



# SST

A guide for building full-stack apps with serverless  
and React

[sst.dev/guide](https://sst.dev/guide)

July 25, 2024 - v9.0

# Contents

Preface	5
Who Is This Guide For? . . . . .	6
What Does This Guide Cover? . . . . .	8
How to Get Help? . . . . .	13
Introduction	15
What is Serverless? . . . . .	16
What is AWS Lambda? . . . . .	19
Why Create Serverless Apps? . . . . .	24
Set up your AWS account	26
Create an AWS Account . . . . .	27
Create an IAM User . . . . .	28
Configure the AWS CLI . . . . .	42
Setting up an SST app	44
What is SST? . . . . .	45
What Is Infrastructure as Code . . . . .	46
Create an SST app . . . . .	50
Create a Hello World API . . . . .	54
Create your AWS resources	58
Create a DynamoDB Table in SST . . . . .	59
Create an S3 Bucket in SST . . . . .	61
Building a serverless API	62
Review Our App Architecture . . . . .	63
Add an API to Create a Note . . . . .	65
Add an API to Get a Note . . . . .	74
Add an API to List All the Notes. . . . .	77
Add an API to Update a Note . . . . .	80
Add an API to Delete a Note . . . . .	83
Users and authentication	86
Auth in Serverless Apps . . . . .	87
Adding Auth to Our Serverless App . . . . .	93

	SST
Secure Our Serverless APIs . . . . .	100
Working with secrets	104
Setup a Stripe Account. . . . .	105
Handling Secrets in SST . . . . .	108
Add an API to Handle Billing. . . . .	110
Serverless unit tests	115
Unit Tests in Serverless . . . . .	116
CORS in serverless	119
Handle CORS in Serverless APIs . . . . .	120
Handle CORS in S3 for File Uploads . . . . .	124
Setting up a React app	126
Create a New React.js App . . . . .	127
Add App Favicons . . . . .	132
Set up Custom Fonts . . . . .	137
Set up Bootstrap . . . . .	140
Routes in React	142
Handle Routes with React Router . . . . .	143
Create Containers . . . . .	145
Adding Links in the Navbar . . . . .	150
Handle 404s. . . . .	154
Adding auth to a React app	157
Configure AWS Amplify . . . . .	158
Create a Login Page . . . . .	162
Login with AWS Cognito . . . . .	166
Add the Session to the State. . . . .	168
Load the State from the Session . . . . .	173
Clear the Session on Logout. . . . .	178
Redirect on Login and Logout . . . . .	179
Give Feedback While Logging In . . . . .	182
Create a Custom React Hook to Handle Form Fields . . . . .	189
Create a Signup Page . . . . .	194
Create the Signup Form . . . . .	195
Signup with AWS Cognito. . . . .	202
Building a React app	206
Add the Create Note Page. . . . .	207
Call the Create API . . . . .	211
Upload a File to S3 . . . . .	214
List All the Notes . . . . .	217

	SST
Call the List API . . . . .	220
Display a Note . . . . .	225
Render the Note Form . . . . .	228
Save Changes to a Note . . . . .	233
Delete a Note . . . . .	236
Create a Settings Page . . . . .	238
Add Stripe Keys to Config . . . . .	241
Create a Billing Form . . . . .	242
Connect the Billing Form . . . . .	247
Securing React pages	252
Set up Secure Pages. . . . .	253
Create a Route That Redirects . . . . .	254
Use the Redirect Routes . . . . .	257
Redirect on Login. . . . .	260
Using Custom Domains	262
Getting Production Ready . . . . .	263
Purchase a Domain with Route 53 . . . . .	265
Custom Domains in serverless APIs . . . . .	271
Custom Domains for React Apps on AWS . . . . .	273
Automating deployments	276
Creating a CI/CD Pipeline for serverless . . . . .	277
Setting up the SST Console . . . . .	279
Deploying Through the SST Console. . . . .	293
Conclusion	297
Wrapping Up . . . . .	298
Further Reading . . . . .	301
Translations. . . . .	302
Giving Back . . . . .	304
Changelog . . . . .	306
Staying up to date . . . . .	314

# **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.

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 SST](#).

We are also catering this solely towards JavaScript/TypeScript 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 (TypeScript) and environment (Node.js) to build your entire application.

## Why TypeScript

We use TypeScript across the board for this guide from the frontend, to the backend, all the way to creating our infrastructure. If you are not familiar with TypeScript you might be wondering why does typesafety matter.

One big advantage is that of using a fully typesafe setup is that your code editor can autocomplete and point out any invalid options in your code. This is really useful when you are first starting out. But it's also useful when you are working with configuring infrastructure through code.

Aside from all the autocomplete goodness, typesafety ends up being critical for the maintainability of codebases. This matters if you are planning to work with the same codebase for years to come.

It should be easy for your team to come in and make changes to parts of your codebase that have not been worked on for a long time. TypeScript allows you to do this! Your codebase no longer feels *brittle* and you are not afraid to make changes.

## TypeScript made easy

If you are not used to TypeScript, you might be wondering, “*Don’t I have to write all these extra types for things?*” or “*Doesn’t TypeScript make my code really verbose and scary?*”.

These are valid concerns. But it turns out, if the libraries you are using are designed well for TypeScript, you won’t need a lot of extra type definitions in your code. In fact, as you’ll see in this tutorial, you’ll get all the benefits of a fully typesafe codebase with code that looks almost like regular JavaScript.

Also, TypeScript can be gradually adopted. Meaning that you can use our TypeScript starter while adding JavaScript files to it. We don’t recommend doing this, but that’s always an option for you.

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



### Help and discussion

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

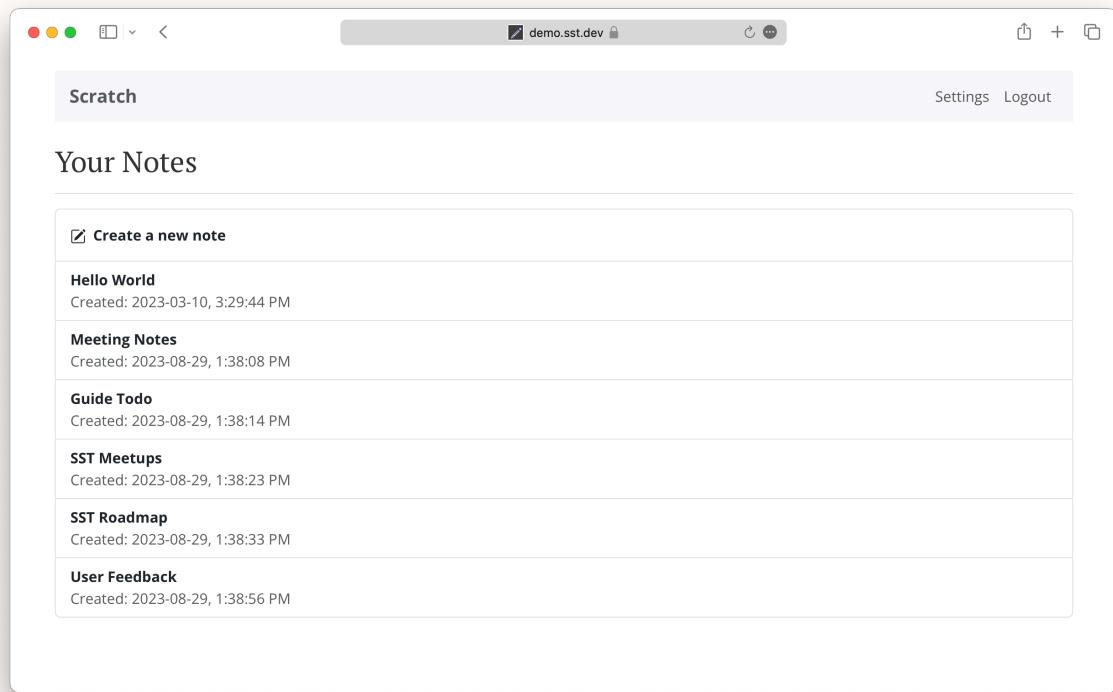
# What Does This Guide Cover?

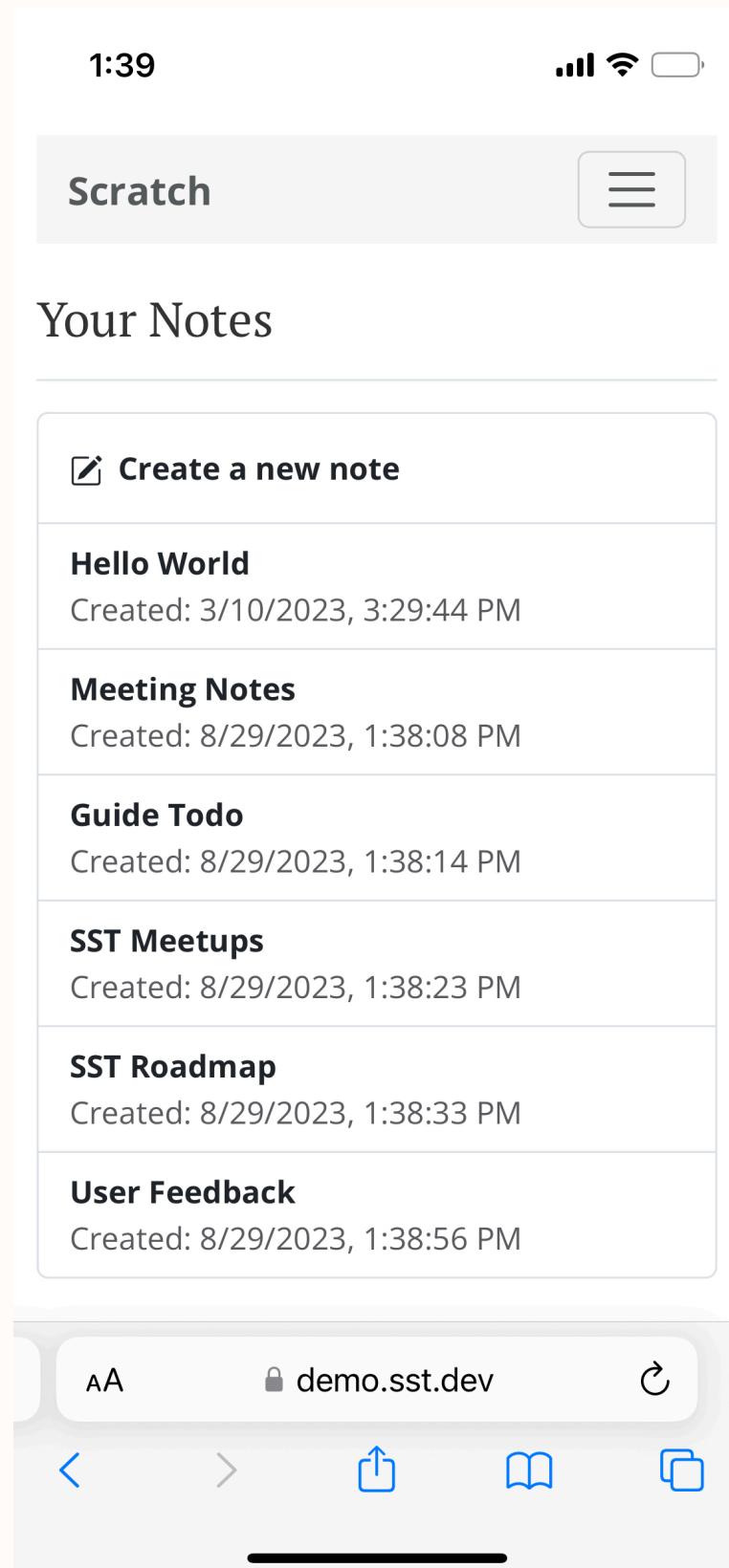
To step through the major concepts involved in building web applications, we are going to be building a simple note taking app called **Scratch**.

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.

## Demo App

The demo app is a single page application powered by a serverless API written completely in TypeScript.





Completed app mobile screenshot

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`

## Demo Source

Here is the complete source of the app we will be building. We recommend bookmarking it and use it as a reference.

- [Demo source](#)

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

- [AWS](#)
  - [S3](#) for file uploads
  - [DynamoDB](#) for our database
  - [Lambda & API Gateway](#) for our serverless API
  - [Cognito](#) for user authentication and management
- [React](#) for our frontend
  - [Bootstrap](#) for the UI Kit
  - [React Router](#) for routing
  - [Vite](#) for building our single page app
- [Vitest](#) for our unit tests

- [GitHub](#) for hosting our project repos
- [Stripe](#) for processing credit card payments

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 will be ready to build **real-world, secure, and fully-functional** web apps. And don't worry we will be around to help!

## Requirements

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

- [Node v20](#) and [npm v10](#)
- A [GitHub account](#)
- Basic knowledge of JavaScript and TypeScript
- 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:

For the backend:

- Configure your AWS account
- Create your database using DynamoDB
- Set up S3 for file uploads
- Write the various backend APIs
- Set up Cognito User Pools to manage user accounts
- Set up Cognito Identity Pool to secure our resources
- Working with secrets
- Adding unit tests

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 with Amplify to login and signup users
- Plugin to the backend APIs to manage our notes
- Use the AWS Amplify to upload files
- Accepting credit cards with the Stripe React SDK

Deploying to prod:

- Use a custom domain for your app
- Deploy your app when you push to git

We believe 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 [Discord](#).



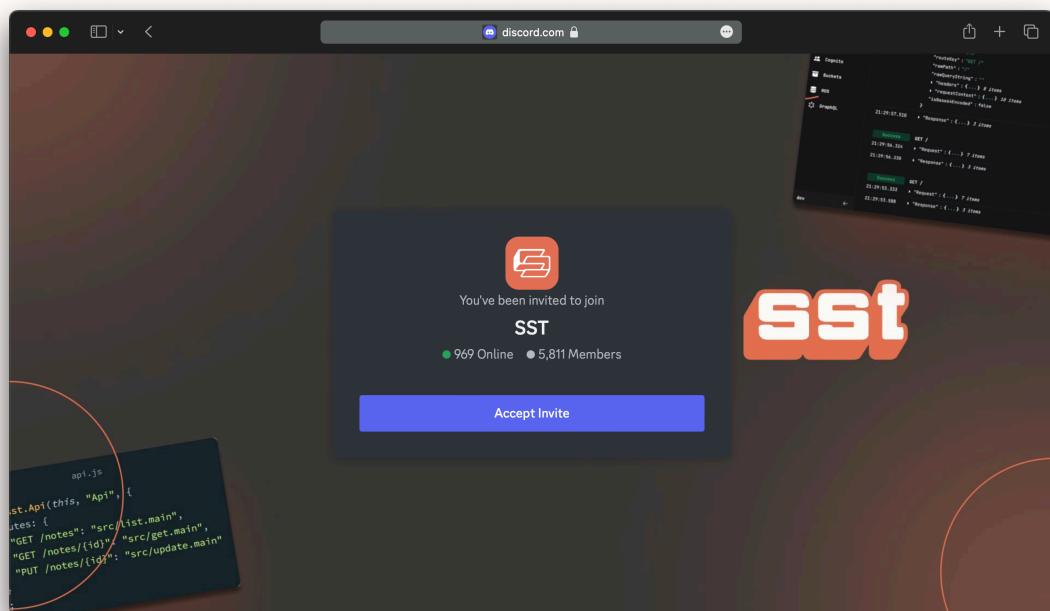
#### Help and discussion

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 also have a very active [Discord](#) community. If you have a question, start a new thread in the `#help` channel.



- 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.

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!

We recommend you start by [joining us on Discord](#)!



**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 next 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. [SST](#), which we are going to be using in this tutorial, has a pre-built Cron component to help with this.

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 18.x, 16.x, and 14.x
- Java 17, 11 and 8
- Python 3.11, 3.10, 3.9, 3.8, and 3.7
- .NET 7 and 6
- Go 1.x
- Ruby 3.2 and 2.7
- Rust

**Info:** .NET Core 2.2 and 3.0 are supported through custom runtimes.

Check out the AWS docs to learn more about the available runtimes.

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

- Memory: 128MB - 10240MB, in 1 MB increments
- Ephemeral disk space: 512MB - 10240MB, in 1 MB increments
- Max execution duration: 900 seconds
- Compressed package size: 50MB
- Uncompressed package size: 250MB
- Container image package size: 10GB

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 will 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. If you need more space, you can package your container as a Docker image which can be up to 10GB. We will take a look at the packaging process below.

## Lambda Function

Finally here is what a Lambda function using Node.js looks like.

```
export const handler = async (event, context) => {
  // Do work

  return {
    statusCode: 200,
    body: "Hello World!"
  };
};
```

Here `handler` is the name of our Lambda function. It's an `async` 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 return. If this function is connected to an API Gateway, your can return the response HTTP status code and body.

## 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 [SST](#). We will 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 - 15 mins 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, as noted in the optimization above, AWS will hang on to an existing container for a few minutes and use that to respond to any requests. So for that container instance, the code around the Lambda function will only be invoked once. While the actual Lambda function will be invoked for each request.

For example, the `createNewDbConnection` method below is called once per container instance and not every time the Lambda function is invoked. The `handler` function on the other hand is called on every invocation.

```
var dbConnection = createNewDbConnection();  
  
export const handler = async (event, context) => {  
  var result = dbConnection.makeQuery();  
  return {  
    statusCode: 200,  
    body: JSON.stringify(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 1ms.

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. You can save up to 17% by purchasing AWS Compute Savings Plans in exchange for a 1 or 3 year commitment. 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
<b>Total</b>		<b>\$6.10</b>

- [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 create your account.

Next we'll 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

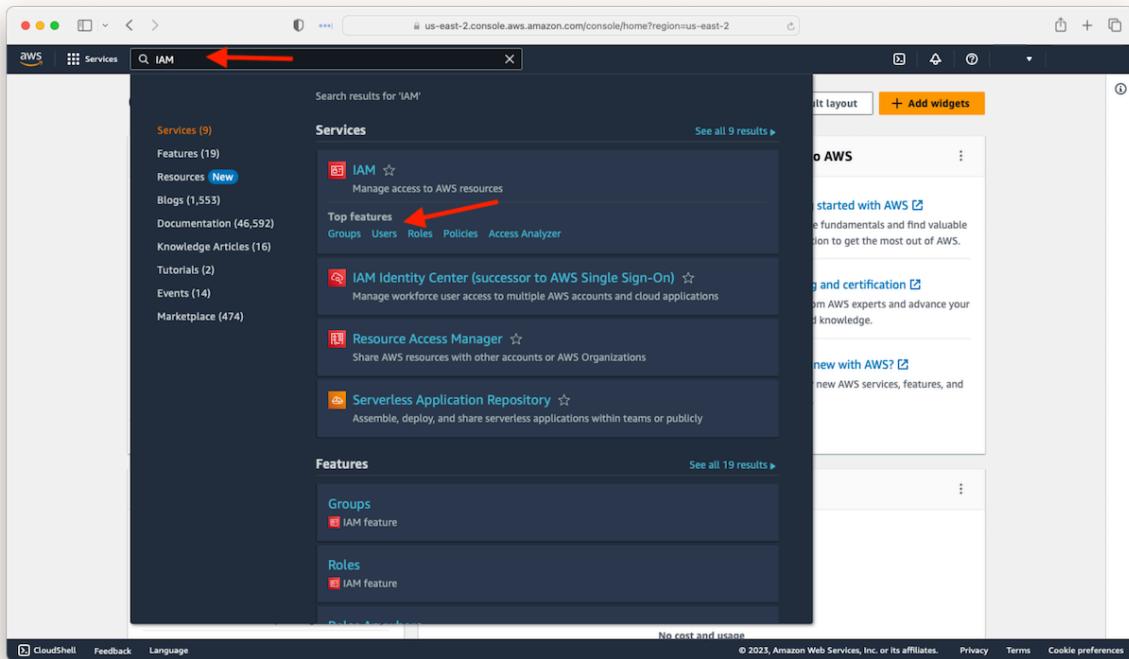
Once we have an AWS account, we'll need to create an IAM user to programmatically interact with it. We'll be using this later to configure our AWS CLI (command-line interface).

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.

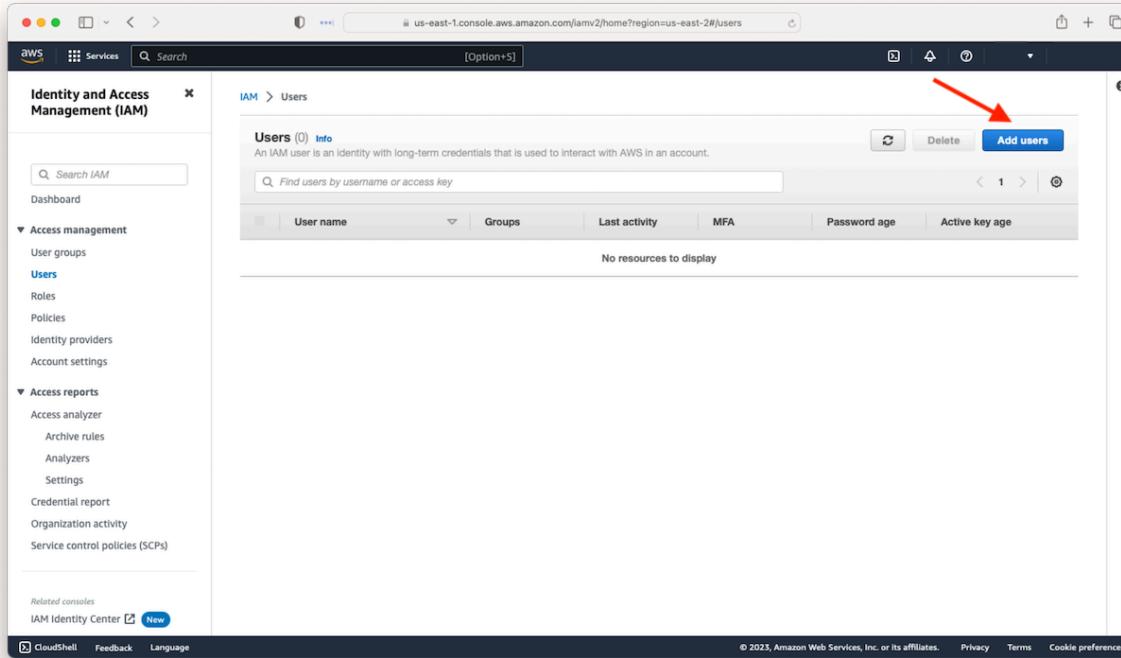
## Create User

First, log in to your [AWS Console](#) and search for IAM in the search bar. Hover or focus on the **IAM card** and then select the **Users** link.



Select IAM Service Screenshot

Select **Add Users.**

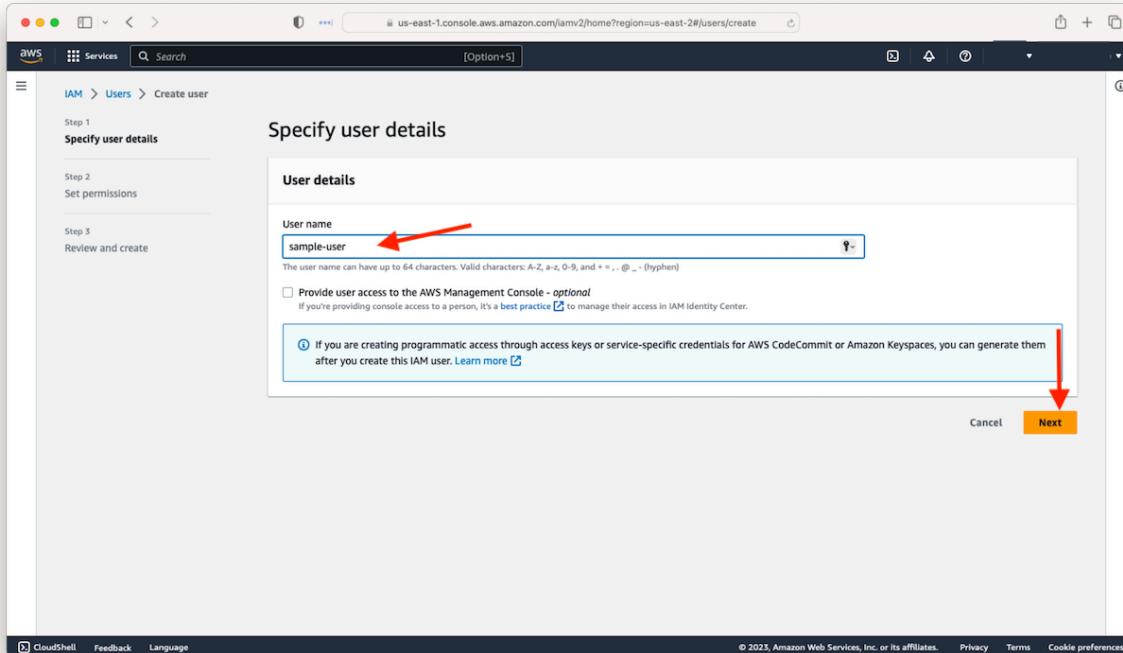


Add IAM User Screenshot

Enter a **User name**, then select **Next**.

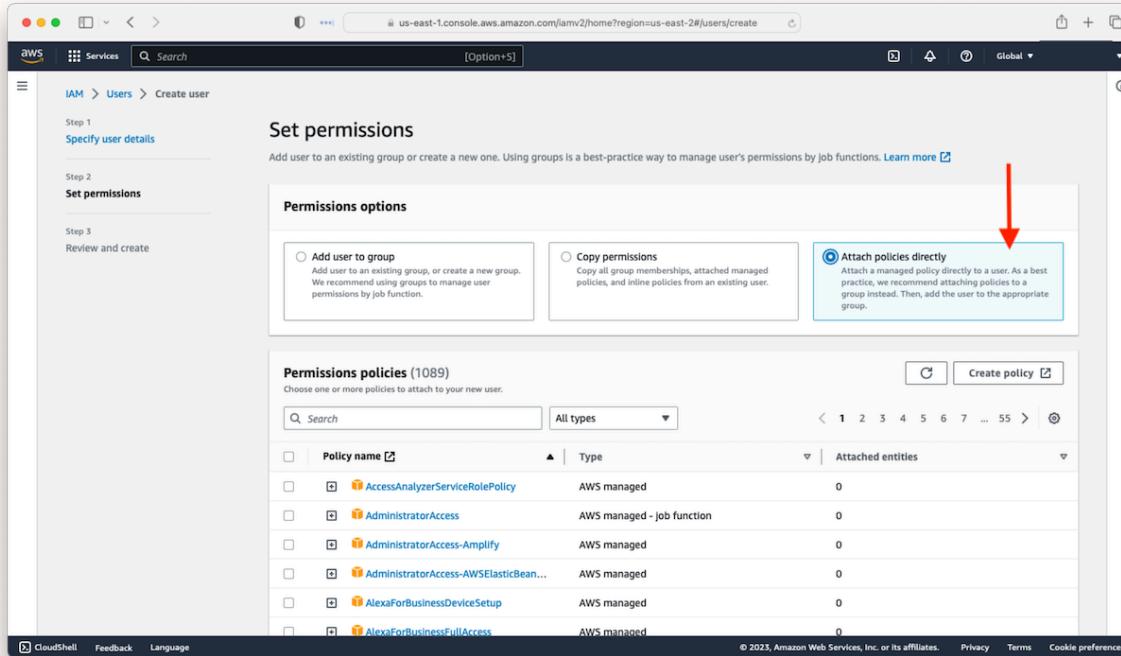
This account will be used by our [AWS CLI](#) and [SST](#). They will be connecting to the AWS API directly and will not be using the Management Console.

**Note:** The best practice is to avoid creating keys when possible. When using programmatic access keys, regularly rotate them. In most cases, there are alternative solutions, see the [AWS IAM User Guide](#) for more information.



Fill in IAM User Info Screenshot

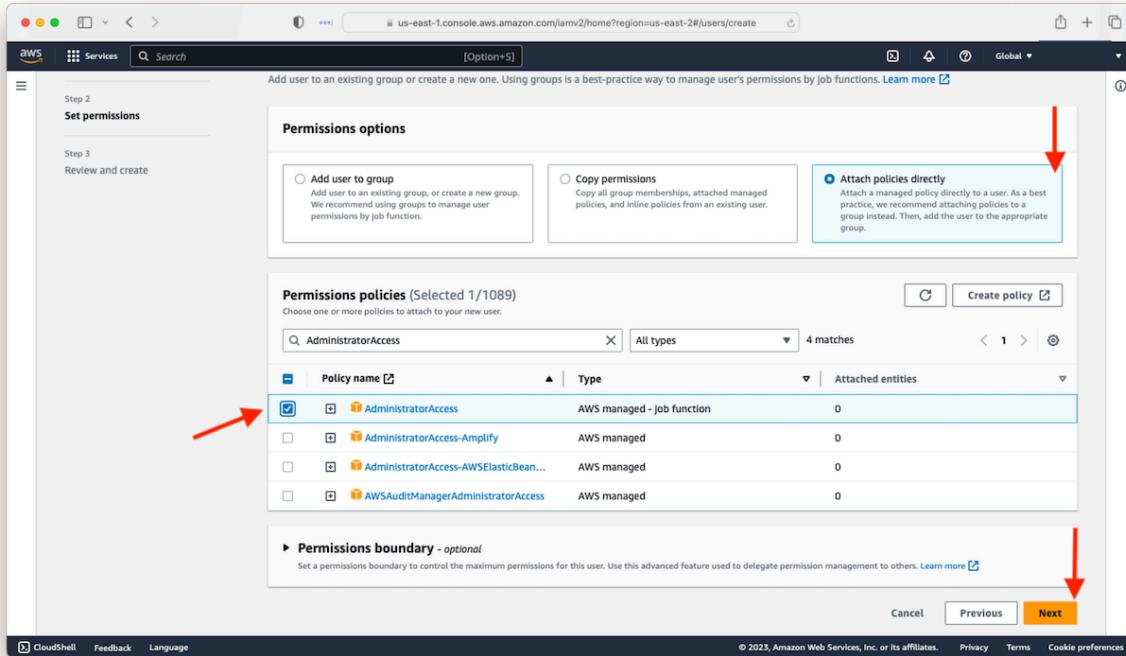
Select **Attach existing policies directly**.



Add IAM User Policy Screenshot

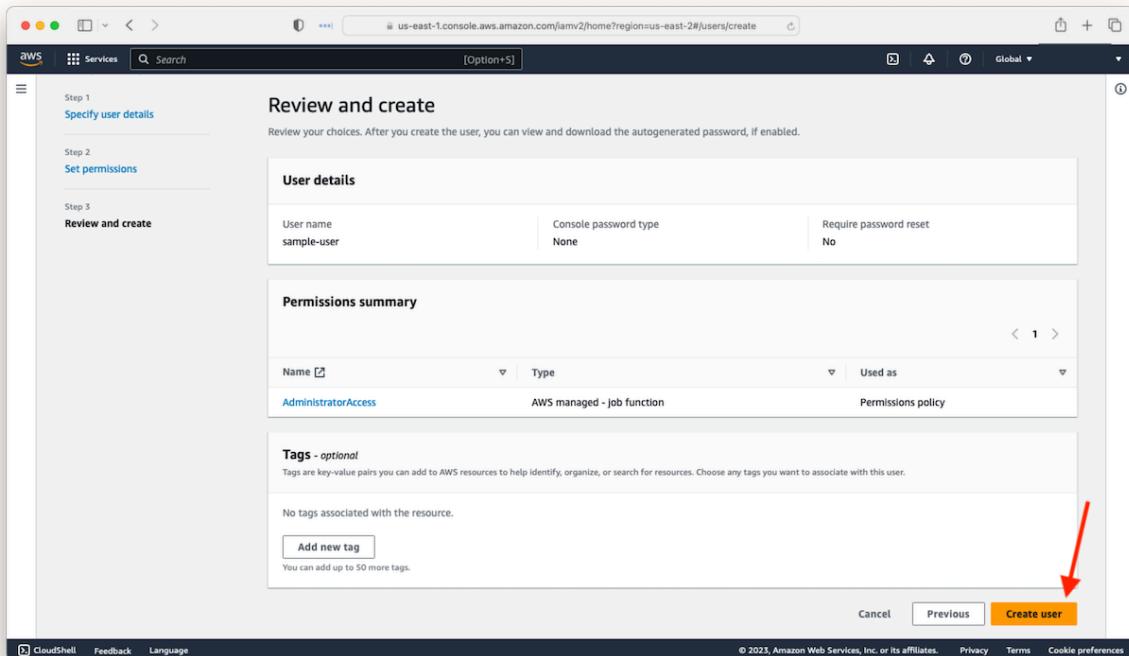
Search for **AdministratorAccess** and select the policy by checking the checkbox, then select **Next**.

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



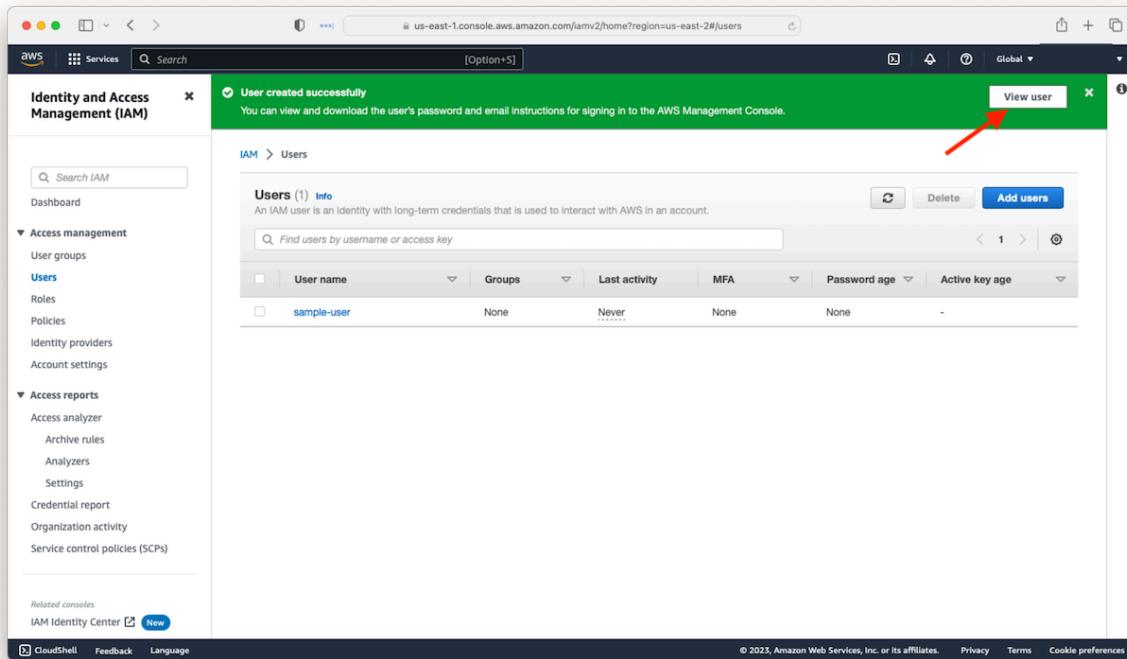
Added Admin Policy Screenshot

Select **Create user**.



Reivew IAM User Screenshot

Select **View user**.



View IAM User Screenshot

## Select **Security credentials**

The screenshot shows the AWS IAM User Security Credentials page for a user named 'sample-user'. The 'Security credentials' tab is selected. A red arrow points to the 'Access key 1' section, which shows 'Not enabled'. Below it, 'Access key 2' is also shown as 'Not enabled'. The 'Permissions policies' section lists a single policy named 'AdministratorAccess' attached directly to the user.

ARN	Console access	Access key 1
arn:aws:iam::755394236926:user/sample-user	Disabled	Not enabled

Created	Last console sign-in	Access key 2
May 16, 2023, 16:41 (UTC-05:00)	-	Not enabled

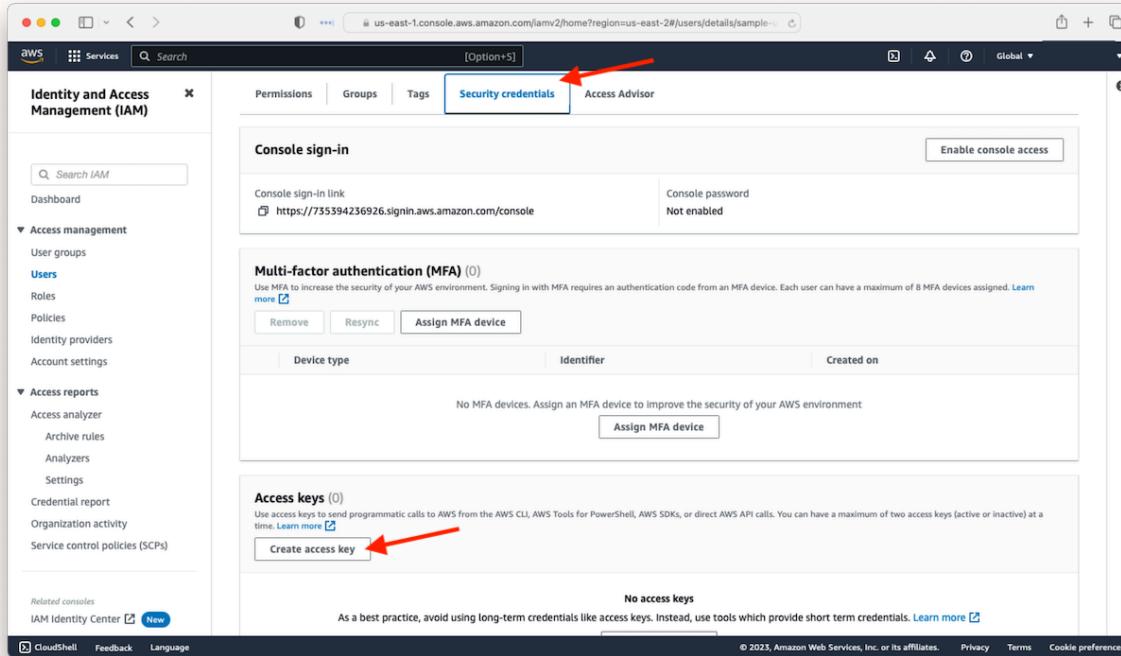
**Permissions policies (1)**  
Permissions are defined by policies attached to the user directly or through groups.

Policy name	Type	Attached via
AdministratorAccess	AWS managed - job function	Directly

**Permissions boundary (not set)**  
Set a permissions boundary to control the maximum permissions for this user. Use this advanced feature used to delegate permission management to others. Learn more

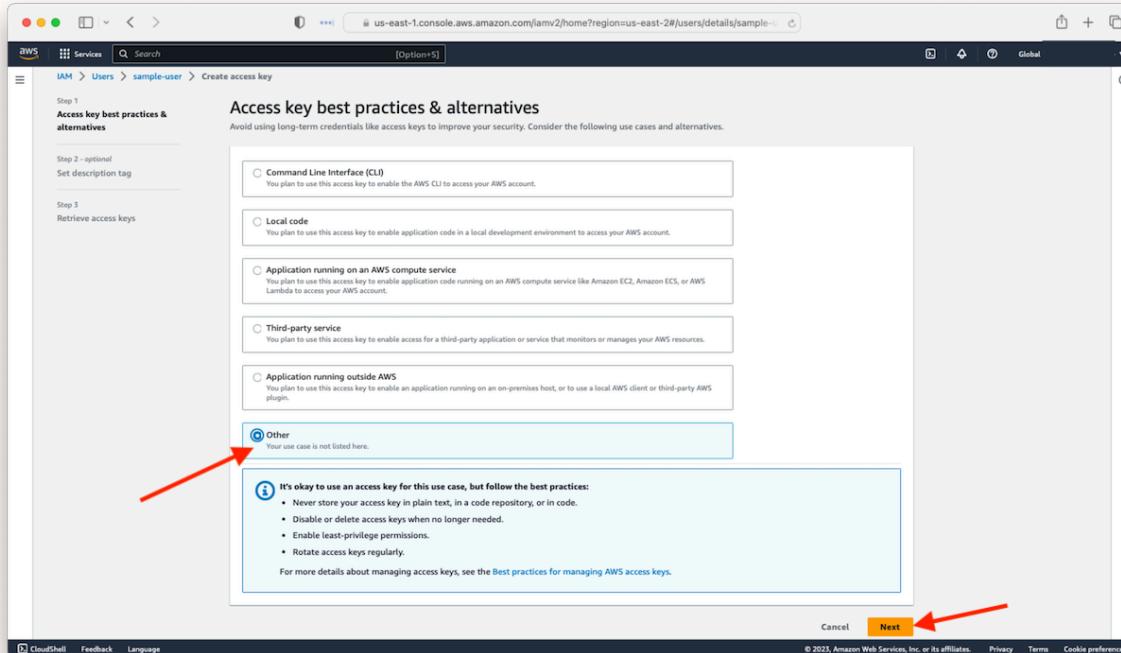
IAM User Security Credentials Screenshot

Select **Create access key**



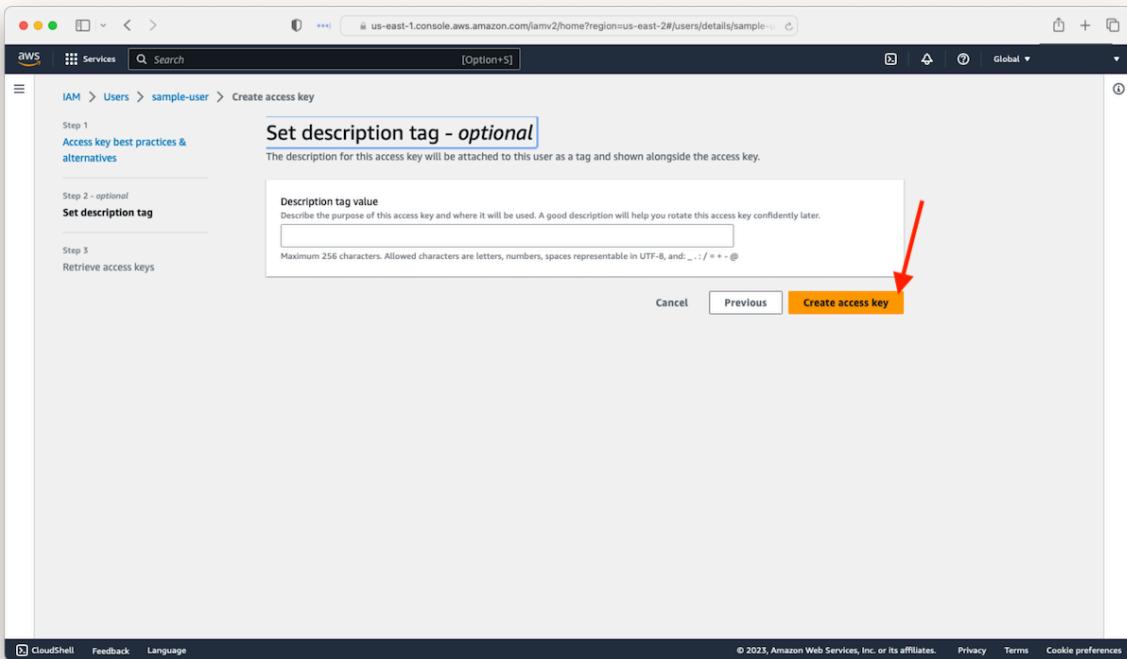
IAM User Create Access Key Screenshot

In keeping with the current guide instructions, we will choose other to generate an access key and secret. Select **Other** and select **Next**



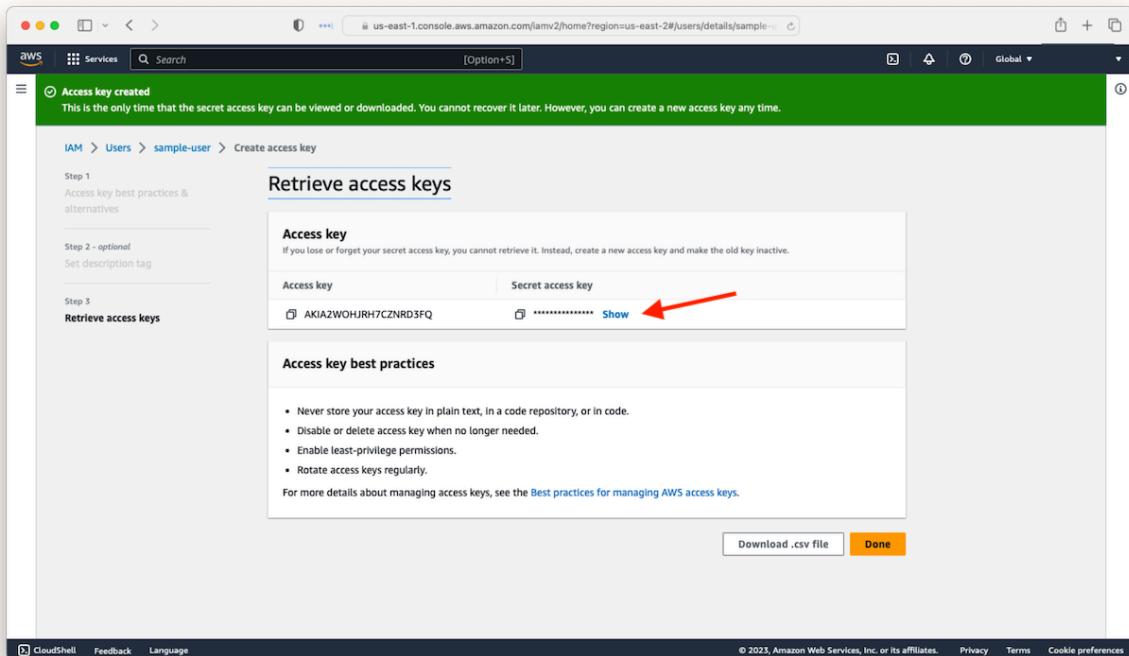
### IAM User Access Key Purpose

You could add a descriptive tag here, but we will skip that in this tutorial, select **Create access key**



### IAM User Access Key Purpose

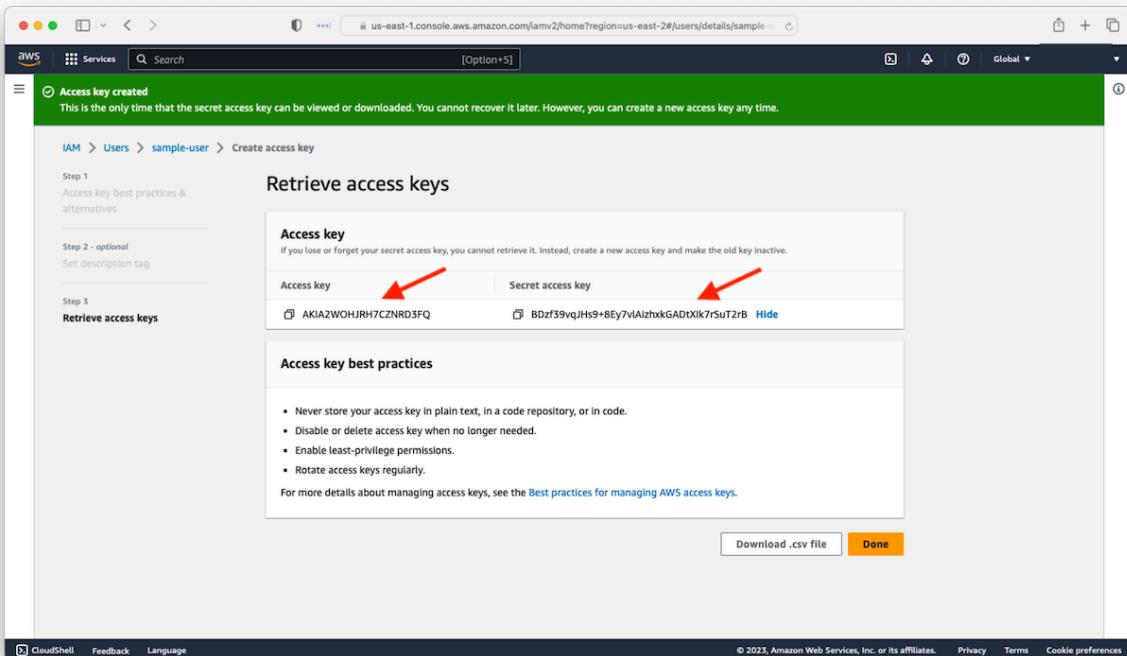
Select **Show** to reveal **Secret access key**.



### IAM User Access Key Show

**Note:** This is the only screen on which you will be able to access this key. Save it to a secure location using best practices to ensure the security of your application.

Take a note of the **Access key** and **Secret access key**. We will be needing this in the next chapter.



IAM Access Credentials Screenshot

Now let's configure our AWS CLI. By configuring the AWS CLI, we can deploy our applications from our command line.



### 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 an SST app**

# What is SST?

We are going to be using [AWS Lambda](#), [Amazon API Gateway](#), and a host of other AWS services to create our application. 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. But working directly with AWS Lambda, API Gateway, and the other AWS services can be a bit cumbersome.

Since these services run on AWS, it can be tricky to test and debug them locally. And a big part of building serverless applications, is being able to define our infrastructure as code. This means that we want our infrastructure to be created programmatically. We don't want to have to click through the AWS Console to create our infrastructure.

To solve these issues we created the [SST](#).

SST makes it easy to build full-stack applications by allowing developers to:

1. Define their *entire* infrastructure in code
2. Use [higher-level components](#) designed for modern full-stack apps
3. Test their applications [Live](#)
4. Debugging with your IDEs
5. Manage their apps with a [web based dashboard](#)
6. Deploy to multiple environments and regions

Before we start creating our application, let's look at the *infrastructure as code* concept in a bit more detail.



## Help and discussion

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

# What Is Infrastructure as Code

[SST](#) converts your infrastructure code into a series of API calls to your cloud providers. Behind the scenes it uses [Pulumi](#) and [Terraform](#), more on this below. Your SST config is a description of the infrastructure that you are trying to create as a part of your project. In our case we'll be defining Lambda functions, API Gateway endpoints, DynamoDB tables, S3 buckets, etc.

While you can configure this using the [AWS console](#), you'll need to do a whole lot of clicking around. It's much better to configure our infrastructure programmatically.

This general pattern is called **Infrastructure as code** and it has some massive benefits. Firstly, it allows us to completely automate the entire process. All you need is a config and a CLI to create your entire app. Secondly, it's not as error prone as doing it by hand.

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 the production environment that your users are interacting with.

## Terraform

[Terraform](#) is a large open source project that maintains *providers* for all the cloud providers out there. Each provider includes resources that allow you to define almost everything the cloud provider has.

Terraform uses something called [HCL](#) to define resources. For example, here's what the Terraform definition of a DynamoDB table looks like.

```
resource "aws_dynamodb_table" "example" {
    name          = "example-table"
    billing_mode = "PAY_PER_REQUEST"

    attribute {
        name = "id"
        type = "S"
```

```
}

attribute {
  name = "sort_key"
  type = "N"
}

hash_key = "id"
range_key = "sort_key"

tags = {
  Name        = "example-table"
  Environment = "production"
}
}
```

## Pulumi

One of the problems with the definition above is that as you work with more complex applications, your definitions start to get really large.

And while HCL (or YAML, or JSON) are easy to get started, it can be hard to reuse and compose them. To fix this [Pulumi](#) takes these providers and translates them into TypeScript (and other programming languages).

So in Pulumi the same DynamoDB would look something like this.

```
import * as aws from "@pulumi/aws";

const table = new aws.dynamodb.Table("exampleTable", {
  attributes: [
    { name: "id", type: "S" },
    { name: "sort_key", type: "N" },
  ],
  hashKey: "id",
  rangeKey: "sort_key",
  billingMode: "PAY_PER_REQUEST",
  tags: {
    Name: "example-table"
  }
});
```

```
Name: "example-table",
Environment: "production",
},
});
```

## Problems with traditional IaC

Traditional IaC, like the Terraform and Pulumi definition above, are made up of low level resources. This has a couple of implications:

1. You need to understand how each of these low level resources work. You need to know what the properties of a resource does.
2. You need a lot of these low level resources. For example, to deploy a Next.js app on AWS, you need around 70 of these low level resources.

This makes IaC really intimidating for most developers. Since you need very specific AWS knowledge to even deploy a simple CRUD app. As a result, IaC has been traditionally only used by DevOps or Platform engineers.

Additionally, traditional IaC tools don't help you with local development. They are only concerned with how you define and deploy your infrastructure.

To fix this, we created SST. SST has high level components that wrap around these resources with sane defaults, so creating a Next.js app is as simple as.

```
new sst.aws.Nextjs("MyWeb");
```

And it comes with a full local development environment with the `sst dev` command.

## Working with IaC

If you have not worked with IaC before, it might feel unfamiliar at first. But as long as you remember a couple of things you'll be fine.

1. **SST automatically manages** the resources in AWS defined in your app.
2. You don't need to **make any manual changes** to them in your AWS Console.

You can learn more about the [SST workflow](#).

Now we are ready to create our first SST app.



#### Help and discussion

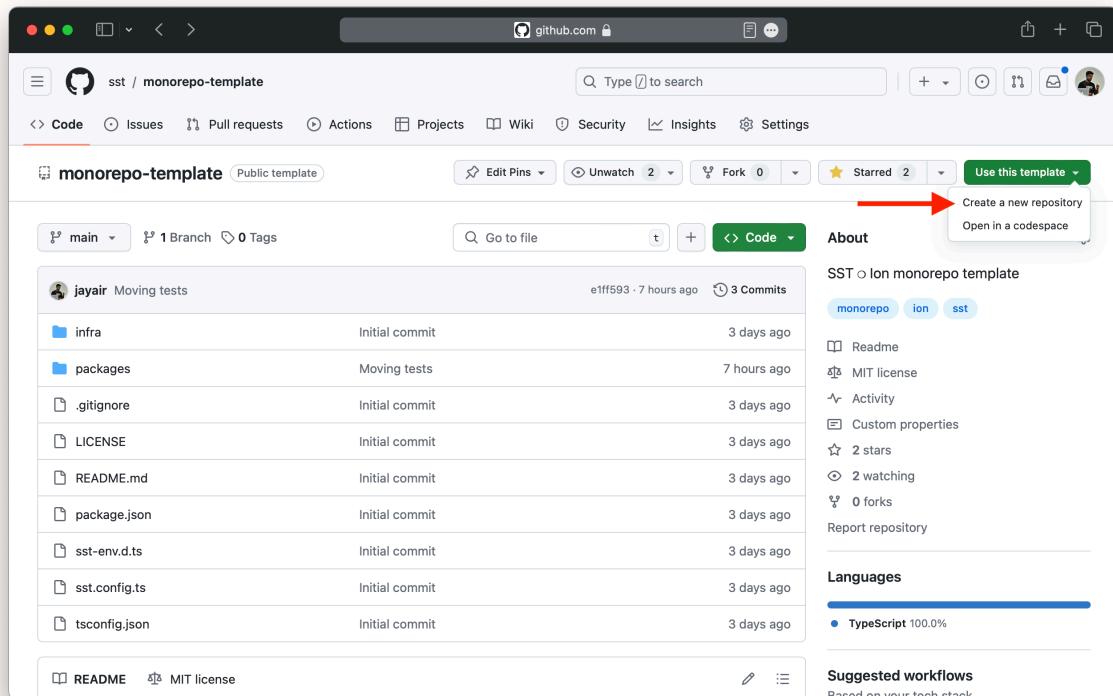
View the [comments for this chapter on our forums](#)

# Create an SST app

Now that we have some background on SST and *infrastructure as code*, we are ready to create our first SST app!

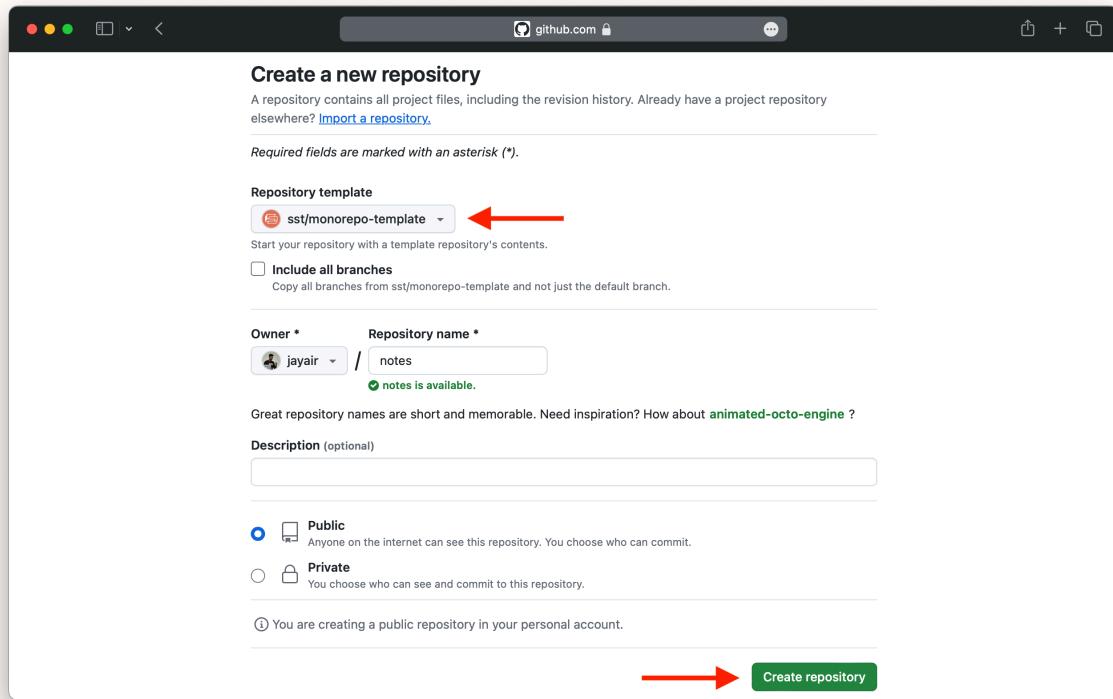
We are going to use a template SST project, it comes with a good monorepo setup. It'll help us organize our frontend and APIs.

Head over to — [github.com/sst-monorepo-template](https://github.com/sst-monorepo-template), click **Use this template > Create a new repository**.



Use the SST monorepo GitHub template screenshot

Give your repository a name, in our case we are calling it **notes**. Next hit **Create repository**.



Name new GitHub repository screenshot

Once your repository is created, copy the repository URL.

◆ CHANGE Run the following in your working directory.

```
$ git clone <REPO_URL>
$ cd notes
```

◆ CHANGE Use your app name in the template.

```
$ npx replace-in-file /monorepo-template/g notes **/*.* --verbose
```

◆ CHANGE Install the dependencies.

```
$ npm install
```

By default, the template is creating an API. You can see this in the `sst.config.ts` in the root.

```
/// <reference path=".sst/platform/config.d.ts" />

export default $config({
  app(input) {
    return {
      name: "notes",
      removal: input?.stage === "production" ? "retain" : "remove",
      home: "aws",
    };
  },
  async run() {
    await import("./infra/storage");
    const api = await import("./infra/api");

    return {
      api: api.myApi.url,
    };
  },
});
```

**Caution:** To rename an app, you'll need to remove the resources from the old one and deploy to the new one.

The name of your app as you might recall is `notes`. A word of caution on IaC, if you rename your app after you deploy it, it doesn't rename the previously created resources in your app. You'll need to remove your old app and redeploy it again with the new name. To get a better sense of this, you can read more about the [SST workflow](#).

## Project layout

An SST app is made up of two parts.

1. `infra/` — App Infrastructure

The code that describes the infrastructure of your serverless app is placed in the `infra/` directory of your project.

2. `packages/` — App Code

The Lambda function code that's run when your API is invoked is placed in the packages/functions directory of your project, the packages/core contains our business logic, and the packages/scripts are for any one-off scripts we might create.

Later on we'll be adding a packages/frontend/ directory for our React app.

The starter project that's created is defining a simple *Hello World* API. In the next chapter, we'll be deploying it and running it locally.



### Help and discussion

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

# Create a Hello World API

With our newly created SST app, we are ready to deploy a simple *Hello World* API. Let's rename some of the files from our template.

## Rename the Template

◆ CHANGE Replace our `infra/api.ts` with the following.

```
import { bucket } from "./storage";

export const api = new sst.aws.ApiGatewayV2("Api");

api.route("GET /", {
  link: [bucket],
  handler: "packages/functions/src/api.handler",
});
```

Here we are creating a simple API with one route, `GET /`. When this API is invoked, the function called `handler` in `packages/functions/src/api.ts` will be executed.

We are also *linking* an S3 Bucket to our API. This allows the functions in our API to access the bucket. We'll be using this bucket later to handle file uploads. For now let's quickly rename it.

◆ CHANGE Replace our `infra/storage.ts` with.

```
// Create an S3 bucket
export const bucket = new sst.aws.Bucket("Uploads");
```

Also let's rename how this bucket is accessed in our app code. We'll go into detail about this in the coming chapters.

◆ CHANGE Rename `Resource.MyBucket.name` line in `packages/functions/src/api.ts`.

```
body: `${Example.hello()} Linked to ${Resource.Uploads.name}.`,
```

Given that we've renamed a few components, let's also make the change in our config.

◆ CHANGE Replace the `run` function in `sst.config.ts`.

```
async run() {
  await import("./infra/storage");
  await import("./infra/api");
},
```

**Note:** By default SST sets you up with a TypeScript project. While the infrastructure is in TypeScript, you are free to use regular JavaScript in your application code.

Let's go ahead and deploy this.

## Start Your Dev Environment

We'll do this by starting up our local development environment. SST's dev environment runs your functions [Live](#). It allows you to work on your serverless apps live.

◆ CHANGE Start your dev environment.

```
$ npx sst dev
```

Running `sst dev` will take a minute or two to deploy your app and bootstrap your account for SST.

```
SST 0.1.17 ready!

\faLongArrowAltRight App:      notes
  Stage:    jayair
  Console:  https://console.sst.dev/local/notes/jayair

  ...
+ Complete
  Api: https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com
```

The API is the API we just created. Let's test our endpoint. If you open the endpoint URL in your browser, you should see *Hello World!* being printed out.



Serverless Hello World API invoked

You'll notice it's also printing out the name of the bucket that it's linked to.

## Deploying to Prod

To deploy our API to prod, we'll need to stop our local development environment and run the following.

```
$ npx sst deploy --stage production
```

We don't have to do this right now. We'll be doing it later once we are done working on our app.

The idea here is that we are able to work on separate environments. So when we are working in our personal stage (`jayair`), it doesn't break the API for our users in production. The environment (or

stage) names in this case are just strings and have no special significance. We could've called them development and prod instead.

We are however creating completely new apps when we deploy to a different environment. This is another advantage of the SST workflow. The infrastructure as code idea makes it easy to replicate to new environments. And the pay per use model of serverless means that we are not charged for these new environments unless we actually use them.

## Commit the Changes

As we work through the guide we'll save our changes.

◆ **CHANGE** Commit what we have and push our changes to GitHub.

```
$ git add .
$ git commit -m "Initial commit"
$ git push
```

Now we are ready to create the backend for our notes app.



### Help and discussion

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

**Create your AWS resources**

# Create a DynamoDB Table in SST

We are now going to start creating our infrastructure in SST. Starting with DynamoDB.

## Create a Table

◆ CHANGE Add the following to our `infra/storage.ts`.

```
// Create the DynamoDB table
export const table = new sst.aws.Dynamo("Notes", {
  fields: {
    userId: "string",
    noteId: "string",
  },
  primaryIndex: { hashKey: "userId", rangeKey: "noteId" },
});
```

Let's go over what we are doing here.

We are using the `Dynamo` component to create our DynamoDB table.

It has two fields:

1. `userId`: The id of the user that the note belongs to.
2. `noteId`: The id of the note.

We are then creating an index for our table.

Each DynamoDB table has a primary key. This 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 (referenced by `primaryIndex` in code block above) 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.

## Deploy Changes

After you make your changes, SST will automatically create the table. You should see something like this at the end of the deploy process.

```
| Created      Notes sst:aws:Dynamo
```

**Info:** You'll need to make sure you have `sst dev` running, if not then restart it by running `npx sst dev`.

Now that our database has been created, let's create an S3 bucket to handle file uploads.



### Help and discussion

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

# Create an S3 Bucket in SST

We'll be storing the files that's uploaded by our users to an S3 bucket. The template we are using comes with a bucket that we renamed back in the [Create a Hello World API](#).

Recall the following from `infra/storage.ts`.

```
// Create an S3 bucket
export const bucket = new sst.aws.Bucket("Uploads");
```

Here we are creating a new S3 bucket using the SST `Bucket` component.

## Commit the Changes

◆ **CHANGE** Let's commit and push our changes to GitHub.

```
$ git add .
$ git commit -m "Adding storage"
$ git push
```

Next, let's create the API for our notes app.



### Help and discussion

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

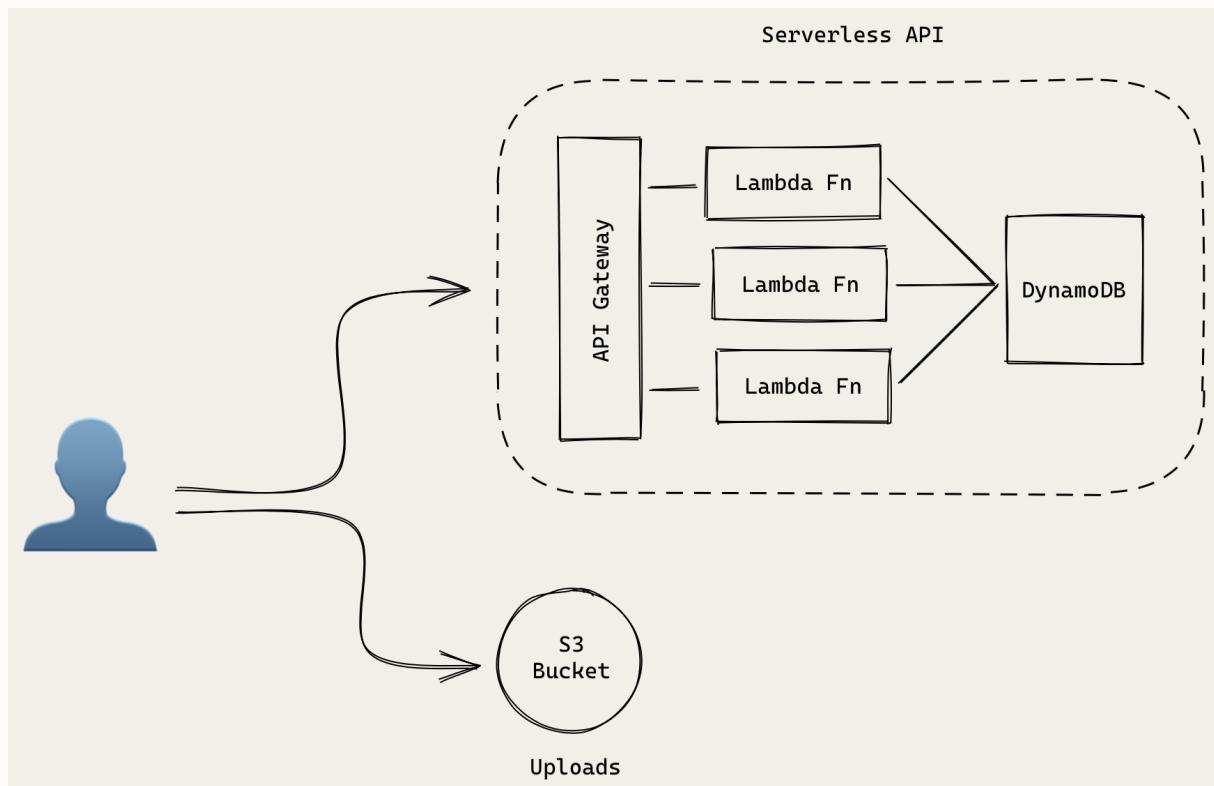
# **Building a serverless API**

# Review Our App Architecture

So far we've deployed our simple [Hello World API](#), created a database ([DynamoDB](#)), and [created an S3 bucket for file uploads](#). We are ready to start working on our backend API but let's get a quick sense of how the aforementioned pieces fit together.

## Notes App API Architecture

Our notes app backend will look something like this.



There are a couple of things of note here:

1. Our database is not exposed publicly and is only invoked by our Lambda functions.
2. But our users will be uploading files directly to the S3 bucket that we created.

The second point is something that is different from a lot of traditional server based architectures. We are typically used to uploading the files to our server and then moving them to a file server. But here we will be directly uploading it to our S3 bucket. We will look at this in more detail when we look at file uploads.

In the coming sections will also be looking at how we can secure access to these resources. We will be setting it up such that only our authenticated users will be allowed to access these resources.

Now that we have a good idea of how our app will be architected, let's get to work!



#### Help and discussion

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

# Add an API to Create a Note

Let's get started by creating the API for our notes app.

We'll first add 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).

## Creating the API

◆ CHANGE Replace the `infra/api.ts` with the following.

```
import { table } from "./storage";

// Create the API
export const api = new sst.aws.ApiGatewayV2("Api", {
  transform: {
    route: {
      handler: {
        link: [table],
      },
    }
  }
});

api.route("POST /notes", "packages/functions/src/create.main");
```

We are doing a couple of things of note here.

- We are creating an API using SST's `Api` component. It creates an [Amazon API Gateway HTTP API](#).
- We are [linking](#) our DynamoDB table to our API using the `link` prop. This will allow our API to access our table.

- The first route we are adding to our API is the POST /notes route. It'll be used to create a note.
- By using the transform prop we are telling the API that we want the given props to be applied to all the routes in our API.

## Add the Function

Now let's add the function that'll be creating our note.

◆ CHANGE Create a new file in packages/functions/src/create.ts with the following.

```
import * as uuid from "uuid";
import { Resource } from "sst";
import { APIGatewayProxyEvent } from "aws-lambda";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const dynamoDb = DynamoDBDocumentClient.from(new DynamoDBClient({}));

export async function main(event: APIGatewayProxyEvent) {
  let data, params;

  // Request body is passed in as a JSON encoded string in 'event.body'
  if (event.body) {
    data = JSON.parse(event.body);
    params = {
      TableName: Resource.Notes.name,
      Item: {
        // The attributes of the item to be created
        userId: "123", // The id of the author
        noteId: uuid.v1(), // A unique uuid
        content: data.content, // Parsed from request body
        attachment: data.attachment, // Parsed from request body
        createdAt: Date.now(), // Current Unix timestamp
      },
    };
  } else {
    return {
      statusCode: 400,
      body: "No request body provided",
    };
  }
}
```

```
    statusCode: 404,
    body: JSON.stringify({ error: true }),
  };
}

try {
  await dynamoDb.send(new PutCommand(params));
}

return {
  statusCode: 200,
  body: JSON.stringify(params.Item),
};

} catch (error) {
  let message;
  if (error instanceof Error) {
    message = error.message;
  } else {
    message = String(error);
  }
  return {
    statusCode: 500,
    body: JSON.stringify({ error: message }),
  };
}
}
```

There are some helpful comments in the code but let's go over them quickly.

- Parse the input from the `event.body`. This represents the HTTP request body.
- It contains the contents of the note, as a string — `content`.
- It also contains an `attachment`, if one exists. It's the filename of a file that will be uploaded to our [S3 bucket](#).
- We can access our linked DynamoDB table through `Resource.Notes.name` using the [SST SDK](#). Here, `Notes` in `Resource.Notes`, is the name of our Table component from the [Create a DynamoDB Table in SST](#) chapter. By doing `link: [table]` earlier in this chapter, we are allowing our API to access our table.
- The `userId` is the id for the author of the note. For now we are hardcoding it to 123. Later we'll be setting this based on the authenticated user.

- Make a call to DynamoDB to put a new object with a generated noteId and the current date as the createdAt.
- And if the DynamoDB call fails then return an error with the HTTP status code 500.

Let's go ahead and install the packages that we are using here.

◆ CHANGE Navigate to the functions folder in your terminal.

```
$ cd packages/functions
```

◆ CHANGE Then, run the following **in the packages/functions/ folder**.

```
$ npm install uuid @aws-sdk/lib-dynamodb @aws-sdk/client-dynamodb  
$ npm install -D @types/uuid @types/aws-lambda
```

- **uuid** generates unique ids.
- **@types/aws-lambda** & **@types/uuid** provides the TypeScript types.
- **@aws-sdk/lib-dynamodb** **@aws-sdk/client-dynamodb** allows us to talk to DynamoDB

## Deploy Our Changes

If you switch over to your terminal, you will notice that your changes are being deployed.

**Caution:** You'll need to have `sst dev` running for this to happen. If you had previously stopped it, then running `npx sst dev` will deploy your changes again.

Once complete, you should see.

+ Complete  
Api: <https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com>

## Test the API

Now we are ready to test our new API.

◆ CHANGE Run the following in your terminal.

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"content":"Hello World","attachment":"hello.jpg"}' \
<YOUR_Api>/notes
```

Replace <YOUR\_Api> with the Api from the output above. For example, our command will look like:

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"content":"Hello World","attachment":"hello.jpg"}' \
https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com/notes
```

Here we are making a POST request to our create note API. We are passing in the content and attachment as a JSON string. In this case the attachment is a made up file name. We haven't uploaded anything to S3 yet.

**Info:** Make sure to keep your local environment, `sst dev` running in another window.

The response should look something like this.

```
{"userId":"123","noteId":"a46b7fe0-008d-11ec-a6d5-
 ↳ a1d39a077784","content":"Hello
 ↳ World","attachment":"hello.jpg","createdAt":1629336889054}
```

Make a note of the `noteId`. 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 refactor this code. Since we'll be doing the same basic actions for all of our APIs, it makes sense to move this into our `core` package.

◆ **CHANGE** Start by replacing our `create.ts` with the following.

```
import * as uuid from "uuid";
import { Resource } from "sst";
import { Util } from "@notes/core/util";
```

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const dynamoDb = DynamoDBDocumentClient.from(new DynamoDBClient({}));

export const main = Util.handler(async (event) => {
  let data = {
    content: "",
    attachment: "",
  };

  if (event.body != null) {
    data = JSON.parse(event.body);
  }

  const params = {
    TableName: Resource.Notes.name,
    Item: {
      // The attributes of the item to be created
      userId: "123", // The id of the author
      noteId: uuid.v1(), // A unique uuid
      content: data.content, // Parsed from request body
      attachment: data.attachment, // Parsed from request body
      createdAt: Date.now(), // Current Unix timestamp
    },
  };
}

await dynamoDb.send(new PutCommand(params));

return JSON.stringify(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.
- 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.



Create a `packages/core/src/util/index.ts` file with the following.

```
import { Context, APIGatewayProxyEvent } from "aws-lambda";\n\nexport module Util {\n    export function handler(\n        lambda: (evt: APIGatewayProxyEvent, context: Context) => Promise<string>\n    ) {\n        return async function(event: APIGatewayProxyEvent, context: Context) {\n            let body: string, statusCode: number;\n\n            try {\n                // Run the Lambda\n                body = await lambda(event, context);\n                statusCode = 200;\n            } catch (error) {\n                statusCode = 500;\n                body = JSON.stringify({\n                    error: error instanceof Error ? error.message : String(error),\n                });\n            }\n\n            // Return HTTP response\n            return {\n                body,\n                statusCode,\n            };\n        };\n    }\n}
```



We are now using the Lambda types in core as well. Run the following **in the `packages/core/` directory**.

```
$ npm install -D @types/aws-lambda
```

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 take 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.
- Exporting the whole thing inside a `Util` module allows us import it as `Util.handler`. It also lets us put other util functions in this module in the future.

**Caution:** You'll need to have `sst dev` running for this to happen. If you had previously stopped it, then running `npx sst dev` will deploy your changes again.

## Remove Template Files

The template we are using comes with some example files that we can now remove.

◆ CHANGE Run the following from the **project root**.

```
$ rm -rf packages/core/src/example packages/functions/src/api.ts
```

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

---

## Common Issues

- path received type undefined

Restarting `npx sst dev` should pick up the new type information and resolve this error.

- Response `statusCode: 500`

If you see a `statusCode: 500` response when you invoke your function, the error has been reported by our code in the `catch` block. You'll see a `console.error` is included in our `util/index.ts` code above. Adding logs like these can help give you insight on issues and how to resolve them.

```
} catch (e) {  
  // Prints the full error  
  console.error(e);
```

```
body = { error: e.message };
statusCode = 500;
}
```



### Help and discussion

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

# Add an API to Get a Note

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 in `packages/functions/src/get.ts` with the following:

```
import { Resource } from "sst";
import { Util } from "@notes/core/util";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { GetCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const dynamoDb = DynamoDBDocumentClient.from(new DynamoDBClient({}));

export const main = Util.handler(async (event) => {
  const params = {
    TableName: Resource.Notes.name,
    // 'Key' defines the partition key and sort key of
    // the item to be retrieved
    Key: {
      userId: "123", // The id of the author
      noteId: event?.pathParameters?.id, // The id of the note from the path
    },
  };

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

```
// Return the retrieved item
return JSON.stringify(result.Item);
});
```

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

## Add the route

Let's add a new route for the get note API.

◆ CHANGE Add the following below the POST `/notes` route in `infra/api.ts`.

```
api.route("GET /notes/{id}", "packages/functions/src/get.main");
```

## Deploy Our Changes

If you switch over to your terminal, you will notice that your changes are being deployed.

**Info:** You'll need to have `sst dev` running for this to happen. If you had previously stopped it, then running `npx sst dev` will deploy your changes again.

You should see that the new API has been deployed.

+ Complete  
Api: <https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com>

## Test the API

Let's test the get notes API. In the [previous chapter](#) we tested our create note API. It should've returned the new note's id as the `noteId`.

◆ CHANGE Run the following in your terminal.

```
$ curl https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com/notes/<NOTE_ID>
```

Make sure to replace the endpoint URL with your `ApiEndpoint` value and the at the end of the URL with the `noteId` that was created previously.

Since we are making a simple GET request, we could also go to this URL directly in your browser.

The response should look something like this.

```
{"attachment": "hello.jpg", "content": "Hello\n    World", "createdAt": 1629336889054, "noteId": "a46b7fe0-008d-11ec-a6d5-\n    a1d39a077784", "userId": "123"}
```

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 an API to List All the Notes

Now we are going to add an API that returns a list of all the notes a user has. This'll be very similar to the [previous chapter](#) where we were returning a single note.

## Add the Function



Create a new file in `packages/functions/src/list.ts` with the following.

```
import { Resource } from "sst";
import { Util } from "@notes/core/util";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { QueryCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const dynamoDb = DynamoDBDocumentClient.from(new DynamoDBClient({}));

export const main = Util.handler(async (event) => {
  const params = {
    TableName: Resource.Notes.name,
    // 'KeyConditionExpression' defines the condition for the query
    // - 'userId = :userId': only return items with matching 'userId'
    //   partition key
    KeyConditionExpression: "userId = :userId",
    // 'ExpressionAttributeValues' defines the value in the condition
    // - ':userId': defines 'userId' to be the id of the author
    ExpressionAttributeValues: {
      ":userId": "123",
    },
  };
  const result = await dynamoDb.send(new QueryCommand(params));
});
```

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

This is pretty much the same as our `get.ts` except we use a condition to only return the items that have the same `userId` as the one we are passing in. In our case, it's still hardcoded to 123.

## Add the Route

Let's add the route for this new endpoint.

◆ CHANGE Add the following above the `POST /notes` route in `infra/api.ts`.

```
api.route("GET /notes", "packages/functions/src/list.main");
```

## Deploy Our Changes

If you switch over to your terminal, you will notice that your changes are being deployed.

**Info:** You'll need to have `sst dev` running for this to happen. If you had previously stopped it, then running `npx sst dev` will deploy your changes again.

You should see that the new API has been deployed.

+ Complete

```
Api: https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com
```

## Test the API

Let's test list all notes API.

◆ CHANGE Run the following in your terminal.

```
$ curl https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com/notes
```

Again, replacing the example URL with your Api value.

Since we are making a simple GET request, we could also go to this URL directly in your browser.

The response should look something like this.

```
[{"attachment": "hello.jpg", "content": "Hello\n    World", "createdAt": 1629336889054, "noteId": "a46b7fe0-008d-11ec-a6d5-\n    a1d39a077784", "userId": "123"}]
```

Note that, we are getting an array of notes. Instead of a single note.

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 API to Update a Note

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

## Add the Function

◆ CHANGE Create a new file in `packages/functions/src/update.ts` and paste the following.

```
import { Resource } from "sst";
import { Util } from "@notes/core/util";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { UpdateCommand, DynamoDBDocumentClient } from
  "@aws-sdk/lib-dynamodb";

const dynamoDb = DynamoDBDocumentClient.from(new DynamoDBClient({}));

export const main = Util.handler(async (event) => {
  const data = JSON.parse(event.body || "{}");

  const params = {
    TableName: Resource.Notes.name,
    Key: {
      // The attributes of the item to be created
      userId: "123", // The id of the author
      noteId: event?.pathParameters?.id, // The id of the note from the path
    },
    // '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,
    }
  };
  const result = await dynamoDb.update(params).promise();
  return Util.success(result);
});
```

```
    ":content": data.content || null,  
},  
};  
  
await dynamoDb.send(new UpdateCommand(params));  
  
return JSON.stringify({ status: true });  
});
```

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

## Add the Route

Let's add a new route for the get note API.

◆ CHANGE Add the following below the GET `/notes/{id}` route in `infra/api.ts`.

```
api.route("PUT /notes/{id}", "packages/functions/src/update.main");
```

## Deploy Our Changes

If you switch over to your terminal, you will notice that your changes are being deployed.

**Info:** You'll need to have `sst dev` running for this to happen. If you had previously stopped it, then running `npx sst dev` will deploy your changes again.

You should see that the new API has been deployed.

+ Complete  
Api: <https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com>

## Test the API

Now we are ready to test the new API. In [an earlier chapter](#) we tested our create note API. It should've returned the new note's id as the `noteId`.

◆ CHANGE Run the following in your terminal.

```
$ curl -X PUT \
-H 'Content-Type: application/json' \
-d '{"content":"New World","attachment":"new.jpg"}' \
https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com/notes/<NOTE_ID>
```

Make sure to replace the id at the end of the URL with the noteId from before.

Here we are making a PUT request to a note that we want to update. We are passing in the new content and attachment as a JSON string.

The response should look something like this.

```
{"status":true}
```

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



### Help and discussion

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

# Add an API to Delete a Note

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

## Add the Function

◆ CHANGE Create a new file in `packages/functions/src/delete.ts` and paste the following.

```
import { Resource } from "sst";
import { Util } from "@notes/core/util";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DeleteCommand, DynamoDBDocumentClient } from
  "@aws-sdk/lib-dynamodb";

const dynamoDb = DynamoDBDocumentClient.from(new DynamoDBClient({}));

export const main = Util.handler(async (event) => {
  const params = {
    TableName: Resource.Notes.name,
    Key: {
      userId: "123", // The id of the author
      noteId: event?.pathParameters?.id, // The id of the note from the path
    },
  };

  await dynamoDb.send(new DeleteCommand(params));

  return JSON.stringify({ status: true });
});
```

This makes a DynamoDB delete call with the `userId` & `noteId` key to delete the note. We are still hard coding the `userId` for now.

## Add the Route

Let's add a new route for the delete note API.

◆ CHANGE Add the following below the PUT /notes{id} route in infra/api.ts.

```
api.route("DELETE /notes/{id}", "packages/functions/src/delete.main");
```

## Deploy Our Changes

If you switch over to your terminal, you will notice that your changes are being deployed.

**Info:** You'll need to have `sst dev` running for this to happen. If you had previously stopped it, then running `npx sst dev` will deploy your changes again.

You should see that the new API stack has been deployed.

```
✓ Deployed:  
  StorageStack  
  ApiStack  
  ApiEndpoint: https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com
```

## Test the API

Let's test the delete note API.

In a [previous chapter](#) we tested our create note API. It should've returned the new note's id as the `noteId`.

◆ CHANGE Run the following in your terminal.

```
$ curl -X DELETE  
↳ https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com/notes/<NOTE_ID>
```

Make sure to replace the id at the end of the URL with the `noteId` from before.

Here we are making a DELETE request to the note that we want to delete. The response should look something like this.

```
{"status":true}
```

## Commit the Changes

◆ CHANGE Let's commit and push our changes to GitHub.

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

So our API is publicly available, this means that anybody can access it and create notes. And it's always connecting to the 123 user id. Let's fix these next by handling users and authentication.



### Help and discussion

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

# **Users and authentication**

# Auth in Serverless Apps

In the last section, we created a serverless REST API and deployed it. But there are a couple of things missing.

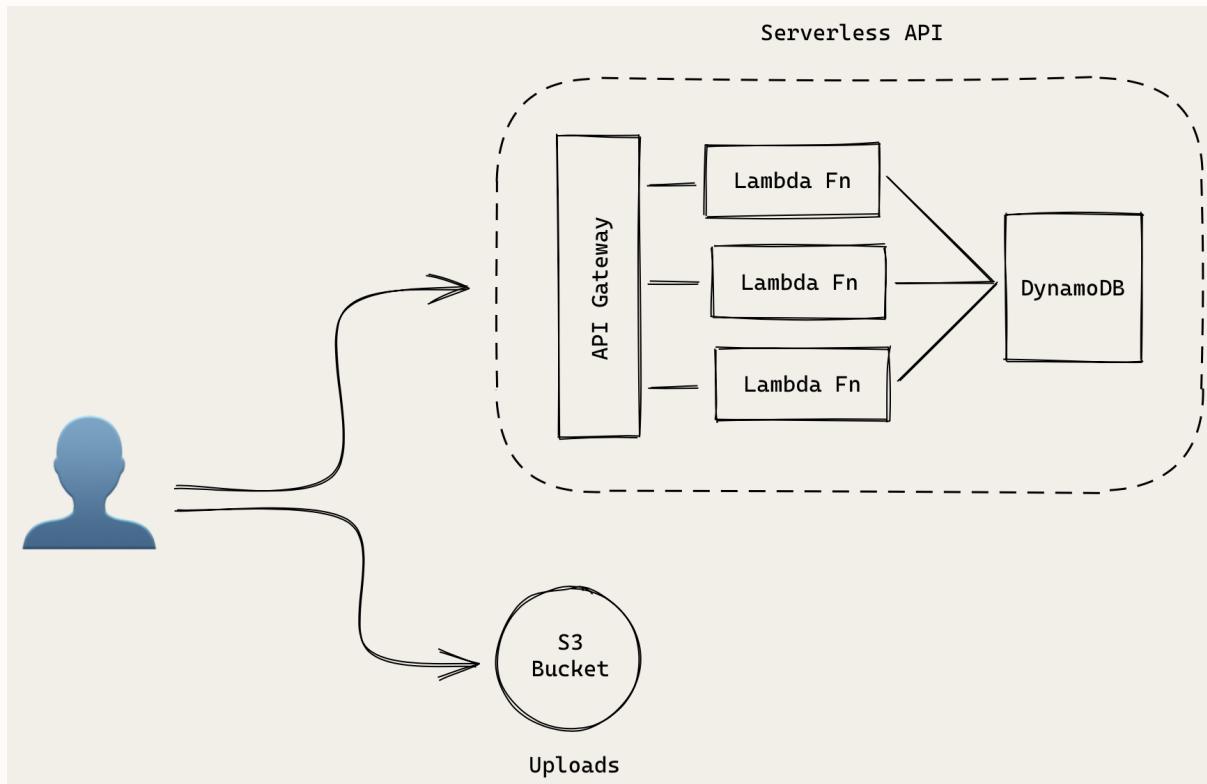
1. It's not secure
2. And, it's not linked to a specific user

These two problems are connected. We need a way to allow users to sign up for our notes app and then only allow authenticated users to access it.

In this section we are going to learn to do just that. Starting with getting an understanding of how authentication (and access control) works in the AWS world.

## Public API Architecture

For reference, here is what we have so far.



Serverless public API architecture

Our users make a request to our serverless API. It starts by hitting our API Gateway endpoint. And depending on the endpoint we request, it'll forward that request to the appropriate Lambda function.

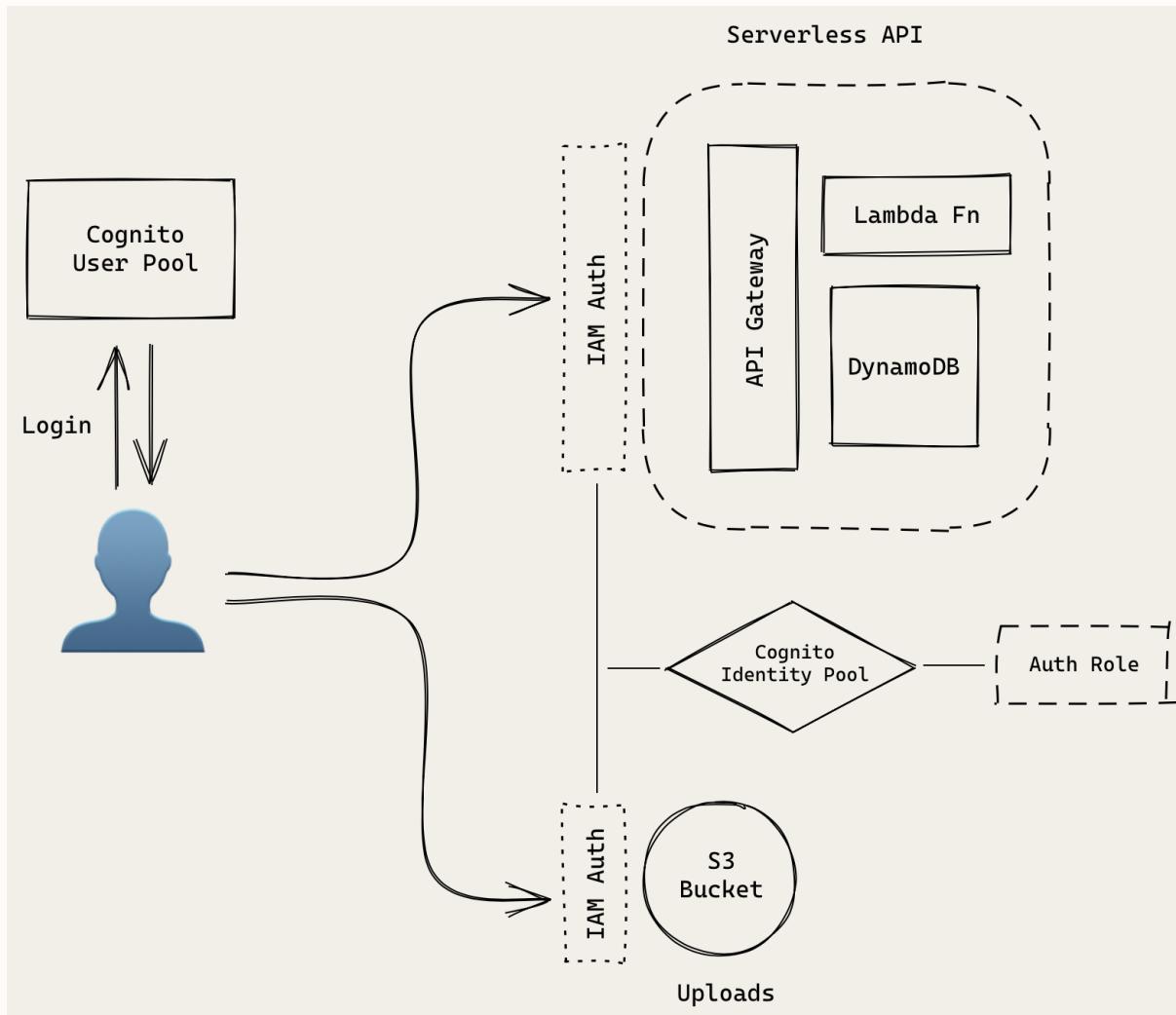
In terms of access control, our API Gateway endpoint is allowed to invoke the Lambda functions we listed in the routes of our `infra/api.ts`. And if you'll recall, our Lambda functions are allowed to connect to our DynamoDB tables.

```
link: [table],
```

For uploading files, our users will directly upload them to the [S3 bucket](#). While we'll look at how our frontend React app uploads files later in the guide, in this section we need to make sure that we secure access to it.

## Authenticated API Architecture

To allow users to sign up for our notes app and to secure our infrastructure, we'll be moving to an architecture that looks something like this.



Serverless Auth API architecture

There's a bit more going on here. So let's go over all the separate parts in detail.

A couple of quick notes before we jump in:

1. The *Serverless API* portion in this diagram is exactly the same as the one we looked at before. It's just simplified for the purpose of this diagram.
2. Here the user effectively represents our React app or the *client*.

## Cognito User Pool

To manage sign up and login functionality for our users, we'll be using an AWS service called, [Amazon Cognito User Pool](#). It'll store our user's login info. It'll also be managing user sessions in our React app.

## Cognito Identity Pool

To manage access control to our AWS infrastructure we use another service called [Amazon Cognito Identity Pools](#). This service decides if our previously authenticated user has access to the resources he/she is trying to connect to. Identity Pools can have different authentication providers (like Cognito User Pools, Facebook, Google etc.). In our case, our Identity Pool will be connected to our User Pool.

If you are a little confused about the differences between a User Pool and an Identity Pool, don't worry. We've got a chapter to help you with just that — [Cognito User Pool vs Identity Pool](#)

## Auth Role

Our Cognito Identity Pool has a set of rules (called an IAM Role) attached to it. It'll list out the resources an authenticated user is allowed to access. These resources are listed using an ID called ARN.

We've got a couple of chapters to help you better understand IAMs and ARNs in detail:

- [What is IAM](#)
- [What is an ARN](#)

But for now our authenticated users use the Auth Role in our Identity Pool to interact with our resources. This will help us ensure that our logged in users can only access our notes API. And not any other API in our AWS account.

## Authentication Flow

Let's look at how the above pieces work together in practice.

### Sign up

A user will sign up for our notes app by creating a new User Pool account. They'll use their email and password. They'll be sent a code to verify their email. This will be handled between our React app and User Pool. No other parts of our infrastructure are involved in this.

## Login

A signed up user can now login using their email and password. Our React app will send this info to the User Pool. If these are valid, then a session is created in React.

## Authenticated API Requests

To connect to our API.

1. The React client makes a request to API Gateway secured using IAM Auth.
2. API Gateway will check with our Identity Pool if the user has authenticated with our User Pool.
3. It'll use the Auth Role to figure out if this user can access this API.
4. If everything looks good, then our Lambda function is invoked and it'll pass in an Identity Pool user id.

## S3 File Uploads

Our React client will be directly uploading files to our S3 bucket. Similar to our API; it'll also check with the Identity Pool to see if we are authenticated with our User Pool. And if the Auth Role has access to upload files to the S3 bucket.

## Alternative Authentication Methods

It's worth quickly mentioning that there are other ways to secure your APIs. We mentioned above that an Identity Pool can use Facebook or Google as an authentication provider. So instead of using a User Pool, you can use Facebook or Google. We have an Extra Credits chapter on Facebook specifically — [Facebook Login with Cognito using AWS Amplify](#)

You can also directly connect the User Pool to API Gateway. The downside with that is that you might not be able to manage access control centrally to the S3 bucket (or any other AWS resources in the future).

Finally, you can manage your users and authentication yourself. This is a little bit more complicated and we are not covering it in this guide. Though we might expand on it later.

Now that we've got a good idea how we are going to handle users and authentication in our serverless app, let's get started by adding the auth infrastructure to our app.

**Help and discussion**

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

# Adding Auth to Our Serverless App

So far we've created the [DynamoDB table](#), [S3 bucket](#), and [API](#) parts of our serverless backend. Now let's add auth into the mix. As we talked about in the [previous chapter](#), we are going to use [Cognito User Pool](#) to manage user sign ups and logins. While we are going to use [Cognito Identity Pool](#) to manage which resources our users have access to.

Setting this all up can be pretty complicated in Terraform. SST has simple [CognitoUserPool](#) and [CognitoIdentityPool](#) components to help with this.

## Create the Components

◆ **CHANGE** Add the following to a new file in `infra/auth.ts`.

```
import { api } from "./api";
import { bucket } from "./storage";

const region = aws.getRegionOutput().name;

export const userPool = new sst.aws.CognitoUserPool("UserPool", {
  usernames: ["email"]
});

export const userPoolClient = userPool.addClient("UserPoolClient");

export const identityPool = new sst.aws.CognitoIdentityPool("IdentityPool", {
  userPools: [
    {
      userPool: userPool.id,
      client: userPoolClient.id,
    },
  ],
},
```

```
permissions: {
  authenticated: [
    {
      actions: ["s3:*"],
      resources: [
        $concat(bucket.arn,
          "/private/${cognito-identity.amazonaws.com:sub}/*"),
      ],
    },
    {
      actions: [
        "execute-api:*",
      ],
      resources: [
        $concat(
          "arn:aws:execute-api:",
          region,
          ":",
          aws.getCallerIdentityOutput({}).accountId,
          ":",
          api.nodes.api.id,
          "/*/*/*"
        ),
      ],
    },
  ],
},
});
```

Let's go over what we are doing here.

- The `CognitoUserPool` component creates a Cognito User Pool for us. We are using the `usernames` prop to state that we want our users to login with their email.
- We are using `addClient` to create a client for our User Pool. You create one for each “*client*” that’ll connect to it. Since we only have a frontend we only need one. You can later add another if you add a mobile app for example.
- The `CognitoIdentityPool` component creates an Identity Pool. The

`attachPermissionsForAuthUsers` function allows us to specify the resources our authenticated users have access to.

- We want them to access our S3 bucket and API. Both of which we are importing from `api.ts` and `storage.ts` respectively. We'll look at this in detail below.

## Securing Access

We are creating an IAM policy to allow our authenticated users to access our API. You can [learn more about IAM here](#).

```
{  
  actions: [  
    "execute-api:*",  
  ],  
  resources: [  
    $concat(  
      "arn:aws:execute-api:",  
      region,  
      ":" ,  
      aws.getcalleridentityoutput({}).accountid,  
      ":" ,  
      api.nodes.api.id,  
      "/*/*/*"  
    ),  
  ],  
},
```

This looks a little complicated but Amazon API Gateway has a format it uses to define its endpoints. We are building that here.

We are also creating a specific IAM policy to secure the files our users will upload to our S3 bucket.

```
// Policy granting access to a specific folder in the bucket  
new iam.PolicyStatement({  
  actions: ["s3:*"],  
  effect: iam.Effect.ALLOW,  
  resources: [
```

```
    bucket.bucketArn + "/private/${cognito-identity.amazonaws.com:sub}/*",
],
}),
},
```

Let's look at how this works.

In the above policy we are granting our logged in users access to the path `private/${cognito-identity.amazonaws.com:sub}/` within our S3 bucket's ARN. 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 allows us to separate access to our user's file uploads within the same S3 bucket.

One other thing to note is that, the federated identity id is a UUID that is assigned by our Identity Pool. This id is different from the one that a user is assigned in a User Pool. This is because you can have multiple authentication providers. The Identity Pool federates these identities and gives each user a unique id.

## Add to the Config

Let's add this to our `sst.config.ts`.

◆ CHANGE Add this below the `await import("./infra/api")` line in your `sst.config.ts`.

```
const auth = await import("./infra/auth");

return {
  UserPool: auth.userPool.id,
  Region: aws.getRegionOutput().name,
  IdentityPool: auth.identityPool.id,
  UserPoolClient: auth.userPoolClient.id,
};
```

Here we are importing our new config and the `return` allows us to print out some useful info about our new auth resources in the terminal.

## Add Auth to the API

We also need to enable authentication in our API.

◆ CHANGE Add the following prop into the transform options below the handler: { block in `infra/api.ts`.

```
args: {
  auth: { iam: true }
},
```

So it should look something like this.

```
// Create the API
export const api = new sst.aws.ApiGatewayV2("Api", {
  transform: {
    route: {
      handler: {
        link: [table],
      },
      args: {
        auth: { iam: true }
      },
    }
  }
});
```

This tells our API that we want to use AWS\_IAM across all our routes.

## Deploy Our Changes

If you switch over to your terminal, you will notice that your changes are being deployed.

**Info:** You'll need to have `sst dev` running for this to happen. If you had previously stopped it, then running `npx sst dev` will deploy your changes again.

You should see that the new auth resources are being deployed.

```
+ Complete
  Api: https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com
  ---
```

```
IdentityPool: us-east-1:9bd0357e-2acl-418d-a609-bc5e7bc064e3
Region: us-east-1
UserPool: us-east-1_TYEz7XP7P
UserPoolClient: 3fetogamdv9aqa0393adsd7viv
```

Let's create a test user so that we can test our API.

## Create a Test User

We'll use AWS CLI to sign up a user with their email and password.

◆ CHANGE In your terminal, run.

```
$ aws cognito-identity sign-up \
--region <COGNITO_REGION> \
--client-id <USER_POOL_CLIENT_ID> \
--username admin@example.com \
--password Passw0rd!
```

Make sure to replace COGNITO\_REGION and USER\_POOL\_CLIENT\_ID with the Region and UserPoolClient from above.

Now we need to verify this email. For now we'll do this via an administrator command.

◆ CHANGE In your terminal, run.

```
$ aws cognito-identity admin-confirm-sign-up \
--region <COGNITO_REGION> \
--user-pool-id <USER_POOL_ID> \
--username admin@example.com
```

Replace the COGNITO\_REGION and USER\_POOL\_ID with the Region and UserPool from above.

**Caution:** The first command uses the USER\_POOL\_CLIENT\_ID while the second command uses the USER\_POOL\_ID. Make sure to replace it with the right values.

Now that the auth infrastructure and a test user has been created, let's use them to secure our APIs and test them.



**Help and discussion**

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

# Secure Our Serverless APIs

Now that our APIs have been [secured with Cognito User Pool and Identity Pool](#), we are ready to use the authenticated user's info in our Lambda functions.

Recall that we've been hard coding our user ids so far (with user id 123). We'll need to grab the real user id from the Lambda function event.

## Cognito Identity Id

Recall the function signature of a Lambda function:

```
export async function main(event: APIGatewayProxyEvent, context: Context) {}
```

Or the refactored version that we are using:

```
export const main = Util.handler(async (event) => {});
```

So far we've used the event object to get the path parameters (event.pathParameters) and request body (event.body).

Now we'll get the id of the authenticated user.

```
event.requestContext.authorizer?.iam.cognitoIdentity.identityId;
```

This is an id that's assigned to our user by our Cognito Identity Pool.

You'll also recall that so far all of our APIs are hard coded to interact with a single user.

```
userId: "123", // The id of the author
```

Let's change that.



Replace the above line in packages/functions/src/create.ts with.

```
userId: event.requestContext.authorizer?.iam.cognitoIdentity.identityId,
```

◆ CHANGE Do the same in these files:

- packages/functions/src/get.ts
- packages/functions/src/update.ts
- packages/functions/src/delete.ts

◆ CHANGE In packages/functions/src/list.ts find this line instead.

":userId": "123",

◆ CHANGE And replace it with.

```
":userId": event.requestContext.authorizer?.iam.cognitoIdentity.identityId,
```

Keep in mind that the `userId` above is the Federated Identity id (or Identity Pool user id). This is not the user id that is assigned in our User Pool. If you want to use the user's User Pool user Id instead, have a look at the [Mapping Cognito Identity Id and User Pool Id](#) chapter.

To test these changes we cannot use the `curl` command anymore. We'll need to generate a set of authentication headers to make our requests. Let's do that next.

## Test the APIs

Let's quickly test our APIs with authentication.

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](#).

```
$ npx aws-api-gateway-cli-test \
--user-pool-id='<USER_POOL_ID>' \
--app-client-id='<USER_POOL_CLIENT_ID>' \
--cognito-region='<COGNITO_REGION>' \
--identity-pool-id='<IDENTITY_POOL_ID>' \
--invoke-url='<API_ENDPOINT>' \
--api-gateway-region='<API_REGION>' \
--username='admin@example.com' \
--password='Passw0rd!' \
--path-template='/notes' \
--method='POST' \
--body='{"content":"hello world","attachment":"hello.jpg"}'
```

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 above.
- Replace `USER_POOL_ID`, `USER_POOL_CLIENT_ID`, `COGNITO_REGION`, and `IDENTITY_POOL_ID` with the `UserPool`, `UserPoolClient`, `Region`, and `IdentityPool` from our [previous chapter](#).
- Replace the `API_ENDPOINT` with the `Api` from back when we [created our API](#).
- And for the `API_REGION` you can use the same Region as we used above. Since our entire app is deployed to the same region.

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. We won't need to do this when we connect from our React.js app.

**Info:** If you are on Windows, you can use the command below. The spaces between each option are very important.

```
$ npx aws-api-gateway-cli-test --username admin@example.com --password
↪ Passw0rd! --user-pool-id <USER_POOL_ID> --app-client-id
↪ <USER_POOL_CLIENT_ID> --cognito-region <COGNITO_REGION>
↪ --identity-pool-id <IDENTITY_POOL_ID> --invoke-url <API_ENDPOINT>
↪ --api-gateway-region <API_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-1:06d418dd-b55b-4f7d-9af4-5d067a69106e',
    noteId: 'b5199840-c0e5-11ec-a5e8-61c040911d73',
    content: 'hello world',
    attachment: 'hello.jpg',
    createdAt: 1650485336004
  }
}
```

## Commit the Changes

◆ CHANGE Let's commit and push our changes to GitHub.

```
$ git add .
$ git commit -m "Securing the API"
$ git push
```

We've now got a serverless API that's secure and handles user authentication. In the next section we are going to look at how we can work with 3rd party APIs in serverless. And how to handle secrets!



### Help and discussion

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

## **Working with secrets**

# Setup a Stripe Account

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.

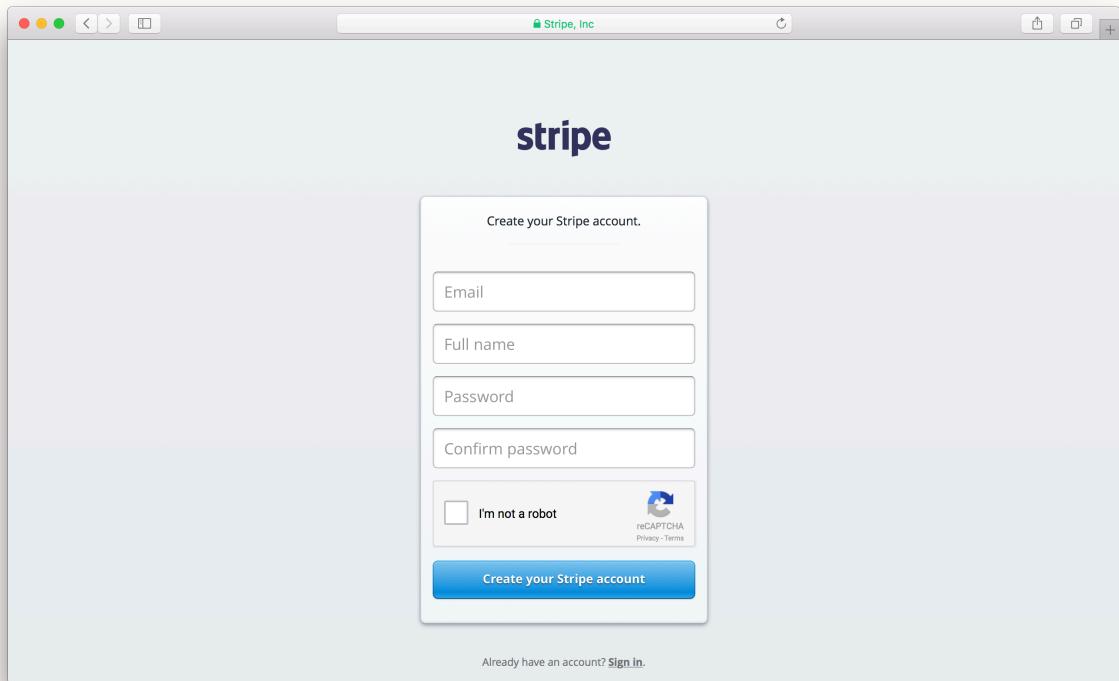
A common extension of the notes app (that we've noticed) is to add a billing API that works with Stripe. In the case of our notes app we are going to allow our users to pay a fee for storing a certain number of notes. The flow is going to look something like this:

1. The user is going to select the number of notes they want to store and puts in their 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.

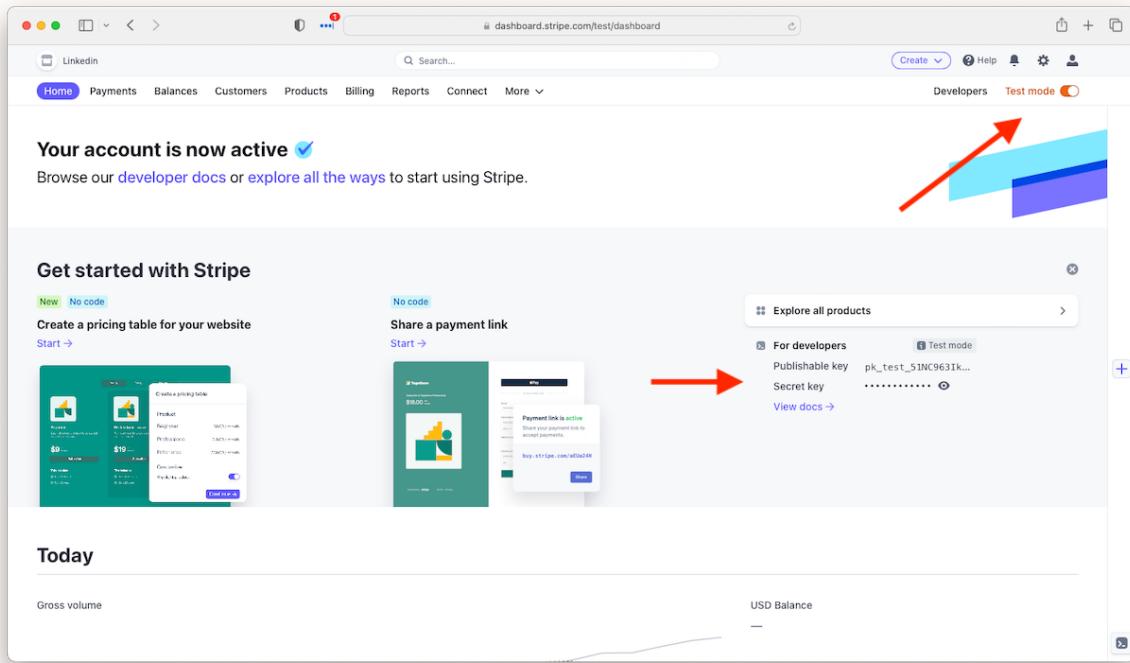
## Sign up for Stripe

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 with a confirmed account, you will be able to use the developer tools.



Stripe dashboard screenshot

The first thing to do is switch to test mode. This is important because we don't want to charge our credit card every time we test our app.

The second thing to note is that Stripe has automatically generated a test and live **Publishable key** and a test and live **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.

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

Next, let's use this in our SST app.



### Help and discussion

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

# Handling Secrets in SST

In the [previous chapter](#), we created a Stripe account and got a pair of keys. Including the Stripe secret key. We need this in our app but we do not want to store this secret in our code. In this chapter, we'll look at how to add secrets in SST.

We will be using the `sst secret` CLI to store our secrets.

◆ CHANGE Run the following in your project root.

```
$ npx sst secret set StripeSecretKey <YOUR_STRIPE_SECRET_TEST_KEY>
```

**Note:** You can specify the stage for a secret. By default, the stage is your personal stage.

You can run `npx sst secret list` to see the secrets for the current stage.

Now that the secret is stored, we can add it into our config using the `Secret` component.

◆ CHANGE Add the following to your `infra/storage.ts`:

```
// Create a secret for Stripe
export const secret = new sst.Secret("StripeSecretKey");
```

◆ CHANGE Import `secret` in `infra/api.ts`. Replace the following.

```
import { table } from "./storage";
```

◆ CHANGE With:

```
import { table, secret } from "./storage";
```

◆ CHANGE Next, link `StripeSecretKey` to the API in `infra/api.ts`. Replace this:

```
link: [table],
```

◆ CHANGE With:

```
link: [table, secret],
```

This will add `StripeSecretKey` in our infrastructure. And allow our API to access the secret.

Now we are ready to add an API to handle billing.



### Help and discussion

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

# Add an API to Handle Billing

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

## Add a Billing Lambda

Start by installing the Stripe npm package.

◆ CHANGE Run the following **in the packages/functions/ directory** of our project.

```
$ npm install stripe
```

◆ CHANGE Create a new file in packages/functions/src/billing.ts with the following.

```
import Stripe from "stripe";
import { Resource } from "sst";
import { Util } from "@notes/core/util";
import { Billing } from "@notes/core/billing";

export const main = Util.handler(async (event) => {
  const { storage, source } = JSON.parse(event.body || "{}");
  const amount = Billing.compute(storage);
  const description = "Scratch charge";

  const stripe = new Stripe(
    // Load our secret key
    Resource.StripeSecretKey.value,
    { apiVersion: "2024-06-20" }
  );
```

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

return JSON.stringify({ 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 `Billing.compute(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 getting this from the secret that we configured in the [previous chapter](#). At the time of writing, we are using `apiVersion 2024-06-20` but you can check the [Stripe docs](#) for the latest version.
- Finally, we use the `stripe.charges.create()` function 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 in our forums](#).

## Add the Business Logic

Now let's implement our `Billing.compute` function. This is primarily the *business logic* in our app.



Create a `packages/core/src/billing/index.ts` and add the following.

```
export module Billing {
  export function compute(storage: number) {
    const rate = storage <= 10 ? 4 : storage <= 100 ? 2 : 1;
```

```
    return rate * storage * 100;  
}  
}
```

A *module* is a good way to organize our business logic. You want to create modules for the various *domains* in your app. This follows some basic principles of [Domain-driven design](#).

The `compute` function 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!

## Add the Route

Let's add a new route for our billing API.

◆ **CHANGE** Add the following below the `DELETE /notes/{id}` route in `infra/api.ts`.

```
api.route("POST /billing", "packages/functions/src/billing.main");
```

## Deploy Our Changes

If you switch over to your terminal, you will notice that your changes are being deployed.

**Info:** You'll need to have `sst dev` running for this to happen. If you had previously stopped it, then running `npx sst dev` will deploy your changes again.

You should see that the new API stack has been deployed.

```
+ Complete  
Api: https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com  
...
```

## Test the Billing API

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

We'll be using the same CLI from [a few chapters ago](#).

◆ **CHANGE** Run the following in your terminal.

```
$ npx aws-api-gateway-cli-test \
--username='admin@example.com' \
--password='Passw0rd!' \
--user-pool-id='<USER_POOL_ID>' \
--app-client-id='<USER_POOL_CLIENT_ID>' \
--cognito-region='<COGNITO_REGION>' \
--identity-pool-id='<IDENTITY_POOL_ID>' \
--invoke-url='<API_ENDPOINT>' \
--api-gateway-region='<API_REGION>' \
--path-template='/billing' \
--method='POST' \
--body='{"source":"tok_visa","storage":21}'
```

**Note:** Make sure to replace the USER\_POOL\_ID, USER\_POOL\_CLIENT\_ID, COGNITO\_REGION, IDENTITY\_POOL\_ID, API\_ENDPOINT, and API\_REGION with the [same values we used a couple of chapters ago](#).

If you have the previous request, update the path-template and body with the new values.

Here we are 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](#).

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: { status: true } }
```

## Commit the Changes

◆ **CHANGE** Let's commit and push our changes to GitHub.

```
$ git add .
$ git commit -m "Adding a billing API"
$ git push
```

Now that we have our new billing API ready. Let's look at how to setup unit tests in serverless. We'll be using that to ensure that our infrastructure and business logic has been configured correctly.



### Help and discussion

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

## **Serverless unit tests**

# Unit Tests in Serverless

In this chapter we'll look at how to write unit tests for our serverless app. Typically you might want to test some of your *business logic*.

The template we are using comes with a setup to help with that. It uses [Vitest](#) for this.

## Writing Tests

We are going to test the business logic that we added in the [previous chapter](#) to compute how much to bill a user.

◆ **CHANGE** Create a new file in `packages/core/src/billing/test/index.test.ts` and add the following.

```
import { test, expect } from "vitest";
import { Billing } from "../";

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

  const cost = 4000;
  const expectedCost = Billing.compute(storage);

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

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

  const cost = 20000;
  const expectedCost = Billing.compute(storage);
```

```
expect(cost).toEqual(expectedCost);  
});  
  
test("Highest tier", () => {  
  const storage = 101;  
  
  const cost = 10100;  
  const expectedCost = Billing.compute(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.

## Run Tests

Now let's run these tests.

◆ CHANGE Run the following in the **packages/core/** directory.

```
$ npm test
```

You should see something like this:

```
✓ src/billing/test/index.test.ts (3)  
  ✓ Lowest tier  
  ✓ Middle tier  
  ✓ Highest tier
```

```
Test Files 1 passed (1)  
Tests 3 passed (3)
```

Internally this is running `sst shell vitest`. The `sst shell` CLI connects any linked resources. This ensures that your tests have the same kind of access as the rest of your application code.

**Info:** You'll need to Ctrl-C to quit the test runner. It's useful to have when you are working on them as it'll reload your tests.

And that's it! We have unit tests all configured. These tests are fairly simple but should give you an idea of how to add more in the future.

## Commit the Changes

◆ **CHANGE** Let's commit our changes and push it to GitHub.

```
$ git add .  
$ git commit -m "Adding unit tests"  
$ git push
```

Now we are almost ready to move on to our frontend. But before we do, we need to ensure that our backend is configured so that our React app will be able to connect to it.



### Help and discussion

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

# **CORS in serverless**

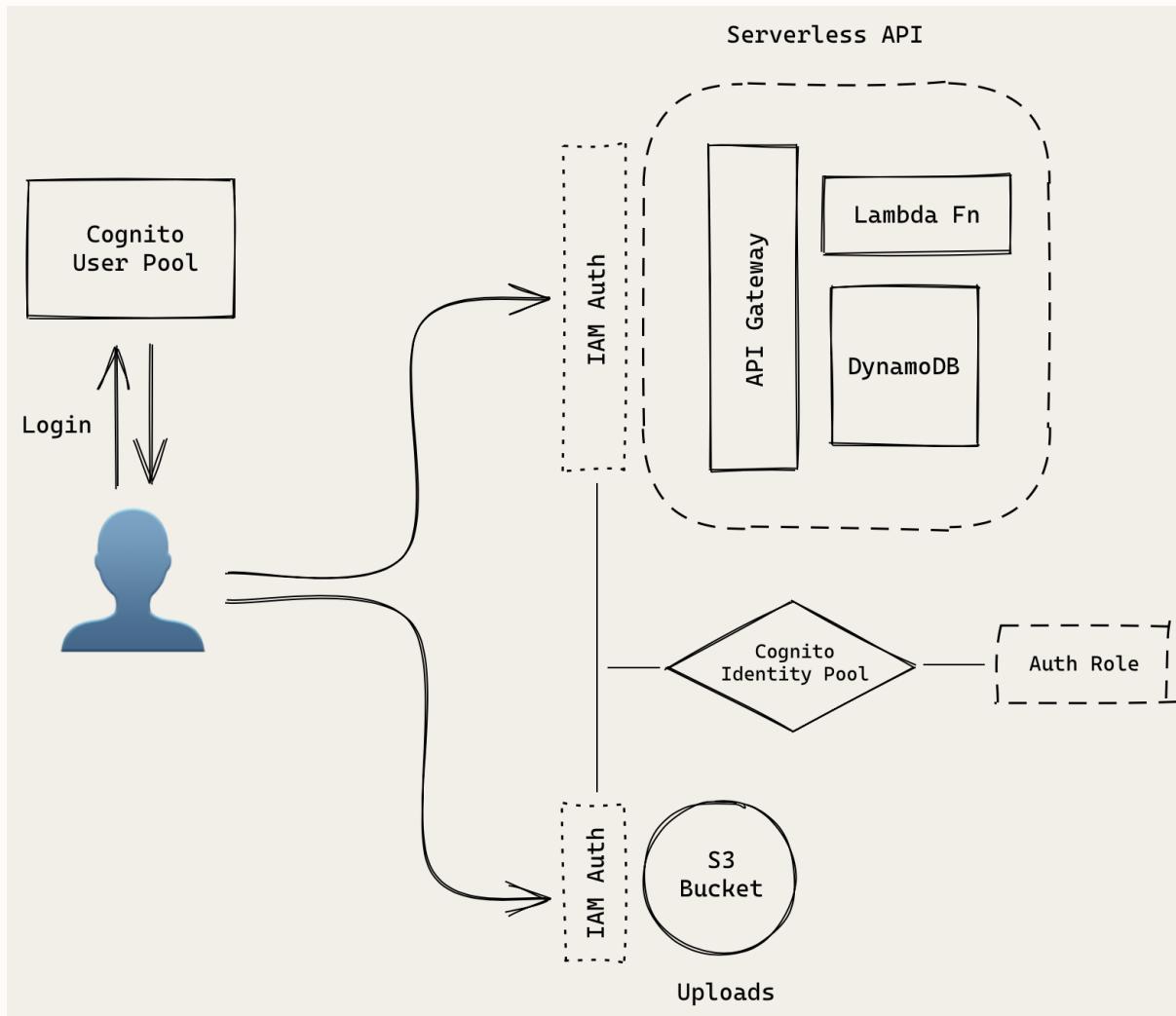
## Handle CORS in Serverless APIs

Let's take stock of our setup so far. We have a serverless API backend that allows users to create notes and an S3 bucket where they can upload files. We are now almost ready to work on our frontend React app.

However, before we can do that. There is one thing that needs to be taken care of — [CORS or Cross-Origin Resource Sharing](#).

Since our React app is going to be run inside a browser (and most likely hosted on a domain separate from our serverless API and S3 bucket), we need to configure CORS to allow it to connect to our resources.

Let's quickly review our backend app architecture.



Serverless Auth API architecture

Our client will be interacting with our API, S3 bucket, and User Pool. CORS in the User Pool part is taken care of by its internals. That leaves our API and S3 bucket. In the next couple of chapters we'll be setting that up.

Let's get a quick background on CORS.

## Understanding CORS

There are two things we need to do to support CORS in our serverless API.

1. Preflight OPTIONS requests

For certain types of cross-domain requests (PUT, DELETE, ones with Authentication headers, etc.), your browser will first make a *preflight* request using the request method OPTIONS. These need to respond with the domains that are allowed to access this API and the HTTP methods that are allowed.

## 2. Respond with CORS headers

For all the other types of requests we need to make sure to include the appropriate CORS headers. These headers, just like the one above, need to include the domains that are allowed.

There's a bit more to CORS than what we have covered here. So make sure to [check out the Wikipedia article for further details](#).

If we don't set the above up, then we'll see something like this in our HTTP responses.

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

And our browser won't show us the HTTP response. This can make debugging our API extremely hard.

## CORS in API Gateway

The [ApiGatewayV2](#) component that we are using enables CORS by default.

```
new sst.aws.ApiGatewayV2("Api", {  
  // Enabled by default  
  cors: true  
});
```

You can further configure the specifics if necessary. You can [read more about this here](#).

```
new sst.aws.ApiGatewayV2("Api", {  
  cors: {  
    allowMethods: ["GET"]  
  }  
});
```

We'll go with the default setting for now.

## CORS Headers in Lambda Functions

Next, we need to add the CORS headers in our Lambda function response.



Replace the return statement in our packages/core/src/handler.ts.

```
return {  
  body,  
  statusCode,  
};
```



With the following.

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

Again you can customize the CORS headers but we'll go with the default ones here.

The two steps we've taken above ensure that if our Lambda functions are invoked through API Gateway, it'll respond with the proper CORS config.

Next, let's add these CORS settings to our S3 bucket as well. Since our frontend React app will be uploading files directly to it.



### Help and discussion

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

# Handle CORS in S3 for File Uploads

In the notes app we are 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.

Similar to the [previous chapter](#), the `Bucket` component enables CORS by default.

```
new sst.aws.Bucket("Uploads", {  
    // Enabled by default  
    cors: true,  
});
```

You can configure this further. [Read more about this here](#).

```
new sst.aws.Bucket("Uploads", {  
    cors: {  
        allowMethods: ["GET"]  
    }  
});
```

## Commit the Changes



Let's commit our changes and push it to GitHub.

```
$ git add .  
$ git commit -m "Enabling CORS"  
$ git push
```

Now we are ready to use our serverless backend to create our frontend React app!



### Help and discussion

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

## **Setting up a React app**

# Create a New React.js App

We are now ready to work on our frontend. So far we've built and deployed our backend API and infrastructure. We are now going to build a web app that connects to our backend.

We are going to create a single page app using [React.js](#). We'll use the [Vite](#) project to set everything up.

## Create a New React App

◆ CHANGE Run the following command **in the packages/ directory**.

```
$ npm create vite@latest frontend -- --template react-ts
```

**Note:** Make sure you use the extra -- in the command.

This will create your new project in the frontend / directory.

◆ CHANGE Let's update the name of the package in the packages/frontend/package.json. Replace this:

```
- "name": "frontend",
+ "name": "@notes/frontend",
```

Make sure to use the name of your app instead of notes.

◆ CHANGE Now install the dependencies.

```
$ cd frontend
$ npm install
```

This should take a second to run.

We also need to make a small change to our Vite config to bundle our frontend.

◆ CHANGE Add the following below the plugins: [react()], line in packages/frontend/vite.config.ts.

```
build: {  
  // NOTE: Needed when deploying  
  chunkSizeWarningLimit: 800,  
},
```

## Add the React App to SST

We are going to be deploying our React app to AWS. To do that we'll be using the SST [StaticSite](#) component.

◆ CHANGE Create a new file in infra/web.ts and add the following.

```
import { api } from "./api";  
import { bucket } from "./storage";  
import { userPool, identityPool, userPoolClient } from "./auth";  
  
const region = aws.getRegionOutput().name;  
  
export const frontend = new sst.aws.StaticSite("Frontend", {  
  path: "packages/frontend",  
  build: {  
    output: "dist",  
    command: "npm run build",  
  },  
  environment: {  
    VITE_REGION: region,  
    VITE_API_URL: api.url,  
    VITE_BUCKET: bucket.name,  
    VITE_USER_POOL_ID: userPool.id,  
    VITE_IDENTITY_POOL_ID: identityPool.id,  
    VITE_USER_POOL_CLIENT_ID: userPoolClient.id,  
  },  
});
```

We are doing a couple of things of note here:

1. We are pointing our `StaticSite` component to the `packages/ frontend/` directory where our React app is.
2. We are passing in the outputs from our other components as [environment variables in Vite](#). This means that we won't have to hard code them in our React app. The `VITE_*` prefix is a convention Vite uses to say that we want to access these in our frontend code.

## Adding to the app

Let's add this to our config.

◆ **CHANGE** Add this below the `await import("./infra/api");` line in your `sst.config.ts`.

```
await import("./infra/web");
```

## Deploy Our Changes

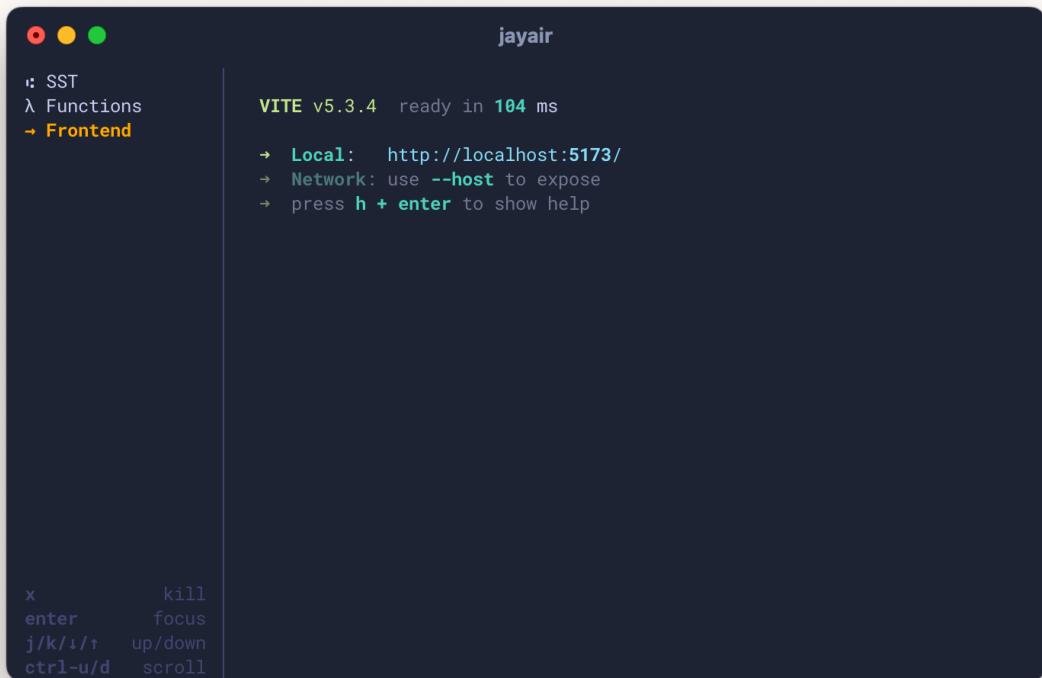
If you switch over to your terminal, you will notice that your changes are being deployed.

**Info:** You'll need to have `sst dev` running for this to happen. If you had previously stopped it, then running `npx sst dev` will deploy your changes again.

```
+ Complete
  Api: https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com
  Frontend: https://d1wyq46yoha2b6.cloudfront.net
  ...
  ...
```

## Starting the React App

The `sst dev` CLI will automatically start our React frontend by running `npm run dev`. It also passes in the environment variables that we have configured above.



sst dev CLI starts frontend

You can click on **Frontend** in the sidebar or navigate to it.

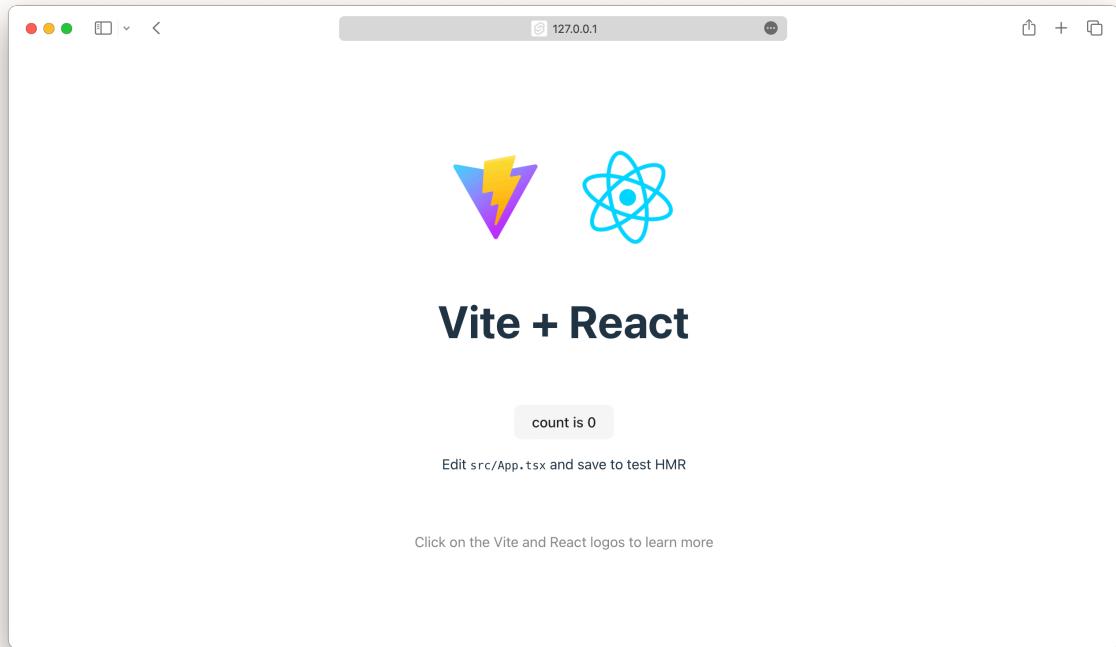
This should show where your frontend is running locally.

VITE v5.3.4 ready in 104 ms

```
\faLongArrowAltRight Local: http://127.0.0.1:5173/
\faLongArrowAltRight Network: use --host to expose
\faLongArrowAltRight press h + enter to show help
```

**Info:** SST doesn't deploy your frontend while you are working locally. This is because most frontends come with their own local dev environments.

If you head to that URL in your browser you should see.



New Vite React App screenshot

## Change the Title

Let's quickly change the title of our note taking app.

◆ CHANGE Open up packages/frontend/index.html and edit the title tag to the following:

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

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



### Help and discussion

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

## Add App Favicons

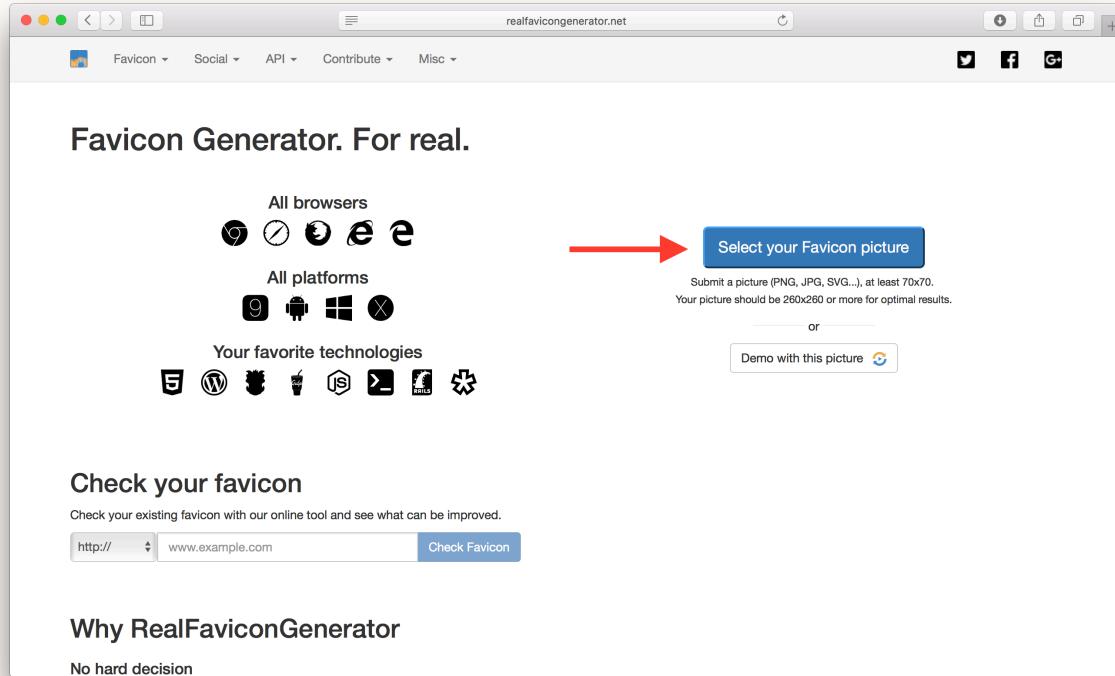
Vite generates a simple favicon for our app and places it in `public/vite.svg` of our app. However, getting the favicon to work on all browsers and mobile platforms requires a little more work. There are quite a few different requirements and dimensions. And this gives us a good opportunity to learn 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. Or head over to this link to download it — `[{{'..../assets/scratch-icon.png' | absolute_url }}]({{ '..../assets/scratch-icon.png' | absolute_url }})`

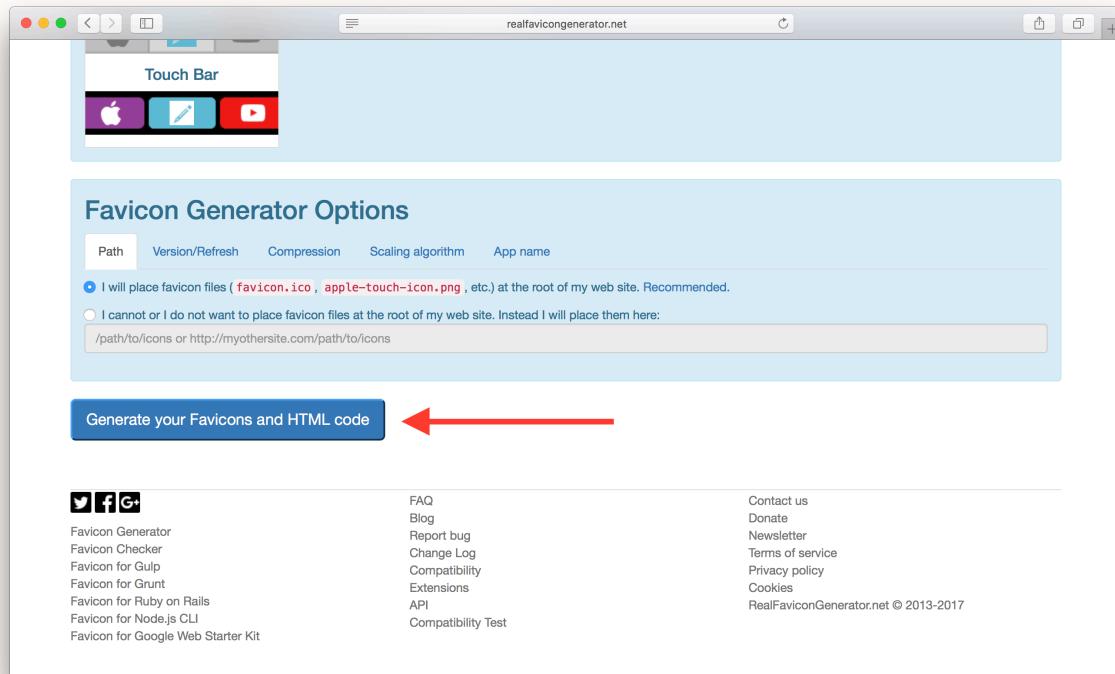
To ensure that our icon works for most of our targeted platforms we'll use a service called the [Favicon Generator](#).

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



Realfavicongenerator.net screenshot

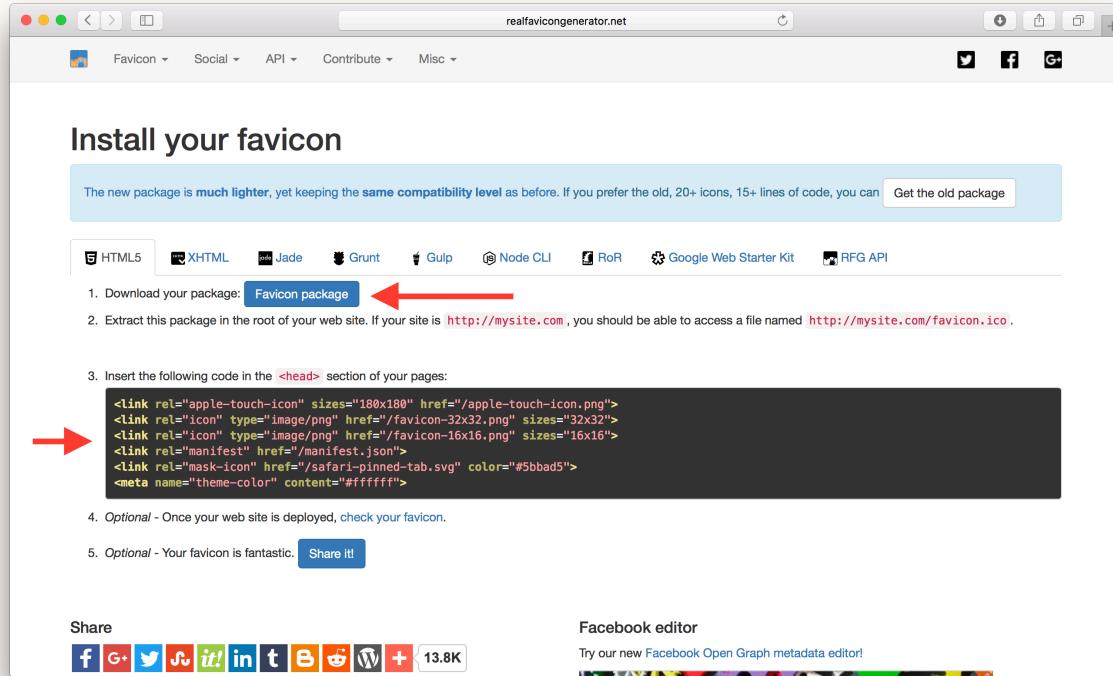
Once you upload your icon, it'll show you a preview of your icon on various platforms. Scroll down the page and hit the **Generate your Favicons and HTML code** button.



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

Let's remove the old icons files.

**Note:** We'll be working exclusively **in the packages/ frontend/ directory** for the rest of the frontend part of the guide.

◆ **CHANGE** Then replace the contents of `public/site.webmanifest` with the following:

```
{
  "short_name": "Scratch",
  "name": "Scratch Note Taking App",
  "icons": [
    {
      "src": "android-chrome-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "android-chrome-256x256.png",
      "sizes": "256x256",
      "type": "image/png"
    }
  ]
}
```

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

◆ **CHANGE** Add this to the <head> in your public/index.html.

```
<link rel="apple-touch-icon" sizes="180x180" href="/apple-touch-icon.png">
<link rel="icon" type="image/png" sizes="32x32" href="/favicon-32x32.png">
<link rel="icon" type="image/png" sizes="16x16" href="/favicon-16x16.png">
<link rel="manifest" href="/site.webmanifest">
<meta name="msapplication-TileColor" content="#da532c">
<meta name="theme-color" content="#ffffff">
<meta name="description" content="A simple note taking app" />
```

◆ **CHANGE** And **remove** the following line that references the original favicon.

```
<link rel="icon" type="image/svg+xml" href="/vite.svg" />
```

Finally head over to your browser and add /favicon-32x32.png to the base URL 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** 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 replace the current styles in `src/index.css` for the body tag to the following.

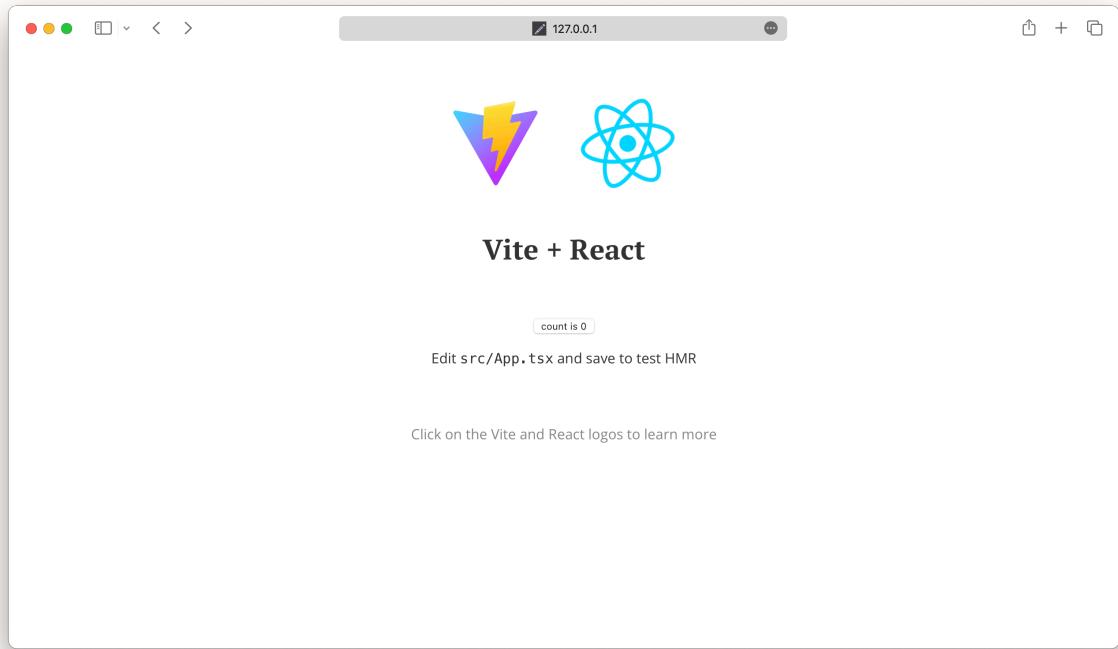
```
body {  
  margin: 0;  
  padding: 0;  
  color: #333;  
  font-size: 16px;  
  -moz-osx-font-smoothing: grayscale;  
  -webkit-font-smoothing: antialiased;  
  font-family: "Open Sans", sans-serif;  
}
```

**◆ 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.

We also need a couple of icons in our application. We'll be using the [React Icons](#) package for this. It allows us to include icons in our React app as standard React components.

## Installing React Bootstrap



Run the following command **in your packages/frontend/ directory**.

```
$ npm install bootstrap react-bootstrap react-icons  
$ npm install -D @types/bootstrap @types/react-bootstrap
```

This installs the packages and dependencies to the package.json of your React app.

## Add Bootstrap Styles

React Bootstrap uses the standard Bootstrap styles, so just import the style sheet.



**Add it above** the import "./index.css" line in src/main.tsx.

```
import "bootstrap/dist/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.



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

```
select.form-control,  
textarea.form-control,  
input.form-control {  
  font-size: 1rem;  
}  
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

# **Routes in React**

# Handle Routes with React Router

Since we are building a single page app, we are going to use [React Router](#) to handle the routes on the client side for us.

React Router allows us to specify a route like: `/login`. And specify a React Component that should be loaded when a user goes to that page.

Let's start by installing React Router.

## Installing React Router

◆ CHANGE Run the following command **in the packages/frontend/ directory**.

```
$ npm install react-router-dom
```

This installs the package and adds the dependency to `package.json` in your React app.

## 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.tsx`. 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 in `src/main.tsx`:

```
<React.StrictMode>
  <App />
</React.StrictMode>
```

◆ CHANGE With this:

```
<React.StrictMode>
  <Router>
    <App />
  </Router>
</React.StrictMode>
```



And import this in the header of `src/main.tsx`.

```
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” (or UI) of our app inside a component and render all the top level components inside them. We are calling the top level components that represent the various pages, 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** Go ahead and remove the code inside `src/App.tsx` and replace it with the following.

```
import Navbar from "react-bootstrap/Navbar";
import "./App.css";

function App() {
  return (
    <div className="App container py-3">
      <Navbar collapseOnSelect bg="light" expand="md" className="mb-3 px-3">
        <Navbar.Brand className="fw-bold text-muted">Scratch</Navbar.Brand>
        <Navbar.Toggle />
      </Navbar>
    </div>
  );
}

export default App;
```

We are doing a few things here:

1. Creating a fixed width container using Bootstrap in `div.container`.

- Using a couple of [Bootstrap spacing utility classes](#) (like `mb-#` and `py-#`) to add margin bottom (`mb`) and padding vertical (`py`). These use a proportional set of spacer units to give a more harmonious feel to our UI.

Let's clear out the styles that came with our template.

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

```
.App {  
}
```

For now we don't have any styles to add but we'll leave this file around, in case you want to add to it later.

Also, let's remove some unused template files.

◆ **CHANGE** Run the following **in the packages/ frontend/ directory**.

```
$ rm -r public/vite.svg src ../../assets/
```

## 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 the packages/ frontend/ 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.tsx`.

```
import "./Home.css";  
  
export default function Home() {
```

```
return (
  <div className="Home">
    <div className="lander">
      <h1>Scratch</h1>
      <p className="text-muted">A simple note taking app</p>
    </div>
  </div>
);
}
```

This 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;
}
```

## 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.tsx` and add the following into it.

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

export default function Links() {
  return (
    <Routes>
```

```
<Route path="/" element={<Home />} />
</Routes>
);
}
```

This component uses this `Routes` 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.tsx`.

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

◆ CHANGE And add the following line below our `Navbar` component inside `src/App.tsx`.

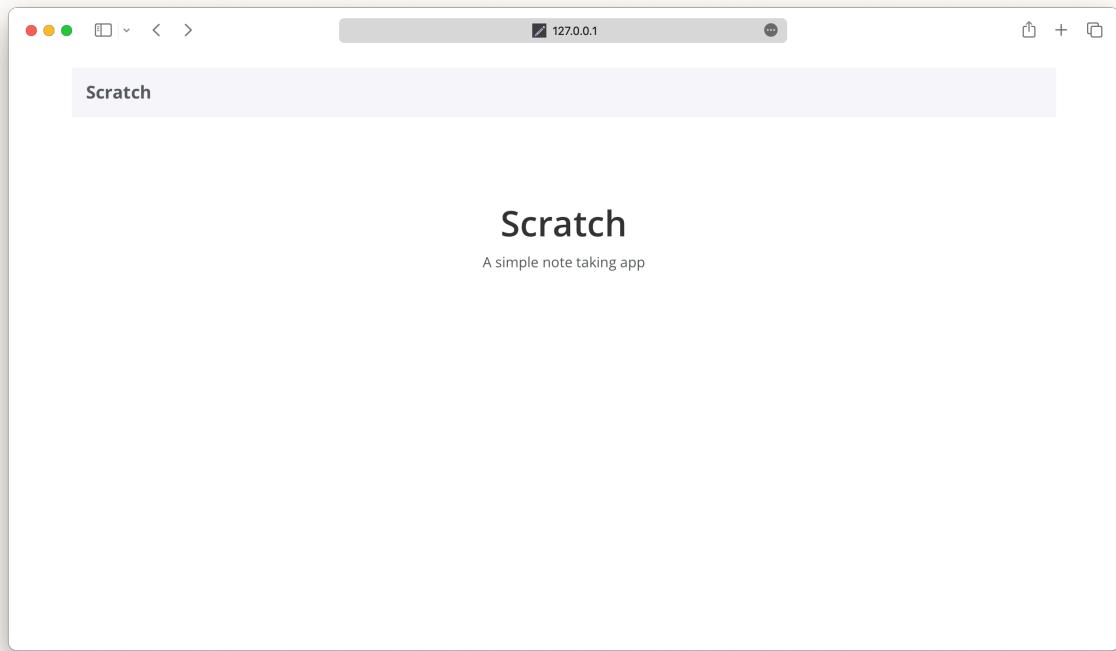
```
<Routes />
```

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

```
function App() {
  return (
    <div className="App container py-3">
      <Navbar collapseOnSelect bg="light" expand="md" className="mb-3 px-3">
        <Navbar.Brand className="fw-bold text-muted">Scratch</Navbar.Brand>
        <Navbar.Toggle />
      </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.tsx` with the following.

```
function App() {
  return (
    <div className="App container py-3">
      <Navbar collapseOnSelect bg="light" expand="md" className="mb-3 px-3">
        <Navbar.Brand className="fw-bold text-muted">Scratch</Navbar.Brand>
        <Navbar.Toggle />
        <Navbar.Collapse className="justify-content-end">
          <Nav>
            <Nav.Link href="/signup">Signup</Nav.Link>
            <Nav.Link href="/login">Login</Nav.Link>
          </Nav>
        </Navbar.Collapse>
      </Navbar>
      <Routes />
    </div>
  );
}
```

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

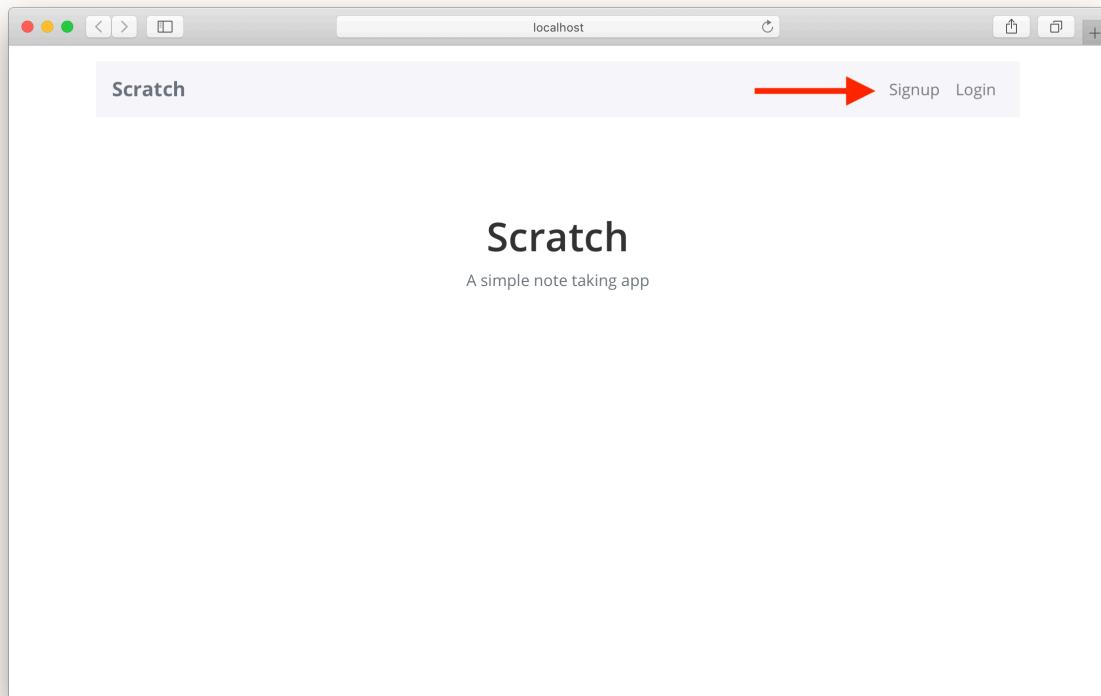
We also added a link to the *Scratch* logo. It links back to the homepage of our app.

And let's include the Nav component in the header.

◆ CHANGE Add the following import to the top of your `src/App.tsx`.

```
import Nav from "react-bootstrap/Nav";
```

Now if you flip over to your browser, you should see the 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 the packages/frontend/ directory**.

```
$ npm install react-router-bootstrap  
$ npm install -D @types/react-router-bootstrap
```

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

```
function App() {
  return (
    <div className="App container py-3">
      <Navbar collapseOnSelect bg="light" expand="md" className="mb-3 px-3">
        <LinkContainer to="/">
          <Navbar.Brand className="fw-bold text-muted">Scratch</Navbar.Brand>
        </LinkContainer>
        <Navbar.Toggle />
        <Navbar.Collapse className="justify-content-end">
          <Nav activeKey={window.location.pathname}>
            <LinkContainer to="/signup">
              <Nav.Link>Signup</Nav.Link>
            </LinkContainer>
            <LinkContainer to="/login">
              <Nav.Link>Login</Nav.Link>
            </LinkContainer>
          </Nav>
        </Navbar.Collapse>
      </Navbar>
      <Routes />
    </div>
  );
}
```

Let's also import it.

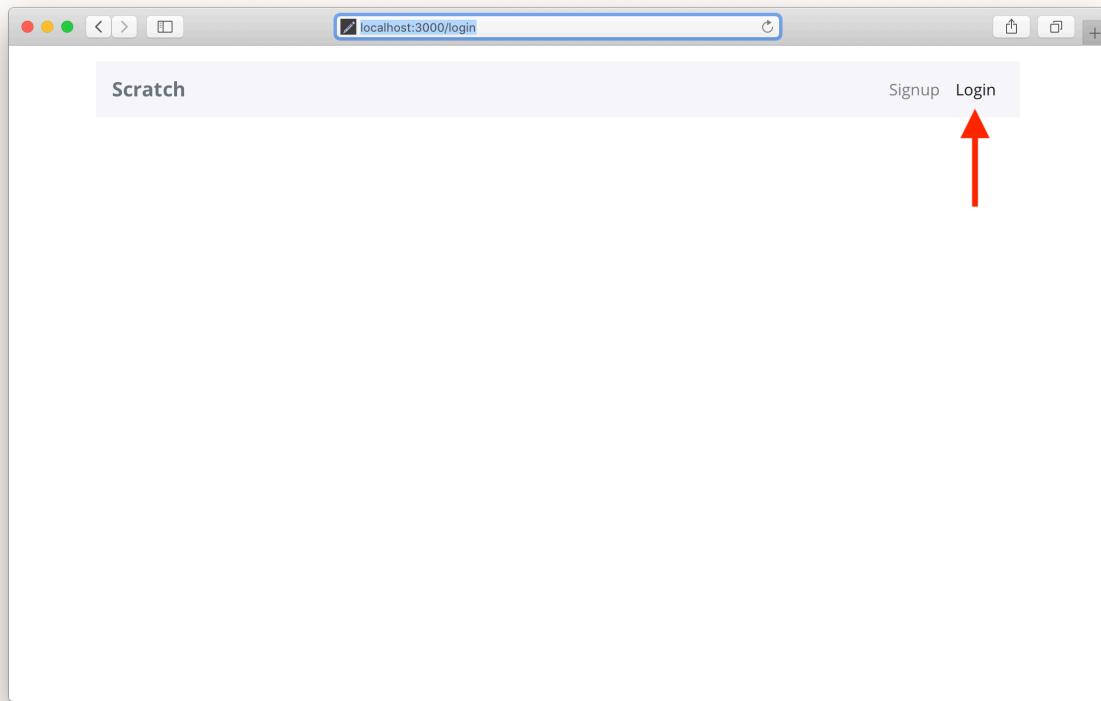
◆ **CHANGE** Add this to the top of your `src/App.tsx`.

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

We are doing one other thing here. We are grabbing the current path the user is on from the `window.location` object. And we set it as the `activeKey` of our `Nav` component. This'll highlight the link when we are on that page.

```
<Nav activeKey={window.location.pathname}>
```

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. These are cases when a user goes to a URL that we are not explicitly handling. We want to show a helpful sign to our users when this happens.

## 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.tsx` and add the following.

```
import "./NotFound.css";

export default function NotFound() {
  return (
    <div className="NotFound text-center">
      <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;
}
```

## Add a Catch All Route

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

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

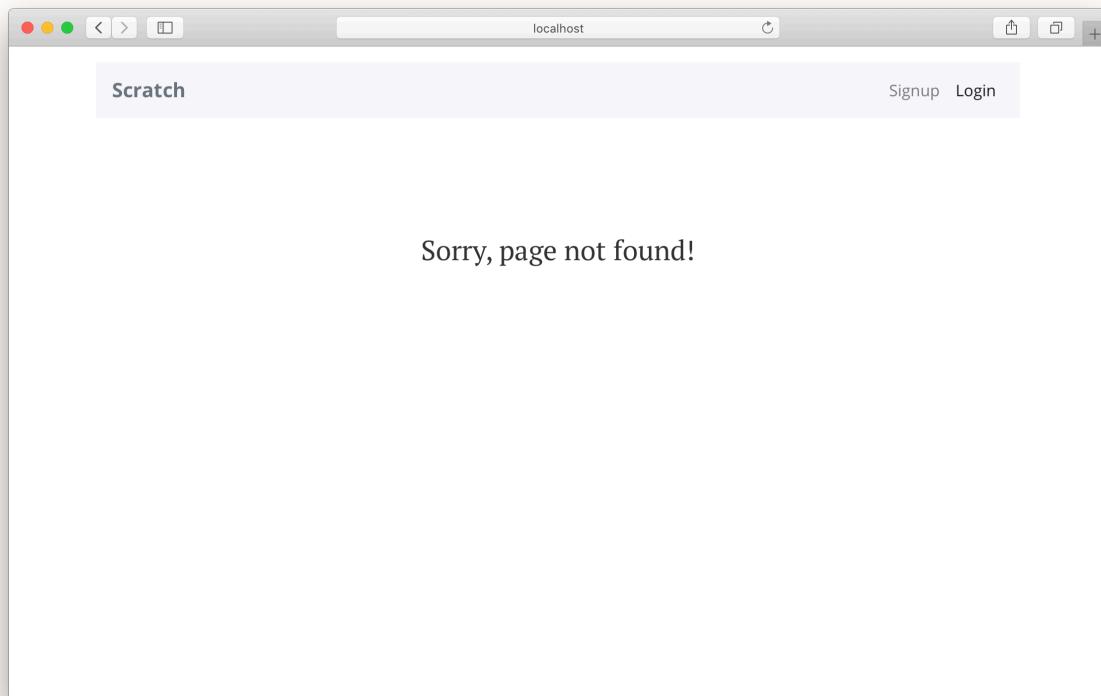
```
{/* Finally, catch all unmatched routes */}  
<Route path="/" element={<NotFound />} />;
```

This needs to always be the last route in the `<Routes>` 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.tsx";
```

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 allow our users to login and sign up for our app!



### Help and discussion

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

## **Adding auth to a React app**

# Configure AWS Amplify

In this section we are going to allow our users to login and sign up for our app. To do this we are going to start connecting the AWS resources that we created in the backend section.

To do this 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 the packages/frontend/ directory**.

```
$ npm install aws-amplify@^5
```

This installs the NPM package and adds the dependency to the package.json of your React app..

## Create a Config

Now, let's create a configuration file in frontend for our app that'll reference all the resources we have created.

◆ CHANGE Create a file at `frontend/src/config.ts` and add the following.

```
const config = {
  // Backend config
  s3: {
    REGION: import.meta.env.VITE_REGION,
    BUCKET: import.meta.env.VITE_BUCKET,
  },
  apiGateway: {
    REGION: import.meta.env.VITE_REGION,
```

```
URL: import.meta.env.VITE_API_URL,  
},  
cognito: {  
  REGION: import.meta.env.VITE_REGION,  
  USER_POOL_ID: import.meta.env.VITE_USER_POOL_ID,  
  APP_CLIENT_ID: import.meta.env.VITE_USER_POOL_CLIENT_ID,  
  IDENTITY_POOL_ID: import.meta.env.VITE_IDENTITY_POOL_ID,  
},  
};  
  
export default config;
```

Here we are loading the environment variables that are set from our serverless backend. We did this back when we were first [setting up our React app](#).

## Add AWS Amplify

Next we'll set up AWS Amplify.

◆ **CHANGE** To initialize AWS Amplify; add the following above the `ReactDOM.createRoot` line in `src/main.tsx`.

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

◆ CHANGE Import it by adding the following to the header of your `src/main.tsx`.

```
import { Amplify } from "aws-amplify";
```

◆ CHANGE And import the config we created above in the header of your `src/main.tsx`.

```
import config from "./config.ts";
```

Amplify has a [3 year old bug](#) that needs a workaround to use it with your frontend.

◆ CHANGE Add the following at the end of your `<head>` tags in `frontend/index.html`.

```
<script>  
  window.global = window;  
  var exports = {};  
</script>
```

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

# 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.tsx` and add the following.

```
import React, { useState } from "react";
import Form from "react-bootstrap/Form";
import Stack from "react-bootstrap/Stack";
import Button from "react-bootstrap/Button";
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: React.FormEvent<HTMLFormElement>) {
    event.preventDefault();
  }

  return (
    <Form>
      <Form.Group controlId="formEmail">
        <Form.Label>Email address
        <Form.Control type="email" value={email} onChange={setEmail}>
      </Form.Group>
      <Form.Group controlId="formPassword">
        <Form.Label>Password
        <Form.Control type="password" value={password} onChange={setPassword}>
      </Form.Group>
      <Form.Text>By signing in, you agree to our Terms of Service and Privacy Policy.</Form.Text>
      <Form.Button type="submit" onClick={handleSubmit}>Submit</Form.Button>
    </Form>
  );
}
```

```
<div className="Login">
  <Form onSubmit={handleSubmit}>
    <Stack gap={3}>
      <Form.Group controlId="email">
        <Form.Label>Email</Form.Label>
        <Form.Control
          autoFocus
          size="lg"
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}>
        />
      </Form.Group>
      <Form.Group controlId="password">
        <Form.Label>Password</Form.Label>
        <Form.Control
          size="lg"
          type="password"
          value={password}
          onChange={(e) => setPassword(e.target.value)}>
        />
      </Form.Group>
      <Button size="lg" type="submit" disabled={!validateForm()}>
        Login
      </Button>
    </Stack>
  </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.
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.



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



Now we link this container up with the rest of our app by adding the following line to `src/Routes.tsx` below our `<Home />` route.

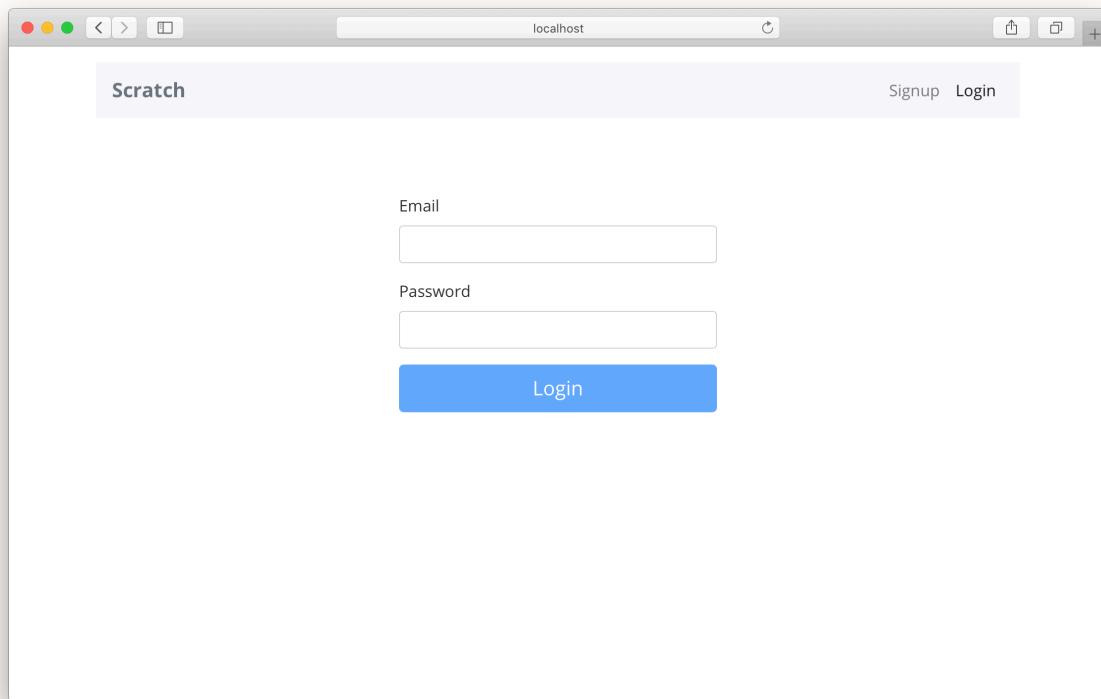
```
<Route path="/login" element={<Login />} />
```



And include our component in the header.

```
import Login from "./containers/Login.tsx";
```

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.

## Login to Amazon Cognito

The login code itself is relatively simple.

◆ CHANGE Simply replace our placeholder handleSubmit method in src/containers/Login.tsx with the following.

```
async function handleSubmit(event: React.FormEvent<HTMLFormElement>) {
    event.preventDefault();

    try {
        await Auth.signIn(email, password);
        alert("Logged in");
    } catch (error) {
        // Prints the full error
        console.error(error);
        if (error instanceof Error) {
            alert(error.message);
        } else {
            alert(String(error));
        }
    }
}
```

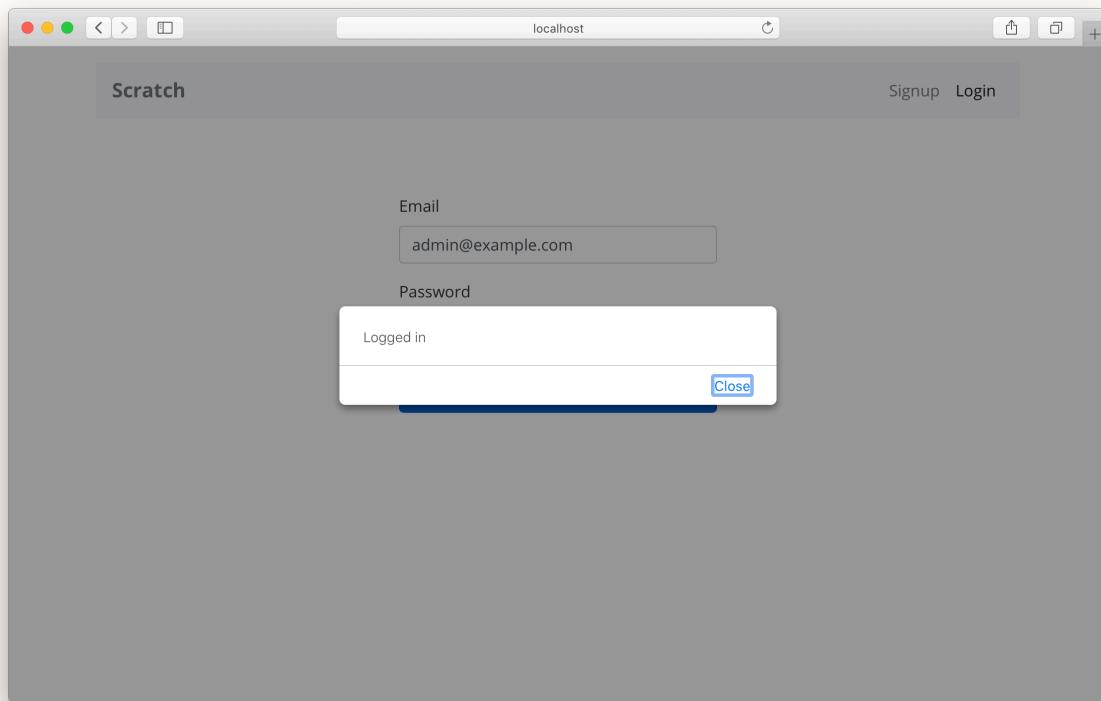
◆ CHANGE And import Auth in the header of src/containers/Login.tsx.

```
import { Auth } from "aws-amplify";
```

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.tsx`.

◆ CHANGE Add the following to the top of our `App` component function.

```
const [isAuthenticated, userHasAuthenticated] = useState(false);
```

◆ CHANGE Then import it.

```
import { useState } from "react";
```

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/lib/` directory in the `packages/frontend/React` directory.

```
$ mkdir src/lib/
```

We'll use this to store all our common code.

◆ CHANGE Add the following file with the content below `src/lib/contextLib.ts`.

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

export interface AppContextType {
  isAuthenticated: boolean;
  userHasAuthenticated: React.Dispatch<React.SetStateAction<boolean>>;
}

export const AppContext = createContext<AppContextType>({
  isAuthenticated: false,
  userHasAuthenticated: useAppContext,
});

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.tsx`.

```
import { AppContext, AppContextType } from "./lib/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.tsx`.

```
<Routes />
```

◆ CHANGE With this.

```
<AppContext.Provider  
  value={{ isAuthenticated, userHasAuthenticated } as AppContextType}  
>  
<Routes />  
</AppContext.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, `src/containers/Login.tsx`.

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

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

◆ **CHANGE** And import it in the header of `src/containers/Login.tsx`.

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

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.tsx`.

```
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.tsx`.

```
<LinkContainer to="/signup">
  <Nav.Link>Signup</Nav.Link>
</LinkContainer>
<LinkContainer to="/login">
  <Nav.Link>Login</Nav.Link>
</LinkContainer>
```

◆ CHANGE And replace it with this:

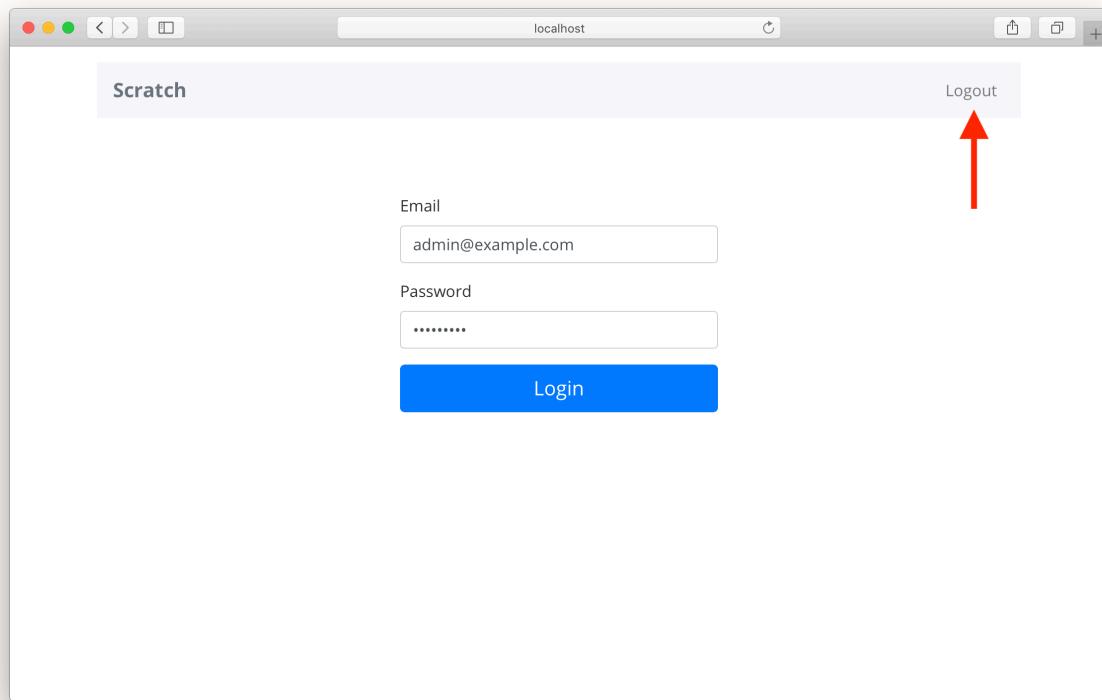
```
{isAuthenticated ? (
  <Nav.Link onClick={handleLogout}>Logout</Nav.Link>
) : (
  <>
    <LinkContainer to="/signup">
      <Nav.Link>Signup</Nav.Link>
    </LinkContainer>
    <LinkContainer to="/login">
      <Nav.Link>Login</Nav.Link>
    </LinkContainer>
  </>
)}
```

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.tsx` 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 [Secure Our Serverless APIs](#) 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.tsx` 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** To load the user session we'll add the following to our `src/App.tsx` 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);  
}
```

◆ **CHANGE** Then include the Auth module by adding the following to the header of `src/App.tsx`.

```
import { Auth } from "aws-amplify";
```

◆ **CHANGE** Let's make sure to include the `useEffect` hook by replacing the React import in the header of `src/App.tsx` with:

```
import { useState, useEffect } from "react";
```

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 every time 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 [authenticated, userHasAuthenticated] = useState(false);  
  
  useEffect(() => {  
    onLoad();  
  }, []);  
  
  ...  
}
```

## 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`.

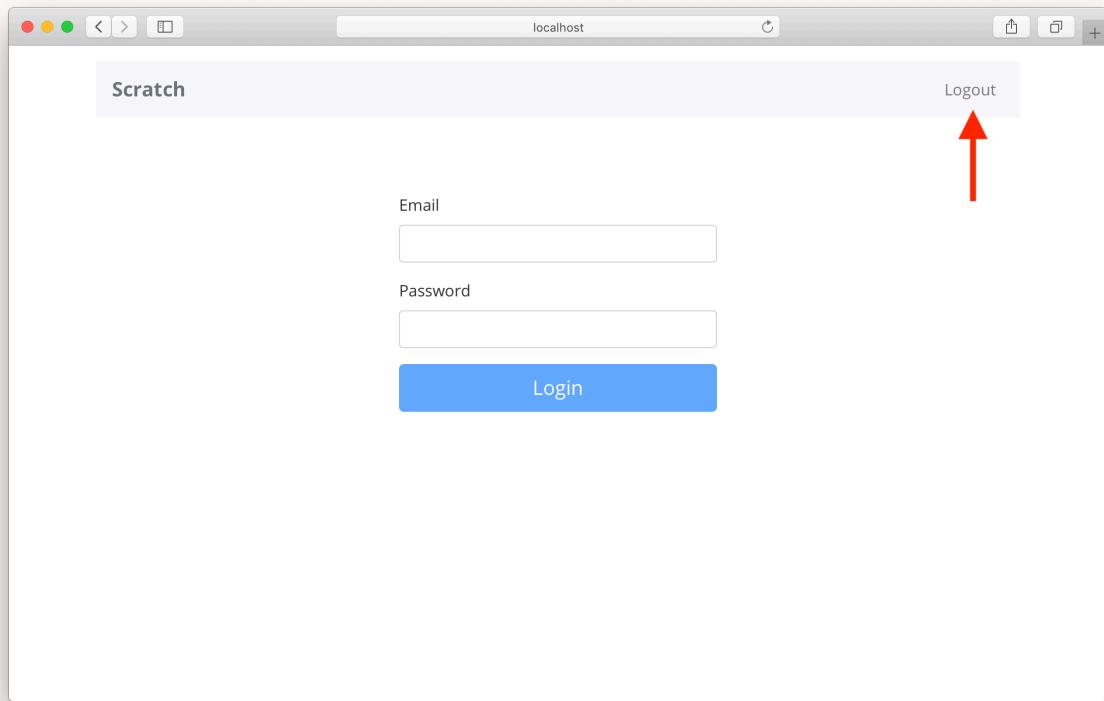
We'll conditionally render our app based on the `isAuthenticating` flag.

◆ CHANGE Replace the `return` statement in `src/App.tsx` with the following.

```
return (  
  !isAuthenticating && (  
    <div className="App container py-3">  
      <Navbar collapseOnSelect bg="light" expand="md" className="mb-3 px-3">  
        <LinkContainer to="/">  
          <Navbar.Brand className="fw-bold text-muted">Scratch</Navbar.Brand>  
        </LinkContainer>  
        <Navbar.Toggle />  
        <Navbar.Collapse className="justify-content-end">  
          <Nav activeKey={window.location.pathname}>  
            {isAuthenticated ? (  
              <Nav.Link onClick={handleLogout}>Logout</Nav.Link>  
            ) : (  
              <>
```

```
<LinkContainer to="/signup">
  <Nav.Link>Signup</Nav.Link>
</LinkContainer>
<LinkContainer to="/login">
  <Nav.Link>Login</Nav.Link>
</LinkContainer>
</>
)
</Nav>
</Navbar.Collapse>
</Navbar>
<AppContext.Provider
  value={{ isAuthenticated, userHasAuthenticated } as AppContextType}>
<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.tsx` 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 `useNavigate` hook that comes with React Router. This will allow us to use the browser's [History API](#).

## Redirect to Home on Login

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

```
const nav = useNavigate();
```

Make sure to add it below the `export default function Login() {` line.

◆ CHANGE Then update the `handleSubmit` method in `src/containers/Login.tsx` to look like this:

```
async function handleSubmit(event: React.FormEvent<HTMLFormElement>) {
  event.preventDefault();

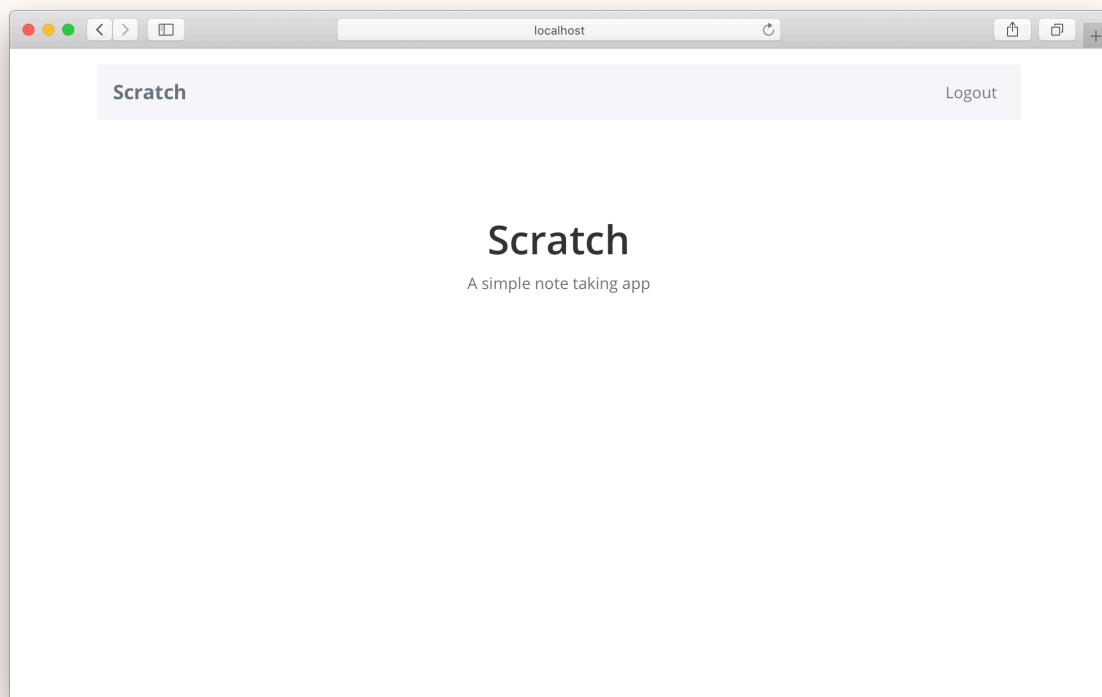
  try {
    await Auth.signIn(email, password);
    userHasAuthenticated(true);
    nav("/");
  } catch (error) {
    if (error instanceof Error) {
      alert(error.message);
    } else {
      alert(String(error));
    }
  }
}
```

```
    }  
  }  
}
```

◆ CHANGE Also, import useNavigate from React Router in the header of src/containers/Login.tsx.

```
import { useNavigate } 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 v6 redirect home after login screenshot

## Redirect to Login After Logout

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

◆ CHANGE Add the useNavigate hook in the beginning of App component in src/App.tsx.

```
const nav = useNavigate();
```

◆ CHANGE Import useNavigate from React Router in the header of `src/App.tsx`.

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

◆ CHANGE Add the following to the bottom of the `handleLogout` function in our `src/App.tsx`.

```
nav("/login");
```

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

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

  userHasAuthenticated(false);

  nav("/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.tsx. 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: React.FormEvent<HTMLFormElement>) {
  event.preventDefault();

  setIsLoading(true);

  try {
    await Auth.signIn(email, password);
    userHasAuthenticated(true);
    nav("/");
  } catch (error) {
    if (error instanceof Error) {
      alert(error.message);
    } else {
      alert(String(error));
    }
    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 the `frontend/` 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.tsx`.

```
import Button from "react-bootstrap/Button";
import { BsArrowRepeat } from "react-icons/bs";
import "./LoaderButton.css";

export default function LoaderButton({
  className = "",
  disabled = false,
  isLoading = false,
  ...props
}) {
  return (
    <Button
      disabled={disabled || isLoading}
      className={`LoaderButton ${className}`}
      {...props}
    >
      {isLoading && <BsArrowRepeat className="spinning" />}
      {props.children}
    </Button>
  );
}
```

This is a really simple component that takes an `isLoading` prop and `disabled` prop. The latter 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.

The `className` prop that we have is to ensure that a CSS class that's set for this component, doesn't override the `LoaderButton` CSS class that we are using internally.

When the `isLoading` flag is on, we show an icon. The icon we include is from the Bootstrap icon set of [React Icons](#).

And let's add a couple of styles to animate our loading icon.

◆ **CHANGE** Add the following to `src/components/LoaderButton.css`.

```
.LoaderButton {  
  margin-top: 12px;  
}  
  
.LoaderButton .spinning {  
  margin-right: 7px;  
  margin-bottom: 1px;  
  animation: spin 1s infinite linear;  
}  
  
@keyframes spin {  
  from {  
    transform: scale(1) rotate(0deg);  
  }  
  to {  
    transform: scale(1) rotate(360deg);  
  }  
}
```

This spins the icon 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.tsx` find the `<Button>` component in the `return` statement.

```
<Button size="lg" type="submit" disabled={!validateForm()}>  
  Login  
</Button>
```

◆ CHANGE And replace it with this.

```
<LoaderButton  
  size="lg"  
  type="submit"  
  isLoading={isLoading}  
  disabled={!validateForm()}  
>  
  Login  
</LoaderButton>
```

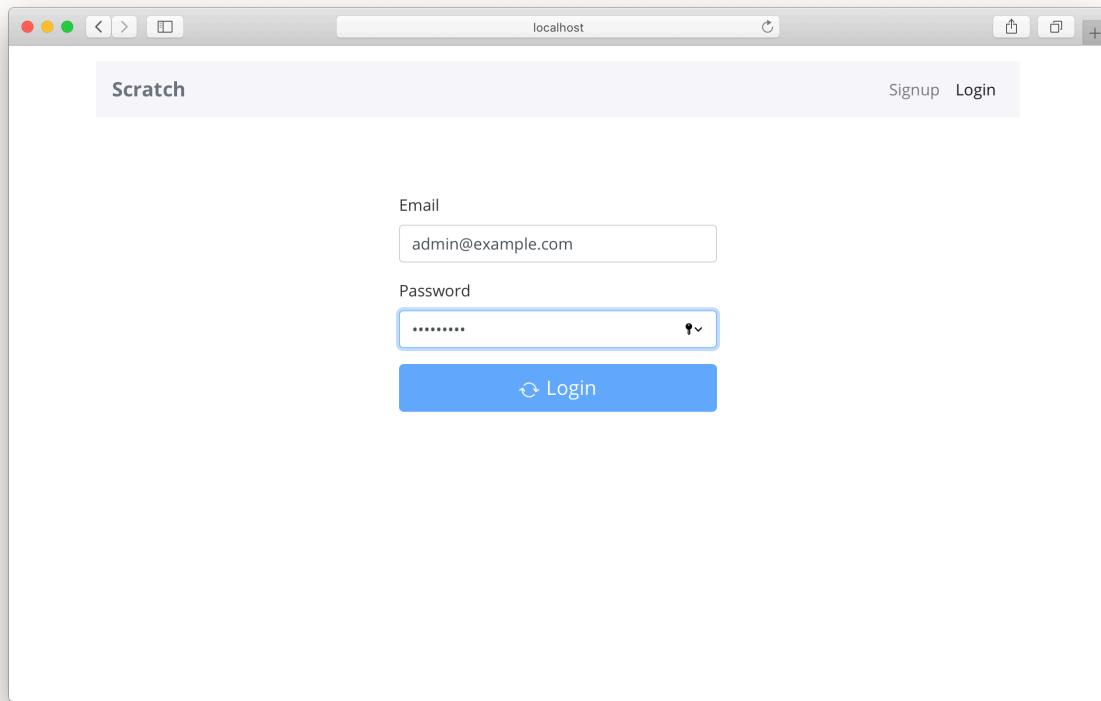
◆ CHANGE Also, let's replace Button import in the header. Remove this.

```
import Button from "react-bootstrap/Button";
```

◆ CHANGE And add the following.

```
import LoaderButton from "../components/LoaderButton.tsx";
```

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, create `src/lib/errorLib.ts` and add the following.

```
export function onError(error: any) {
  let message = String(error);

  if (!(error instanceof Error) && error.message) {
    message = String(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 (`containers/Login.tsx`).

◆ **CHANGE** Replace the `catch` statement in the `handleSubmit` function with:

```
catch (error) {  
  onError(error);  
  setIsLoading(false);  
}
```

◆ **CHANGE** And import the new error lib in the header of `src/containers/Login.tsx`.

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

We'll do something similar in the App component.

◆ **CHANGE** Replace the `catch` statement in the `onLoad` function with:

```
catch (error) {  
  if (error !== "No current user") {  
    onError(error);  
  }  
}
```

◆ **CHANGE** And import the error lib in the header of `src/App.tsx`.

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

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

**Info:** If you would like to add a *Forgot Password* feature 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/lib/hooksLib.ts`.

```
import { useState, ChangeEvent, ChangeEventHandler } from "react";

interface FieldsType {
  [key: string | symbol]: string;
}
```

```
export function useFormFields(
  initialState: FieldsType
): [FieldsType, ChangeEventHandler] {
  const [fields, setValues] = useState(initialState);

  return [
    fields,
    function (event: ChangeEvent<HTMLInputElement>) {
      setValues({
        ...fields,
        [event.target.id]: event.target.value,
      });
      return;
    },
  ];
}
```

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 `Form.Group` element:

```
<Form.Group controlId="email">
  <Form.Label>Email</Form.Label>
  <Form.Control
    autoFocus
    type="email"
    value={email}
    onChange={(e) => setEmail(e.target.value)}>
```

```
    />
  </Form.Group>
```

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

We need to make a couple of changes to the component to use our custom hook.

◆ CHANGE Let's start by importing it in `src/containers/Login.tsx`.

```
import { useFormFields } from "../lib/hooksLib";
```

◆ CHANGE Replace the variable declarations.

```
const [email, setEmail] = useState("");
const [password, setPassword] = useState("");
```

◆ CHANGE With:

```
const [fields, handleFieldChange] = useFormFields({
  email: "",
  password: "",
});
```

◆ CHANGE Replace the `validateForm` function with.

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

◆ CHANGE In the `handleSubmit` function, replace the `Auth.signIn` call with.

```
await Auth.signIn(fields.email, fields.password);
```



Replace our two form fields, starting with the `<Form.Control type="email">`.

```
<Form.Control  
  autoFocus  
  size="lg"  
  type="email"  
  value={fields.email}  
  onChange={handleFieldChange}  
/>
```



And finally the password `<Form.Control type="password">`.

```
<Form.Control  
  size="lg"  
  type="password"  
  value={fields.password}  
  onChange={handleFieldChange}  
/>
```

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



Create a new container at `src/containers/Signup.tsx` with the following.

```
import React, { useState } from "react";
import Form from "react-bootstrap/Form";
import Stack from "react-bootstrap/Stack";
import { useNavigate } from "react-router-dom";
import { useFormFields } from "../lib/hooksLib";
import { useApplicationContext } from "../lib/contextLib";
import LoaderButton from "../components/LoaderButton";
import "./Signup.css";

export default function Signup() {
  const [fields, handleFieldChange] = useFormFields({
    email: "",
    password: "",
    confirmPassword: "",
    confirmationCode: "",
  });
  const nav = useNavigate();
  const { userHasAuthenticated } = useApplicationContext();
  const [isLoading, setIsLoading] = useState(false);
  const [newUser, setNewUser] = useState<null | string>(null);

  function validateForm() {
    return (
      fields.email !== "" &&
      fields.password !== "" &&
      fields.confirmPassword === fields.password &&
      fields.confirmationCode === "4242"
    );
  }

  function handleSubmit(event) {
    event.preventDefault();
    if (!validateForm()) {
      return;
    }
    setIsLoading(true);
    const newUser = {
      email: fields.email,
      password: fields.password,
      confirmationCode: fields.confirmationCode,
    };
    fetch("http://localhost:5001/users", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(newUser),
    })
      .then((res) => res.json())
      .then((data) => {
        if (data.error) {
          alert(data.error);
        } else {
          alert(`Welcome ${data.user.name}!`);
          userHasAuthenticated(true);
          setNewUser(data.user);
          nav("/");
        }
      })
      .catch((err) => {
        console.error(err);
      });
  }
}
```

```
    fields.email.length > 0 &&
    fields.password.length > 0 &&
    fields.password === fields.confirmPassword
  );
}

function validateConfirmationForm() {
  return fields.confirmationCode.length > 0;
}

async function handleSubmit(event: React.FormEvent<HTMLFormElement>) {
  event.preventDefault();
  setIsLoading(true);
  setNewUser("test");
  setIsLoading(false);
}

async function handleConfirmationSubmit(
  event: React.FormEvent<HTMLFormElement>
) {
  event.preventDefault();
  setIsLoading(true);
}

function renderConfirmationForm() {
  return (
    <Form onSubmit={handleConfirmationSubmit}>
      <Stack gap={3}>
        <Form.Group controlId="confirmationCode">
          <Form.Label>Confirmation Code</Form.Label>
          <Form.Control
            size="lg"
            autoFocus
            type="tel"
            onChange={handleFieldChange}
            value={fields.confirmationCode}
          />
          <Form.Text muted>Please check your email for the code.</Form.Text>
      </Stack>
    </Form>
  );
}
```

```
</Form.Group>
<LoaderButton
  size="lg"
  type="submit"
  variant="success"
  isLoading={isLoading}
  disabled={!validateConfirmationForm()}>
  <Stack gap={3}>
    <Form>
      <Form.Group controlId="email">
        <Form.Label>Email</Form.Label>
        <Form.Control
          size="lg"
          autoFocus
          type="email"
          value={fields.email}
          onChange={handleFieldChange}>
        />
      </Form.Group>
      <Form.Group controlId="password">
        <Form.Label>Password</Form.Label>
        <Form.Control
          size="lg"
          type="password"
          value={fields.password}
          onChange={handleFieldChange}>
        />
      </Form.Group>
    </Form>
  </Stack>
  Verify
</LoaderButton>
</Form>
);
}

function renderForm() {
  return (
    <Form onSubmit={handleSubmit}>
      <Stack gap={3}>
        <Form.Group controlId="email">
          <Form.Label>Email</Form.Label>
          <Form.Control
            size="lg"
            autoFocus
            type="email"
            value={fields.email}
            onChange={handleFieldChange}>
          />
        </Form.Group>
        <Form.Group controlId="password">
          <Form.Label>Password</Form.Label>
          <Form.Control
            size="lg"
            type="password"
            value={fields.password}
            onChange={handleFieldChange}>
          />
        </Form.Group>
      </Stack>
    </Form>
  );
}
```

```
<Form.Group controlId="confirmPassword">
  <Form.Label>Confirm Password</Form.Label>
  <Form.Control
    size="lg"
    type="password"
    onChange={handleFieldChange}
    value={fields.confirmPassword}
  />
</Form.Group>
<LoaderButton
  size="lg"
  type="submit"
  variant="success"
  isLoading={isLoading}
  disabled={!validateForm()}
>
  Signup
</LoaderButton>
</Stack>
</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.

```
{
  newUser === null ? renderForm() : renderConfirmationForm();
}
```

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.

```
<Form.Control autoFocus type="email" ... />
```

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.

```
const [fields, handleFieldChange] = useFormFields({  
  email: "",  
  password: "",  
  confirmPassword: "",  
  confirmationCode: "",  
});
```



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;  
  }  
}
```

## Add the Route



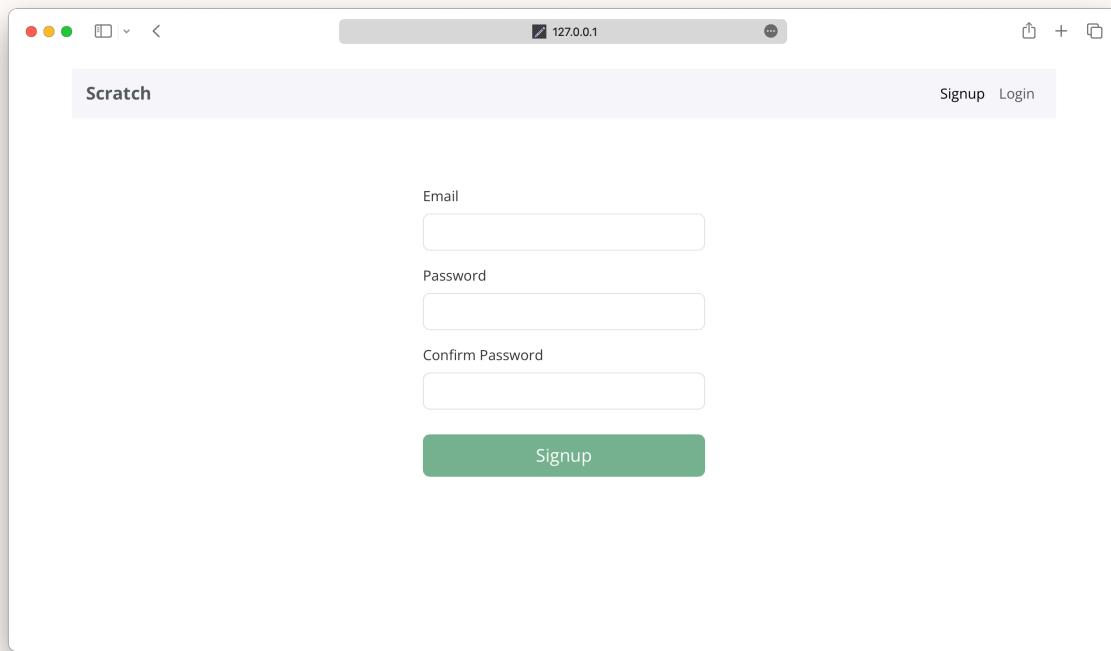
Finally, add our container as a route in `src/Routes.tsx` below our login route.

```
<Route path="/signup" element={<Signup />} />
```

◆ CHANGE And include our component in the header.

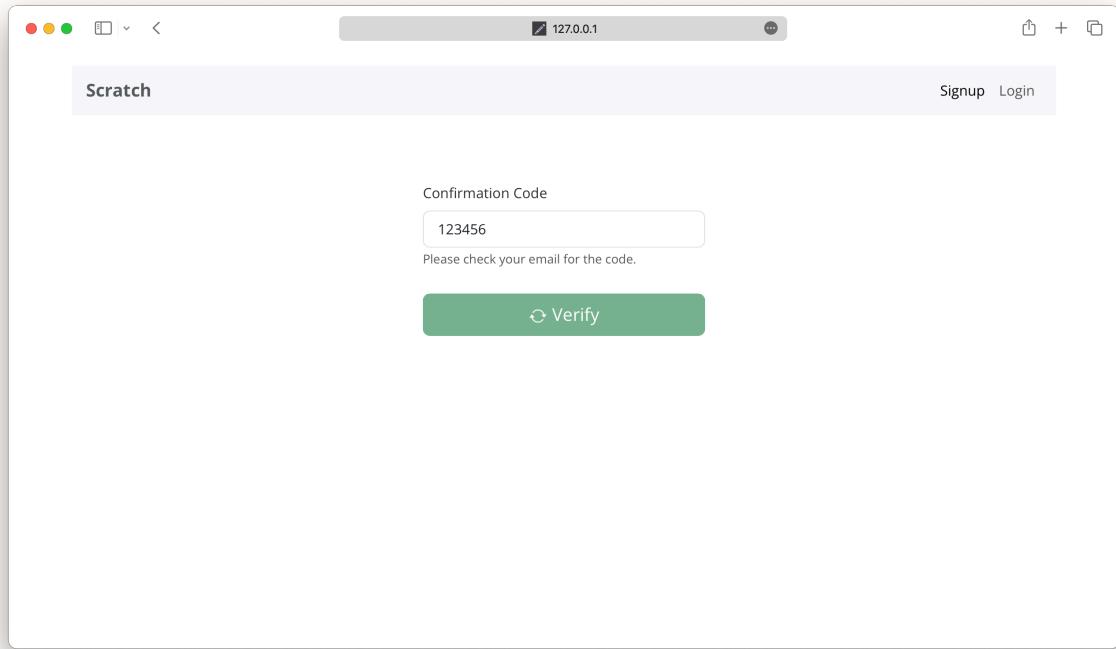
```
import Signup from "./containers/Signup.tsx";
```

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 confirmation password.



Signup page added screenshot

Then, after hitting submit, you'll get the confirmation code form. This'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.tsx` with the following.

```
async function handleSubmit(event: React.FormEvent<HTMLFormElement>) {
  event.preventDefault();
  setIsLoading(true);
  try {
    const newUser = await Auth.signUp({
      username: fields.email,
      password: fields.password,
    });
    setIsLoading(false);
    setNewUser(newUser);
  } catch (e) {
    onError(e);
    setIsLoading(false);
  }
}

async function handleConfirmationSubmit(
  event: React.FormEvent<HTMLFormElement>
) {
  event.preventDefault();
  setIsLoading(true);
  try {
    await Auth.confirmSignUp(fields.email, fields.confirmationCode);
    await Auth.signIn(fields.email, fields.password);
    userHasAuthenticated(true);
  }
}
```

```
    nav("/");
} catch (e) {
  onError(e);
  setIsLoading(false);
}
}
```

◆ CHANGE Also, include the Amplify Auth, onError, and ISignUpResult type in our header.

```
import { Auth } from "aws-amplify";
import { onError } from "../lib/errorLib";
import { ISignUpResult } from "amazon-cognito-identity-js";
```

Let's use the right type for the new user object.

◆ CHANGE Replace the const [newUser, setNewUser] line with.

```
const [newUser, setNewUser] = useState<null | ISignUpResult>(null);
```

Let's install the npm package.

◆ CHANGE Run the following in the packages/frontend/ directory.

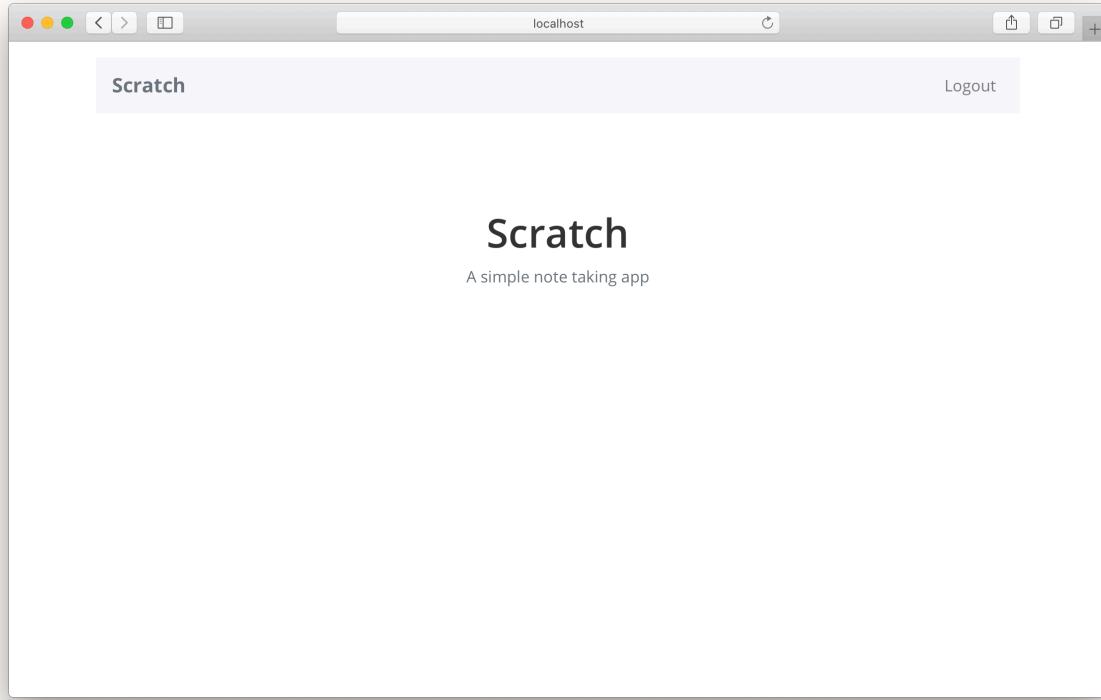
```
$ npm install amazon-cognito-identity-js
```

The flow here is pretty simple:

1. In handleSubmit we make a call to signup a user using Auth.signUp(). 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 with Auth.confirmSignUp().
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. By calling Auth.signIn().
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 <COGNITO_REGION> \  
  --user-pool-id <USER_POOL_ID> \  
  --username <YOUR_USER_EMAIL>
```

Just be sure to use your Cognito USER\_POOL\_ID and the *email address* you used to create the account.

**Info:** 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](#)

## **Building a React app**

# 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.tsx` and add the following.

```
import React, {useRef, useState} from "react";
import Form from "react-bootstrap/Form";
import Stack from "react-bootstrap/Stack";
import {useNavigate} from "react-router-dom";
import LoaderButton from "../components/LoaderButton";
import config from "../config";
import "./NewNote.css";

export default function NewNote() {
  const file = useRef<null | File>(null);
  const nav = useNavigate();
  const [content, setContent] = useState("");
  const [isLoading, setIsLoading] = useState(false);

  function validateForm() {
    return content.length > 0;
  }

  function handleFileChange(event: React.ChangeEvent<HTMLInputElement>) {
    if (event.currentTarget.files === null) return;
    file.current = event.currentTarget.files[0];
  }
}
```

```
}

async function handleSubmit(event: React.FormEvent<HTMLFormElement>) {
  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}>
      <Form.Group controlId="content">
        <Form.Control
          value={content}
          as="textarea"
          onChange={(e) => setContent(e.target.value)}
        />
      </Form.Group>
      <Form.Group className="mt-2" controlId="file">
        <Form.Label>Attachment</Form.Label>
        <Form.Control onChange={handleFileChange} type="file" />
      </Form.Group>
      <Stack>
        <LoaderButton
          size="lg"
          type="submit"
          variant="primary"
          isLoading={isLoading}
          disabled={!validateForm()}\>
```

```
>
  Create
  </LoaderButton>
</Stack>
</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.ts` below the `const config = {` line.

```
// Frontend config
MAX_ATTACHMENT_SIZE: 5000000,
```

◆ **CHANGE** Let's also add the styles for our form in `src/containers/NewNote.css`.

```
.NewNote form textarea {
  height: 300px;
  font-size: 1.5rem;
}
```

## Add the Route

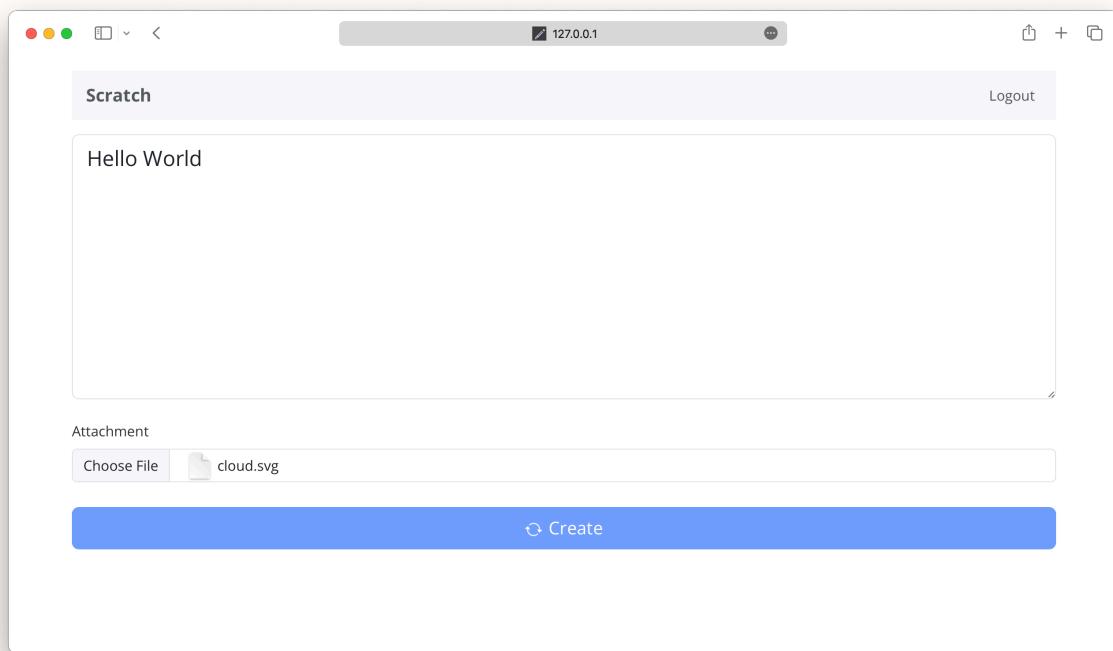
◆ **CHANGE** Finally, add our container as a route in `src/Routes.tsx` below our `signup` route.

```
<Route path="/notes/new" element={<NewNote />} />
```

◆ CHANGE And include our component in the header.

```
import NewNote from "./containers/NewNote.tsx";
```

Now if we switch to our browser and navigate /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.

## Define the type for a note

Let's start by creating a type definition for the note object. Create a new directory for our types.

◆ CHANGE Run the following **in the packages/ frontend/ directory**.

```
$ mkdir src/types
```

◆ CHANGE Add a new file `src/types/note.ts` with the following.

```
export interface NoteType {
  noteId?: string;
  content: string;
  createdAt?: string;
  attachment?: string;
  attachmentURL?: string;
}
```

## Call the Create API

◆ CHANGE Next, we'll replace our `handleSubmit` function in `src/containers/NewNote.tsx` with.

```
function createNote(note: NoteType) {
```

```
return API.post("notes", "/notes", {
  body: note,
});
}

async function handleSubmit(event: React.FormEvent<HTMLFormElement>) {
  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 });
    nav("/");
  } catch (e) {
    onError(e);
    setIsLoading(false);
  }
}
```

◆ **CHANGE** And include the API module by adding the following to the header of `src/containers/NewNote.tsx`.

```
import { API } from "aws-amplify";
import { NoteType } from "../types/note";
import { onError } from "../lib/errorLib";
```

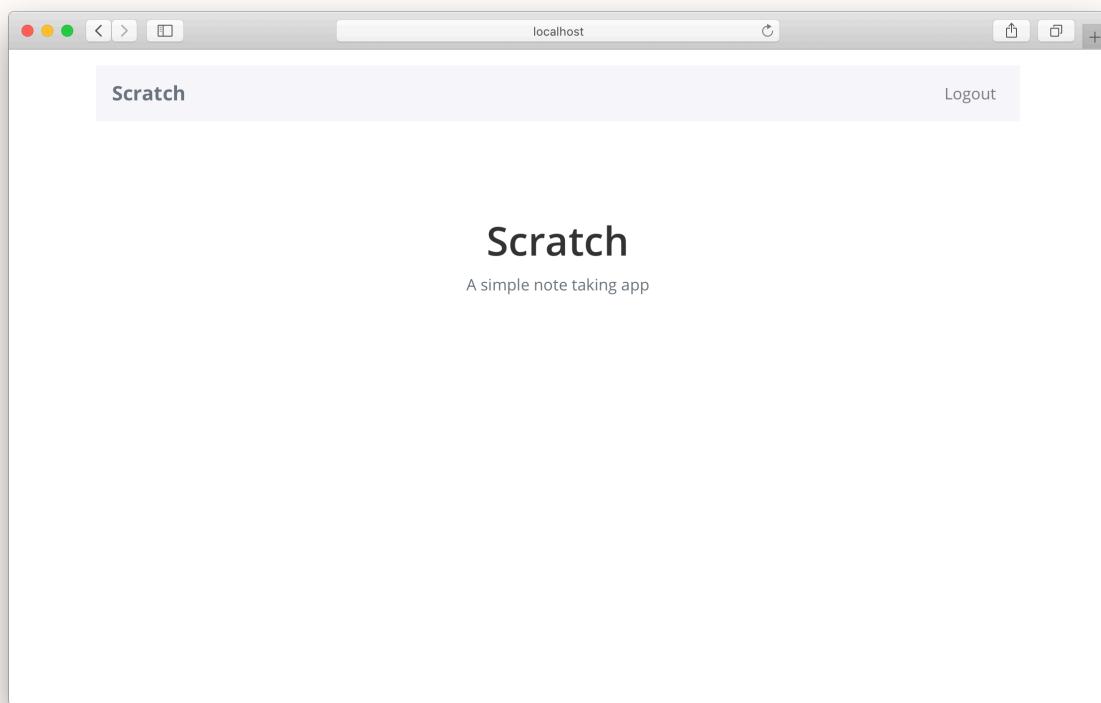
We are doing a couple of things with these functions.

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/lib/awsLib.ts` and add the following:

```
import { Storage } from "aws-amplify";

export async function s3Upload(file: 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.tsx` with the following.

```
async function handleSubmit(event: React.FormEvent<HTMLFormElement>) {
  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)
    : undefined;

  await createNote({ content, attachment });
  nav("/");
} catch (e) {
```

```
    onError(e);
    setIsLoading(false);
}
}
```

◆ **CHANGE** And make sure to include `s3Upload` by adding the following to the header of `src/containers/NewNote.tsx`.

```
import { s3Upload } from "../lib/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.tsx` with the following.

```
import { useState } from "react";
import ListGroup from "react-bootstrap/ListGroup";
import { useContext } from "../lib/contextLib";
import "./Home.css";

export default function Home() {
  const [notes, setNotes] = useState([]);
  const { isAuthenticated } = useContext();
  const [isLoading, setIsLoading] = useState(true);

  function renderNotesList(notes: { [key: string | symbol]: any }) {
    return null;
  }

  function renderLander() {
    return (
      <div className="lander">
        <h1>Scratch</h1>
        <p className="text-muted">A simple note taking app</p>
      </div>
    );
  }
}
```

```
function renderNotes() {  
  return (  
    <div className="notes">  
      <h2 className="pb-3 mt-4 mb-3 border-bottom">Your Notes</h2>  
      <ListGroup>{!isLoading && renderNotesList(notes)}</ListGroup>  
    </div>  
  );  
}  
  
return (  
  <div className="Home">  
    {isAuthenticated ? renderNotes() : renderLander()}  
  </div>  
);  
}
```

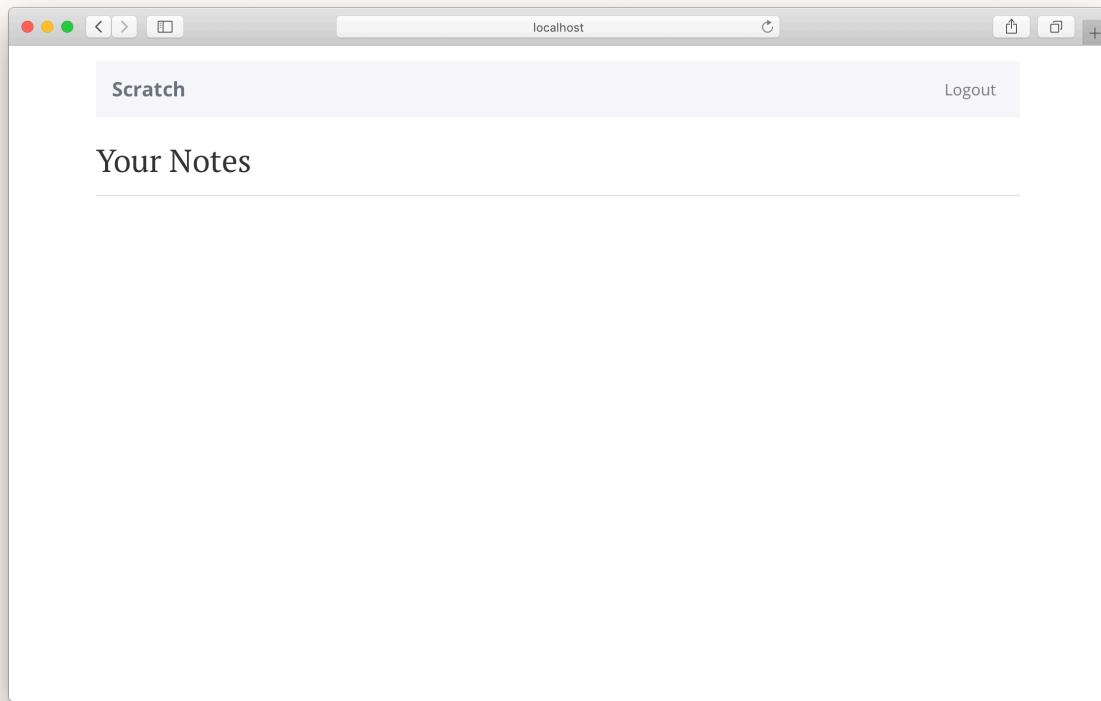
We are doing a few things of note here:

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

```
{  
  isAuthenticated ? renderNotes() : renderLander();  
}
```

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.
4. We're using the [Bootstrap utility classes](#) `pb-3` (padding bottom), `mt-4` (margin top), `mb-3` (margin bottom), and `border-bottom` to style the *Your Notes* header.

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 Replace the const [notes, setNotes] state declaration in `src/containers/Home.tsx` with.

```
const [notes, setNotes] = useState<Array<NoteType>>([]);
```

◆ CHANGE Add the following right below the state variable declarations at the top of the **Home** function.

```
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 Add useEffect into the import from React

```
import { useState, useEffect } from "react";
```

◆ CHANGE And import the other dependencies.

```
import { API } from "aws-amplify";  
import { NoteType } from "../types/note";  
import { onError } from "../lib/errorLib";
```

Now let's render the results.

## Render the List

◆ CHANGE Replace our renderNotesList placeholder method with the following.

```
function formatDate(str: undefined | string) {  
  return !str ? "" : new Date(str).toLocaleString();  
}  
  
function renderNotesList(notes: NoteType[]) {  
  return (  
    <>
```

```
<LinkContainer to="/notes/new">
  <ListGroup.Item action className="py-3 text-nowrap text-truncate">
    <BsPencilSquare size={17} />
    <span className="ms-2 fw-bold">Create a new note</span>
  </ListGroup.Item>
</LinkContainer>
{notes.map(({ noteId, content, createdAt }) => (
  <LinkContainer key={noteId} to={`/notes/${noteId}`}>
    <ListGroup.Item action className="text-nowrap text-truncate">
      <span className="fw-bold">{content.trim().split("\n")[0]}</span>
      <br />
      <span className="text-muted">
        Created: {formatDate(createdAt)}
      </span>
    </ListGroup.Item>
  </LinkContainer>
))}>
</>
);
}
```

◆ CHANGE And include the LinkContainer and BsPencilSquare icon at the top of src/containers/Home.tsx.

```
import { BsPencilSquare } from "react-icons/bs";
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). And it links to [the create note page that we previously created](#).

```
<LinkContainer to="/notes/new">
  <ListGroup.Item action className="py-3 text-nowrap text-truncate">
    <BsPencilSquare size={17} />
    <span className="ms-2 fw-bold">Create a new note</span>
  </ListGroup.Item>
</LinkContainer>
```

2. In the button we use a BsPencilSquare icon from the [React Icons Bootstrap icon set](#).

3. We then render a list of all the notes.

```
notes.map(({ noteId, content, createdAt }) => (...)
```

4. The first line of each note's content is set as the `ListGroup.Item` header.

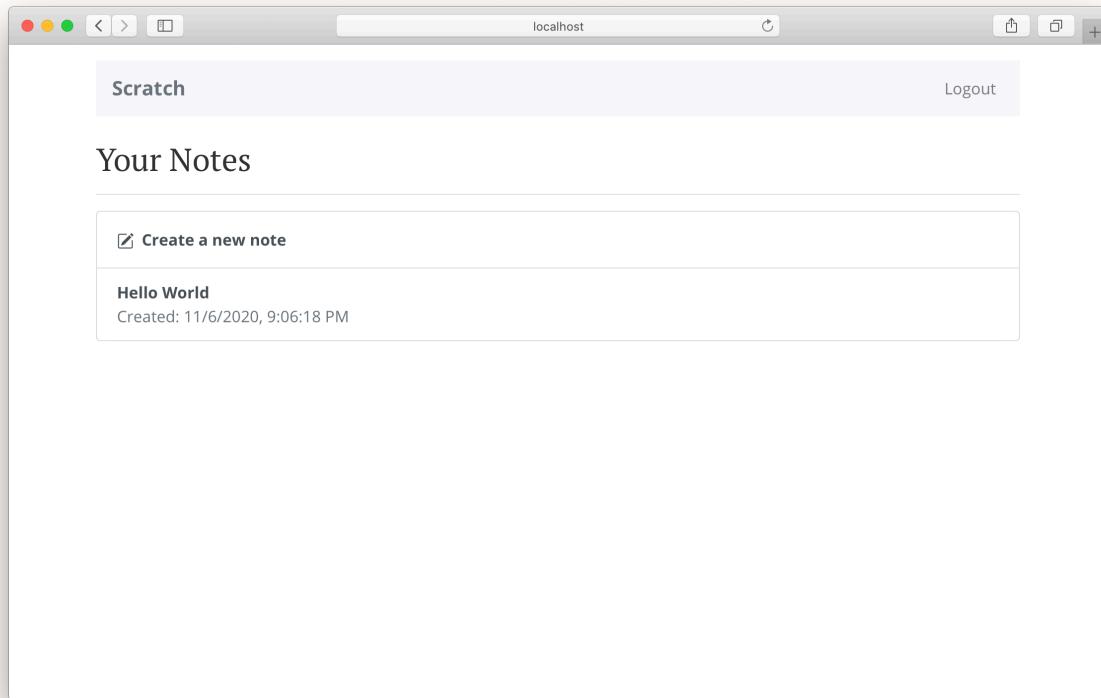
```
note.content.trim().split("\n")[0];
```

5. And we convert the date the note was created to a more friendly format.

```
!str ? "" : new Date(str).toLocaleString()
```

6. The `LinkContainer` component directs our app to each of the items.

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.tsx` **below** our `/notes/new` route.

```
<Route path="/notes/:id" element={<Notes />} />
```

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.tsx";
```

Of course this component doesn't exist yet and we are going to create it now.

## Add the Container

◆ CHANGE Create a new file `src/containers/Notes.tsx` and add the following.

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

export default function Notes() {
  const file = useRef<null | File>(null)
  const { id } = useParams();
  const nav = useNavigate();
  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);
      }
    }

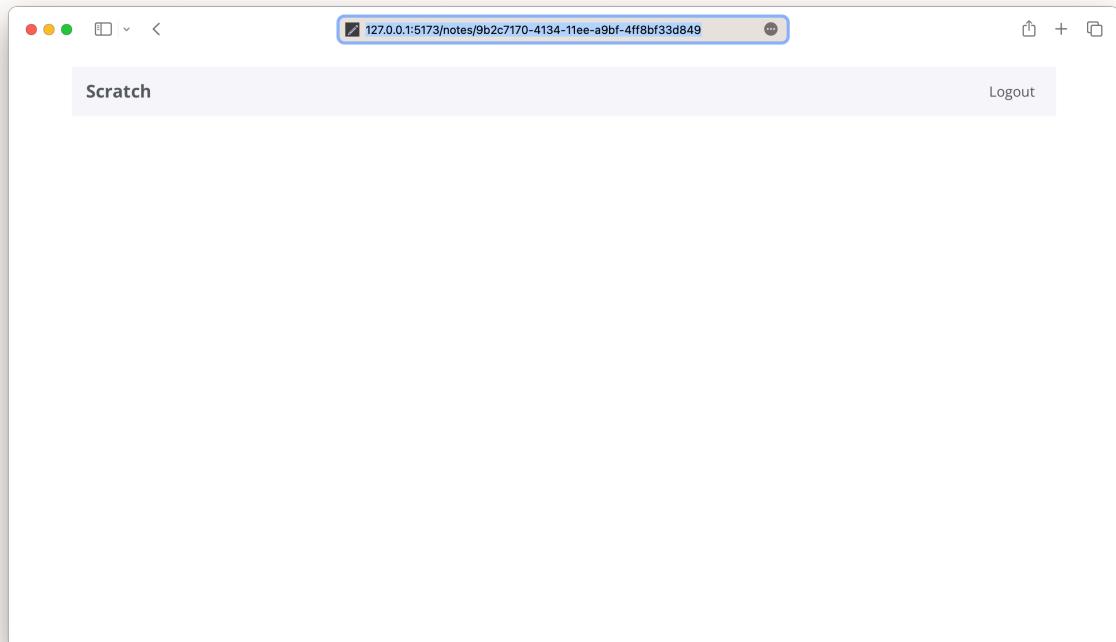
    onLoad();
  }, [id]);
}

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.tsx` with the following.

```
function validateForm() {
  return content.length > 0;
}

function formatFilename(str: string) {
  return str.replace(/\w+-/, "");
}

function handleFileChange(event: React.ChangeEvent<HTMLInputElement>) {
  if (event.currentTarget.files === null) return;
  file.current = event.currentTarget.files[0];
}

async function handleSubmit(event: React.FormEvent<HTMLFormElement>) {
  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: React.FormEvent<HTMLFormElement>) {
    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}>
                <Stack gap={3}>
                    <Form.Group controlId="content">
                        <Form.Control
                            size="lg"
                            as="textarea"
                            value={content}
                            onChange={(e) => setContent(e.target.value)}
                        />
                    </Form.Group>
                    <Form.Group className="mt-2" controlId="file">
                        <Form.Label>Attachment</Form.Label>
                        {note.attachment && (
                            <p>
                                <a
                                    target="_blank"

```

```
        rel="noopener noreferrer"
        href={note.attachmentURL}
      >
    {formatFilename(note.attachment)}
  </a>
</p>
)}
<Form.Control onChange={handleFileChange} type="file" />
</Form.Group>
<Stack gap={1}>
  <LoaderButton
    size="lg"
    type="submit"
    isLoading={isLoading}
    disabled={!validateForm()}>
    Save
  </LoaderButton>
  <LoaderButton
    size="lg"
    variant="danger"
    onClick={handleDelete}
    isLoading={isDeleting}>
    Delete
  </LoaderButton>
</Stack>
</Stack>
</Form>
)
);
</div>
);
```

◆ **CHANGE** To complete this, let's add `isLoading` and `isDeleting` below the state and ref declarations at the top of our Notes component function.

```
const [isLoading, setIsLoading] = useState(false);
const [isDeleting, setIsDeleting] = useState(false);
```

◆ CHANGE Replace the const [note, setNote] definition with the right type.

```
const [note, setNote] = useState<null | NoteType>(null);
```

◆ CHANGE Let's also add some styles by adding the following to src/containers/Notes.css.

```
.Notes form textarea {  
  height: 300px;  
  font-size: 1.5rem;  
}
```

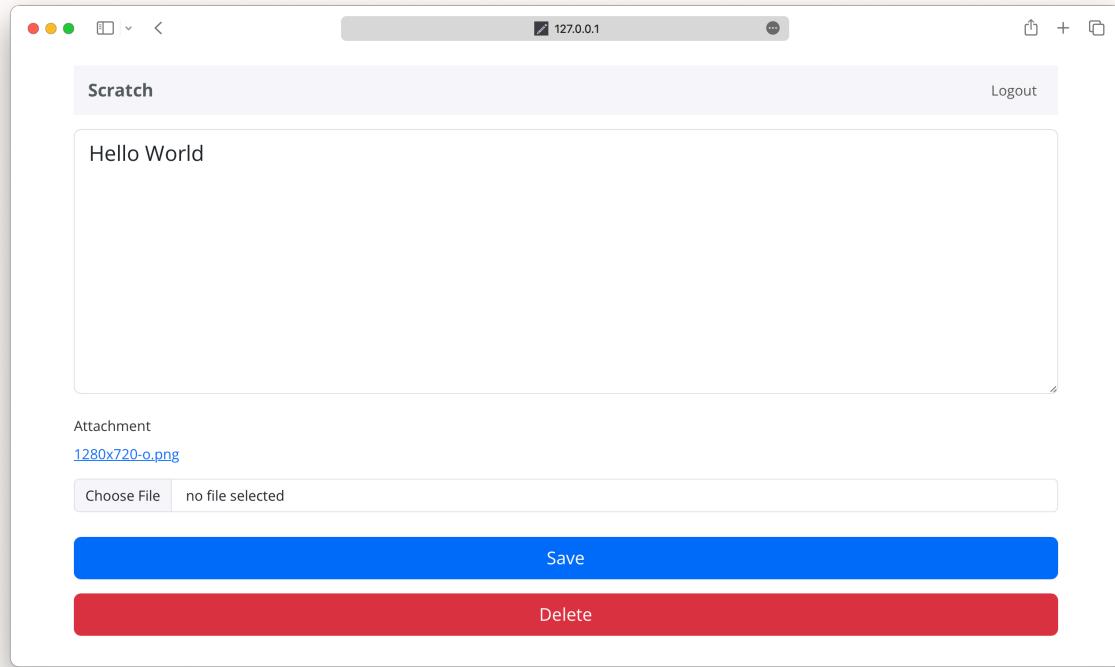
◆ CHANGE And finally, let's add the imports.

```
import config from "../config";  
import Form from "react-bootstrap/Form";  
import { NoteType } from "../types/note";  
import Stack from "react-bootstrap/Stack";  
import LoaderButton from "../components/LoaderButton";  
import "./Notes.css";
```

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.

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.tsx` with the following.

```
function saveNote(note: NoteType) {
  return API.put("notes", `/notes/${id}`, {
    body: note,
  });
}

async function handleSubmit(event: React.FormEvent<HTMLFormElement>) {
  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);
  } else if (note && note.attachment) {
```

```
attachment = note.attachment;
}

await saveNote({
  content: content,
  attachment: attachment,
});
nav("/");
} catch (e) {
  onError(e);
  setIsLoading(false);
}
}
```

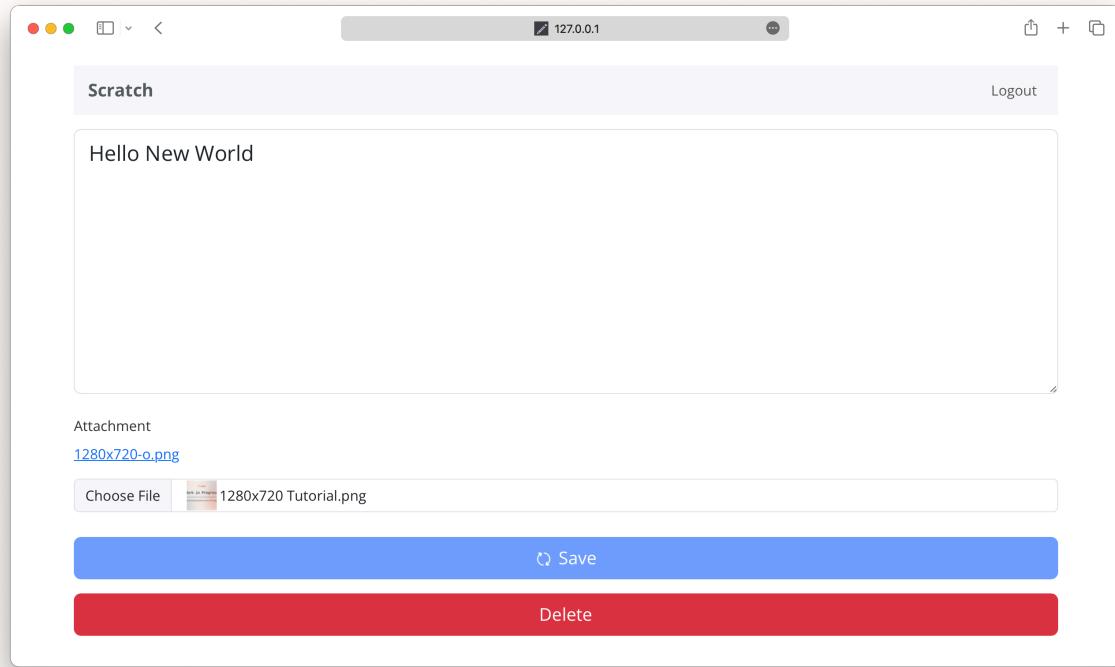
◆ CHANGE And include our s3Upload helper method in the header:

```
import { s3Upload } from "../lib/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.tsx`.

```
function deleteNote() {
  return API.del("notes", `/notes/${id}`, {});
}

async function handleDelete(event: React.FormEvent<HTMLModElement>) {
  event.preventDefault();

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

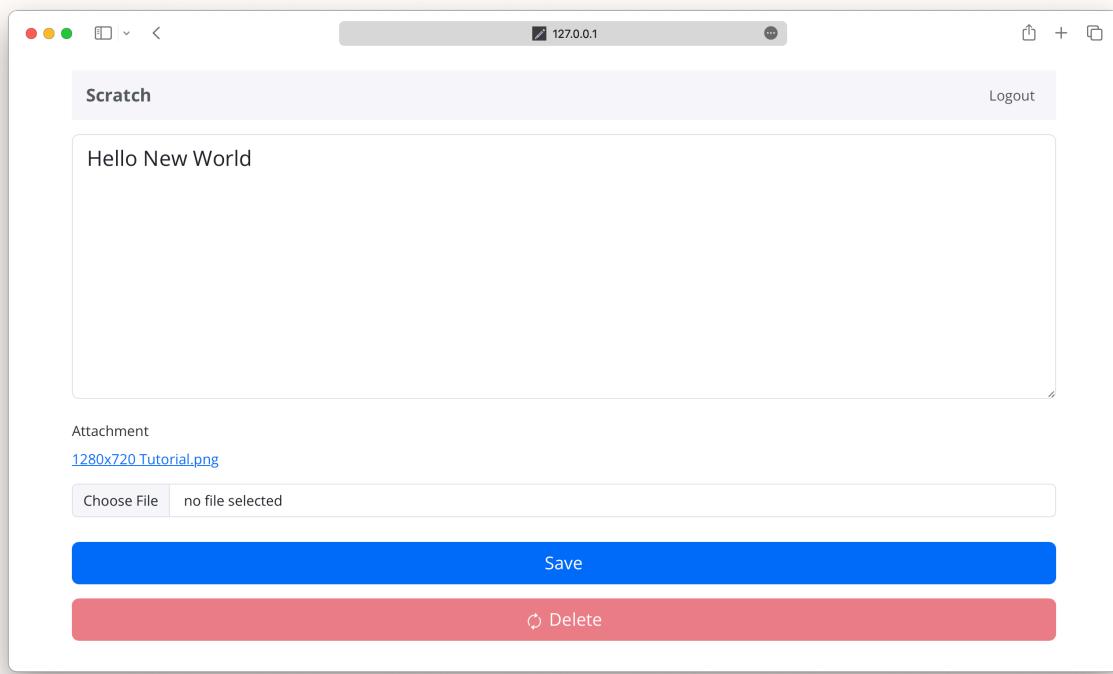
  if (!confirmed) {
    return;
  }

  setIsDeleting(true);

  try {
    await deleteNote();
    nav("/");
  } 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/types/billing.ts` and add the following to define a type for our billing API.

```
export interface BillingType {  
  storage: string;  
  source?: string;  
}
```

◆ **CHANGE** Create a new file in `src/containers/Settings.tsx` and add the following.

```
import { useState } from "react";  
import config from "../config";  
import { API } from "aws-amplify";  
import { onError } from "../lib/errorLib";  
import { useNavigate } from "react-router-dom";  
import { BillingType } from "../types/billing";  
  
export default function Settings() {  
  const nav = useNavigate();  
  const [isLoading, setIsLoading] = useState(false);  
  
  function billUser(details: BillingType) {
```

```
    return API.post("notes", "/billing", {
      body: details,
    });
}

return <div className="Settings"></div>;
}
```

◆ CHANGE Next, add the following below the `/signup` route in our `<Routes>` block in `src/Routes.tsx`.

```
<Route path="/settings" element={<Settings />} />
```

◆ CHANGE And import this component in the header of `src/Routes.js`.

```
import Settings from "./containers/Settings.tsx";
```

Next add a link to our settings page in the navbar.

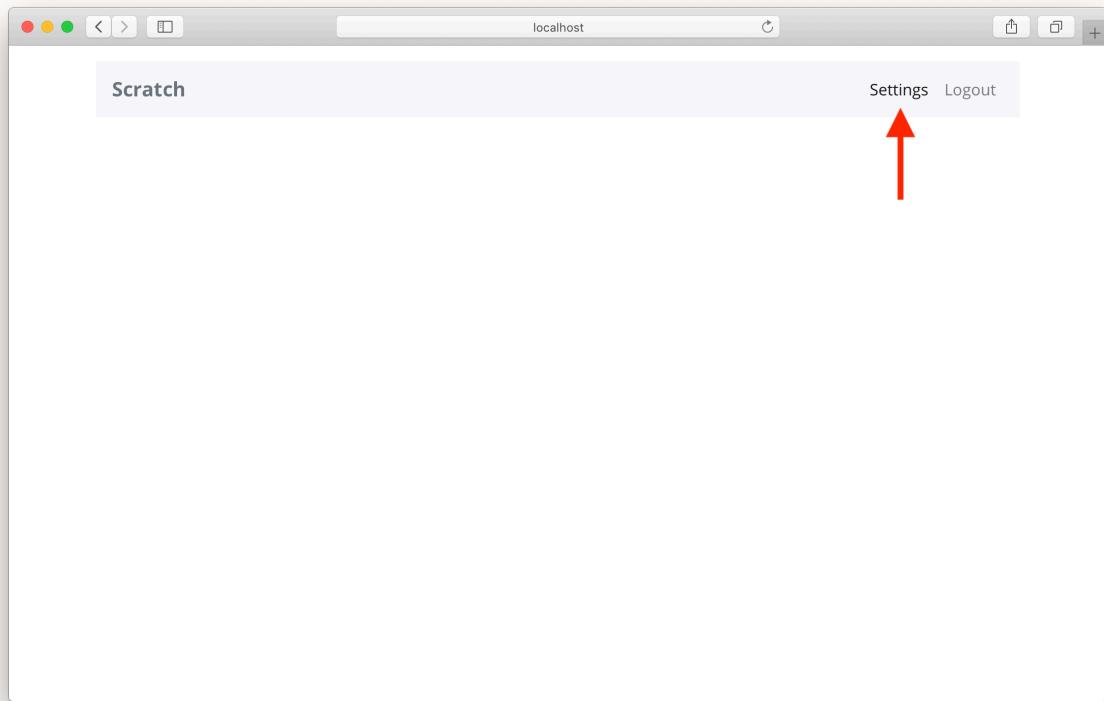
◆ CHANGE Replace the following line in the `return` statement `src/App.tsx`.

```
<Nav.Link onClick={handleLogout}>Logout</Nav.Link>
```

◆ CHANGE With.

```
<>
<LinkContainer to="/settings">
  <Nav.Link>Settings</Nav.Link>
</LinkContainer>
<Nav.Link onClick={handleLogout}>Logout</Nav.Link>
</>
```

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 below the `const config = {` line in your `src/config.ts`.

```
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 add the `Stripe.js` packages

◆ **CHANGE** Run the following **in the packages/ frontend/ directory**.

```
$ npm install @stripe/stripe-js


```

And load the Stripe config in our settings page.

◆ **CHANGE** Add the following **below the imports** in `src/containers/Settings.tsx`.

```
import { loadStripe } from "@stripe/stripe-js";

const stripePromise = loadStripe(config.STRIPE_KEY);


```

This loads the Stripe object from `Stripe.js` with the Stripe key when our settings page loads. We'll be using this in the coming chapters.

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 Run the following **in the packages/frontend/ directory**.

```
$ npm install @stripe/react-stripe-js
```

Next let's create our billing form component.

◆ CHANGE Add the following to a new file in `src/components/BillingForm.tsx`.

```
import React, { useState } from "react";
import Form from "react-bootstrap/Form";
import Stack from "react-bootstrap/Stack";
import { useFormFields } from "../lib/hooksLib";
import { Token, StripeError } from "@stripe/stripe-js";
import LoaderButton from "../components/LoaderButton";
import { CardElement, useStripe, useElements } from "@stripe/react-stripe-js";
import "./BillingForm.css";

export interface BillingFormType {
  isLoading: boolean;
  onSubmit: (
    storage: string,
    info: { token?: Token; error?: StripeError }
  ) => Promise<void>;
}

export function BillingForm({ isLoading, onSubmit }: BillingFormType) {
  const stripe = useStripe();
  const elements = useElements();
```

```
const [fields, handleFieldChange] = useFormFields({
  name: "",
  storage: "",
});
const [isProcessing, setIsProcessing] = useState(false);
const [isCardComplete, setIsCardComplete] = useState(false);

isLoading = isProcessing || isLoading;

function validateForm() {
  return (
    stripe &&
    elements &&
    fields.name !== "" &&
    fields.storage !== "" &&
    isCardComplete
  );
}

async function handleSubmitClick(event: React.FormEvent<HTMLFormElement>) {
  event.preventDefault();

  if (!stripe || !elements) {
    // Stripe.js has not loaded yet. Make sure to disable
    // form submission until Stripe.js has loaded.
    return;
  }

  if (!elements.getElement(CardElement)) {
    return;
  }

  setIsProcessing(true);

  const cardElement = elements.getElement(CardElement);

  if (!cardElement) {
    return;
  }
}
```

```
}

const { token, error } = await stripe.createToken(cardElement);

setIsProcessing(false);

onSubmit(fields.storage, { token, error });

}

return (
  <Form className="BillingForm" onSubmit={handleSubmitClick}>
    <Form.Group controlId="storage">
      <Form.Label>Storage</Form.Label>
      <Form.Control
        min="0"
        size="lg"
        type="number"
        value={fields.storage}
        onChange={handleFieldChange}
        placeholder="Number of notes to store"
      />
    </Form.Group>
    <hr />
    <Stack gap={3}>
      <Form.Group controlId="name">
        <Form.Label>Cardholder's name</Form.Label>
        <Form.Control
          size="lg"
          type="text"
          value={fields.name}
          onChange={handleFieldChange}
          placeholder="Name on the card"
        />
      </Form.Group>
      <div>
        <Form.Label>Credit Card Info</Form.Label>
        <CardElement
          className="card-field"
        </CardElement>
      </div>
    </Stack>
  </Form>
)
```

```
        onChange={(e) => setIsCardComplete(e.complete)}
        options={{
          style: {
            base: {
              fontSize: "16px",
              fontWeight: "400",
              color: "#495057",
              fontFamily: "'Open Sans', sans-serif",
            },
          },
        }}
      />
    </div>
    <LoaderButton
      size="lg"
      type="submit"
      isLoading={isLoading}
      disabled={!validateForm()}>
      Purchase
    </LoaderButton>
  </Stack>
</Form>
);
}
```

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

- To begin with we are getting a reference to the Stripe object by calling `useStripe`.
- 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 `CardElement`. It uses this to generate a token for the specific call. 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 SDK 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 {  
  line-height: 1.5;  
  padding: 0.65rem 0.75rem;  
  background-color: var(--bs-body-bg);  
  border: 1px solid var(--bs-border-color);  
  border-radius: var(--bs-border-radius-lg);  
  transition: border-color 0.15s ease-in-out, box-shadow 0.15s ease-in-out;  
}  
  
.BillingForm .card-field.StripeElement--focus {  
  outline: 0;  
  border-color: #86B7FE;  
  box-shadow: 0 0 0 .25rem rgba(13, 110, 253, 0.25);  
}
```

These styles might look complicated. But we are just copying them from the other form fields on the page to make sure that the card field looks like them.

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.tsx` with this.

```
const handleFormSubmit: BillingFormType["onSubmit"] = async (
  storage,
  info
) => {
  if (info.error) {
    onError(info.error);
    return;
  }

  setIsLoading(true);

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

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

return (
  <div className="Settings">
```

```
<Elements
  stripe={stripePromise}
  options={{
    fonts: [
      {
        cssSrc:
          "https://fonts.googleapis.com/css?family=Open+Sans:300,400,600,700,800",
      },
    ],
  }}
>
  <BillingForm isLoading={isLoading} onSubmit={handleFormSubmit} />
</Elements>
</div>
);
```

◆ CHANGE Add the following imports to the header.

```
import { Elements } from "@stripe/react-stripe-js";
import { BillingForm, BillingFormType } 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.

To initialize the Stripe Elements we pass in the `Stripe.js` object that we loaded [a couple of chapters ago](#). This Elements component needs to wrap around any Stripe React components.

The Stripe elements are loaded inside an `IFrame`. So if we are using any custom fonts, we'll need to include them explicitly. As covered in the [Stripe docs](#).

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.

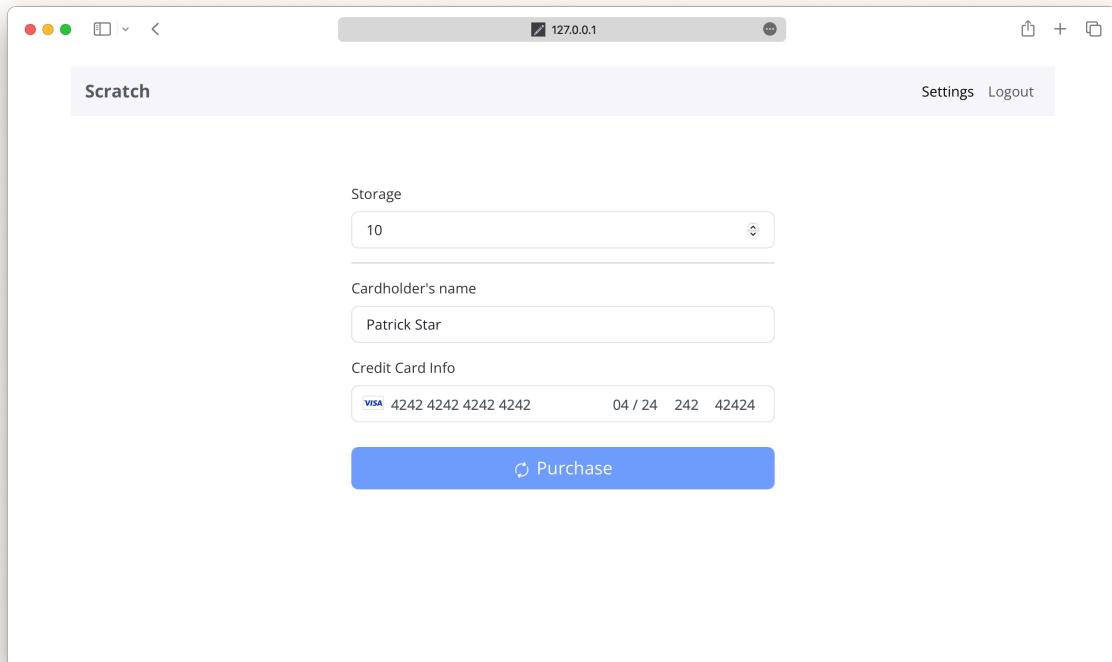
```
@media all and (min-width: 480px) {
  .Settings {
    padding: 60px 0;
```

```
}

.Settings form {
  margin: 0 auto;
  max-width: 480px;
}

}
```

This ensures that our form displays properly for larger screens.



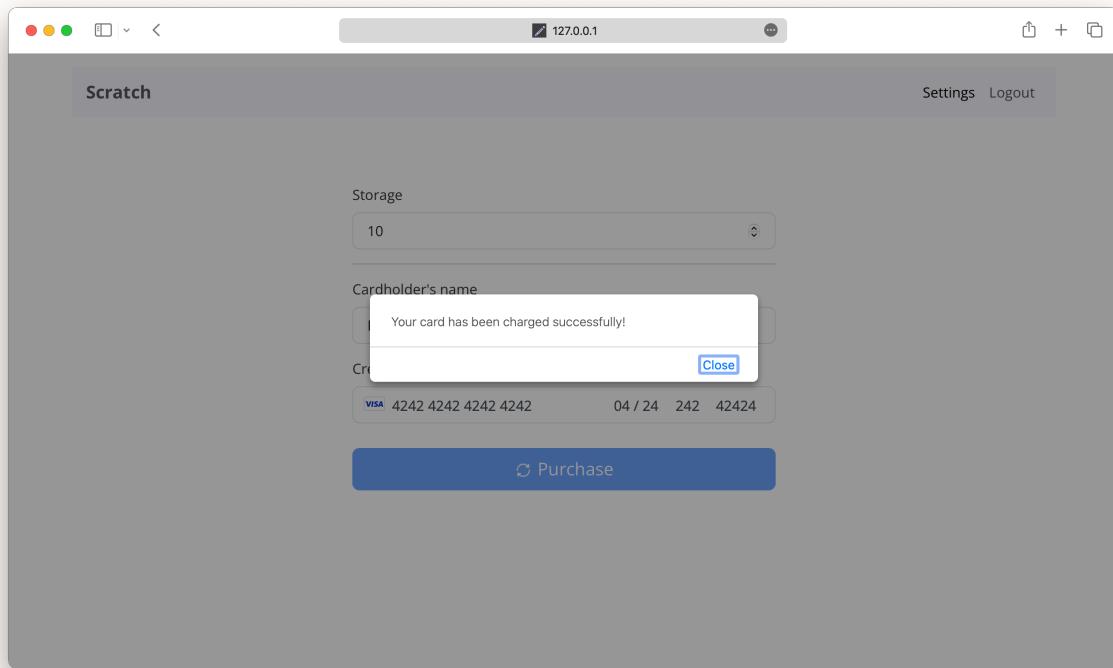
Settings screen with billing form screenshot

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.

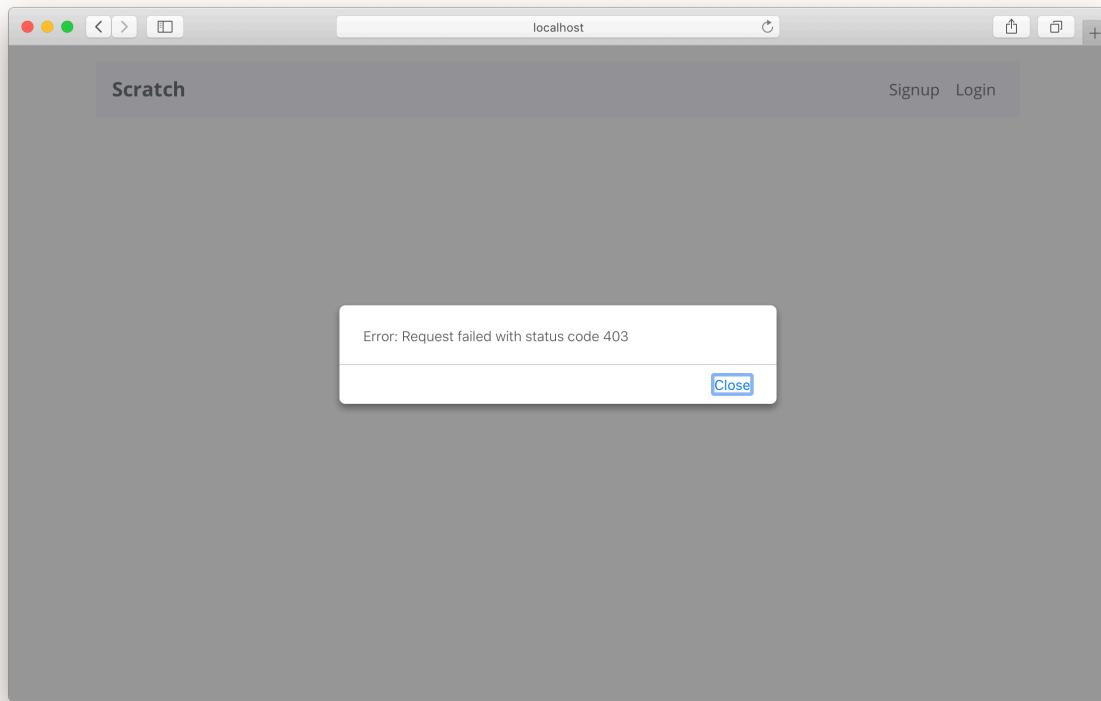
You can read more about the Stripe test cards in the [Stripe API Docs here](#).

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](#)

## **Securing React pages**

# 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 file in components src/components/AuthenticatedRoute.tsx.

```
import { ReactElement } from "react";
import { Navigate, useLocation } from "react-router-dom";
import { useAppContext } from "../lib/contextLib";

export default function AuthenticatedRoute({
  children,
}: {
  children: ReactElement;
}): ReactElement {
  const { pathname, search } = useLocation();
  const { isAuthenticated } = useAppContext();

  if (!isAuthenticated) {
    return <Navigate to={`/login?redirect=${pathname}${search}`} />;
  }

  return children;
}
```

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 `Navigate` 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 Next, add the following file in components `src/components/UnauthenticatedRoute.tsx`.

```
import { cloneElement, ReactElement } from "react";
import { Navigate } from "react-router-dom";
import { useApplicationContext } from "../lib/contextLib";

interface Props {
  children: ReactElement;
}

export default function UnauthenticatedRoute(props: Props): ReactElement {
  const { isAuthenticated } = useApplicationContext();
  const { children } = props;

  if (isAuthenticated) {
    return <Navigate to="/" />;
  }

  return cloneElement(children, props);
}
```

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 `Navigate` component to simply send the user to the homepage.

The `cloneElement` above makes sure that passed in `state` is handled correctly for child components of `UnauthenticatedRoute` routes.

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, we switch to our new redirect routes.

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

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

◆ CHANGE They should now look like so:

```
<Route
  path="/login"
  element={
    <UnauthenticatedRoute>
      <Login />
    </UnauthenticatedRoute>
  }
/>
<Route
  path="/signup"
  element={
    <UnauthenticatedRoute>
      <Signup />
    </UnauthenticatedRoute>
  }
/>
<Route
```

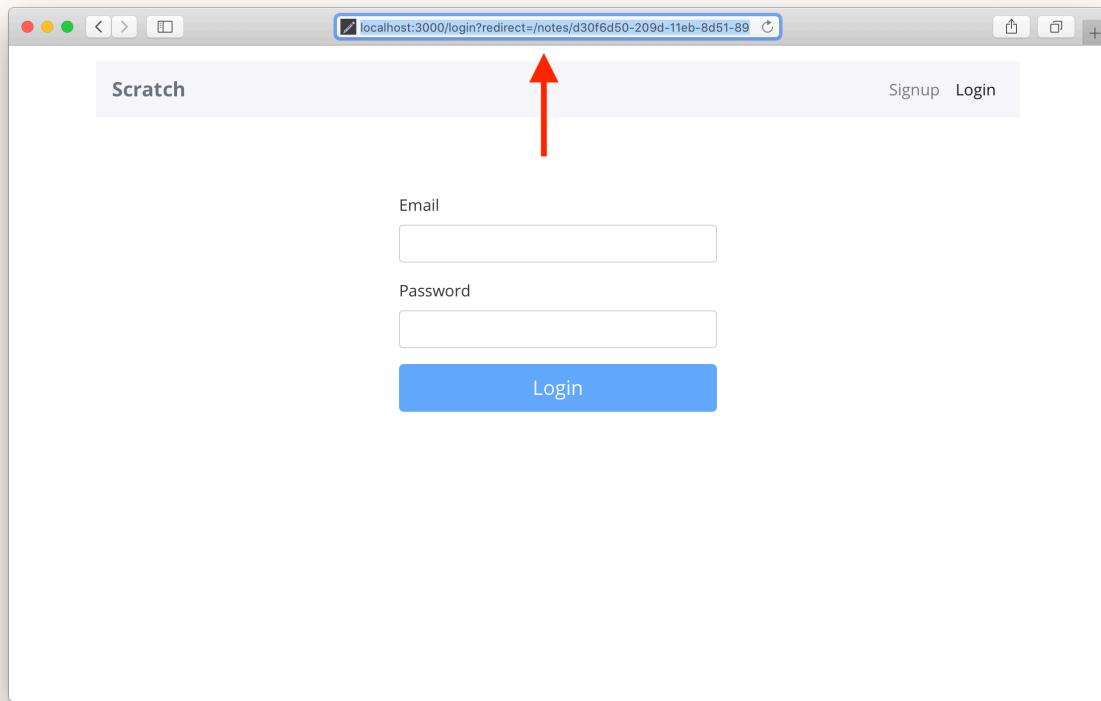
```
path="/settings"
element={
  <AuthenticatedRoute>
    <Settings />
  </AuthenticatedRoute>
}
/>
<Route
  path="/notes/new"
  element={
    <AuthenticatedRoute>
      <NewNote />
    </AuthenticatedRoute>
  }
/>

<Route
  path="/notes/:id"
  element={
    <AuthenticatedRoute>
      <Notes />
    </AuthenticatedRoute>
  }
/>
```

◆ CHANGE Then import them in the header of `src/Routes.tsx`.

```
import AuthenticatedRoute from "./components/AuthenticatedRoute.tsx";
import UnauthenticatedRoute from "./components/UnauthenticatedRoute.tsx";
```

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.tsx` below the imports and interface.

```
function querystring(name: string, url = window.location.href) {  
  const parsedName = name.replace(/[]/g, "\\$&");  
  const regex = new RegExp(`[?&]${parsedName}(=([^\#]*|\#|$)` , "i");  
  const results = regex.exec(url);  
  
  if (!results || !results[2]) {  
    return false;  
  }  
  
  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(props: Props) {  
  const { isAuthenticated } = useApplicationContext();  
  const { children } = props;  
  const redirect = querystring("redirect");
```

```
if (isAuthenticated) {  
  return <Navigate to={redirect || "/"} />;  
}  
  
return cloneElement(children, props);  
}
```

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

```
nav("/");
```

◆ CHANGE Also, remove the hook declaration.

```
const nav = useNavigate();
```

◆ CHANGE Finally, remove the import.

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

Now our login page should redirect after we login.

## 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
```

And that's it! Our app is ready to go live.

Next we'll be deploying our serverless app to production. And we'll do it using our own domain!



### Help and discussion

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

# **Using Custom Domains**

# Getting Production Ready

Now that we've gone through the basics of creating a full-stack serverless app, you are ready to deploy it to production.

## Deploy to Prod

We are now going to deploy our app to prod. You can go ahead and stop the local development environments for SST and React.

◆ **CHANGE** Run the following **in your project root**.

```
$ npx sst deploy --stage production
```

This command will take a few minutes as it'll deploy your app to a completely new environment. Recall that we are deploying to a separate prod environment because we don't want to affect our users while we are actively developing our app. This ensures that we have a separate local dev environment and a separate prod environment.

**Info:** The production name that we are using is arbitrary. We can call it `prod` or `live`. SST just uses the string internally to create a new version of your app.

At the end of the deploy process you should see something like this.

```
+ Complete
  Api: https://7qdwu0iuga.execute-api.us-east-1.amazonaws.com
  Frontend: https://d1wyq16hcztjw.cloudfront.net
  ...
  ...
```

## Set Secrets in Prod

We also need to configure our secrets for production. You'll recall we had previously [configured secrets for our local stage](#).

We'll do the same here but for production.

◆ CHANGE Run the following **in your project root**.

```
$ npx sst secret set --stage production StripeSecretKey  
↳ <YOUR_STRIPE_SECRET_TEST_KEY>
```

**Note:** For this guide we are using the same Stripe secret key in production but you'll likely be using different values in production.

You can run `npx sst secret list --stage production` to see the secrets for prod.

Our full-stack serverless app is almost ready to go. You can play around with the prod version.

## Custom Domains

However the API is currently on an endpoint that's auto-generated by [API Gateway](#).

```
https://5bv7x0iuga.execute-api.us-east-1.amazonaws.com
```

And the frontend React app is hosted on an auto-generated [CloudFront](#) domain.

```
https://d3j4c16hcztjw.cloudfront.net
```

We want to host these on our own domain. Let's look at that next.



### Help and discussion

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

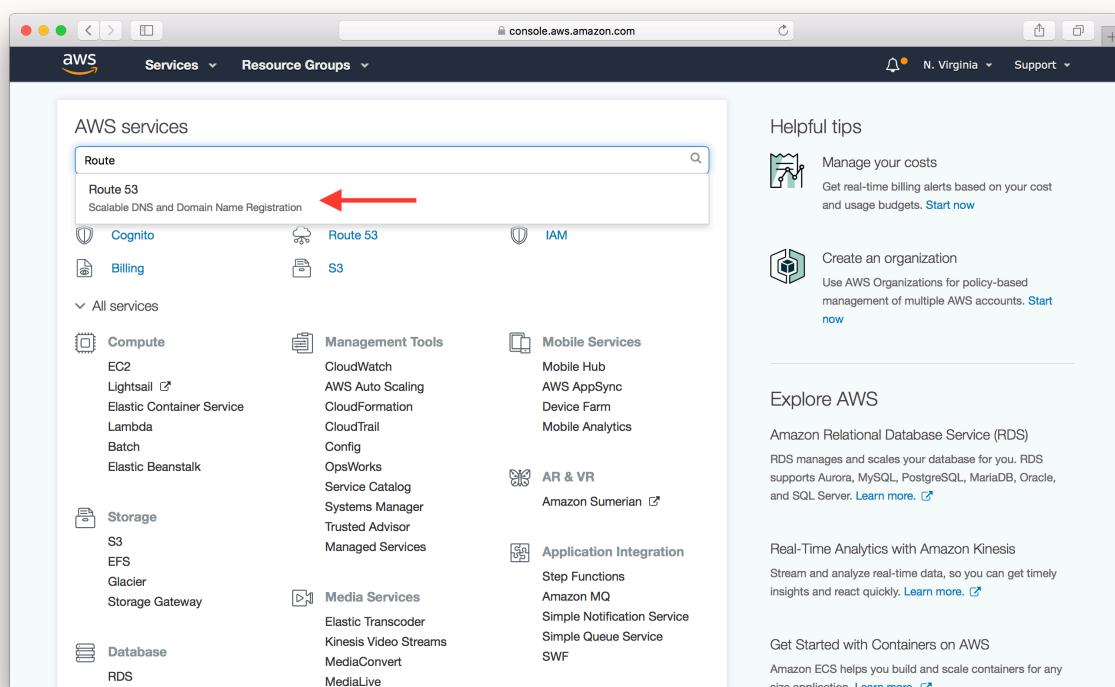
# Purchase a Domain with Route 53

To host our app on our own custom domain, we'll be using [Amazon Route 53](#).

If you are following this guide but are not ready to purchase a new domain, you can skip this chapter.

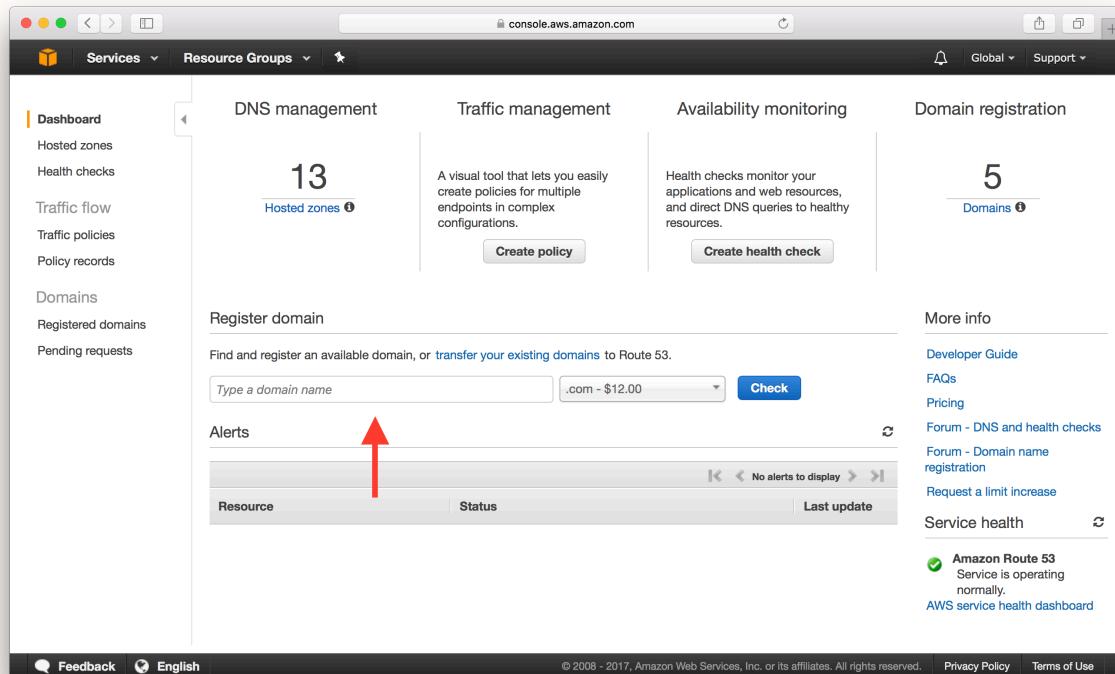
On the other hand, if you have an existing domain that is not on AWS, follow these docs to [move it over to Amazon Route 53](#).

Let's get started. In your [AWS Console](#) head over to the Route 53 section in the list of services.



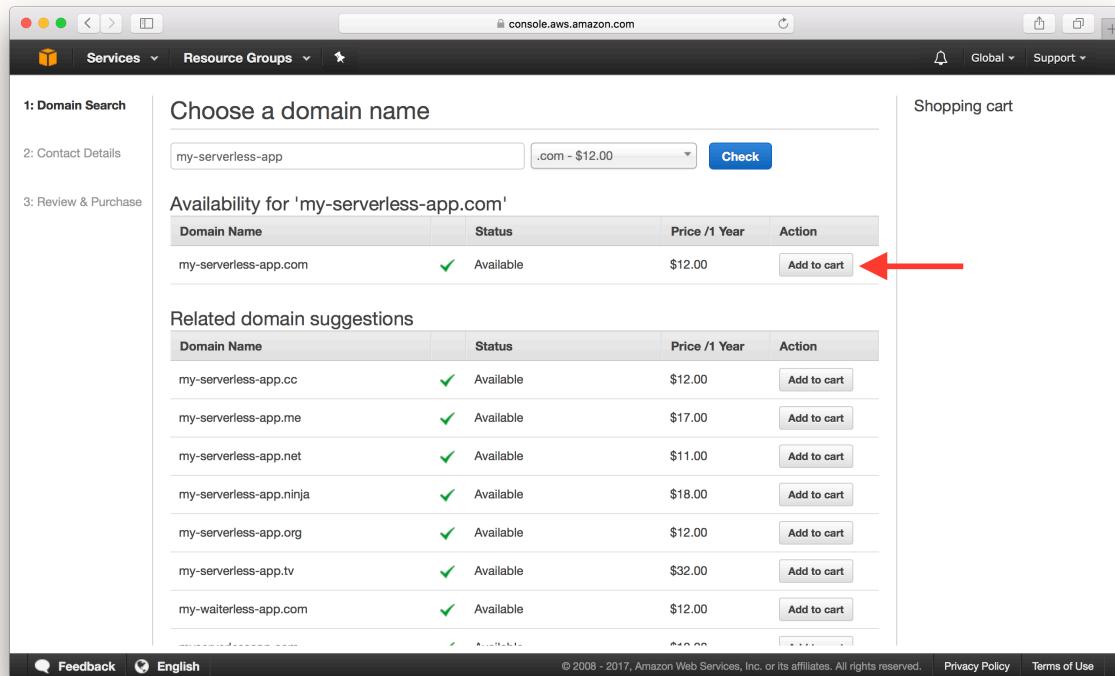
Select Route 53 service screenshot

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

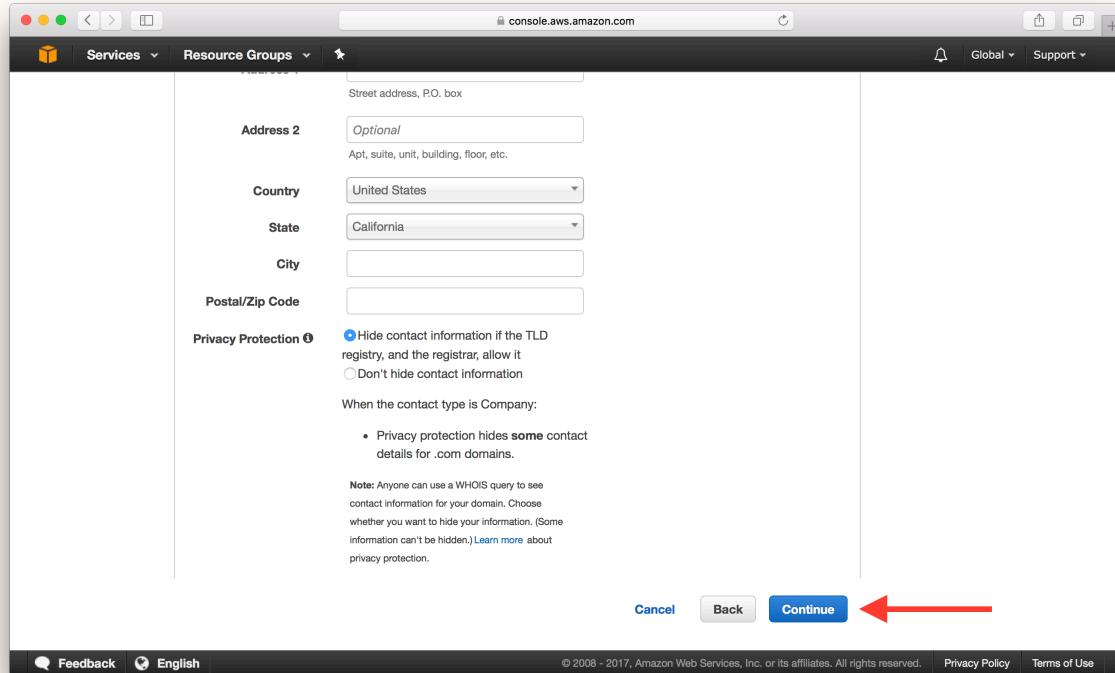
And hit **Continue** at the bottom of the page.

The screenshot shows a list of related domain suggestions for a serverless application. The table includes columns for Domain Name, Status, Price /1 Year, and Action (Add to cart). Most domains are listed as Available at various prices like \$12.00, \$17.00, etc. A red arrow points to the '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-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

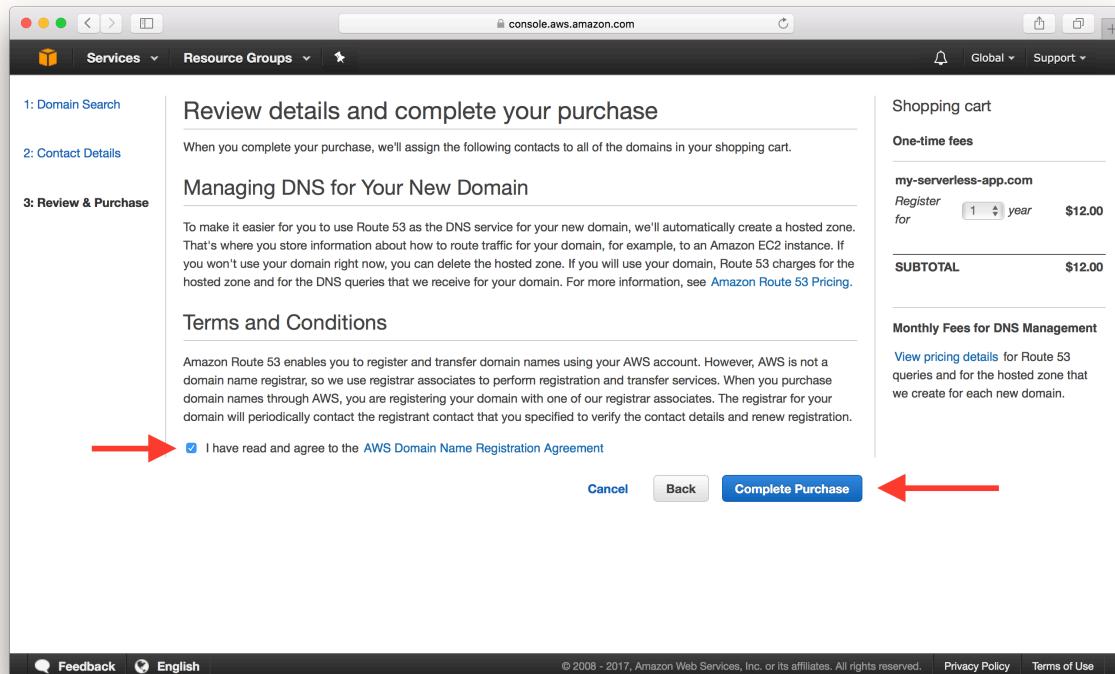
Continue to contact details screenshot

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, let's use this domain in our serverless app.



### Help and discussion

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

# Custom Domains in serverless APIs

In the [previous chapter](#) we purchased a new domain on [Route 53](#). Now let's use it for our serverless API.

◆ CHANGE In your `infra/api.ts` add this above the `transform: {}` line.

```
domain: $app.stage === "production" ? "<api.yourdomainhere.com>" : undefined,
```

**Note:** Without specifying the API subdomain, the deployment will attempt to create duplicate A (IPv4) and AAAA (IPv6) DNS records and error.

This tells SST that we want to use a custom domain **if** we are deploying to the `production` stage. We are not setting one for our `dev` stage, or any other stage.

We could for example, base it on the stage name, `api-${app.stage}.my-serverless-app.com`. So for `dev` it might be `api-dev.my-serverless-app.com`. But we'll leave that as an exercise for you.

The `$app` is a global variable that's available in our config. You can [learn more about it here](#).

## Deploy the App

Let's deploy these changes to prod.

◆ CHANGE Run the following from **your project root**.

```
$ npx sst deploy --stage production
```

**Note:** Deploying changes to custom domains can take a few minutes.

At the end of the deploy process you should see something like this.

```
+ Complete  
  Api: https://api.my-serverless-app.com  
  ...
```

This is great! We now have our app deployed to prod and our API has a custom domain.

Next, let's use our custom domain for our React app as well.



### Help and discussion

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

# Custom Domains for React Apps on AWS

In the [previous chapter](#) we configured a custom domain for our serverless API. Now let's do the same for our frontend React app.

◆ **CHANGE** In the `infra/web.ts` add the following above the `environment: {}` line.

```
domain:  
  $app.stage === "production"  
  ? {  
    name: "<yourdomainhere.com>",  
    redirects: ["www.<yourdomainhere.com>"],  
  }  
  : undefined,
```

Just like the API case, we want to use our custom domain **if** we are deploying to the production stage. This means that when we are using our app locally or deploying to any other stage, it won't be using the custom domain.

Of course, you can change this if you'd like to use a custom domain for the other stages. You can use something like  `${app.stage}.my-serverless-app.com`. So for dev it'll be `dev.my-serverless-app.com`. But we'll leave this as an exercise for you.

The `redirects` prop is necessary because we want visitors of `www.my-serverless-app.com` to be redirected to the URL we want to use. It's a good idea to support both the `www.` and root versions of our domain. You can switch these around so that the root domain redirects to the `www.` version as well.

You won't need to set the `redirects` for the non-prod versions because we don't need `www.` versions for those.

## Deploy the App

Just like the previous chapter, we need to update these changes in prod.

◆ CHANGE Run the following from your project root.

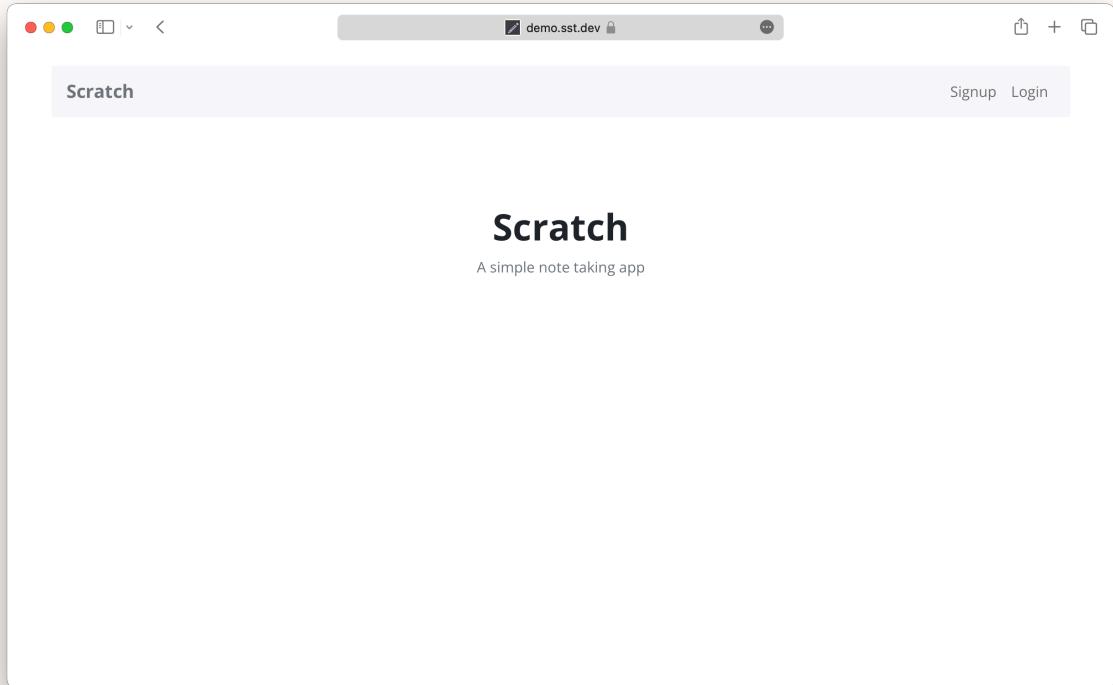
```
$ npx sst deploy --stage production
```

**Note:** Deploying changes to custom domains can take a few minutes.

At the end of the deploy process you should see something like this.

```
+ Complete
Api: https://api.my-serverless-app.com
Frontend: https://my-serverless-app.com
...
```

And that's it! Our React.js app is now deployed to prod under our own domain!



App update live screenshot

## Commit the Changes

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

```
$ git add .  
$ git commit -m "Setting up custom domains"  
$ git push
```

At this stage our full-stack serverless app is pretty much complete. In the next couple of optional sections we are going 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.



### Help and discussion

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

# **Automating deployments**

# Creating a CI/CD Pipeline for serverless

So to recap, here's what we've created so far.

- A full-stack serverless app that includes:
  - Storage with DynamoDB and S3
  - API
  - Auth with Cognito
  - Frontend in React
- A way to handle secrets locally
- A way to run unit tests
- Deployed to a prod environment with a custom domain

All of this is neatly committed in a Git repo.

So far we've been deploying our app locally through our command line — `npx sst deploy`. But if we had multiple people on our team, or if we were working on different features at the same time, we won't be able to work on our app because the changes would overwrite each other.

To fix this we are going to implement a CI/CD pipeline for our full-stack serverless app.

## What is a CI/CD Pipeline

CI/CD or Continuous Integration/Continuous Delivery is the process of automating deployments by tying it to our source control system. So that when new code changes are pushed, our app is automatically deployed.

A CI/CD pipeline usually includes multiple environments. An environment is one where there are multiple instances of our deployed app. So we can have an environment called *production* that our users will be using. And *development* environments that we can use while developing our app.

Here is what our workflow is going to look like:

- Our repo will be connected to our CI/CD service.
- Any commits that are pushed to the production branch will be automatically deployed to the production stage.
- Any PRs will be automatically deployed as preview environments.

Our workflow is fairly simple. But as your team grows, you'll need to add additional dev, staging, or preview environments.

## CI/CD for Serverless

There are many common CI/CD services, like [GitHub Actions](#), [Travis CI](#) or [CircleCI](#). These usually require you to manually configure the above pipeline. It involves a fair bit of scripts and configuration.

SST makes this easier with the [SST Console](#)'s [Autodeploy](#) feature. It also shows you all the resources in your app and allows you to monitor and debug them.

We should mention that you don't have to use the SST Console. And this section is completely optional.

Let's get started with setting up the SST Console.



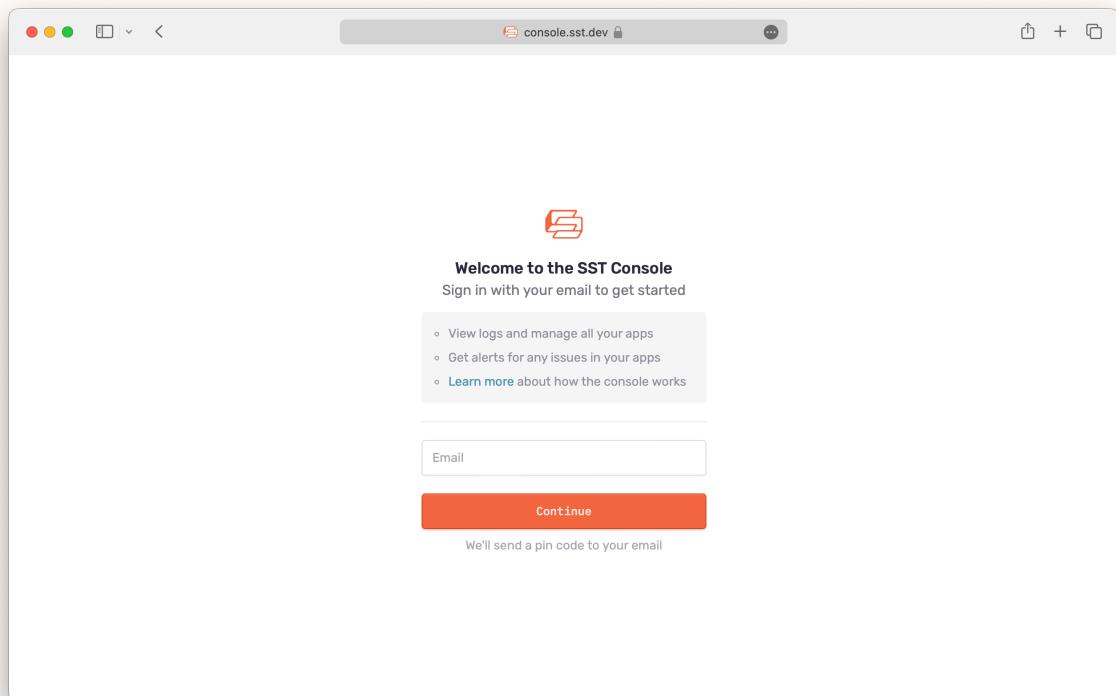
### Help and discussion

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

# Setting up the SST Console

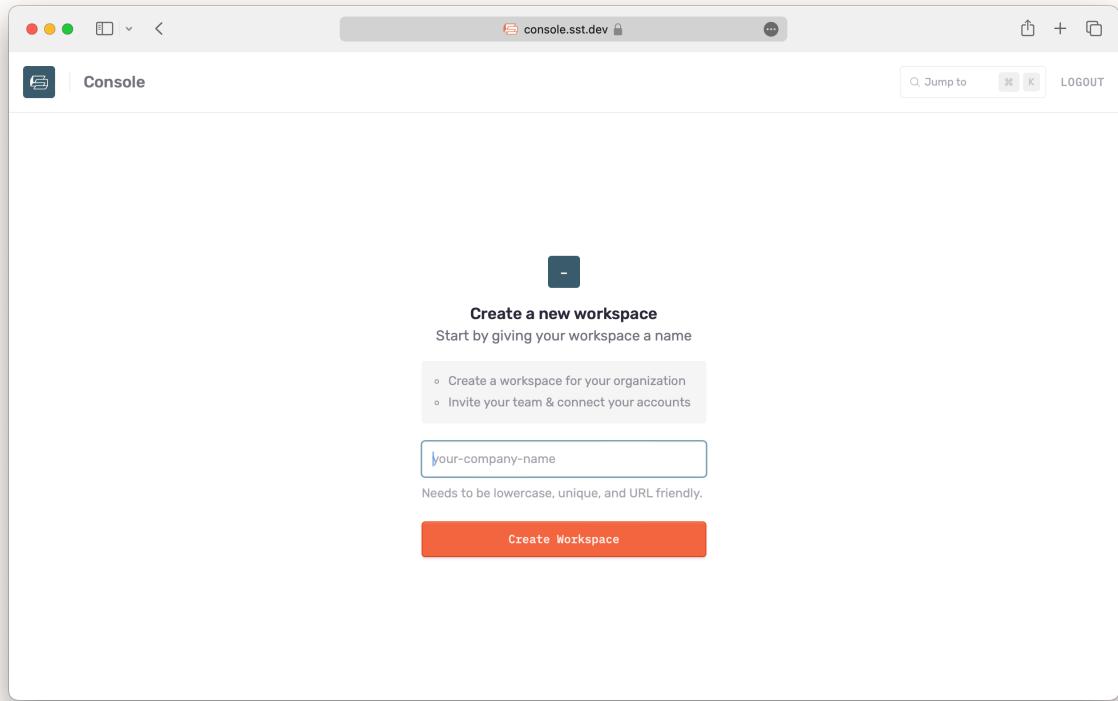
We are going to set up the [SST Console](#) to auto-deploy our app and manage our environments.

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



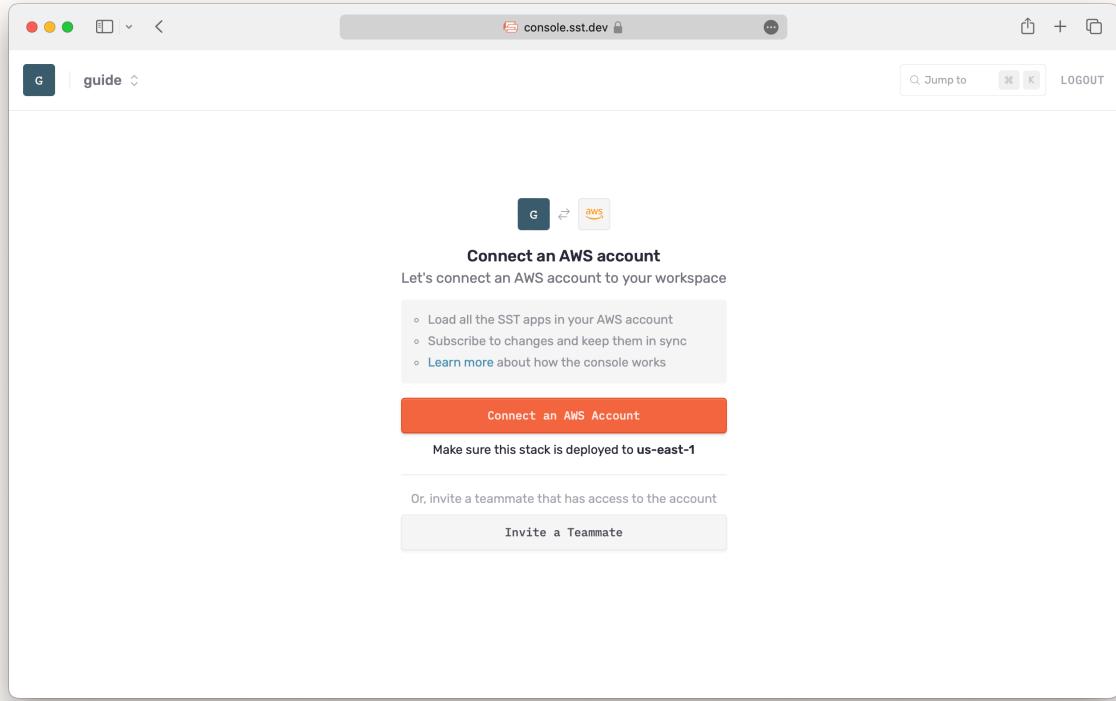
Create new SST Console account

Let's **create your workspace**.



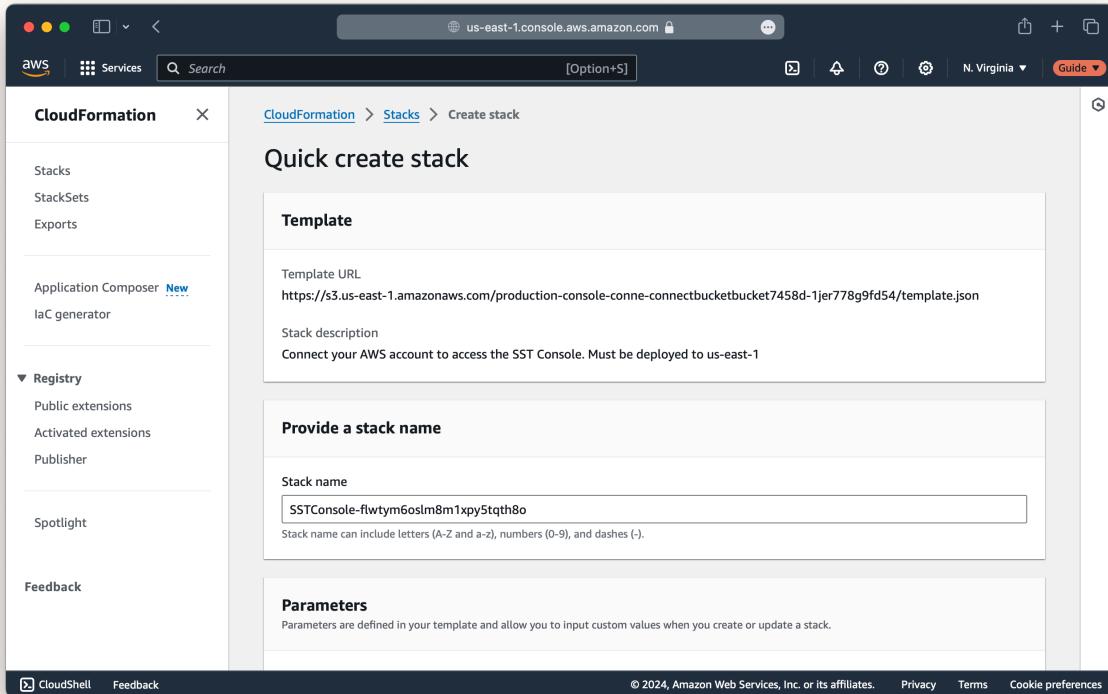
Create SST Console workspace

Next, **connect your AWS account**.



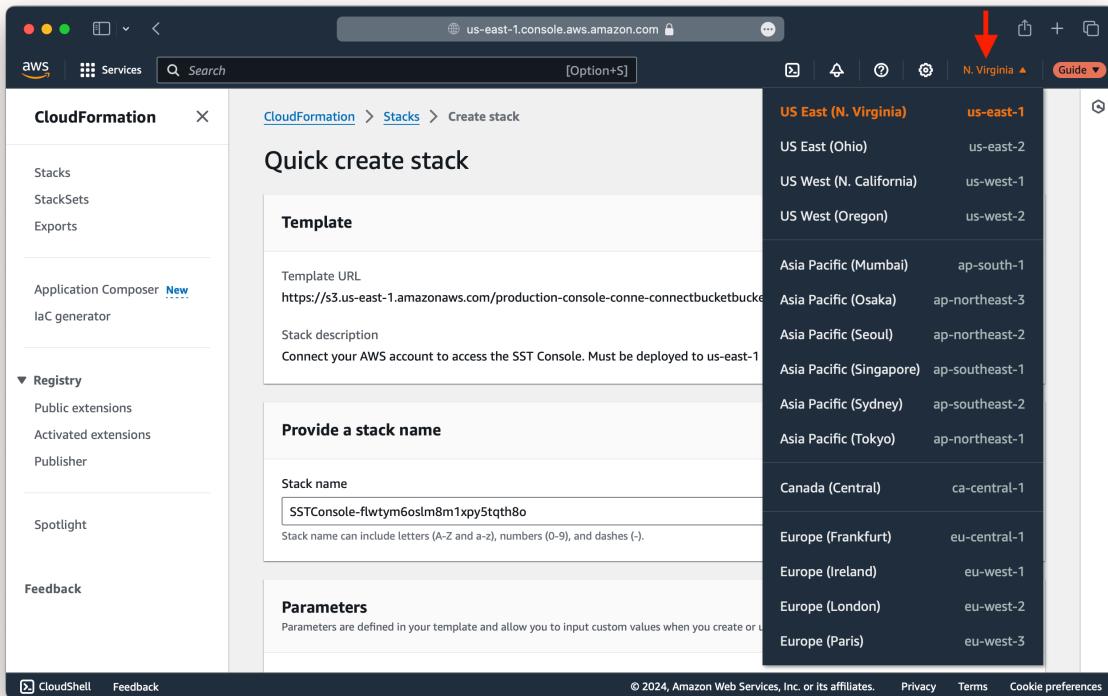
### Connect AWS account in SST Console

This will send you to the AWS Console and ask you to create a CloudFormation stack.



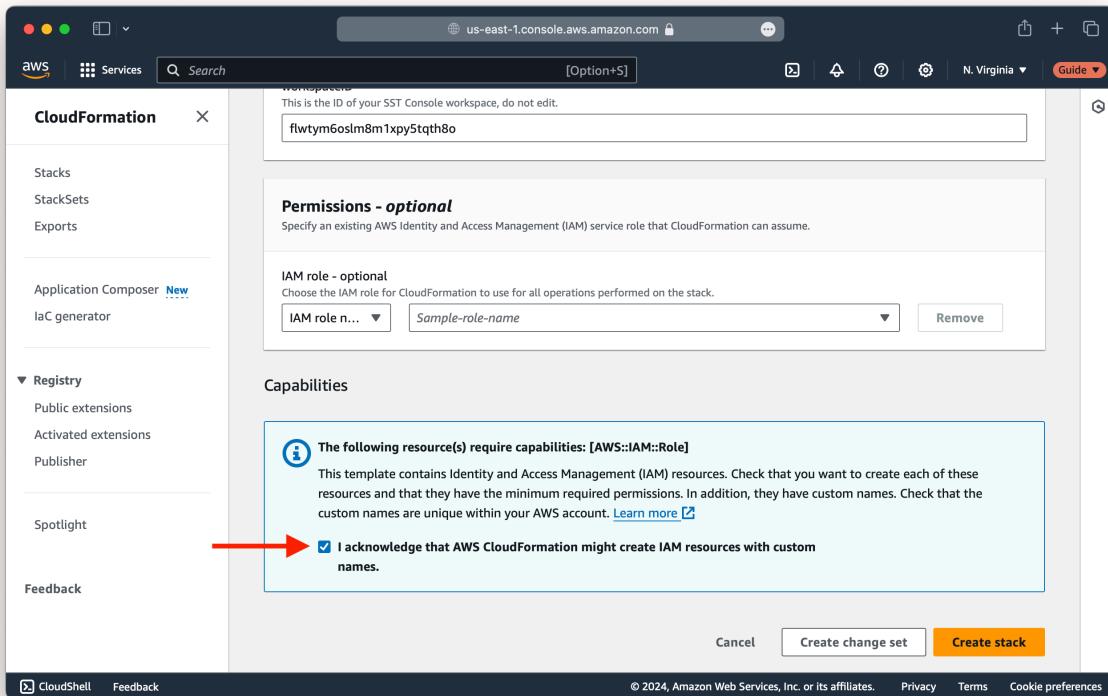
### AWS create CloudFormation stack

This stack needs to be in **us-east-1**. So make sure you use the dropdown at the top right to check that you are in the right region.



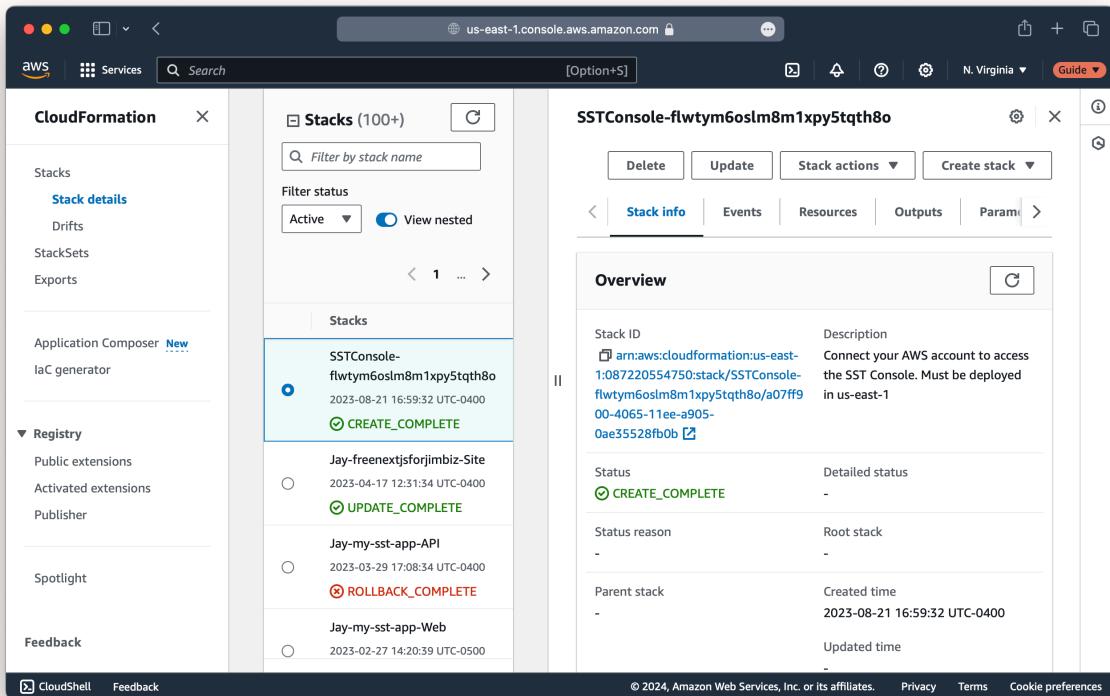
AWS Console check region

Scroll down, **confirm** the checkbox at the bottom and click **Create stack**.



AWS click create CloudFormation stack

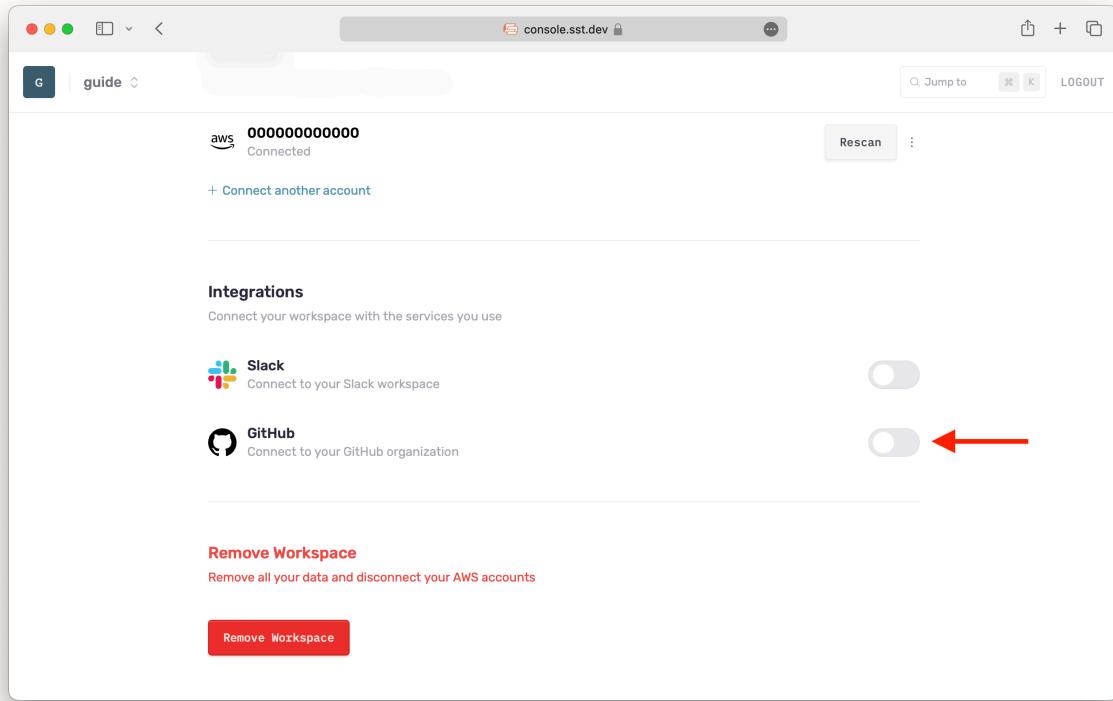
It will take a couple of minutes to create the stack.



### AWS CloudFormation stack create complete

Once complete, head back to the SST Console. It'll take a minute to scan your AWS account for your SST apps.

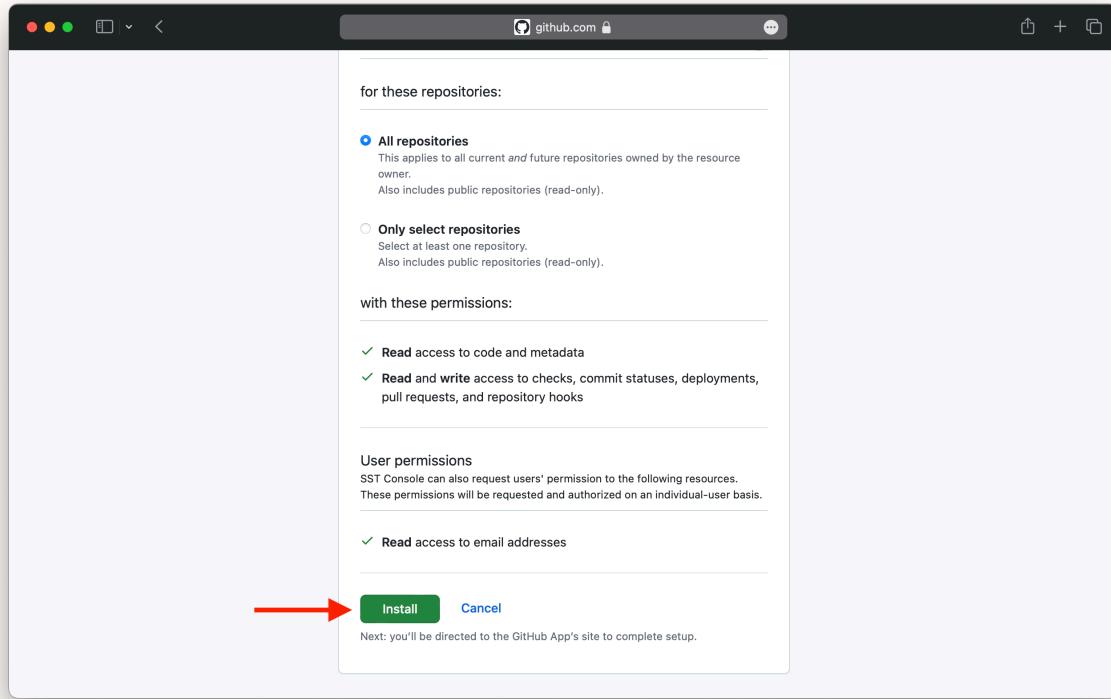
While it's doing that, let's link our GitHub. Click on **Manage workspace**, scroll down to the **Integrations**, and enable **GitHub**.



### Enable GitHub integration in SST Console

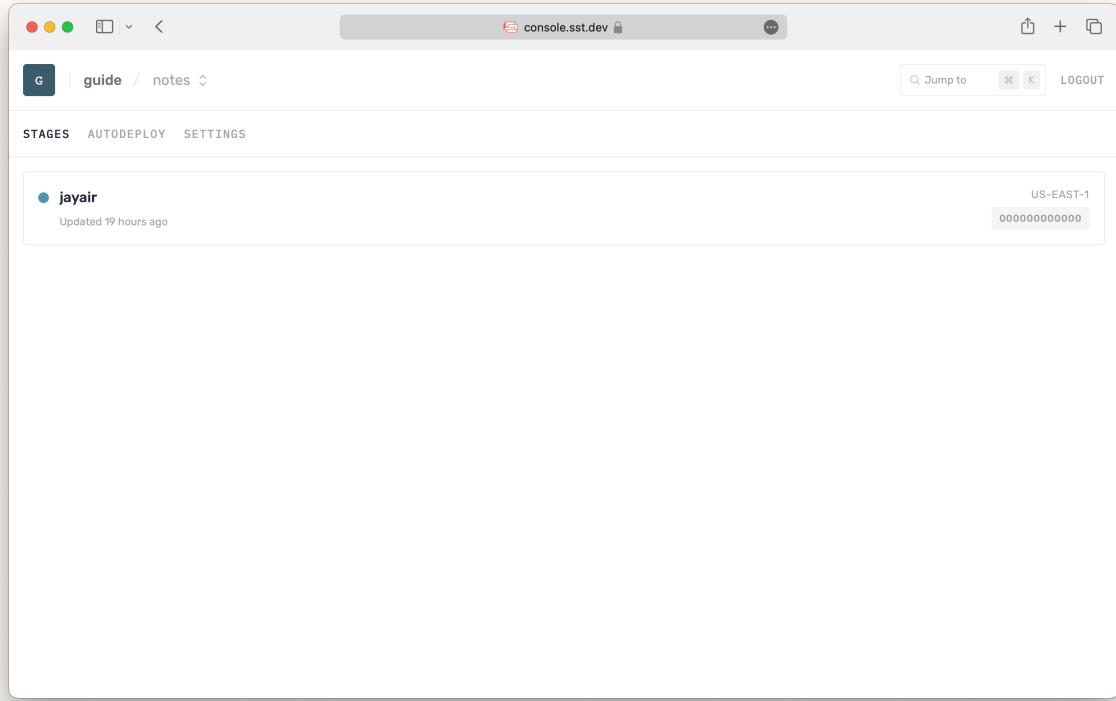
You'll be asked to select where you want to install the SST Console integration. You can either pick your personal account or any organizations you are a part of. This is where your notes app repo has been created.

Once you select where you want to install it, scroll down and click **Install**.



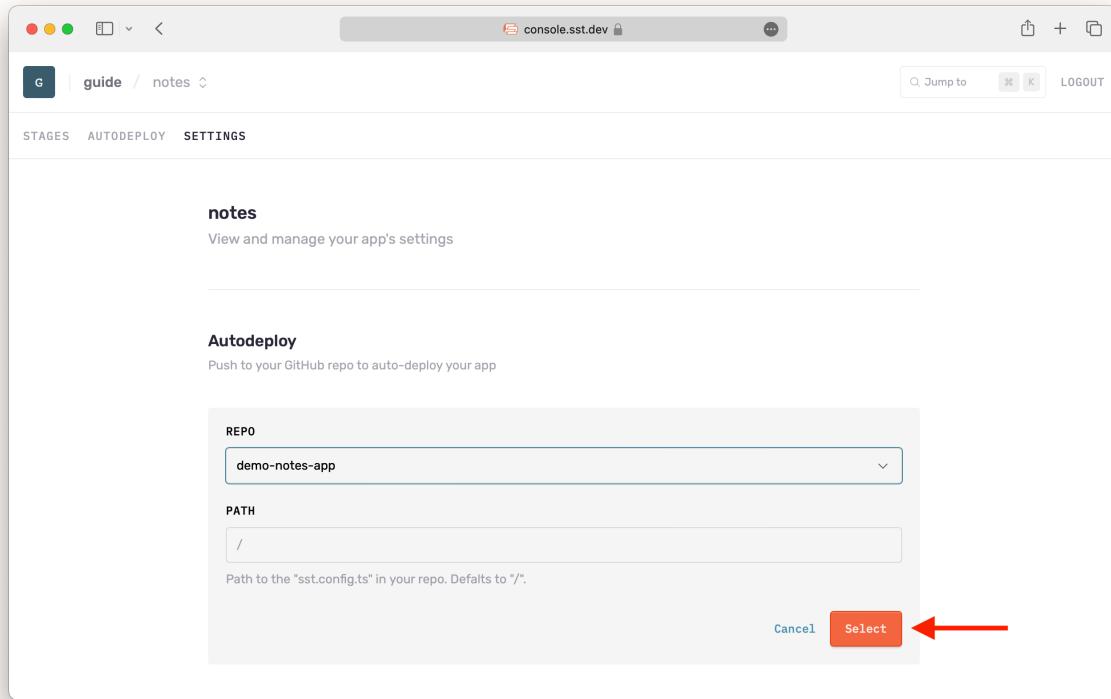
Install SST Console in GitHub

Now your GitHub integration should be enabled. And hopefully the Console should be done scanning your AWS account. You should see your notes app with your personal stage.



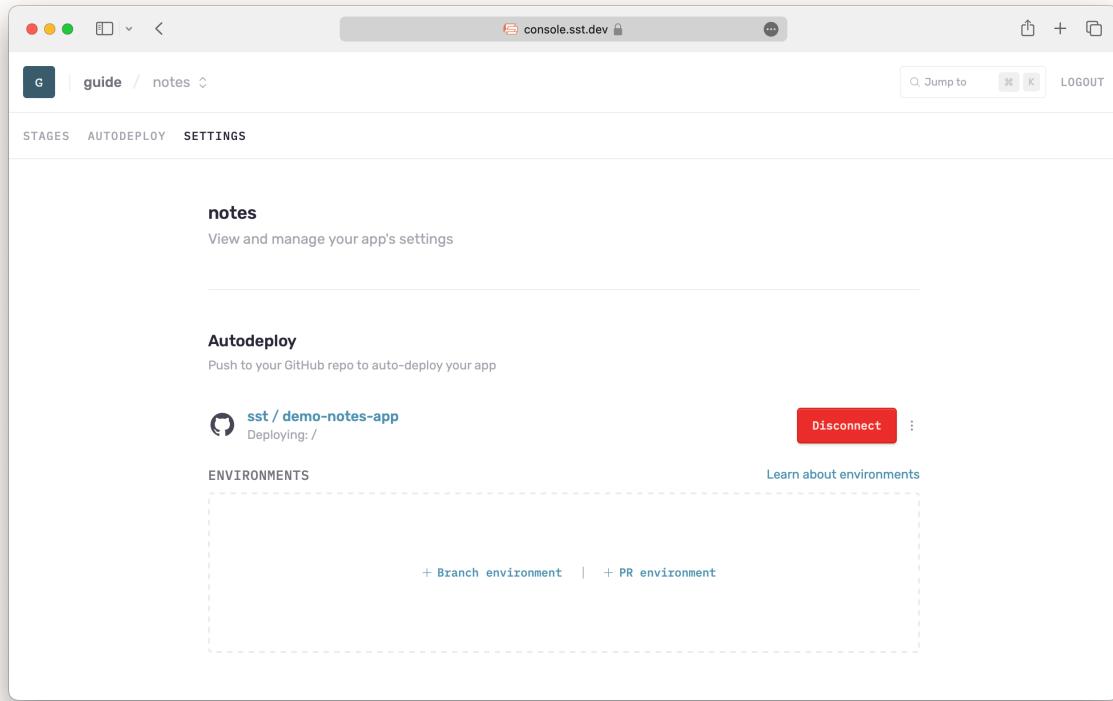
Notes app in SST Console

Here you can see the resources in your stage, the logs from your functions, and any issues that have been detected. For now, let's head over to the **Settings** > **Autodeploy** > pick your repo > and click **Select**.



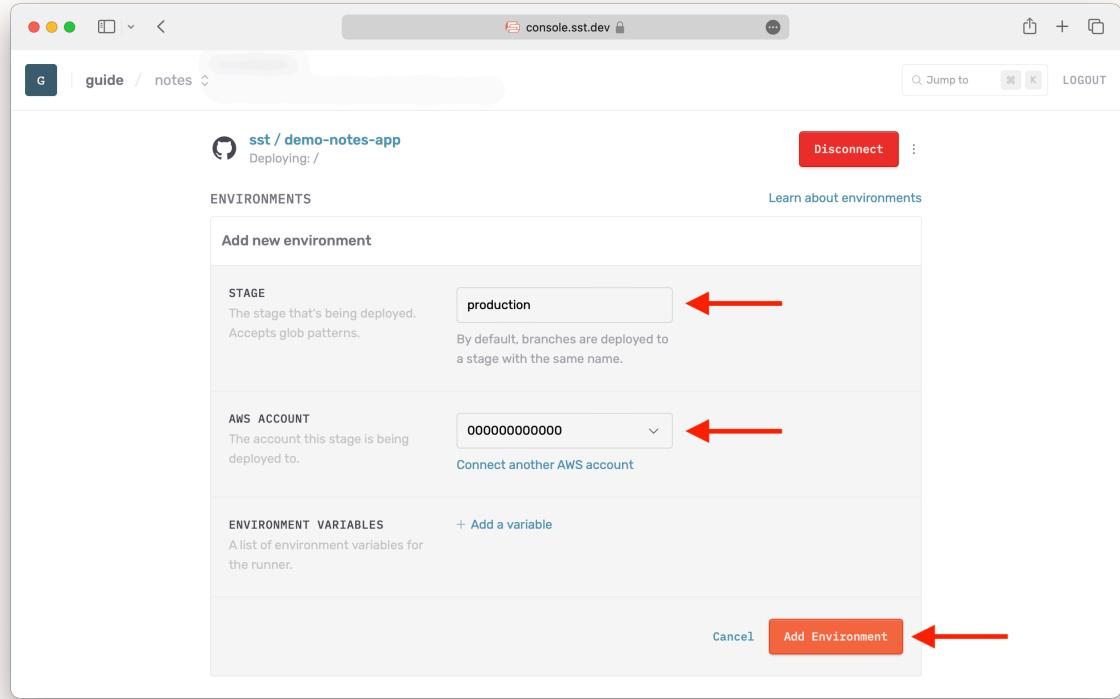
Select GitHub repo SST Console

Let's create a couple of environments. This tells the SST Console when to auto-deploy your app.



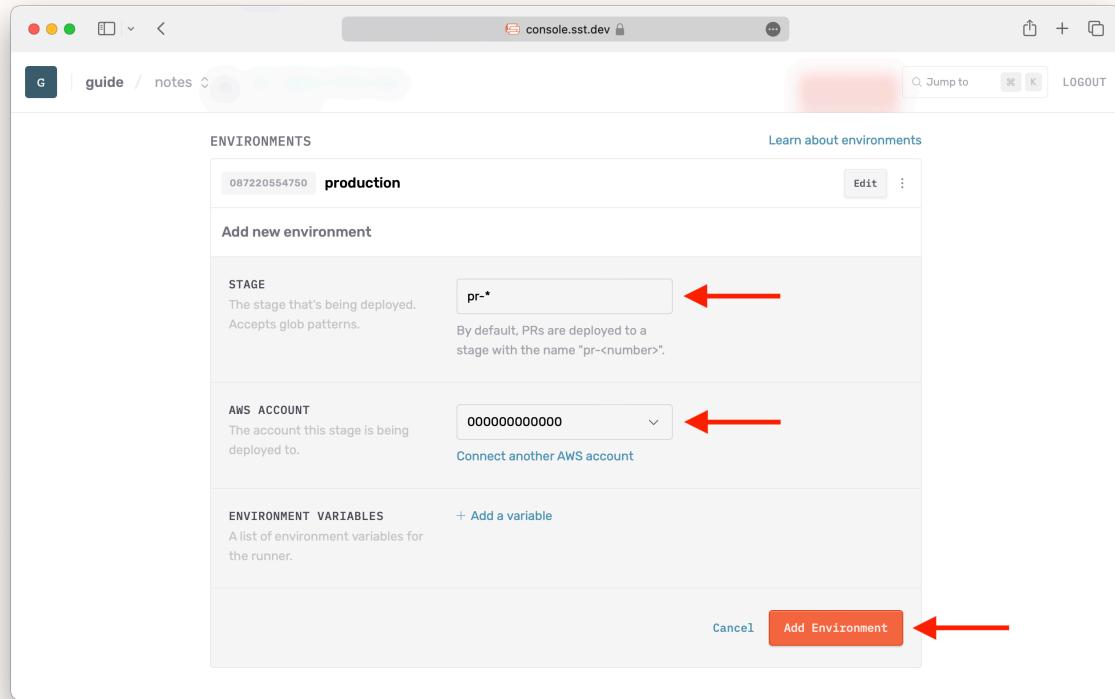
GitHub repo selected SST Console

We are going to create two environments. Starting with a **Branch environment**. Use **production** as the name, select your AWS account, and click **Add Environment**.



Create branch environment SST Console

Do the same for a **PR environment**.



### Create PR environment SST Console

The two above environments tell the Console that any stage with the name `production` or starting with `pr-` should be auto-deployed to the given AWS account. By default, the stage names are derived from the name of the branch.

So if you `git push` to a branch called `production`, the SST Console will auto-deploy that to a stage called `production`.

Let's do that next.



#### Help and discussion

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

# Deploying Through the SST Console

Now, we are ready to `git push` to deploy our app to production with the SST Console. If you recall from the [previous chapter](#), we configured it to auto-deploy the production branch.

Let's do that by first creating a production branch.

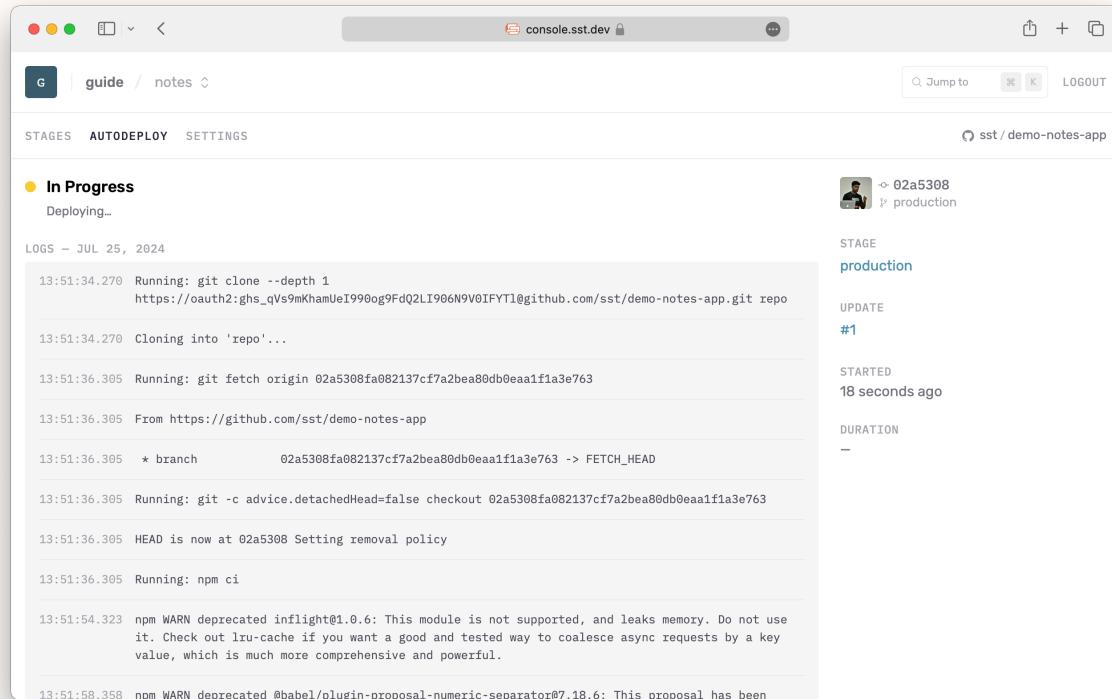
◆ CHANGE Run the following in the **project root**.

```
$ git checkout -b production
```

◆ CHANGE Now let's push this to GitHub.

```
$ git push --set-upstream origin production  
$ git push
```

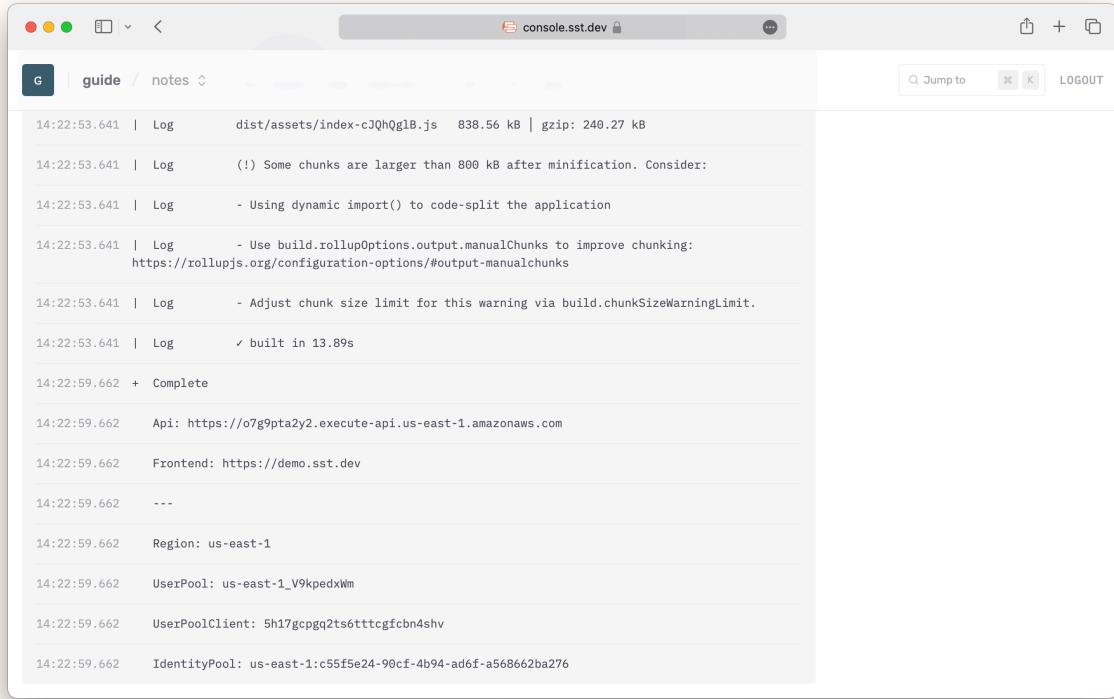
Now if you head into the **Autodeploy** tab for your app in the SST Console, you'll notice a new deployment in progress.



```
Logs — Jul 25, 2024
13:51:34.270 Running: git clone --depth 1
https://oauth2:ghs_qVs9mKhamUeI990og9FdQ2LI906N9V0IFYTl@github.com/sst/demo-notes-app.git repo
13:51:34.270 Cloning into 'repo'...
13:51:36.305 Running: git fetch origin 02a5308fa082137cf7a2bea80db0eaa1f1a3e763
13:51:36.305 From https://github.com/sst/demo-notes-app
13:51:36.305 * branch          02a5308fa082137cf7a2bea80db0eaa1f1a3e763 -> FETCH_HEAD
13:51:36.305 Running: git -c advice.detachedHead=false checkout 02a5308fa082137cf7a2bea80db0eaa1f1a3e763
13:51:36.305 HEAD is now at 02a5308 Setting removal policy
13:51:36.305 Running: npm ci
13:51:54.323 npm WARN deprecated inflight@1.0.6: This module is not supported, and leaks memory. Do not use it. Check out lru-cache if you want a good and tested way to coalesce async requests by a key value, which is much more comprehensive and powerful.
13:51:58.358 npm WARN deprecated @babel/plugin-proposal-numeric-separator@7.18.6: This proposal has been
```

### SST Console production deploy in progress

Once the deploy is complete, you'll notice the outputs at the bottom.

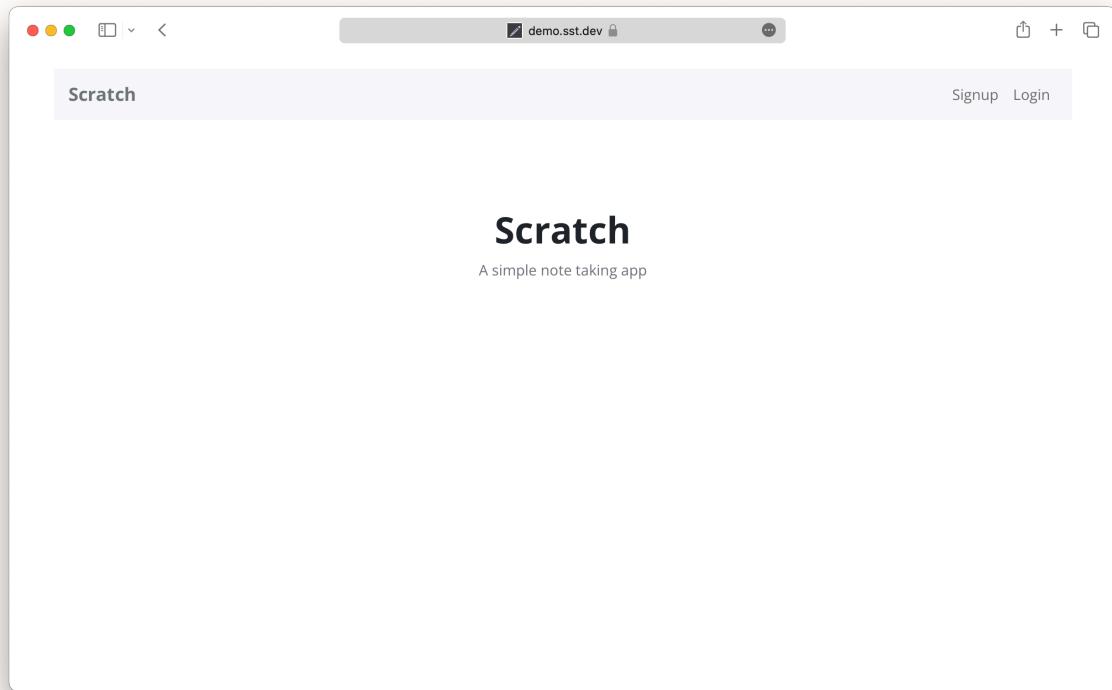
A screenshot of a web browser window titled "console.sst.dev". The page displays deployment logs for a project named "guide". The logs show the following output:

```
14:22:53.641 | Log      dist/assets/index-cJQhQg1B.js  838.56 kB | gzip: 240.27 kB
14:22:53.641 | Log      (!) Some chunks are larger than 800 kB after minification. Consider:
14:22:53.641 | Log      - Using dynamic import() to code-split the application
14:22:53.641 | Log      - Use build.rollupOptions.output.manualChunks to improve chunking:
14:22:53.641 | Log      https://rollupjs.org/configuration-options/#output-manualchunks
14:22:53.641 | Log      - Adjust chunk size limit for this warning via build.chunkSizeWarningLimit.
14:22:53.641 | Log      ✓ built in 13.89s
14:22:59.662 + Complete
14:22:59.662   Api: https://o7g9pta2y2.execute-api.us-east-1.amazonaws.com
14:22:59.662   Frontend: https://demo.sst.dev
14:22:59.662   ---
14:22:59.662   Region: us-east-1
14:22:59.662   UserPool: us-east-1_V9kpexWm
14:22:59.662   UserPoolClient: 5h17gcpgq2ts6tttcgfcbn4shv
14:22:59.662   IdentityPool: us-east-1:c55f5e24-90cf-4b94-ad6f-a568662ba276
```

Prod build stack outputs

## Test Our App in Production

Let's check out our app in production.



App update live screenshot

To give it a quick test, sign up for a new account and create a note. You can also test updating and removing a note. And also test out the billing page.

### Congrats! Your app is now live!

Let's wrap things up next.



### Help and discussion

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

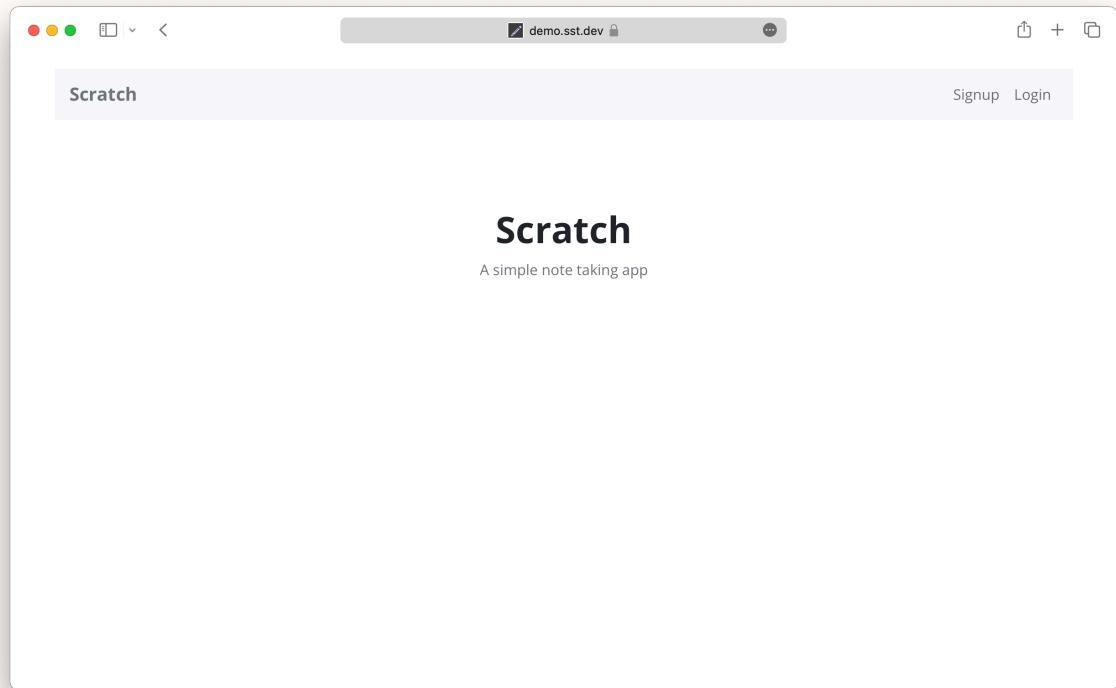
# **Conclusion**

# Wrapping Up

Congratulations on completing the guide!

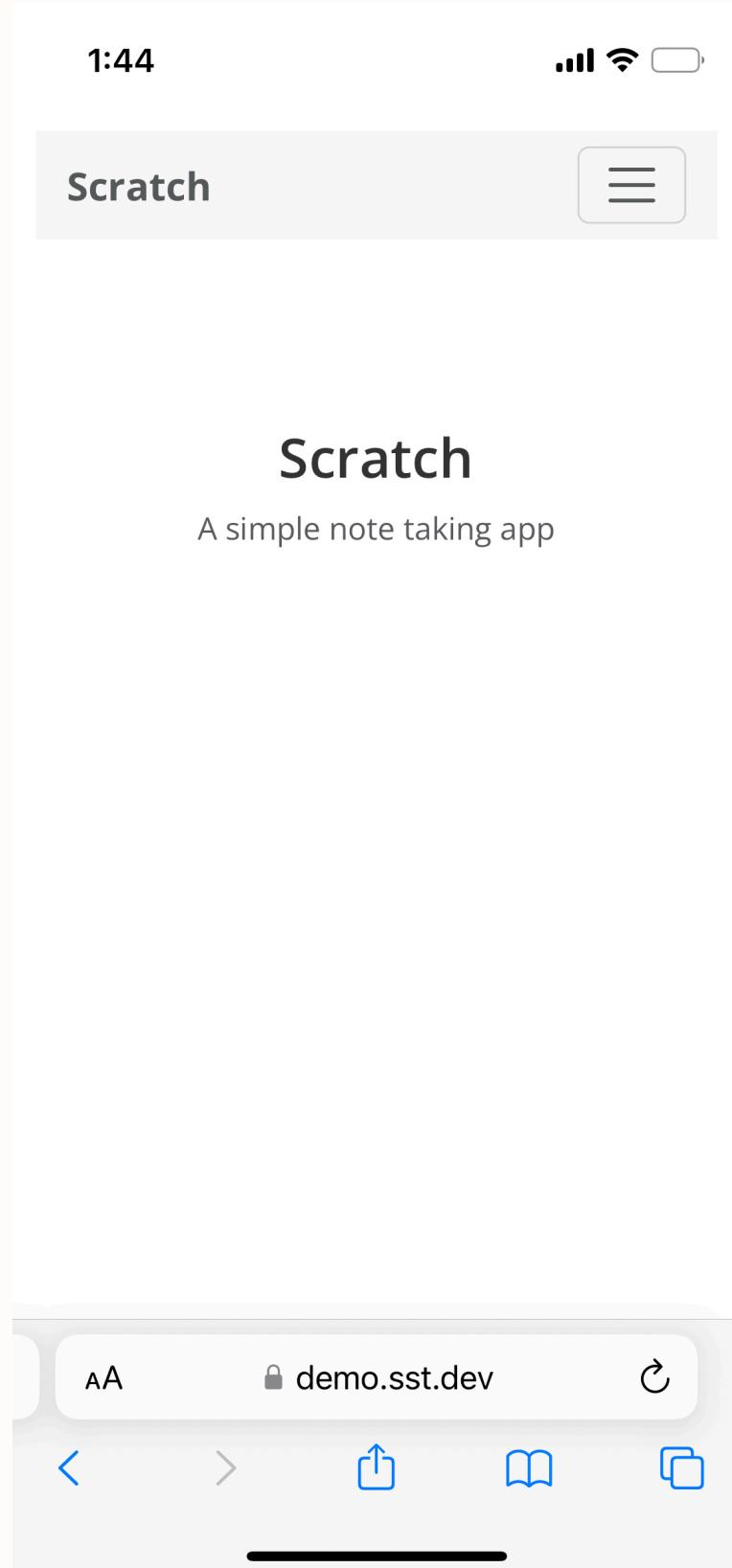
## App in Prod

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!



Completed app mobile screenshot

---

We hope what you've learned here can be adapted to fit the use case you have in mind.

We'd love to hear from you about your experience following this guide. Please send us any comments or feedback you might have, via [email](#). And [star our repo on GitHub](#).

Also, we'd love to feature what you've built with SST, please [send us a URL and brief description](#).

Thank you and we hope you found this guide helpful!



#### **Help and discussion**

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



#### **For reference, here is the code we are using**

[Notes App Source](#)

# Further Reading

Once you've completed the guide, you are probably going to use SST 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.

- [SST Documentation](#): Documentation for SST
- [DynamoDB, explained](#): A Primer on the DynamoDB NoSQL database
- [Learn React](#): The official React docs
- [Vite docs](#): The official Vite docs
- [React-Bootstrap Docs](#): The official React-Bootstrap docs
- [Learn React Router](#): The official React Router docs
- [AWS Amplify API Reference](#): The AWS Amplify API reference
- [Vitest Docs](#): The official Vitest docs
- [SST Console Docs](#): The SST Console 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.

The screenshot shows a web browser displaying the [sst.dev](https://sst.dev) website. The main content is the "What is Serverless?" page. At the top, there's a navigation bar with links for Docs, Guide, Blog, and Examples. To the right of the navigation is a search bar and social sharing icons. A prominent callout box on the right side is titled "PART OF THE SST Guide" and describes it as a guide to building full-stack apps with serverless and React. Below the main content, there's a "Table of Contents" sidebar listing various chapters and sections of the guide.

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](#).

A big thanks to our contributors for helping make SST 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**

SST 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 SST 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 SST 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 SST 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.

- **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 SST, 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 SST, 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 **guide and demo 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

### v8.0.1: Updating Seed to IAM roles (Current)

Sep 15, 2023: Minor changes to the setting up your project on Seed chapter.

- [Tutorial changes](#)

### v8.0: Updating to TypeScript

Aug 31, 2023: Using TS by default, switching to pnpm, and using Vite instead of Create React App. And archiving old chapters.

- [Tutorial changes](#)
- [Demo notes app source](#)

## v7.4: Upgrading to SST v2.5

Apr 9, 2023: Updating the guide and code to use the latest `sst bind`.

- [Tutorial changes](#)
- [Demo notes app source](#)

## v7.3: Upgrading to SST v2

Mar 10, 2023: Updating the guide and code snippets to SST v2.

- [Tutorial changes](#)
- [Demo notes app source](#)

## v7.2: Upgrading to SST v1

May 24, 2022: Updating the guide and code snippets to SST v1.

- [Tutorial changes](#)
- [Demo notes app source](#)

## v7.1: Upgrading to CDK v2

Feb 2, 2022: Updating SST version for CDK v2 upgrade.

- [Tutorial changes](#)
- [Demo notes app source](#)

## v7.0.3: Renaming lib to stacks

Sep 24, 2021: Renaming lib to stacks in the SST app.

- [Tutorial changes](#)
- [Demo notes app source](#)

## v7.0: Creating separate SST and Serverless Framework sections

Aug 25, 2021: Creating a separate SST version and Serverless Framework version of the guide.

- Tutorial changes
- Demo notes app source
- Serverless Framework version
  - API
  - Client

### v6.1: Adding Extra Credit AppSync and Auth chapters

Aug 3, 2021: Adding Extra Credit chapters on AppSync and Auth.

- Tutorial changes

### v6.0: Upgrading Bootstrap and reorganizing chapters

Nov 11, 2020: Upgrading to Bootstrap 4 and React Bootstrap 1.4. Also, a major reorganizing of the chapters.

- Tutorial changes
- API
- Client

### v5.0.2: Fixing encoding issue in eBook

Oct 23, 2020: Generating new eBook version to fix encoding issues.

### v5.0.1: Updating to new eBook format

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 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 SST

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 [SST Newsletter](#). The newsletter is a:

- Short plain text email
- Outlines the recent updates to SST
- Never sent out more than once a month
- One click unsubscribe
- And you get the entire guide as a 300 page ebook

You can also follow us on Twitter.



### Help and discussion

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