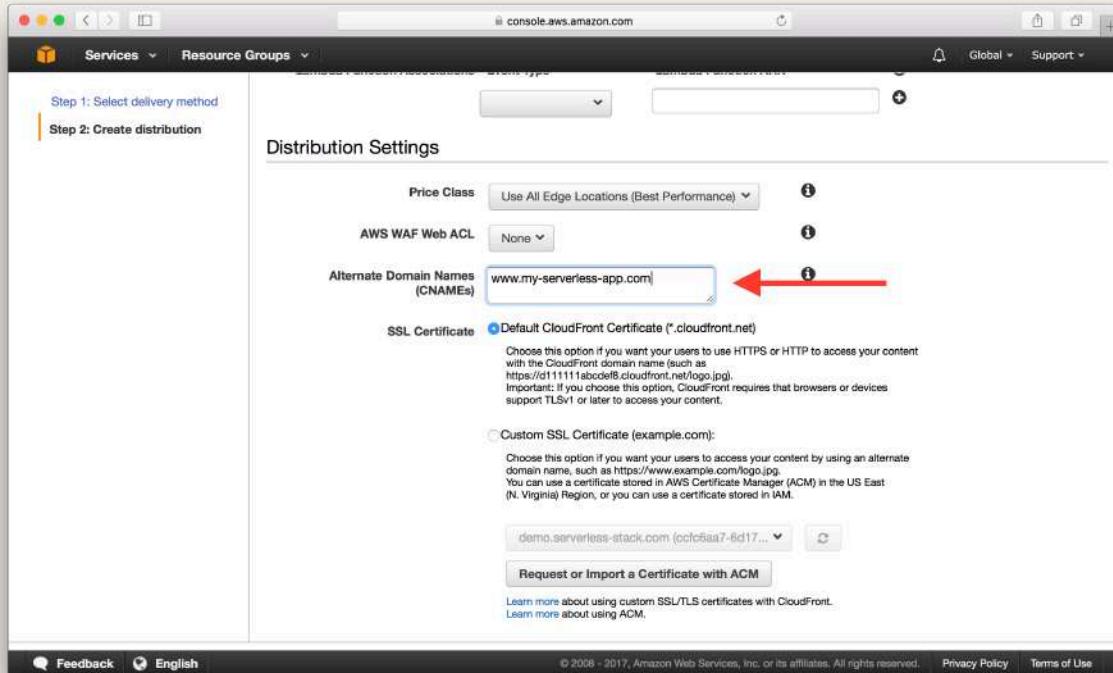
Set origin domain name and protocol policy screenshot

Set the Viewer Protocol Policy to **Redirect HTTP to HTTPS**.



Set viewer protocol policy screenshot

Next, scroll down to the **Alternate Domain Names (CNAMEs)** and use the www version of our domain name here.



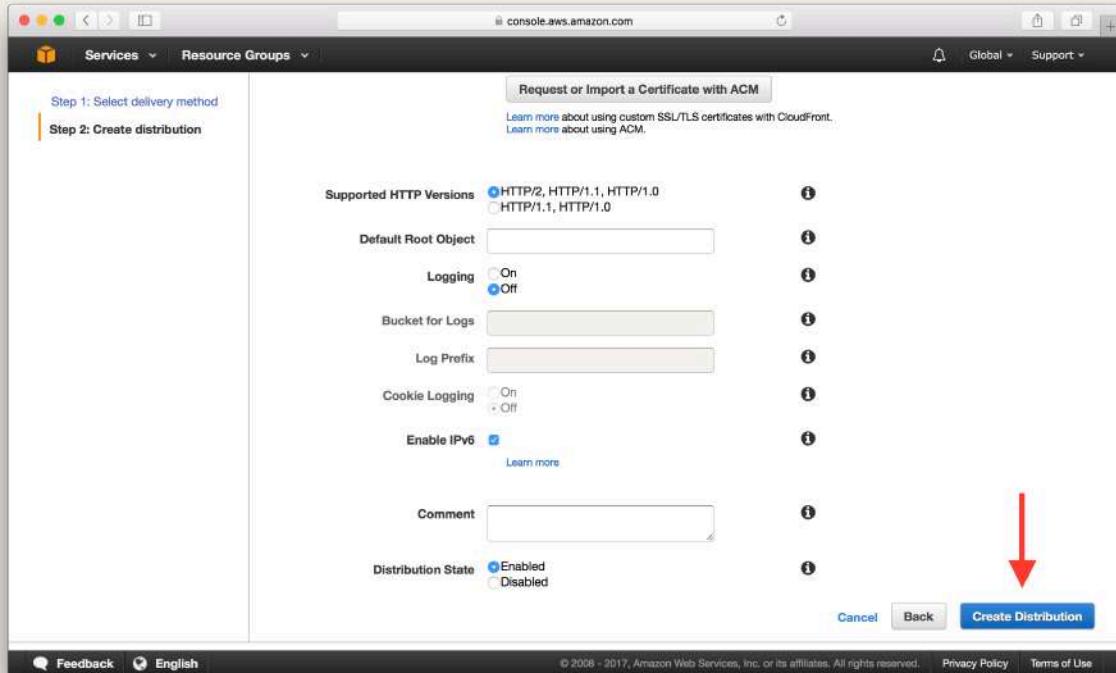
Set alternate domain name screenshot

As before, switch the **SSL Certificate** to **Custom SSL Certificate** and select the certificate we created from the drop down.

The screenshot shows the 'Distribution Settings' page in the AWS CloudFront console. The 'Alternate Domain Names (CNAMEs)' field contains 'www.my-serverless-app.com'. Under 'SSL Certificate', the 'Custom SSL Certificate (example.com):' option is selected, with 'demo.serverless-stack.com (93691abe-1)' listed. The 'Custom SSL Client Support' section is set to 'Clients that Support Server Name Indication (SNI) - (Recommended)'. At the bottom, there is a 'Request or Import a Certificate with ACM' button.

Select custom SSL certificate

Then hit **Create Distribution**.

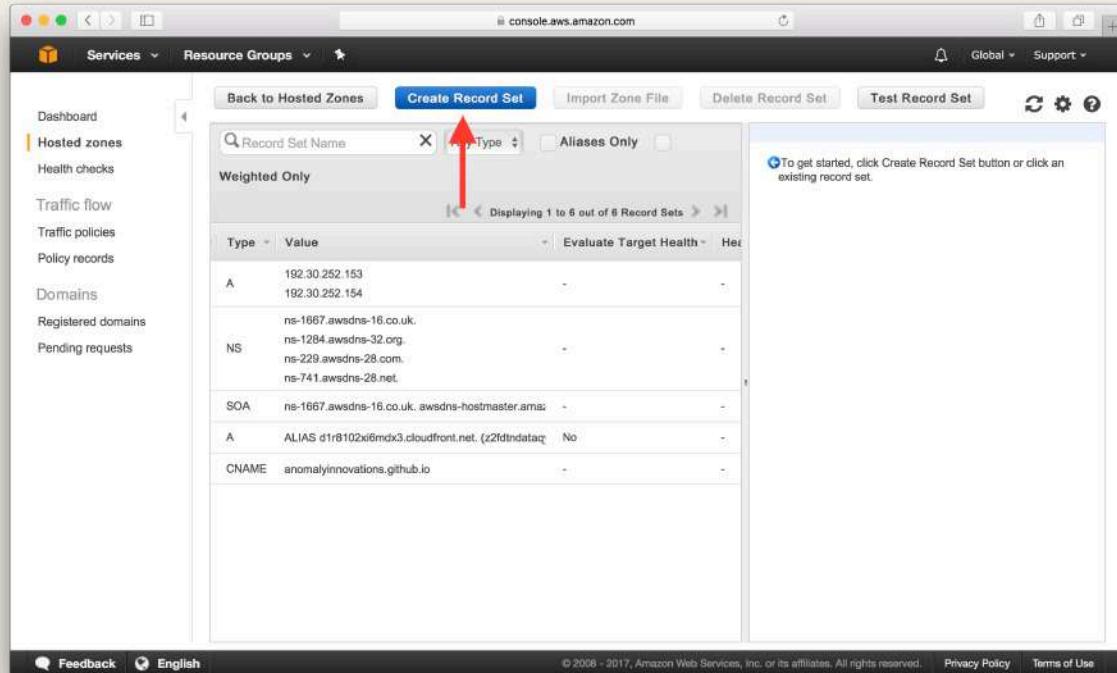


Hit create distribution screenshot

Finally, we'll point our www domain to this CloudFront Distribution.

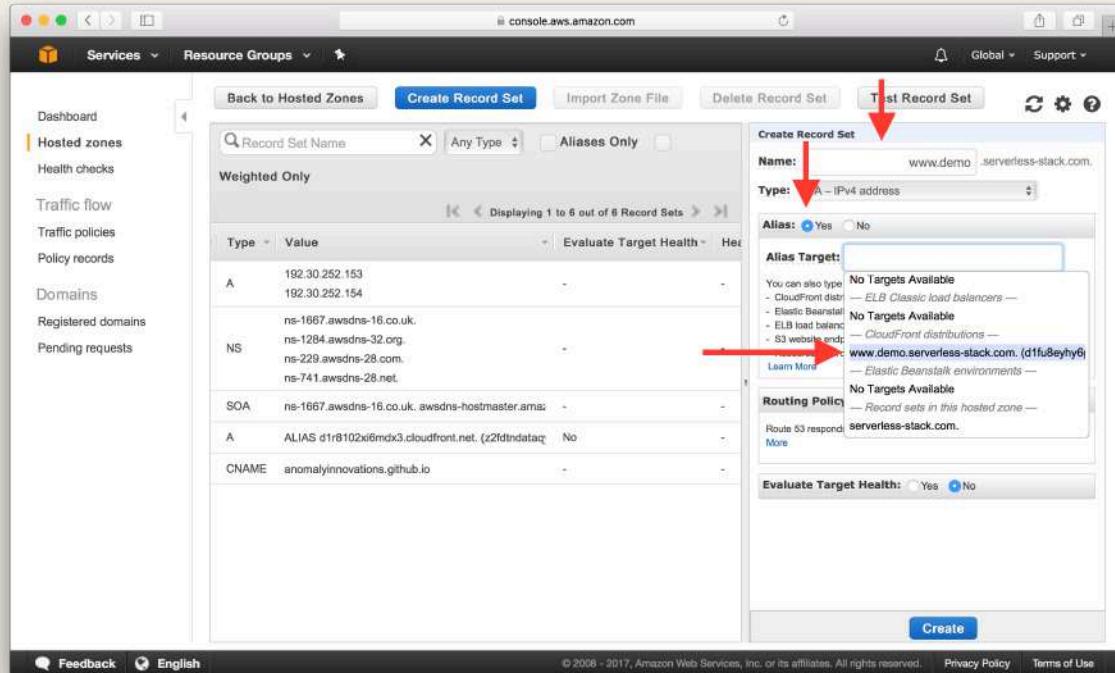
Point WWW Domain to CloudFront Distribution

Head over to your domain in Route 53 and hit **Create Record Set**.



Select Create Record Set screenshot

This time fill in **www** as the **Name** and select **Alias** as **Yes**. And pick your new CloudFront Distribution from the **Alias Target** dropdown.

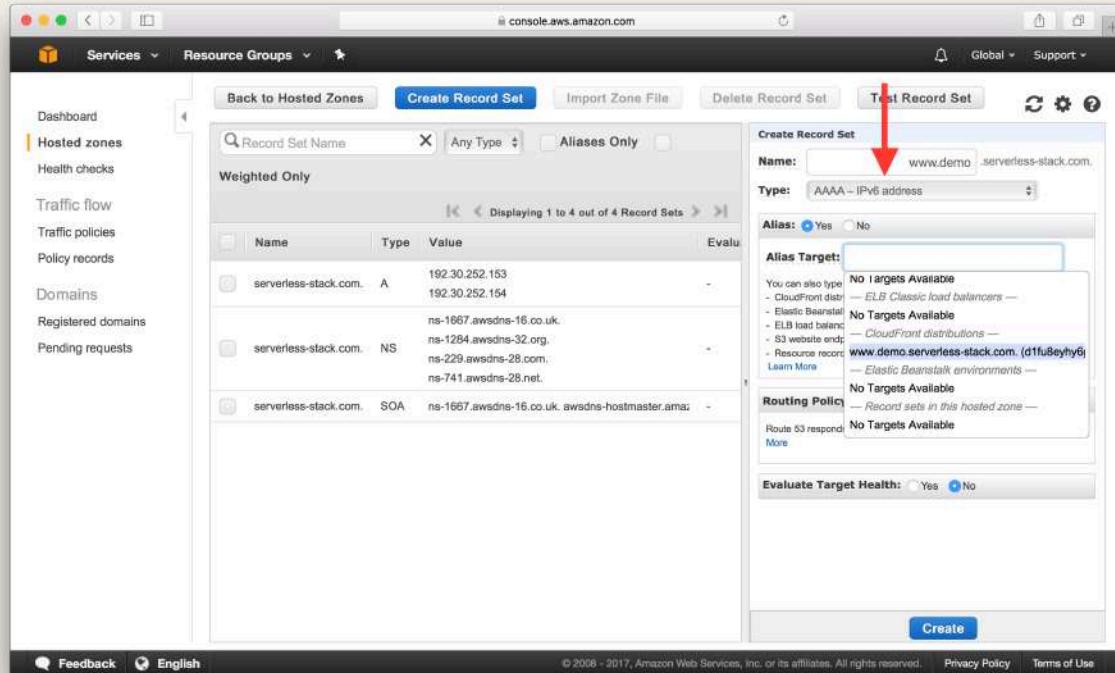


Fill in record set details screenshot

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**.



Fill in AAAA IPv6 record set details screenshot

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 up, let's look at the process of deploying updates to our app.



Help and discussion

View the [comments for this chapter on our forums](#)

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 its 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](#) chapter.

```
$ aws s3 sync build/ s3://YOUR_S3_DEPLOY_BUCKET_NAME --delete
```

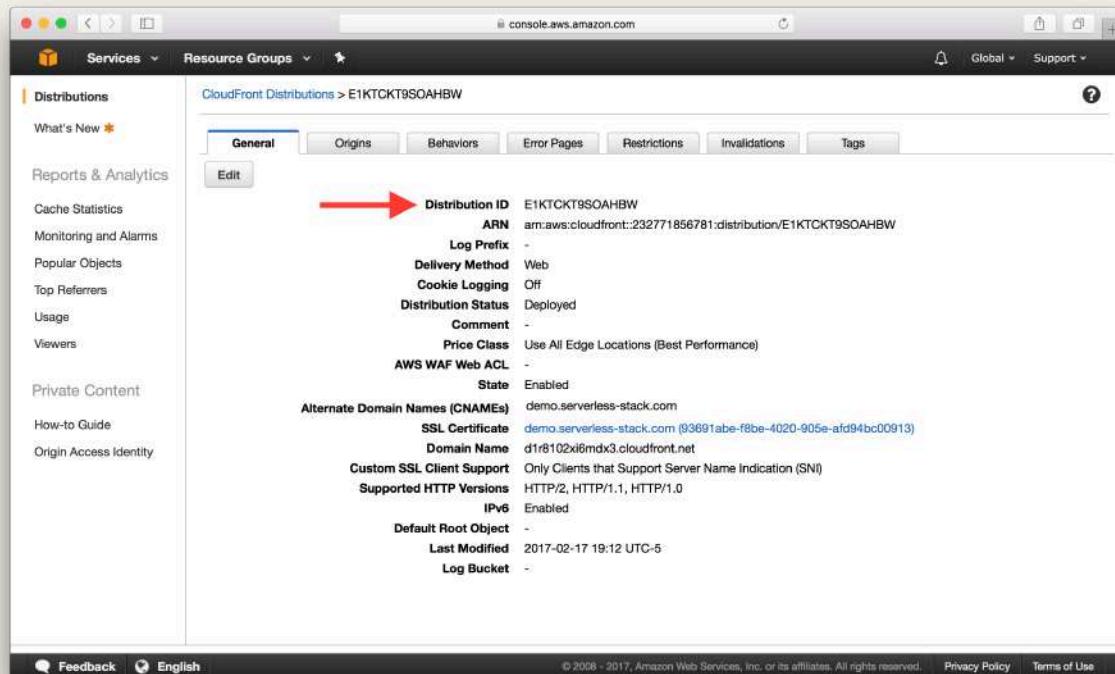
Note the `--delete` flag here; this is telling S3 to delete all the files that are in the bucket that we aren't uploading this time around. Create React App generates unique bundles when we build it and without this flag we'll end up retaining all the files from the previous builds. Our changes should 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.

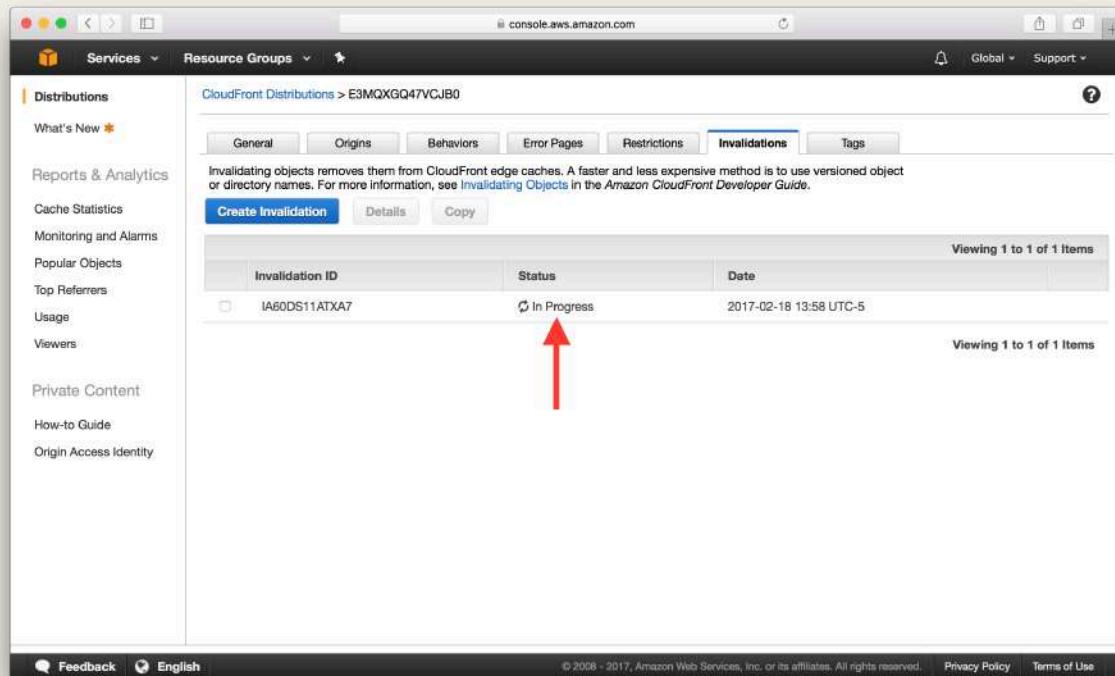


CloudFront Distributions ID screenshot

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.



CloudFront Invalidation in progress screenshot

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 (InvalidArgumentException) 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.



Help and discussion

View the [comments for this chapter on our forums](#)

Manage User Accounts in AWS Amplify

If you've followed along with the first part of [Serverless Stack](#) 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](#) as our authentication and user management provider. And on the frontend we are using [AWS Amplify](#) with our [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](#). For these chapters we are going to use a forked version of the notes app. You can [view the hosted version here](#) and the [source is available in a repo here](#).

Let's get started by allowing users to reset their password in case they have forgotten it.



Help and discussion

View the [comments for this chapter on our forums](#)



For reference, here is the code we are using

[User Management Frontend Source](#)

Handle Forgot and Reset Password

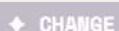
In our [Serverless notes app](#) we've used [Cognito User Pool](#) to sign up and login our users. In the frontend we've used [AWS Amplify](#) in our React app. However, if our users have forgotten their passwords, we need to have a way for them to reset their password. In this chapter we will look at 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>
- And can be accessed through: <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



We are going to create a `src/containers/ResetPassword.js`.

```
import React, { useState } 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 { useFormFields } from "../libs/hooksLib";
import { onError } from "../libs/errorLib";
import "./ResetPassword.css";
```

```
export default function ResetPassword() {
  const [fields, handleFieldChange] = useFormFields({
    code: "",
    email: "",
    password: "",
    confirmPassword: "",
  });
  const [codeSent, setCodeSent] = useState(false);
  const [confirmed, setConfirmed] = useState(false);
  const [isConfirming, setIsConfirming] = useState(false);
  const [isSendingCode, setIsSendingCode] = useState(false);

  function validateCodeForm() {
    return fields.email.length > 0;
  }

  function validateResetForm() {
    return (
      fields.code.length > 0 &&
      fields.password.length > 0 &&
      fields.password === fields.confirmPassword
    );
  }

  async function handleSendCodeClick(event) {
    event.preventDefault();

    setIsSendingCode(true);

    try {
      await Auth.forgotPassword(fields.email);
      setCodeSent(true);
    } catch (error) {
      onError(error);
      setIsSendingCode(false);
    }
  }
}
```

```
}

async function handleConfirmClick(event) {
  event.preventDefault();

  setIsConfirming(true);

  try {
    await Auth.forgotPasswordSubmit(
      fields.email,
      fields.code,
      fields.password
    );
    setConfirmed(true);
  } catch (error) {
    onError(error);
    setIsConfirming(false);
  }
}

function renderRequestCodeForm() {
  return (
    <form onSubmit={handleSendCodeClick}>
      <FormGroup bsSize="large" controlId="email">
        <ControlLabel>Email</ControlLabel>
        <FormControl
          autoFocus
          type="email"
          value={fields.email}
          onChange={handleFieldChange}
        />
      </FormGroup>
      <LoaderButton
        block
        type="submit"
        bsSize="large"
        isLoading={isSendingCode}
      >
        <span>Send Code</span>
      </LoaderButton>
    </form>
  );
}

function handleSendCodeClick() {
  const email = fields.email;
  if (!email) {
    return;
  }

  setConfirmed(false);
  setIsConfirming(true);

  Auth.forgotPasswordSubmit(
    email,
    fields.code,
    fields.password
  ).then(() => {
    setConfirmed(true);
    setIsConfirming(false);
    setTimeout(() => {
      setIsConfirming(false);
    }, 2000);
  }).catch((error) => {
    onError(error);
    setIsConfirming(false);
  });
}
```

```
        disabled={!validateCodeForm()}
```

```
>
```

```
    Send Confirmation
```

```
  </LoaderButton>
```

```
  </form>
```

```
);
```

```
}
```

```
function renderConfirmationForm() {
```

```
  return (
```

```
    <form onSubmit={handleConfirmClick}>
```

```
      <FormGroup bsSize="large" controlId="code">
```

```
        <ControlLabel>Confirmation Code</ControlLabel>
```

```
        <FormControl
```

```
          autoFocus
```

```
          type="tel"
```

```
          value={fields.code}
```

```
          onChange={handleFieldChange}
```

```
        />
```

```
        <HelpBlock>
```

```
          Please check your email ({fields.email}) for the confirmation code.
```

```
        </HelpBlock>
```

```
      </FormGroup>
```

```
      <hr />
```

```
      <FormGroup bsSize="large" controlId="password">
```

```
        <ControlLabel>New Password</ControlLabel>
```

```
        <FormControl
```

```
          type="password"
```

```
          value={fields.password}
```

```
          onChange={handleFieldChange}
```

```
        />
```

```
      </FormGroup>
```

```
      <FormGroup bsSize="large" controlId="confirmPassword">
```

```
        <ControlLabel>Confirm Password</ControlLabel>
```

```
        <FormControl
```

```
          type="password"
```

```
          value={fields.confirmPassword}
```

```
        onChange={handleFieldChange}
      />
    </FormGroup>
    <LoaderButton
      block
      type="submit"
      bsSize="large"
      isLoading={isConfirming}
      disabled={!validateResetForm()}
    >
      Confirm
    </LoaderButton>
  </form>
);
}

function 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>
  );
}

return (
  <div className="ResetPassword">
    {!codeSent
      ? renderRequestCodeForm()
      : !confirmed
      ? renderConfirmationForm()
      : 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 `renderRequestCodeForm()`.
- Once the user submits this form, we start the process by calling `Auth.forgotPassword(fields.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 `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(fields.email, fields.code, fields.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 exact path="/login/reset">
  <ResetPassword />
</UnauthenticatedRoute>
```

◆ 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>
```

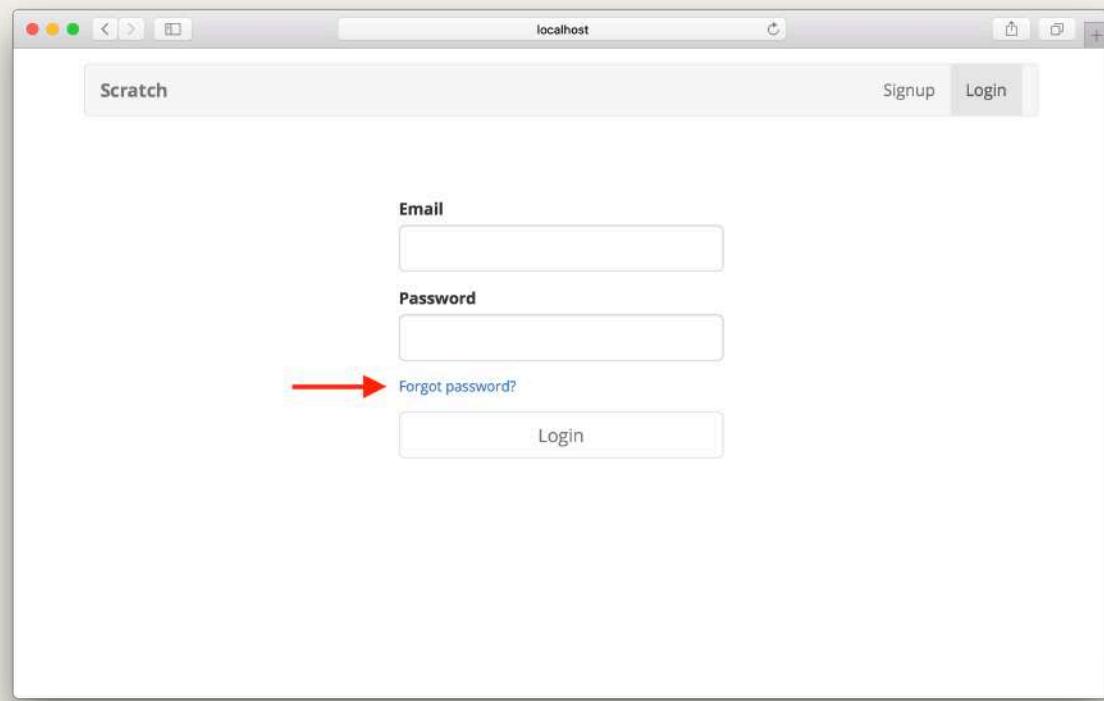
◆ CHANGE And import the `Link` component in the header.

```
import { Link } from "react-router-dom";
```

◆ CHANGE And finally add some style to the link by adding the following to src/containers/Login.css

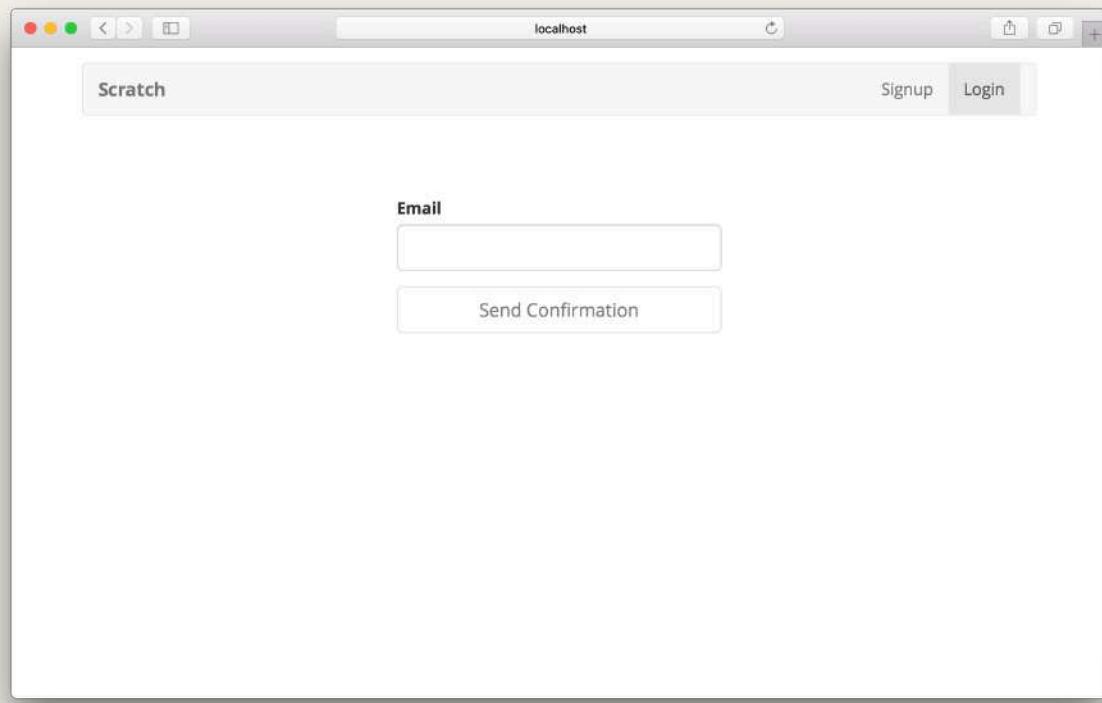
```
.Login form a {  
  margin-bottom: 15px;  
  display: block;  
  font-size: 14px;  
}
```

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.



Login page forgot password link screenshot

And from there they can put in their email to reset their password.



Forgot password page screenshot

Next, let's look at how our logged in users can change their password.



Help and discussion

View the [comments](#) for this chapter on our forums



For reference, here is the code we are using

[User Management Frontend Source](#)

Allow Users to Change Passwords

For our [Serverless notes app](#), 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>
- And it can be accessed through: <https://demo-user-mgmt.serverless-stack.com>

Let's start by editing our settings page so that our users can use to change their password.

Add a Settings Page



Replace the `return` statement in `src/containers/Settings.js` with.

```
return (
  <div className="Settings">
    <LinkContainer to="/settings/email">
      <LoaderButton block bsSize="large">
        Change Email
      </LoaderButton>
    </LinkContainer>
    <LinkContainer to="/settings/password">
      <LoaderButton block bsSize="large">
        Change Password
      </LoaderButton>
    </LinkContainer>
    <hr />
    <StripeProvider stripe={stripe}>
      <Elements>
```

```
<Elements>
</StripeProvider>
</div>
);
```

◆ CHANGE And import the following as well.

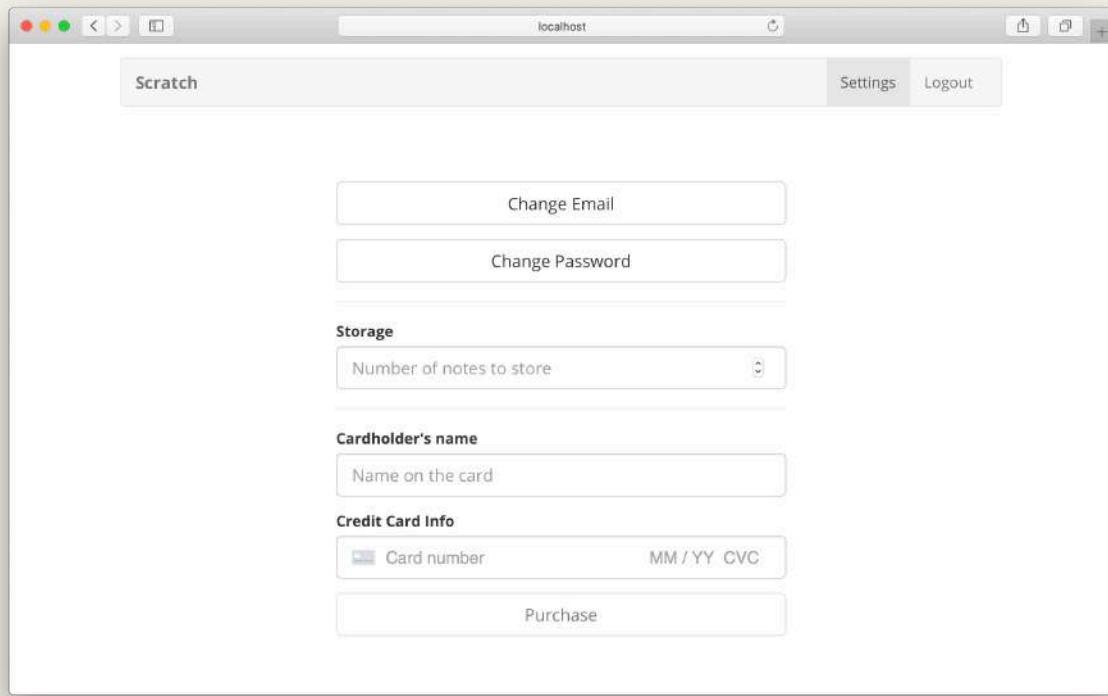
```
import { LinkContainer } from "react-router-bootstrap";
import LoaderButton from "../components/LoaderButton";
```

All this does is add two links to a page that allows our users to change their password and email.

◆ CHANGE Replace our src/containers/Settings.css with the following.

```
@media all and (min-width: 480px) {
  .Settings {
    padding: 60px 0;
    margin: 0 auto;
    max-width: 480px;
  }

  .Settings > .LoaderButton:first-child {
    margin-bottom: 15px;
  }
}
```



Settings page screenshot

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, { useState } from "react";
import { Auth } from "aws-amplify";
import { useHistory } from "react-router-dom";
import { FormGroup, FormControl, ControlLabel } from "react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import { useFormFields } from "../libs/hooksLib";
import { onError } from "../libs/errorLib";
import "./ChangePassword.css";

export default function ChangePassword() {
```

```
const history = useHistory();
const [fields, handleFieldChange] = useFormFields({
  password: '',
  oldPassword: '',
  confirmPassword: '',
});
const [isChanging, setIsChanging] = useState(false);

function validateForm() {
  return (
    fields.oldPassword.length > 0 &&
    fields.password.length > 0 &&
    fields.password === fields.confirmPassword
  );
}

async function handleClick(event) {
  event.preventDefault();

  setIsChanging(true);

  try {
    const currentUser = await Auth.currentAuthenticatedUser();
    await Auth.changePassword(
      currentUser,
      fields.oldPassword,
      fields.password
    );
    history.push("/settings");
  } catch (error) {
    onError(error);
    setIsChanging(false);
  }
}

return (

```

```
<div className="ChangePassword">
  <form onSubmit={handleChangeClick}>
    <FormGroup bsSize="large" controlId="oldPassword">
      <ControlLabel>Old Password</ControlLabel>
      <FormControl
        type="password"
        onChange={handleFieldChange}
        value={fields.oldPassword}
      />
    </FormGroup>
    <hr />
    <FormGroup bsSize="large" controlId="password">
      <ControlLabel>New Password</ControlLabel>
      <FormControl
        type="password"
        onChange={handleFieldChange}
        value={fields.password}
      />
    </FormGroup>
    <FormGroup bsSize="large" controlId="confirmPassword">
      <ControlLabel>Confirm Password</ControlLabel>
      <FormControl
        type="password"
        onChange={handleFieldChange}
        value={fields.confirmPassword}
      />
    </FormGroup>
    <LoaderButton
      block
      type="submit"
      bsSize="large"
      disabled={!validateForm()}
      isLoading={isChanging}
    >
      Change Password
    </LoaderButton>
  </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,
  fields.oldPassword,
  fields.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;
  }
}
```

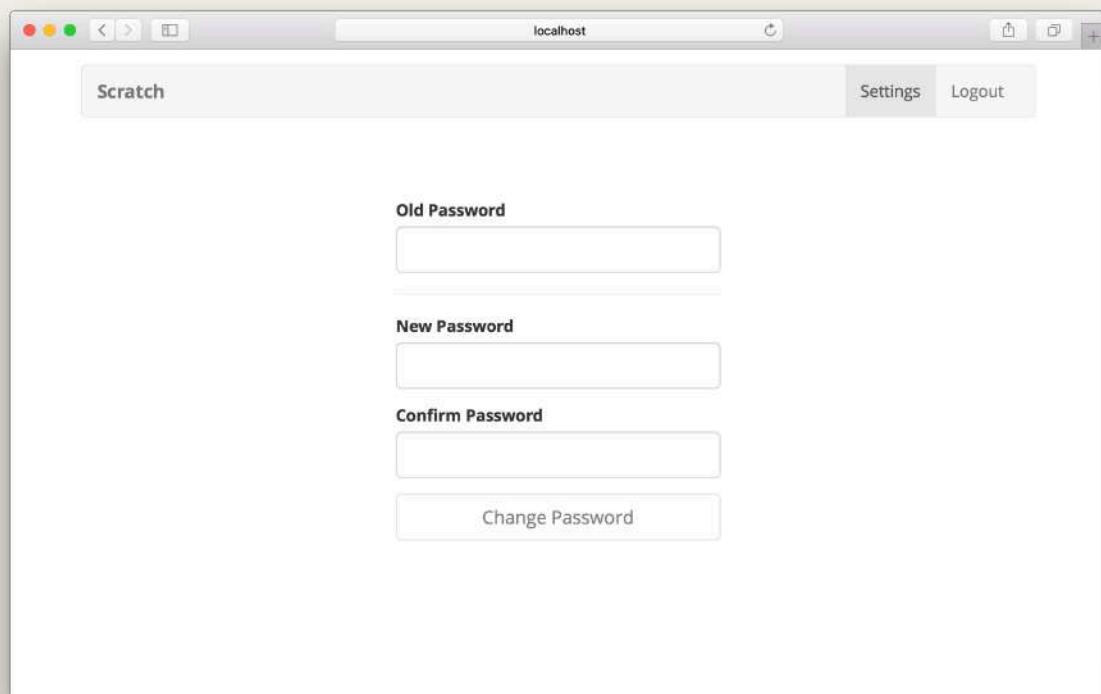
◆ CHANGE Let's add our new page to src/Routes.js.

```
<AuthenticatedRoute exact path="/settings/password">
  <ChangePassword />
</AuthenticatedRoute>
```

◆ **CHANGE** And import it.

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

That should do it. The /settings/password page should allow us to change our password.



Change password page screenshot

Next, let's look at how to implement a change email form for our users.



Help and discussion

View the [comments](#) for this chapter on our forums



For reference, here is the code we are using

[User Management Frontend Source](#)

Allow Users to Change Their Email

We want the users of our [Serverless notes app](#) 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>
- And it can be accessed through: <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, { useState } from "react";
import { Auth } from "aws-amplify";
import { useHistory } from "react-router-dom";
import {
  HelpBlock,
  FormGroup,
  FormControl,
  ControlLabel,
} from "react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import { useFormFields } from "../libs/hooksLib";
import { onError } from "../libs/errorLib";
import "./ChangeEmail.css";
```

```
export default function ChangeEmail() {
  const history = useHistory();
  const [codeSent, setCodeSent] = useState(false);
  const [fields, handleFieldChange] = useFormFields({
    code: '',
    email: '',
  });
  const [isConfirming, setIsConfirming] = useState(false);
  const [isSendingCode, setIsSendingCode] = useState(false);

  function validateEmailForm() {
    return fields.email.length > 0;
  }

  function validateConfirmForm() {
    return fields.code.length > 0;
  }

  async function handleUpdateClick(event) {
    event.preventDefault();

    setIsSendingCode(true);

    try {
      const user = await Auth.currentAuthenticatedUser();
      await Auth.updateUserAttributes(user, { email: fields.email });
      setCodeSent(true);
    } catch (error) {
      onError(error);
      setIsSendingCode(false);
    }
  }

  async function handleConfirmClick(event) {
    event.preventDefault();

    setIsConfirming(true);
```

```
try {
    await Auth.verifyCurrentUserAttributeSubmit("email", fields.code);

    history.push("/settings");
} catch (error) {
    onError(error);
    setIsConfirming(false);
}
}

function renderUpdateForm() {
    return (
        <form onSubmit={handleUpdateClick}>
            <FormGroup bsSize="large" controlId="email">
                <ControlLabel>Email</ControlLabel>
                <FormControl
                    autoFocus
                    type="email"
                    value={fields.email}
                    onChange={handleFieldChange}
                />
            </FormGroup>
            <LoaderButton
                block
                type="submit"
                bsSize="large"
                isLoading={isSendingCode}
                disabled={!validateEmailForm()}
            >
                Update Email
            </LoaderButton>
        </form>
    );
}

function renderConfirmationForm() {
```

```
return (
  <form onSubmit={handleConfirmClick}>
    <FormGroup bsSize="large" controlId="code">
      <ControlLabel>Confirmation Code</ControlLabel>
      <FormControl
        autoFocus
        type="tel"
        value={fields.code}
        onChange={handleFieldChange}
      />
      <HelpBlock>
        Please check your email ({fields.email}) for the confirmation code.
      </HelpBlock>
    </FormGroup>
    <LoaderButton
      block
      type="submit"
      bsSize="large"
      isLoading={isConfirming}
      disabled={!validateConfirmForm()}
    >
      Confirm
    </LoaderButton>
  </form>
);
}

return (
  <div className="ChangeEmail">
    {!codeSent ? renderUpdateForm() : 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 `renderUpdateForm()`. Once the user submits this form, we call:

```
const user = await Auth.currentAuthenticatedUser();
Auth.updateUserAttributes(user, { email: fields.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 `renderConfirmationForm()`. Upon submitting this form we call:

```
Auth.verifyCurrentUserAttributeSubmit("email", fields.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;
  }
}
```

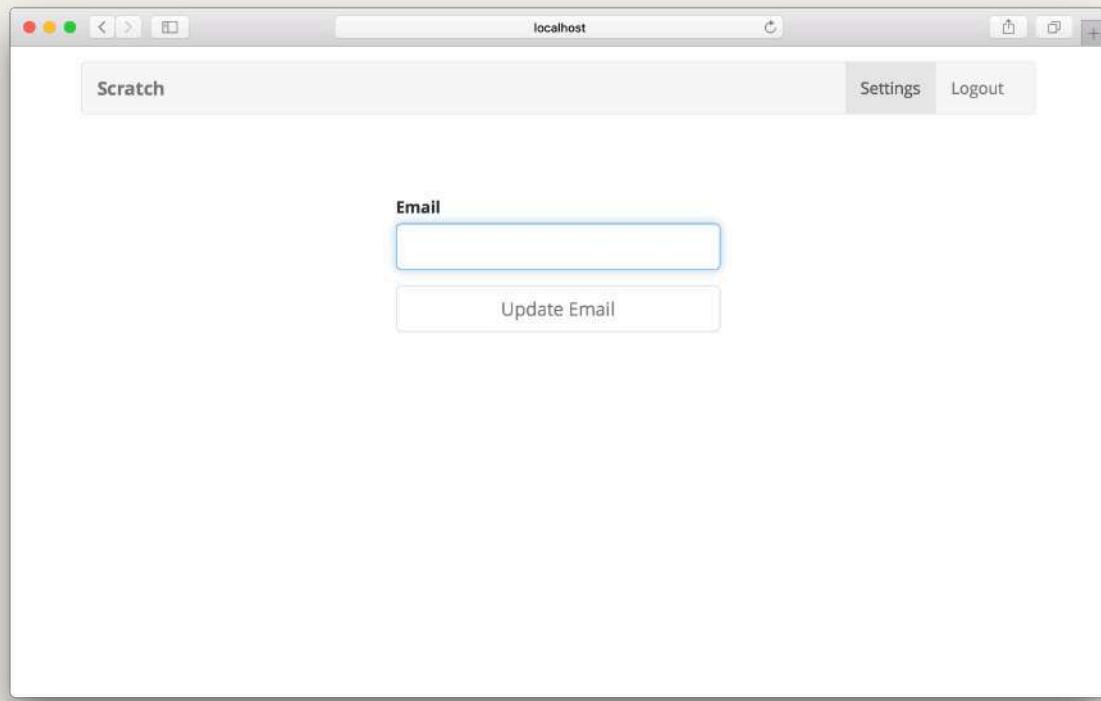
◆ CHANGE Finally, let's add our new page to `src/Routes.js`.

```
<AuthenticatedRoute exact path="/settings/email">
  <ChangeEmail />
</AuthenticatedRoute>
```

◆ 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.



Change email page screenshot

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", fields.code)`.

This can make your change email flow more robust and handle the case where a user forgets to verify their new email.

**Help and discussion**

View the [comments](#) for this chapter on our forums

**For reference, here is the code we are using**

[User Management Frontend Source](#)

Facebook Login with Cognito using AWS Amplify

In our guide so far we have used the [Cognito User Pool](#) to sign up users to our [demo notes app](#). 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 :

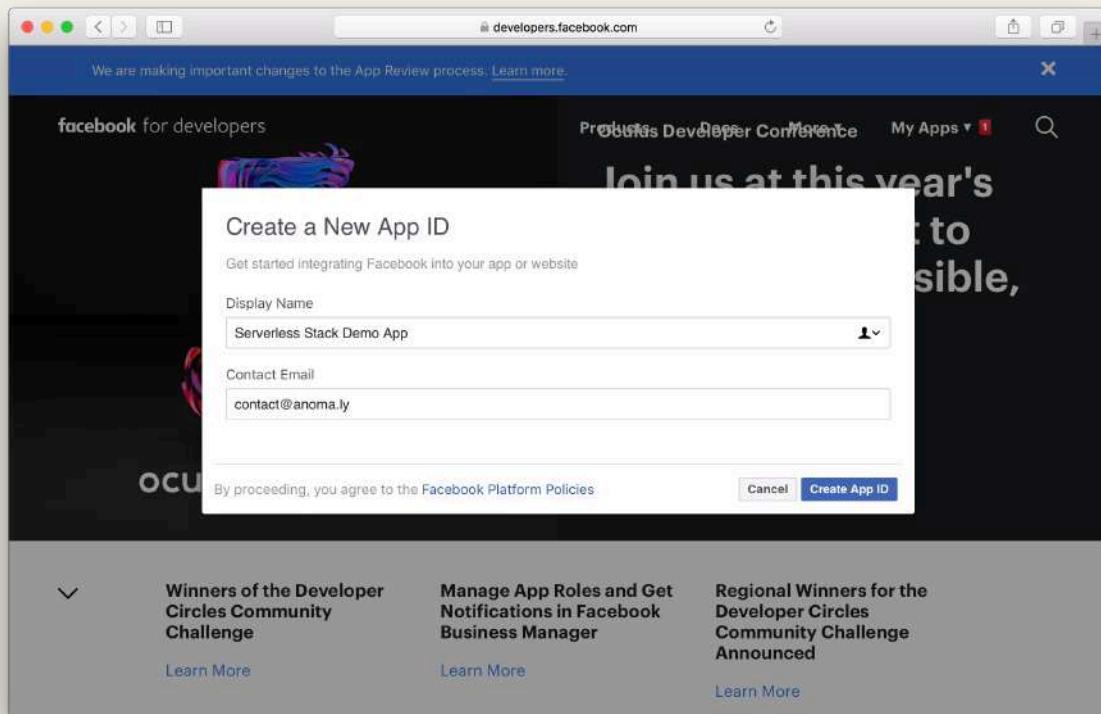
- A separate GitHub repository: <https://github.com/AnomalyInnovations/serverless-stack-demo-fb-login-client>
- And can be accessed through: <https://demo-fb-login.serverless-stack.com>

The main ideas and code for this chapter have been contributed by our long time reader and contributor [Peter Eman Paver Abastillas](#).

To get started let's create a Facebook app that our users will use to login.

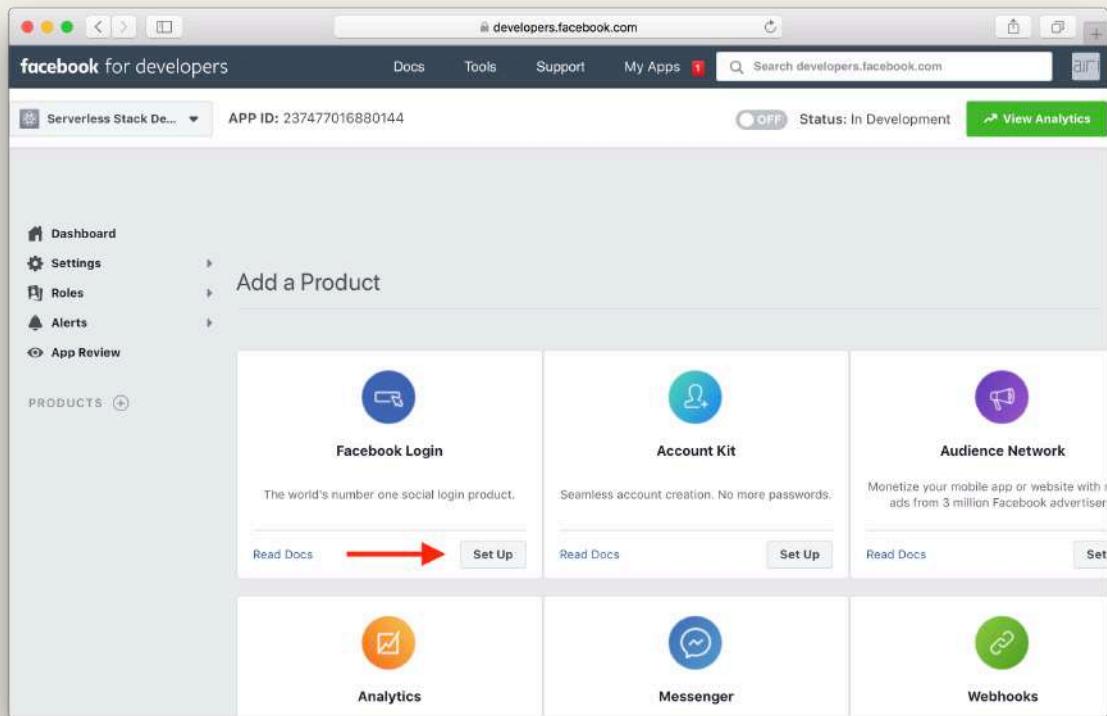
Creating a Facebook App

Head over to <http://developers.facebook.com/> and create a new app by clicking **My Apps > Add New App**.



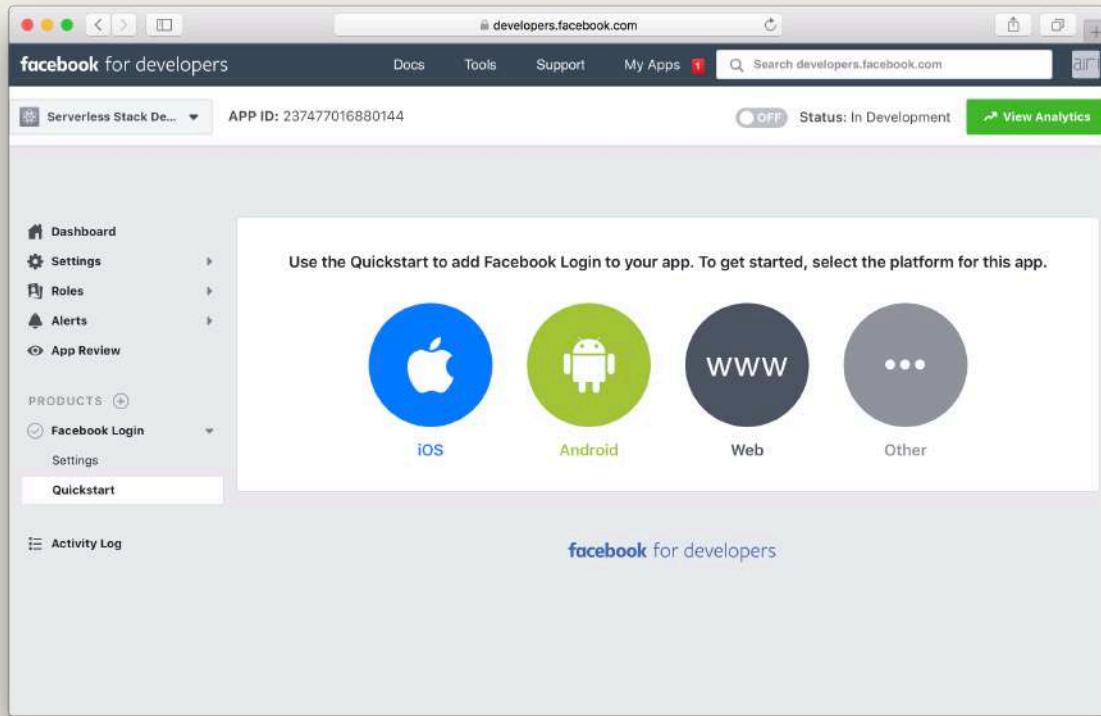
Create a Facebook app screenshot

Under **Facebook Login**, select **Set Up**.



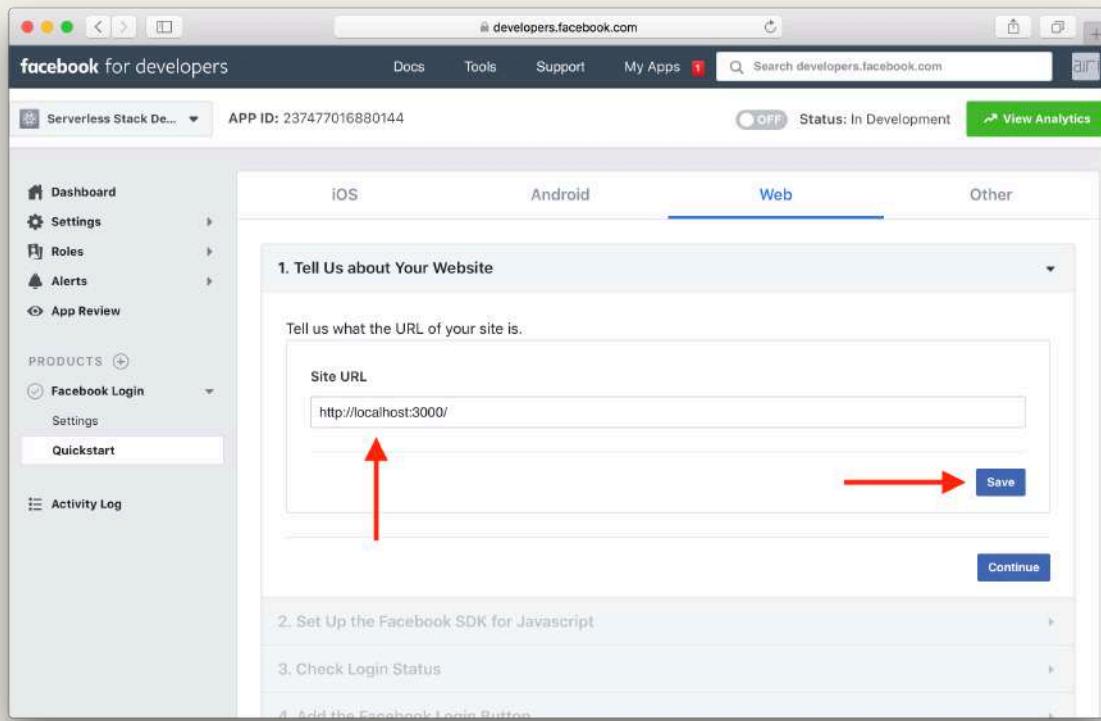
Select Facebook Login screenshot

And select **Web**.



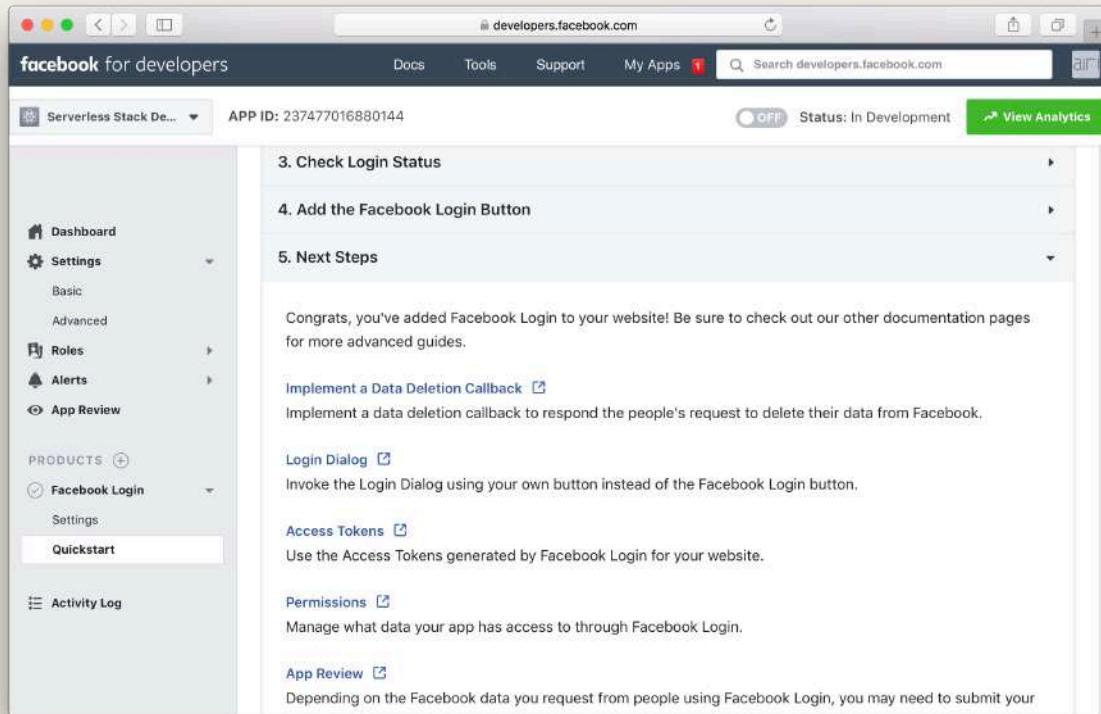
Select Web option for Login screenshot

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](#). Hit **Save**.



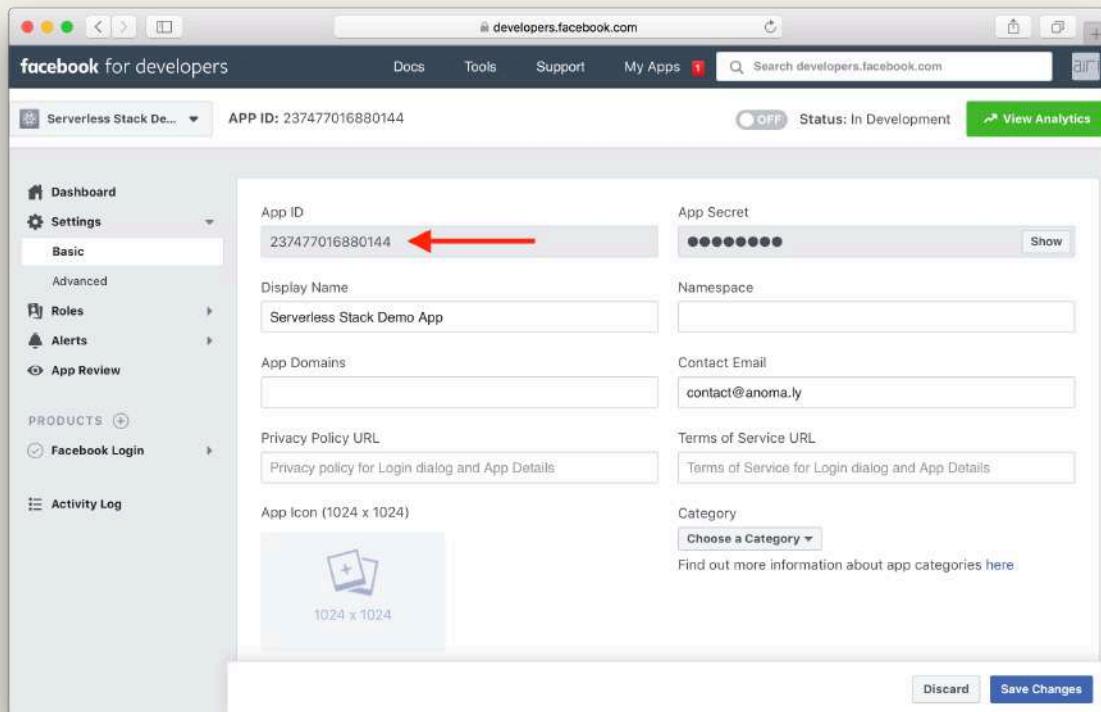
Set Website URL screenshot

You can hit **Continue** to go through the rest of the Quickstart.



Complete Quickstart screenshot

Finally, head over to **Settings > Basic** and make a note of your **App ID**.



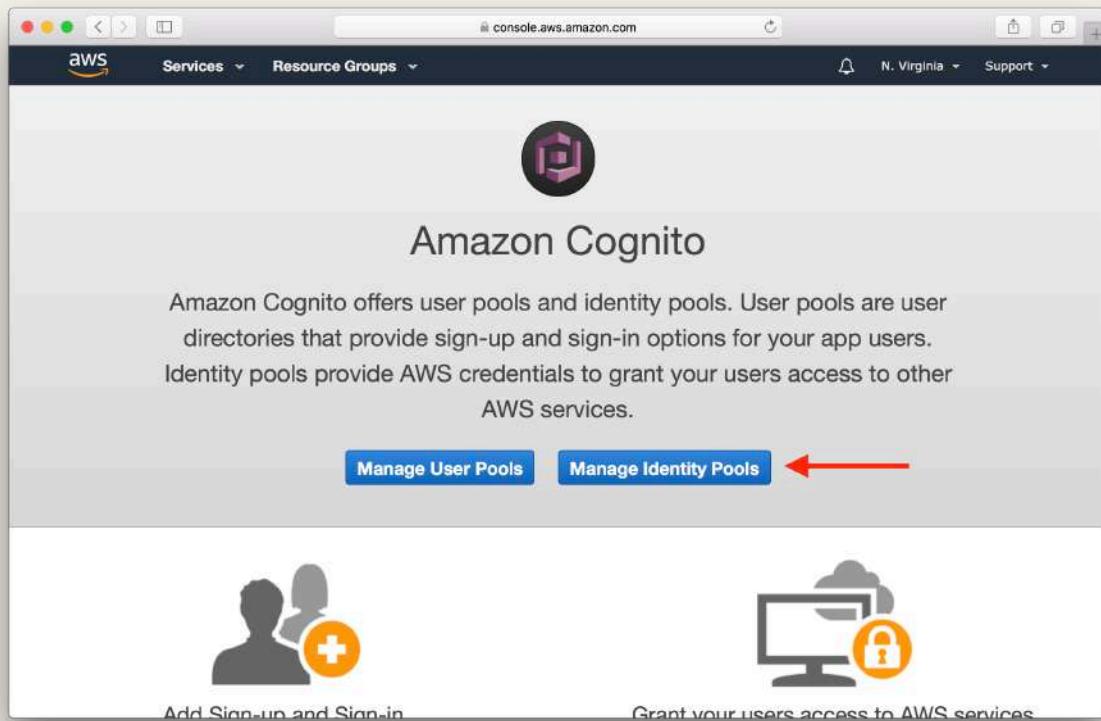
Copy App ID from Settings screenshot

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](#) 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](#) chapter.

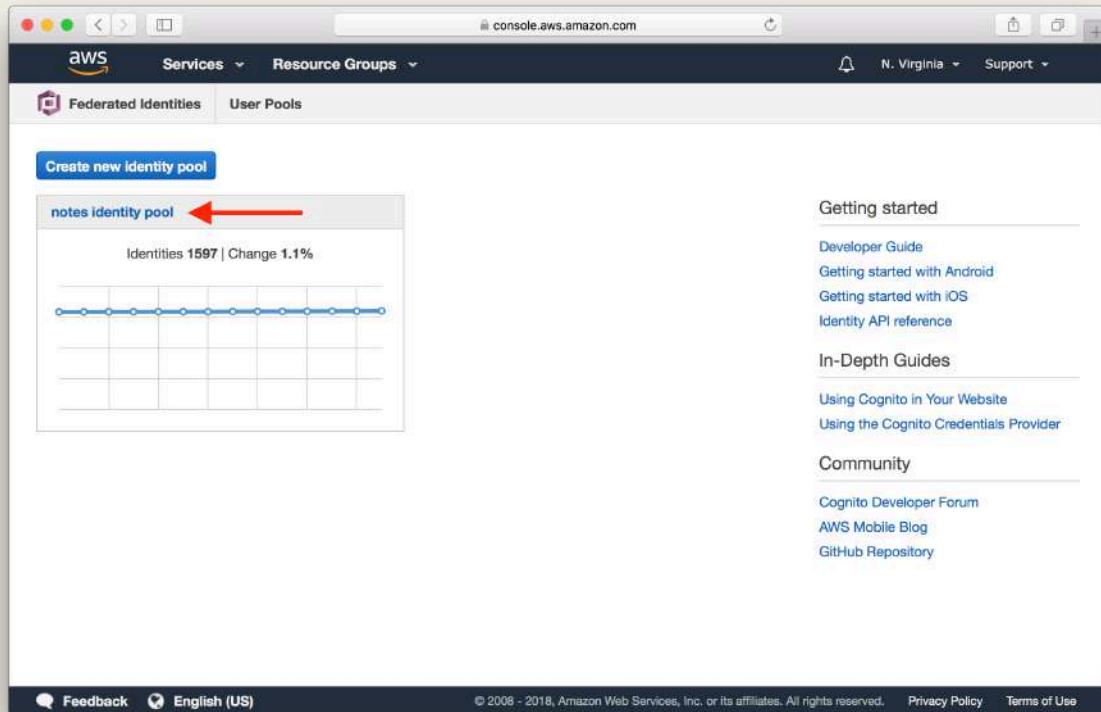
Add Facebook as an Authentication Provider

Head over to your [AWS Console](#), and go to **Cognito** and click **Manage Identity Pools**.



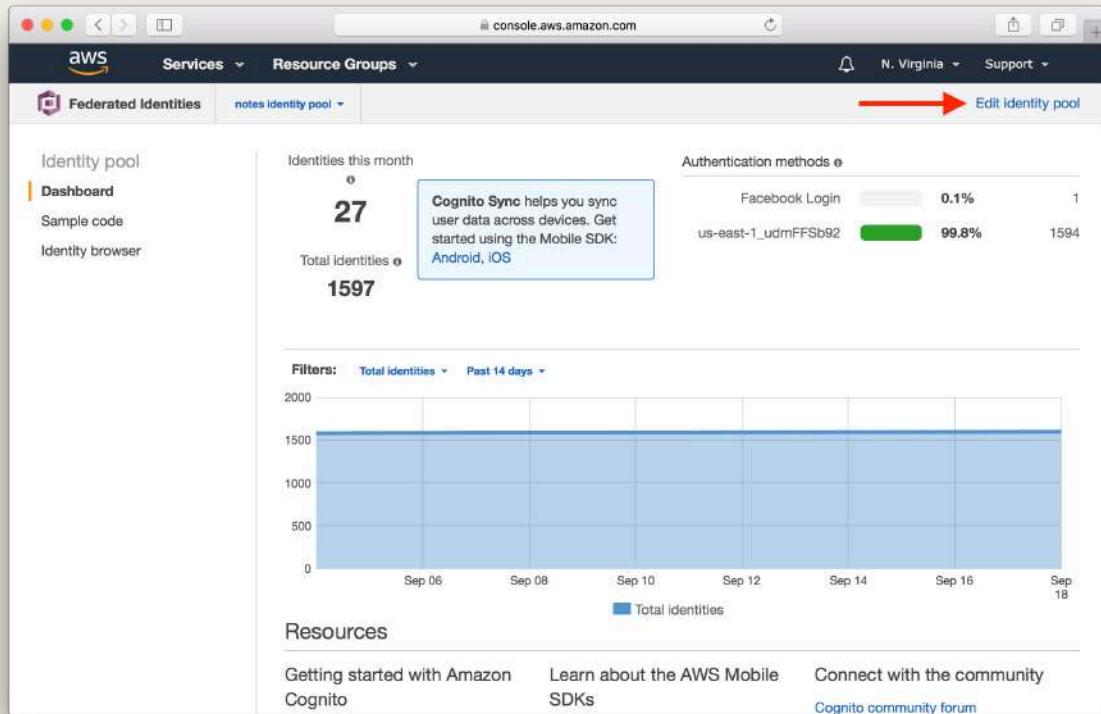
Select Manage Identity Pools screenshot

Select the Identity Pool that you are using for your app.



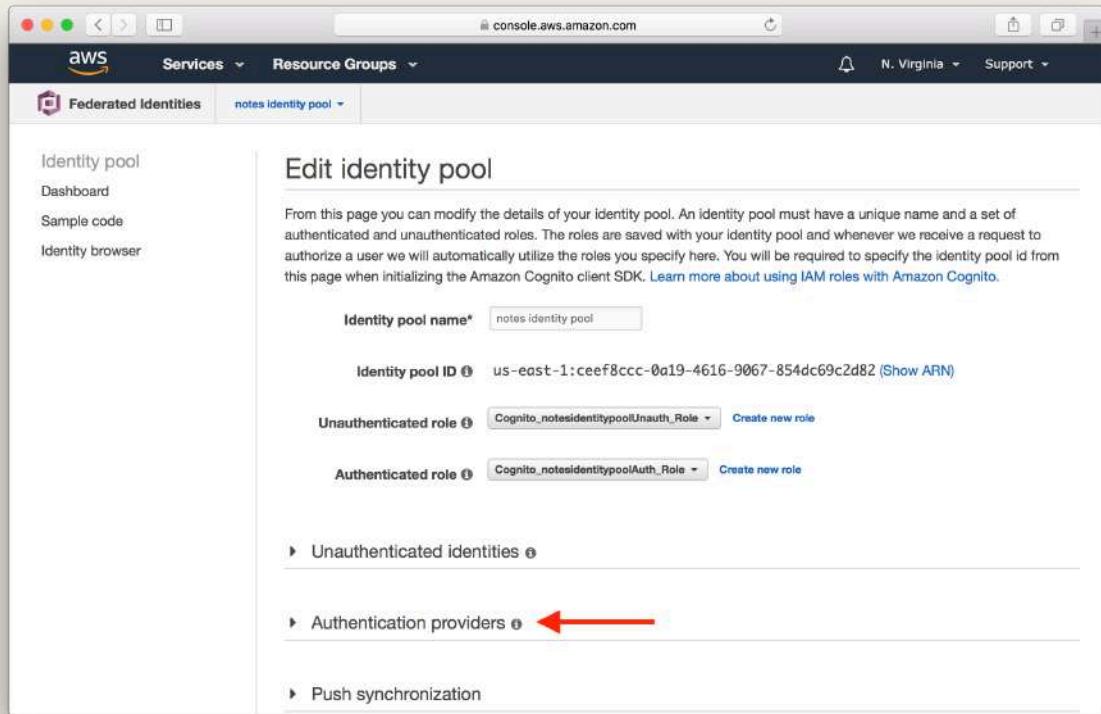
Select Identity Pool for app screenshot

Hit **Edit identity pool** from the top.



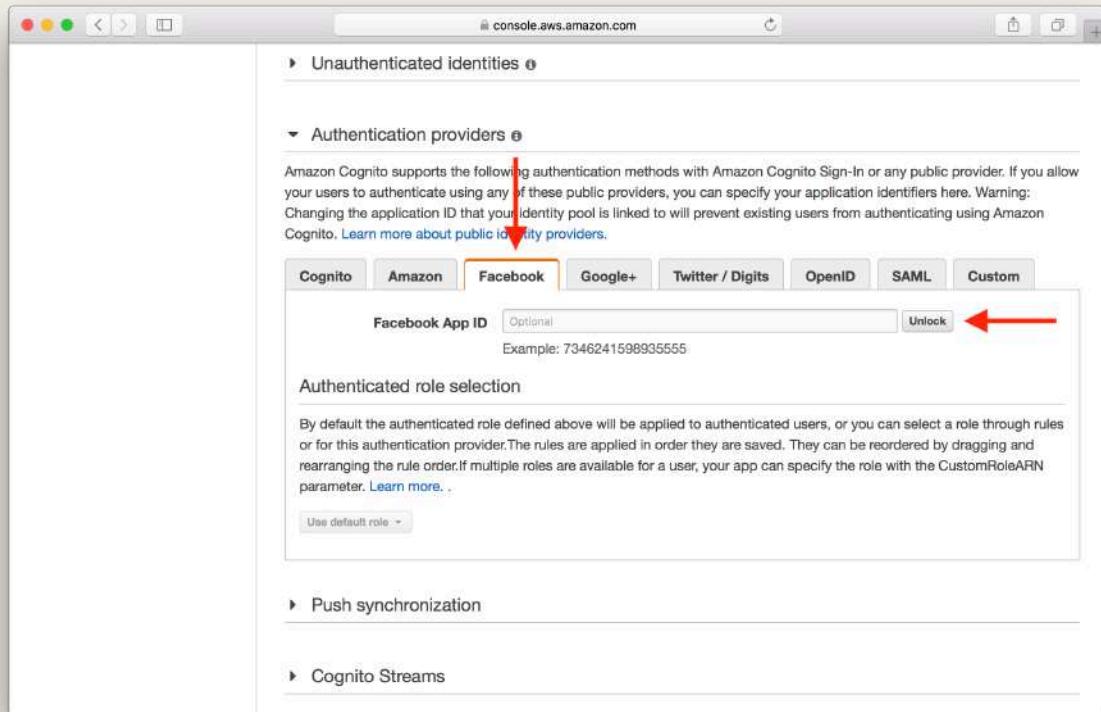
Hit Edit identity pool screenshot

Scroll down and expand the **Authentication providers**.



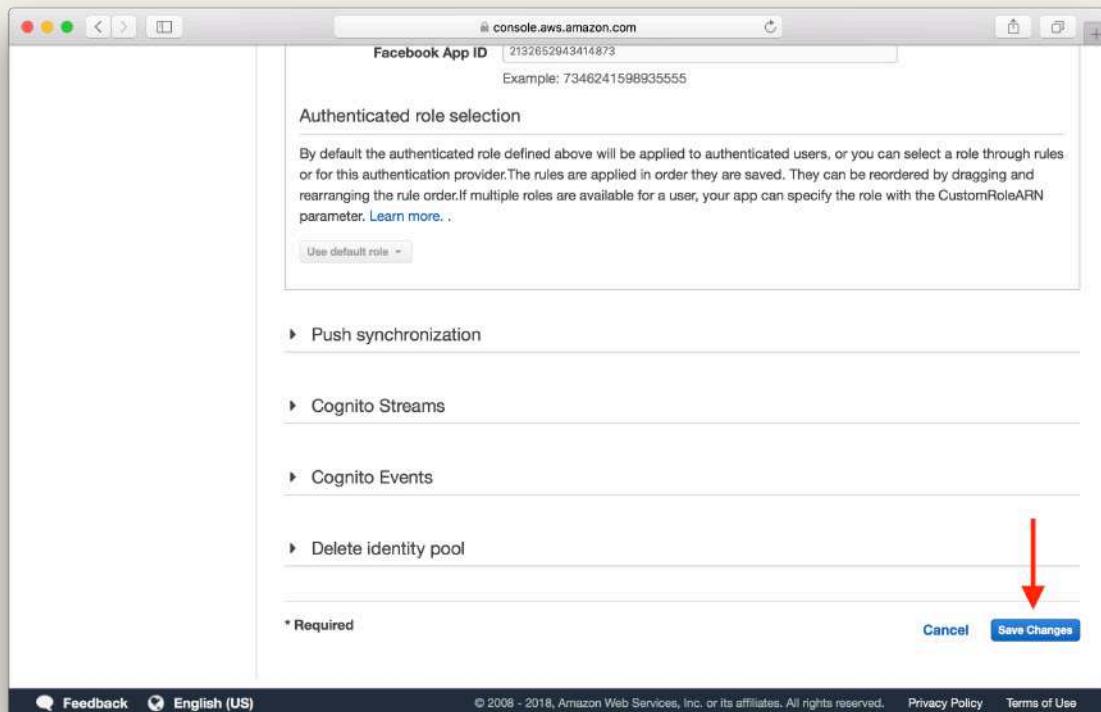
Expand Authentication providers screenshot

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.



Set Facebook App ID in Authentication providers screenshot

And scroll down and hit **Save Changes**.



Hit Save Changes in Identity Pool screenshot

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](#).

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 JSSDK 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({ isAuthenticated: 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 />
```

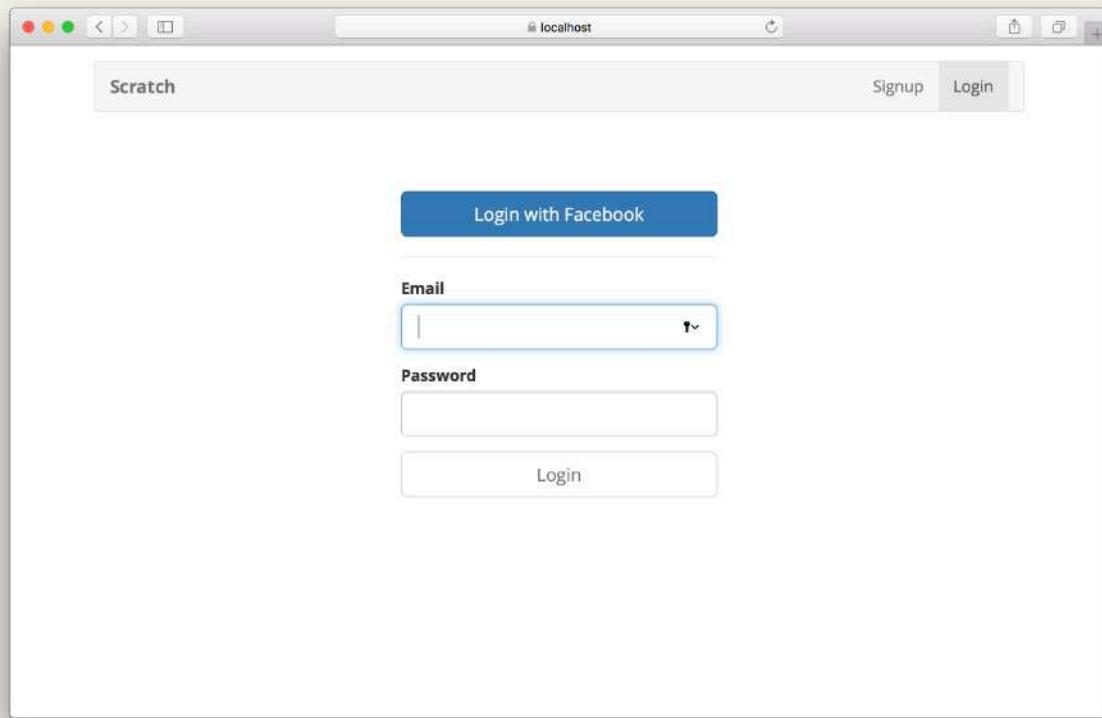
Add the button above our login and signup form. And don't forget to import it using `import FacebookButton from "../components/FacebookButton";`.

◆ CHANGE Also, add the handler method as well.

```
handleFbLogin = () => {  
  this.props.userHasAuthenticated(true);  
};
```

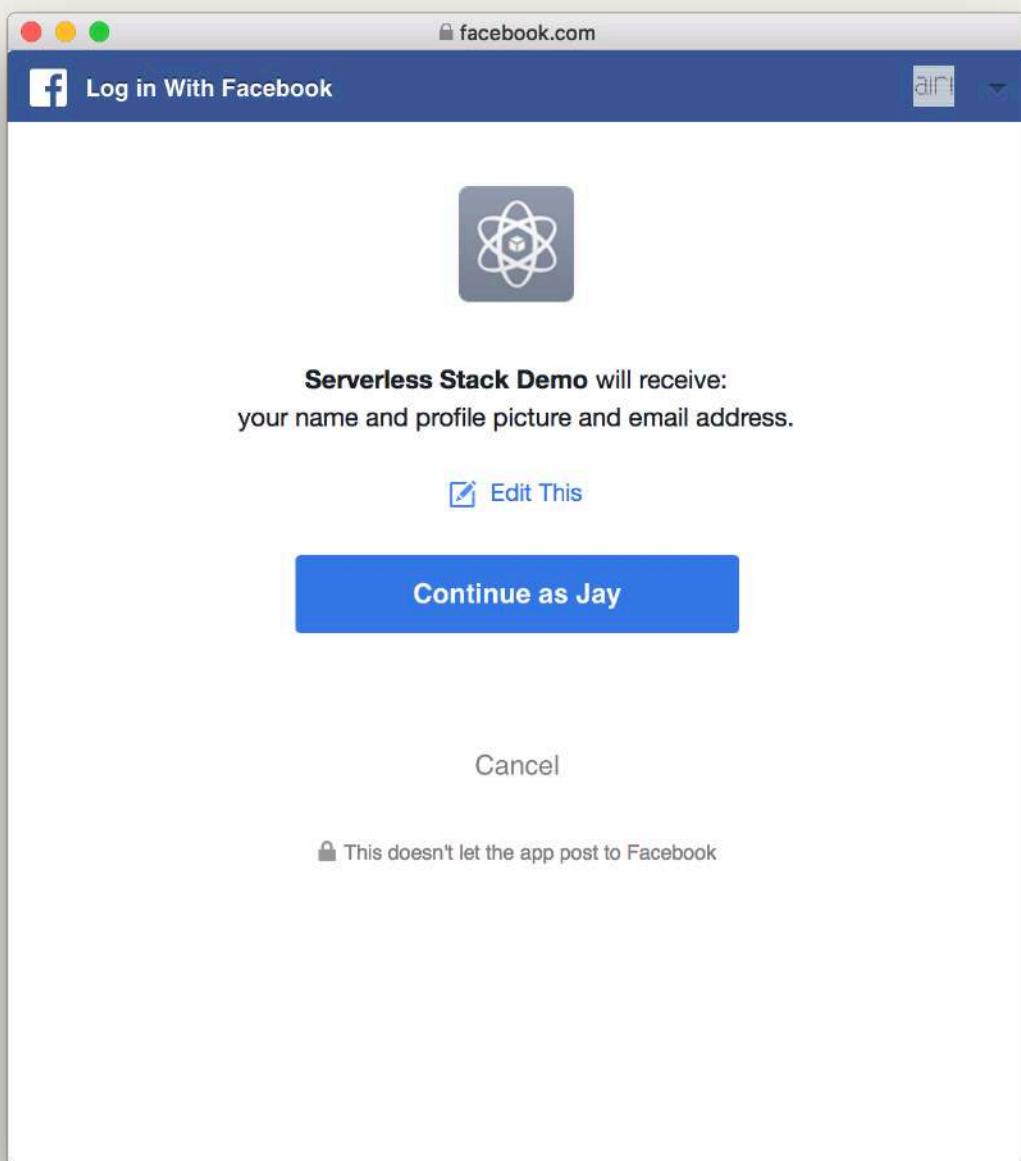
The above logs the user in to our React app, once the Facebook sign up process is complete. Make sure to add these to `src/containers/Signup.js` as well.

And that's it, if you head over to your app you should see the *login with Facebook* option.



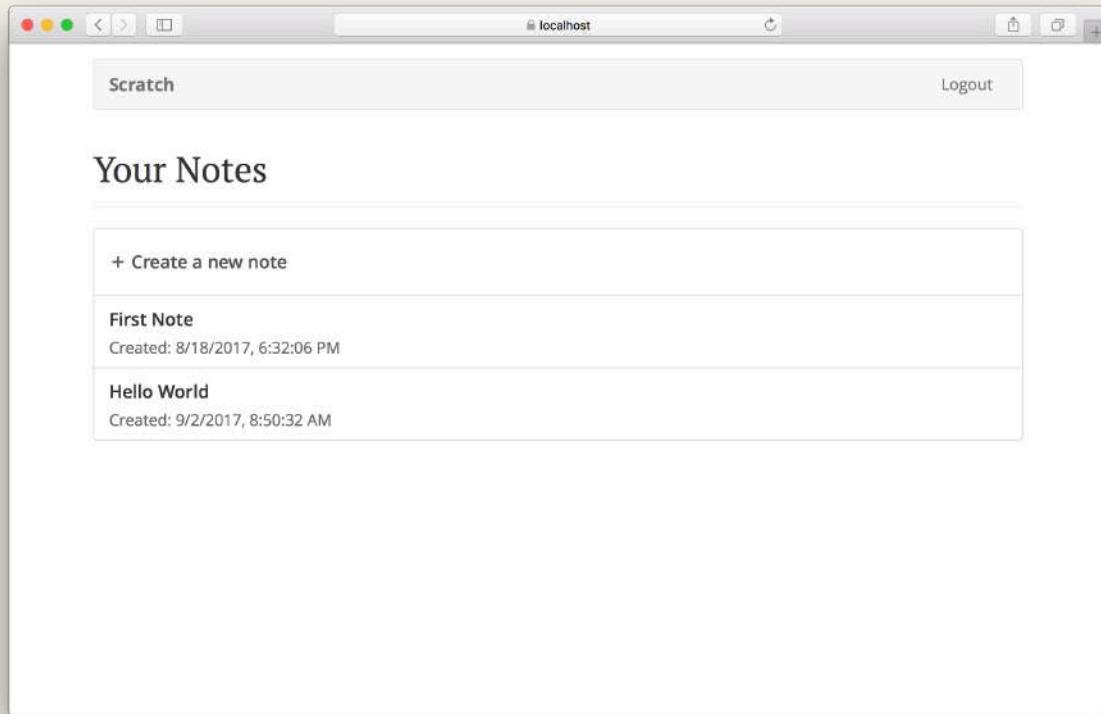
Login with Facebook option screenshot

Clicking on it should bring up the Facebook dialog asking you to login with your app.



Facebook login dialog screenshot

Once you are logged in, you should be able to interact with the app just as before.



Logged in demo app screenshot

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.



Help and discussion

View the [comments for this chapter on our forums](#)



For reference, here is the code we are using

[Facebook Login Frontend Source](#)