

Serverless Stack

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

February 5, 2018

Table of Contents

Introduction

- Who is this guide for?
- What does this guide cover?
- How to get help?
- What is serverless?
- Why create serverless apps?

Set up your AWS account

- Create an AWS account
- Create an IAM user
 - What is IAM
 - What is an ARN
- Configure the AWS CLI

Setting up the Serverless Backend

- Create a DynamoDB table
- Create an S3 bucket for file uploads
- Create a Cognito user pool
 - Create a Cognito test user
- Set up the Serverless Framework
 - Add support for ES6/ES7 JavaScript

Building a Serverless REST API

- Add a create note API
- Add a get note API
- Add a list all the notes API
- Add an update note API
- Add a delete note API

Deploying the Backend

- Deploy the APIs
- Create a Cognito identity pool
 - Cognito user pool vs identity pool
- Test the APIs

Setting up a React App

- Create a new React.js app
 - Add app favicons
 - Set up custom fonts
 - Set up Bootstrap
- Handle routes with React Router
 - Create containers
 - Adding links in the navbar
 - Handle 404s

Building a React App

- Create a login page
 - Login with AWS Cognito
 - Add the session to the state
 - Load the state from the session
 - Clear the session on logout
 - Redirect on login and logout
 - Give feedback while logging in
- Create a signup page
 - Create the signup form
 - Signup with AWS Cognito
- Add the create note page
 - Connect to API Gateway with IAM auth
 - Call the create API
 - Upload a file to S3
 - Clear AWS Credentials Cache
- List all the notes
 - Call the list API
- Display a note

- Render the note form
- Save changes to a note
- Delete a note
- Set up secure pages
 - Create a route that redirects
 - Use the redirect routes
 - Redirect on login

Deploying a React app on AWS

- Deploy the Frontend
 - Create an S3 bucket
 - Deploy to S3
 - Create a CloudFront distribution
 - Set up your domain with CloudFront
 - Set up www domain redirect
 - Set up SSL
- Deploy updates
 - Update the app
 - Deploy again

Conclusion

- Wrapping up
- Giving back
- Changelog
- Staying up to date

Extra Credit

Backend

- API Gateway and Lambda Logs
- Debugging Serverless API Issues
- Serverless environment variables
- Stages in Serverless Framework
- Configure multiple AWS profiles

- Customize the Serverless IAM Policy

Frontend

- Code Splitting in Create React App
- Environments in Create React App

Tools

- Serverless Node.js Starter

Who Is This Guide For?

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

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

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

On a personal note, the serverless approach has been a giant revelation for us and we wanted to create a resource where we could share what we've learnt. You can read more about us [here](#) (/about.html).

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

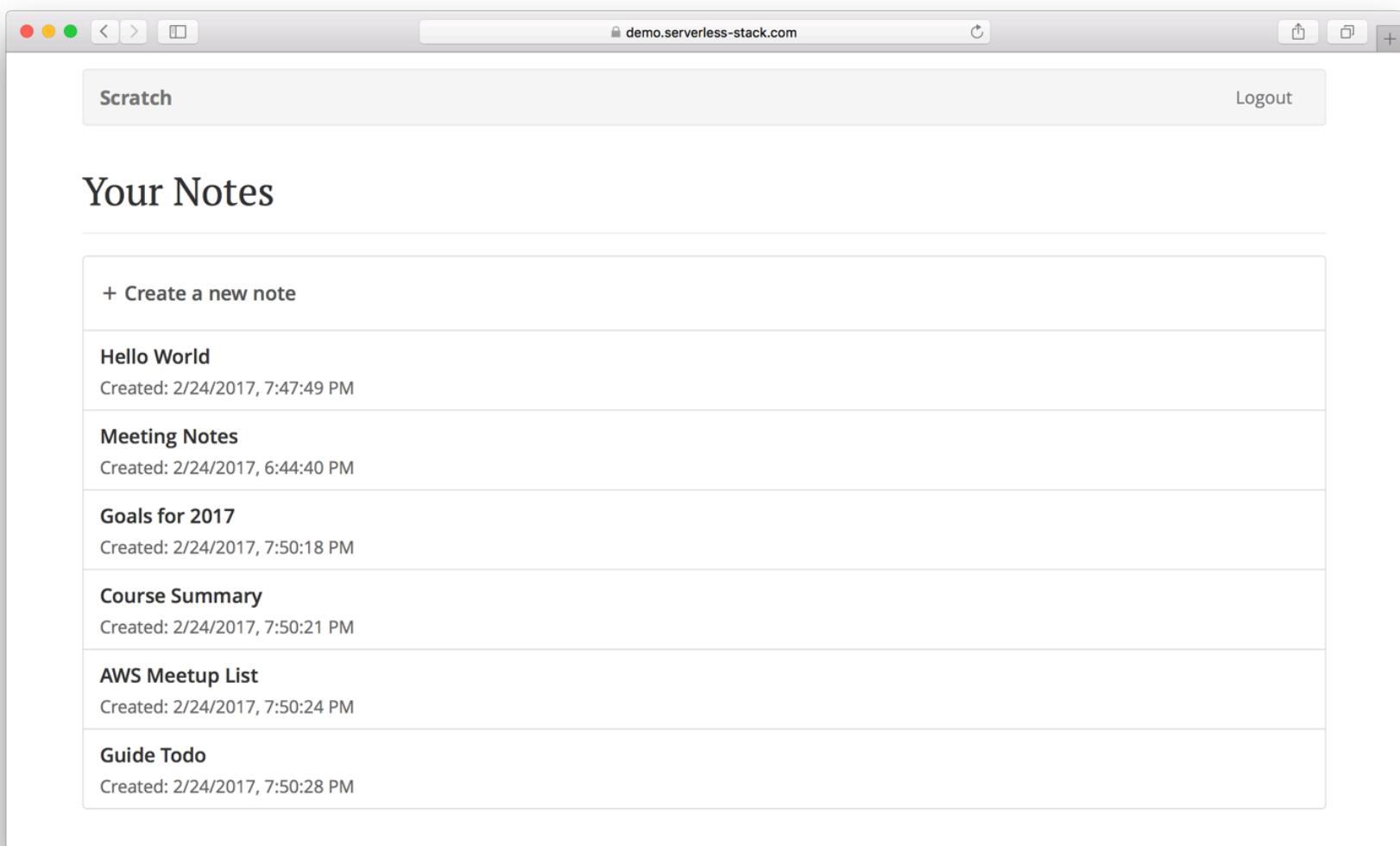
For help and discussion

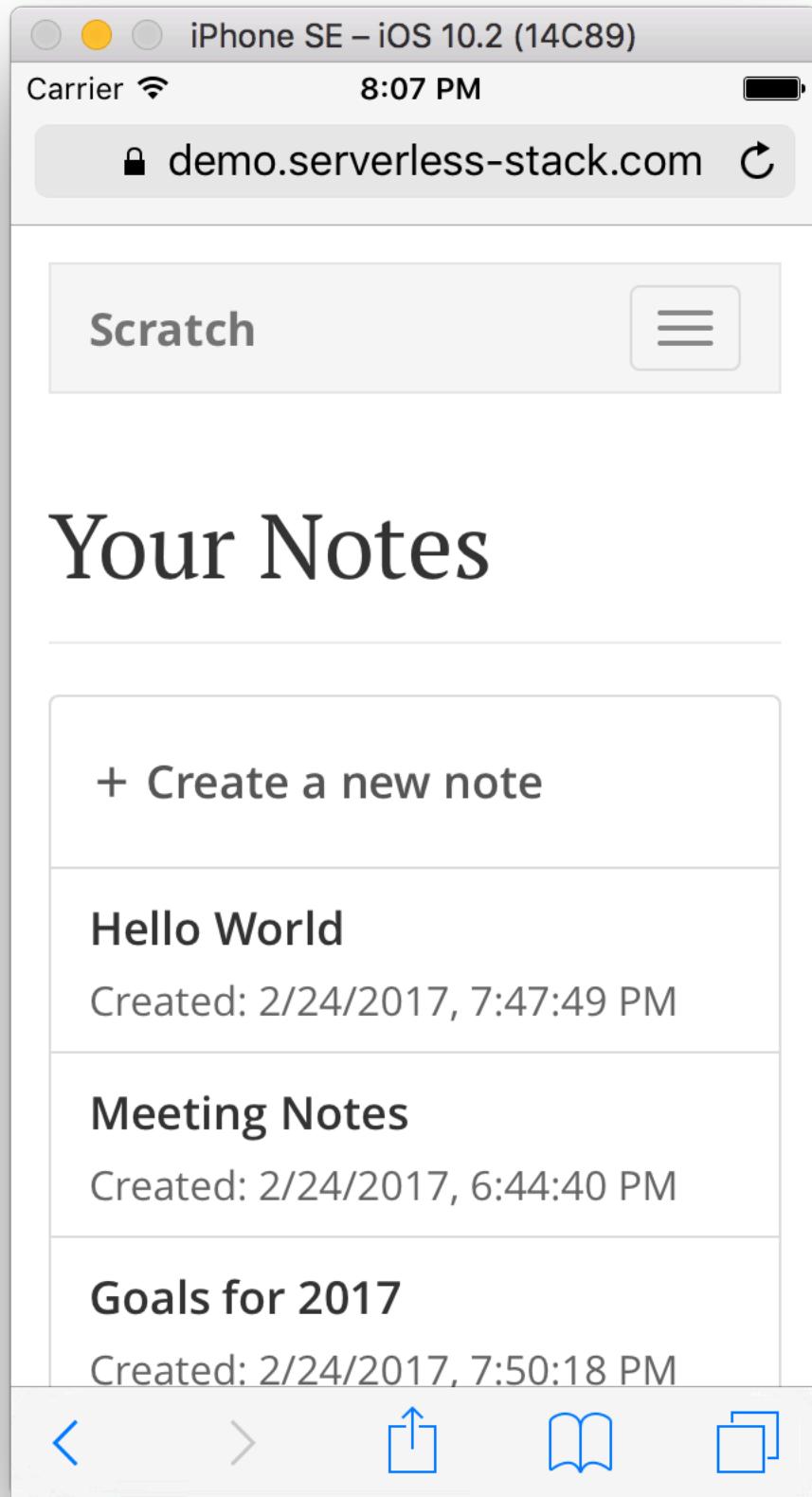
 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/8>)

What Does This Guide Cover?

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





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

- Should allow users to signup and login to their accounts
- Users should be able to create notes with some content

- Each note can also have an uploaded file as an attachment
- Allow users to modify their note and the attachment
- Users can also delete their notes
- App should be served over HTTPS on a custom domain
- The backend APIs need to be secure
- The app needs to be responsive

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

- Lambda (<https://aws.amazon.com/lambda/>) & API Gateway (<https://aws.amazon.com/api-gateway/>) for our serverless API
- DynamoDB (<https://aws.amazon.com/dynamodb/>) for our database
- Cognito (<https://aws.amazon.com/cognito/>) for user authentication and securing our APIs
- S3 (<https://aws.amazon.com/s3/>) for hosting our app and file uploads
- CloudFront (<https://aws.amazon.com/cloudfront/>) for serving out our app
- Route 53 (<https://aws.amazon.com/route53/>) for our domain
- Certificate Manager (<https://aws.amazon.com/certificate-manager>) for SSL
- React.js (<https://facebook.github.io/react/>) for our single page app
- React Router (<https://github.com/ReactTraining/react-router>) for routing
- Bootstrap (<http://getbootstrap.com>) for the UI Kit

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

The guide covers the following concepts in order.

For the backend:

- Configure your AWS account
- Create your database using DynamoDB
- Set up S3 for file uploads
- Set up Cognito User Pools to manage user accounts
- Set up Cognito Identity Pool to secure our file uploads
- Set up the Serverless Framework to work with Lambda & API Gateway
- Write the various backend APIs

For the frontend:

- Set up our project with Create React App
- Add favicons, fonts, and a UI Kit using Bootstrap
- Set up routes using React-Router
- Use AWS Cognito SDK to login and signup users
- Plugin to the backend APIs to manage our notes
- Use the AWS JS SDK to upload files
- Create an S3 bucket to upload our app
- Configure CloudFront to serve out our app
- Point our domain with Route 53 to CloudFront
- Set up SSL to serve our app over HTTPS

We think this will give you a good foundation on building full-stack serverless applications. If there are any other concepts or technologies you'd like us to cover, feel free to let us know via email (<mailto:contact@anoma.ly>).

For help and discussion

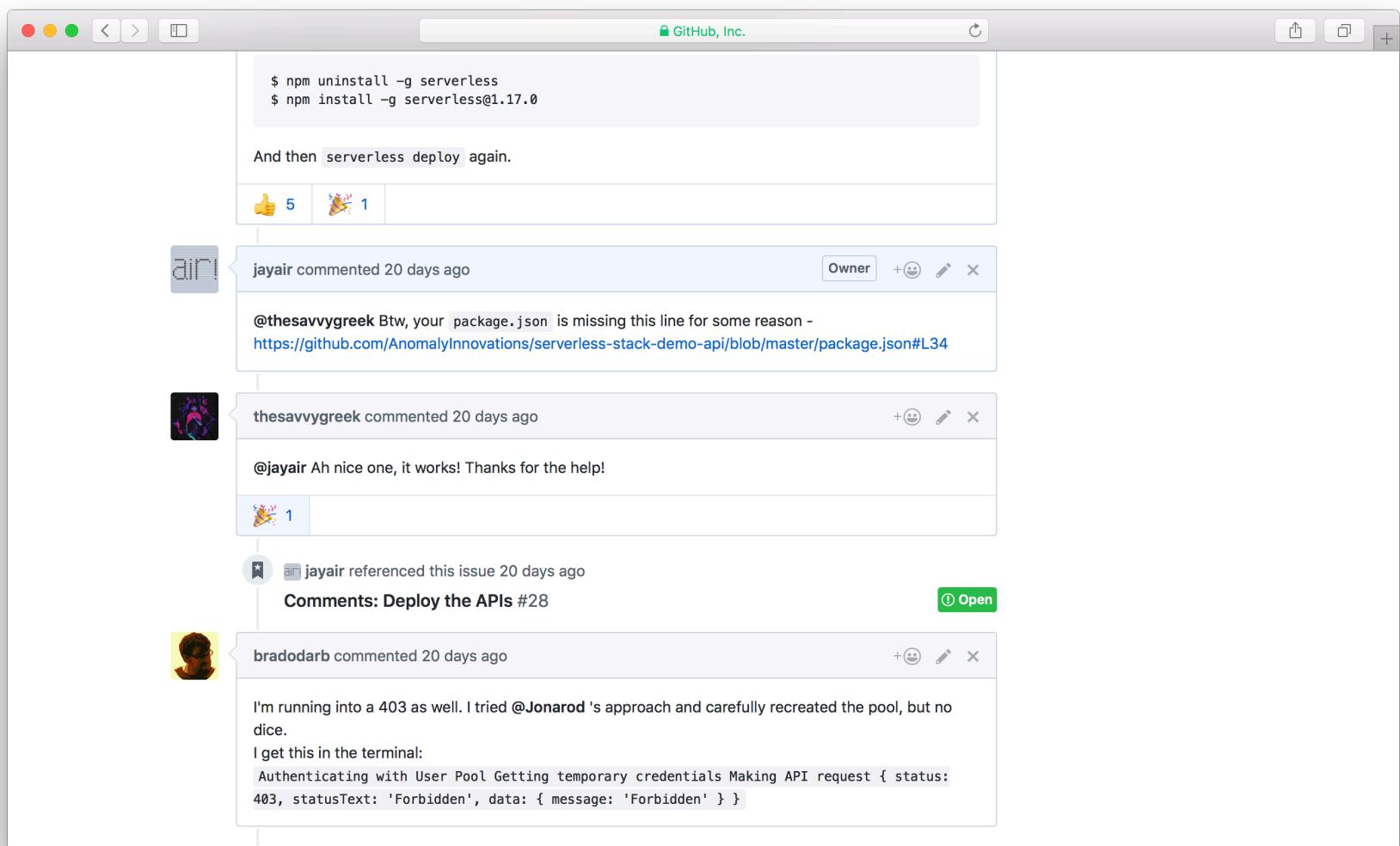
 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/9>)

How to Get Help?

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

- We use GitHub issues (<https://github.com/AnomalyInnovations/serverless-stack-com/issues?q=is%3Aissue+is%3Aopen+label%3ADiscussion+sort%3Aupdated-desc>) as our comments and we've helped resolve quite a few issues in the past. So make sure to check the comments under each chapter to see if somebody else has run into the same issue as you have.
- Post in the comments detailing your issue and one of us will respond.
- Or send us an email (<mailto:contact@anoma.ly>) directly.



Also, this entire guide is hosted on GitHub (<https://github.com/AnomalyInnovations/serverless-stack-com>). So you can always:

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

For help and discussion

💬 **Comments on this chapter**

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/10>)

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 organisations 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.

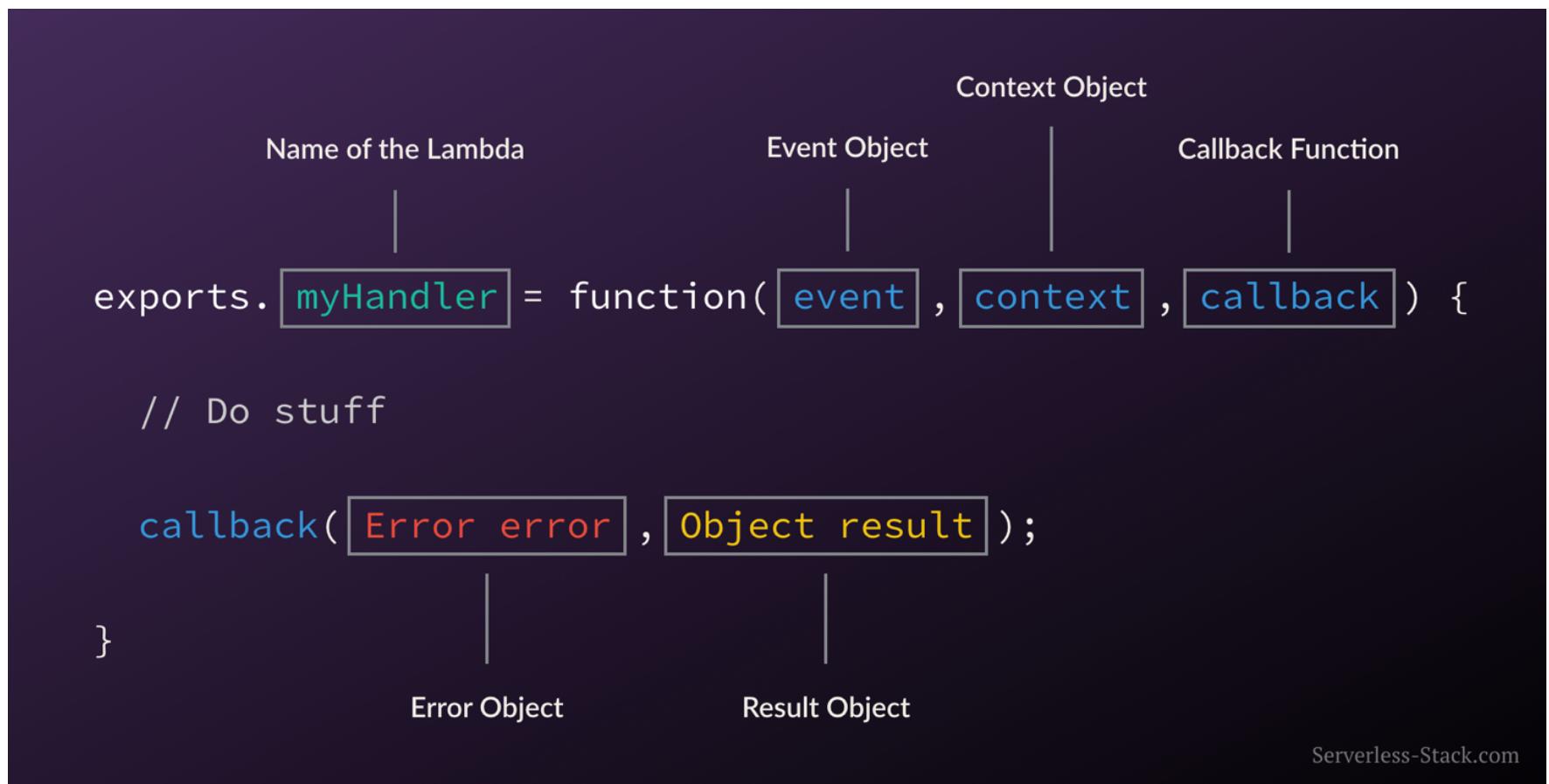
As developers we've been looking for a solution to these problems and this is where serverless comes in. Serverless allows us to build applications where we simply hand the cloud provider (AWS, Azure, or Google Cloud) our code and it runs it for us. It also allocates the appropriate amount of resources to respond to the usage. On our end we only get charged for the time it took our code to execute and the resources it consumed. If we are undergoing a spike of usage, the cloud provider simply creates more instances of our code to respond to the requests. Additionally, our code runs in a secured environment where the cloud provider takes care of keeping the server up to date and secure.

AWS Lambda

In serverless applications we are not responsible for handling the requests that come in to our

server. Instead the cloud provider handles the requests and sends us an object that contains the relevant info and asks us how we want to respond to it. The request is treated as an event and our code is simply a function that takes this as the input. As a result we are writing functions that are meant to respond to these events. So when a user makes a request, the cloud provider creates a container and runs our function inside it. If there are two concurrent requests, then two separate containers are created to respond to the requests.

In the AWS world the serverless function is called AWS Lambda (<https://aws.amazon.com/lambda/>) and our serverless backend is simply a collection of Lambdas. Here is what a Lambda function looks like.



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

While this example is in JavaScript (or Node.js), AWS Lambda supports Python, Java, and C# as well. Lambda functions are charged for every 100ms that it uses and as mentioned above they automatically scale to respond to the usage. The Lambda runtime also comes with 512MB of ephemeral disk space and up to 3008MB of memory.

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/101>)

Why Create Serverless Apps?

It is important to address why it is worth learning how to create serverless apps. There are a couple of 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. And 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
	\$0.50 per hosted zone + \$0.40/M queries	

Route53		\$0.50
Certificate Manager	Free	\$0.00
Total		\$6.10

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

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

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

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

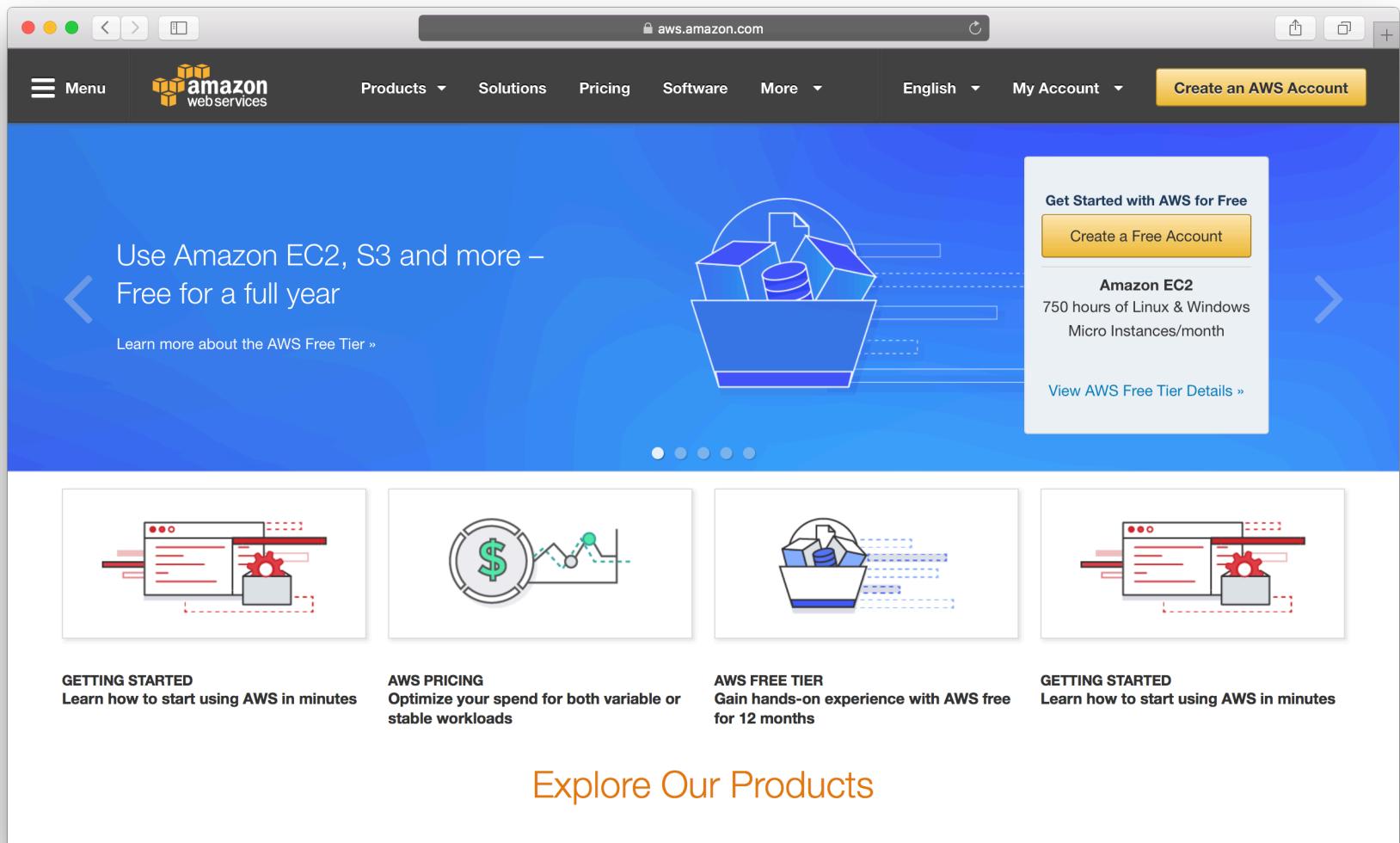
For help and discussion

💬 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/11>)

Create an AWS Account

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



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

For help and discussion

💬 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/12>)

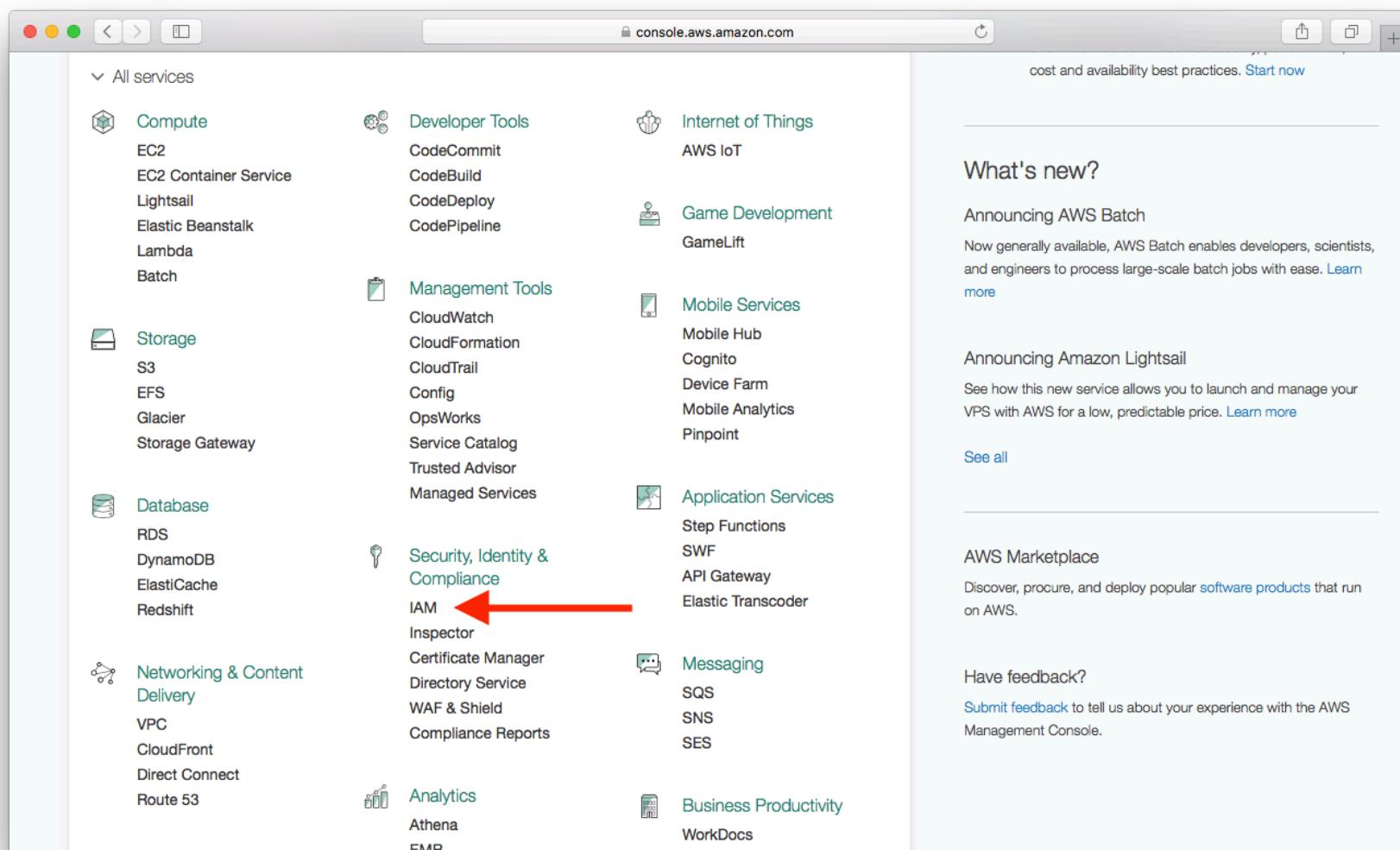
Create an IAM User

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

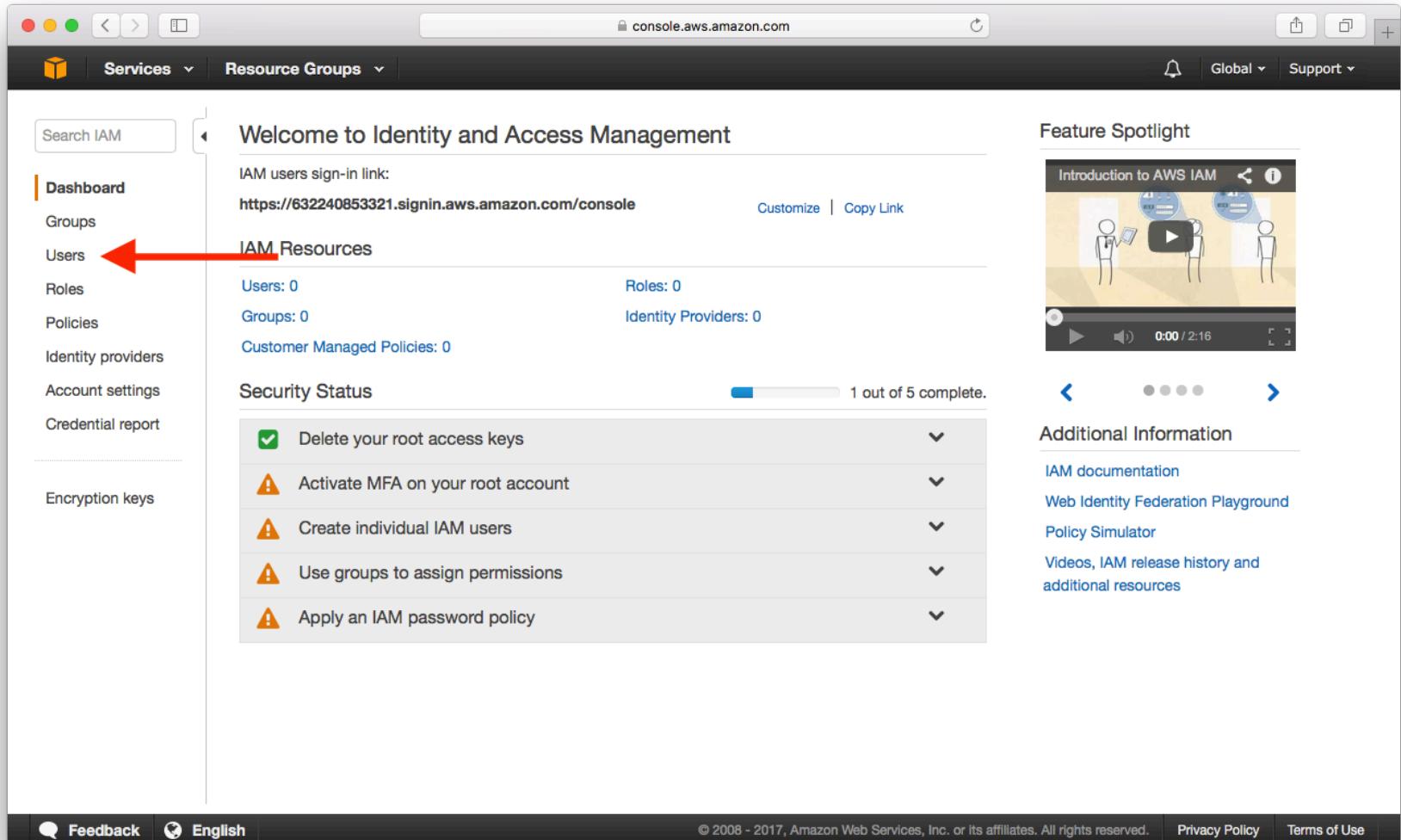
In this chapter, we are going to create a new IAM user for a couple of the AWS related tools we are going to be using later.

Create User

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



Select Users.



The screenshot shows the AWS Identity and Access Management (IAM) service dashboard. On the left, a sidebar menu lists various IAM management options: Dashboard, Groups, Users (which is highlighted with a red arrow), Roles, Policies, Identity providers, Account settings, Credential report, and Encryption keys. The main content area is titled "Welcome to Identity and Access Management". It displays an "IAM users sign-in link" and a "Customize | Copy Link" button. Below this, there's a summary of IAM resources: Users: 0, Roles: 0, Groups: 0, and Identity Providers: 0. A "Customer Managed Policies: 0" link is also present. A "Security Status" section indicates "1 out of 5 complete." and lists five items with dropdown arrows: "Delete your root access keys" (checked), "Activate MFA on your root account", "Create individual IAM users", "Use groups to assign permissions", and "Apply an IAM password policy". To the right, a "Feature Spotlight" box contains a video thumbnail titled "Introduction to AWS IAM" with a play button and a progress bar showing 0:00 / 2:16. Below the spotlight, there's an "Additional Information" section with links to "IAM documentation", "Web Identity Federation Playground", "Policy Simulator", and "Videos, IAM release history and additional resources". At the bottom of the page, there are links for "Feedback", "English", "© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.", "Privacy Policy", and "Terms of Use".

Select Add User.

A screenshot of the AWS IAM service in the AWS Management Console. The left sidebar shows navigation options like Dashboard, Groups, Roles, Policies, Identity providers, Account settings, Credential report, and Encryption keys. The main area is titled 'Resource Groups' and shows the 'Users' section selected. At the top, there are buttons for 'Add user' (highlighted with a red arrow) and 'Delete users'. Below is a search bar with placeholder text 'Find users by username or access key' and a 'Showing 0 results' message. A table header row includes columns for 'User name', 'Groups', 'Password', 'Last sign-in', 'Access keys', and 'Creation time'. The bottom of the page includes standard AWS footer links for Feedback, English, Copyright notice (© 2008 - 2017), Privacy Policy, and Terms of Use.

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

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

Screenshot of the AWS IAM 'Add user' wizard, Step 1: Set user details.

User name*: admin

Access type*: Programmatic access (highlighted with a red arrow)

* Required

Next: Permissions

Select Attach existing policies directly.

Screenshot of the AWS IAM 'Add user' wizard, Step 2: Set permissions for admin.

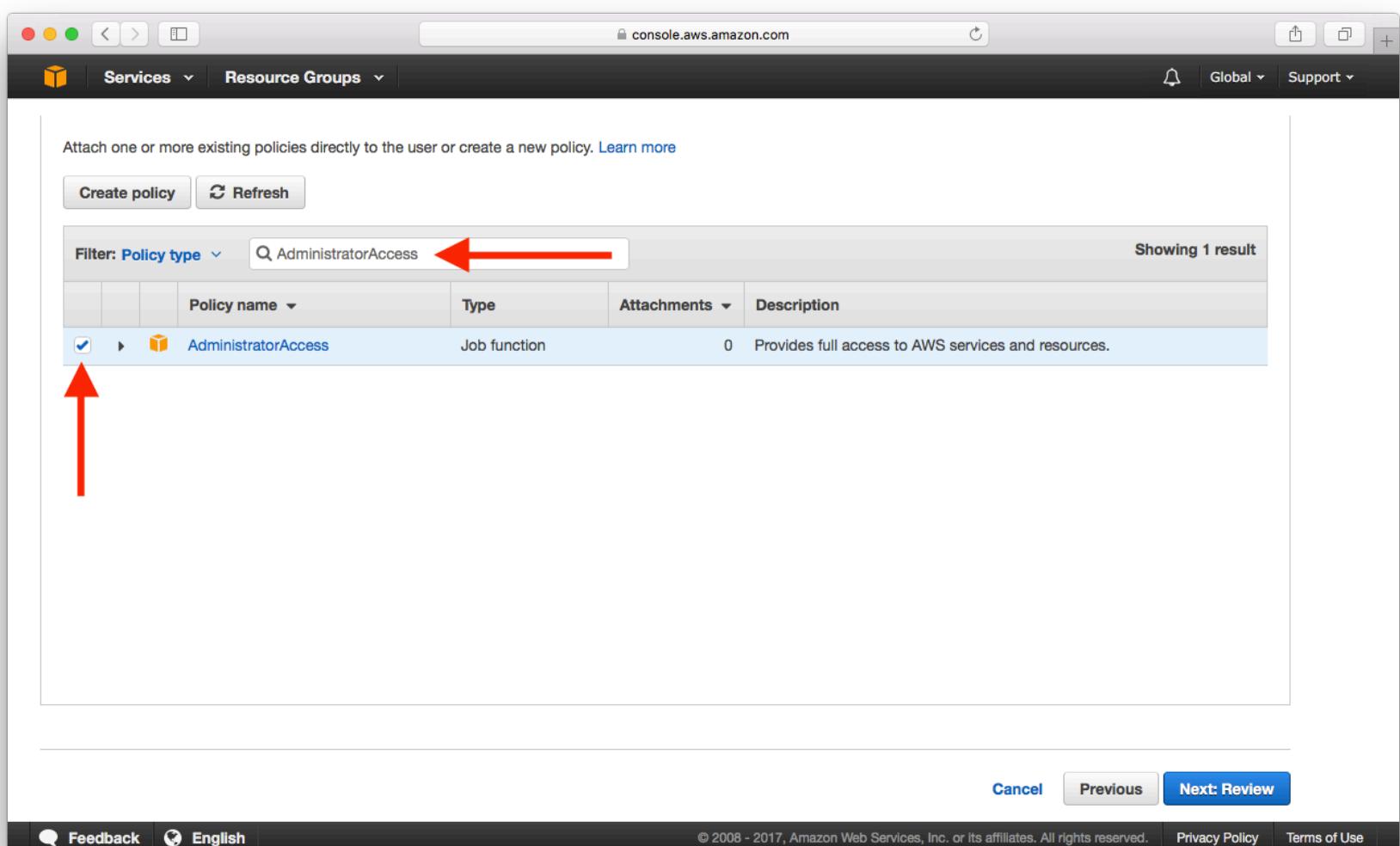
Attach existing policies directly (highlighted with a red arrow)

Attach one or more existing policies directly to the user or create a new policy. Learn more

Policy name	Type	Attachments	Description
AWSDirectConnectReadOnlyAccess	AWS managed	0	Provides read only access to AWS Direct Connect via the AWS Management Console.
AmazonGlacierReadOnlyAccess	AWS managed	0	Provides read only access to Amazon Glacier via the AWS Management Console.

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

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



The screenshot shows the AWS IAM Policies search interface. A red arrow points to the search bar at the top right, which contains the text "AdministratorAccess". Another red arrow points to the first row in the results table, which displays the "AdministratorAccess" policy details. The table has columns for Policy name, Type, Attachments, and Description. The "AdministratorAccess" policy is listed as a Job function with 0 attachments, providing full access to AWS services and resources.

Policy name	Type	Attachments	Description
AdministratorAccess	Job function	0	Provides full access to AWS services and resources.

Select **Create user**.

Add user

Review

User details

User name	admin
AWS access type	Programmatic access - with an access key

Permissions summary

The following policies will be attached to the user shown above.

Type	Name
Managed policy	AdministratorAccess

Cancel Previous **Create user**

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

Add user

Success

You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: <https://632240853321.signin.aws.amazon.com/console>

Download .csv

	User	Access key ID	Secret access key
▶ <input checked="" type="checkbox"/>	admin	AKIAJX32LZLTWK47WQA	***** Show 

Close

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

The screenshot shows the AWS Management Console with the URL `console.aws.amazon.com` in the address bar. The top navigation bar includes 'Services', 'Resource Groups', 'Global', and 'Support'. A progress bar at the top right indicates four steps: 'Details' (step 1), 'Permissions' (step 2), 'Review' (step 3), and 'Complete' (step 4, highlighted in blue). Below the progress bar, a green box displays a 'Success' message: 'You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.' It also provides a link for users with AWS Management Console access to sign in. A 'Download .csv' button is available. A table lists the newly created user 'admin' with columns for 'User', 'Access key ID', and 'Secret access key'. The 'Access key ID' is AKIAJX332LZLTWK47WQA and the 'Secret access key' is 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. Red arrows point upwards from the text 'For help and discussion' to the 'Access key ID' and 'Secret access key' fields in the table.

User	Access key ID	Secret access key
admin	AKIAJX332LZLTWK47WQA	xPdbuOEs5y1vC43KyTyniX8k+cVsgEdh7unl 07c4 Hide

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

For help and discussion

Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/13>)

What is IAM

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

Let's start with the official definition of IAM.

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

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

What is an IAM User

When you first create an AWS account, you are the root user. The email address and password you used to create the account is called your root account credentials. You can use them to sign in to the AWS Management Console. When you do, you have complete, unrestricted access to all resources in your AWS account, including access to your billing information and the ability to change your password.



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

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



Serverless-Stack.com

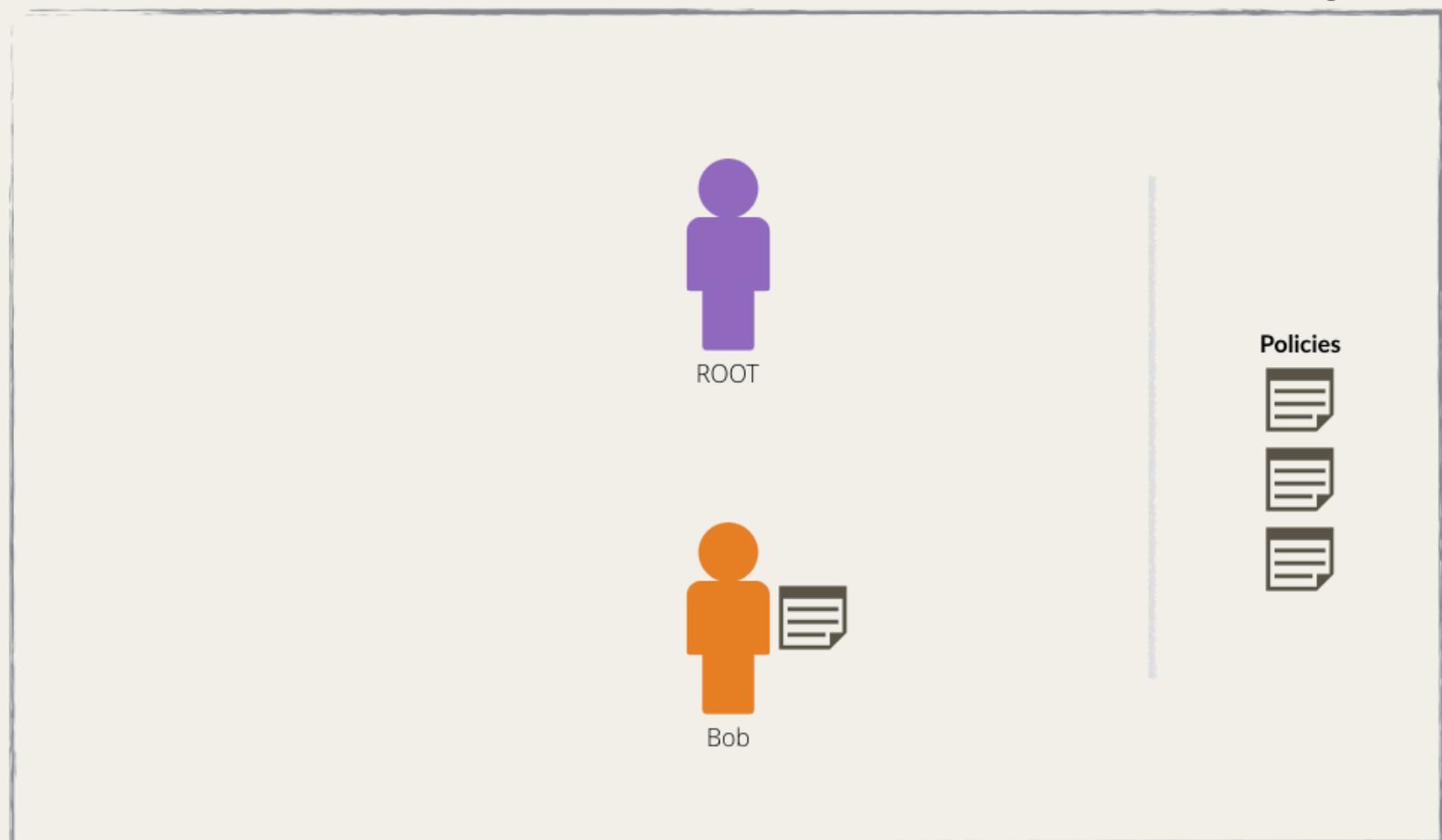
By default, users can't access anything in your account. You grant permissions to a user by creating a policy and attaching the policy to the user. You can grant one or more of these policies to restrict what the user can and cannot access.

What is an IAM Policy?

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

Policies can be granted in a number of ways:

- Attaching a *managed policy*. AWS provides a list of pre-defined policies such as *AmazonS3ReadOnlyAccess*.
- Attaching an *inline policy*. An inline policy is a custom policy created by hand.
- Adding the user to a group that has appropriate permission policies attached. We'll look at groups in detail below.
- Cloning the permission of an existing IAM user.



Serverless-Stack.com

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

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

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

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

```
"Condition": {"StringEquals": {"s3:prefix": "Bobs-"}}}
```

```
}
```

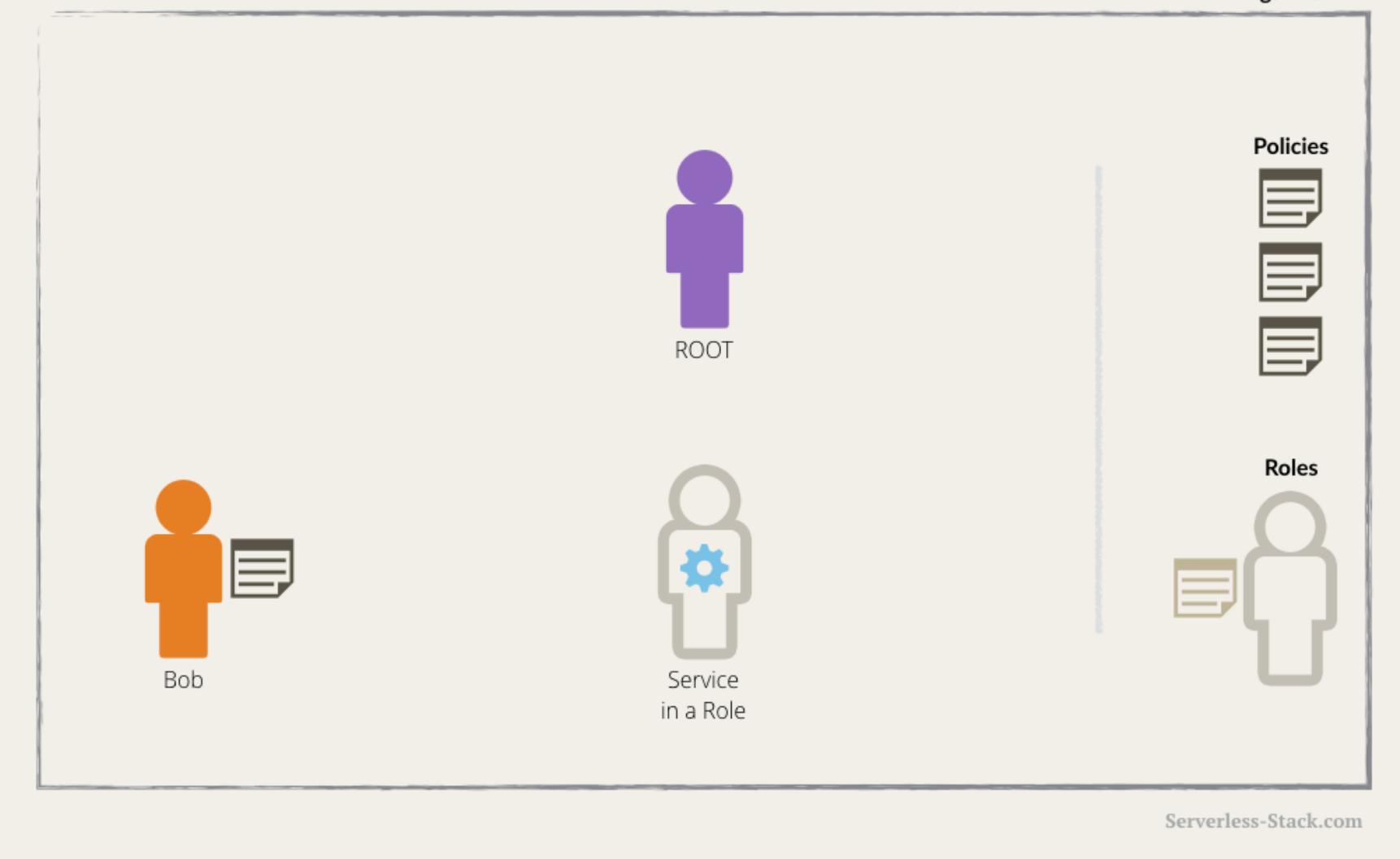
We are using S3 resources in the above examples. But a policy looks similar for any of the AWS services. It just depends on the resource ARN for **Resource** property. An ARN is an identifier for a resource in AWS and we'll look at it in more detail in the next chapter. We also add the corresponding service actions and condition context keys in **Action** and **Condition** property. You can find all the available AWS Service actions and condition context keys for use in IAM Policies here

(https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_actionsconditions.html). Aside from attaching a policy to a user, you can attach them to a role or a group.

What is an IAM Role

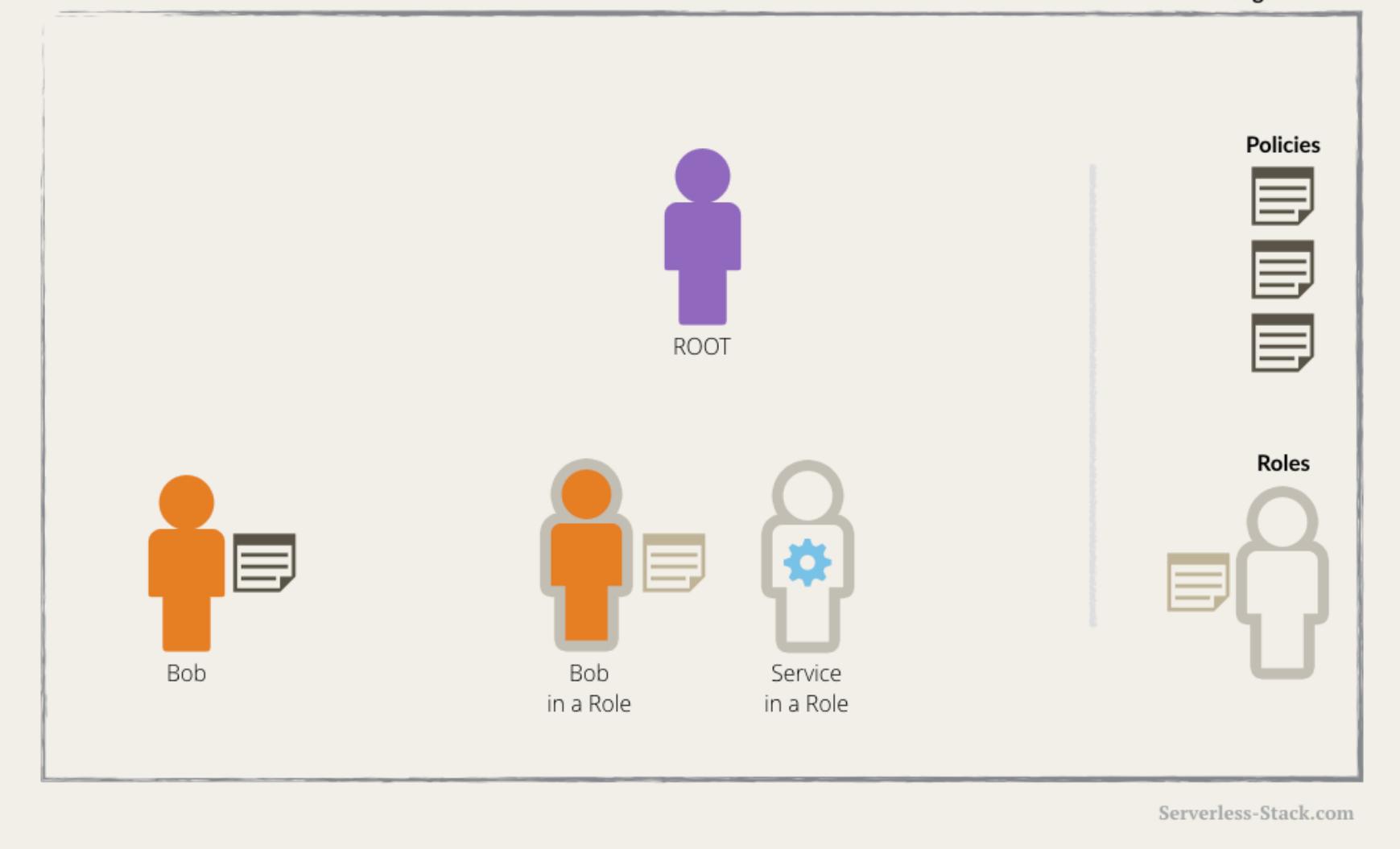
Sometimes your AWS resources need to access other resources in your account. For example, you have a Lambda function that queries your DynamoDB to retrieve some data, process it, and then send Bob an email with the results. In this case, we want Lambda to only be able to make read queries so it does not change the database by mistake. We also want to restrict Lambda to be able to email Bob so it does not spam other people. This can be done by creating an IAM user and putting the user's credentials to the Lambda function or embed the credentials in the Lambda code. But this is just not secure. If somebody was to get hold of these credentials, they could make those calls on your behalf. This is where IAM role comes in to play.

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

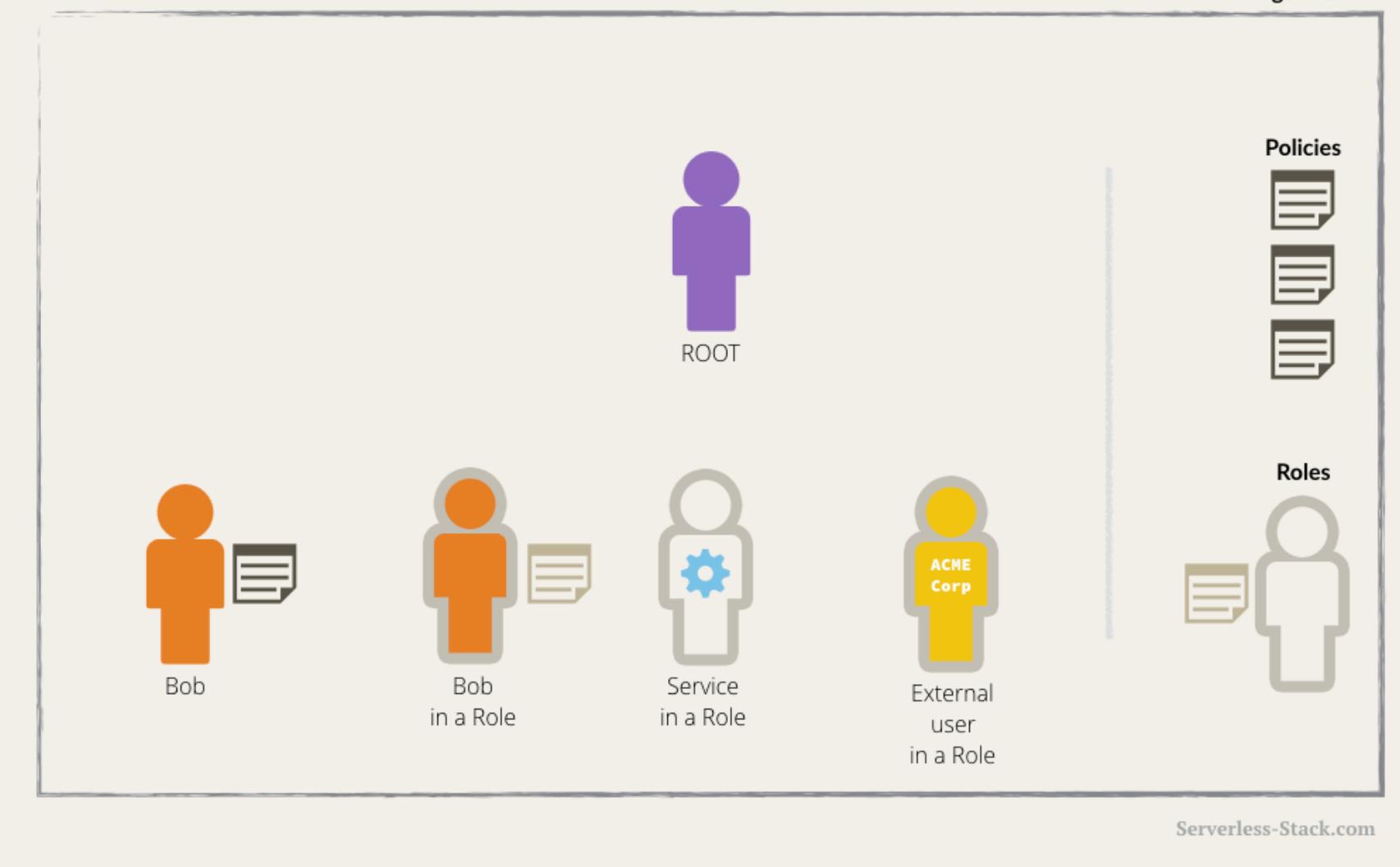


Serverless-Stack.com

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

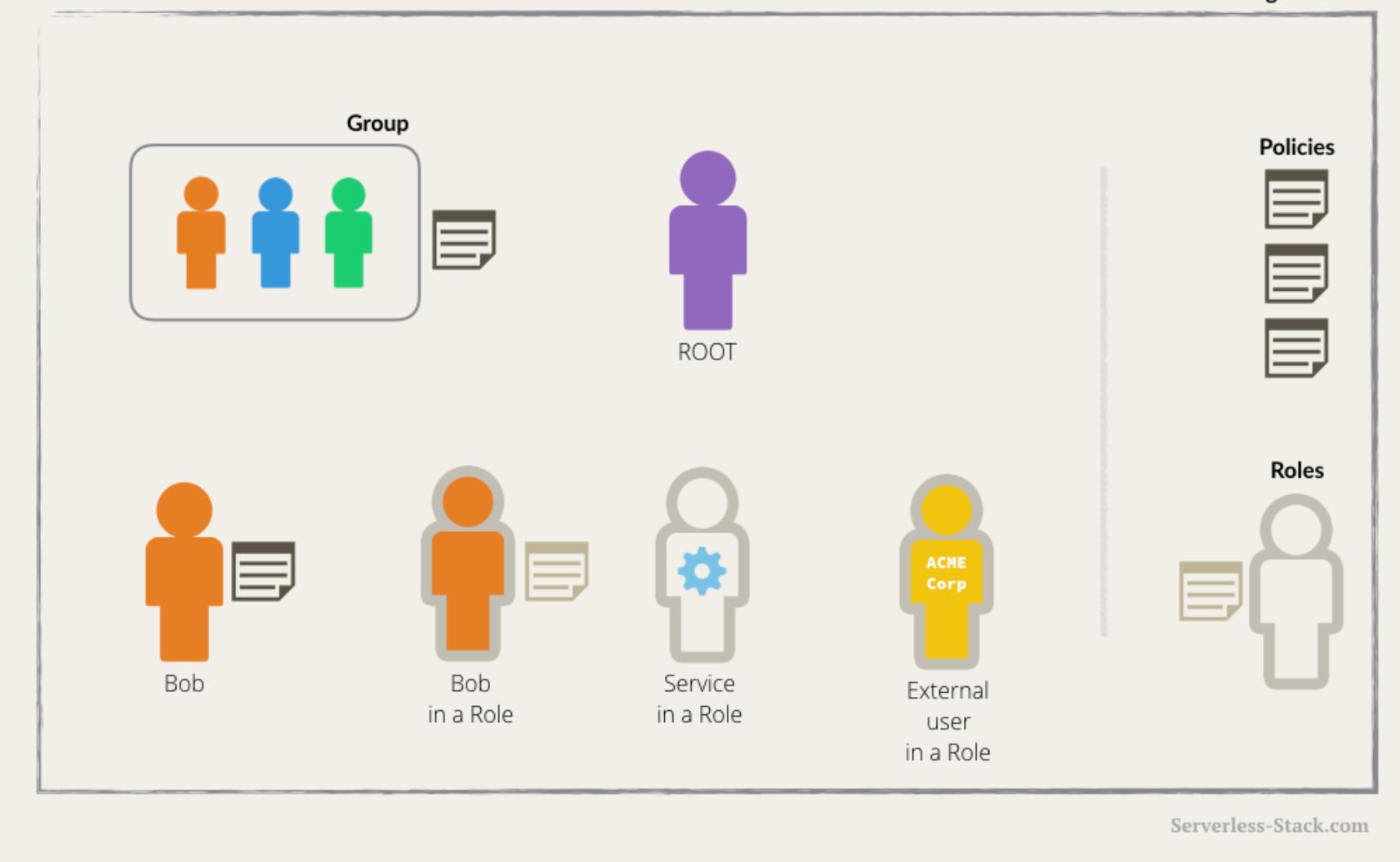


You can also have a role tied to the ARN of a user from a different organization. This allows the external user to assume that role as a part of your organization. This is typically used when you have a third party service that is acting on your AWS Organization. You'll be asked to create a **Cross-Account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



What is an IAM Group

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



Serverless-Stack.com

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/85>)

What is an ARN

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

Here is the official definition:

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

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

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

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

```
<!-- Elastic Beanstalk application version -->  
arn:aws:elasticbeanstalk:us-east-1:123456789012:environment/My  
App/MyEnvironment  
  
<!-- IAM user name -->  
arn:aws:iam::123456789012:user/David  
  
<!-- Amazon RDS instance used for tagging -->  
arn:aws:rds:eu-west-1:123456789012:db:mysql-db  
  
<!-- Object in an Amazon S3 bucket -->  
arn:aws:s3::::my_corporate_bucket/exampleobject.png
```

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

1. Communication

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

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

2. IAM Policy

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

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

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

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/86>)

Configure the AWS CLI

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

Install the AWS CLI

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

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

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

```
$ sudo pip install awscli
```

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

Add Your Access Key to AWS CLI

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

It should look something like this:

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

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

```
$ aws configure
```

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

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/14>)

Create a DynamoDB Table

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

About DynamoDB

Amazon DynamoDB is a fully managed NoSQL database that provides fast and predictable performance with seamless scalability. Similar to other databases, DynamoDB stores data in tables. Each table contains multiple items, and each item is composed of one or more attributes.

Create Table

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

cost and availability best practices. [Start now](#)

What's new?

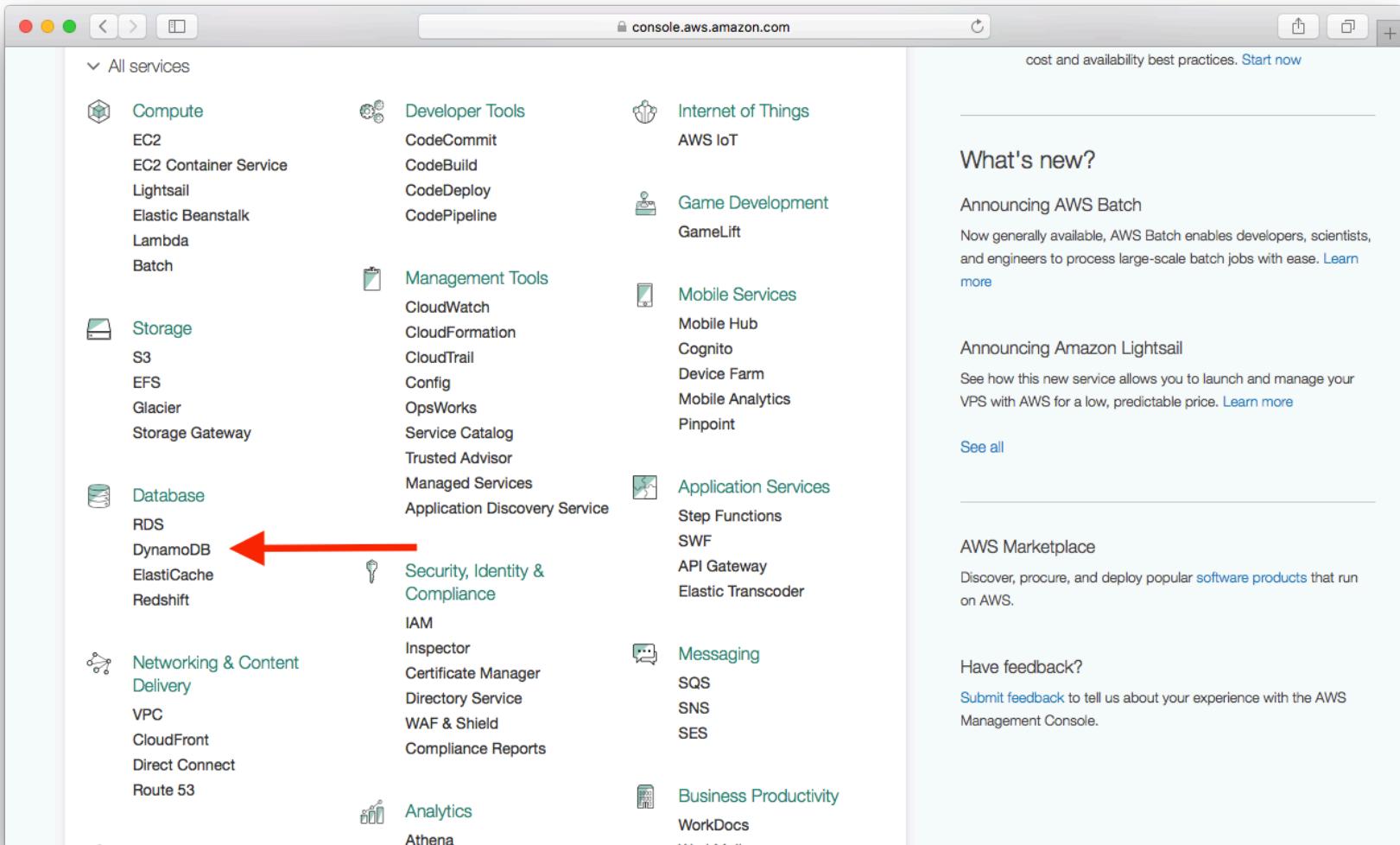
Announcing AWS Batch
Now generally available, AWS Batch enables developers, scientists, and engineers to process large-scale batch jobs with ease. [Learn more](#)

Announcing Amazon Lightsail
See how this new service allows you to launch and manage your VPS with AWS for a low, predictable price. [Learn more](#)

[See all](#)

AWS Marketplace
Discover, procure, and deploy popular [software products](#) that run on AWS.

Have feedback?
[Submit feedback](#) to tell us about your experience with the AWS Management Console.



Select **Create table**.

console.aws.amazon.com

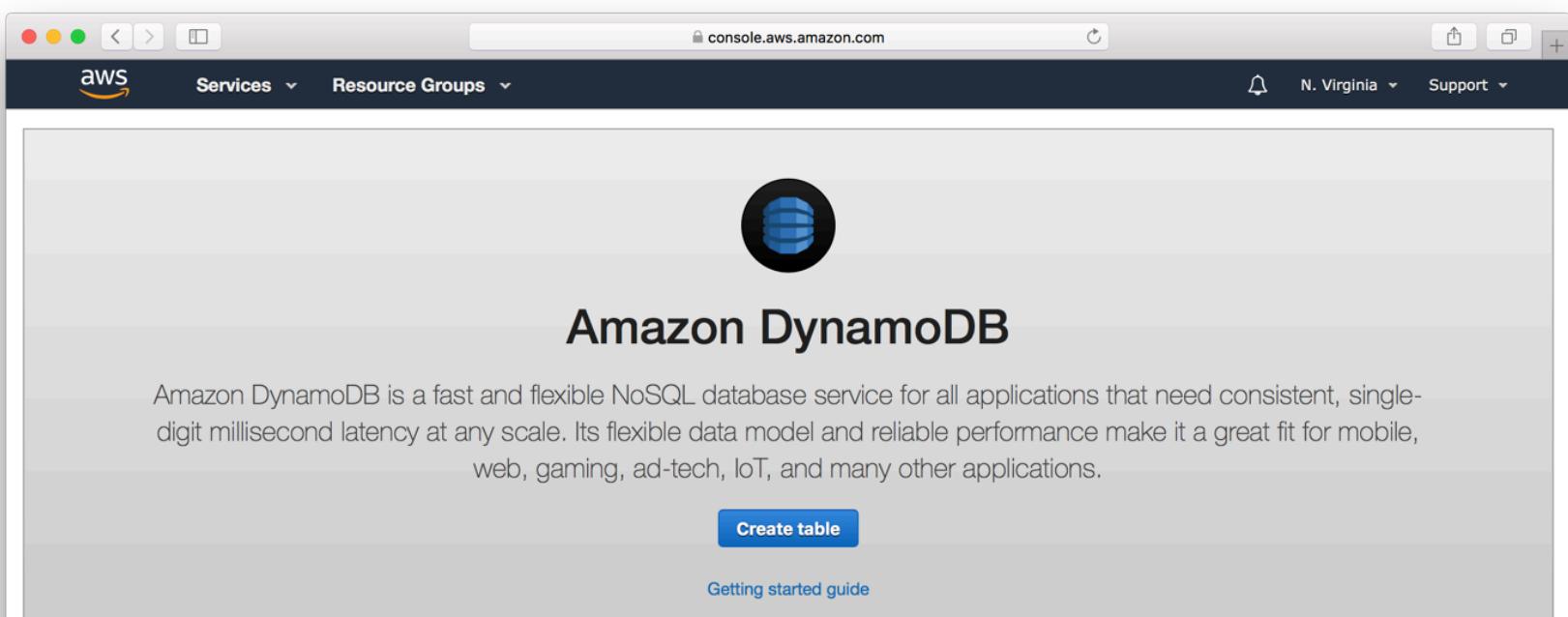
AWS Services Resource Groups N. Virginia Support

Amazon DynamoDB

Amazon DynamoDB is a fast and flexible NoSQL database service for all applications that need consistent, single-digit millisecond latency at any scale. Its flexible data model and reliable performance make it a great fit for mobile, web, gaming, ad-tech, IoT, and many other applications.

[Create table](#)

[Getting started guide](#)




Create tables



Add and query items



Monitor and manage tables

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

DynamoDB is a schema-less database that only requires a table name and primary key. The table's primary key is made up of one or two attributes that uniquely identify items, partition the data, and sort data within each partition.

Table name* notes

Primary key* Partition key

userId String Add sort key ↗
noteId String ↗

Table settings

Default settings provide the fastest way to get started with your table. You can modify these default settings now or after your table has been created.

Use default settings

- No secondary indexes.
- Provisioned capacity set to 5 reads and 5 writes.
- Basic alarms with 80% upper threshold using SNS topic "dynamodb".

Info You do not have the required role to enable Auto Scaling by default.
Please refer to documentation.

Additional charges may apply if you exceed the AWS Free Tier levels for CloudWatch or Simple Notification Service. Advanced alarm settings are available in the CloudWatch management console.

Feedback English (US) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

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

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

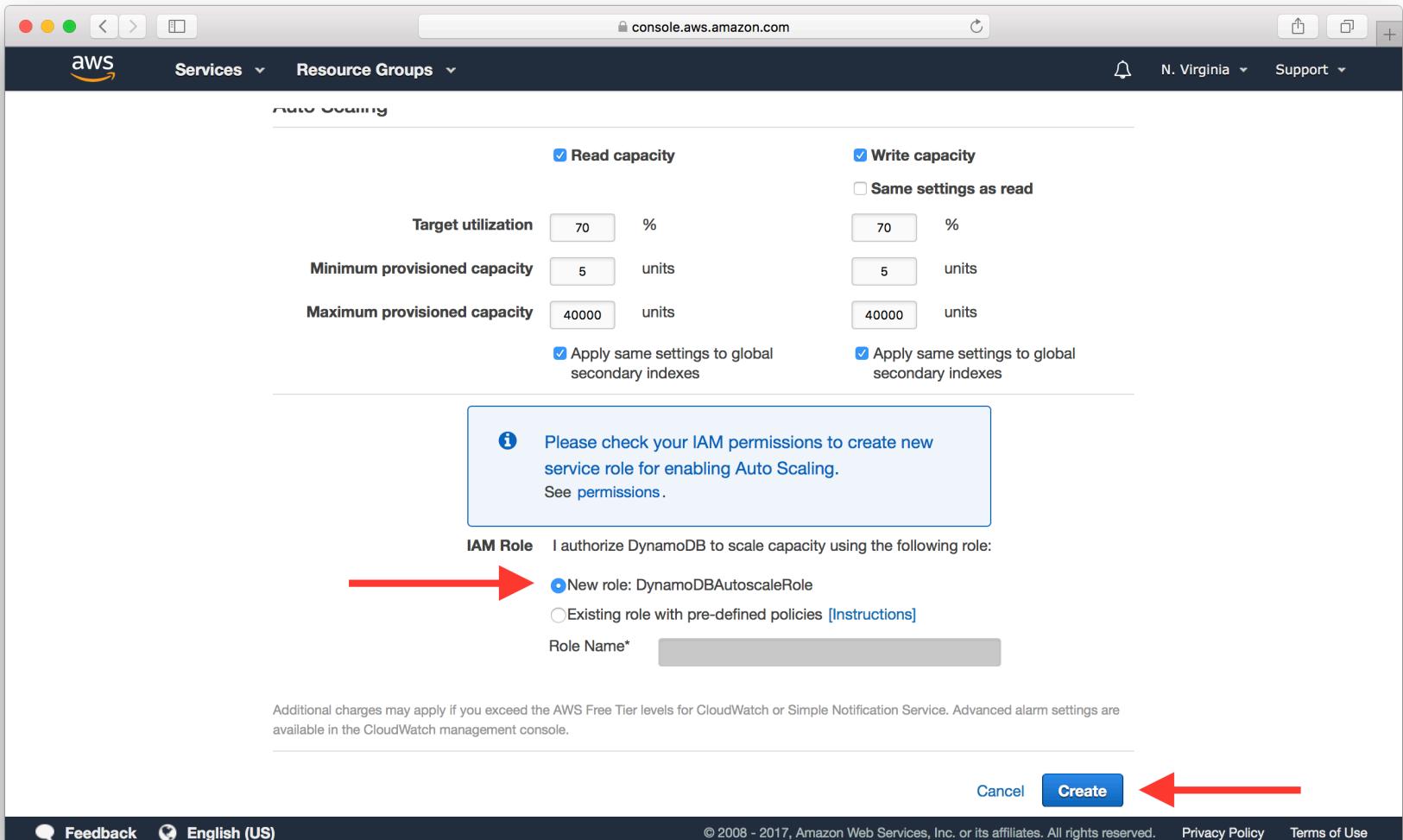
We are going to use the composite primary key which gives us additional flexibility when querying the data. For example, if you provide only the value for `userId`, DynamoDB would retrieve all of the notes by that user. Or you could provide a value for `userId` and a value for `noteId`, to retrieve a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

To get a further understanding on how indexes work in DynamoDB, you can read more here [DynamoDB Core Components](#) (<http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html>).

If you see the following message, deselect **Use default settings**.

The screenshot shows the AWS DynamoDB 'Create Table' wizard. In the 'Table settings' section, there is a checkbox labeled 'Use default settings' which is currently checked. A red arrow points to this checkbox. Below it, a note in a callout box states: 'You do not have the required role to enable Auto Scaling by default. Please refer to documentation.' Another red arrow points to this note. At the bottom right of the page are 'Cancel' and 'Create' buttons.

Scroll to the bottom, ensure that **New role: DynamoDBAutoscaleRole** is selected, and select **Create**.



Otherwise, simply ensure that **Use default settings** is checked, then select **Create**.

Note that the default setting provisions 5 reads and 5 writes. When you create a table, you specify how much provisioned throughput capacity you want to reserve for reads and writes. DynamoDB will reserve the necessary resources to meet your throughput needs while ensuring consistent, low-latency performance. One read capacity unit can read up to 8 KB per second and one write capacity unit can write up to 1 KB per second. You can change your provisioned throughput settings, increasing or decreasing capacity as needed.

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

The screenshot shows the AWS DynamoDB console. On the left, the navigation menu includes 'DynamoDB', 'Tables' (selected), 'Reserved capacity', 'DAX', 'Dashboard', 'Clusters', 'Subnet groups', 'Parameter groups', and 'Events'. The main area displays the 'notes' table. At the top right of the table view are tabs: 'Overview' (selected), 'Items', 'Metrics', 'Alarms', 'Capacity', 'Indexes', 'Triggers', 'Access control', and 'Tags'. Below these tabs is a section titled 'Recent alerts' with the message 'No CloudWatch alarms have been triggered for this table.' Under 'Stream details', it shows 'Stream enabled: No', 'View type: -', and 'Latest stream ARN: -'. A 'Manage Stream' button is available. The 'Table details' section provides the following information:

	notes
Primary partition key	userId (String)
Primary sort key	notId (String)
Time to live attribute	DISABLED Manage TTL
Table status	Active
Creation date	September 30, 2017 at 1:50:55 AM UTC+8
Provisioned read capacity units	5 (Auto Scaling Enabled)
Provisioned write capacity units	5 (Auto Scaling Enabled)
Last decrease time	-
Last increase time	-
Storage size (in bytes)	0 bytes

At the bottom of the page are links for 'Feedback', 'English (US)', 'Privacy Policy', and 'Terms of Use'.

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

For help and discussion

Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/15>)

Create an S3 Bucket for File Uploads

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

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

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

Create Bucket

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

Screenshot of the AWS Management Console homepage. A red arrow points from the text "Select Create Bucket." in the previous step to the "S3" link under the "Storage" category in the sidebar.

AWS services

Find a service by name (for example, EC2, S3, Elastic Beanstalk).

Recently visited services: IAM, DynamoDB, Cognito, CloudWatch, S3.

All services:

- Compute: EC2, EC2 Container Service, Lightsail, Elastic Beanstalk, Lambda, Batch.
- Storage: S3, EFS, Glacier, Storage Gateway.
- Database: RDS.
- Developer Tools: CodeCommit, CodeBuild, CodeDeploy, CodePipeline.
- Management Tools: CloudWatch, CloudFormation, CloudTrail, Config, OpsWorks, Service Catalog, Trusted Advisor, Managed Services, Application Discovery Service.
- Internet of Things: AWS IoT.
- Game Development: GameLift.
- Mobile Services: Mobile Hub, Cognito, Device Farm, Mobile Analytics, Pinpoint.
- Application Services: Step Functions, SWF.

Featured next steps:

- Manage your costs: Get real-time billing alerts based on your cost and usage budgets. [Start now](#).
- Get best practices: Use AWS Trusted Advisor for security, performance, cost and availability best practices. [Start now](#).

What's new?

- Announcing AWS Batch: Now generally available, AWS Batch enables developers, scientists, and engineers to process large-scale batch jobs with ease. [Learn more](#).
- Announcing Amazon Lightsail: See how this new service allows you to launch and manage your VPS with AWS for a low, predictable price. [Learn more](#).

See all

Select **Create Bucket**.

Want to manage your data based on what it is instead of where it's stored? [Try S3 Object Tagging](#)

Did you know? To view a window that lists properties, lifecycle, and permissions for the bucket, choose a bucket row. To edit, choose the section in the window that contains the information that you want to edit.

Amazon S3 [Switch to the old console](#)

+ [Create bucket](#) Delete bucket Empty bucket 0 Buckets 0 Regions

You do not have any buckets. Here is how to get started with Amazon S3.

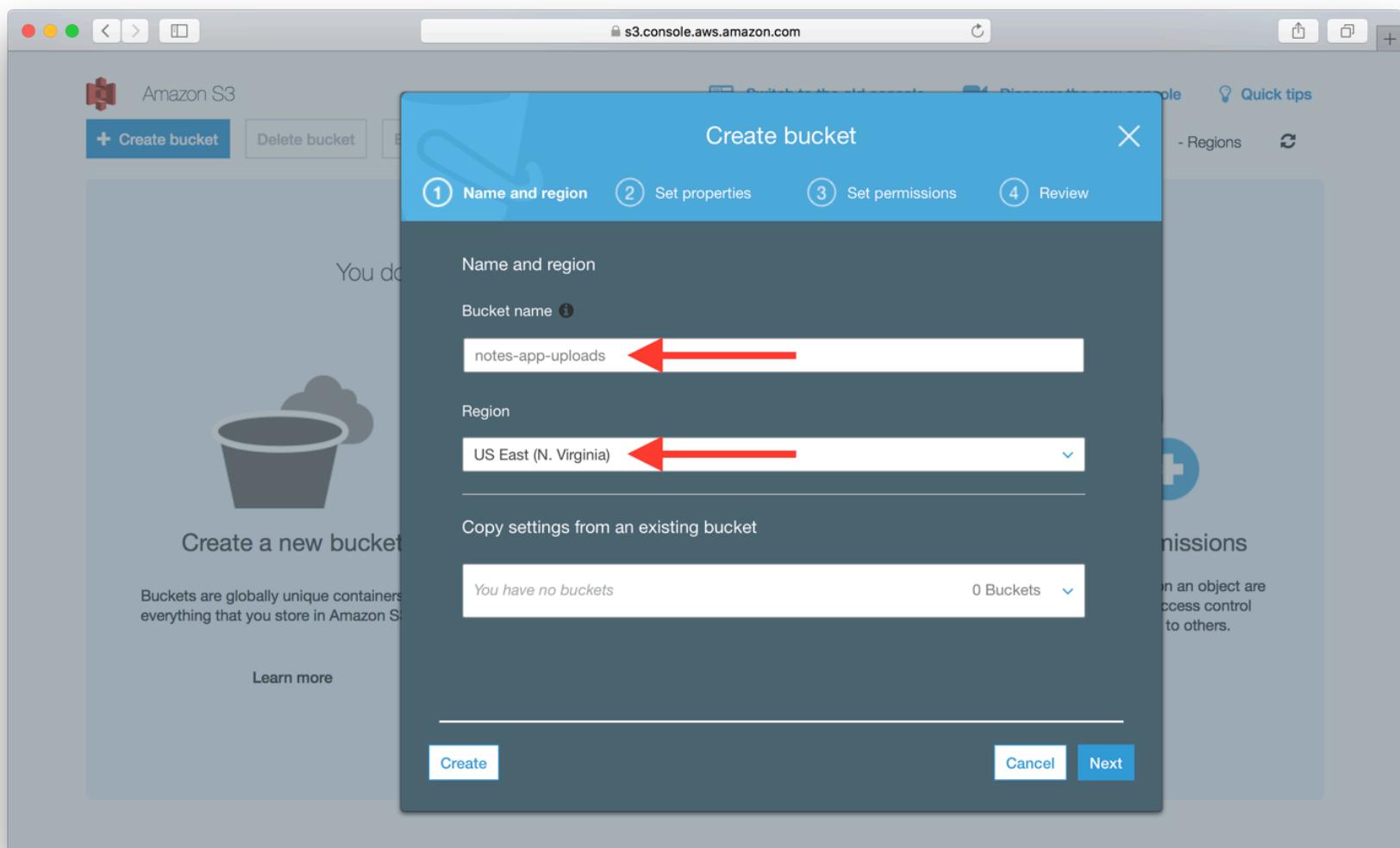
Create a new bucket

Upload your data

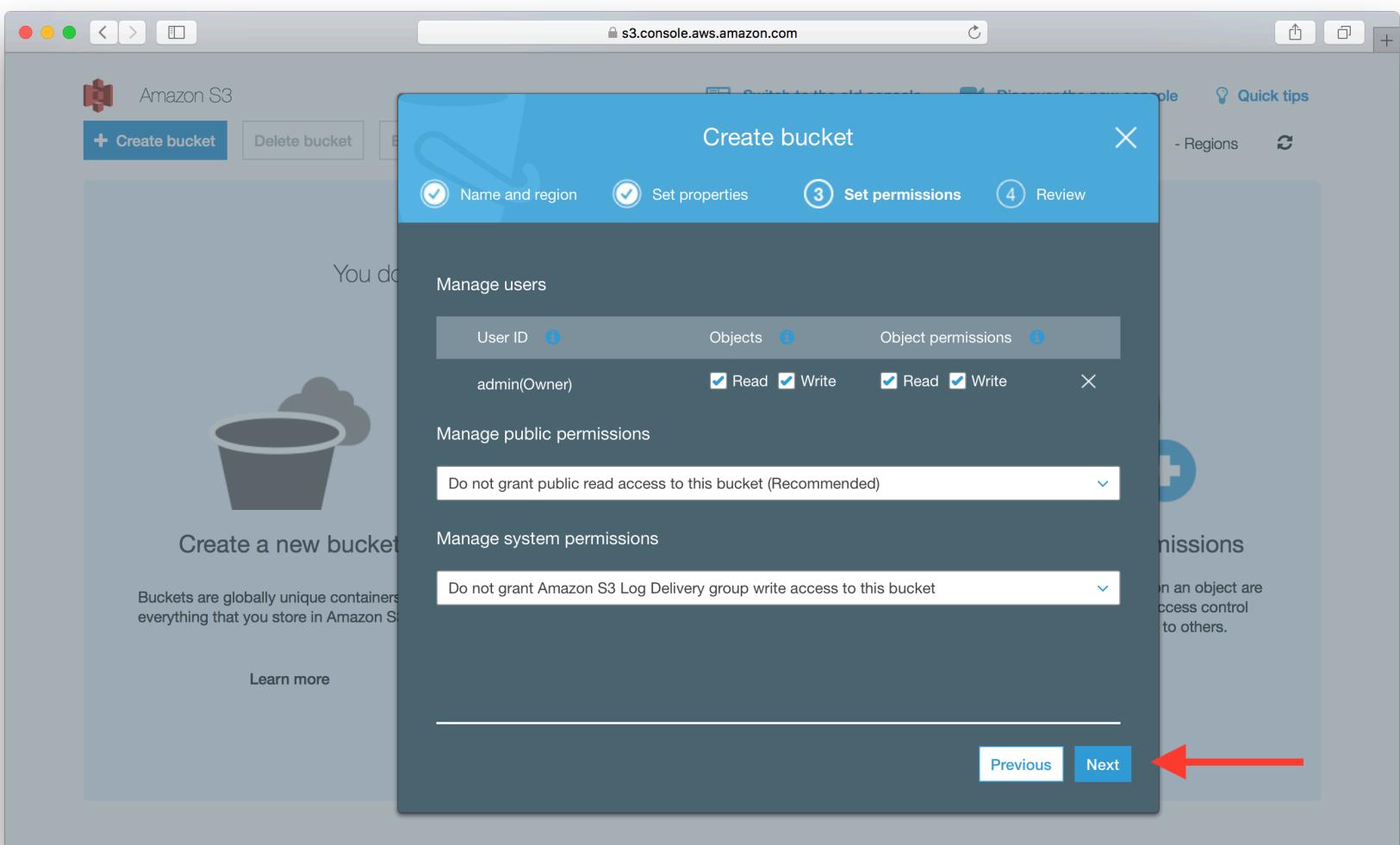
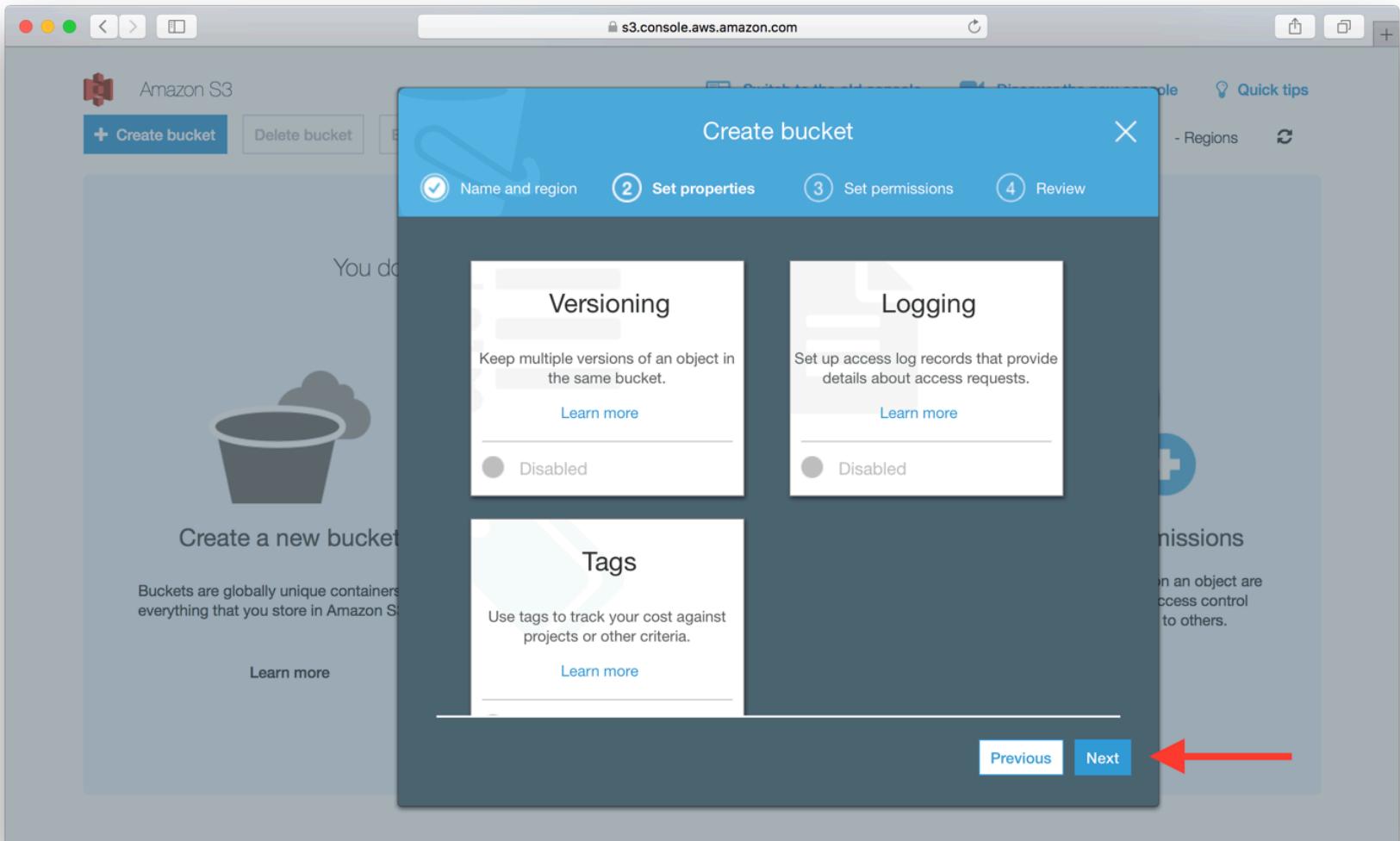
Set up your permissions

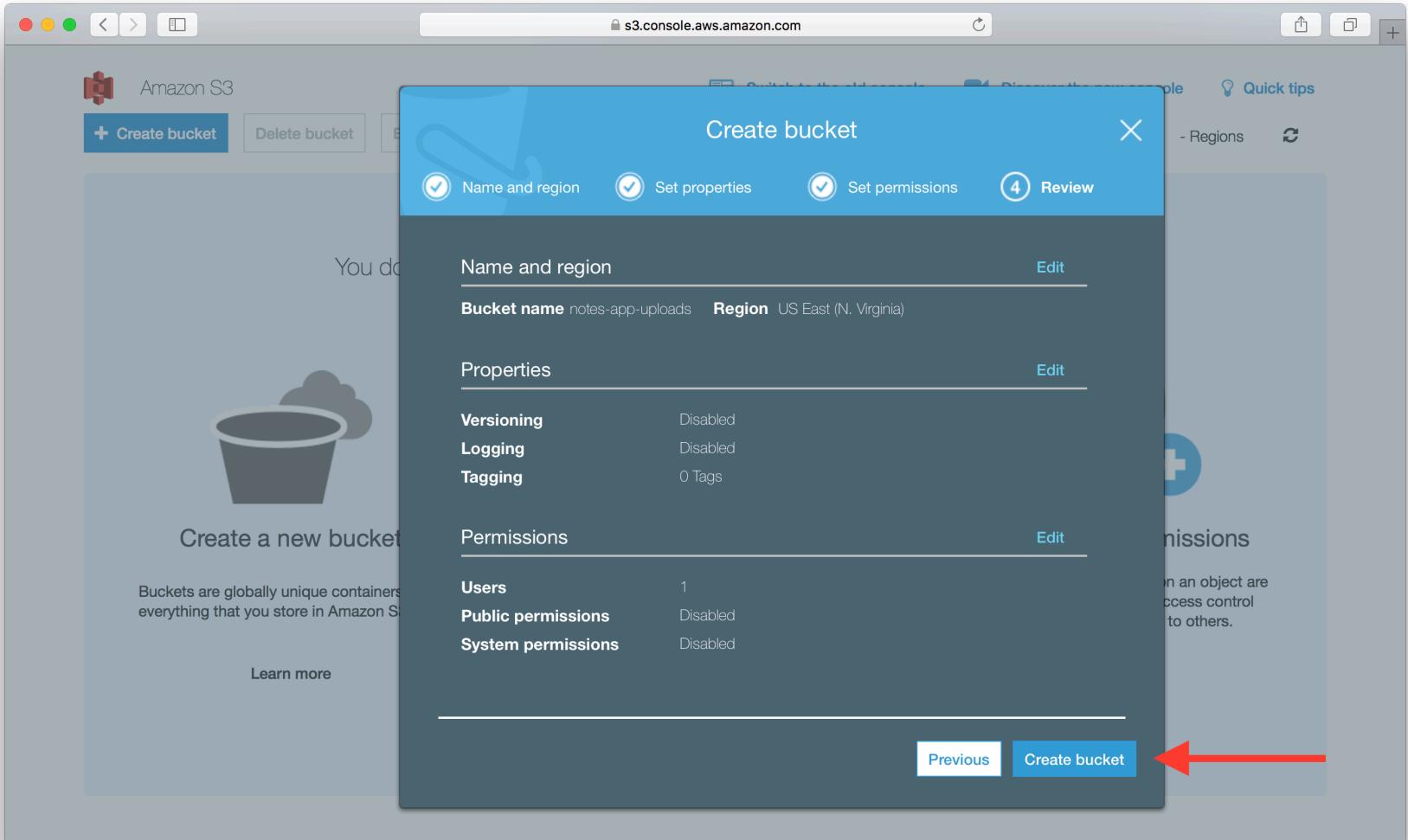
Pick a name of the bucket and select a region. Then select **Create**.

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



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





Enable CORS

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

Select the bucket we just created.

The screenshot shows the AWS S3 console interface. At the top, there's a banner with a link to 'Try S3 Object Tagging'. Below the banner, the 'Amazon S3' logo is on the left, and a 'Switch to the old console' button is on the right. Underneath, there are buttons for '+ Create bucket', 'Delete bucket', and 'Empty bucket'. It displays 1 Bucket and 1 Region. A search bar is present. The main area lists buckets with columns for 'Bucket name', 'Region', and 'Date created'. The bucket 'notes-app-uploads' is listed, with a red arrow pointing to its name.

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

The screenshot shows the 'Permissions' tab selected in the AWS S3 bucket configuration. A red arrow points to the 'Permissions' tab. Another red arrow points to the 'CORS configuration' button. The 'CORS configuration editor' section contains an ARN: arn:aws:s3:::notes-app-upload and a text area with a sample CORS policy. Buttons for 'Delete', 'Cancel', and 'Save' are at the bottom right.

```
1 <!-- Sample policy -->
2 <CORSConfiguration>
3   <CORSRule>
4     <AllowedOrigin>*</AllowedOrigin>
5     <AllowedMethod>GET</AllowedMethod>
6     <MaxAgeSeconds>3000</MaxAgeSeconds>
7     <AllowedHeader>Authorization</AllowedHeader>
8   </CORSRule>
9 </CORSConfiguration>
```

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

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

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

The screenshot shows the AWS S3 console interface for managing a bucket named 'notes-app-uploads'. The 'Management' tab is selected, and the 'CORS configuration' tab is active. A red arrow points from the provided XML code to the 'Save' button, indicating where to click to apply the changes. The XML code is identical to the one shown above.

```
1 <CORSConfiguration>
2   <CORSRule>
3     <AllowedOrigin>*</AllowedOrigin>
4     <AllowedMethod>GET</AllowedMethod>
5     <AllowedMethod>PUT</AllowedMethod>
6     <AllowedMethod>POST</AllowedMethod>
7     <AllowedMethod>HEAD</AllowedMethod>
8     <MaxAgeSeconds>3000</MaxAgeSeconds>
9     <AllowedHeader>*</AllowedHeader>
10   </CORSRule>
11 </CORSConfiguration>
```

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/16>)

Create a Cognito User Pool

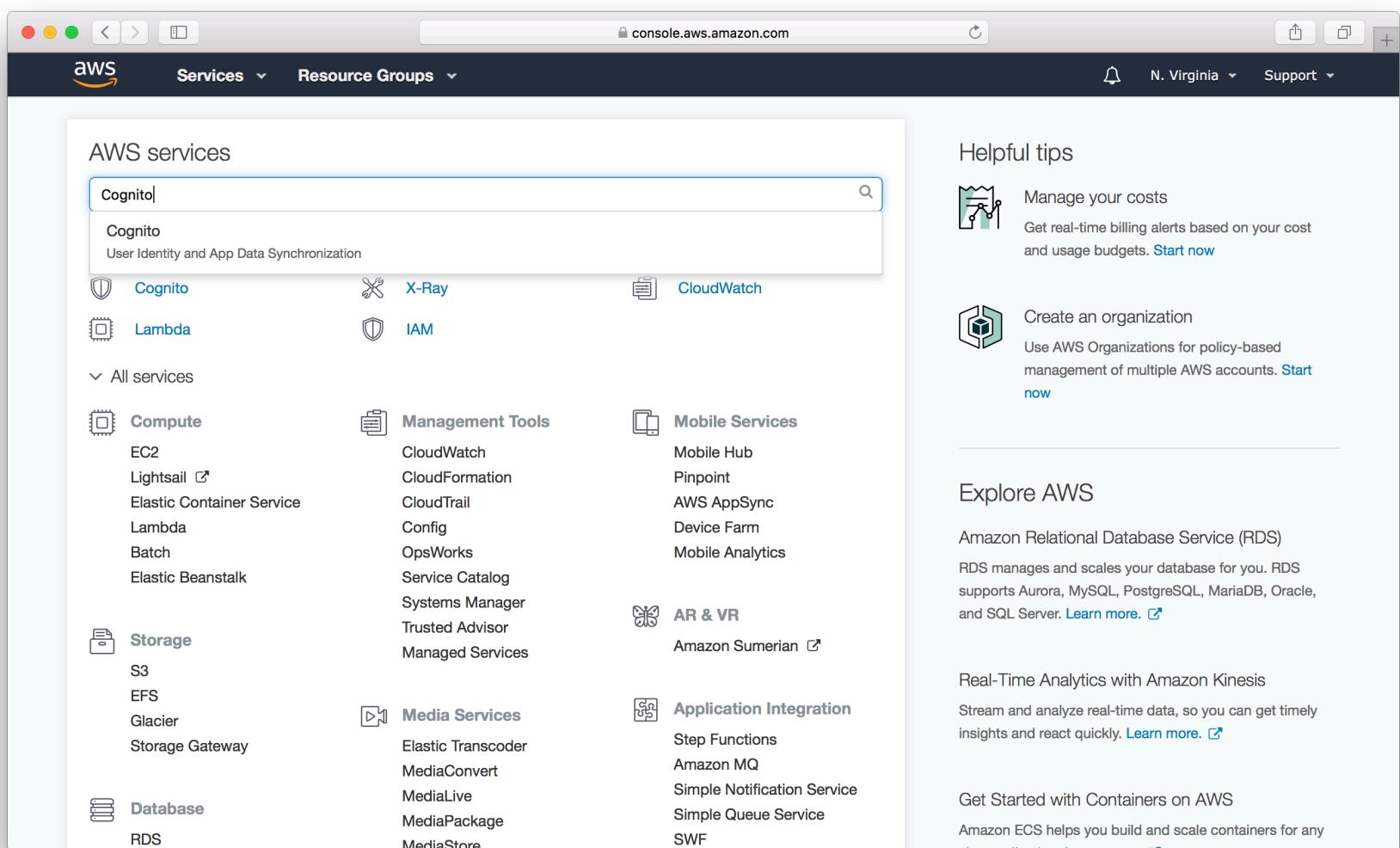
Our notes app needs to handle user accounts and authentication in a secure and reliable way. To do this we are going to use Amazon Cognito (<https://aws.amazon.com/cognito/>).

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

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

Create User Pool

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



Select Manage your User Pools.

The screenshot shows the Amazon Cognito service page in the AWS Management Console. At the top, there's a navigation bar with icons for Services, Resource Groups, and regions (N. Virginia). Below the header is the Amazon Cognito logo and the title "Amazon Cognito". A descriptive text block explains the service's purpose: "Amazon Cognito makes it easy for you to have users sign up and sign in to your apps, federate identities from social identity providers, secure access to AWS resources and synchronize data across multiple devices, platforms, and applications." Two blue buttons are visible: "Manage your User Pools" and "Manage Federated Identities". A red arrow points upwards from the bottom of the "Manage your User Pools" button towards the "Manage your User Pools" link in the main content area. The main content area is divided into three sections: "Add Sign-up and Sign-in" (with a user icon and a plus sign), "Federate User Identities" (with a computer monitor and lock icon), and "Synchronize Data Across Devices" (with a cloud and upload icon). Each section has a brief description and a small link below it.

Manage your User Pools

Manage Federated Identities

Add Sign-up and Sign-in

Federate User Identities

Synchronize Data Across Devices

With Cognito Your User Pools, you can easily and securely add sign-up and sign-in functionality to your mobile and web applications.

With Cognito Federated Identities, your users can sign-in through social identity providers such as Facebook and Twitter.

With Cognito Sync, your app can save user data, such as preferences and game state, and sync that data to make it available across devices.

Select **Create a User Pool**.

You have no user pools. [Click here to create a user pool.](#)

Create a User Pool

Enter Pool name and select Review defaults.

What do you want to name your user pool?

Give your user pool a descriptive name so you can easily identify it in the future.

Pool name

notes-user-pool

Cancel

Name

Attributes

Policies

Verifications

Message customizations

Tags

Devices

App clients

Triggers

Review

How do you want to create your user pool?

Review defaults

Start by reviewing the defaults and then customize as desired

Step through settings

Step through each setting to make your choices

Select Choose username attributes....

The screenshot shows the 'Create a user pool' interface in the AWS Cognito console. The 'Review' step is selected. In the 'Required attributes' section, 'email' is listed. Below it, under 'Username attributes', there is a link 'Choose username attributes...' which has a red arrow pointing to it. Other sections visible include 'Alias attributes', 'Custom attributes', and 'Minimum password length'. The 'Tags' and 'App clients' sections are also present.

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

Screenshot of the AWS User Pools 'Create a user pool' configuration page. The 'Attributes' tab is selected. A red arrow points to the 'Email address or phone number' sign-in option, which is selected.

How do you want your end users to sign in?

You can choose to have users sign in with an email address, phone number, username or preferred username plus their password. [Learn more](#).

Username - Users can use a username and optionally multiple alternatives to sign up and sign in.

Also allow sign in with verified email address

Also allow sign in with verified phone number

Also allow sign in with preferred username (a username that your users can change)

Email address or phone number - Users can use an email address or phone number as their "username" to sign up and sign in.

Allow email addresses

Allow phone numbers

Allow both email addresses and phone numbers (users can choose one)

Which standard attributes do you want to require?

All of the standard attributes can be used for user profiles, but the attributes you select will be required for sign up. You will not be able to change these requirements after the pool is created. If you select an attribute to be an alias, users will be able to sign-in using that value or their username. [Learn more about attributes](#).

Required	Attribute	Required	Attribute
<input type="checkbox"/>	address	<input type="checkbox"/>	nickname
<input type="checkbox"/>	birthdate	<input type="checkbox"/>	phone number
<input type="checkbox"/>	email	<input type="checkbox"/>	picture
<input type="checkbox"/>	family name	<input type="checkbox"/>	preferred username
<input type="checkbox"/>	gender	<input type="checkbox"/>	profile

Scroll down and select **Next step**.

Screenshot of the AWS User Pools 'Create a user pool' configuration page, showing the 'Standard attributes' section. A red arrow points to the 'Next step' button at the bottom.

All of the standard attributes can be used for user profiles, but the attributes you select will be required for sign up. You will not be able to change these requirements after the pool is created. If you select an attribute to be an alias, users will be able to sign-in using that value or their username. [Learn more about attributes](#).

Required	Attribute	Required	Attribute
<input type="checkbox"/>	address	<input type="checkbox"/>	nickname
<input type="checkbox"/>	birthdate	<input type="checkbox"/>	phone number
<input type="checkbox"/>	email	<input type="checkbox"/>	picture
<input type="checkbox"/>	family name	<input type="checkbox"/>	preferred username
<input type="checkbox"/>	gender	<input type="checkbox"/>	profile
<input type="checkbox"/>	given name	<input type="checkbox"/>	zoneinfo
<input type="checkbox"/>	locale	<input type="checkbox"/>	updated at
<input type="checkbox"/>	middle name	<input type="checkbox"/>	website
<input type="checkbox"/>	name		

Do you want to add custom attributes?

Enter the name and select the type and settings for custom attributes.

[Add custom attribute](#)

[Back](#) [Next step](#)

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

The screenshot shows the 'Create a user pool' interface in the AWS Management Console. The 'Review' tab is selected. In the 'Username attributes' section, 'email' is listed under 'Required attributes'. A red arrow points to this selection. On the left sidebar, the 'Review' tab is highlighted with a yellow background. A red arrow also points upwards from the 'Review' tab towards the 'Username attributes' section. At the bottom right of the page, there is a large blue 'Create pool' button.

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

The screenshot shows the AWS User Pools console in the 'Review' step. The 'Review' tab is selected in the navigation bar. On the left, there's a sidebar with 'Devices', 'App clients', 'Triggers', and 'Review'. The main area contains several configuration sections:

- Username attributes:** email
- Custom attributes:** Choose custom attributes...
- Minimum password length:** 8
- Password policy:** uppercase letters, lowercase letters, special characters, numbers
- User sign ups allowed?**: Users can sign themselves up
- MFA:** Enable MFA...
- Verifications:** Email
- Tags:** Choose tags for your user pool
- App clients:** Add app client...
- Triggers:** Add triggers...

A large red arrow points to the blue 'Create pool' button at the bottom right of the review section.

Now that the User Pool is created. Take a note of the **Pool Id** and **Pool ARN** which will be required later.

The screenshot shows the AWS Cognito User Pools console. A message at the top right says "Your user pool was created successfully." Below it, the "Pool Id" is listed as "us-east-1_XNvE8Ack3" and the "Pool ARN" is listed as "arn:aws:cognito-idp:us-east-1:232771856781:userpool/us-east-1_XNvE8Ack3". Red arrows point to both of these fields. The left sidebar shows navigation options like General settings, App integration (beta), and Federation (beta). The main content area displays various pool configuration details.

Your user pool was created successfully.

Pool Id us-east-1_XNvE8Ack3 ←

Pool ARN arn:aws:cognito-idp:us-east-1:232771856781:userpool/us-east-1_XNvE8Ack3 ←

Estimated number of users 0

Required attributes email

Alias attributes none

Custom attributes Choose custom attributes...

Minimum password length 8

Password policy uppercase letters, lowercase letters, special characters, numbers

User sign ups allowed? Users can sign themselves up

MFA Enable MFA... ↑

Verifications email ↑

Create App Client

Select App clients from the left panel.

Your user pool was created successfully.

Pool Id us-east-1_XNvE8Ack3
Pool ARN arn:aws:cognito-idp:us-east-1:232771856781:userpool/us-east-1_XNvE8Ack3

Estimated number of users 0

Required attributes email
Alias attributes none
Custom attributes Choose custom attributes...

Minimum password length 8
Password policy uppercase letters, lowercase letters, special characters, numbers
User sign ups allowed? Users can sign themselves up

MFA Enable MFA...
Verifications email

Select Add an app client.

Which app clients will have access to this user pool?

The app clients that you add below will be given a unique ID and an optional secret key to access this user pool.

Add an app client [Return to pool details](#)

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

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

The screenshot shows the AWS Lambda User Pools configuration page for a pool named 'notes-user-pool'. On the left, there's a sidebar with various settings like General settings, Users and groups, Attributes, Policies, Verifications, Message customizations, Tags, Devices, App clients (which is selected), and Triggers. Below that are sections for App integration (beta) and Federation (beta). The main area is titled 'Which app clients will have access to this user pool?' and contains fields for 'App client name' (set to 'notes-app'), 'Refresh token expiration (days)' (set to '30'), and checkboxes for 'Generate client secret' (unchecked), 'Enable sign-in API for server-based authentication (ADMIN_NO_SRP_AUTH)' (checked), and 'Only allow Custom Authentication (CUSTOM_AUTH_FLOW_ONLY)' (unchecked). At the bottom are 'Cancel' and 'Create app client' buttons, with the latter being highlighted by a red arrow.

Now that the app client is created. Take a note of the **App client id** which will be required in the later chapters.

The screenshot shows the AWS Lambda console with the function 'notes-lambda' selected. The left sidebar lists 'Function configuration', 'Triggers', 'Logs', and 'Metrics'. The main area displays the 'Handler' section, which includes the code snippet:

```
function handler(event, context) {
```

Below the code, there are sections for 'Environment variables' (with 'none' listed), 'Role' (selected role: 'Lambda execution role'), and 'Test' (with a dropdown menu for 'Create test event'). At the bottom, there are 'Save changes' and 'Cancel' buttons.

Create Domain Name

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

The screenshot shows the AWS Cognito User Pools console. A new user pool named "notes-user-pool" is being created. In the "Domain prefix" field, the URL "https://notes-app" is entered, with a red arrow pointing to the input field. In the "App integration" section, the "Domain name" field is highlighted with a red arrow. Other fields like "UI customization" and "Resource servers" are also visible.

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

For help and discussion

Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/17>)

Create a Cognito Test User

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

Create User

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

◆ CHANGE In your terminal, run.

```
$ aws cognito-idp sign-up \
--region us-east-1 \
--client-id YOUR_COGNITO_APP_CLIENT_ID \
--username admin@example.com \
--password Passw0rd!
```

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

◆ CHANGE In your terminal, run.

```
$ aws cognito-idp admin-confirm-sign-up \
--region us-east-1 \
--user-pool-id YOUR_USER_POOL_ID \
--username admin@example.com
```

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

Comments on this chapter
(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/18>)

Set up the Serverless Framework

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

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

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

Install Serverless

◆ CHANGE Create a directory for our API backend.

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

◆ CHANGE Install Serverless globally.

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

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

◆ CHANGE At the root of the project; create an AWS Node.js service.

```
$ serverless create --template aws-nodejs
```

Now the directory should contain the two following files, namely **handler.js** and **serverless.yml**.

```
$ ls  
handler.js      serverless.yml
```

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

Install AWS Related Dependencies

◆ CHANGE At the root of the project, run.

```
$ npm init -y
```

This creates a new Node.js project for you. This will help us manage any dependencies our project might have.

◆ CHANGE Next, install these two packages.

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

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

Now the directory should contain three files and one directory.

```
$ ls  
handler.js      node_modules      package.json      serverless.yml
```

- **node_modules** contains the Node.js dependencies that we just installed.
- **package.json** contains the Node.js configuration for our project.

Next, we are going to set up a standard JavaScript environment for us by adding support for

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/21>)

For reference, here is the code so far

🔗 Backend Source :setup-the-serverless-framework

(<https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/setup-the-serverless-framework>)

Add Support for ES6/ES7 JavaScript

By default, AWS Lambda only supports a specific version of JavaScript. It doesn't have an up-to-date Node.js engine. And looking a bit further ahead, we'll be using a more advanced flavor of JavaScript with ES6/ES7 features. So it would make sense to follow the same syntax on the backend and have a transpiler convert it to the target syntax. This would mean that we won't need to worry about writing different types of code on the backend or the frontend.

In this chapter, we are going to enable ES6/ES7 for AWS Lambda using the Serverless Framework. We will do this by setting up Babel (<https://babeljs.io>) and Webpack (<https://webpack.github.io>) to transpile and package our project. If you would like to code with AWS Lambda's default JavaScript version, you can skip this chapter. But you will not be able to directly use the sample code in the later chapters, as they are written in ES6 syntax.

Install Babel and Webpack

◆ CHANGE At the root of the project, run.

```
$ npm install --save-dev \
  babel-core \
  babel-loader \
  babel-plugin-transform-runtime \
  babel-preset-env \
  babel-preset-stage-3 \
  serverless-webpack \
  webpack \
  webpack-node-externals

$ npm install --save babel-runtime
```

Most of the above packages are only needed while we are building our project and they won't be deployed to our Lambda functions. We are using the `serverless-webpack` plugin to help trigger the Webpack build when we run our Serverless commands. The `webpack-node-`

`externals` is necessary because we do not want Webpack to bundle our `aws-sdk` module, since it is not compatible.

◆ CHANGE Create a file called `webpack.config.js` in the root with the following.

```
const slsw = require("serverless-webpack");
const nodeExternals = require("webpack-node-externals");

module.exports = {
  entry: slsw.lib.entries,
  target: "node",
  // Since 'aws-sdk' is not compatible with webpack,
  // we exclude all node dependencies
  externals: [nodeExternals()],
  // Run babel on all .js files and skip those in node_modules
  module: {
    rules: [
      {
        test: /\.js$/,
        loader: "babel-loader",
        include: __dirname,
        exclude: /node_modules/
      }
    ]
  }
};
```

This is the configuration Webpack will use to package our app. The main part of this config is the `entry` attribute that we are automatically generating using the `slsw.lib.entries` that is a part of the `serverless-webpack` plugin. This automatically picks up all our handler functions and packages them (we expand on this config at the end of our guide (`/chapters/serverless-nodejs-starter.html`) to make it a bit easier to use).

◆ CHANGE Next create a file called `.babelrc` in the root with the following.

```
{
  "plugins": ["transform-runtime"],
  "presets": ["env", "stage-3"]
```

}

The presets are telling Babel the type of JavaScript we are going to be using.

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

```
service: notes-app-api

# Use serverless-webpack plugin to transpile ES6/ES7
plugins:
  - serverless-webpack

# Enable auto-packing of external modules
custom:
  webpackIncludeModules: true

provider:
  name: aws
  runtime: nodejs6.10
  stage: prod
  region: us-east-1
```

And now we are ready to build our backend.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/22>)

For reference, here is the code so far

⌚ Backend Source : [add-support-for-es6-es7-javascript](#)

(<https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/add-support-for-es6-es7-javascript>)

Add a Create Note API

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

Add the Function

Let's add our first function.

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

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

AWS.config.update({ region: "us-east-1" });
const dynamoDb = new AWS.DynamoDB.DocumentClient();

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

    const data = JSON.parse(event.body);

    const params = {
        TableName: "notes",
        // 'Item' contains the attributes of the item to be created
        // - 'userId': user identities are federated through the
        //               Cognito Identity Pool, we will use the identity id
        //               as the user id of the authenticated user
        // - 'noteId': a unique uuid
        // - 'content': parsed from request body
        // - 'attachment': parsed from request body
        // - 'createdAt': current Unix timestamp
    }
}
```

```
Item: {

    userId: event.requestContext.identity.cognitoIdentityId,
    noteId: uuid.v1(),
    content: data.content,
    attachment: data.attachment,
    createdAt: new Date().getTime()

}

};

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

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

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

}
```

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

- We are setting the AWS JS SDK to use the region `us-east-1` while connecting to DynamoDB.
- Parse the input from the `event.body`. This represents the HTTP request parameters.
- The `userId` is a Federated Identity id that comes in as a part of the request. This is set after our user has been authenticated via the User Pool. We are going to expand more on this in the coming chapters when we set up our Cognito Identity Pool.
- Make a call to DynamoDB to put a new object with a generated `noteId` and the current date as the `createdAt`.
- Upon success, return the newly create note object with the HTTP status code `200` and response headers to enable **CORS (Cross-Origin Resource Sharing)**.
- And if the DynamoDB call fails then return an error with the HTTP status code `500`.

Configure the API Endpoint

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

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

```
service: notes-app-api

# Use serverless-webpack plugin to transpile ES6/ES7
plugins:
  - serverless-webpack

# Enable auto-packing of external modules
custom:
  webpackIncludeModules: true

provider:
  name: aws
  runtime: nodejs6.10
  stage: prod
  region: us-east-1

# 'iamRoleStatement' defines the permission policy for the Lambda
```

function.

```
# In this case Lambda functions are granted with permissions to  
access DynamoDB.
```

iamRoleStatements:

- Effect: Allow

Action:

- dynamodb:DescribeTable
- dynamodb:Query
- dynamodb:Scan
- dynamodb:GetItem
- dynamodb:PutItem
- dynamodb:UpdateItem
- dynamodb:DeleteItem

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

functions:

```
# Defines an HTTP API endpoint that calls the main function in  
create.js
```

```
# - path: url path is /notes  
# - method: POST request  
# - cors: enabled CORS (Cross-Origin Resource Sharing) for browser  
cross
```

```
# domain api call
```

```
# - authorizer: authenticate using the AWS IAM role
```

create:

```
handler: create.main
```

events:

- http:

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

Here we are adding our newly added create function to the configuration. We specify that it handles `post` requests at the `/notes` endpoint. We set CORS support to true. This is because our frontend is going to be served from a different domain. As the authorizer we are going to restrict access to our API based on the user's IAM credentials. We will touch on this

and how our User Pool works with this, in the Cognito Identity Pool chapter.

Test

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

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

```
$ mkdir mocks  
$ cd mocks
```

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

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

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

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

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

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

```
$ AWS_PROFILE=myProfile serverless invoke local --function create --
```

```
path mocks/create-event.json
```

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

The response should look similar to this.

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

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

Refactor Our Code

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

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

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

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

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

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

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

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

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

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

AWS.config.update({ region: "us-east-1" });

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

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

Here we are using the promise form of the DynamoDB methods. Promises are a method for managing asynchronous code that serve as an alternative to the standard callback function syntax. It will make our code a lot easier to read.

◆ CHANGE Now, we'll go back to our `create.js` and use the helper functions we created. Our `create.js` should now look like the following.

```
import uuid from "uuid";
```

```

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

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

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

```

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

Common Issues

- Response `statusCode: 500`

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

```

catch(e) {
  console.log(e);
}

```

```
    callback(null, failure({status: false}));  
}
```

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/23>)

For reference, here is the code so far

🔗 Backend Source : [add-a-create-note-api](#)

(<https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/add-a-create-note-api>)

Add a Get Note API

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

Add the Function

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

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

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

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

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

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

Configure the API Endpoint

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

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

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

Test

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

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

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

And we invoke our newly created function.

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

The response should look similar to this.

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

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/24>)

For reference, here is the code so far

Q Backend Source : add-a-get-note-api
(<https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/add-a-get-note-api>)

Add a List All the Notes API

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

Add the Function

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

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

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

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

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

Configure the API Endpoint

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

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

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

Test

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

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

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

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

The response should look similar to this.

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

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

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/25>)

For reference, here is the code so far

🔗 Backend Source :add-a-list-all-the-notes-api

(<https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/add-a-list-all-the-notes-api>)

Add an Update Note API

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

Add the Function

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

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

export async function main(event, context, callback) {
  const data = JSON.parse(event.body);
  const params = {
    TableName: "notes",
    // 'Key' defines the partition key and sort key of the item to be
    // updated
    // - 'userId': Identity Pool identity id of the authenticated user
    // - 'noteId': path parameter
    Key: {
      userId: event.requestContext.identity.cognitoIdentityId,
      noteId: event.pathParameters.id
    },
    // 'UpdateExpression' defines the attributes to be updated
    // 'ExpressionAttributeValues' defines the value in the update
    // expression
    UpdateExpression: "SET content = :content, attachment =
:attachment",
    ExpressionAttributeValues: {
      ":attachment": data.attachment ? data.attachment : null,
      ":content": data.content ? data.content : null
    },
    ReturnValues: "ALL_NEW"
  };
}
```

```
try {
  const result = await dynamoDbLib.call("update", params);
  callback(null, success({ status: true }));
} catch (e) {
  callback(null, failure({ status: false }));
}
}
```

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

Configure the API Endpoint

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

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

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

Test

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

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

```
{  
  "body": "{\"content\": \"new world\", \"attachment\": \"new.jpg\"}",  
  "pathParameters": {  
    "id": "578eb840-f70f-11e6-9d1a-1359b3b22944"  
  },  
  "requestContext": {  
    "identity": {  
      "cognitoIdentityId": "USER-SUB-1234"  
    }  
  }  
}
```

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

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

The response should look similar to this.

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

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/26>)

For reference, here is the code so far

Q Backend Source : add-an-update-note-api
(<https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/add-an-update-note-api>)

Add a Delete Note API

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

Add the Function

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

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

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

  try {
    const result = await dynamoDbLib.call("delete", params);
    callback(null, success({ status: true }));
  } catch (e) {
    callback(null, failure({ status: false }));
  }
}
```

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

Configure the API Endpoint

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

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

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

Test

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

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

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

Invoke our newly created function from the root directory.

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

And the response should look similar to this.

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

Now that our APIs are complete; we'll deploy them next.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/27>)

For reference, here is the code so far

⌚ Backend Source :add-a-delete-note-api

(<https://github.com/AnomalyInnovations/serverless-stack-demo-api/tree/add-a-delete-note-api>)

Deploy the APIs

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

◆ CHANGE Run the following in your working directory.

```
$ serverless deploy
```

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

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

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

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

```
Service Information
service: notes-app-api
stage: prod
region: us-east-1
api keys:
  None
endpoints:
  POST - https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/notes
  GET  - https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/notes/{id}
  GET  - https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/notes
  PUT  - https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/notes/{id}
```

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

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

Deploy a Single Function

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

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

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

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/28>)

For reference, here is the complete code



Backend Source

(<https://github.com/AnomalyInnovations/serverless-stack-demo-api>)

Create a Cognito Identity Pool

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

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

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

Let's get started.

Create Pool

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

Screenshot of the AWS Management Console Services page.

The sidebar shows recently visited services: Cognito, IAM, and S3. Under All services, the Mobile Services section is expanded, showing Mobile Hub, Cognito (highlighted with a red arrow), Device Farm, Mobile Analytics, and Pinpoint.

Featured next steps:

- Manage your costs: Get real-time billing alerts based on your cost and usage budgets. [Start now](#)
- Get best practices: Use AWS Trusted Advisor for security, performance, cost and availability best practices. [Start now](#)

What's new?

- Announcing AWS Batch: Now generally available, AWS Batch enables developers, scientists, and engineers to process large-scale batch jobs with ease. [Learn more](#)
- Announcing Amazon Lightsail: See how this new service allows you to launch and manage your VPS with AWS for a low, predictable price. [Learn more](#)

See all

Select Manage Federated Identities.

Screenshot of the Amazon Cognito service page.

The page features a purple hexagonal logo and the heading "Amazon Cognito". A descriptive text explains the service's capabilities: "Amazon Cognito makes it easy for you to have users sign up and sign in to your apps, federate identities from social identity providers, secure access to AWS resources and synchronize data across multiple devices, platforms, and applications." Below this, there are three main buttons: "Manage your User Pools" (highlighted with a red arrow) and "Manage Federated Identities".

Three cards are displayed below:

- Add Sign-up and Sign-in**: Shows two user silhouettes and a plus sign icon. Description: "With Cognito User Pools, you can easily and securely add sign-up and sign-in functionality to your mobile and web applications."
- Federate User Identities**: Shows a computer monitor with a lock icon. Description: "With Cognito Federated Identities, your users can sign-in through social identity providers such as Facebook and Google."
- Synchronize Data Across Devices**: Shows three clouds with an upward arrow icon. Description: "With Cognito Sync, your app can save user data, such as preferences and game state, and sync that data to make it available across devices."

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

The screenshot shows the 'Getting started wizard' for creating a new identity pool. On the left, a sidebar lists 'Step 1: Create identity pool' (selected) and 'Step 2: Set permissions'. The main panel is titled 'Create new identity pool' and contains instructions: 'Identity pools are used to store end user identities. To declare a new identity pool, enter a unique name.' A red arrow points to the 'Identity pool name*' input field, which contains 'notes identity pool'. Below it is a note: 'Example: My App Name'. Under 'Unauthenticated identities', there's a checkbox 'Enable access to unauthenticated identities' which is unchecked. Under 'Authentication providers', there's a section with a plus sign. At the bottom, a note says '* Required' and there are 'Cancel' and 'Create Pool' buttons.

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

Amazon Cognito can support unauthenticated identities by providing a unique identifier and AWS credentials for users who do not authenticate with an identity provider. If your application allows customers to use the application without logging in, you can enable access for unauthenticated identities. [Learn more about unauthenticated identities.](#)

Enable access to unauthenticated identities

▼ Authentication providers ⓘ ←

Amazon Cognito supports the following authentication methods with Amazon Cognito Sign-In or any public provider. If you allow your users to authenticate using any of these public providers, you can specify your application identifiers here. Warning: Changing the application ID that your identity pool is linked to will prevent existing users from authenticating using Amazon Cognito. [Learn more about public identity providers.](#)

Cognito **Amazon** **Facebook** **Google+** **Twitter / Digits** **OpenID** **SAML** **Custom**

Configure your Cognito Identity Pool to accept users federated with your Cognito User Pool by supplying the User Pool ID and the App Client ID.

User Pool ID: us-east-1_h7i5igIRb
ex: us-east-1_Ab129faBb

App Client ID: m9amduqqnniraapr8s2qo9kf6
ex: 7lhkkfbfb4q5kpp90urffao

Add Another Provider

* Required Cancel **Create Pool**

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

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

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

The screenshot shows the AWS IAM Role Summary page. At the top, there's a message: "Your Cognito identities require access to your resources". Below it, a note says: "Assigning a role to your application end users helps you restrict access to your AWS resources. Amazon Cognito integrates with Identity and Access Management (IAM) and lets you select specific roles for both your authenticated and unauthenticated identities. [Learn more about IAM](#)". A red arrow points to the "Hide Details" link. Another red arrow points to the "Hide Policy Document" link. A third red arrow points to the "Edit" button next to the policy document JSON code. At the bottom right, there are "Don't Allow" and "Allow" buttons.

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

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

The screenshot shows the same IAM Role Summary page as before, but with an "Edit Policy" dialog box overlaid. The dialog box contains the text: "Be sure to read the [Documentation](#) before editing." It has "Cancel" and "Ok" buttons. The background page is identical to the one above, with its respective UI elements and annotations.

◆ **CHANGE** Add the following policy into the editor. Replace

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

In our case `YOUR_S3_UPLOADS_BUCKET_NAME` is `notes-app-uploads`, `YOUR_API_GATEWAY_ID` is `1y55wbovq4`, and `YOUR_API_GATEWAY_REGION` is `us-east-1`.

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

```
"execute-api:Invoke"
],
"Resource": [
    "arn:aws:execute-
api:YOUR_API_GATEWAY_REGION:*:YOUR_API_GATEWAY_ID/*"
]
}
]
```

Note **cognito-identity.amazonaws.com:sub** is the authenticated user's federated identity ID. This policy grants the authenticated user access to files with filenames prefixed by the user's id in the S3 bucket as a security measure.

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

1. Files in our S3 bucket that are prefixed with their federated identity id
2. And, the APIs we deployed using API Gateway

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

Select **Allow**.

The screenshot shows the AWS Cognito IAM Role creation interface. At the top, there's a message about assigning roles to applications and users. Below that, a note says by default, Amazon Cognito creates a new role with limited permissions. The main section is titled "Role Summary" with a "Role Description" of "Your authenticated identities would like access to Cognito." It includes fields for "IAM Role" (set to "Create a new IAM Role") and "Role Name" ("Cognito_notesAuth_Role"). A "Hide Policy Document" button is visible. Below this is a large text area containing a JSON policy document:

```
"Effect": "Allow",
"Action": [
    "execute-api:Invoke"
],
"Resource": [
    "arn:aws:execute-api:us-east-1::ly55wbovq4/*"
]
```

At the top right of the policy editor is an "Edit" link. At the bottom right are "Don't Allow" and "Allow" buttons. A red arrow points from the text above to the "Allow" button.

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

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

The screenshot shows the AWS Cognito Federated Identities dashboard for the 'notes identity pool'. At the top right, there is a blue button labeled 'Edit identity pool'. On the far left, under 'Identity pool', there is a link 'Dashboard' which is highlighted with a red arrow. The main area displays 'Identities this month' (0), 'Total identities' (0), and a note about 'Cognito Sync'. Below this is a chart showing 'Total identities' from Feb 16 to Mar 01, with a single data point at 0. Under the heading 'Resources', there are links to 'Getting started with Amazon Cognito', 'Learn about the AWS Mobile SDKs', and 'Connect with the community'.

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

The screenshot shows the 'Edit identity pool' page for the 'notes identity pool'. The left sidebar shows the 'Identity pool' section with a link 'Dashboard'. The main content area is titled 'Edit identity pool' and contains a paragraph about modifying identity pool details. It features two input fields: 'Identity pool name*' (set to 'notes identity pool') and 'Identity pool ID*' (set to 'us-east-1:3eb9653f-e567-4dba-8a85-a0c7c3e390a3'). Below these are sections for 'Unauthenticated role' and 'Authenticated role', each with a dropdown menu and a 'Create new role' button. At the bottom, there are several expandable sections: 'Unauthenticated identities', 'Authentication providers', 'Push synchronization', and 'Cognito Streams'.

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

For help and discussion

Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/19>)

Cognito User Pool vs Identity Pool

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

Here is what AWS defines the Cognito User Pool as:

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

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

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

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

User Pool

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

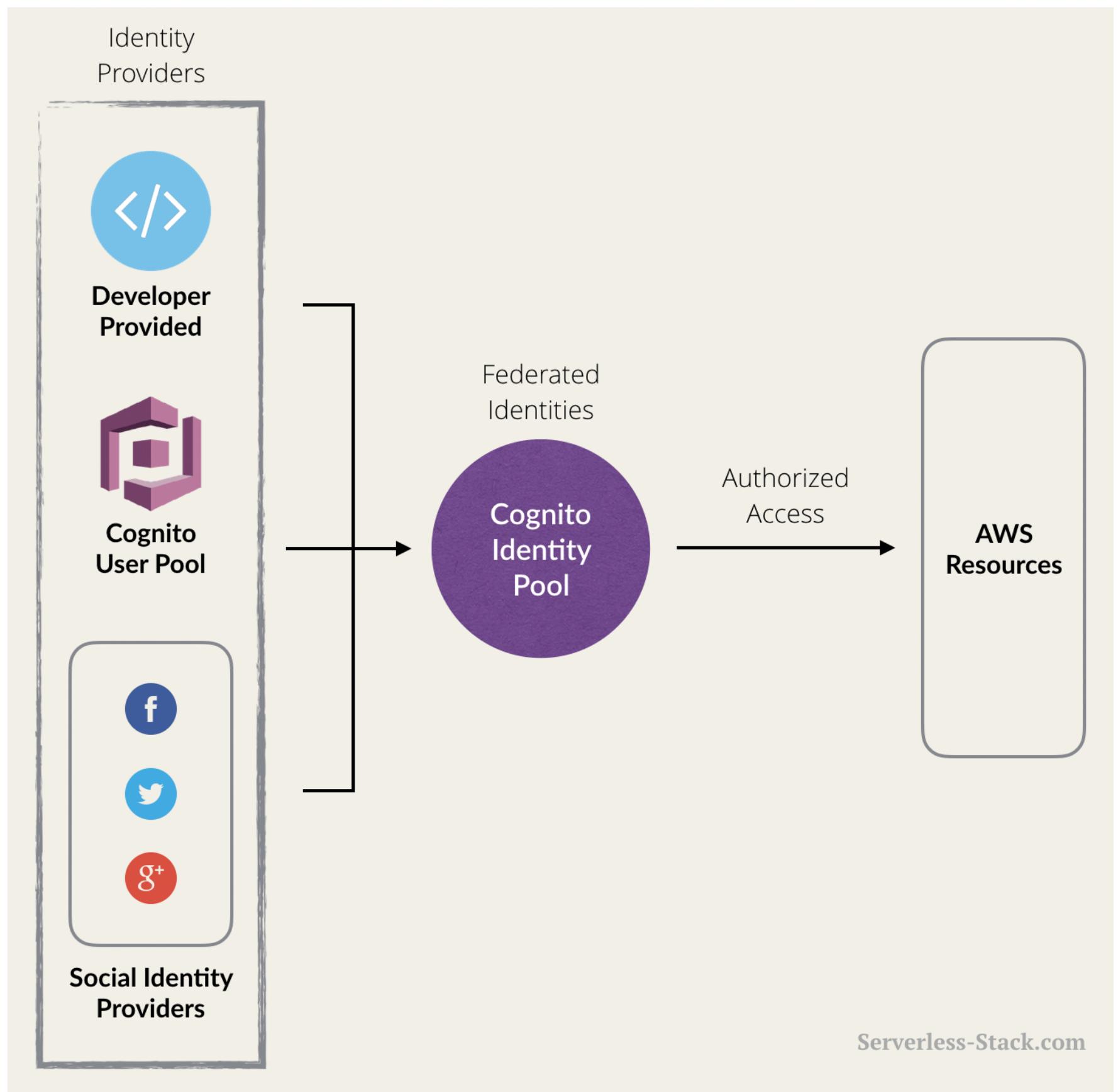
Identity Pool

Cognito Identity Pool (or Cognito Federated Identities) on the other hand is a way to authorize your users to use the various AWS services. Say you wanted to allow a user to have access to

your S3 bucket so that they could upload a file; you could specify that while creating an Identity Pool. And to create these levels of access, the Identity Pool has its own concept of an identity (or user). The source of these identities (or users) could be a Cognito User Pool or even Facebook or Google.

User Pool vs Identity Pool

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



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

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

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

For help and discussion

 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/20>)

Test the APIs

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

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

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

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

You can install it by running the following.

```
$ npm install -g aws-api-gateway-cli-test
```

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

- Use the username and password of the user created in the Create a Cognito test user (/chapters/create-a-cognito-test-user.html) chapter.
- Replace **YOUR_COGNITO_USER_POOL_ID**, **YOUR_COGNITO_APP_CLIENT_ID**, and **YOUR_COGNITO_REGION** with the values from the Create a Cognito user pool (/chapters/create-a-cognito-user-pool.html) chapter. In our case the region is `us-east-1`.
- Replace **YOUR_IDENTITY_POOL_ID** with the one from the Create a Cognito identity pool (/chapters/create-a-cognito-identity-pool.html) chapter.
- Use the **YOUR_API_GATEWAY_URL** and **YOUR_API_GATEWAY_REGION** with the ones from the Deploy the APIs (/chapters/deploy-the-apis.html) chapter. In our case the URL is `https://1y55wbovq4.execute-api.us-east-1.amazonaws.com/prod` and the region is `us-east-1`.

And run the following.

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

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

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

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

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

```
Authenticating with User Pool
Getting temporary credentials
Making API request
{
  status: 200,
  statusText: 'OK',
  data:
    { userId: 'us-east-1:9bdc031d-ee9e-4ffa-9a2d-123456789',
```

```
noteId: '8f7da030-650b-11e7-a661-123456789',  
content: 'hello world',  
attachment: 'hello.jpg',  
createdAt: 1499648598452 } }
```

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

Common Issues

- Response {status: 403}

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

- Ensure the `--path-template` option in the `apig-test` command is pointing to `/notes` and not `notes`. The format matters for securely signing our request.
- There are no trailing slashes for `YOUR_API_GATEWAY_URL`. In our case, the URL is `https://1y55wbovq4.execute-api.us-east-1.amazonaws.com/prod`. Notice that it does not end with a `/`.
- If you're on Windows and are using Git Bash, try adding a trailing slash to `YOUR_API_GATEWAY_URL` while removing the leading slash from `--path-template`. In our case, it would result in `--invoke-url https://1y55wbovq4.execute-api.us-east-1.amazonaws.com/prod/ --path-template notes`. You can follow the discussion on this here (<https://github.com/AnomalyInnovations/serverless-stack-com/issues/112#issuecomment-345996566>).

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

Next, you can enable API Gateway logs (/chapters/api-gateway-and-lambda-logs.html#enable-api-gateway-cloudwatch-logs) and follow these instructions

(/chapters/api-gateway-and-lambda-logs.html#viewing-api-gateway-cloudwatch-logs) to read the requests that are being logged. This should give you a better idea of what is going on.

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

- Response `{status: false}`

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

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

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

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

For help and discussion

💬 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/112>)

For reference, here is the complete code

🔗 Backend Source

(<https://github.com/AnomalyInnovations/serverless-stack-demo-api>)

Create a New React.js App

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

Install Create React App

◆ CHANGE Create a new project in directory separate from the backend. Run the following command.

```
$ npm install -g create-react-app
```

This installs the Create React App NPM package globally.

Create a New App

◆ CHANGE From your working directory, run the following command to create the client for our notes app.

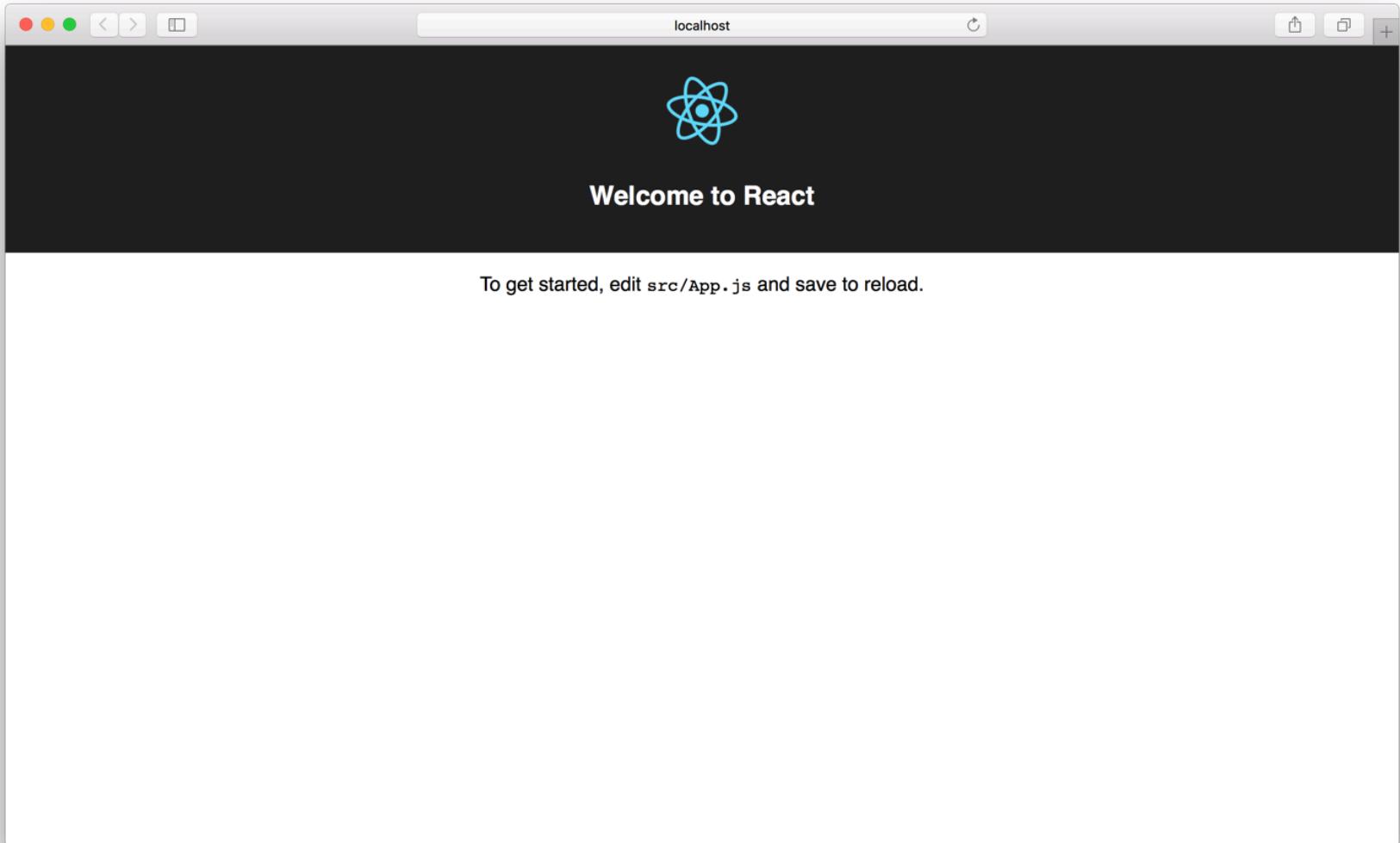
```
$ create-react-app notes-app-client
```

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

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

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

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



Change the Title

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

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

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

Next, we are going to create our app icon and update the favicons.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/29>)

For reference, here is the code so far

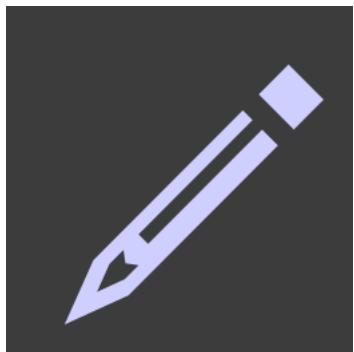
Q Frontend Source :create-a-new-reactjs-app
(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/create-a-new-reactjs-app>)

Add App Favicons

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

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

Right-click to download the following image.



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

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

The screenshot shows the homepage of realfavicongenerator.net. At the top, there's a navigation bar with links for Favicon, Social, API, Contribute, and Misc. Below the header, the main title "Favicon Generator. For real." is displayed. Underneath the title, there are three sections: "All browsers" (with icons for Chrome, Firefox, Opera, and Edge), "All platforms" (with icons for iOS, Android, Windows, and Mac), and "Your favorite technologies" (with icons for WordPress, Tumblr, GitHub, Stack Overflow, Node.js, and others). To the right of these sections is a large blue button labeled "Select your Favicon picture" with a red arrow pointing to it. Below this button, text instructions say "Submit a picture (PNG, JPG, SVG...), at least 70x70. Your picture should be 260x260 or more for optimal results." There's also a link "Demo with this picture" and an "or" option. Below the main form, there's a section titled "Check your favicon" with a sub-instruction "Check your existing favicon with our online tool and see what can be improved." It includes a URL input field with "http://www.example.com" and a "Check Favicon" button. A sub-section titled "Why RealFaviconGenerator" with the sub-instruction "No hard decision" is also visible.

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.

The screenshot shows a web browser window for realfavicongenerator.net. At the top, there's a 'Touch Bar' section with three icons: Apple, a pencil, and YouTube. Below it, the main content area has a title 'Favicon Generator Options'. There are five tabs: Path (selected), Version/Refresh, Compression, Scaling algorithm, and App name. Under the Path tab, there are two radio button options: one selected (blue outline) for placing files at the root of the website, and another unselected (grey outline) for placing them elsewhere. A text input field shows the path '/path/to/icons'. At the bottom of this section is a blue button labeled 'Generate your Favicons and HTML code' with a red arrow pointing to it from the left.

Touch Bar

I will place favicon files ([favicon.ico](#), [apple-touch-icon.png](#), etc.) at the root of my web site. Recommended.

I cannot or I do not want to place favicon files at the root of my web site. Instead I will place them here:
/path/to/icons or <http://myothersite.com/path/to/icons>

Generate your Favicons and HTML code

Favicon Generator
Favicon Checker
Favicon for Gulp
Favicon for Grunt
Favicon for Ruby on Rails
Favicon for Node.js CLI
Favicon for Google Web Starter Kit

FAQ
Blog
Report bug
Change Log
Compatibility
Extensions
API
Compatibility Test

Contact us
Donate
Newsletter
Terms of service
Privacy policy
Cookies

RealFaviconGenerator.net © 2013-2017

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.

The screenshot shows a web browser window with the URL `realfavicongenerator.net` in the address bar. The page title is "Install your favicon". A message at the top states: "The new package is **much lighter**, yet keeping the **same compatibility level** as before. If you prefer the old, 20+ icons, 15+ lines of code, you can [Get the old package](#)". Below this are several navigation links: HTML5, XHTML, Jade, Grunt, Gulp, Node CLI, RoR, Google Web Starter Kit, and RFG API. The main content area contains five numbered steps:

1. Download your package: [Favicon package](#) (highlighted with a red arrow)
2. Extract this package in the root of your web site. If your site is `http://mysite.com`, you should be able to access a file named `http://mysite.com/favicon.ico`.
3. Insert the following code in the `<head>` section of your pages:

```
<link rel="apple-touch-icon" sizes="180x180" href="/apple-touch-icon.png">
<link rel="icon" type="image/png" href="/favicon-32x32.png" sizes="32x32">
<link rel="icon" type="image/png" href="/favicon-16x16.png" sizes="16x16">
<link rel="manifest" href="/manifest.json">
<link rel="mask-icon" href="/safari-pinned-tab.svg" color="#5bbad5">
<meta name="theme-color" content="#ffffff">
```

(highlighted with a red box and arrow)
4. *Optional* - Once your web site is deployed, [check your favicon](#).
5. *Optional* - Your favicon is fantastic. [Share it!](#)

At the bottom, there are "Share" and "Facebook editor" sections with social sharing icons and a counter of 13.8K.

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

```
"background_color": "#ffffff"  
}
```

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

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

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

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

```
<meta name="theme-color" content="#000000">  
<link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
```

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

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/30>)

For reference, here is the code so far

⌚ Frontend Source :add-app-favicons

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/add-app-favicons>)

Set up Custom Fonts

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

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

Include Google Fonts

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

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

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

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

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

Add the Fonts to the Styles

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

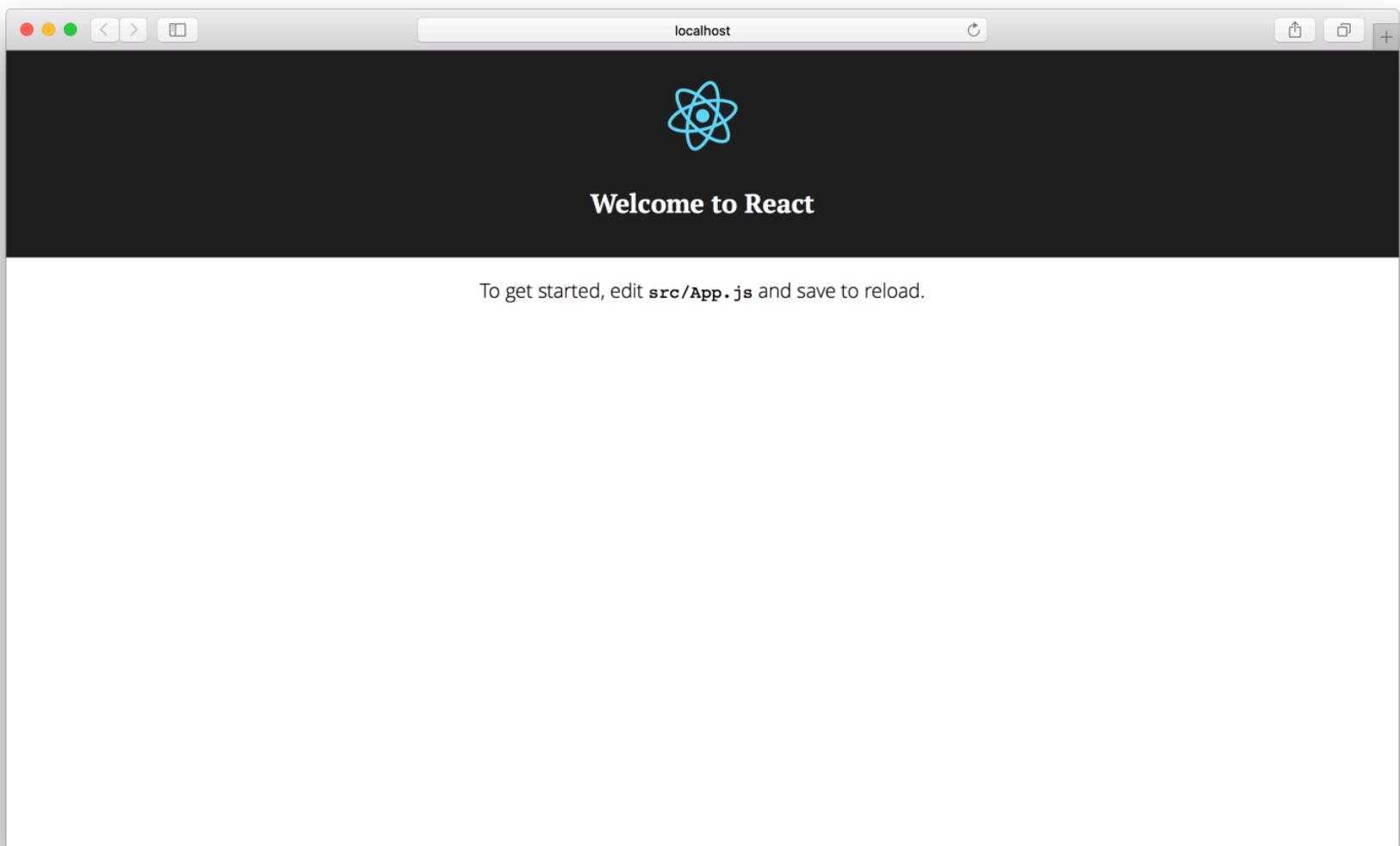
◆ CHANGE Let's change the current font in `src/index.css` for the `body` tag from `font-family: sans-serif;` to the following.

```
font-family: "Open Sans", sans-serif;  
font-size: 16px;  
color: #333;
```

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

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

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



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

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/31>)

For reference, here is the code so far

⌚ Frontend Source :[setup-custom-fonts](#)

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/setup-custom-fonts>)

Set up Bootstrap

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

Installing React Bootstrap

◆ CHANGE Run the following command in your working directory

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

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

Add Bootstrap Styles

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

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

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

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

```
select.form-control,  
textarea.form-control,  
input.form-control {
```

```
font-size: 16px;  
}  
  
input[type=file] {  
    width: 100%;  
}
```

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

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

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/32>)

For reference, here is the code so far

💻 Frontend Source :setup-bootstrap

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/setup-bootstrap>)

Handle Routes with React Router

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

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

Installing React Router v4

◆ CHANGE Run the following command in your working directory.

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

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

Setting up React Router

Even though we don't have any routes set up in our app, we can get the basic structure up and running. Our app currently runs from the `App` component in `src/App.js`. We are going to be using this component as the container for our entire app. To do that we'll encapsulate our `App` component within a `Router`.

◆ CHANGE Replace code in `src/index.js` with the following.

```
import React from "react";
import ReactDOM from "react-dom";
import { BrowserRouter as Router } from "react-router-dom";
import App from "./App";
import registerServiceWorker from "./registerServiceWorker";
import "./index.css";
```

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

We've made two small changes here.

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

Now if you head over to your browser, your app should load just like before. The only difference being that we are using React Router to serve out our pages.

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/33>)

For reference, here is the code so far

🔗 Frontend Source :handle-routes-with-react-router

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/handle-routes-with-react-router>)

Create Containers

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

Add a Navbar

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

◆ **CHANGE** And go ahead and remove the code inside `src/App.js` and replace it with the following. Also, you can go ahead and remove `src/logo.svg`.

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

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

```
</div>
);
}

}

export default App;
```

We are doing a few things here:

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

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

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

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

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

Add the Home container

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

◆ CHANGE Create a `src/containers` directory and add the following inside `src/containers/Home.js`.

```
import React, { Component } from "react";
import "./Home.css";
```

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

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

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

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

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

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

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

Set up the Routes

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

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

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

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

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

Render the Routes

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

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

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

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

```
<Routes />
```

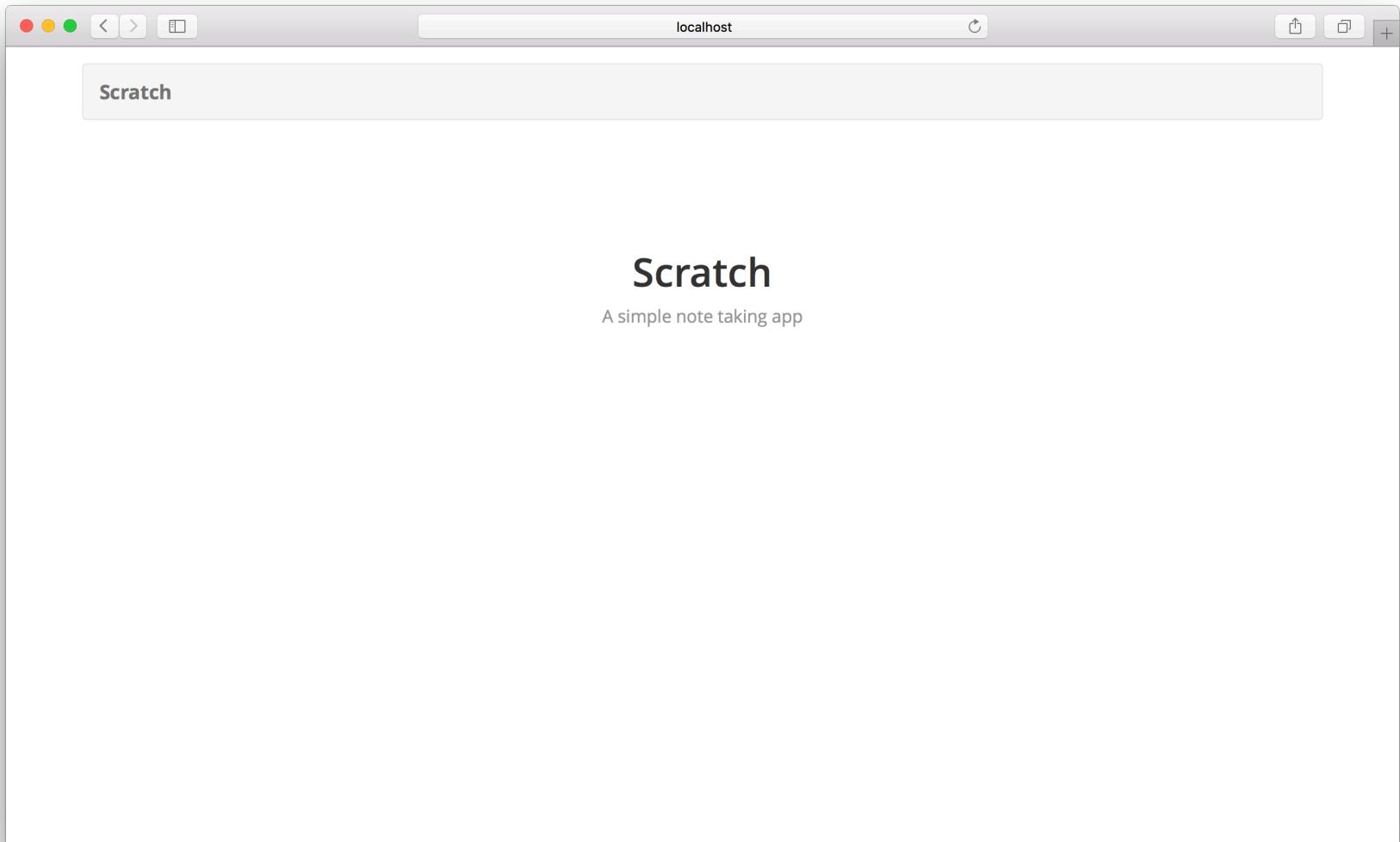
So the `render` method of our `src/App.js` should now look like this.

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

```
</Navbar.Brand>
<Navbar.Toggle />
</Navbar.Header>
</Navbar>
<Routes />
</div>
);
}
```

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

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



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

For help and discussion



Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/34>)

For reference, here is the code so far

Frontend Source :create-containers

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/create-containers>)

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 `render` method in `src/App.js` with the following.

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

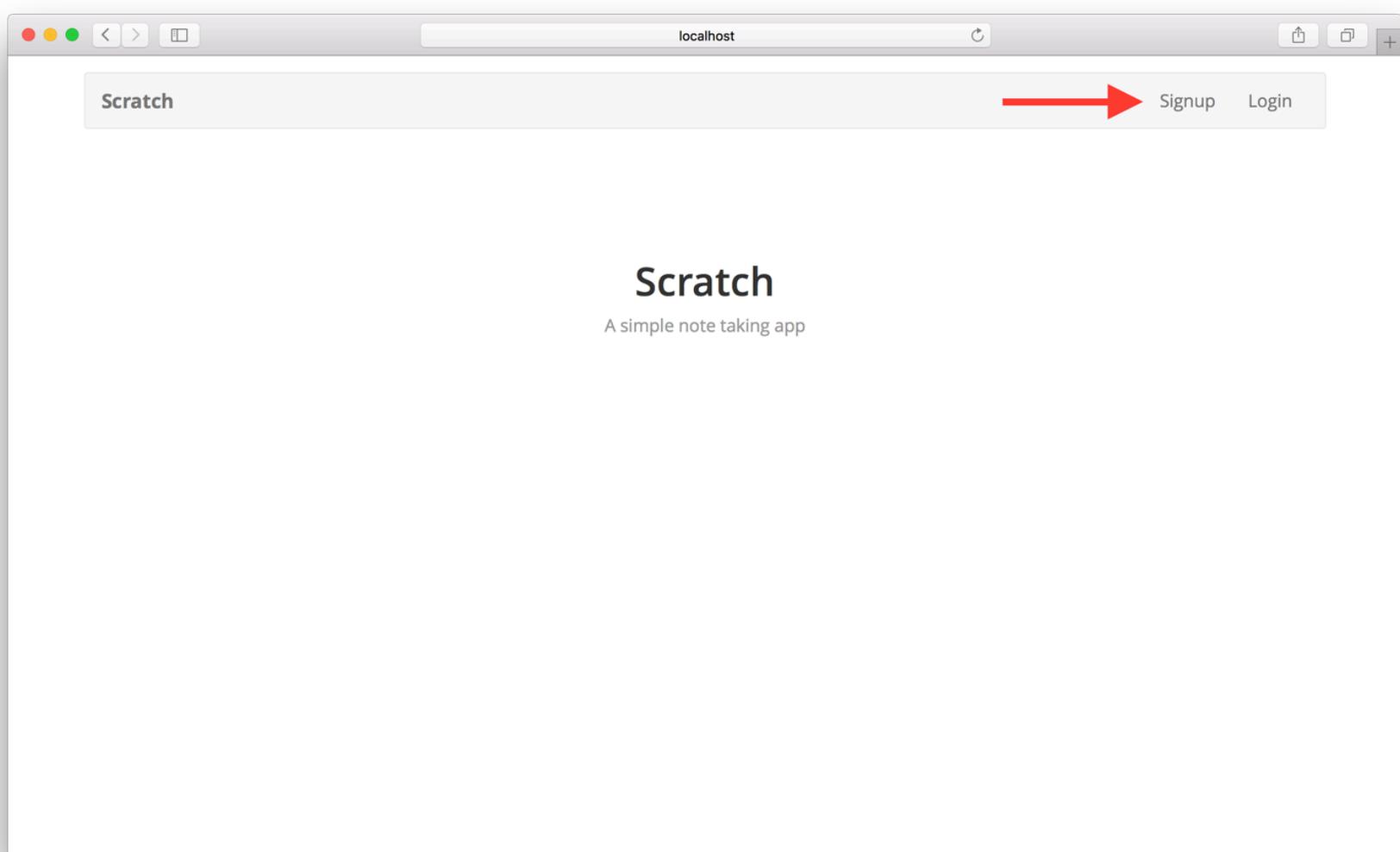
This adds two links to our navbar using the `NavItem` Bootstrap component. The `Navbar.Collapse` component ensures that on mobile devices the two links will be collapsed.

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

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

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

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



Unfortunately, they don't do a whole lot when you click on them. We also need them to highlight when we navigate to that page. To fix this we are going to use a useful feature of the React-Router. We are going to use the `Route` component to detect when we are on a certain page and then render based on it. Since we are going to do this twice, let's make this into a component that can be re-used.

◆ CHANGE Create a `src/components/` directory and add the following inside `src/components/RouteNavItem.js`.

```
import React from "react";
import { Route } from "react-router-dom";
import { NavItem } from "react-bootstrap";
```

```
export default props =>
  <Route
    path={props.href}
    exact
    children={({ match, history }) =>
      <NavItem
        onClick={e =>
          history.push(e.currentTarget.getAttribute("href"))}
        {...props}
        active={match ? true : false}
      >
        {props.children}
      </NavItem>}
    />;
  }
```

This is doing a couple of things here:

1. We look at the `href` for the `NavItem` and check if there is a match.
2. React-Router passes in a `match` object in case there is a match. We use that and set the `active` prop for the `NavItem`.
3. React-Router also passes us a `history` object. We use this to navigate to the new page using `history.push`.

Now let's use this component.

◆ CHANGE Import this component in the header of our `src/App.js`.

```
import RouteNavItem from "./components/RouteNavItem";
```

◆ CHANGE And remove the `NavItem` from the header of `src/App.js`, so that the `react-bootstrap` import looks like this.

```
import { Nav, Navbar } from "react-bootstrap";
```

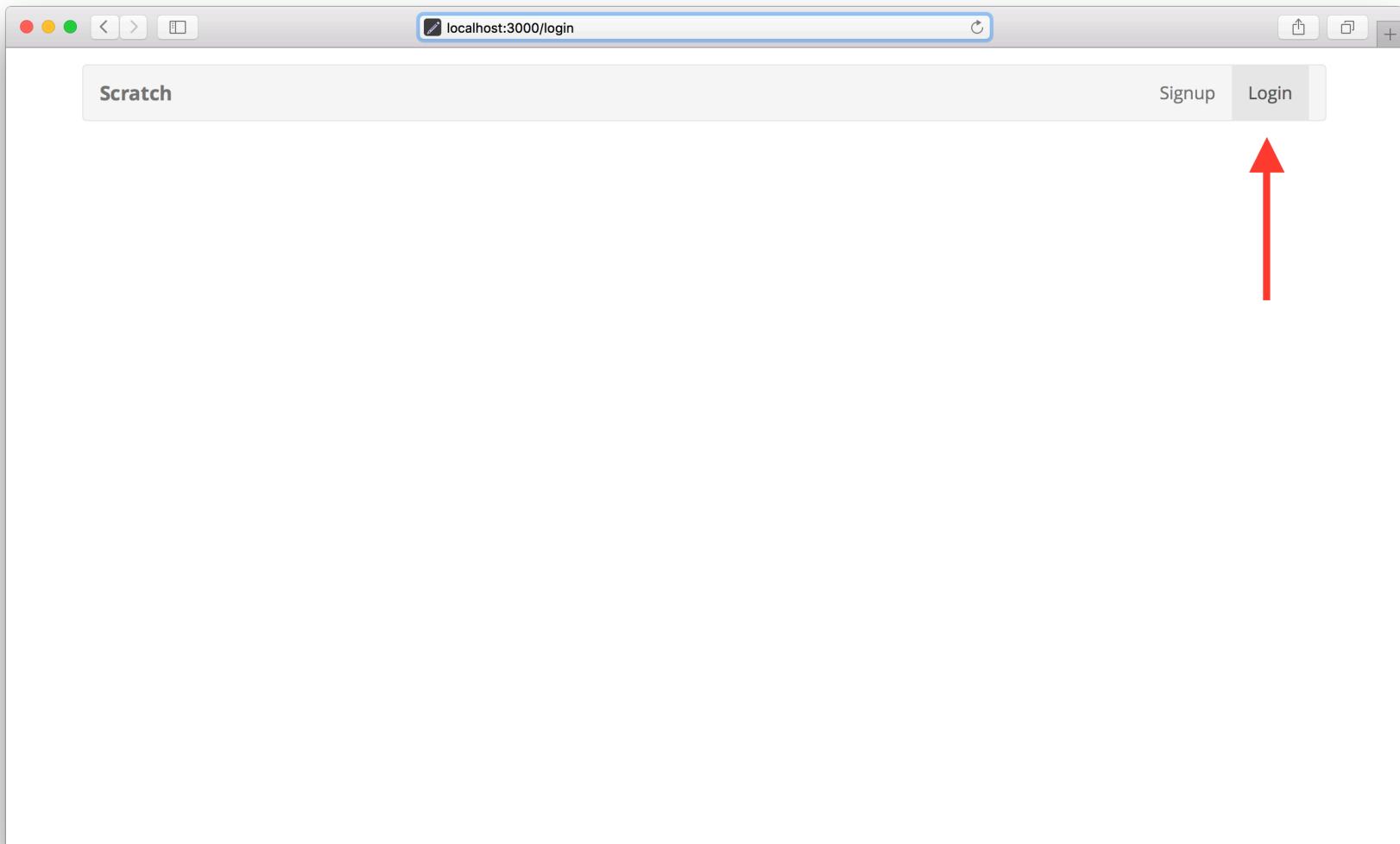
◆ CHANGE Now replace the `NavItem` components in `src/App.js`.

```
<NavItem href="/signup">Signup</NavItem>  
<NavItem href="/login">Login</NavItem>
```

◆ CHANGE With the following.

```
<RouteNavItem href="/signup">Signup</RouteNavItem>  
<RouteNavItem href="/login">Login</RouteNavItem>
```

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.



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

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

For help and discussion

Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/35>)

For reference, here is the code so far

⌚ Frontend Source :[adding-links-in-the-navbar](#)

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/adding-links-in-the-navbar>)

Handle 404s

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

Create a Component

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

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

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

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

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

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

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

Add a Catch All Route

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

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

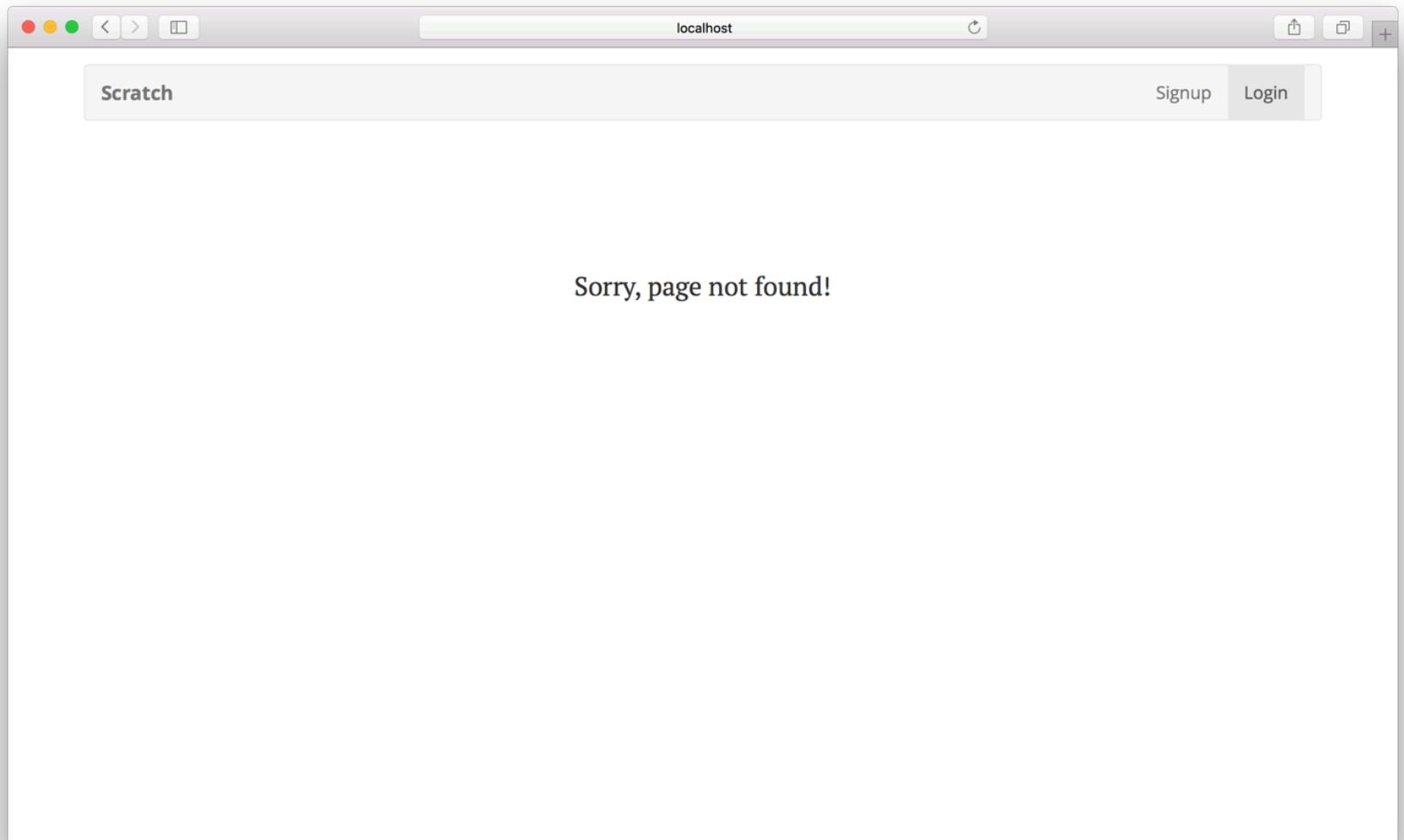
```
{ /* Finally, catch all unmatched routes */ }  
<Route component={NotFound} />
```

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

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

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

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



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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/36>)

For reference, here is the code so far

💡 Frontend Source : `handle-404s`

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/handle-404s>)

Create a Login Page

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

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

Add the Container

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

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

export default class Login extends Component {
  constructor(props) {
    super(props);

    this.state = {
      email: "",
      password: ""
    };
  }

  validateForm() {
    return this.state.email.length > 0 && this.state.password.length >
0;
  }

  handleChange = event => {
```

```
this.setState({
  [event.target.id]: event.target.value
});

handleSubmit = event => {
  event.preventDefault();
}

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

```
</form>
</div>
);
}
}
```

We are introducing a couple of new concepts in this.

1. In the constructor of our component we create a state object. This will be where we'll store what the user enters in the form.
2. We then connect the state to our two fields in the form by setting `this.state.email` and `this.state.password` as the `value` in our input fields. This means that when the state changes, React will re-render these components with the updated value.
3. But to update the state when the user types something into these fields, we'll call a handle function named `handleChange`. This function grabs the `id` (set as `controlId` for the `<FormGroup>`) of the field being changed and updates its state with the value the user is typing in. Also, to have access to the `this` keyword inside `handleChange` we store the reference to an anonymous function like so: `handleChange = (event) => {}`.
4. We are setting the `autoFocus` flag for our email field, so that when our form loads, it sets focus to this field.
5. We also link up our submit button with our state by using a validate function called `validateForm`. This simply checks if our fields are non-empty, but can easily do something more complicated.
6. Finally, we trigger our callback `handleSubmit` when the form is submitted. For now we are simply suppressing the browsers default behavior on submit but we'll do more here later.

◆ CHANGE Let's add a couple of styles to this in the file `src/containers/Login.css`.

```
@media all and (min-width: 480px) {
  .Login {
    padding: 60px 0;
  }
}
```

```
.Login form {  
  margin: 0 auto;  
  max-width: 320px;  
}  
}
```

These styles roughly target any non-mobile screen sizes.

Add the Route

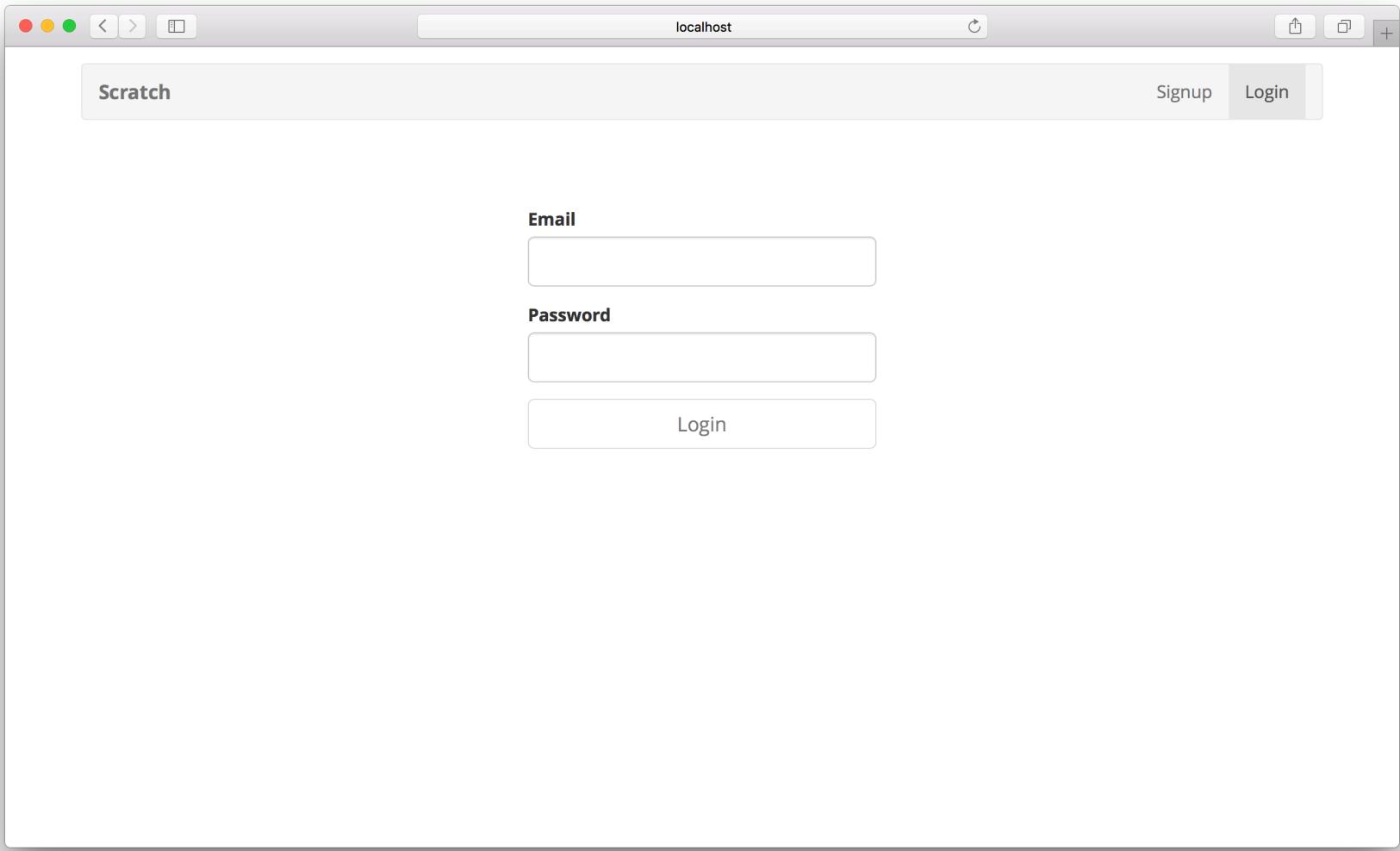
◆ CHANGE Now we link this container up with the rest of our app by adding the following line to `src/Routes.js` below our home `<Route>`.

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

◆ CHANGE And include our component in the header.

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

Now if we switch to our browser and navigate to the login page we should see our newly created form.



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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/37>)

For reference, here is the code so far

⌚ Frontend Source :create-a-login-page

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/create-a-login-page>)

Login with AWS Cognito

Before we link up our login form with our Amazon Cognito setup let's grab our Cognito details and load it into our application as a part of its config.

Load Cognito Details

◆ CHANGE Save the following into `src/config.js` and replace `YOUR_COGNITO_USER_POOL_ID` and `YOUR_COGNITO_APP_CLIENT_ID` with the Cognito Pool Id and App Client id from the Create a Cognito user pool ([/chapters/create-a-cognito-user-pool.html](#)) chapter.

```
export default {
  cognito: {
    USER_POOL_ID: "YOUR_COGNITO_USER_POOL_ID",
    APP_CLIENT_ID: "YOUR_COGNITO_APP_CLIENT_ID"
  }
};
```

◆ CHANGE And to load it into our login form simply import it by adding the following to the header of our Login container in `src/containers/Login.js`.

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

Login to Amazon Cognito

We are going to use the NPM module `amazon-cognito-identity-js` to login to Cognito.

◆ CHANGE Install it by running the following in your project root.

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

◆ CHANGE And include the following in the header of our `src/containers/Login.js`.

```
import {
  CognitoUserPool,
  AuthenticationDetails,
  CognitoUser
} from "amazon-cognito-identity-js";
```

The login code itself is relatively simple.

◆ CHANGE Add the following method to `src/containers/Login.js` as well.

```
login(email, password) {
  const userPool = new CognitoUserPool({
    UserPoolId: config.cognito.USER_POOL_ID,
    ClientId: config.cognito.APP_CLIENT_ID
  });

  const user = new CognitoUser({ Username: email, Pool: userPool });
  const authenticationData = { Username: email, Password: password };
  const authenticationDetails = new
    AuthenticationDetails(authenticationData);

  return new Promise((resolve, reject) =>
    user.authenticateUser(authenticationDetails, {
      onSuccess: result => resolve(),
      onFailure: err => reject(err)
    })
  );
}
```

This function does a few things for us:

1. It creates a new `CognitoUserPool` using the details from our config. And it creates a new `CognitoUser` using the email that is passed in.
2. It then authenticates our user using the authentication details with the `user.authenticateUser` method.
3. Since, the login call is asynchronous we return a `Promise` object. This way we can call this method directly without fidgeting with callbacks.

Trigger Login onSubmit

◆ CHANGE To connect the above `login` method to our form simply replace our placeholder `handleSubmit` method in `src/containers/Login.js` with the following.

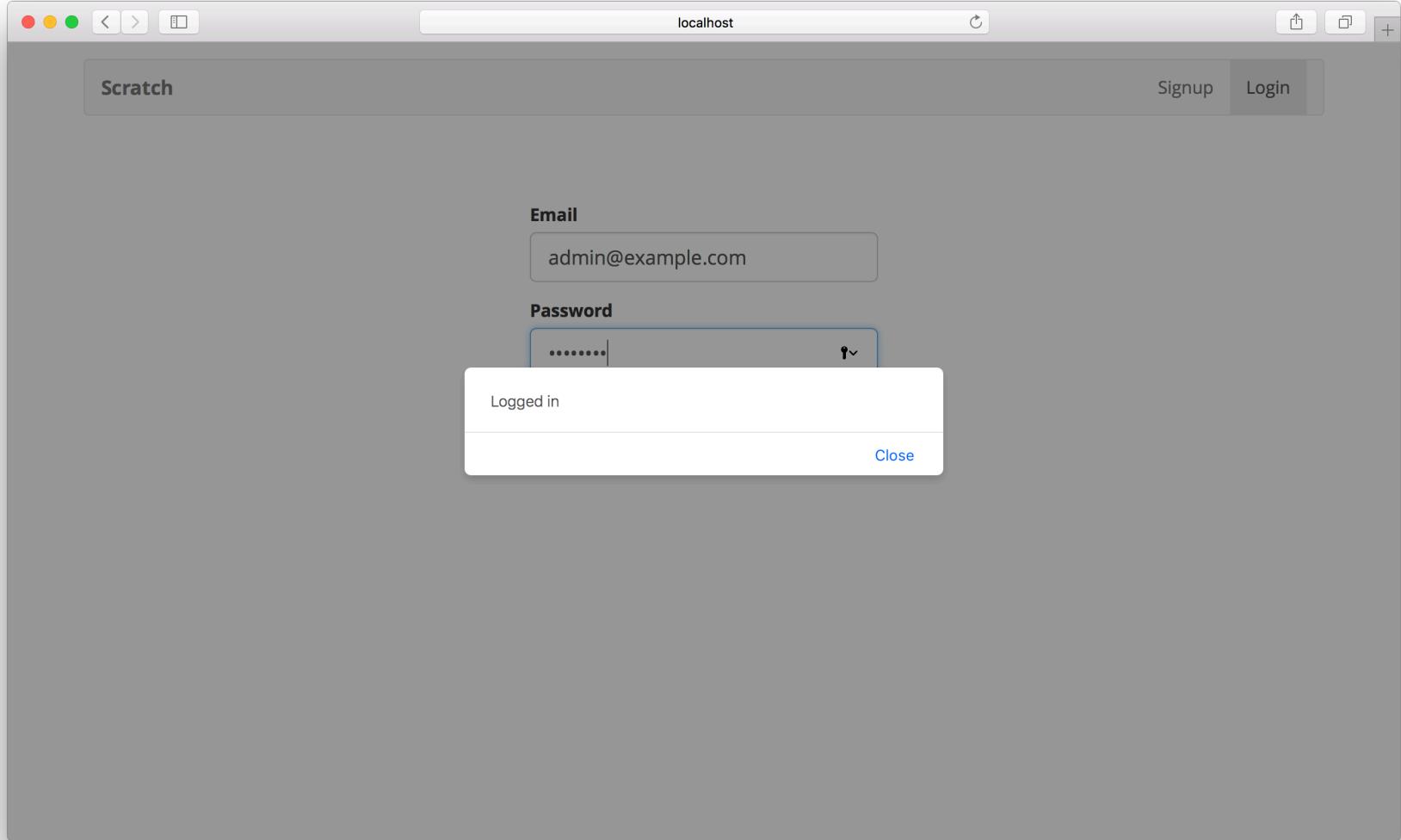
```
handleSubmit = async event => {
  event.preventDefault();

  try {
    await this.login(this.state.email, this.state.password);
    alert("Logged in");
  } catch (e) {
    alert(e);
  }
}
```

We are doing two things of note here.

1. We grab the `email` and `password` from `this.state` and call our `login` method with it.
2. We use the `await` keyword to invoke the `login` method that returns a promise. And we need to label our `handleSubmit` method as `async`.

Now if you try to login using the `admin@example.com` user (that we created in the Create a Cognito Test User (`/chapters/create-a-cognito-test-user.html`) chapter), you should see the browser alert that tells you that the login was successful.



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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/38>)

For reference, here is the code so far

⌚ Frontend Source :login-with-aws-cognito

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/login-with-aws-cognito>)

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.

◆ **CHANGE** Add the following to `src/App.js` right below the `class App extends Component {` line.

```
constructor(props) {
  super(props);

  this.state = {
    isAuthenticated: false
  };
}

userHasAuthenticated = authenticated => {
  this.setState({ isAuthenticated: authenticated });
}
```

This initializes the `isAuthenticated` flag in the App's state. And calling `userHasAuthenticated` updates it. But for the `Login` container to call this method we need to pass a reference of this method to it.

Pass the Session State to the Routes

We can do this by passing in a couple of props to the child component of the routes that the `App` component creates.

◆ CHANGE Add the following right below the `render() {` line in `src/App.js`.

```
const childProps = {
  isAuthenticated: this.state.isAuthenticated,
  userHasAuthenticated: this.userHasAuthenticated
};
```

◆ CHANGE And pass them into our `Routes` component by replacing the following line in the `render` method of `src/App.js`.

```
<Routes />
```

◆ CHANGE With this.

```
<Routes childProps={childProps} />
```

Currently, our `Routes` component does not do anything with the passed in `childProps`. We need it to apply these props to the child component it is going to render. In this case we need it to apply them to our `Login` component.

◆ CHANGE To do this, create a new component in `src/components/AppliedRoute.js` and add the following.

```
import React from "react";
import { Route } from "react-router-dom";

export default ({ component: C, props: cProps, ...rest }) =>
  <Route {...rest} render={props => <C {...props} {...cProps} />} />;
```

This simple component creates a `Route` where the child component that it renders contains the passed in props. Let's take a quick look at how this is being done.

- The `Route` component takes a prop called `component` that represents the component that will be rendered when a matching route is found. We want our `childProps` to be

sent to this component.

- The `Route` component can also take a `render` method in place of the `component`. This allows us to control what is passed in to our component.
- Based on this we can create a component that returns a `Route` and takes a `component` and `childProps` prop. This allows us to pass in the component we want rendered and the props that we want applied.
- Finally, we take `component` (set as `C`) and `props` (set as `cProps`) and render inside our `Route` using the inline function; `props => <C {...props} {...cProps} />`. Note, the `props` variable in this case is what the `Route` component passes us. Whereas, the `cProps` is the `childProps` that want to set.

Now to use this component, we are going to include it in the routes where we need to have the `childProps` passed in.

◆ **CHANGE** Replace the `export default () => (` method in `src/Routes.js` with the following.

```
export default ({ childProps }) =>
  <Switch>
    <AppliedRoute path="/" exact component={Home} props={childProps} />
    <AppliedRoute path="/login" exact component={Login} props={childProps} />
    { /* Finally, catch all unmatched routes */ }
    <Route component={NotFound} />
  </Switch>;
```

◆ **CHANGE** And import the new component in the header of `src/Routes.js`.

```
import AppliedRoute from "./components/AppliedRoute";
```

Now in the `Login` container we'll call the `userHasAuthenticated` method.

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

```
this.props.userHasAuthenticated(true);
```

Create a Logout Button

We can now use this to display a Logout button once the user logs in. Find the following in our `src/App.js`.

```
<RouteNavItem href="/signup">Signup</RouteNavItem>
<RouteNavItem href="/login">Login</RouteNavItem>
```

◆ CHANGE And replace it with this:

```
{this.state.isAuthenticated
? <NavItem onClick={this.handleLogout}>Logout</NavItem>
: [
  <RouteNavItem key={1} href="/signup">
    Signup
  </RouteNavItem>,
  <RouteNavItem key={2} href="/login">
    Login
  </RouteNavItem>
]}
```

Also, import the `NavItem` in the header.

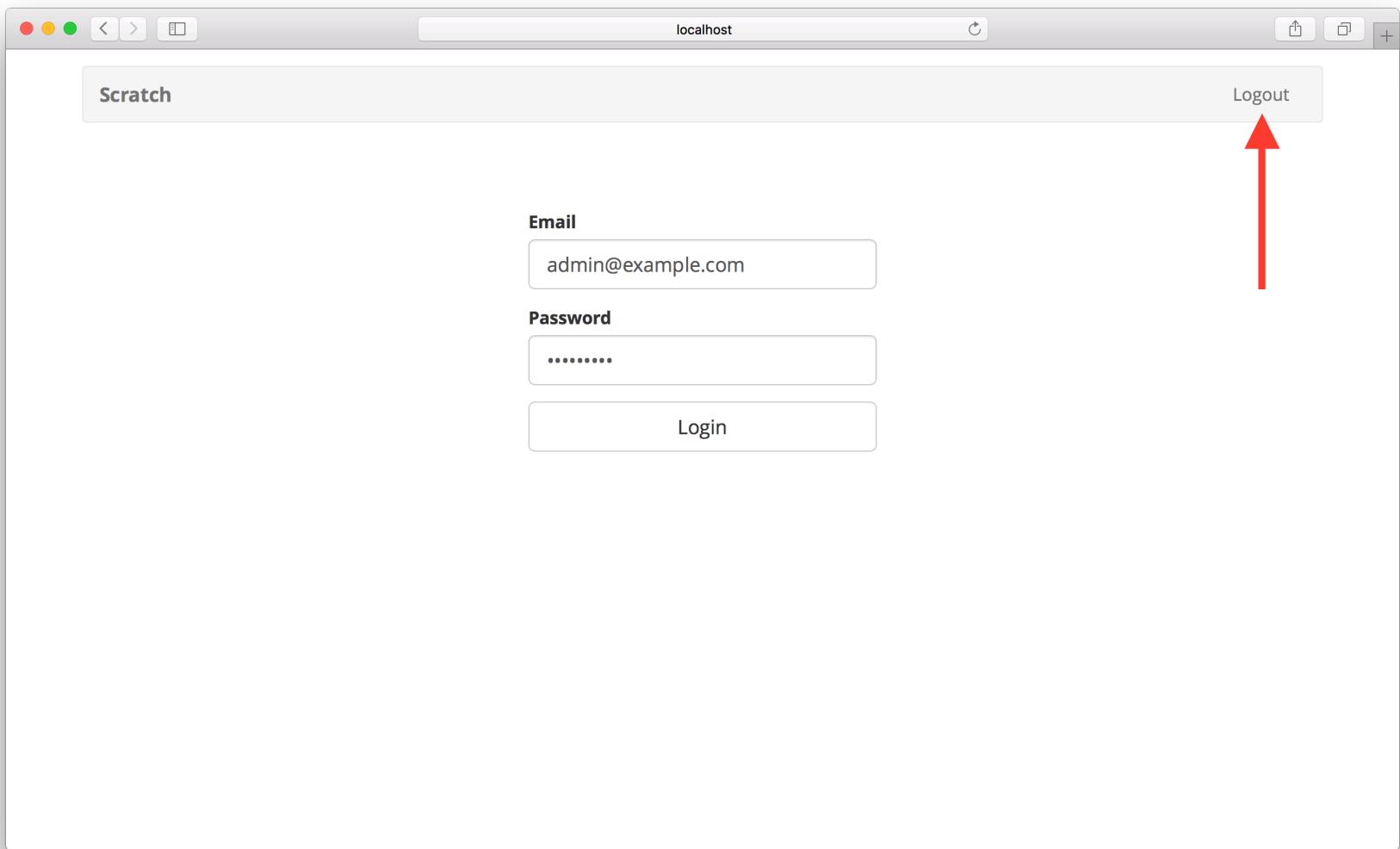
◆ CHANGE Replace the `react-bootstrap` import in the header of `src/App.js` with the following.

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

◆ CHANGE And add this `handleLogout` method to `src/App.js` above the `render() {` line as well.

```
handleLogout = event => {
  this.userHasAuthenticated(false);
}
```

Now head over to your browser and try logging in with the admin credentials we created in the Create a Cognito Test User (/chapters/create-a-cognito-test-user.html) chapter. You should see the Logout button appear right away.



Now if you refresh your page you should be logged out again. This is because we are not initializing the state from the browser session. Let's look at how to do that next.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/39>)

For reference, here is the code so far

⌚ Frontend Source :add-the-session-to-the-state

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/add-the-session-to-the-state>)

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 Cognito JS SDK does that for us automatically and we just need to read from it and load it into our application state.

Get Current User and Token

We are going to do this step a couple of times, so let's create a helper function for it.

◆ CHANGE Add the following to `src/libs/awsLib.js`. Make sure to create the `src/libs/` directory first.

```
import { CognitoUserPool } from "amazon-cognito-identity-js";
import config from "../config";

export async function authUser() {
  const currentUser = getCurrentUser();

  if (currentUser === null) {
    return false;
  }

  await getUserToken(currentUser);

  return true;
}

function getUserToken(currentUser) {
  return new Promise((resolve, reject) => {
    currentUser.getSession(function(err, session) {
      if (err) {
        reject(err);
      }
    });
  });
}
```

```

    return;
}

resolve(session.getIdToken().getJwtToken()));
});

});

}

function getCurrentUser() {
  const userPool = new CognitoUserPool({
    UserPoolId: config.cognito.USER_POOL_ID,
    ClientId: config.cognito.APP_CLIENT_ID
  });
  return userPool.getCurrentUser();
}

```

The `authUser` method is getting the current user from the Local Storage using the Cognito JS SDK. We then get that user's session and their user token in `getUserToken`. The `currentUser.getSession` also refreshes the user session in case it has expired. Finally in the `authUser` method we return `true` if we are able to authenticate the user and `false` if the user is not logged in.

Load User Session in to the State

Now that we can ensure the session user is authenticated using the `authUser` method, let's load this when our app loads. We are going to do this in `componentDidMount`. And since `authUser` is going to be called `async`; 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 a flag to our `src/App.js` state called `isAuthenticating`. The initial state in our `constructor` should look like the following.

```

this.state = {
  isAuthenticated: false,
  isAuthenticating: true
};

```

◆ **CHANGE** Let's include the `authUser` method that we created by adding it to the header of

`src/App.js`.

```
import { authUser } from "./libs/awsLib";
```

◆ CHANGE Now to load the user session we'll add the following to our `src/App.js`.

```
async componentDidMount() {
  try {
    if (await authUser()) {
      this.userHasAuthenticated(true);
    }
  } catch(e) {
    alert(e);
  }

  this.setState({ isAuthenticated: false });
}
```

All this does is check if there is a valid user in the session. It then updates the `isAuthenticating` flag once the process is complete.

Render When the State Is Ready

Since loading the user session is an asynchronous process, we want to ensure that our app does not change states when it first loads. To do this we'll hold off rendering our app till `isAuthenticating` is `false`.

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

◆ CHANGE Our `render` method in `src/App.js` should be as follows.

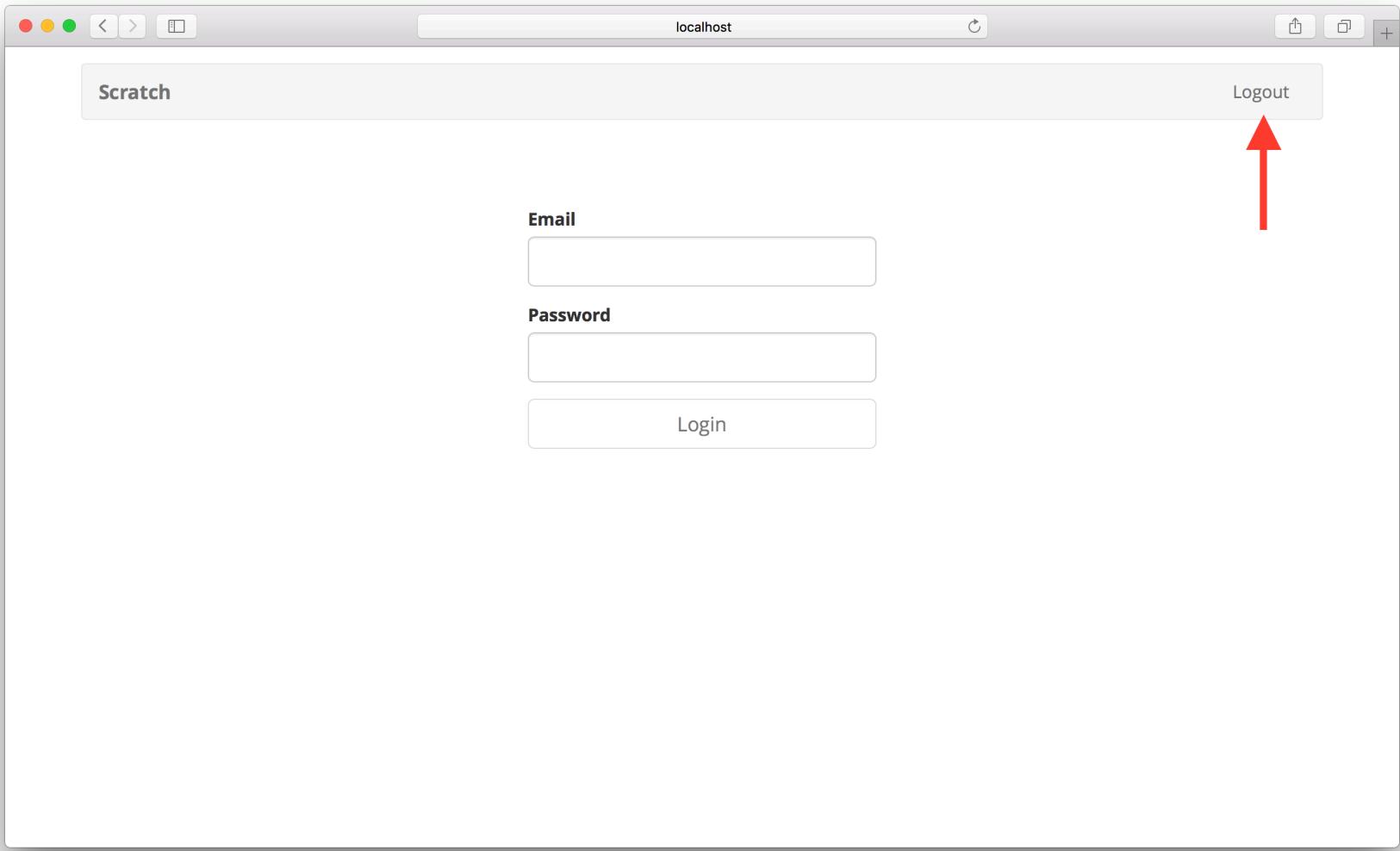
```
render() {
  const childProps = {
    isAuthenticated: this.state.isAuthenticated,
    userHasAuthenticated: this.userHasAuthenticated
  };
```

```

return (
  !this.state.isAuthenticated &&
  <div className="App container">
    <Navbar fluid collapseOnSelect>
      <Navbar.Header>
        <Navbar.Brand>
          <Link to="/">Scratch</Link>
        </Navbar.Brand>
        <Navbar.Toggle />
      </Navbar.Header>
      <Navbar.Collapse>
        <Nav pullRight>
          {this.state.isAuthenticated
            ? <NavItem onClick={this.handleLogout}>Logout</NavItem>
            : [
              <RouteNavItem key={1} href="/signup">
                Signup
              </RouteNavItem>,
              <RouteNavItem key={2} href="/login">
                Login
              </RouteNavItem>
            ]
          }
        </Nav>
      </Navbar.Collapse>
    </Navbar>
    <Routes childProps={childProps} />
  </div>
);
}

```

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



Unfortunately, when we hit Logout and refresh the page; we are still logged in. To fix this we are going to clear the session on logout next.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/40>)

For reference, here is the code so far

⌚ Frontend Source :load-the-state-from-the-session

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/load-the-state-from-the-session>)

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, in effect logging them back in.

◆ CHANGE Let's create a `signOutUser` method and add it to our `src/libs/awsLib.js`.

```
export function signOutUser() {
  const currentUser = getCurrentUser();

  if (currentUser !== null) {
    currentUser.signOut();
  }
}
```

Here we are using the AWS Cognito JS SDK to log the user out by calling `currentUser.signOut()`.

◆ CHANGE Next we'll include that in our `App` component. Replace the `import { authUser }` line in the header of `src/App.js` with:

```
import { authUser, signOutUser } from "./libs/awsLib";
```

◆ CHANGE And replace the `handleLogout` method in our `src/App.js` with this:

```
handleLogout = event => {
  signOutUser();

  this.userHasAuthenticated(false);
}
```

Now if you head over to your browser, logout and then refresh the page; you should be logged out completely.

If you try out the entire login flow from the beginning you'll notice that, we continue to stay on the login page through out the entire process. Next, we'll look at redirecting the page after we login and logout to make the flow make more sense.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/41>)

For reference, here is the code so far

🌐 Frontend Source :clear-the-session-on-logout

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/clear-the-session-on-logout>)

Redirect on Login and Logout

To complete the login flow we are going to need to do two more things.

1. Redirect the user to the homepage after they login.
2. And redirect them back to the login page after they logout.

We are going to use the `history.push` method that comes with React Router v4.

Redirect to Home on Login

Since our `Login` component is rendered using a `Route`, it adds the router props to it. So we can redirect using the `this.props.history.push` method.

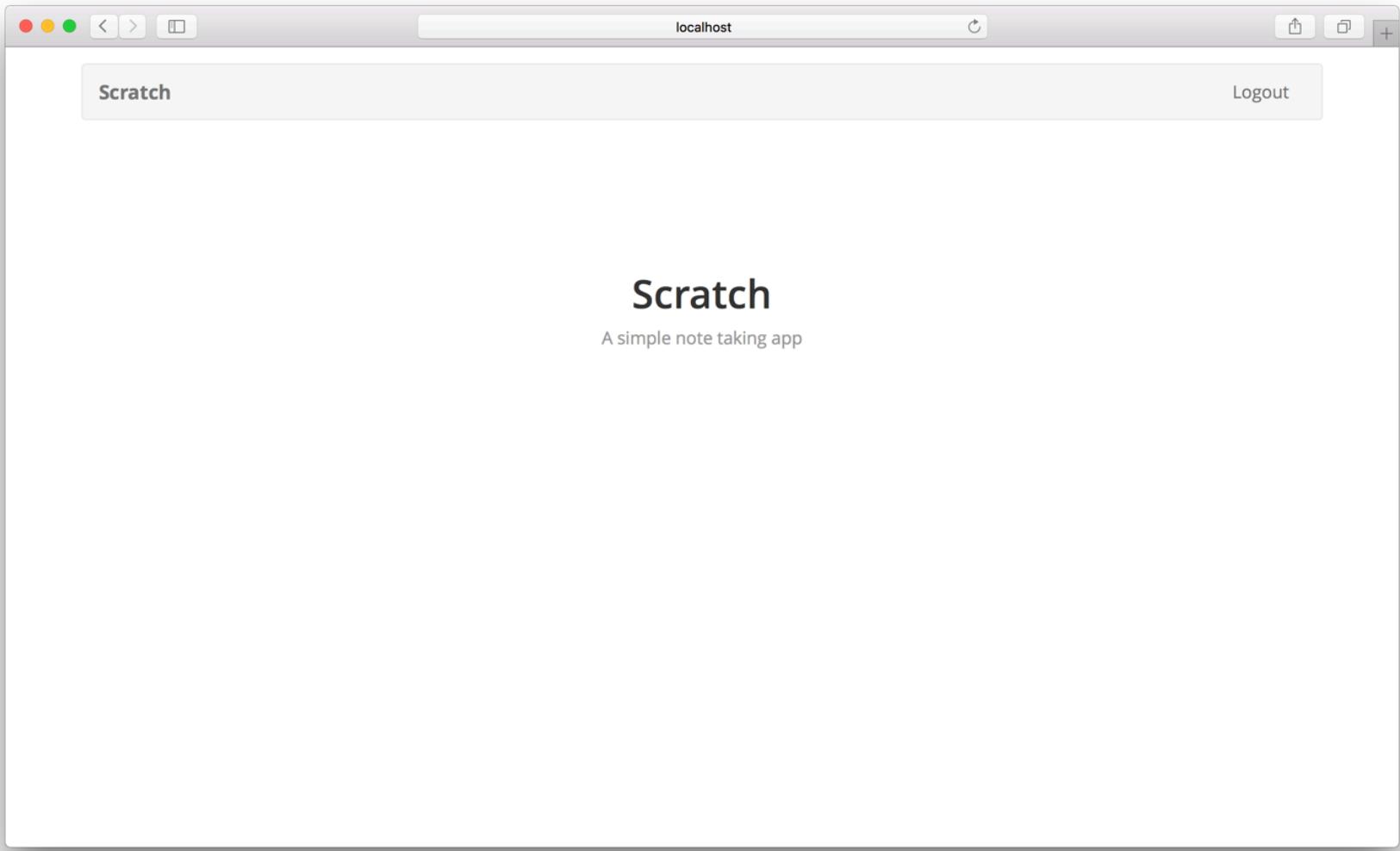
```
this.props.history.push("/");
```

◆ **CHANGE** Update the `handleSubmit` method in `src/containers/Login.js` to look like this:

```
handleSubmit = async event => {
  event.preventDefault();

  try {
    await this.login(this.state.email, this.state.password);
    this.props.userHasAuthenticated(true);
    this.props.history.push("/");
  } catch (e) {
    alert(e);
  }
}
```

Now if you head over to your browser and try logging in, you should be redirected to the homepage after you've been logged in.



Redirect to Login After Logout

Now we'll do something very similar for the logout process. However, the `App` component does not have access to the router props directly since it is not rendered inside a `Route` component. To be able to use the router props in our `App` component we will need to use the `withRouter` Higher-Order Component (<https://facebook.github.io/react/docs/higher-order-components.html>) (or HOC). You can read more about the `withRouter` HOC here (<https://reacttraining.com/react-router/web/api withRouter>).

To use this HOC, we'll change the way we export our `App` component.

◆ **CHANGE** Replace the following line in `src/App.js`.

```
export default App;
```

◆ **CHANGE** With this.

```
export default withRouter(App);
```

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

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

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

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

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

```
handleLogout = event => {
  signOutUser();

  this.userHasAuthenticated(false);

  this.props.history.push("/login");
}
```

This redirects us back to the login page once the user logs out.

Now if you switch over to your browser and try logging out, you should be redirected to the login page.

You might have noticed while testing this flow that since the login call has a bit of a delay, we might need to give some feedback to the user that the login call is in progress. Let's do that next.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/42>)

For reference, here is the code so far

🌐 Frontend Source :redirect-on-login-and-logout

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/redirect-on-login-and-logout>)

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 a isLoading Flag

◆ CHANGE To do this we are going to add a `isLoading` flag to the state of our `src/containers/Login.js`. So the initial state in the `constructor` looks like the following.

```
this.state = {  
  isLoading: false,  
  email: "",  
  password: ""  
};
```

◆ CHANGE And we'll update it while we are logging in. So our `handleSubmit` method now looks like so:

```
handleSubmit = async event => {  
  event.preventDefault();  
  
  this.setState({ isLoading: true });  
  
  try {  
    await this.login(this.state.email, this.state.password);  
    this.props.userHasAuthenticated(true);  
    this.props.history.push("/");  
  } catch (e) {  
    alert(e);  
  }  
  this.setState({ isLoading: 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 Add the following in `src/components/LoaderButton.js`.

```
import React from "react";
import { Button, Glyphicon } from "react-bootstrap";
import "./LoaderButton.css";

export default ({
  isLoading,
  text,
  loadingText,
  className = "",
  disabled = false,
  ...props
}) =>
  <Button
    className={`LoaderButton ${className}`}
    disabled={disabled || isLoading}
    {...props}>
    {isLoading && <Glyphicon glyph="refresh" className="spinning" />}
    {!isLoading ? text : loadingText}
  </Button>;
```

This is a really simple component that takes a `isLoading` flag and the text that the button displays in the two states (the default state and the loading state). The `disabled` prop is a result of what we have currently in our `Login` button. And we ensure that the button is disabled when `isLoading` is `true`. This makes it so that the user can't click it while we are in the process of logging them in.

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

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

```
.LoaderButton .spinning.glyphicon {  
  margin-right: 7px;  
  top: 2px;  
  animation: spin 1s infinite linear;  
}  
  
@keyframes spin {  
  from { transform: scale(1) rotate(0deg); }  
  to { transform: scale(1) rotate(360deg); }  
}
```

This spins the refresh Glyphicon infinitely with each spin taking a second. And by adding these styles as a part of the `LoaderButton` we keep them self contained within the component.

Render Using the isLoading Flag

Now we can use our new component in our `Login` container.

◆ CHANGE In `src/containers/Login.js` find the `<Button>` component in the `render` method.

```
<Button  
  block  
  bsSize="large"  
  disabled={!this.validateForm()}  
  type="submit"  
>  
  Login  
</Button>
```

◆ CHANGE And replace it with this.

```
<LoaderButton  
  block  
  bsSize="large"
```

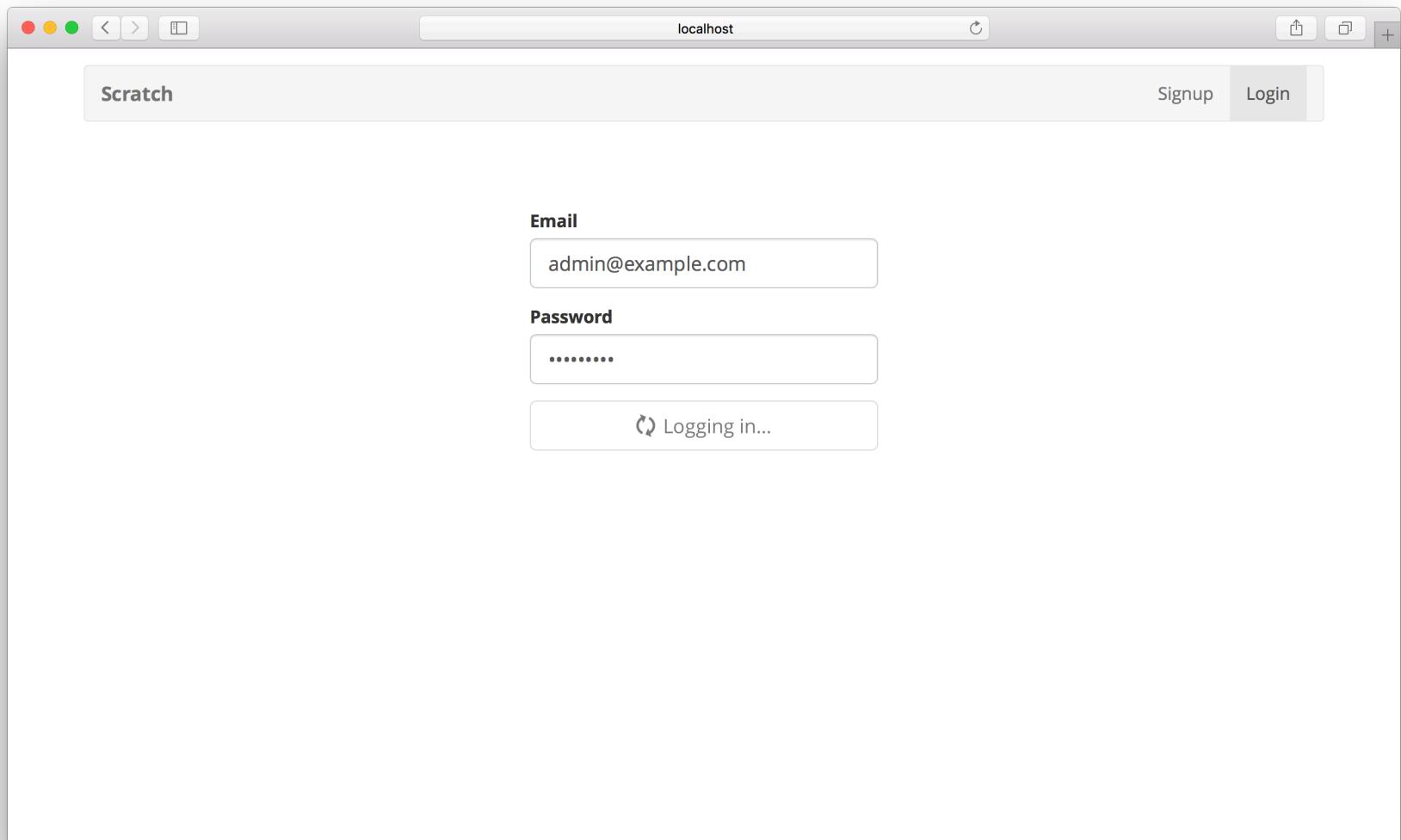
```
disabled={!this.validateForm()}

type="submit"
isLoading={this.state.isLoading}
text="Login"
loadingText="Logging in..."
/>>
```

◆ CHANGE Also, import the `LoaderButton` in the header. And remove the reference to the `Button` component.

```
import { FormGroup, FormControl, ControlLabel } from "react-
bootstrap";
import LoaderButton from "../components/LoaderButton";
```

And now when we switch over to the browser and try logging in, you should see the intermediate state before the login completes.



Next let's implement the sign up process for our app.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/43>)

For reference, here is the code so far

⌚ Frontend Source :give-feedback-while-logging-in

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/give-feedback-while-logging-in>)

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 using AWS Cognito 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 send the confirmation code to AWS Cognito.
5. We authenticate the newly created user.
6. Finally, we update the app state with the session.

So let's get started by creating the basic sign up form first.

For help and discussion

 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/44>)

Create the Signup Form

Let's start by creating the signup form that'll get the user's email and password.

Add the Container

◆ CHANGE Create a new container at `src/containers/Signup.js` with the following.

```
import React, { Component } from "react";
import {
  HelpBlock,
  FormGroup,
  FormControl,
  ControlLabel
} from "react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import "./Signup.css";

export default class Signup extends Component {
  constructor(props) {
    super(props);

    this.state = {
      isLoading: false,
      email: "",
      password: "",
      confirmPassword: "",
      confirmationCode: "",
      newUser: null
    };
  }

  validateForm() {
    return (
      <Form>
        <FormGroup controlId="email">
          <ControlLabel>Email</ControlLabel>
          <FormControl type="text" value={this.state.email}></FormControl>
          <HelpBlock>Please enter your email address.</HelpBlock>
        </FormGroup>
        <FormGroup controlId="password">
          <ControlLabel>Password</ControlLabel>
          <FormControl type="password" value={this.state.password}></FormControl>
          <HelpBlock>Please enter a password (at least 8 characters).</HelpBlock>
        </FormGroup>
        <FormGroup controlId="confirmPassword">
          <ControlLabel>Confirm Password</ControlLabel>
          <FormControl type="password" value={this.state.confirmPassword}></FormControl>
          <HelpBlock>Please re-enter your password.</HelpBlock>
        </FormGroup>
        <FormGroup controlId="confirmationCode">
          <ControlLabel>Confirmation Code</ControlLabel>
          <FormControl type="text" value={this.state.confirmationCode}></FormControl>
          <HelpBlock>Please enter the confirmation code sent to your email.</HelpBlock>
        </FormGroup>
        <LoaderButton
          buttonType="submit"
          onClick={this.handleSubmit}
          isLoading={this.state.isLoading}>Sign Up</LoaderButton>
      </Form>
    );
  }

  handleSubmit(event) {
    event.preventDefault();
    const validationErrors = this.validateForm();
    if (Object.keys(validationErrors).length === 0) {
      this.setState({ isLoading: true });
      // ...
    }
  }
}
```

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

validateConfirmationForm() {
    return this.state.confirmationCode.length > 0;
}

handleChange = event => {
    this.setState({
        [event.target.id]: event.target.value
    });
}

handleSubmit = async event => {
    event.preventDefault();

    this.setState({ isLoading: true });

    this.setState({ newUser: "test" });

    this.setState({ isLoading: false });
}

handleConfirmationSubmit = async event => {
    event.preventDefault();

    this.setState({ isLoading: true });
}

renderConfirmationForm() {
    return (
        <form onSubmit={this.handleConfirmationSubmit}>
            <FormGroup controlId="confirmationCode" bsSize="large">
                <ControlLabel>Confirmation Code</ControlLabel>
                <FormControl
                    type="text"
                    value={this.state.confirmationCode}
                    onChange={this.handleChange}
                />
            </FormGroup>
        </form>
    );
}
```

```

        autoFocus
        type="tel"
        value={this.state.confirmationCode}
        onChange={this.handleChange}
    />
    <HelpBlock>Please check your email for the code.</HelpBlock>
</FormGroup>
<LoaderButton
    block
    bsSize="large"
    disabled={!this.validateConfirmationForm()}
    type="submit"
    isLoading={this.state.isLoading}
    text="Verify"
    loadingText="Verifying..."
/>
</form>
);
}

```

```

renderForm() {
    return (
        <form onSubmit={this.handleSubmit}>
            <FormGroup controlId="email" bsSize="large">
                <ControlLabel>Email</ControlLabel>
                <FormControl
                    autoFocus
                    type="email"
                    value={this.state.email}
                    onChange={this.handleChange}
                />
            </FormGroup>
            <FormGroup controlId="password" bsSize="large">
                <ControlLabel>Password</ControlLabel>
                <FormControl
                    value={this.state.password}
                    onChange={this.handleChange}
                    type="password"
                />
            </FormGroup>
        </form>
    );
}

```

```

        />

    </FormGroup>
    <FormGroup controlId="confirmPassword" bsSize="large">
        <ControlLabel>Confirm Password</ControlLabel>
        <FormControl
            value={this.state.confirmPassword}
            onChange={this.handleChange}
            type="password"
        />
    </FormGroup>
    <LoaderButton
        block
        bsSize="large"
        disabled={!this.validateForm()}
        type="submit"
        isLoading={this.state.isLoading}
        text="Signup"
        loadingText="Signing up..."
    />
</form>
);
}

render() {
    return (
        <div className="Signup">
            {this.state.newUser === null
                ? this.renderForm()
                : this.renderConfirmationForm()}
        </div>
    );
}
}

```

Most of the things we are doing here are fairly straightforward but let's go over them quickly.

1. Since we need to show the user a form to enter the confirmation code, we are conditionally rendering two forms based on if we have a user object or not.

2. We are using the `LoaderButton` component that we created earlier for our submit buttons.
3. Since we have two forms we have two validation methods called `validateForm` and `validateConfirmationForm`.
4. We are setting the `autoFocus` flags on the email and the confirmation code fields.
5. For now our `handleSubmit` and `handleConfirmationSubmit` don't do a whole lot besides setting the `isLoading` state and a dummy value for the `newUser` state.

◆ **CHANGE** Also, let's add a couple of styles in `src/containers/Signup.css`.

```
@media all and (min-width: 480px) {  
  .Signup {  
    padding: 60px 0;  
  }  
  
  .Signup form {  
    margin: 0 auto;  
    max-width: 320px;  
  }  
}  
  
.Signup form span.help-block {  
  font-size: 14px;  
  padding-bottom: 10px;  
  color: #999;  
}
```

Add the Route

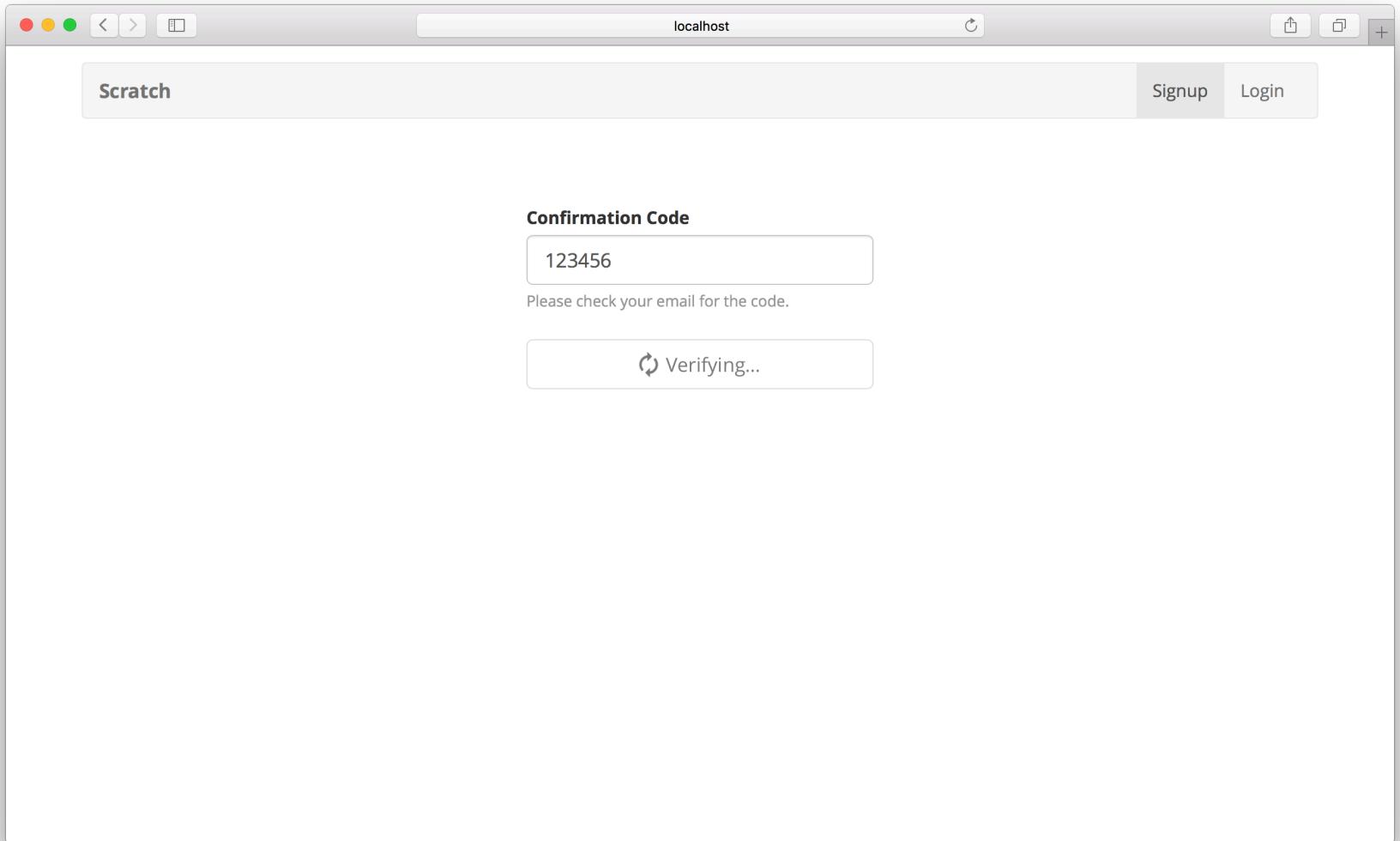
◆ **CHANGE** Finally, add our container as a route in `src/Routes.js` below our login route. We are using the `AppliedRoute` component that we created in the Add the session to the state (/chapters/add-the-session-to-the-state.html) chapter.

```
<AppliedRoute path="/signup" exact component={Signup} props={childProps} />
```

And include our component in the header.

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

Now if we switch to our browser and navigate to the signup page we should see our newly created form. Our form doesn't do anything when we enter in our info but you can still try to fill in an email address, password, and the confirmation code. It'll give you an idea of how the form will behave once we connect it to Cognito.



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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/45>)

For reference, here is the code so far

Frontend Source :create-the-signup-form
(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/create-the-signup-form>)

Signup with AWS Cognito

Now let's go ahead and implement the `handleSubmit` and `handleConfirmationSubmit` methods and connect it up with our AWS Cognito setup.

◆ CHANGE Replace our `handleSubmit` and `handleConfirmationSubmit` methods in `src/containers/Signup.js` with the following.

```
handleSubmit = async event => {
  event.preventDefault();

  this.setState({ isLoading: true });

  try {
    const newUser = await this.signup(this.state.email,
this.state.password);
    this.setState({
      newUser: newUser
    });
  } catch (e) {
    alert(e);
  }

  this.setState({ isLoading: false });
}

handleConfirmationSubmit = async event => {
  event.preventDefault();

  this.setState({ isLoading: true });

  try {
    await this.confirm(this.state.newUser,
this.state.confirmationCode);
  }
}
```

```
await this.authenticate(  
    this.state.newUser,  
    this.state.email,  
    this.state.password  
) ;  
  
this.props.userHasAuthenticated(true);  
this.props.history.push("/");  
} catch (e) {  
    alert(e);  
    this.setState({ isLoading: false });  
}  
}  
  
signup(email, password) {  
    const userPool = new CognitoUserPool({  
        UserPoolId: config.cognito.USER_POOL_ID,  
        ClientId: config.cognito.APP_CLIENT_ID  
    });  
  
    return new Promise((resolve, reject) =>  
        userPool.signUp(email, password, [], null, (err, result) => {  
            if (err) {  
                reject(err);  
                return;  
            }  
  
            resolve(result.user);  
        })  
    );  
}  
  
confirm(user, confirmationCode) {  
    return new Promise((resolve, reject) =>  
        user.confirmRegistration(confirmationCode, true, function(err,  
result) {  
            if (err) {  
                reject(err);  
            }  
        })  
    );  
}
```

```

    return;
}
resolve(result);
})
);
}

authenticate(user, email, password) {
  const authenticationData = {
    Username: email,
    Password: password
  };
  const authenticationDetails = new
AuthenticationDetails(authenticationData);

  return new Promise((resolve, reject) =>
    user.authenticateUser(authenticationDetails, {
      onSuccess: result => resolve(),
      onFailure: err => reject(err)
    })
  );
}

```

◆ CHANGE Also, include the following in our header.

```

import {
  AuthenticationDetails,
  CognitoUserPool
} from "amazon-cognito-identity-js";
import config from "../config";

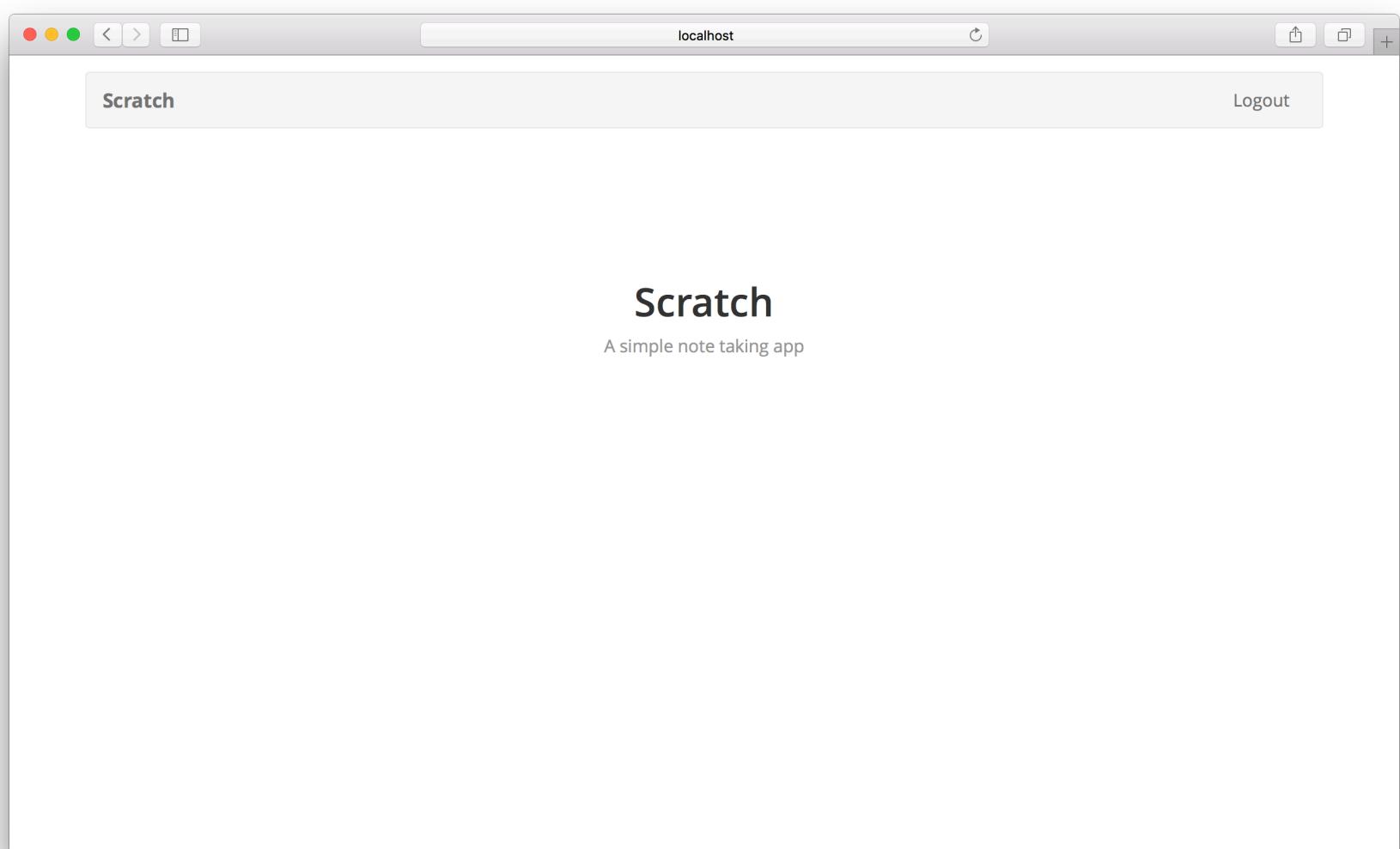
```

The flow here is pretty simple:

1. In `handleSubmit` we make a call to signup a user. This creates a new user object.
2. Save that user object to the state as `newUser`.
3. In `handleConfirmationSubmit` use the confirmation code to confirm the user.

- With the user now confirmed, Cognito now knows that we have a new user that can login to our app.
- Use the email and password to authenticate the newly created user using the `newUser` object that we had previously saved in the state.
- Update the App's state using the `userHasAuthenticated` method.
- Finally, redirect to the homepage.

Now if you were to switch over to your browser and try signing up for a new account it should redirect you to the homepage after sign up successfully completes.



A quick note on the signup flow here. If the user refreshes their page at the confirm step, they won't be able to get back and confirm that account. It forces them to create a new account instead. We are keeping things intentionally simple here but you can fix this by creating a separate page that handles the confirm step based on the email address. Here (<http://docs.aws.amazon.com/cognito/latest/developerguide/using-amazon-cognito-user-identity-pools-javascript-examples.html#using-amazon-cognito-identity-user-pools-javascript-example-confirming-user>) is some sample code that you can use to confirm an unauthenticated user.

However, 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 us-east-1 \
--user-pool-id YOUR_USER_POOL_ID \
--username YOUR_USER_EMAIL
```

Just be sure to use your Cognito User Pool Id and the email you used to create the account.

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/46>)

For reference, here is the code so far

🌐 Frontend Source :signup-with-aws-cognito

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/signup-with-aws-cognito>)

Add the Create Note Page

Now that we can signup users and also log them in. Let's get started with the most important part of our note taking app; the creation of a note.

First we are going to create the form for a note. It'll take some content and a file as an attachment.

Add the Container

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

```
import React, { Component } from "react";
import { FormGroup, FormControl, ControlLabel } from "reactstrap";
import LoaderButton from "../components/LoaderButton";
import config from "../config";
import "./NewNote.css";

export default class NewNote extends Component {
  constructor(props) {
    super(props);

    this.file = null;

    this.state = {
      isLoading: null,
      content: ""
    };
  }

  validateForm() {
    return this.state.content.length > 0;
  }
}
```

```
handleChange = event => {
  this.setState({
    [event.target.id]: event.target.value
  });
}

handleFileChange = event => {
  this.file = event.target.files[0];
}

handleSubmit = async event => {
  event.preventDefault();

  if (this.file && this.file.size > config.MAX_ATTACHMENT_SIZE) {
    alert("Please pick a file smaller than 5MB");
    return;
  }

  this.setState({ isLoading: true });
}

render() {
  return (
    <div className="NewNote">
      <form onSubmit={this.handleSubmit}>
        <FormGroup controlId="content">
          <FormControl
            onChange={this.handleChange}
            value={this.state.content}
            componentClass="textarea"
          />
        </FormGroup>
        <FormGroup controlId="file">
          <ControlLabel>Attachment</ControlLabel>
          <FormControl onChange={this.handleFileChange} type="file">
        </FormGroup>
    
```

```

<LoaderButton
  block
  bsStyle="primary"
  bsSize="large"
  disabled={!this.validateForm()}
  type="submit"
  isLoading={this.state.isLoading}
  text="Create"
  loadingText="Creating..."
/>
</form>
</div>
);
}
}

```

Everything is fairly standard here, except for the file input. Our form elements so far have been controlled components (<https://facebook.github.io/react/docs/forms.html>), as in their value is directly controlled by the state of the component. The file input simply calls a different `onChange` handler (`handleFileChange`) that saves the file object as a class property. We use a class property instead of saving it in the state because the file object we save does not change or drive the rendering of our component.

Currently, our `handleSubmit` does not do a whole lot other than limiting the file size of our attachment. We are going to define this in our config.

◆ CHANGE So add the following to our `src/config.js` below the `export default {` line.

```
MAX_ATTACHMENT_SIZE: 5000000,
```

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

```

.NewNote form {
  padding-bottom: 15px;
}

.NewNote form textarea {
  height: 300px;
}

```

```
font-size: 24px;
```

```
}
```

Add the Route

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

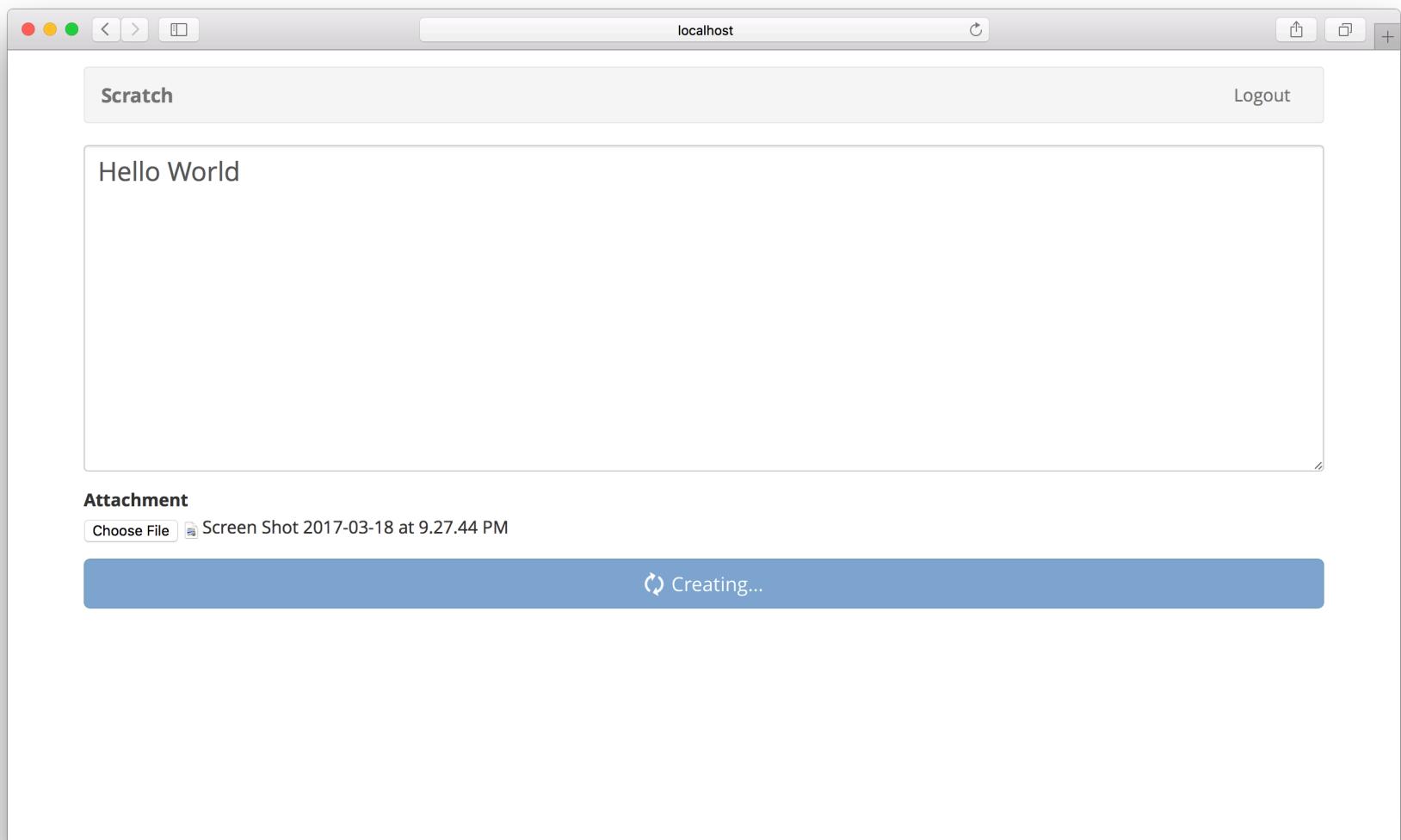
We are using the `AppliedRoute` component that we created in the Add the session to the state (/chapters/add-the-session-to-the-state.html) chapter.

```
<AppliedRoute path="/notes/new" exact component={NewNote} props={childProps} />
```

◆ CHANGE And include our component in the header.

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

Now if we switch to our browser and navigate `http://localhost:3000/notes/new` we should see our newly created form. Try adding some content, uploading a file, and hitting submit to see it in action.



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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/47>)

For reference, here is the code so far

🌐 Frontend Source :add-the-create-note-page

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/add-the-create-note-page>)

Connect to API Gateway with IAM Auth

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. As we had done while testing our APIs, we need to follow these steps.

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

In our React app we do step 1 by calling the `authUser` method when the App component loads. So let's do step 2 and use the `userToken` to generate temporary IAM credentials.

Generate Temporary IAM Credentials

Our authenticated users can get a set of temporary IAM credentials to access the AWS resources that we've previously specified. We can do this using the AWS JS SDK.

◆ CHANGE Install it by running the following in your project root.

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

◆ CHANGE Let's add a helper function in `src/libs/awsLib.js`.

```
function getAwsCredentials(userToken) {
  const authenticator = `cognito-idp.${config.cognito
    .REGION}.amazonaws.com/${config.cognito.USER_POOL_ID}`;
  AWS.config.update({ region: config.cognito.REGION });

  AWS.config.credentials = new AWS.CognitoIdentityCredentials({
```

```
IdentityPoolId: config.cognito.IDENTITY_POOL_ID,  
Logins: {  
  [authenticator]: userToken  
}  
});  
  
return AWS.config.credentials.getPromise();  
}
```

This method takes the `userToken` and uses our Cognito User Pool as the authenticator to request a set of temporary credentials.

◆ CHANGE Also include the **AWS SDK** in our header.

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

◆ CHANGE To get our AWS credentials we need to add the following to our `src/config.js` in the `cognito` block. Make sure to replace `YOUR_IDENTITY_POOL_ID` with your **Identity pool ID** from the Create a Cognito identity pool (`/chapters/create-a-cognito-identity-pool.html`) chapter and `YOUR_COGNITO_REGION` with the region your Cognito User Pool is in.

```
REGION: "YOUR_COGNITO_REGION",  
IDENTITY_POOL_ID: "YOUR_IDENTITY_POOL_ID",
```

Now let's use the `getAwsCredentials` helper function.

◆ CHANGE Replace the `authUser` in `src/libs/awsLib.js` with the following:

```
export async function authUser() {  
  if (  
    AWS.config.credentials &&  
    Date.now() < AWS.config.credentials.expireTime - 60000  
  ) {  
    return true;  
  }  
  
  const currentUser = getCurrentUser();
```

```
if (currentUser === null) {  
    return false;  
}  
  
const userToken = await getToken(currentUser);  
  
await getAwsCredentials(userToken);  
  
return true;  
}
```

We are passing `getAwsCredentials` the `userToken` that Cognito gives us to generate the temporary credentials. These credentials are valid till the `AWS.config.credentials.expireTime`. So we simply check to ensure our credentials are still valid before requesting a new set. This also ensures that we don't generate the `userToken` every time the `authUser` method is called.

Next let's sign our request using Signature Version 4.

Sign API Gateway Requests with Signature Version 4

All secure AWS API requests need to be signed using Signature Version 4 (<http://docs.aws.amazon.com/general/latest/gr/signature-version-4.html>). We could use API Gateway to generate an SDK and use that to make our requests. But that can be a bit annoying to use during development since we would need to regenerate it every time we made a change to our API. So we re-worked the generated SDK to make a little helper function that can sign the requests for us.

To create this signature we are going to need the Crypto NPM package.

◆ CHANGE Install it by running the following in your project root.

```
$ npm install crypto-js --save
```

◆ CHANGE Copy the following file to `src/libs/sigV4Client.js`.

→ `sigV4Client.js` (<https://raw.githubusercontent.com/AnomalyInnovations/serverless-stack-demo-client/master/src/libs/sigV4Client.js>)

This file can look a bit intimidating at first but it is just using the temporary credentials and the request parameters to create the necessary signed headers. To create a new `sigV4Client` we need to pass in the following:

```
// Pseudocode

sigV4Client.newClient({
    // Your AWS temporary access key
    accessKey,
    // Your AWS temporary secret key
    secretKey,
    // Your AWS temporary session token
    sessionToken,
    // API Gateway region
    region,
    // API Gateway URL
    endpoint
});
```

And to sign a request you need to use the `signRequest` method and pass in:

```
// Pseudocode

const signedRequest = client.signRequest({
    // The HTTP method
    method,
    // The request path
    path,
    // The request headers
    headers,
    // The request query parameters
    queryParams,
    // The request body
    body
});
```

And `signedRequest.headers` should give you the signed headers that you need to make

the request.

Now let's go ahead and use the `sigV4Client` and invoke API Gateway.

Call API Gateway

We are going to call the code from above to make our request. Let's write a helper function to do that.

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

```
export async function invokeApig({
  path,
  method = "GET",
  headers = {},
  queryParams = {},
  body
}) {
  if (!await authUser()) {
    throw new Error("User is not logged in");
  }

  const signedRequest = sigV4Client
    .newClient({
      accessKey: AWS.config.credentials.accessKeyId,
      secretKey: AWS.config.credentials.secretAccessKey,
      sessionToken: AWS.config.credentials.sessionToken,
      region: config.apiGateway.REGION,
      endpoint: config.apiGateway.URL
    })
    .signRequest({
      method,
      path,
      headers,
      queryParams,
      body
  });
}
```

```

body = body ? JSON.stringify(body) : body;
headers = signedRequest.headers;

const results = await fetch(signedRequest.url, {
  method,
  headers,
  body
}) ;

if (results.status !== 200) {
  throw new Error(await results.text());
}

return results.json();
}

```

We are simply following the steps to make a signed request to API Gateway here. We first ensure the user is authenticated and we generate their temporary credentials using `authUser`. Then using the `sigV4Client` we sign our request. We then use the signed headers to make a HTTP `fetch` request.

◆ CHANGE Include the `sigV4Client` by adding this to the header of our file.

```
import sigV4Client from "./sigV4Client";
```

◆ CHANGE Also, add the details of our API to `src/config.js` above the `cognito: {` line. Remember to replace `YOUR_API_GATEWAY_URL` and `YOUR_API_GATEWAY_REGION` with the ones from the Deploy the APIs (/chapters/deploy-the-apis.html) chapter.

```

apiGateway: {
  URL: "YOUR_API_GATEWAY_URL",
  REGION: "YOUR_API_GATEWAY_REGION"
},

```

In our case the URL is `https://1y55wbovg4.execute-api.us-east-1.amazonaws.com/prod` and the region is `us-east-1`.

We are now ready to use this to make a request to our create note API.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/113>)

For reference, here is the code so far

💡 Frontend Source :connect-to-api-gateway-with-iam-auth

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/connect-to-api-gateway-with-iam-auth>)

Call the Create API

Now that we know how to connect to API Gateway securely, let's make the API call to create our note.

◆ CHANGE Let's include our `awsLib` by adding the following to the header of `src/containers/NewNote.js`.

```
import { invokeApig } from "../libs/awsLib";
```

◆ CHANGE And replace our `handleSubmit` function with the following.

```
handleSubmit = async event => {
  event.preventDefault();

  if (this.file && this.file.size > config.MAX_ATTACHMENT_SIZE) {
    alert("Please pick a file smaller than 5MB");
    return;
  }

  this.setState({ isLoading: true });

  try {
    await this.createNote({
      content: this.state.content
    });
    this.props.history.push("/");
  } catch (e) {
    alert(e);
    this.setState({ isLoading: false });
  }
}

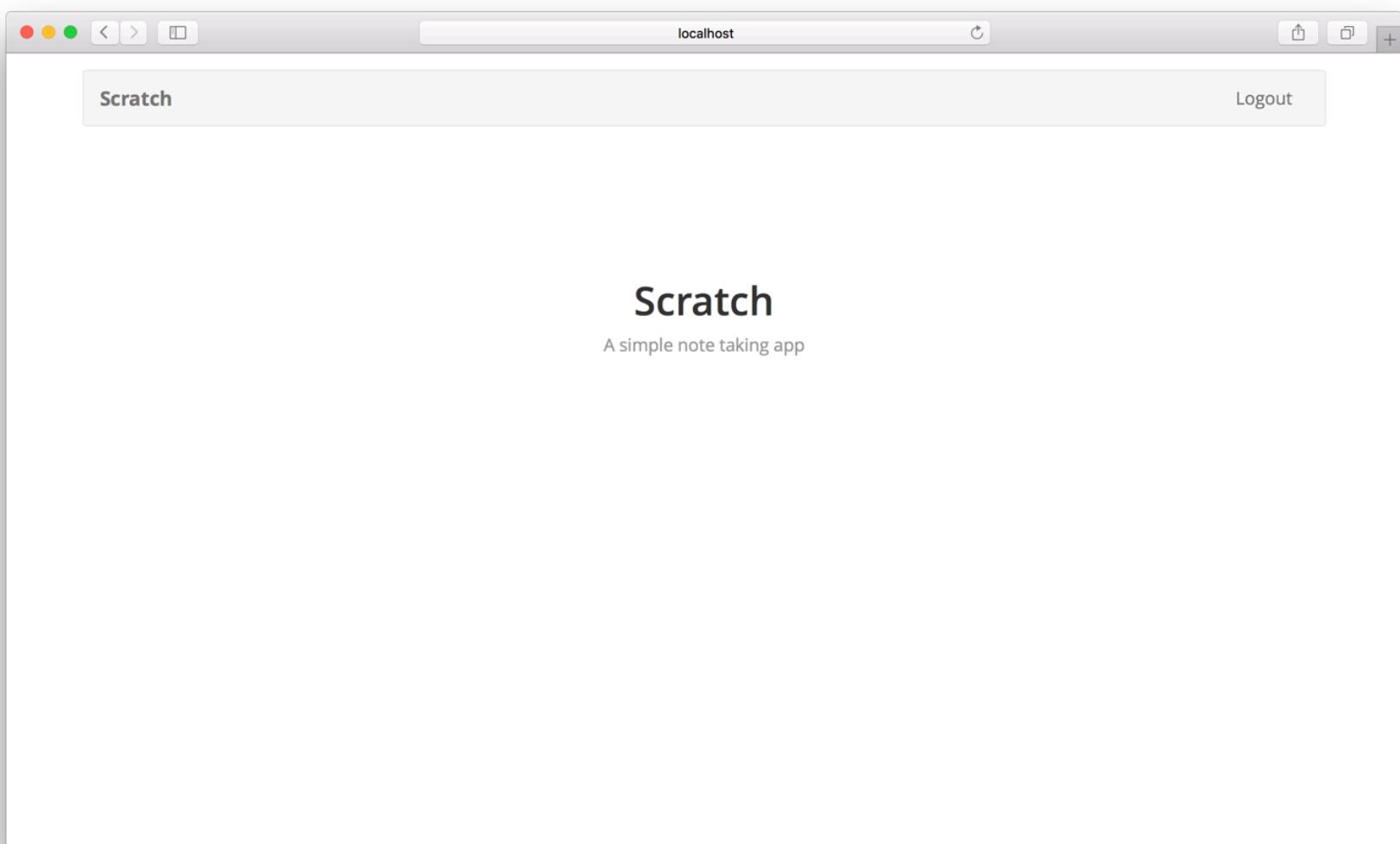
createNote(note) {
```

```
return invokeApig({
  path: "/notes",
  method: "POST",
  body: note
}) ;
}
```

This does a couple of simple things.

1. We make our create call in `createNote` by making a POST request to `/notes` and passing in our note object.
2. For now the note object is simply the content of the note. We are creating these notes without an attachment for now.
3. Finally, after the note is created we redirect to our homepage.

And that's it; if you switch over to your browser and try submitting your form, it should successfully navigate over to our homepage.



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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/48>)

For reference, here is the code so far

⌚ Frontend Source :call-the-create-api

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/call-the-create-api>)

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 space and we get a URL back.
3. Create a note with the file URL as the attachment.

We are going to use the AWS JS SDK to upload our files to S3. The S3 Bucket that we created previously, is secured using our Cognito Identity Pool. So before we can upload a file we should ensure that our user is authenticated and has a set of temporary IAM credentials. This is exactly the same process as when we were making secured requests to our API in the Connect to API Gateway with IAM auth (/chapters/connect-to-api-gateway-with-iam-auth.html) chapter.

Upload to S3

◆ CHANGE Append the following in `src/libs/awsLib.js`.

```
export async function s3Upload(file) {
  if (!await authUser()) {
    throw new Error("User is not logged in");
  }

  const s3 = new AWS.S3({
    params: {
      Bucket: config.s3.BUCKET
    }
  });
  const filename = `${AWS.config.credentials
    .identityId}-${Date.now()}-${file.name}`;

  return s3
    .upload({
      Key: filename,
```

```
    Body: file,
    ContentType: file.type,
    ACL: "public-read"
  })
  .promise();
}
```

◆ CHANGE And add this to our `src/config.js` above the `apiGateway` block. Make sure to replace `YOUR_S3_UPLOADS_BUCKET_NAME` with the your S3 Bucket name from the Create an S3 bucket for file uploads (/chapters/create-an-s3-bucket-for-file-uploads.html) chapter.

```
s3: {
  BUCKET: "YOUR_S3_UPLOADS_BUCKET_NAME"
},
```

The above method does a couple of things.

1. It takes a file object as a parameter.
2. Generates a unique file name prefixed with the `identityId`. This is necessary to secure the files on a per-user basis.
3. Upload the file to S3 and set its permissions to `public-read` to ensure that we can download it later.
4. And return a Promise object.

Upload Before Creating a Note

Now that we have our upload methods ready, let's call them from the create note method.

◆ CHANGE Replace the `handleSubmit` method in `src/containers/NewNote.js` with the following.

```
handleSubmit = async event => {
  event.preventDefault();

  if (this.file && this.file.size > config.MAX_ATTACHMENT_SIZE) {
    alert("Please pick a file smaller than 5MB");
  }
}
```

```
return;

}

this.setState({ isLoading: true });

try {
  const uploadedFilename = this.file
    ? (await s3Upload(this.file)).Location
    : null;

  await this.createNote({
    content: this.state.content,
    attachment: uploadedFilename
  });
  this.props.history.push("/");
} catch (e) {
  alert(e);
  this.setState({ isLoading: false });
}
}
```

◆ CHANGE And make sure to include `s3Upload` in the header by replacing the `import { invokeApig }` line with this:

```
import { invokeApig, s3Upload } from "../libs/awsLib";
```

The change we've made in the `handleSubmit` is that:

1. We upload the file using `s3Upload`.
2. Use the returned URL 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 make sure we clear out AWS credentials that are cached by the AWS JS SDK before we move on.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/49>)

For reference, here is the code so far

💡 Frontend Source :upload-a-file-to-s3

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/upload-a-file-to-s3>)

Clear AWS Credentials Cache

To be able to upload our files to S3 we needed to get the AWS credentials first. And the AWS JS SDK saves those credentials in our browser's Local Storage.

But we need to make sure that we clear out those credentials when we logout. If we don't, the next user that logs in on the same browser, might end up with the incorrect credentials.

◆ CHANGE To do that let's replace the `signOutUser` method in our `src/libs/awsLib.js` with this:

```
export function signOutUser() {
  const currentUser = getCurrentUser();

  if (currentUser !== null) {
    currentUser.signOut();
  }

  if (AWS.config.credentials) {
    AWS.config.credentials.clearCachedId();
    AWS.config.credentials = new AWS.CognitoIdentityCredentials({});
  }
}
```

Here we are clearing the AWS JS SDK cache and resetting the credentials that it saves in the browser's Local Storage.

Next up we are going to allow users to see a list of the notes they've created.

For help and discussion

Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack->

com/issues/50)

For reference, here is the code so far

🔗 Frontend Source :clear-aws-credentials-cache
(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/clear-aws-credentials-cache>)

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 containers is very simple. Let's add the conditional rendering in there.

◆ CHANGE Replace our `src/containers/Home.js` with the following.

```
import React, { Component } from "react";
import { PageHeader, ListGroup } from "react-bootstrap";
import "./Home.css";

export default class Home extends Component {
  constructor(props) {
    super(props);

    this.state = {
      isLoading: true,
      notes: []
    };
  }

  renderNotesList(notes) {
    return null;
  }

  renderLander() {
    return (
      <div className="lander">
        <h1>Scratch</h1>
        <p>A simple note taking app</p>
    
```

```

        </div>
    );
}

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

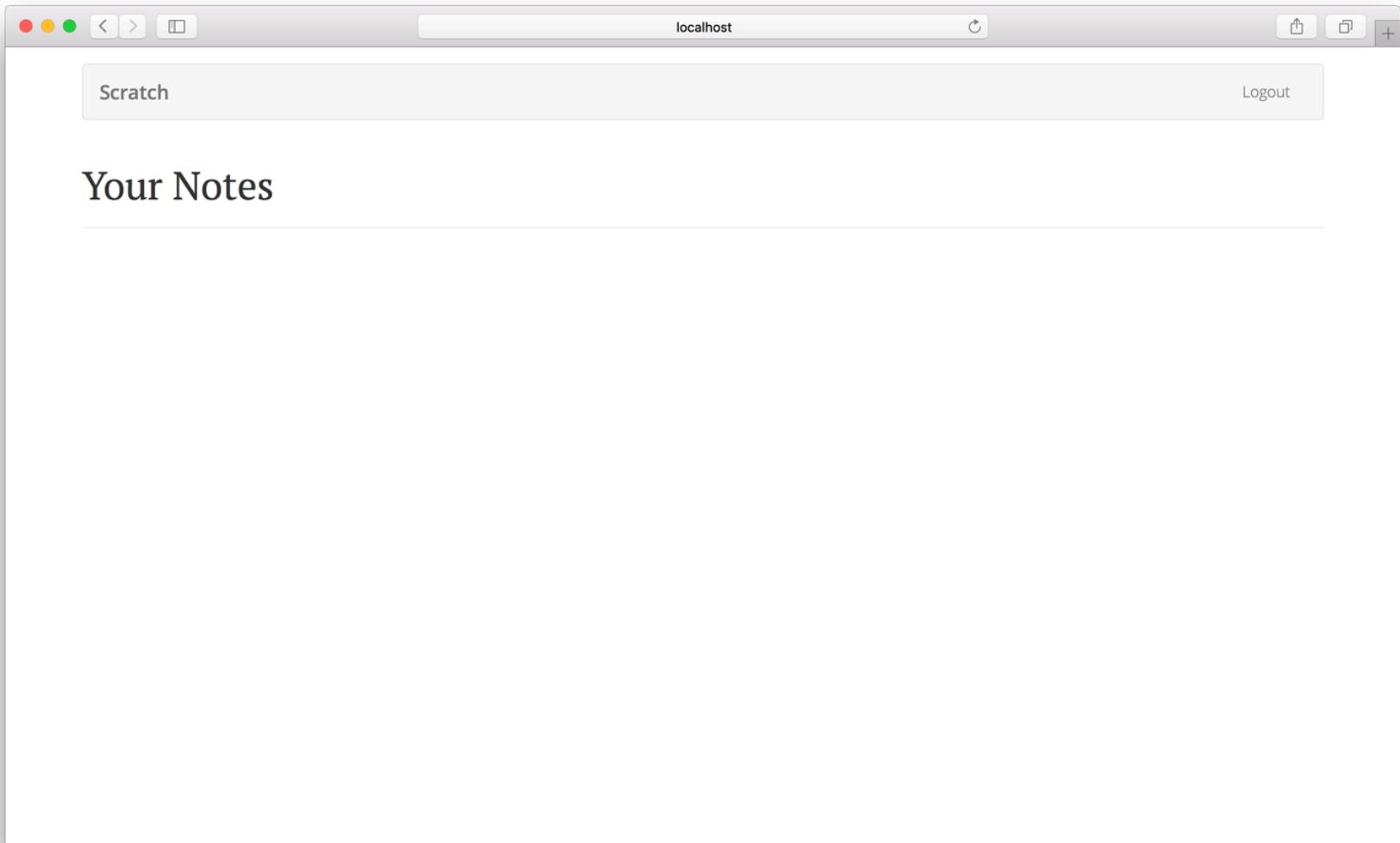
render() {
    return (
        <div className="Home">
            {this.props.isAuthenticated ? this.renderNotes() :
this.renderLander()}
        </div>
    );
}
}

```

We are doing a few things of note here:

1. Rendering the lander or the list of notes based on `this.props.isAuthenticated`.
2. Store our notes in the state. Currently, it's empty but we'll be calling our API for it.
3. Once we fetch our list we'll use the `renderNotesList` method to render the items in the list.

And that's our basic setup! Head over to the browser and the homepage of our app should render out an empty list.



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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/51>)

For reference, here is the code so far

🌐 Frontend Source :list-all-the-notes

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/list-all-the-notes>)

Call the List API

Now that we have our basic homepage set up, let's make the API call to render our list of notes.

Make the Request

◆ CHANGE Add the following below the `constructor` block in `src/containers/Home.js`.

```
async componentDidMount() {
  if (!this.props.isAuthenticated) {
    return;
  }

  try {
    const results = await this.notes();
    this.setState({ notes: results });
  } catch (e) {
    alert(e);
  }

  this.setState({ isLoading: false });
}

notes() {
  return invokeApig({ path: "/notes" });
}
```

◆ CHANGE And include our API Gateway Client helper in the header.

```
import { invokeApig } from '../libs/awsLib';
```

All this does, is make a GET request to `/notes` on `componentDidMount` and puts the results in the `notes` object in the state.

Now let's render the results.

Render the List

◆ CHANGE Replace our `renderNotesList` placeholder method with the following.

```
renderNotesList(notes) {  
  return [{}].concat(notes).map(  
    (note, i) =>  
      i !== 0  
        ? <ListGroupItem  
            key={note.noteId}  
            href={`/notes/${note.noteId}`}  
            onClick={this.handleNoteClick}  
            header={note.content.trim().split("\n")[0]}  
          >  
            {"Created: " + new Date(note.createdAt).toLocaleString()}  
          </ListGroupItem>  
        : <ListGroupItem  
            key="new"  
            href="/notes/new"  
            onClick={this.handleNoteClick}  
          >  
            <h4>  
              <b>{\u26a1}</b> Create a new note  
            </h4>  
          </ListGroupItem>  
    );  
  }  
  
handleNoteClick = event => {  
  event.preventDefault();  
  this.props.history.push(event.currentTarget.getAttribute("href"));  
}
```

◆ CHANGE And include the `ListGroupItem` in the header so that our `react-bootstrap` import looks like so.

```
import { PageHeader, ListGroup, ListGroupItem } from "react-bootstrap";
```

The code above does a few things.

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

◆ **CHANGE** Let's also add a couple of styles to our `src/containers/Home.css`.

```
.Home .notes h4 {  
  font-family: "Open Sans", sans-serif;  
  font-weight: 600;  
  overflow: hidden;  
  line-height: 1.5;  
  white-space: nowrap;  
  text-overflow: ellipsis;  
}  
.Home .notes p {  
  color: #666;  
}
```

Now head over to your browser and you should see your list displayed.

The screenshot shows a web browser window with the URL 'localhost' in the address bar. The page title is 'Scratch'. In the top right corner, there is a 'Logout' button. The main content area is titled 'Your Notes' and contains a button '+ Create a new note'. Below it, there is a single note card with the title 'Hello World' and the creation timestamp 'Created: 2/14/2017, 4:18:38 PM'.

And if you click on the links they should take you to their respective pages.

Next up we are going to allow users to view and edit their notes.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/52>)

For reference, here is the code so far

⌚ Frontend Source :call-the-list-api

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/call-the-list-api>)

Display a Note

Now that we have a listing of all the notes, let's create a page that displays a note and let's the user edit it.

The first thing we are going to need to do is load the note when our container loads. Just like what we did in the `Home` container. So let's get started.

Add the Route

Let's add a route for the note page that we are going to create.

◆ **CHANGE** Add the following line to `src/Routes.js` below our `/notes/new` route. We are using the `AppliedRoute` component that we created in the Add the session to the state (`/chapters/add-the-session-to-the-state.html`) chapter.

```
<AppliedRoute path="/notes/:id" exact component={Notes} props={{childProps}} />
```

This is important because we are going to be pattern matching to extract our note id from the URL.

By using the route path `/notes/:id` we are telling the router to send all matching routes to our component `Notes`. This will also end up matching the route `/notes/new` with an `id` of `new`. To ensure that doesn't happen, we put our `/notes/new` route before the pattern matching one.

◆ **CHANGE** And include our component in the header.

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

Of course this component doesn't exist yet and we are going to create it now.

Add the Container

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

```
import React, { Component } from "react";
import { invokeApig } from "../libs/awsLib";

export default class Notes extends Component {
  constructor(props) {
    super(props);

    this.file = null;

    this.state = {
      note: null,
      content: ""
    };
  }

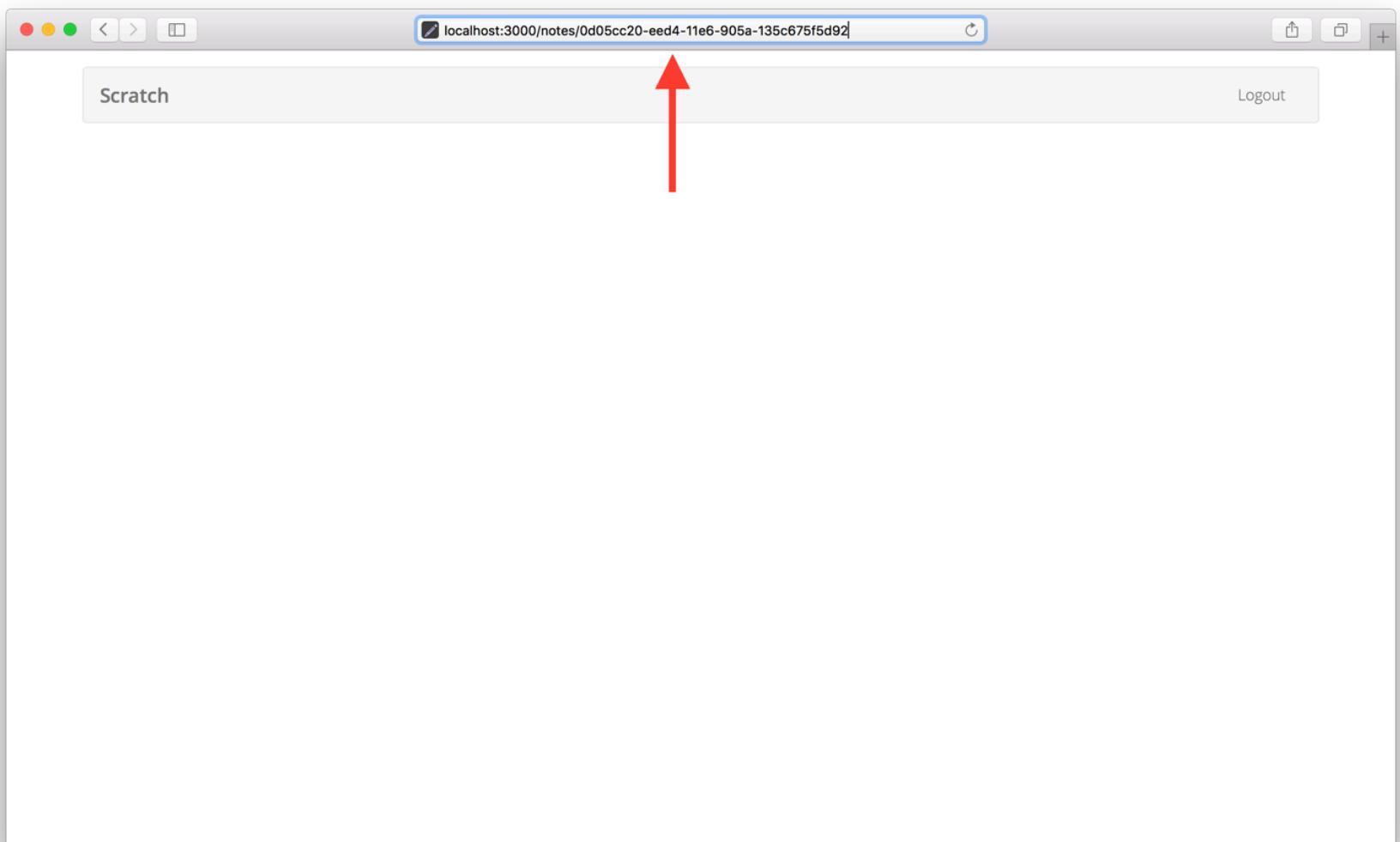
  async componentDidMount() {
    try {
      const results = await this.getNote();
      this.setState({
        note: results,
        content: results.content
      });
    } catch (e) {
      alert(e);
    }
  }

  getNote() {
    return invokeApig({ path: `/notes/${this.props.match.params.id}` });
  }

  render() {
    return <div className="Notes" />;
  }
}
```

All this does is load the note on `componentDidMount` and save it to the state. We get the `id` of our note from the URL using the props automatically passed to us by React-Router in `this.props.match.params.id`. The keyword `id` is a part of the pattern matching in our route (`/notes/:id`).

And now if you switch over to your browser and navigate to a note that we previously created, you'll notice that the page renders an empty container.



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

For help and discussion

💬 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/53>)

For reference, here is the code so far

⌚ [Frontend Source :display-a-note](#)

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/display-a-note>)

Render the Note Form

Now that our container loads a note on `componentDidMount`, let's go ahead and render the form that we'll use to edit it.

◆ **CHANGE** Replace our placeholder `render` method in `src/containers/Notes.js` with the following.

```
validateForm() {
  return this.state.content.length > 0;
}

formatFilename(str) {
  return str.length < 50
    ? str
    : str.substr(0, 20) + "..." + str.substr(str.length - 20,
str.length);
}

handleChange = event => {
  this.setState({
    [event.target.id]: event.target.value
  });
}

handleFileChange = event => {
  this.file = event.target.files[0];
}

handleSubmit = async event => {
  event.preventDefault();

  if (this.file && this.file.size > config.MAX_ATTACHMENT_SIZE) {
    alert("Please pick a file smaller than 5MB");
  }
}
```

```
        return;
    }

    this.setState({ isLoading: true });
}

handleDelete = async event => {
    event.preventDefault();

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

    if (!confirmed) {
        return;
    }

    this.setState({ isDeleting: true });
}

render() {
    return (
        <div className="Notes">
            {this.state.note &&
                <form onSubmit={this.handleSubmit}>
                    <FormGroup controlId="content">
                        <FormControl
                            onChange={this.handleChange}
                            value={this.state.content}
                            componentClass="textarea"
                        />
                    </FormGroup>
            {this.state.note.attachment &&
                <FormGroup>
                    <ControlLabel>Attachment</ControlLabel>
                    <FormControl.Static>
                        <a
                            target="_blank"
                        >
                    </FormControl.Static>
                </FormGroup>
            }
        </div>
    );
}
```

```

        rel="noopener noreferrer"
        href={this.state.note.attachment}
      >
      {this.formatFilename(this.state.note.attachment)}
    </a>
  </FormControl.Static>
</FormGroup>
<FormGroup controlId="file">
  {!this.state.note.attachment &&
    <ControlLabel>Attachment</ControlLabel>}
  <FormControl onChange={this.handleFileChange} type="file"
/>
</FormGroup>
<LoaderButton
  block
  bsStyle="primary"
  bsSize="large"
  disabled={!this.validateForm()}
  type="submit"
  isLoading={this.state.isLoading}
  text="Save"
  loadingText="Saving..."
/>
<LoaderButton
  block
  bsStyle="danger"
  bsSize="large"
  isLoading={this.state.isDeleting}
  onClick={this.handleDelete}
  text="Delete"
  loadingText="Deleting..."
/>
</form>
</div>
);
}

```

We are doing a few things here:

1. We render our form only when `this.state.note` is available.
2. Inside the form we conditionally render the part where we display the attachment by using `this.state.note.attachment`.
3. We form the attachment URL using `formatFilename` (since S3 gives us some very long URLs).
4. We also added a delete button to allow users to delete the note. And just like the submit button it too needs a flag that signals that the call is in progress. We call it `isDeleting`.
5. We handle attachments with a file input exactly like we did in the `NewNote` component.
6. Our delete button also confirms with the user if they want to delete the note using the browser's `confirm` dialog.

To complete this code, let's add `isLoading` and `isDeleting` to the state.

◆ CHANGE So our new initial state in the `constructor` looks like so.

```
this.state = {  
  isLoading: null,  
  isDeleting: null,  
  note: null,  
  content: ""  
};
```

◆ CHANGE Let's also add some styles by adding the following to `src/containers/Notes.css`.

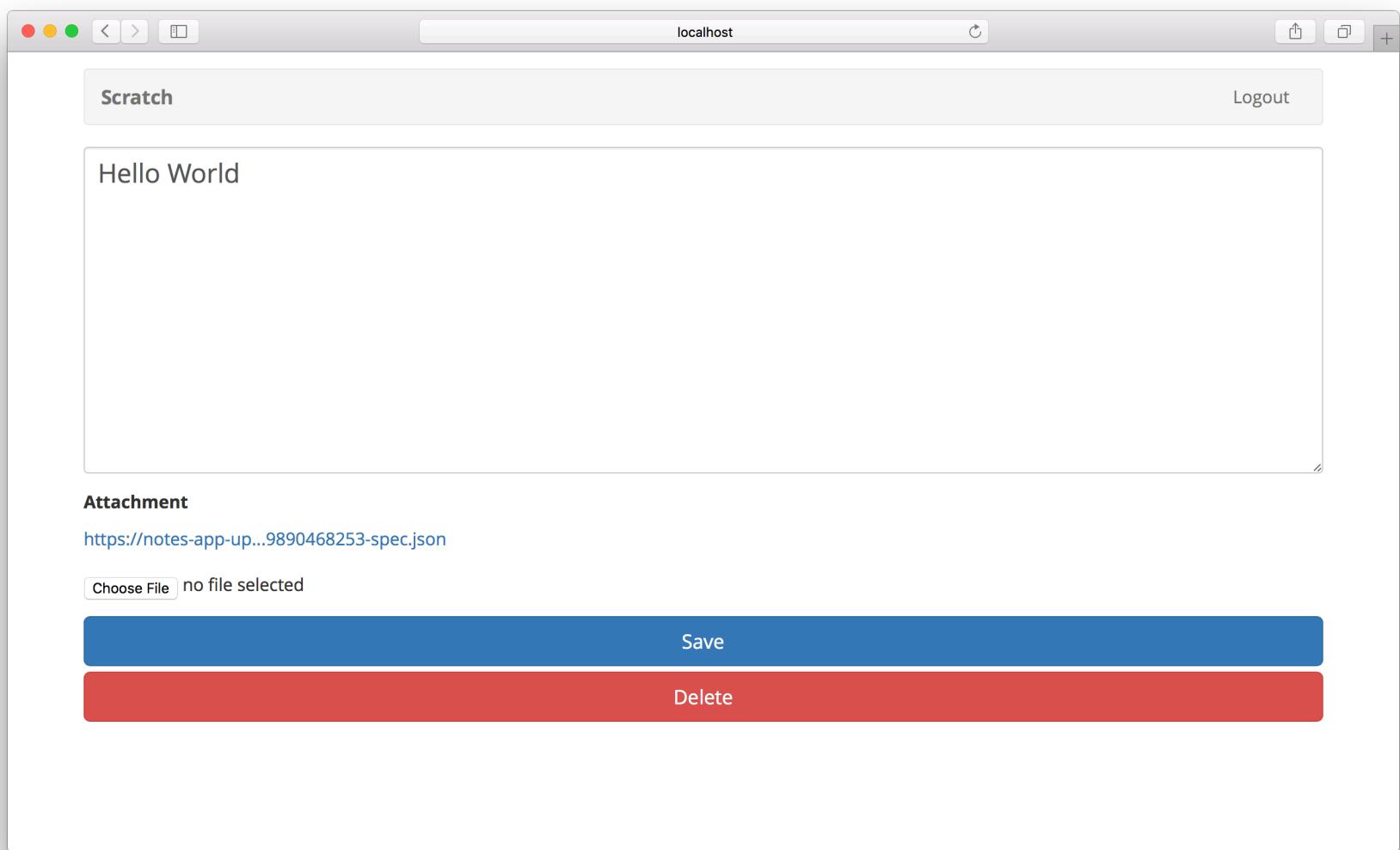
```
.Notes form {  
  padding-bottom: 15px;  
}
```

```
.Notes form textarea {  
  height: 300px;  
  font-size: 24px;  
}
```

◆ CHANGE Also, let's include the React-Bootstrap components that we are using here by adding the following to our header. And our styles, the `LoaderButton`, and the `config`.

```
import { FormGroup, FormControl, ControlLabel } from "react-bootstrap";
import LoaderButton from "../components/LoaderButton";
import config from "../config";
import "./Notes.css";
```

And that's it. If you switch over to your browser, you should see the note loaded.



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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/54>)

For reference, here is the code so far

Q Frontend Source :render-the-note-form
(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/render-the-note-form>)

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` method in `src/containers/Notes.js` with the following.

```
saveNote(note) {
  return invokeApig({
    path: `/notes/${this.props.match.params.id}`,
    method: "PUT",
    body: note
  });
}

handleSubmit = async event => {
  let uploadedFilename;

  event.preventDefault();

  if (this.file && this.file.size > config.MAX_ATTACHMENT_SIZE) {
    alert("Please pick a file smaller than 5MB");
    return;
  }

  this.setState({ isLoading: true });

  try {
    if (this.file) {
      uploadedFilename = (await s3Upload(this.file))
        .Location;
    }
  }
  await this.saveNote({
```

```
...this.state.note,  
content: this.state.content,  
attachment: uploadedFilename || this.state.note.attachment  
});  
  
this.props.history.push("/");  
} catch (e) {  
alert(e);  
this.setState({ isLoading: false });  
}  
}
```

◆ CHANGE And include our `s3Upload` helper method in the header:

```
import { invokeApig, s3Upload } from "../libs/awsLib";
```

The code above is doing a couple of things that should be very similar to what we did in the `NewNote` container.

1. If there is a file to upload we call `s3Upload` to upload it and save the URL.
2. We save the note by making `PUT` request with the note object to `/notes/note_id` where we get the `note_id` from `this.props.match.params.id`.
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.

The screenshot shows a web application interface. At the top, there's a header with 'localhost' and a 'Logout' button. Below the header, the main content area contains a note with the title 'Hello New World'. Underneath the note, there's an 'Attachment' section. It displays a URL: <https://notes-app-up...9890468253-spec.json>. There's a 'Choose File' button with the text 'no file selected' next to it. Below the button is a blue progress bar with a circular arrow icon and the text 'Saving...'. At the bottom of the attachment section is a red button labeled 'Delete'.

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 JS SDK Docs (<http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/S3.html#deleteObject-property>) on how to a delete file from S3.

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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/55>)

For reference, here is the code so far

⌚ Frontend Source :save-changes-to-a-note

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/save-changes-to-a-note>)

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` method in `src/containers/Notes.js`.

```
deleteNote() {
  return invokeApig({
    path: `/notes/${this.props.match.params.id}`,
    method: "DELETE"
  });
}

handleDelete = async event => {
  event.preventDefault();

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

  if (!confirmed) {
    return;
  }

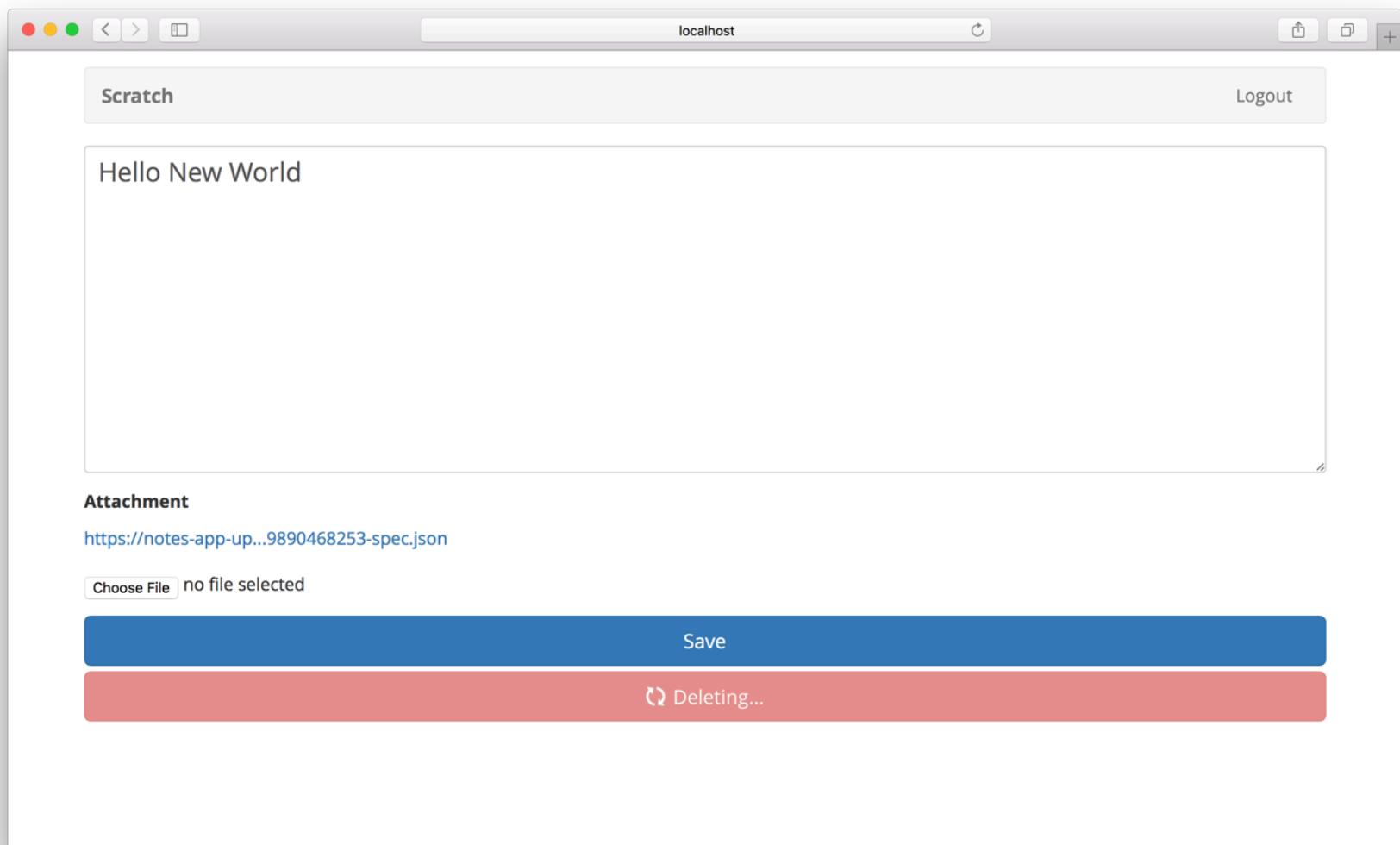
  this.setState({ isDeleting: true });

  try {
    await this.deleteNote();
    this.props.history.push("/");
  } catch (e) {
    alert(e);
    this.setState({ isDeleting: false });
  }
}
```

}

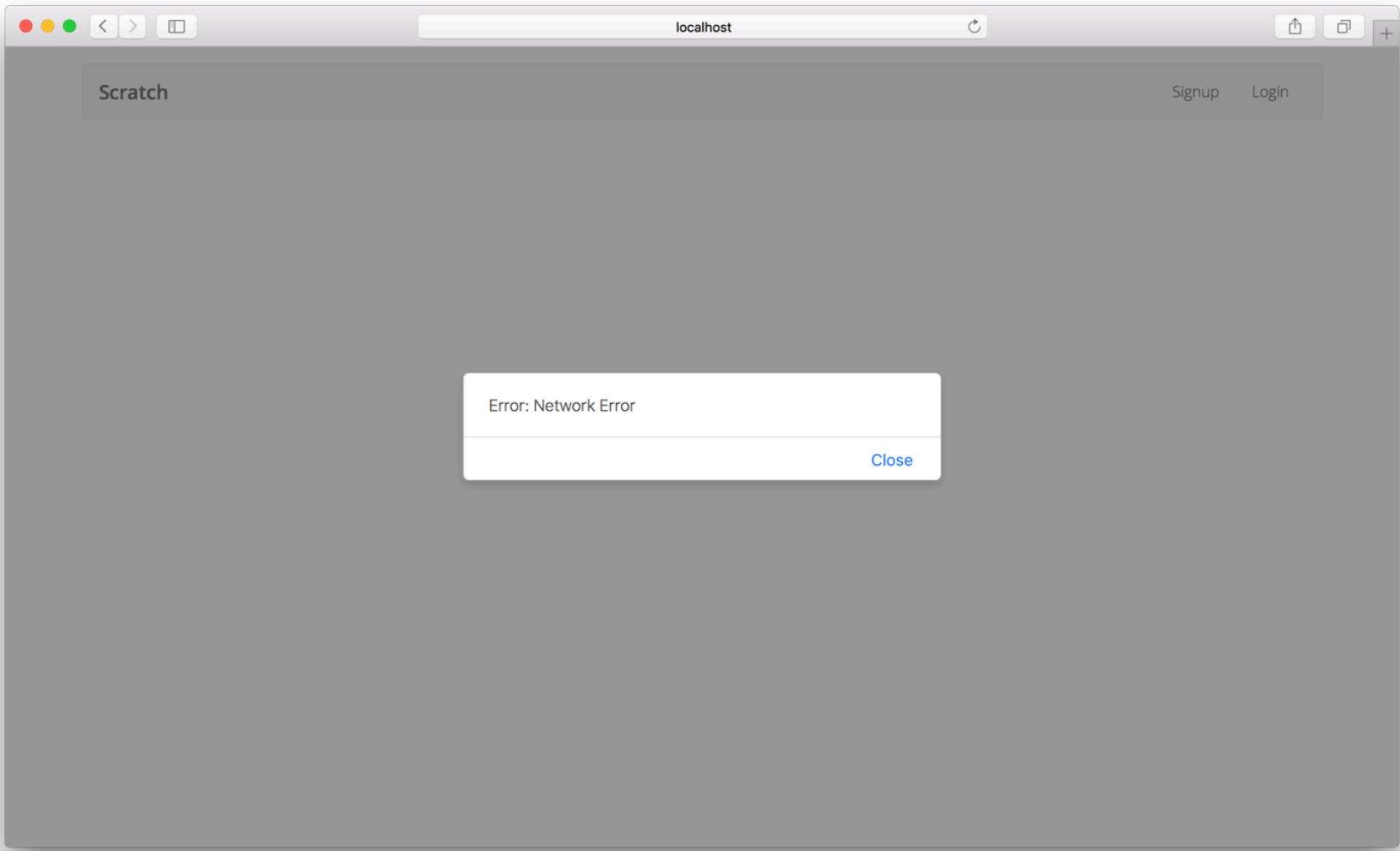
We are simply making a `DELETE` request to `/notes/note_id` where we get the `id` from `this.props.match.params.id`. This calls our delete API and we redirect to the homepage on success.

Now if you switch over to your browser and try deleting a note you should see it confirm your action and then delete the note.



Again, you might have noticed that we are not deleting the attachment when we are deleting a note. We are leaving that up to you to keep things simple. Check the AWS JS SDK Docs (<http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/S3.html#deleteObject-property>) on how to a delete file from S3.

Now with our app nearly complete, we'll look at securing some the pages of our app that require a login. Currently if you visit a note page while you are logged out, it throws an ugly error.



Instead, we would like it to redirect us to the login page and then redirect us back after we login. Let's look at how to do that next.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/56>)

For reference, here is the code so far

⌚ Frontend Source :delete-a-note

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/delete-a-note>)

Set up Secure Pages

We are almost done putting together our app. All the pages are done but there are a few pages that should not be accessible if a user is not logged in. For example, a page with the note should not load if a user is not logged in. Currently, we get an error when we do this. This is because the page loads and since there is no user in the session, the call to our API fails.

We also have a couple of pages that need to behave in sort of the same way. We want the user to be redirected to the homepage if they type in the login (`/login`) or signup (`/signup`) URL. Currently, the login and sign up page end up loading even though the user is already logged in.

There are many ways to solve the above problems. The simplest would be to just check the conditions in our containers and redirect. But since we have a few containers that need the same logic we can create a special route (like the `AppliedRoute` from the Add the session to the state (/chapters/add-the-session-to-the-state.html) chapter) for it.

We are going to create two different route components to fix the problem we have.

1. A route called the `AuthenticatedRoute`, that checks if the user is authenticated before routing.
2. And a component called the `UnauthenticatedRoute`, that ensures the user is not authenticated.

Let's create these components next.

For help and discussion

 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/57>)

Create a Route That Redirects

Let's first create a route that will check if the user is logged in before routing.

◆ CHANGE Add the following to `src/components/AuthenticatedRoute.js`.

```
import React from "react";
import { Route, Redirect } from "react-router-dom";

export default ({ component: C, props: cProps, ...rest }) =>
  <Route
    {...rest}
    render={props =>
      cProps.isAuthenticated
        ? <C {...props} {...cProps} />
        : <Redirect
            to={`/login?
redirect=${props.location.pathname}${props.location
  .search}`}
          />}
    />};
  
```

This component is similar to the `AppliedRoute` component that we created in the Add the session to the state (/chapters/add-the-session-to-the-state.html) chapter. The main difference being that we look at the props that are passed in to check if a user is authenticated. If the user is authenticated, then we simply render the passed in component. And if the user is not authenticated, then we use the `Redirect` React Router v4 component to redirect the user to the login page. We also pass in the current path to the login page (`redirect` in the querystring). We will use this later to redirect us back after the user logs in.

We'll do something similar to ensure that the user is not authenticated.

◆ CHANGE Add the following to `src/components/UnauthenticatedRoute.js`.

```
import React from "react";
import { Route, Redirect } from "react-router-dom";

export default ({ component: C, props: cProps, ...rest }) =>
<Route
  {...rest}
  render={props =>
    !cProps.isAuthenticated
      ? <C {...props} {...cProps} />
      : <Redirect to="/" />}
/>;
```

Here we are checking to ensure that the user is not authenticated before we render the component that is passed in. And in the case where the user is authenticated, we use the `Redirect` component to simply send the user to the homepage.

Next, let's use these components in our app.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/58>)

For reference, here is the code so far

⌚ Frontend Source :create-a-route-that-redirects

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/create-a-route-that-redirects>)

Use the Redirect Routes

Now that we created the `AuthenticatedRoute` and `UnauthenticatedRoute` in the last chapter, let's use them on the containers we want to secure.

◆ CHANGE First import them in the header of `src/Routes.js`.

```
import AuthenticatedRoute from "./components/AuthenticatedRoute";
import UnauthenticatedRoute from "./components/UnauthenticatedRoute";
```

Next, we simply switch to our new redirect routes.

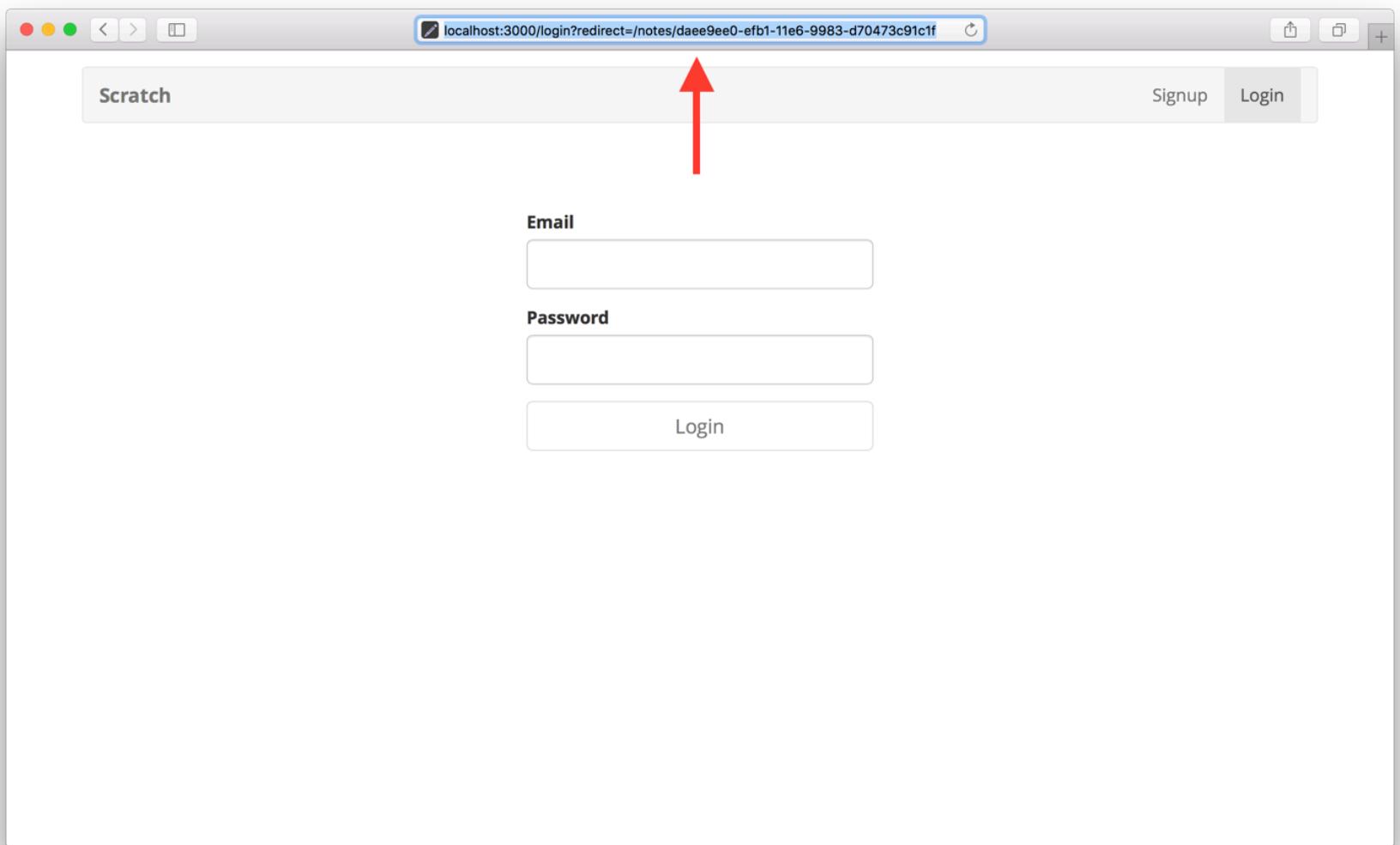
So the following routes in `src/Routes.js` would be affected.

```
<AppliedRoute path="/login" exact component={Login} props={childProps}>
<AppliedRoute path="/signup" exact component={Signup} props={childProps}>
<AppliedRoute path="/notes/new" exact component={NewNote} props={childProps}>
<AppliedRoute path="/notes/:id" exact component={Notes} props={childProps}>
```

◆ CHANGE They should now look like so:

```
<UnauthenticatedRoute path="/login" exact component={Login} props={childProps}>
<UnauthenticatedRoute path="/signup" exact component={Signup} props={childProps}>
<AuthenticatedRoute path="/notes/new" exact component={NewNote} props={childProps}>
<AuthenticatedRoute path="/notes/:id" exact component={Notes} props={childProps}>
```

And now if we tried to load a note page while not logged in, we would be redirected to the login page with a reference to the note page.



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

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/59>)

For reference, here is the code so far

🔗 Frontend Source :use-the-redirect-routes

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/use-the-redirect-routes>)

Redirect on Login

Our secured pages redirect to the login page when the user is not logged in, with a referral to the originating page. To redirect back after they login, we need to do a couple of more things. Currently, our `Login` component does the redirecting after the user logs in. We are going to move this to the newly created `UnauthenticatedRoute` component.

Let's start by adding a method to read the `redirect` URL from the querystring.

◆ CHANGE Add the following method to your
`src/components/UnauthenticatedRoute.js` below the imports.

```
function querystring(name, url = window.location.href) {  
  name = name.replace(/[\[]/g, "\\\\$&");  
  
  const regex = new RegExp("[?&]" + name + "(=([^&#]*|&|#|$)", "i");  
  const results = regex.exec(url);  
  
  if (!results) {  
    return null;  
  }  
  if (!results[2]) {  
    return "";  
  }  
  
  return decodeURIComponent(results[2].replace(/\+/g, " "));  
}
```

This method takes the querystring param we want to read and returns it.

Now let's update our `Redirect` component to use this when it redirects.

◆ CHANGE Replace our current `export default ({ component: C, props: cProps, ...rest }) => {` method with the following.

```
export default ({ component: C, props: cProps, ...rest }) => {
  const redirect = querystring("redirect");
  return (
    <Route
      {...rest}
      render={props =>
        !cProps.isAuthenticated
          ? <C {...props} {...cProps} />
          : <Redirect
              to={redirect === "" || redirect === null ? "/" :
redirection}
            />}
      />
    );
};
```

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

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

Now our login page should redirect after we login.

And that's it! Our app is ready to go live. Let's look at how we are going to deploy it using our serverless setup.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/60>)

For reference, here is the code so far

🔗 Frontend Source :redirect-on-login

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/redirect-on-login>)

Deploy the Frontend

Now that we have our setup working in our local environment, let's do our first deploy and look into what we need to do to host our serverless application.

The basic setup we are going to be using will look something like this:

1. Upload the assets of our app
2. Use a CDN to serve out our assets
3. Point our domain to the CDN distribution
4. Switch to HTTPS with a SSL certificate

AWS provides quite a few services that can help us do the above. We are going to use S3 (<https://aws.amazon.com/s3/>) to host our assets, CloudFront (<https://aws.amazon.com/cloudfront/>) to serve it, Route 53 (<https://aws.amazon.com/route53/>) to manage our domain, and Certificate Manager (<https://aws.amazon.com/certificate-manager/>) to handle our SSL certificate.

So let's get started by first configuring our S3 bucket to upload the assets of our app.

For help and discussion

 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/61>)

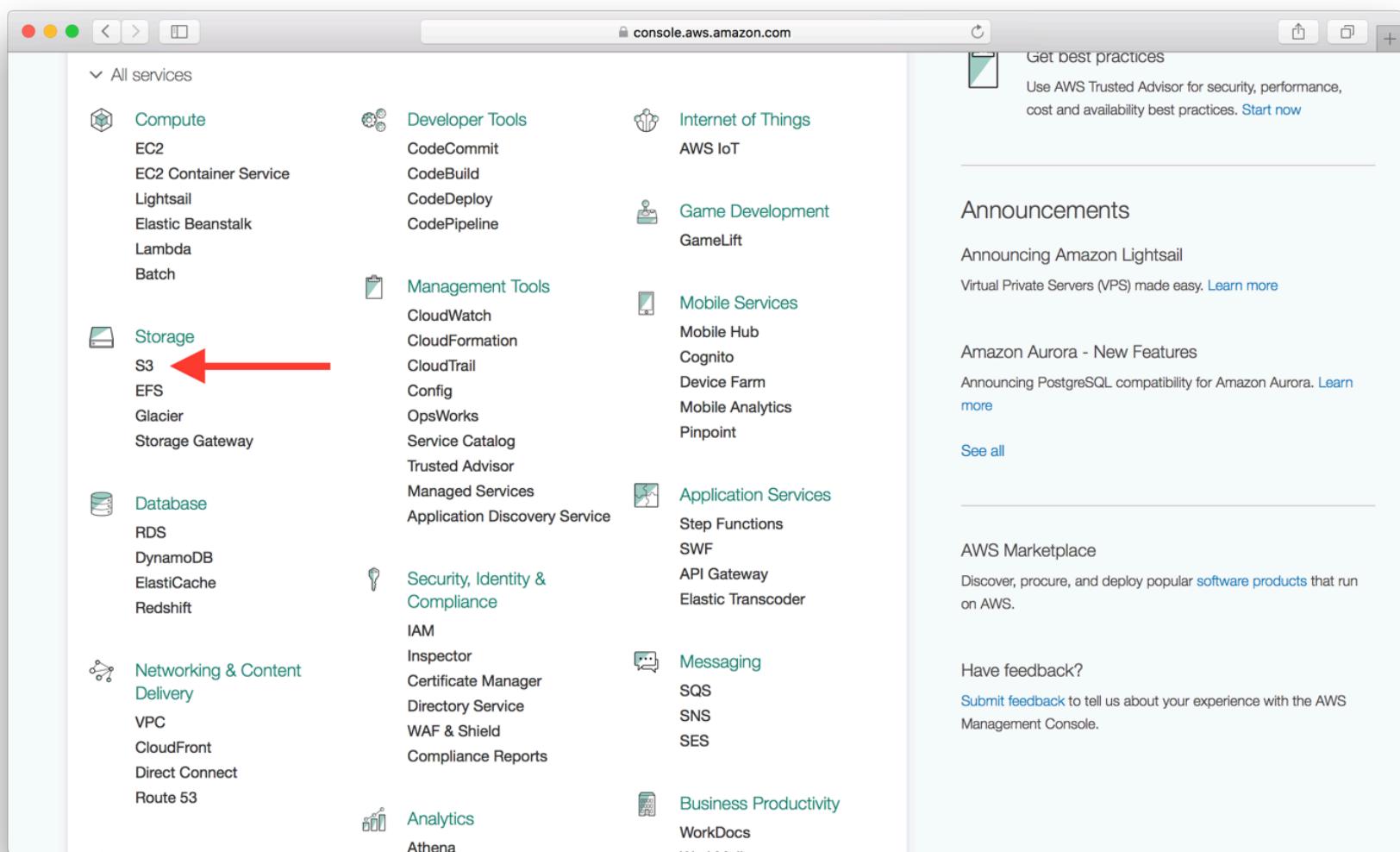
Create an S3 Bucket

To be able to host our note taking app, we need to upload the assets that are going to be served out statically on S3. S3 has a concept of buckets (or folders) to separate different types of files.

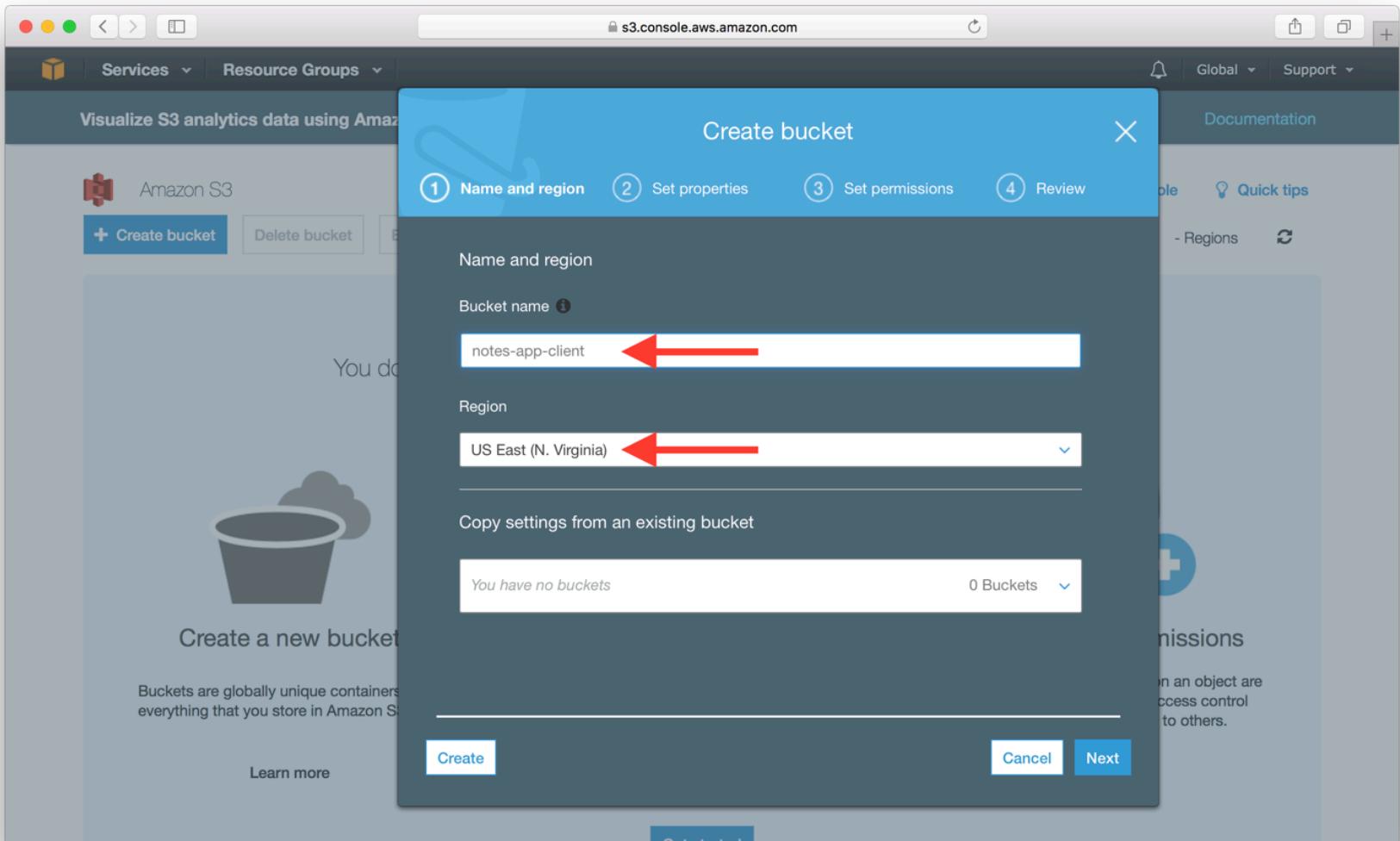
A bucket can also be configured to host the assets in it as a static website and is automatically assigned a publicly accessible URL. So let's get started.

Create the Bucket

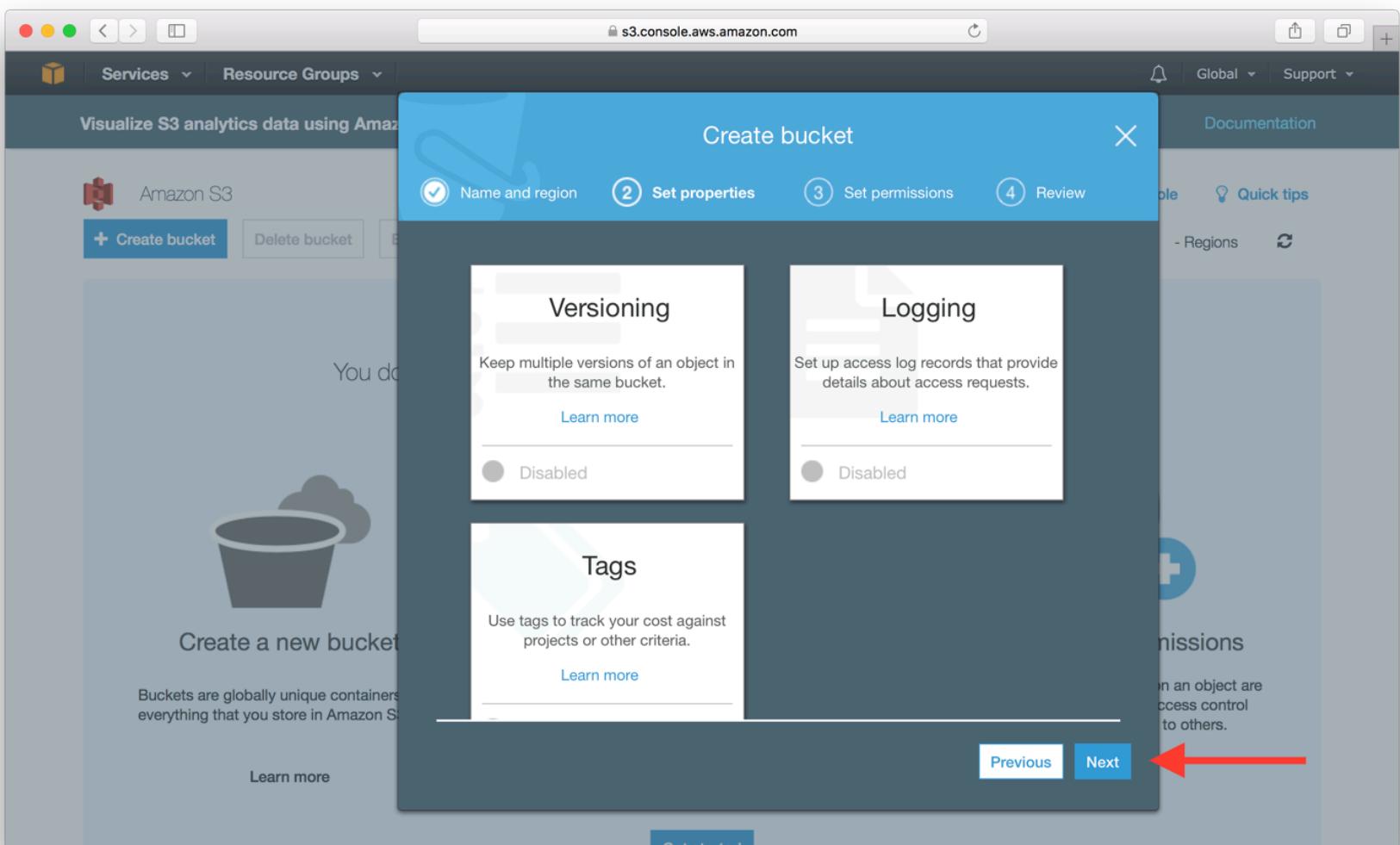
First, log in to your AWS Console (<https://console.aws.amazon.com>) and select S3 from the list of services.

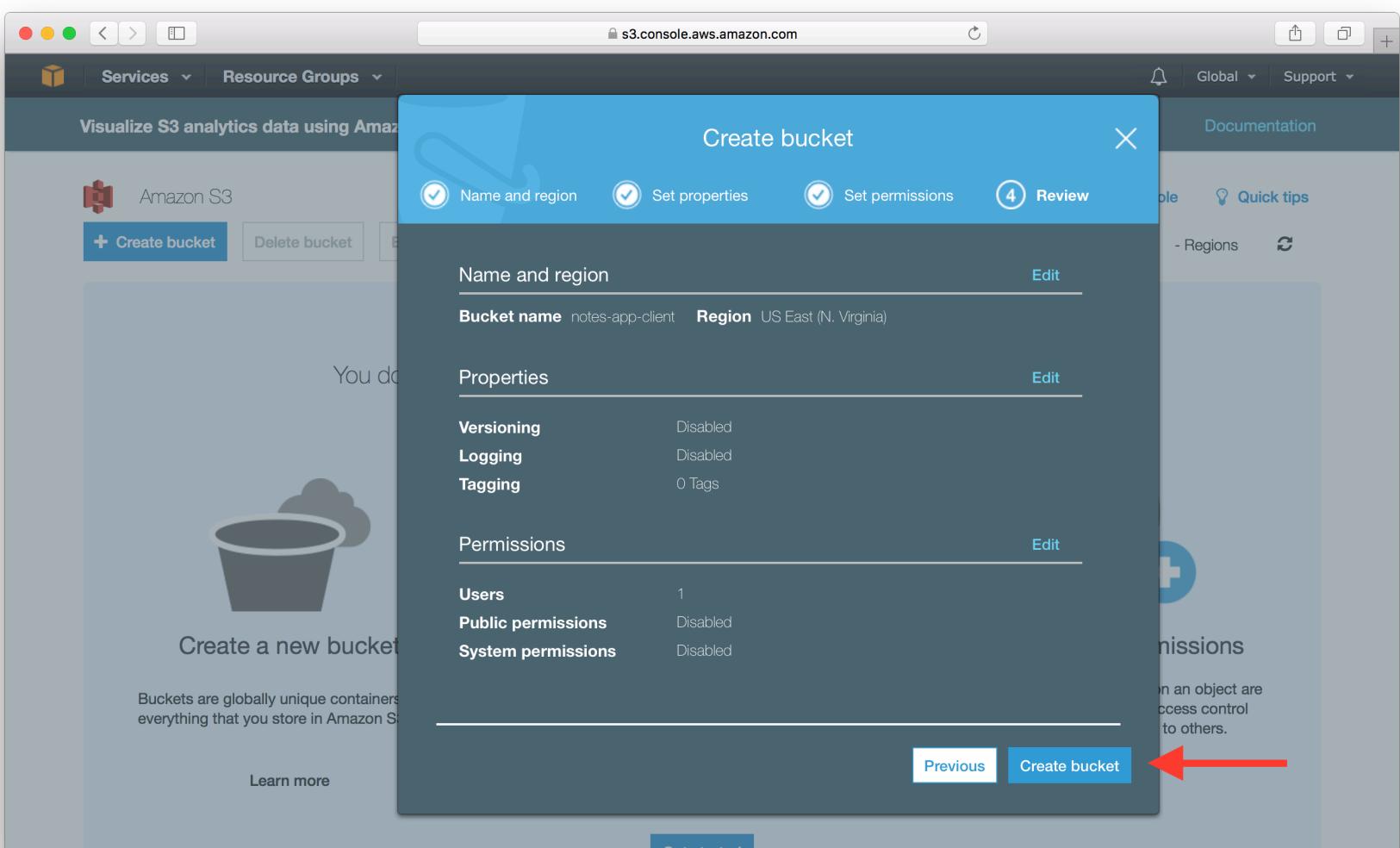
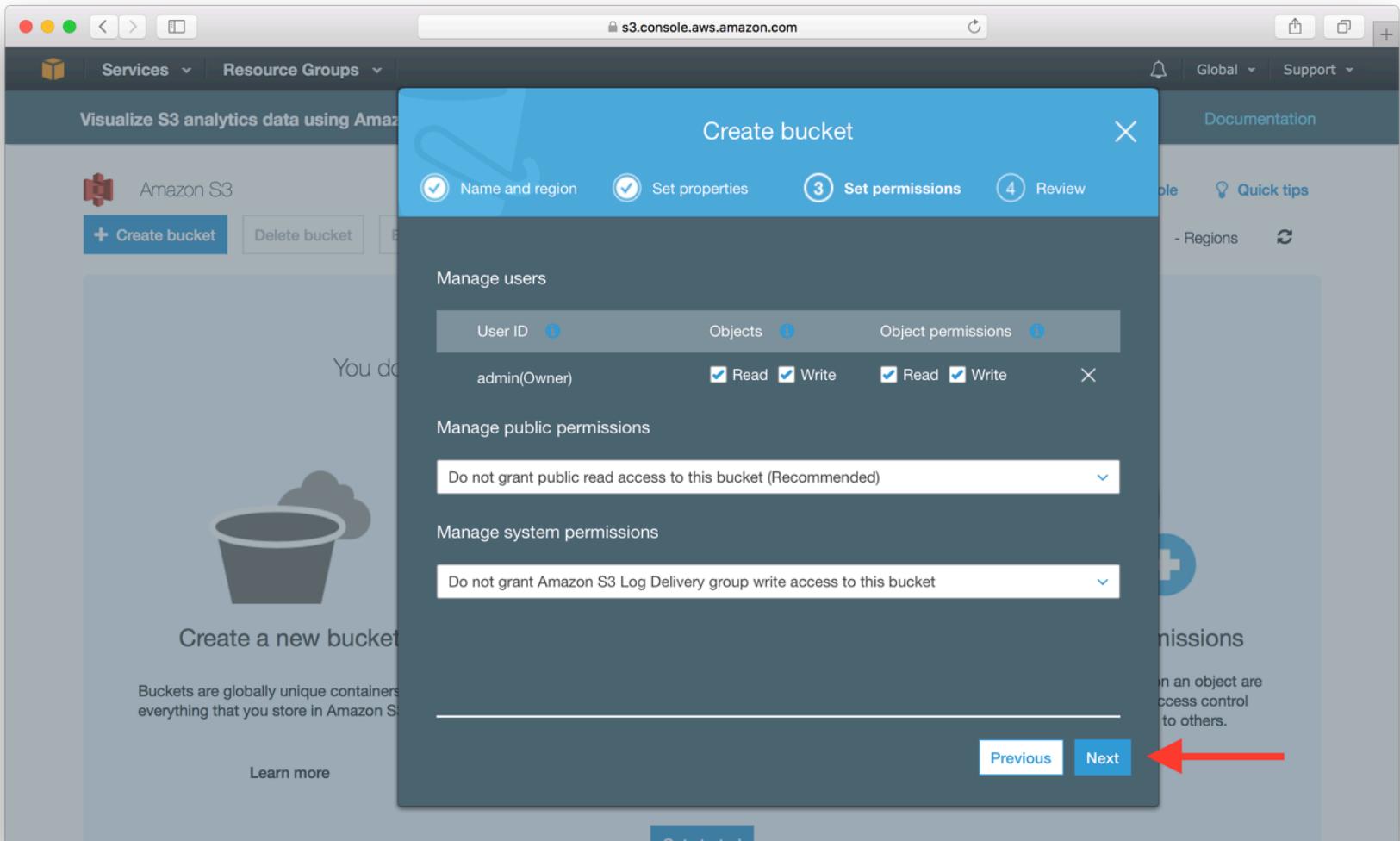


Select **Create Bucket** and pick a name for your application and select the **US East (N. Virginia)** Region. Since our application is being served out using a CDN, the region should not matter to us.



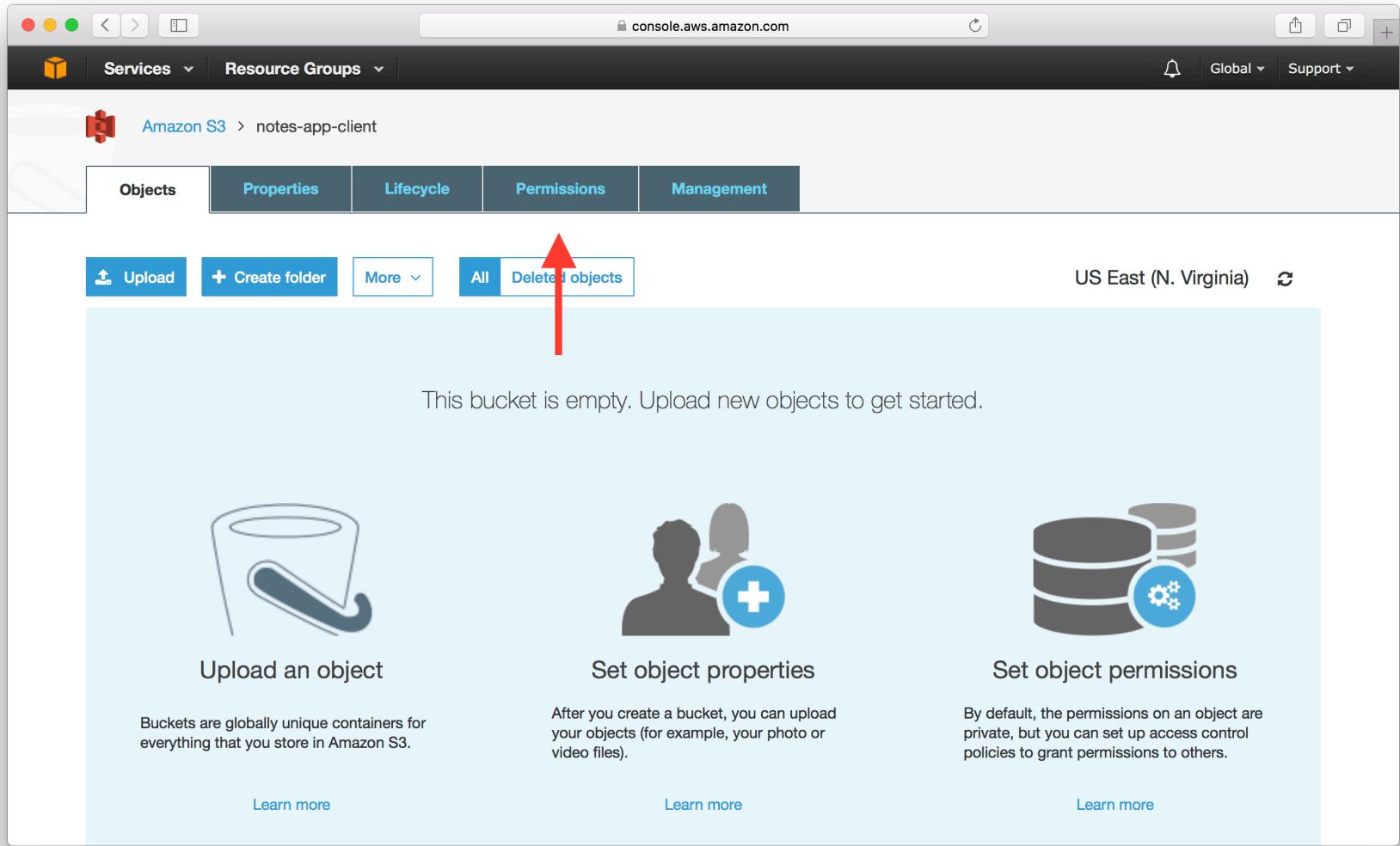
Go through the next steps and leave the defaults by clicking **Next**.





Now click on your newly created bucket from the list and navigate to its permissions panel by

clicking Permissions.



Add Permissions

Buckets by default are not publicly accessible, so we need to change the S3 Bucket Permission. Select the **Bucket Policy** from the permissions panel.

The screenshot shows the AWS S3 console for a bucket named 'notes-app-client'. The 'Management' tab is selected. Under the 'Management' tab, the 'Bucket Policy' tab is highlighted. The page displays sections for managing users and public permissions, but the main focus is on the bucket policy editor.

◆ CHANGE Add the following bucket policy into the editor. Where `notes-app-client` is the name of our S3 bucket. Make sure to use the name of your bucket here.

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Sid": "PublicReadForGetBucketObjects",  
      "Effect": "Allow",  
      "Principal": "*",  
      "Action": [ "s3:GetObject" ],  
      "Resource": [ "arn:aws:s3:::notes-app-client/*" ]  
    }  
  ]  
}
```

The screenshot shows the AWS S3 Bucket Policy editor. At the top, there's a navigation bar with 'Services' (selected), 'Resource Groups', and 'Global' and 'Support' dropdowns. Below that, the 'Amazon S3' icon and the bucket name 'notes-app-client' are displayed. A tab bar at the top includes 'Objects', 'Properties', 'Lifecycle', 'Permissions' (selected), and 'Management'. Under 'Management', three buttons are shown: 'Access Control List', 'Bucket Policy' (which is highlighted in blue), and 'CORS configuration'. A large text area contains the JSON policy code:

```
1  {
2    "Version": "2012-10-17",
3    "Statement": [
4      {
5        "Sid": "PublicReadForGetBucketObjects",
6        "Effect": "Allow",
7        "Principal": "*",
8        "Action": ["s3:GetObject"],
9        "Resource": ["arn:aws:s3:::notes-app-client/*"]
10       }
11     ]
12   }
```

Below the text area are three buttons: 'Delete', 'Cancel', and 'Save' (highlighted in blue). At the bottom left, there are links for 'Documentation' and 'Policy generator'.

And hit **Save**.

Enable Static Web Hosting

And finally we need to turn our bucket into a static website. Select the **Properties** tab from the top panel.

The screenshot shows the AWS S3 Properties tab for a bucket named 'notes-app-client'. The 'Properties' tab is selected. A red arrow points to the 'Static website hosting' section.

Versioning
Keep multiple versions of an object in the same bucket.
[Learn more](#)
 Disabled

Logging
Set up access log records that provide details about access requests.
[Learn more](#)
 Disabled

Static website hosting
Host a static website, which does not require server-side technologies.
[Learn more](#)
 Disabled

Advanced settings

Tags
Use tags to track your cost against projects or other criteria.
[Learn more](#)

Cross-region replication
Automate copying objects across different AWS Regions.
[Learn more](#)

Transfer acceleration
Enable fast, easy and secure transfers of files to and from your bucket.
[Learn more](#)

Select **Static website hosting**.

The screenshot shows the AWS S3 Properties tab for the same bucket. A red arrow points to the 'Static website hosting' section.

Versioning
Keep multiple versions of an object in the same bucket.
[Learn more](#)
 Disabled

Logging
Set up access log records that provide details about access requests.
[Learn more](#)
 Disabled

Static website hosting ←
Host a static website, which does not require server-side technologies.
[Learn more](#)
 Disabled

Advanced settings

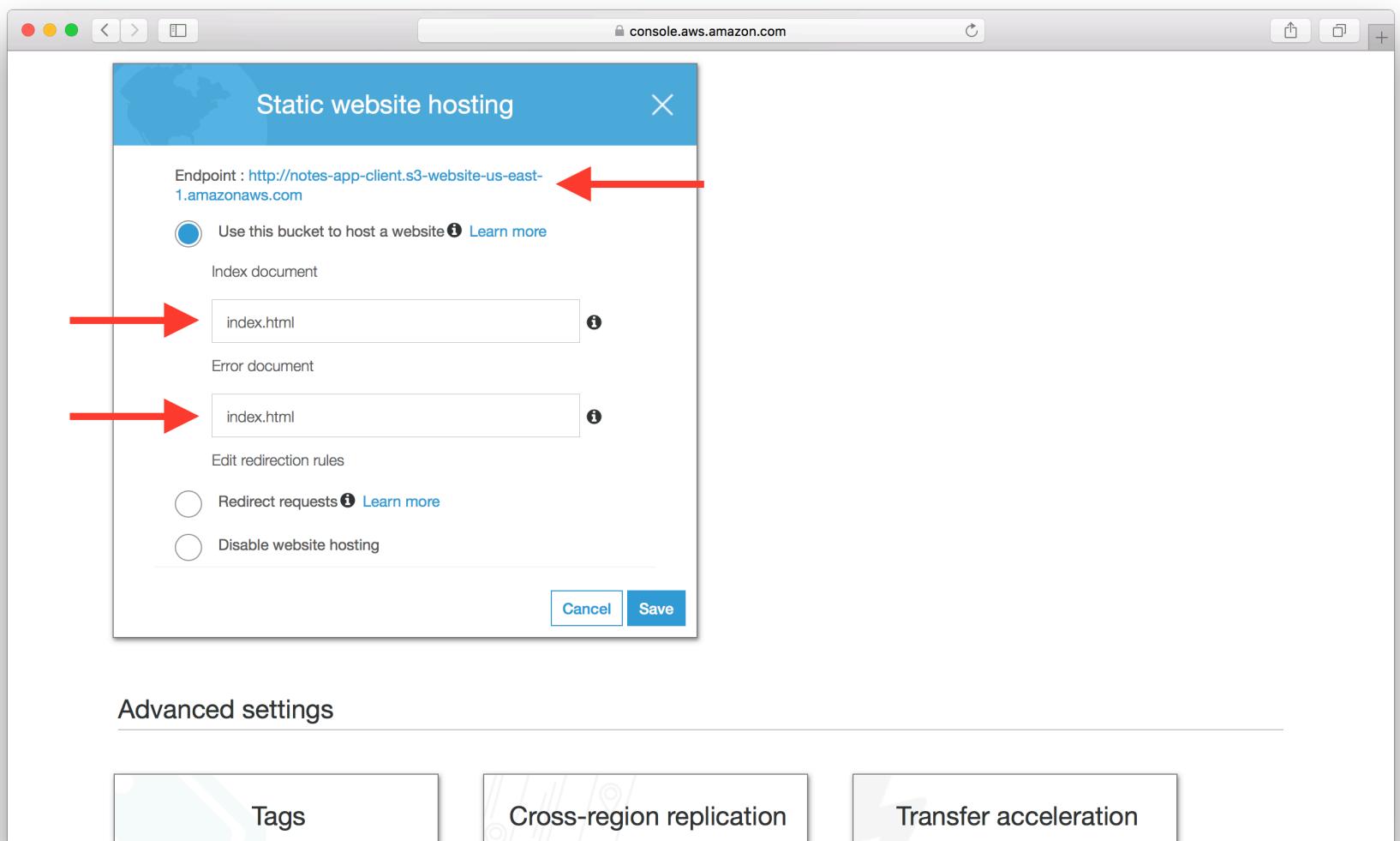
Tags
Use tags to track your cost against projects or other criteria.
[Learn more](#)

Cross-region replication
Automate copying objects across different AWS Regions.
[Learn more](#)

Transfer acceleration
Enable fast, easy and secure transfers of files to and from your bucket.
[Learn more](#)

Now select **Use this bucket to host a website** and add our `index.html` as the **Index Document** and the **Error Document**. Since we are letting React handle 404s, we can simply redirect our errors to our `index.html` as well. Hit **Save** once you are done.

This panel also shows us where our app will be accessible. AWS assigns us a URL for our static website. In this case the URL assigned to me is `notes-app-client.s3-website-us-east-1.amazonaws.com`.



Now that our bucket is all set up and ready, let's go ahead and upload our assets to it.

For help and discussion

 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/62>)

Deploy to S3

Now that our S3 Bucket is created we are ready to upload the assets of our app.

Build Our App

Create React App comes with a convenient way to package and prepare our app for deployment. From our working directory simply run the following command.

```
$ npm run build
```

This packages all of our assets and places them in the `build/` directory.

Upload to S3

Now to deploy simply run the following command; where `YOUR_S3_DEPLOY_BUCKET_NAME` is the name of the S3 Bucket we created in the Create an S3 bucket (/chapters/create-an-s3-bucket.html) chapter.

```
$ aws s3 sync build/ s3://YOUR_S3_DEPLOY_BUCKET_NAME
```

All this command does is that it syncs the `build/` directory with our bucket on S3. Just as a sanity check, go into the S3 section in your AWS Console (<https://console.aws.amazon.com/console/home>) and check if your bucket has the files we just uploaded.

Amazon S3 > notes-app-client

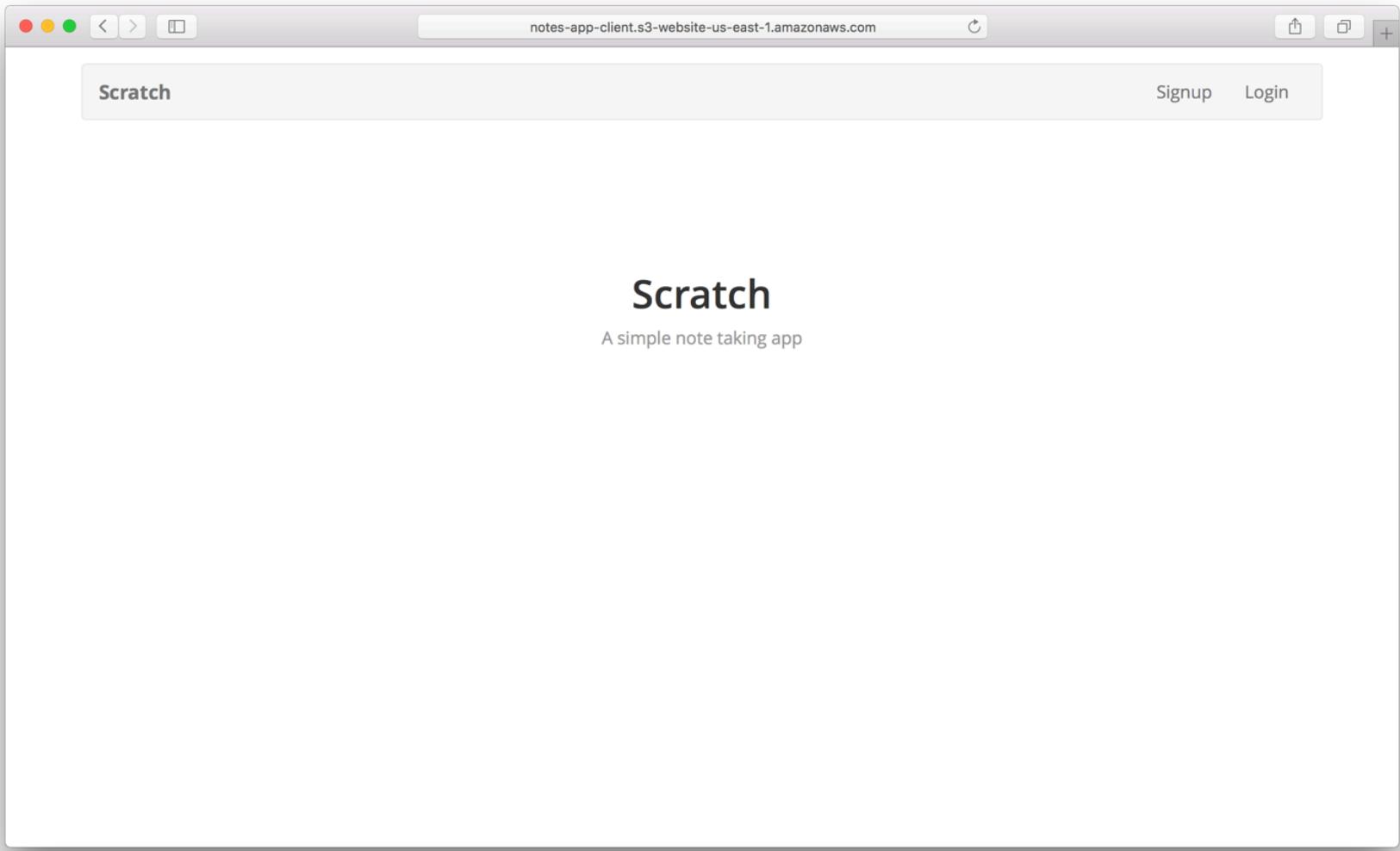
Objects Properties Lifecycle Permissions Management

Upload Create folder More All Deleted objects US East (N. Virginia)

Type a prefix and press Enter to search. Press ESC to clear.

Name	Last modified	Size	Storage class
static	--	--	--
.DS_Store	Feb 16, 2017 1:45:26 PM	6.0 KB	Standard
android-chrome-192x192.png	Feb 16, 2017 1:45:26 PM	2.9 KB	Standard
android-chrome-256x256.png	Feb 16, 2017 1:45:26 PM	7.5 KB	Standard
apple-touch-icon.png	Feb 16, 2017 1:45:26 PM	2.6 KB	Standard
asset-manifest.json	Feb 16, 2017 1:45:26 PM	196.0 B	Standard
browserconfig.xml	Feb 16, 2017 1:45:26 PM	246.0 B	Standard
favicon-16x16.png	Feb 16, 2017 1:45:26 PM	659.0 B	Standard
favicon-32x32.png	Feb 16, 2017 1:45:26 PM	885.0 B	Standard

And our app should be live on S3! If you head over to the URL assigned to you (in my case it is <http://notes-app-client.s3-website-us-east-1.amazonaws.com>) (http://notes-app-client.s3-website-us-east-1.amazonaws.com)), you should see it live.



App Bundle Size

Just a quick note on the size of our frontend app. The main JS file in our app is under `build/static/js/main.id.js` and it is quite large because of our AWS import. One of our readers did a bit of research on this and found a simple way to almost half the size of the bundle.

This step is optional for completing the tutorial but can be helpful if you are basing your projects on it. In our `src/libs/awsLib.js` you can replace the `import AWS from "aws-sdk";` with the following:

```
import AWS from 'aws-sdk/global';
import S3 from 'aws-sdk/clients/s3';
```

And when uploading a file to S3, replace `new AWS.S3({}` with `new S3({}` instead. You can read more about the changes here (<https://github.com/AnomalyInnovations/serverless-stack-demo-client/pull/15>).

If you try deploying again, you should notice that your app bundle is a lot smaller.

Next we'll configure CloudFront to serve our app out globally.

For help and discussion

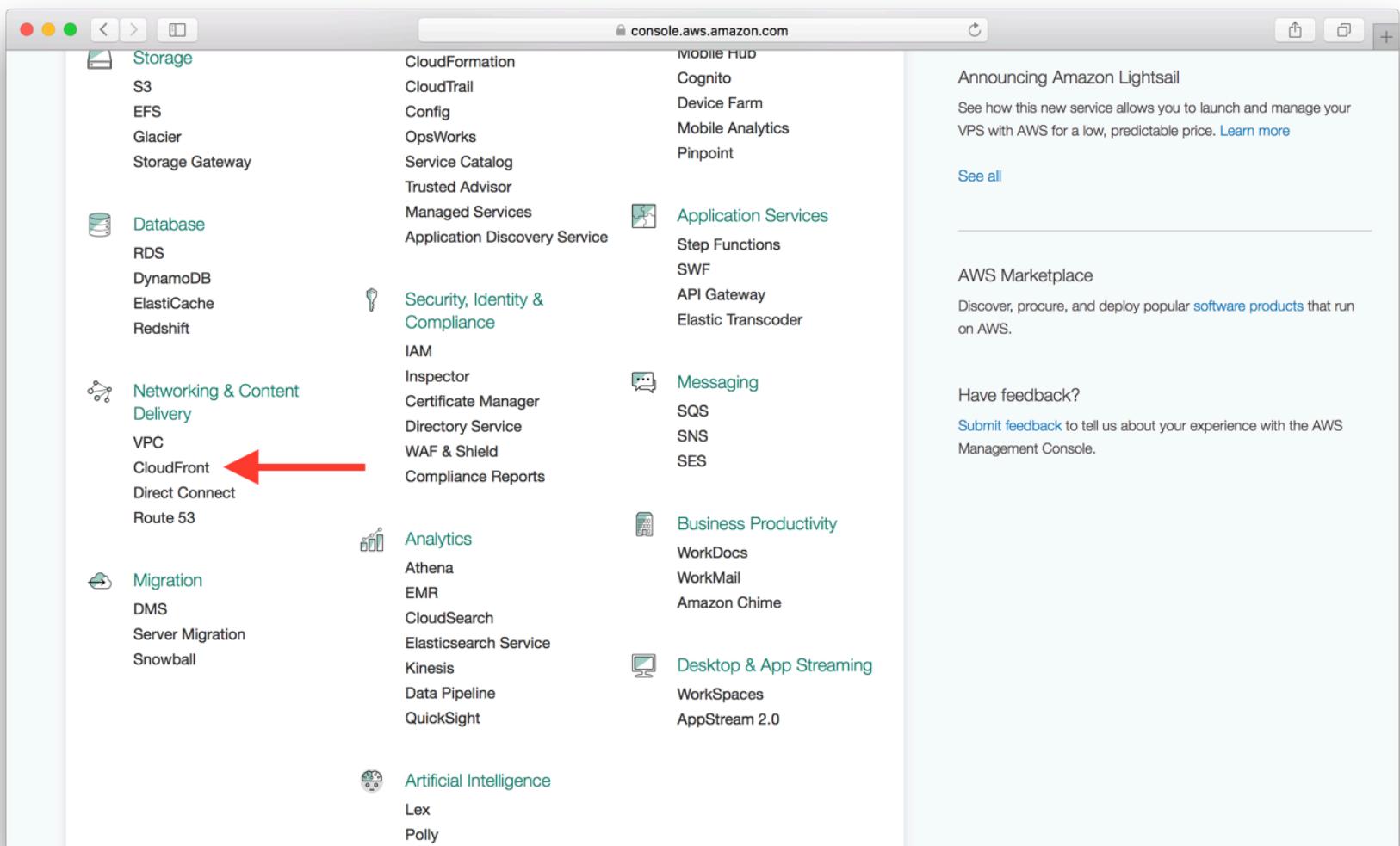
💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/63>)

Create a CloudFront Distribution

Now that we have our app up and running on S3, let's serve it out globally through CloudFront. To do this we need to create an AWS CloudFront Distribution.

Select CloudFront from the list of services in your AWS Console (<https://console.aws.amazon.com>).



Then select **Create Distribution**.

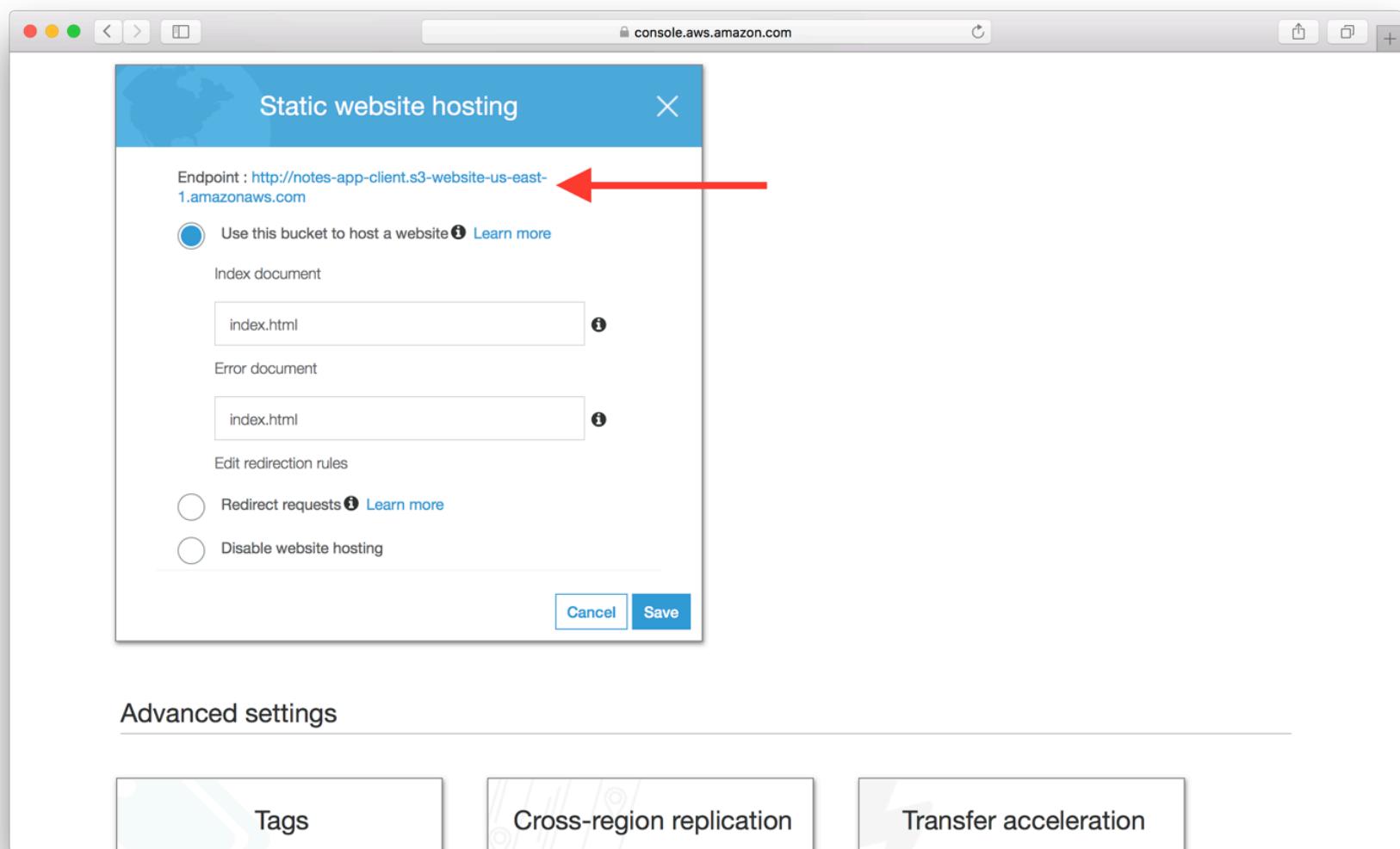
The screenshot shows the AWS CloudFront Distributions page. On the left sidebar, under the 'Distributions' section, there is a 'Create Distribution' button highlighted with a red arrow. The main area displays a table with columns for Delivery Method, ID, Domain Name, Comment, Origin, CNAMEs, Status, State, and Last Modified. The table header includes filters for 'Viewing', 'Any Delivery Method', 'Any State', and search fields.

And then in the Web section select **Get Started**.

The screenshot shows the 'Select a delivery method for your content' step in the CloudFront distribution creation wizard. The 'Web' tab is selected. The 'Get Started' button is highlighted with a red arrow. The page also includes sections for RTMP and a 'Cancel' button.

In the Create Distribution form we need to start by specifying the Origin Domain Name for our Web CloudFront Distribution. This field is pre-filled with a few options including the S3 bucket we created. But we are **not** going to select on the options in the dropdown. This is because the options here are the REST API endpoints for the S3 bucket instead of the one that is set up as a static website.

You can grab the S3 website endpoint from the **Static website hosting** panel for your S3 bucket. We had configured this in the previous chapter. Copy the URL in the **Endpoint** field.



And paste that URL in the **Origin Domain Name** field. In my case it is, `http://notes-app-client.s3-website-us-east-1.amazonaws.com`.

The screenshot shows the 'Create Distribution' page in the AWS CloudFront console. The left sidebar has 'Step 1: Select delivery method' and 'Step 2: Create distribution' sections. The main area is titled 'Create Distribution' and 'Origin Settings'. It includes fields for 'Origin Domain Name' (set to 'notes-app-client.s3-website-us-east-1.a'), 'Origin Path', 'Origin ID', and 'Origin Custom Headers'. Below this is the 'Default Cache Behavior Settings' section with options like 'Path Pattern', 'Viewer Protocol Policy' (set to 'HTTP and HTTPS'), 'Allowed HTTP Methods' (set to 'GET, HEAD'), 'Cached HTTP Methods' (set to 'GET, HEAD (Cached by default)'), 'Forward Headers' (set to 'None (Improves Caching)'), and 'Object Caching' (set to 'Use Origin Cache Headers'). A red arrow points to the 'Origin Domain Name' field.

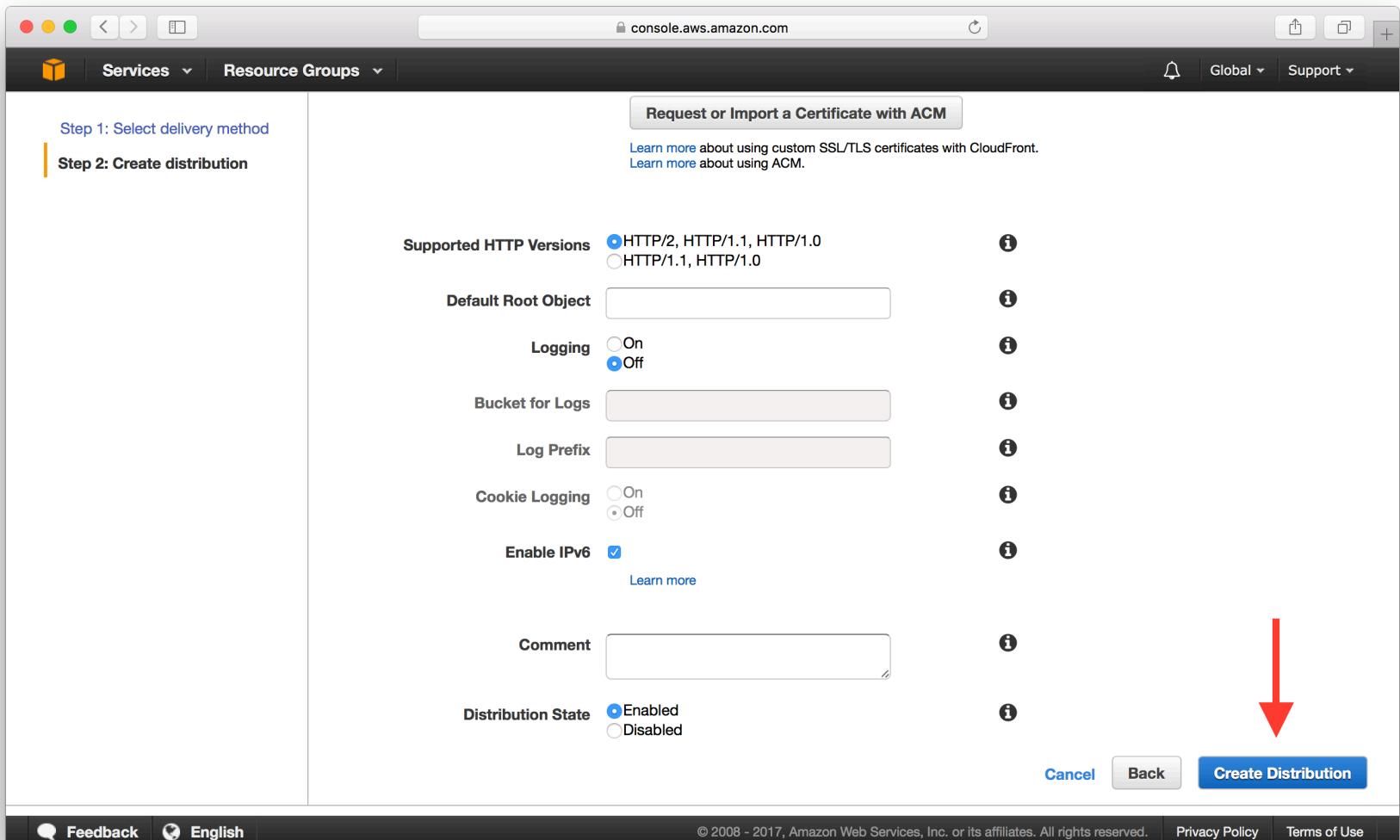
And now scroll down the form and switch **Compress Objects Automatically** to **Yes**. This will automatically Gzip compress the files that can be compressed and speed up the delivery of our app.

The screenshot shows the 'Step 2: Create distribution' section of the AWS CloudFront configuration. Under 'Delivery Method Settings', the 'Compress Objects Automatically' option is selected ('Yes') with a red arrow pointing to it. Other settings include 'Default TTL' (86400), 'Forward Cookies' (None), 'Query String Forwarding and Caching' (None), 'Smooth Streaming' (No), 'Restrict Viewer Access' (No), and a 'Lambda Function ARN' dropdown.

Next, scroll down a bit further to set the **Default Root Object** to `index.html`.

The screenshot shows the 'Step 2: Create distribution' section of the AWS CloudFront configuration. Under 'Distribution Settings', the 'Default Root Object' field is highlighted with a red arrow and contains the value `index.html`. Other settings include 'Supported HTTP Versions' (HTTP/2, HTTP/1.1, HTTP/1.0), 'Logging' (Off), 'Bucket for Logs' (empty), 'Log Prefix' (empty), 'Cookie Logging' (Off), 'Enable IPv6' (checked), and a 'Comment' field.

And finally, hit **Create Distribution**.



It takes AWS a little while to create a distribution. But once it is complete you can find your CloudFront Distribution by clicking on your newly created distribution from the list and looking up its domain name.

Screenshot of the AWS CloudFront Distribution configuration page. The left sidebar shows 'Distributions' selected. The main area shows 'CloudFront Distributions > E1KTCKT9SOAHBW'. The 'General' tab is selected. The distribution details include:

Distribution ID	E1KTCKT9SOAHBW
ARN	arn:aws:cloudfront::232771856781:distribution/E1KTCKT9SOAHBW
Log Prefix	-
Delivery Method	Web
Cookie Logging	Off
Distribution Status	InProgress
Comment	-
Price Class	Use All Edge Locations (Best Performance)
AWS WAF Web ACL	-
State	Enabled
Alternate Domain Names (CNAMEs)	
SSL Certificate	Default CloudFront Certificate (*.cloudfront.net)
Domain Name	d1r8102xi6mdx3.cloudfront.net
Custom SSL Client Support	-
Supported HTTP Versions	HTTP/2, HTTP/1.1, HTTP/1.0
IPv6	Enabled
Default Root Object	-
Last Modified	2017-02-16 18:42 UTC-5
Log Bucket	-

And if you navigate over to that in your browser, you should see your app live.

Screenshot of a web browser displaying the 'Scratch' application. The title bar shows the URL 'd1r8102xi6mdx3.cloudfront.net'. The page content includes:

Scratch

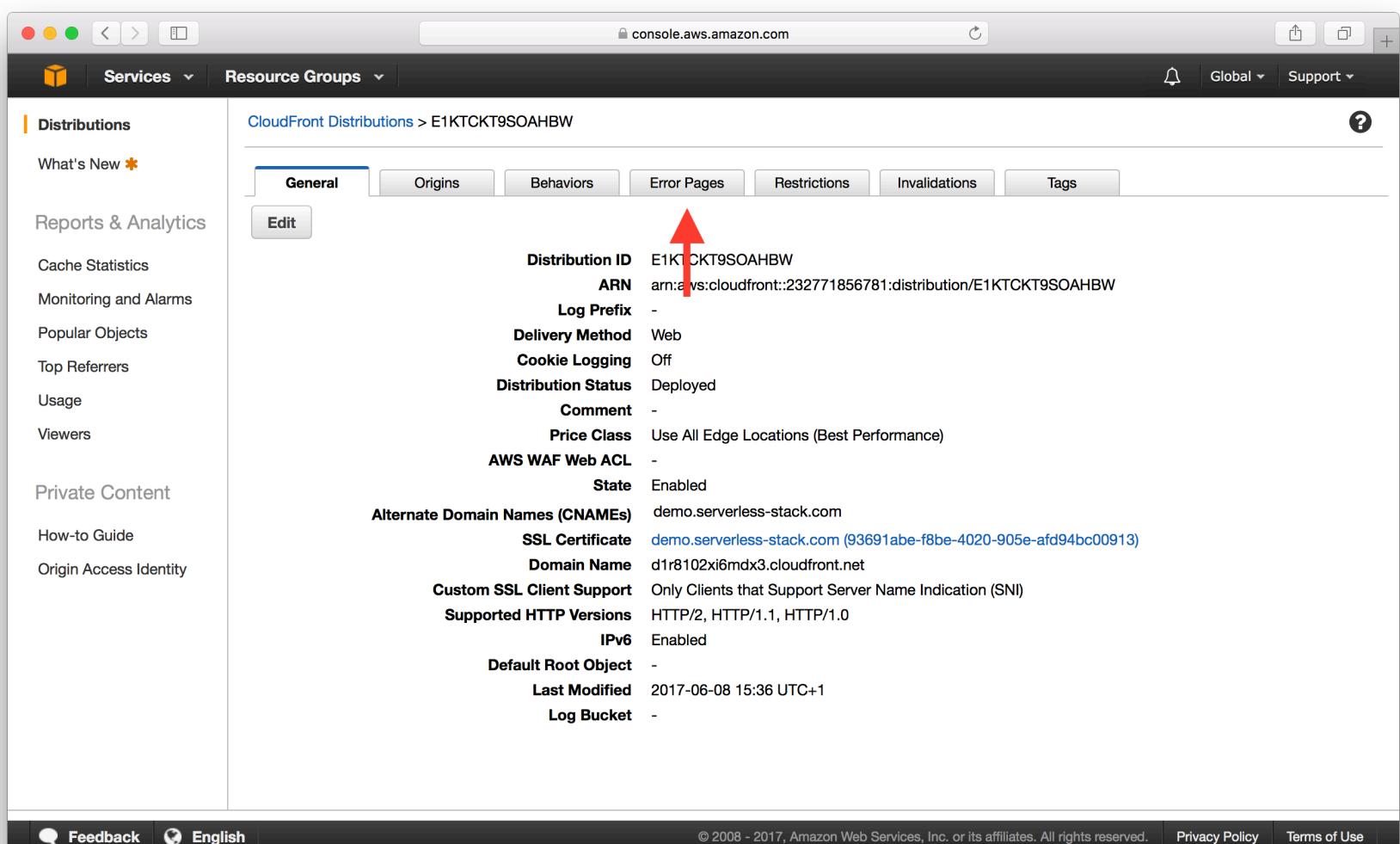
A simple note taking app

Now before we move on there is one last thing we need to do. Currently, our static website returns our `index.html` as the error page. We set this up back in the chapter where we created our S3 bucket. However, it returns a HTTP status code of 404 when it does so. We want to return the `index.html` but since the routing is handled by React Router; it does not make sense that we return the 404 HTTP status code. One of the issues with this is that certain corporate firewalls and proxies tend to block 4xx and 5xx responses.

Custom Error Responses

So we are going to create a custom error response and return a 200 status code instead. The downside of this approach is that we are going to be returning 200 even for cases where we don't have a route in our React Router. Unfortunately, there isn't a way around this. This is because CloudFront or S3 are not aware of the routes in our React Router.

To set up a custom error response, head over to the **Error Pages** tab in our Distribution.



The screenshot shows the AWS CloudFront Distribution configuration page. On the left, there's a sidebar with various service links like Services, Resource Groups, Distributions, and Reports & Analytics. The main area shows a distribution named "E1KTCKT9SOAHBW". At the top, there are tabs for General, Origins, Behaviors, Error Pages (which is highlighted with a red arrow), Restrictions, Invalidations, and Tags. Below the tabs, there are several configuration settings:

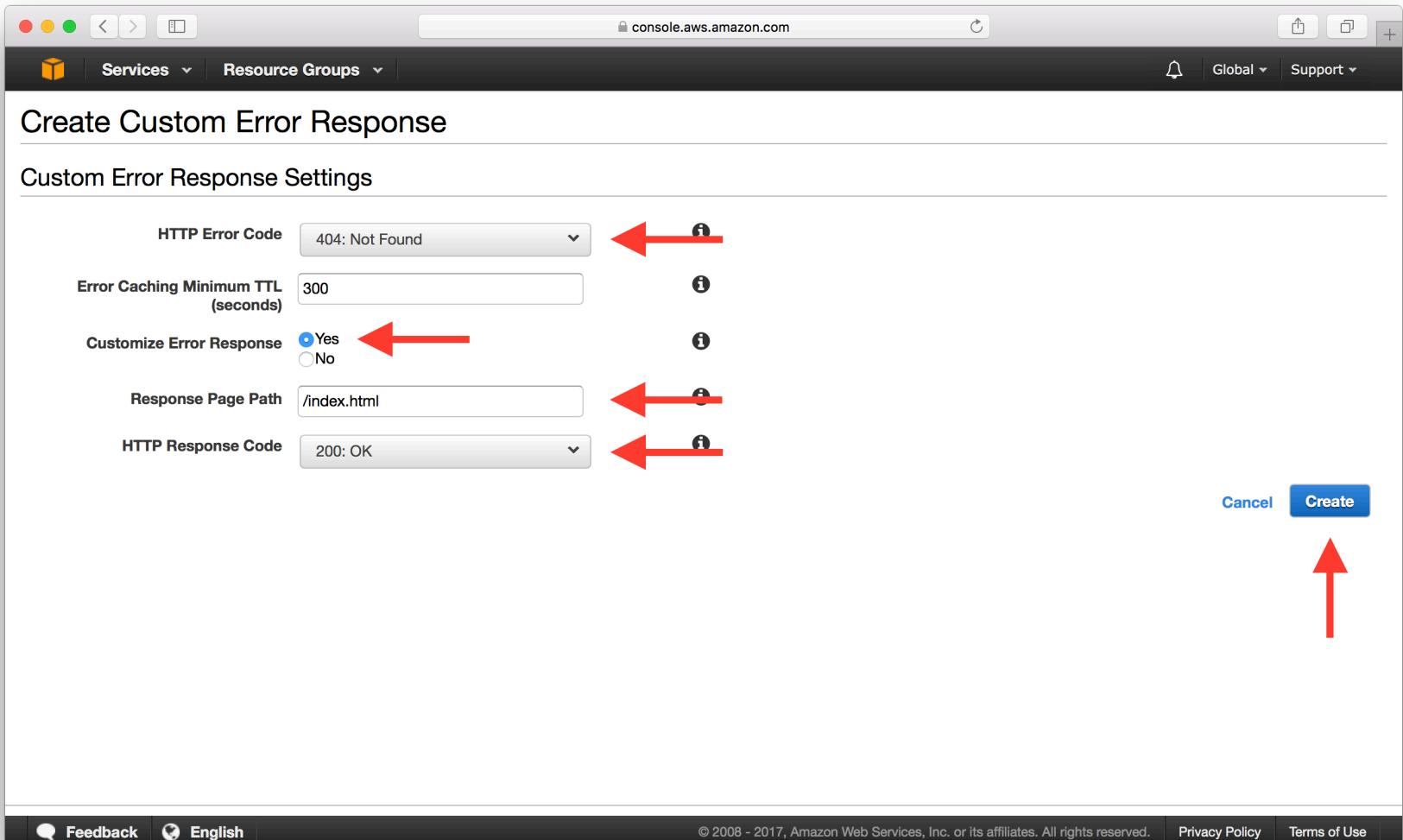
Setting	Value
Distribution ID	E1KTCKT9SOAHBW
ARN	arn:aws:cloudfront::232771856781:distribution/E1KTCKT9SOAHBW
Log Prefix	-
Delivery Method	Web
Cookie Logging	Off
Distribution Status	Deployed
Comment	-
Price Class	Use All Edge Locations (Best Performance)
AWS WAF Web ACL	-
State	Enabled
Alternate Domain Names (CNAMEs)	demo.serverless-stack.com
SSL Certificate	demo.serverless-stack.com (93691abe-f8be-4020-905e-afd94bc00913)
Domain Name	d1r8102x16mdx3.cloudfront.net
Custom SSL Client Support	Only Clients that Support Server Name Indication (SNI)
Supported HTTP Versions	HTTP/2, HTTP/1.1, HTTP/1.0
IPv6	Enabled
Default Root Object	-
Last Modified	2017-06-08 15:36 UTC+1
Log Bucket	-

And select **Create Custom Error Response**.

The screenshot shows the AWS CloudFront Distributions page with the distribution ID E3MQXGQ47VCJB0 selected. The 'Error Pages' tab is active. A red arrow points to the 'Create Custom Error Response' button. The table below shows one entry:

HTTP Error Code	Error Caching Minimum TTL	Response Page Path	HTTP Response Code
		No Data	

Pick **404** for the **HTTP Error Code** and select **Customize Error Response**. Enter **/index.html** for the **Response Page Path** and **200** for the **HTTP Response Code**.



And hit **Create**. This is basically telling CloudFront to respond to any 404 responses from our S3 bucket with the `index.html` and a 200 status code. Creating a custom error response should take a couple of minutes to complete.

Next up, let's point our domain to our CloudFront Distribution.

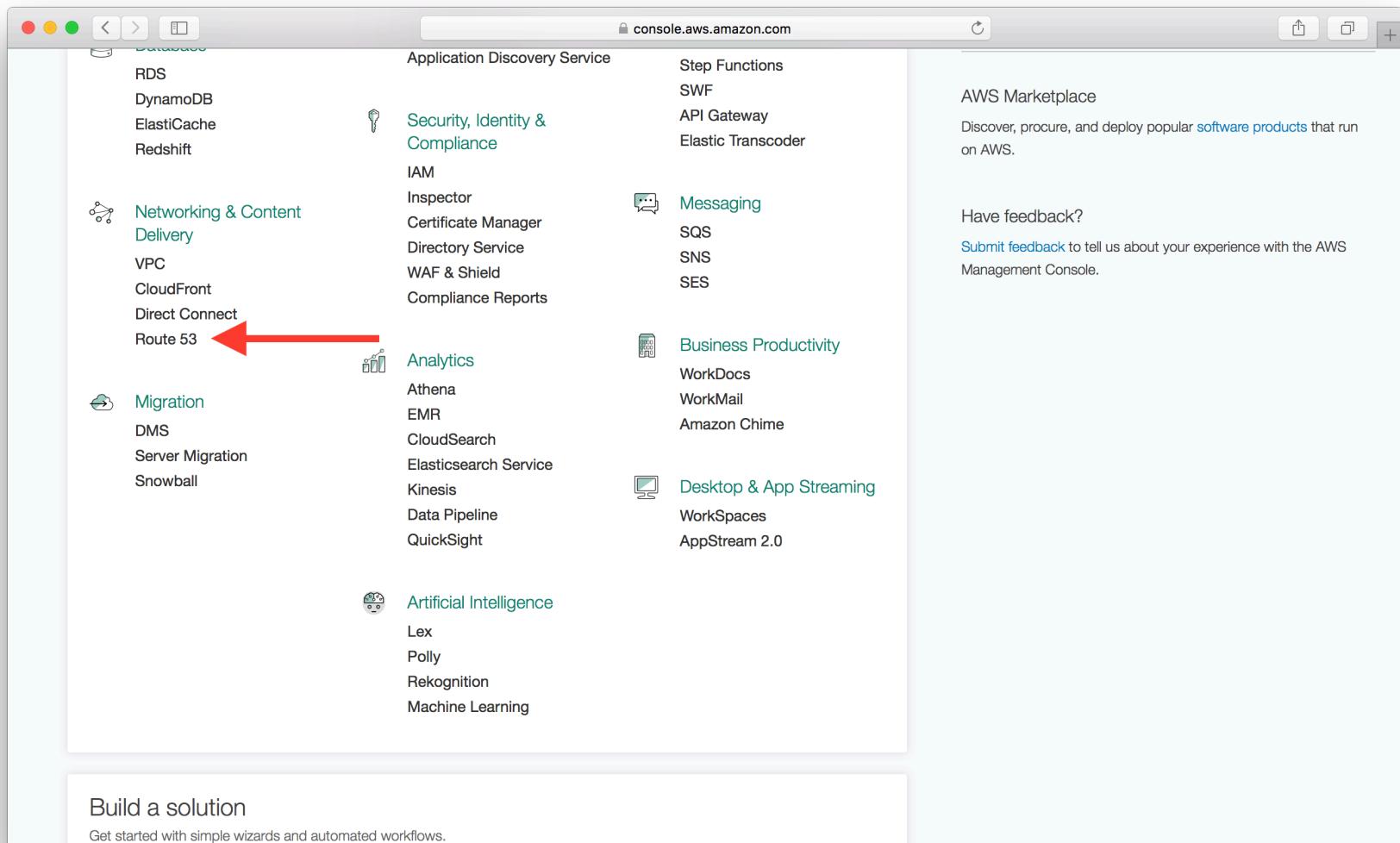
For help and discussion

Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/64>)

Set up Your Domain with CloudFront

Now that we have our CloudFront distribution live, let's set up our domain with it. You can purchase a domain right from the AWS Console (<https://console.aws.amazon.com>) by heading to the Route 53 section in the list of services.



Purchase a Domain with Route 53

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

The screenshot shows the AWS Route 53 service dashboard. On the left sidebar, there are several navigation options: Dashboard, Hosted zones, Health checks, Traffic flow, Traffic policies, Policy records, Domains, Registered domains, and Pending requests. The main content area is divided into four sections: DNS management, Traffic management, Availability monitoring, and Domain registration. The DNS management section displays a count of 13 Hosted zones. The Domain registration section displays a count of 5 Domains. A red arrow points upwards from the 'Check' button in the 'Register domain' section towards the top of the page.

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

The screenshot shows the 'Choose a domain name' step in the AWS Domain Registration process. The user has entered 'my-serverless-app' into the search bar. The results show that 'my-serverless-app.com' is available for \$12.00 per year. A red arrow points to the 'Add to cart' button next to this result. The sidebar on the left shows steps 1: Domain Search, 2: Contact Details, and 3: Review & Purchase. The right sidebar shows a shopping cart icon.

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

The screenshot shows the AWS Route 53 console with the 'Resource Groups' tab selected. A table titled 'Related Domain Suggestions' lists various domain names along with their status and price. Each row includes an 'Add to cart' button. To the right of the table, there is a sidebar with information about monthly fees for DNS management. At the bottom of the page, there are 'Cancel' and 'Continue' buttons, with a red arrow pointing to the 'Continue' button.

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

Monthly Fees for DNS Management
View pricing details for Route 53 queries and for the hosted zone that we create for each new domain.

Cancel **Continue** ←

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

Screenshot of the AWS Domain Purchase Step 3: Contact Details page.

The page shows fields for Address 2, Country (United States), State (California), City, and Postal/Zip Code. Under Privacy Protection, the "Hide contact information if the TLD registry, and the registrar, allow it" option is selected. A note explains that anyone can use a WHOIS query to see contact information for your domain. The "Continue" button is highlighted with a red arrow.

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

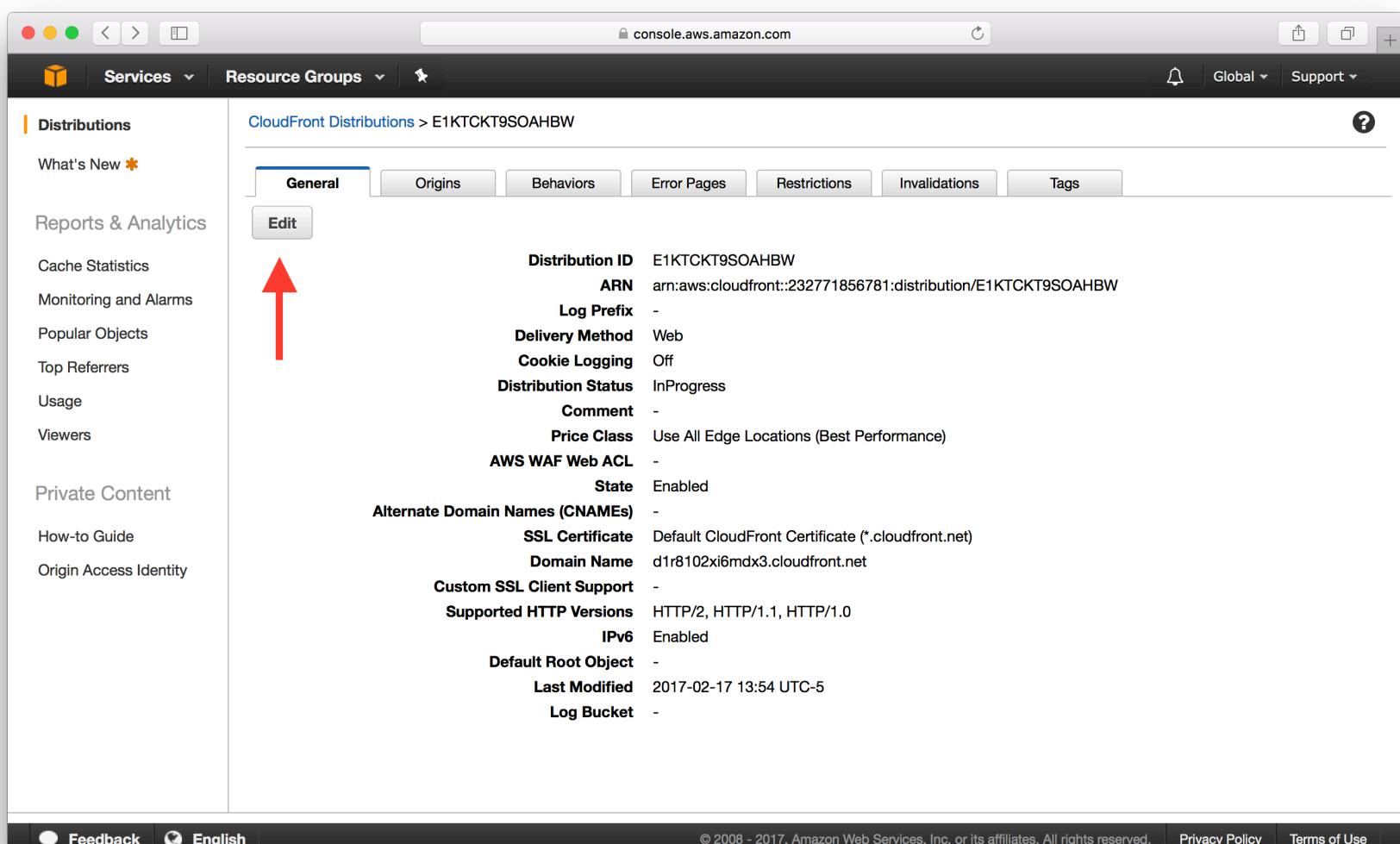
Screenshot of the AWS Domain Purchase Step 4: Review & Purchase page.

The left sidebar shows steps 1: Domain Search, 2: Contact Details, and 3: Review & Purchase. The main content area displays the "Review details and complete your purchase" section, which includes a note about assigning contacts to domains, managing DNS for the new domain, and terms and conditions. It also shows a "Shopping cart" summary for "my-serverless-app.com" with a one-time fee of \$12.00 for a 1-year registration. A red arrow points to the "I have read and agree to the AWS Domain Name Registration Agreement" checkbox. The "Complete Purchase" button is highlighted with a red arrow.

Next, we'll add an alternate domain name for our CloudFront Distribution.

Add Alternate Domain for CloudFront Distribution

Head over to the details of your CloudFront Distribution and hit **Edit**.



The screenshot shows the AWS CloudFront Distribution Details page. On the left, there's a sidebar with various links like 'Distributions', 'What's New', 'Reports & Analytics', etc. In the main area, the 'General' tab is selected under the 'Edit' section. A red arrow points to the 'Edit' button. The distribution details shown include:

Distribution ID	E1KTCKT9SOAHBW
ARN	arn:aws:cloudfront::232771856781:distribution/E1KTCKT9SOAHBW
Log Prefix	-
Delivery Method	Web
Cookie Logging	Off
Distribution Status	InProgress
Comment	-
Price Class	Use All Edge Locations (Best Performance)
AWS WAF Web ACL	-
State	Enabled
Alternate Domain Names (CNAMEs)	
SSL Certificate	Default CloudFront Certificate (*.cloudfront.net)
Domain Name	d1r8102xi6mdx3.cloudfront.net
Custom SSL Client Support	-
Supported HTTP Versions	HTTP/2, HTTP/1.1, HTTP/1.0
IPv6	Enabled
Default Root Object	-
Last Modified	2017-02-17 13:54 UTC-5
Log Bucket	-

And type in your new domain name in the **Alternate Domain Names (CNAMEs)** field.

Services ▾ Resource Groups ▾

Edit Distribution

Distribution Settings

Price Class: Use All Edge Locations (Best Performance) i

AWS WAF Web ACL: None i

Alternate Domain Names (CNAMEs): i ←

SSL Certificate: Default CloudFront Certificate (*.cloudfront.net)
Choose this option if you want your users to use HTTPS or HTTP to access your content with the CloudFront domain name (such as https://d111111abcdef8.cloudfront.net/logo.jpg).
Important: If you choose this option, CloudFront requires that browsers or devices support TLSv1 or later to access your content.

Custom SSL Certificate (example.com):
Choose this option if you want your users to access your content by using an alternate domain name, such as https://www.example.com/logo.jpg.
You can use a certificate stored in AWS Certificate Manager (ACM) in the US East (N. Virginia) Region, or you can use a certificate stored in IAM.

↻

Request or Import a Certificate with ACM

Learn more about using custom SSL/TLS certificates with CloudFront.
Learn more about using ACM.

Feedback English © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Scroll down and hit Yes, Edit to save the changes.

Services ▾ Resource Groups ▾

Request or Import a Certificate with ACM

Learn more about using custom SSL/TLS certificates with CloudFront.
Learn more about using ACM.

Supported HTTP Versions: HTTP/2, HTTP/1.1, HTTP/1.0 i
 HTTP/1.1, HTTP/1.0

Default Root Object:

Logging: On i
 Off i

Bucket for Logs:

Log Prefix:

Cookie Logging: On i
 Off i

Enable IPv6: i
Learn more

Comment:

Distribution State: Enabled i
 Disabled

Yes, Edit Cancel

Feedback English © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Next, let's point our domain to the CloudFront Distribution.

Point Domain to CloudFront Distribution

Head back into Route 53 and hit the **Hosted Zones** button. If you don't have an existing **Hosted Zone**, you'll need to create one by adding the **Domain Name** and selecting **Public Hosted Zone** as the **Type**.

The screenshot shows the AWS Route 53 dashboard. On the left sidebar, under the 'DNS management' section, there is a red arrow pointing upwards towards the 'Hosted zones' link. The 'Hosted zones' link is highlighted in blue and has a count of '13' next to it. Below this, there is a 'Register domain' section with a search bar and a 'Check' button. To the right of the 'Hosted zones' section are three other tabs: 'Traffic management', 'Availability monitoring', and 'Domain registration'. The 'Domain registration' tab shows a count of '5' domains. At the bottom of the dashboard, there are links for 'More info', 'Developer Guide', 'FAQs', 'Pricing', 'Forum - DNS and health checks', 'Forum - Domain name registration', 'Request a limit increase', 'Service health' (which shows 'Amazon Route 53' is operating normally), and 'AWS service health dashboard'.

Select your domain from the list and hit **Create Record Set** in the details screen.

The screenshot shows the AWS CloudFront DNS interface. On the left, a sidebar lists services: Dashboard, Hosted zones (selected), Health checks, Traffic flow, Traffic policies, Policy records, Domains, Registered domains, and Pending requests. The main area displays a record set named "Weighted Only". A red arrow points to the "Type" dropdown menu in the search bar. The record set table shows the following data:

Type	Value
A	192.30.252.153 192.30.252.154
NS	ns-1667.awsdns-16.co.uk. ns-1284.awsdns-32.org. ns-229.awsdns-28.com. ns-741.awsdns-28.net.
SOA	ns-1667.awsdns-16.co.uk. awsdns-hostmaster.awa...

To the right, a note says: "To get started, click Create Record Set button or click an existing record set." The bottom navigation bar includes links for Feedback, English, Copyright notice (© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.), Privacy Policy, and Terms of Use.

Leave the **Name** field empty since we are going to point our bare domain (without the www.) to our CloudFront Distribution.

The screenshot shows the AWS Route 53 'Create Record Set' interface. On the left, there's a sidebar with options like Dashboard, Hosted zones, Health checks, Traffic flow, Traffic policies, Policy records, Domains, Registered domains, and Pending requests. The 'Hosted zones' option is selected. In the main area, there's a search bar for 'Record Set Name' and a dropdown for 'Type'. A red arrow points to the 'Type' dropdown, which is currently set to 'A - IPv4 address'. Below it, there's a section for 'Alias' with a radio button set to 'Yes'. Other fields include 'TTL (Seconds)', 'Value' (with examples), 'Routing Policy' (set to 'Simple'), and a 'Create' button.

And select **Alias** as **Yes** since we are going to simply point this to our CloudFront domain.

This screenshot is identical to the one above, showing the 'Create Record Set' interface. A red arrow points to the 'Alias' field, which has the radio button set to 'Yes'. Below it, the 'Alias Target' field is highlighted with a red arrow, containing the placeholder text 'Enter target name'. The rest of the interface elements are visible, including the 'Name' field (serverless-stack.com), 'Type' dropdown (A - IPv4 address), 'TTL (Seconds)' field, 'Value' examples, 'Routing Policy' dropdown, and the 'Create' button.

In the Alias Target dropdown, select your CloudFront Distribution.

The screenshot shows the AWS Route 53 console. On the left, the navigation pane includes options like Dashboard, Hosted zones, Health checks, Traffic flow, Traffic policies, Policy records, Domains, Registered domains, and Pending requests. The 'Hosted zones' option is selected. In the main content area, there's a search bar for 'Record Set Name' and a dropdown for 'Type' set to 'Any Type'. A checkbox for 'Aliases Only' is unchecked. Below this, a table titled 'Weighted Only' displays four record sets. The first record set is of type 'A' with values '192.30.252.153' and '192.30.252.154'. The second is of type 'NS' with values 'ns-1667.awsdns-16.co.uk.', 'ns-1284.awsdns-32.org.', 'ns-229.awsdns-28.com.', and 'ns-741.awsdns-28.net.'. The third is of type 'SOA' with values 'ns-1667.awsdns-16.co.uk.' and 'awsdns-hostmaster.amazon.com.'. To the right of the table is a 'Create Record Set' form. The 'Name:' field contains 'serverless-stack.com.'. The 'Type:' dropdown is set to 'A – IPv4 address'. The 'Alias:' section has a radio button for 'Yes' selected. A dropdown menu for 'Alias Target:' is open, showing a list of targets. An arrow points from the text 'In the Alias Target dropdown, select your CloudFront Distribution.' to the 'CloudFront distributions' item in the dropdown list. Other items in the list include 'ELB Classic load balancers', 'Elastic Beanstalk environments', and 'S3 website endpoints'. At the bottom of the form is a 'Create' button.

Finally, hit **Create** to add your new record set.

The screenshot shows the AWS Route 53 service interface. On the left, there's a sidebar with options like Dashboard, Hosted zones (which is selected), Health checks, Traffic flow, Traffic policies, Policy records, Domains, Registered domains, and Pending requests. The main area is titled 'Create Record Set' and shows a table of record sets. One record set is displayed in detail:

Type	Value
A	192.30.252.153 192.30.252.154
NS	ns-1667.awsdns-16.co.uk. ns-1284.awsdns-32.org. ns-229.awsdns-28.com. ns-741.awsdns-28.net.
SOA	ns-1667.awsdns-16.co.uk. awsdns-hostmaster.ama

On the right, there are configuration fields for a new record set:

- Name:** serverless-stack.com.
- Type:** A – IPv4 address
- Alias:** Yes (radio button selected)
- Alias Target:** d1r8102xi6mdx3.cloudfront.net
- Alias Hosted Zone ID:** Z2FDTNDATAQYW2
- Routing Policy:** Simple
- Evaluate Target Health:** No (radio button selected)

A large red arrow points to the 'Create' button at the bottom right of the form.

Add IPv6 Support

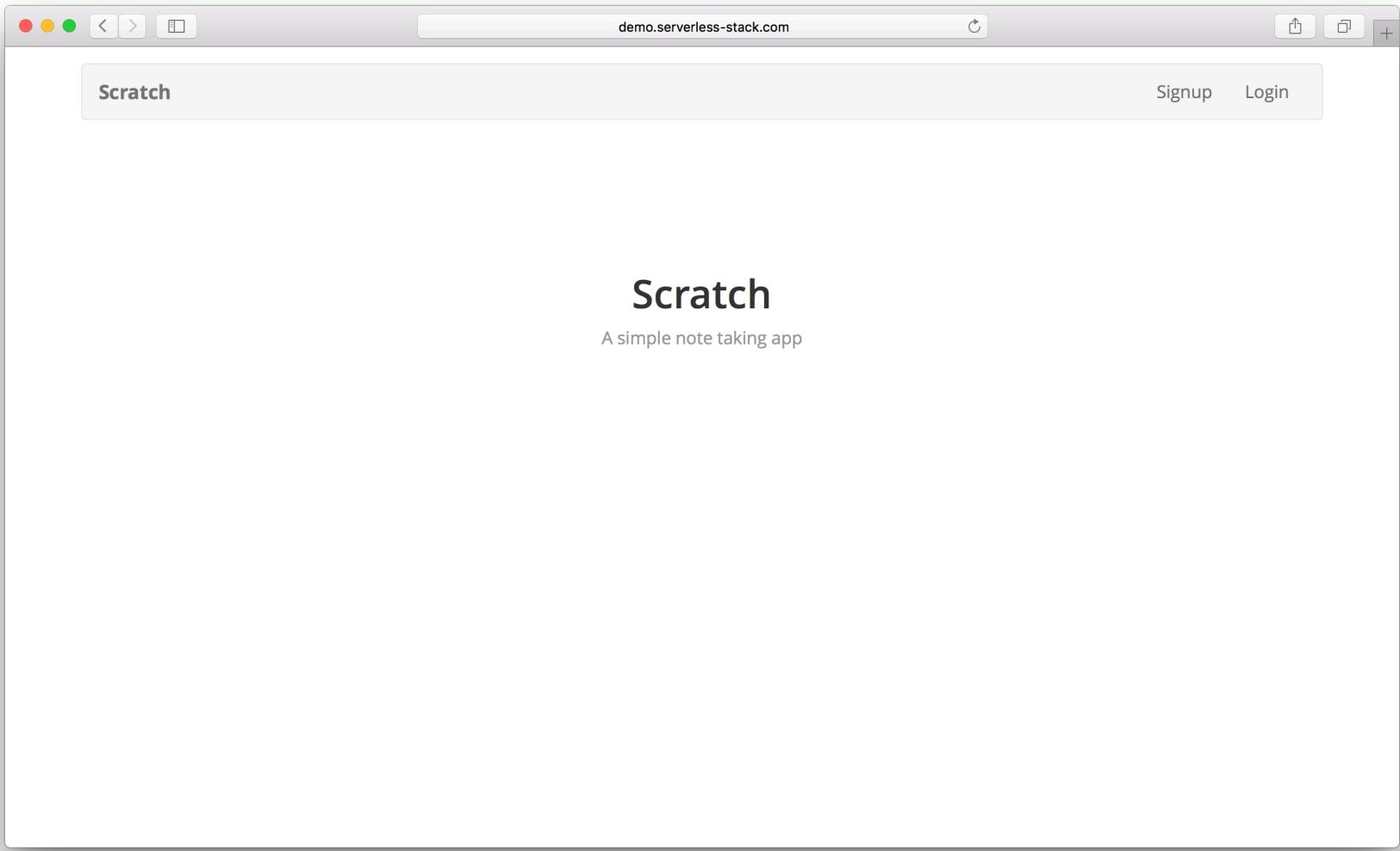
CloudFront Distributions have IPv6 enabled by default and this means that we need to create an AAAA record as well. It is set up exactly the same way as the Alias record.

Create a new Record Set with the exact settings as before, except make sure to pick **AAAA - IPv6 address** as the **Type**.

The screenshot shows the AWS Route 53 'Create Record Set' interface. On the left, a sidebar lists various services like Dashboard, Hosted zones, and Domains. The 'Hosted zones' option is selected. The main area shows a table of existing record sets for the domain 'serverless-stack.com'. A red arrow points to the 'Type' dropdown in the 'Create Record Set' form, which is set to 'AAAA – IPv6 address'. The 'Alias' section is expanded, showing the 'Alias Target' dropdown menu open with options like 'CloudFront distributions', 'ELB Classic load balancers', and 'Record sets in this hosted zone'. At the bottom right of the form is a 'Create' button.

And hit **Create** to add your AAAA record set.

It can take around an hour to update the DNS records but once it's done, you should be able to access your app through your domain.



Next up, we'll take a quick look at ensuring that our www. domain also directs to our app.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/65>)

Set up WWW Domain Redirect

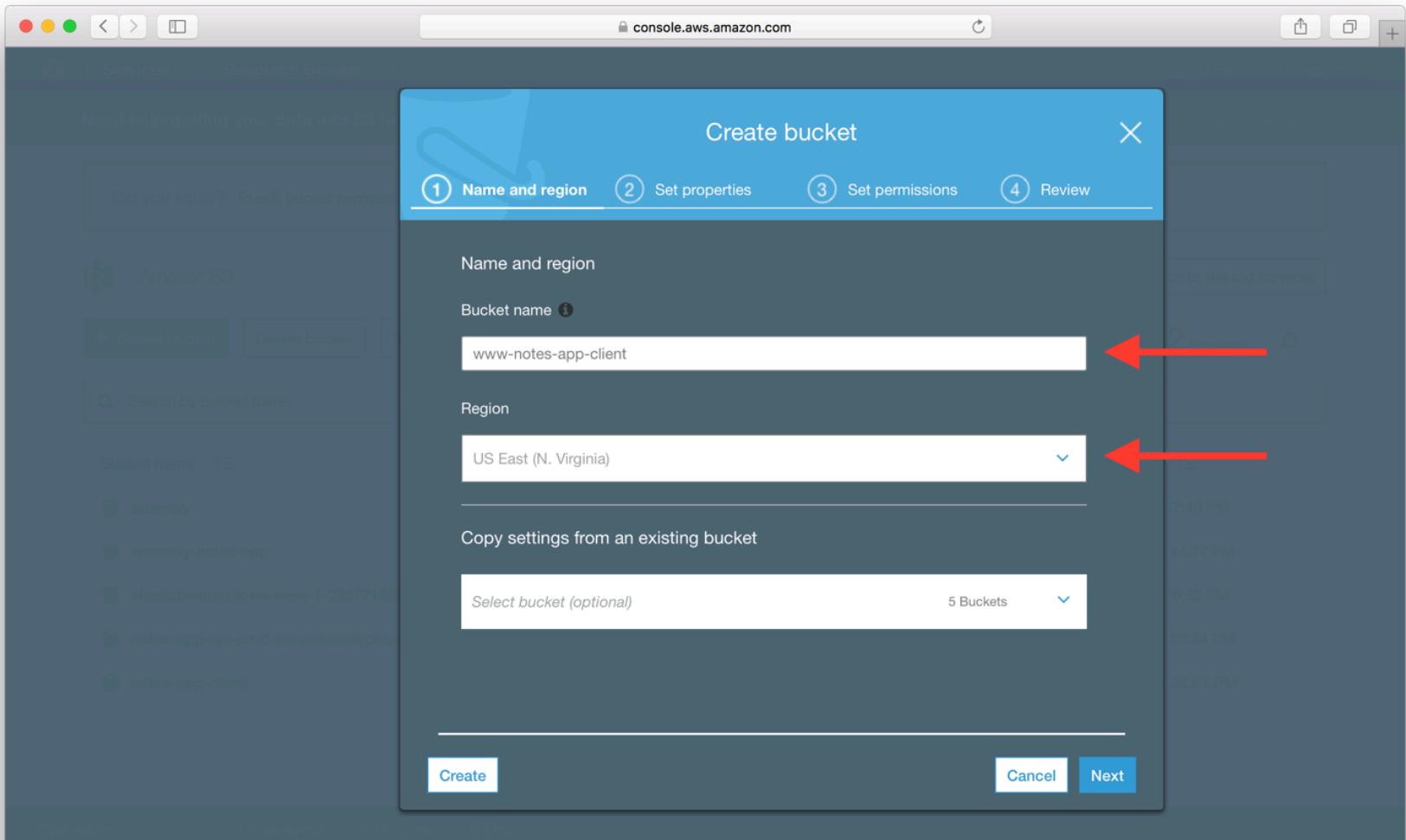
There's plenty of debate over the www vs non-www domains and while both sides have merit; we'll go over how to set up another domain (in this case the www) and redirect it to our original. The reason we do a redirect is to tell the search engines that we only want one version of our domain to appear in the search results. If you prefer having the www domain as the default simply swap this step with the last one where we created a bare domain (non-www).

To create a www version of our domain and have it redirect we are going to create a new S3 Bucket and a new CloudFront Distribution. This new S3 Bucket will simply respond with a redirect to our main domain using the redirection feature that S3 Buckets have.

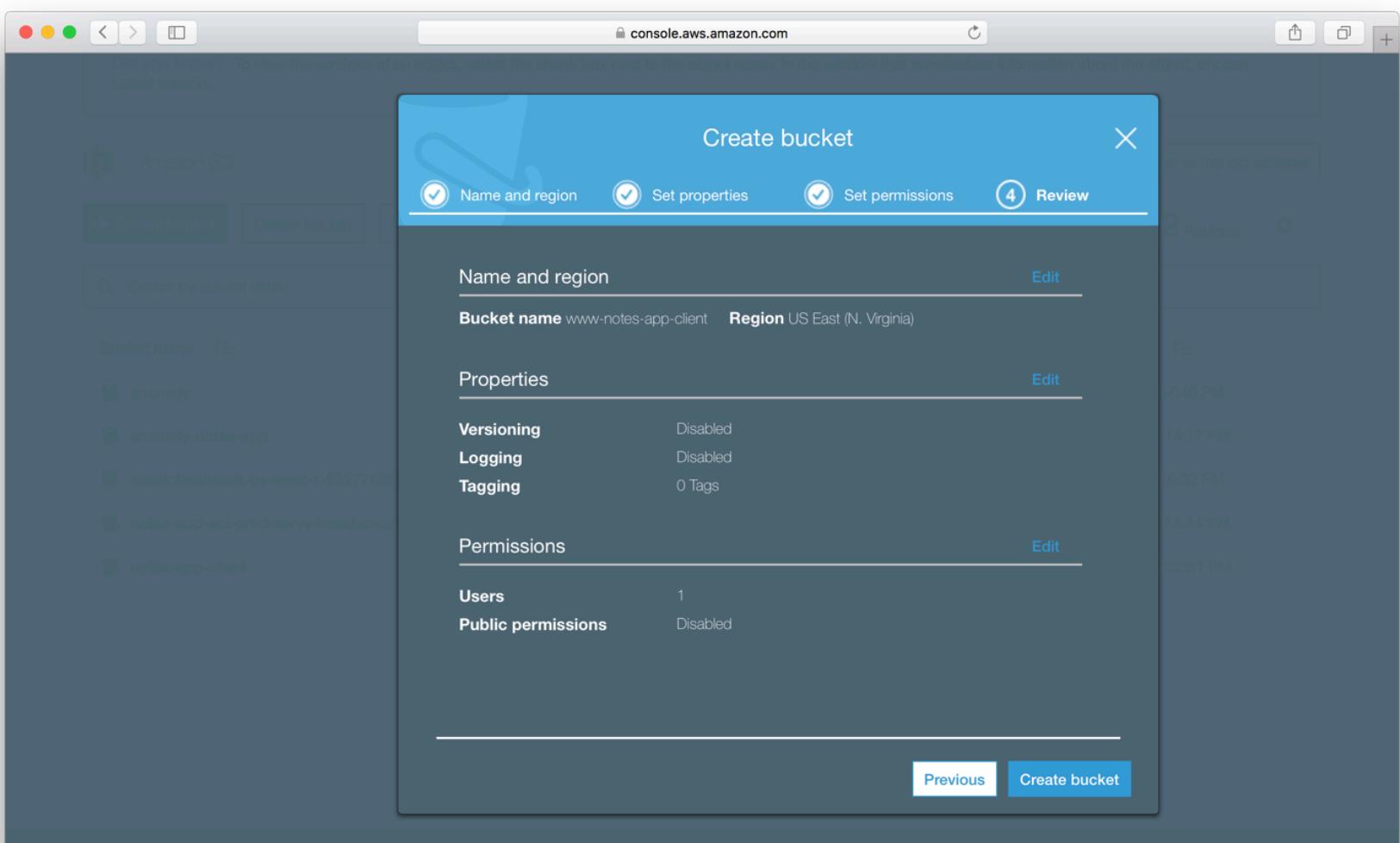
So let's start by creating a new S3 redirect Bucket for this.

Create S3 Redirect Bucket

Create a **new S3 Bucket** through the AWS Console (<https://console.aws.amazon.com>). The name doesn't really matter but it pick something that helps us distinguish between the two. Again, remember that we need a separate S3 Bucket for this step and we cannot use the original one we had previously created.



Next just follow through the steps and leave the defaults intact.

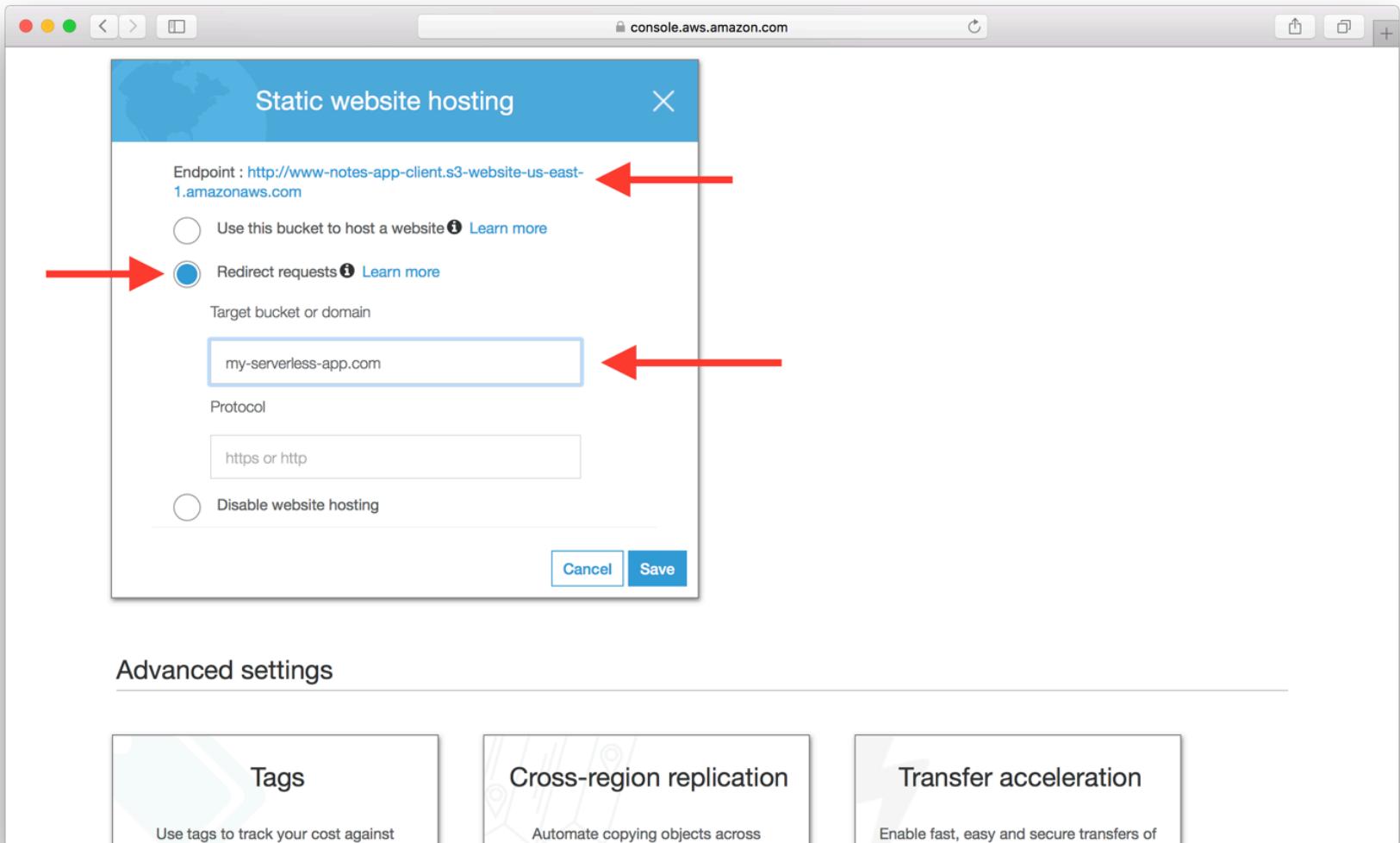


Now go into the **Properties** of the new bucket and click on the **Static website hosting**.

The screenshot shows the AWS S3 Properties page for a bucket named 'www-notes-app-client'. The 'Properties' tab is selected. On the right side, there are several sections: 'Versioning' (disabled), 'Logging' (disabled), 'Static website hosting' (disabled, highlighted with a red arrow), 'Advanced settings' (Tags, Cross-region replication, Transfer acceleration), and 'Endpoint' (disabled).

But unlike last time we are going to select the **Redirect requests** option and fill in the domain we are going to be redirecting towards. This is the domain that we set up in our last chapter.

Also, make sure to copy the **Endpoint** as we'll be needing this later.



And hit **Save** to make the changes. Next we'll create a CloudFront Distribution to point to this S3 redirect Bucket.

Create a CloudFront Distribution

Create a new **CloudFront Distribution**. And copy the **S3 Endpoint** from the step above as the **Origin Domain Name**. Make sure to **not** use the one from the dropdown. In my case, it is

`http://www-notes-app-client.s3-website-us-east-1.amazonaws.com`.

Step 1: Select delivery method

Step 2: Create distribution

Create Distribution

Origin Settings

Origin Domain Name

Origin Path

Origin ID S3-Website-www-notes-app-client.s3-w

Origin Custom Headers

Header Name	Value	Info
<input type="text"/>	<input type="text"/>	

Default Cache Behavior Settings

Path Pattern Default (*)

Viewer Protocol Policy HTTP and HTTPS Redirect HTTP to HTTPS HTTPS Only

Allowed HTTP Methods GET, HEAD GET, HEAD, OPTIONS GET, HEAD, OPTIONS, PUT, POST, PATCH, DELETE

Cached HTTP Methods GET, HEAD (Cached by default)

Forward Headers None (Improves Caching)

Object Caching Use Origin Cache Headers

Feedback English © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Scroll down to the **Alternate Domain Names (CNAMEs)** and use the www version of our domain name here.

Screenshot of the AWS CloudFront Distribution Settings page. The 'Alternate Domain Names (CNAMEs)' field contains 'www.my-serverless-app.com'. A red arrow points to this field.

Step 1: Select delivery method

Step 2: Create distribution

Distribution Settings

Price Class: Use All Edge Locations (Best Performance)

AWS WAF Web ACL: None

Alternate Domain Names (CNAMEs): www.my-serverless-app.com

SSL Certificate: Default CloudFront Certificate (*.cloudfront.net)

Choose this option if you want your users to use HTTPS or HTTP to access your content with the CloudFront domain name (such as <https://d111111abcdef8.cloudfront.net/logo.jpg>).
Important: If you choose this option, CloudFront requires that browsers or devices support TLSv1 or later to access your content.

Custom SSL Certificate (example.com):

Choose this option if you want your users to access your content by using an alternate domain name, such as <https://www.example.com/logo.jpg>. You can use a certificate stored in AWS Certificate Manager (ACM) in the US East (N. Virginia) Region, or you can use a certificate stored in IAM.

demo.serverless-stack.com (ccfc6aa7-6d17...)

Request or Import a Certificate with ACM

Learn more about using custom SSL/TLS certificates with CloudFront.
Learn more about using ACM.

And hit Create Distribution.

Screenshot of the AWS CloudFront Create Distribution page. The 'Create Distribution' button is highlighted with a red arrow.

Step 1: Select delivery method

Step 2: Create distribution

Request or Import a Certificate with ACM

Learn more about using custom SSL/TLS certificates with CloudFront.
Learn more about using ACM.

Supported HTTP Versions: HTTP/2, HTTP/1.1, HTTP/1.0

Default Root Object:

Logging: Off

Bucket for Logs:

Log Prefix:

Cookie Logging: Off

Enable IPv6:

Comment:

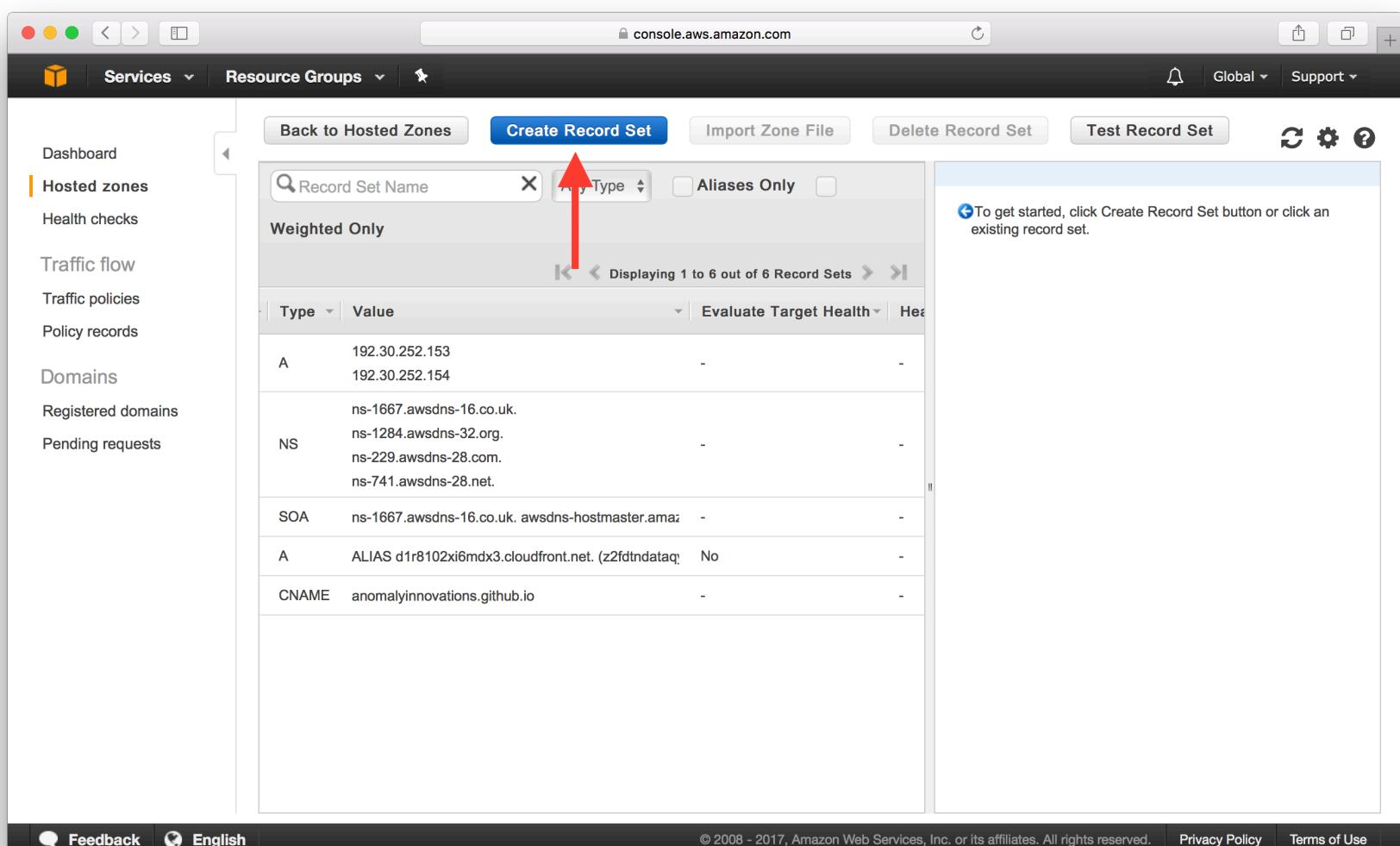
Distribution State: Enabled

Create Distribution

Finally, we'll point our www domain to this CloudFront Distribution.

Point WWW Domain to CloudFront Distribution

Head over to your domain in Route 53 and hit **Create Record Set**.



This time fill in **www** as the **Name** and select **Alias** as **Yes**. And pick your new CloudFront Distribution from the **Alias Target** dropdown.

The screenshot shows the AWS Route 53 service dashboard under the 'Hosted zones' section. On the left, there's a sidebar with various navigation links. The main area displays a list of record sets for a specific zone, with columns for Type, Value, and other details. To the right, a modal window titled 'Create Record Set' is open. It has fields for 'Name' (set to 'www.demo.serverless-stack.com.') and 'Type' (set to 'A - IPv4 address'). Below these, there's an 'Alias' section with a radio button for 'Yes' (selected) and 'No'. An 'Alias Target' dropdown is open, showing suggestions like 'CloudFront distributions' and 'Elastic Beanstalk environments'. At the bottom of the modal is a 'Create' button.

Add IPv6 Support

Just as before, we need to add an AAAA record to support IPv6.

Create a new Record Set with the exact same settings as before, except make sure to pick **AAAA - IPv6 address** as the **Type**.

The screenshot shows the AWS Route 53 service dashboard under the 'Resource Groups' section. On the left sidebar, 'Hosted zones' is selected. In the main area, there's a table of existing record sets for the domain 'serverless-stack.com'. To the right, a 'Create Record Set' form is open. The 'Name' field contains 'www.demo'. A red arrow points to this field. The 'Type' dropdown is set to 'AAAA – IPv6 address'. The 'Alias' section is expanded, showing 'Alias Target:' dropdown with options like 'CloudFront distributions', 'ELB Classic load balancers', and 'Record sets in this hosted zone'. The 'Routing Policy' section is also visible. At the bottom right of the form is a 'Create' button.

And that's it! Just give it some time for the DNS to propagate and if you visit your www version of your domain, it should redirect you to your non-www version.

Next, we'll set up SSL and add HTTPS support for our domains.

For help and discussion

[Comments on this chapter](#)

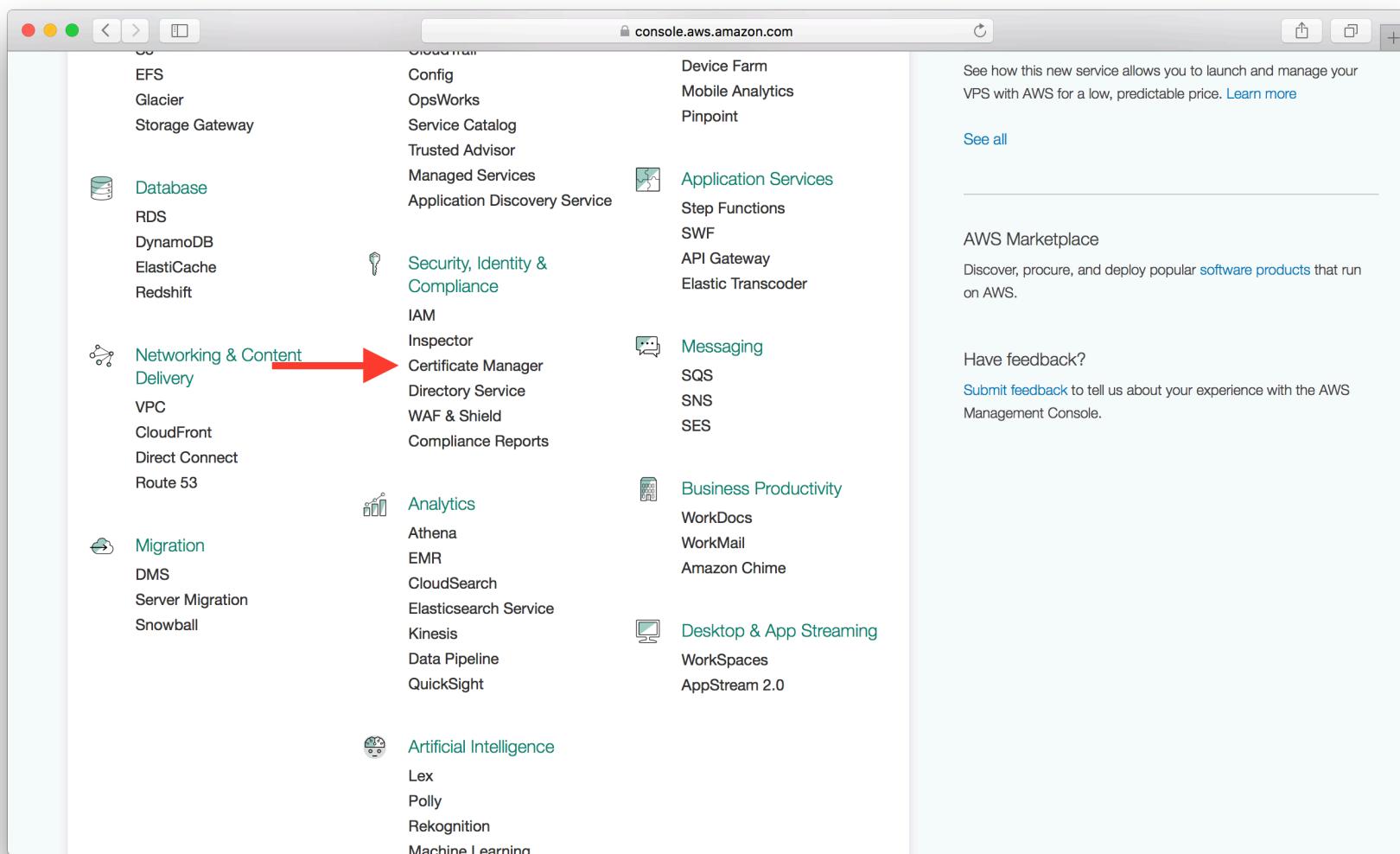
(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/66>)

Set up SSL

Now that our app is being served through our domain, let's add a layer of security to it by switching to HTTPS. AWS makes this fairly easy to do, thanks to Certificate Manager.

Request a Certificate

Select **Certificate Manager** from the list of services in your AWS Console (<https://console.aws.amazon.com>). Ensure that you are in the **US East (N. Virginia)** region. This is because a certificate needs to be from this region for it to work with CloudFront (<http://docs.aws.amazon.com/acm/latest/userguide/acm-regions.html>).



If this is your first certificate, you'll need to hit **Get started**. If not then hit **Request a certificate** from the top.

S | Services | Resource Groups | N. Virginia | Support

AWS Certificate Manager

AWS Certificate Manager (ACM) makes it easy to provision, manage, deploy, and renew SSL/TLS certificates on the AWS platform.

[Get started](#)

[User guide](#)



Provision certificates

Provide the name of your site, establish your identity, and let ACM do the rest. ACM manages renewal of SSL/TLS certificates issued by Amazon for you.



Deploy SSL/TLS-based sites and applications

Create an Elastic Load Balancer or Amazon CloudFront distribution and use ACM-provided or imported certificates with SSL/TLS to securely identify your site.



Manage certificates

See all of your ACM-provided and imported certificates in one place in the AWS Management Console. Automate management tasks by using the ACM API, SDK, or CLI.

And type in the name of our domain. Hit **Add another name to this certificate** and add our www version of our domain as well. Hit **Review and request** once you are done.

The screenshot shows the AWS Certificate Manager interface. The top navigation bar includes 'Services', 'Resource Groups', and 'Support'. The main title is 'Request a certificate'. On the left, a sidebar lists 'Step 1: Add domain names', 'Step 2: Review and request', and 'Step 3: Validation'. A note in a blue box states: 'You can use AWS Certificate Manager certificates only with Elastic Load Balancing and Amazon CloudFront. [Learn more.](#)'

The central section is titled 'Add domain names'. It contains a text input field with placeholder 'Domain name*' and a 'Remove' button. Two red arrows point to the inputs 'www.my-serverless-app.com' and 'my-serverless-app.com'. Below these inputs is a blue button labeled 'Add another name to this certificate' with a red arrow pointing to it. A note below the button says: 'You can add additional names to this certificate. For example, if you're requesting a certificate for "www.example.com", you might want to add the name "example.com" so that customers can reach your site by either name. [Learn more.](#)'

At the bottom, a required field notice reads '*At least one domain name is required'. To the right are 'Cancel' and 'Review and request' buttons. The footer includes links for 'Feedback', 'English', 'Privacy Policy', and 'Terms of Use'.

On the next screen review to make sure you filled in the right domain names and hit **Confirm and request**.

Screenshot of the AWS Certificate Manager 'Review and request' step. The page shows the domain names being requested: www.my-serverless-app.com and my-serverless-app.com. The 'Confirm and request' button is highlighted with a red arrow.

Request a certificate

Step 1: Add domain names

Step 2: Review and request

Step 3: Validation

Review and request

After you request the certificate, email will be sent to the registered owner of each domain name below. The domain owner or an authorized representative can validate control of the domain and approve the certificate by following the instructions in the body of the email. After all of the domains are validated, the certificate will be issued.

Domain name

The names you want to secure with an SSL/TLS certificate.

Domain name: www.my-serverless-app.com
Additional name: my-serverless-app.com

Cancel Previous Confirm and request

Feedback English © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

And finally on the **Validation** screen, AWS let's you know which email addresses it's going to send emails to verify that it is your domain. Hit **Continue**, to send the verification emails.

Screenshot of the AWS Certificate Manager 'Request a certificate' page. The page shows a step-by-step process:

- Step 1: Add domain names
- Step 2: Review and request
- Step 3: Validation**

The main content area displays a message: "Request in progress. A certificate request with a status of Pending validation has been created. Further action is needed to complete the validation and approval of the certificate." Below this, the "Validation" section lists the domains being validated:

- ▶ my-serverless-app.com
- ▶ www.my-serverless-app.com

If you or an authorized representative did not receive the email we sent, or if you want to learn more, click the help icon (?) above.

A red arrow points to the "Continue" button at the bottom right of the validation section.

Now since we are setting up a certificate for two domains (the non-www and www versions), we'll be receiving two emails with a link to verify that you own the domains. Make sure to hit **I Approve** on both the emails.

Amazon Web Services (AWS) has received a request to issue an SSL certificate for www.demo.serverless-stack.com. You are listed as one of the authorized representatives for this domain name. Your authorization is required prior to issuing this certificate.

Verify that the domain name, AWS account ID, and certificate identifier below correspond to a request from you or a person authorized to request certificates for this domain name.

Domain name	www.my-serverless-app.com
AWS account number	2327-7185-6781
AWS Region	us-east-1
Certificate identifier	93691abe-f8be-4020-905e-af94bc00913

Review the information presented above and click **I Approve** only if you recognize the request and the account requesting it. By clicking **I Approve**, you authorize Amazon to request a certificate for the above domain name.

If you choose not to approve this request, close this page.

If you have concerns about the validity of this request, forward the email you received with a brief explanation of your concern to: validation-questions@amazon.com

Next, we'll associate this certificate with our CloudFront Distributions.

Update CloudFront Distributions with Certificate

Open up our first CloudFront Distribution from our list of distributions and hit the **Edit** button.

The screenshot shows the AWS CloudFront Distributions page. On the left sidebar, under 'Distributions', there are links for 'What's New', 'Reports & Analytics', 'Cache Statistics', 'Monitoring and Alarms', 'Popular Objects', 'Top Referrers', 'Usage', 'Viewers', 'Private Content', 'How-to Guide', and 'Origin Access Identity'. The main content area shows a distribution with the ID E1KTCKT9SOAHBW. The 'General' tab is selected, and an arrow points to the 'Edit' button. The distribution details include:

	Value
Distribution ID	E1KTCKT9SOAHBW
ARN	arn:aws:cloudfront::232771856781:distribution/E1KTCKT9SOAHBW
Log Prefix	-
Delivery Method	Web
Cookie Logging	Off
Distribution Status	Deployed
Comment	-
Price Class	Use All Edge Locations (Best Performance)
AWS WAF Web ACL	-
State	Enabled
Alternate Domain Names (CNAMEs)	demo.serverless-stack.com
SSL Certificate	Default CloudFront Certificate (*.cloudfront.net)
Domain Name	d1r8102xi6mdx3.cloudfront.net
Custom SSL Client Support	-
Supported HTTP Versions	HTTP/2, HTTP/1.1, HTTP/1.0
IPv6	Enabled
Default Root Object	-
Last Modified	2017-02-17 17:41 UTC-5
Log Bucket	-

Now switch the **SSL Certificate** to **Custom SSL Certificate** and select the certificate we just created from the drop down. And scroll down to the bottom and hit **Yes, Edit**.

SSL Certificate Default CloudFront Certificate (*.cloudfront.net)
Choose this option if you want your users to use HTTPS or HTTP to access your content with the CloudFront domain name (such as <https://d111111abcdef8.cloudfront.net/logo.jpg>).
Important: If you choose this option, CloudFront requires that browsers or devices support TLSv1 or later to access your content.

Custom SSL Certificate (example.com):
Choose this option if you want your users to access your content by using an alternate domain name, such as <https://www.example.com/logo.jpg>. You can use a certificate stored in AWS Certificate Manager (ACM) in the US East (N. Virginia) Region, or you can use a certificate stored in IAM.

demo.serverless-stack.com (93691abe-f8be...)

Request or Import a Certificate with ACM

Learn more about using custom SSL/TLS certificates with CloudFront.
Learn more about using ACM.

Custom SSL Client Support Only Clients that Support Server Name Indication (SNI)
CloudFront serves your content over HTTPS only to clients that support SNI. Older browsers and other clients that do not support SNI can not access your content over HTTPS.
[Learn More](#)

All Clients (\$600/month prorated charge applies. [Learn about pricing](#))
CloudFront allocates dedicated IP addresses at each CloudFront edge location to serve your content over HTTPS. Any client can access your content.
[Learn More](#)

Supported HTTP Versions HTTP/2, HTTP/1.1, HTTP/1.0
 HTTP/1.1, HTTP/1.0

Next, head over to the **Behaviors** tab from the top.

Distributions

CloudFront Distributions > E1KTCKT9SOAHBW

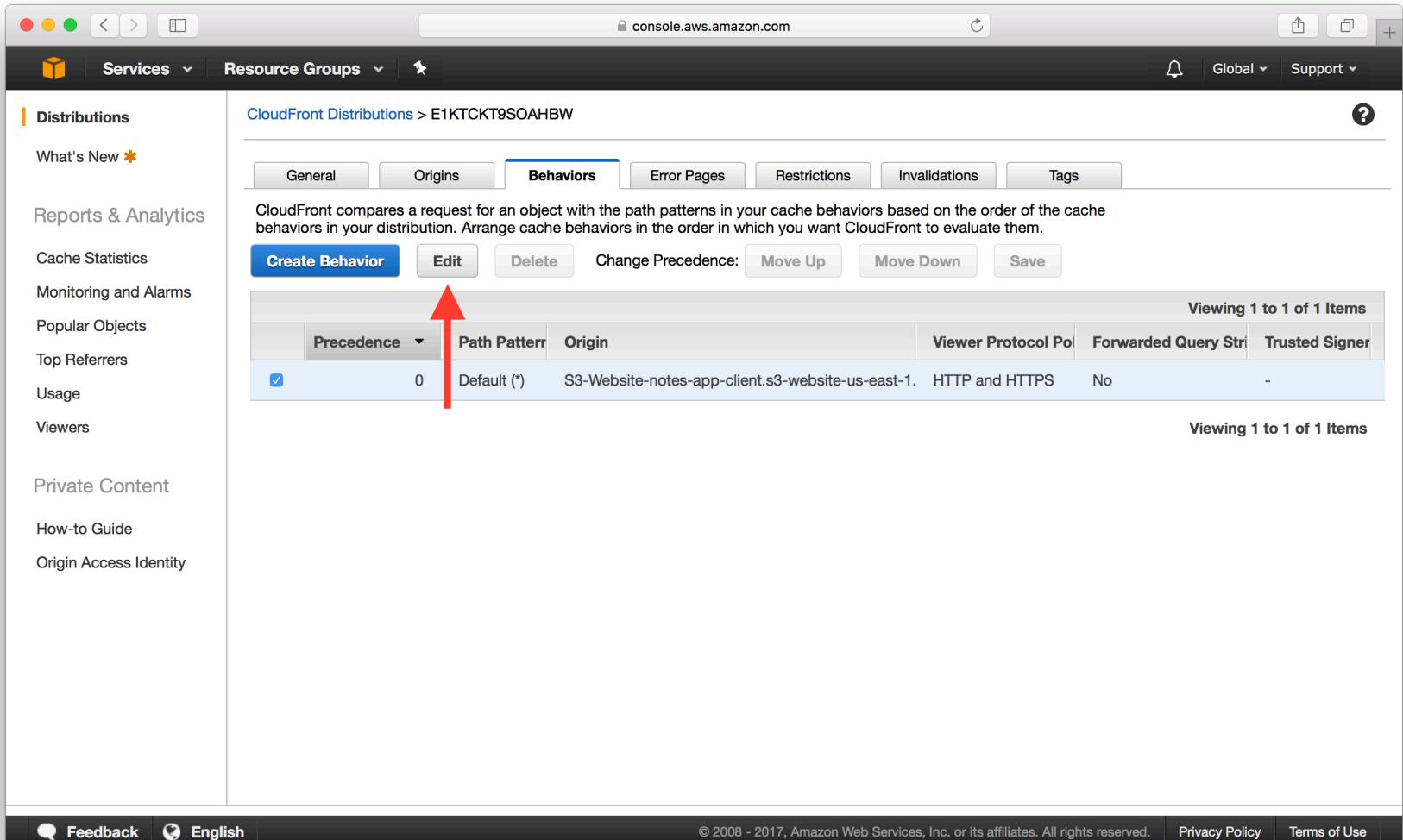
General Origins Behaviors Error Pages Restrictions Invalidations Tags

CloudFront compares a request for an object with the path patterns in your cache behaviors based on the order of the cache behaviors in your distribution. Arrange cache behaviors in the order in which you want CloudFront to evaluate them.

Create Behavior Edit Delete Change Precedence: Move Up Move Down Save

Precedence	Path Pattern	Origin	Viewer Protocol Po	Forwarded Query Str	Trusted Signer
0	Default (*)	S3-Website-notes-app-client.s3-website-us-east-1.	HTTP and HTTPS	No	-

And select the only one we have and hit **Edit**.



The screenshot shows the AWS CloudFront Distributions page. On the left sidebar, under the 'Distributions' section, there is a single item: 'CloudFront Distributions > E1KTCKT9SOAHBW'. The 'Behaviors' tab is selected. In the main content area, there is a table with one row. The table has columns for 'Precedence', 'Path Pattern', 'Origin', 'Viewer Protocol Po', 'Forwarded Query Str', and 'Trusted Signer'. The first row shows a Precedence of 0, Path Pattern 'Default (*)', Origin 'S3-Website-notes-app-client.s3-website-us-east-1.', Viewer Protocol Policy 'HTTP and HTTPS', Forwarded Query String 'No', and Trusted Signer '-'. There are buttons for 'Create Behavior', 'Edit', 'Delete', 'Change Precedence' (with 'Move Up' and 'Move Down' options), and 'Save'. The status bar at the bottom indicates 'Viewing 1 to 1 of 1 Items'.

Precedence	Path Pattern	Origin	Viewer Protocol Po	Forwarded Query Str	Trusted Signer
0	Default (*)	S3-Website-notes-app-client.s3-website-us-east-1.	HTTP and HTTPS	No	-

Then switch the **Viewer Protocol Policy** to **Redirect HTTP to HTTPS**. And scroll down to the bottom and hit **Yes, Edit**.

The screenshot shows the 'Edit Behavior' page for a CloudFront distribution. Under 'Default Cache Behavior Settings', the 'Viewer Protocol Policy' section is highlighted. It includes options for 'HTTP and HTTPS' (unchecked), 'Redirect HTTP to HTTPS' (checked), and 'HTTPS Only' (unchecked). A red arrow points to the 'Redirect HTTP to HTTPS' checkbox. Other settings shown include 'Allowed HTTP Methods' (GET, HEAD selected), 'Cached HTTP Methods' (GET, HEAD selected), 'Forward Headers' (None selected), 'Object Caching' (Use Origin Cache Headers selected), and TTL settings (Minimum TTL 0, Maximum TTL 31536000, Default TTL 86400).

Now let's do the same for our other CloudFront Distribution.

The screenshot shows the 'SSL Certificate' configuration page for a CloudFront distribution. The 'Custom SSL Certificate (example.com)' option is selected. It provides instructions for using a custom domain name and links to ACM for certificate management. Below this, 'Custom SSL Client Support' is set to 'Only Clients that Support Server Name Indication (SNI)'. It explains SNI support and provides a link to learn more. An alternative option 'All Clients (\$600/month prorated charge applies)' is also listed. At the bottom, 'Supported HTTP Versions' is set to 'HTTP/2, HTTP/1.1, HTTP/1.0'.

But leave the **Viewer Protocol Policy** as **HTTP and HTTPS**. This is because we want our users to go straight to the HTTPS version of our non-www domain. As opposed to redirecting to the HTTPS version of our www domain before redirecting again.

The screenshot shows the 'Edit Behavior' page for a CloudFront distribution. Under 'Default Cache Behavior Settings', the 'Viewer Protocol Policy' is set to 'HTTP and HTTPS', which is highlighted with a red arrow. Other options shown are 'Redirect HTTP to HTTPS' and 'HTTPS Only'. The 'Allowed HTTP Methods' are set to 'GET, HEAD'. The 'Cached HTTP Methods' are 'GET, HEAD (Cached by default)'. The 'Forward Headers' are 'None (Improves Caching)'. The 'Object Caching' section has 'Use Origin Cache Headers' selected. Time-to-Live (TTL) settings are listed: Minimum TTL (0), Maximum TTL (31536000), and Default TTL (86400). The bottom of the screen shows standard AWS navigation links like Feedback, English, and links to Privacy Policy and Terms of Use.

Update S3 Redirect Bucket

The S3 Redirect Bucket that we created in the last chapter is redirecting to the HTTP version of our non-www domain. We should switch this to the HTTPS version to prevent the extra redirect.

Open up the S3 Redirect Bucket we created in the last chapter. Head over to the **Properties** tab and select **Static website hosting**.

The screenshot shows the AWS S3 console with the bucket 'www-notes-app-client'. The 'Properties' tab is selected. In the 'Static website hosting' section, there is a checkbox labeled 'Redirect all requests' which is checked. A red arrow points upwards from the 'Static website hosting' section towards the 'Versioning' section, and another red arrow points downwards from the 'Static website hosting' section towards the 'Advanced settings' section.

Versioning

Keep multiple versions of an object in the same bucket.

Learn more

Disabled

Logging

Set up access log records that provide details about access requests.

Learn more

Disabled

Static website hosting

Host a static website, which does not require server-side technologies.

Learn more

Redirect all requests

Advanced settings

Tags

Use tags to track your cost against projects or other criteria.

Learn more

Cross-region replication

Automate copying objects across different AWS Regions.

Learn more

Transfer acceleration

Enable fast, easy and secure transfers of files to and from your bucket.

Learn more

Change the **Protocol** to **https** and hit **Save**.

The screenshot shows the 'Static website hosting' configuration dialog box. It displays the endpoint and two options: 'Use this bucket to host a website' and 'Redirect requests'. The 'Redirect requests' option is selected. The 'Target bucket or domain' field contains 'my-serverless-app.com'. The 'Protocol' field has 'https' entered. A red arrow points to the 'Protocol' input field. At the bottom are 'Cancel' and 'Save' buttons.

Static website hosting

Endpoint : <http://www-notes-app-client.s3-website-us-east-1.amazonaws.com>

Use this bucket to host a website [Learn more](#)

Redirect requests [Learn more](#)

Target bucket or domain

my-serverless-app.com

Protocol

https

Disable website hosting

Cancel Save

Advanced settings

Tags

Use tags to track your cost against projects or other criteria.

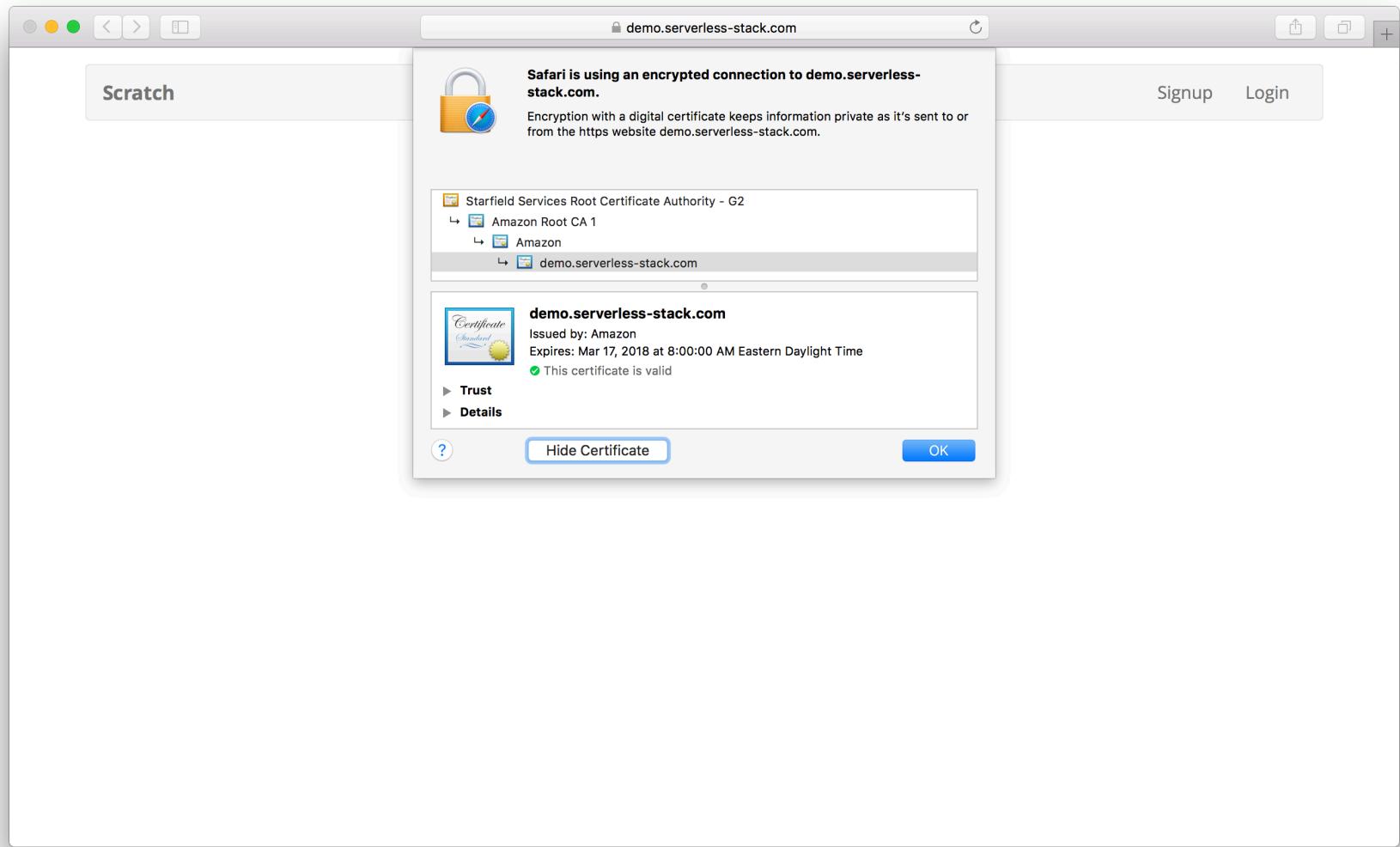
Cross-region replication

Automate copying objects across different AWS Regions.

Transfer acceleration

Enable fast, easy and secure transfers of files to and from your bucket.

And that's it. Our app should be served out on our domain through HTTPS.



Next up, let's look at the process of deploying updates to our app.

For help and discussion

💬 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/67>)

Deploy Updates

Now let's look at how we make changes and update our app. The process is very similar to how we deployed our code to S3 but with a few changes. Here is what it looks like.

1. Build our app with the changes
2. Deploy to the main S3 Bucket
3. Invalidate the cache in both our CloudFront Distributions

We need to do the last step since CloudFront caches our objects in its edge locations. So to make sure that our users see the latest version, we need to tell CloudFront to invalidate its cache in the edge locations.

Let's start by making a couple of changes to our app and go through the process of deploying them.

For help and discussion

 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/68>)

Update the App

Let's make a couple of quick changes to test the process of deploying updates to our app.

We are going to add a Login and Signup button to our lander to give users a clear call to action.

◆ CHANGE To do this update our `renderLander` method in `src/containers/Home.js`.

```
renderLander() {  
  return (  
    <div className="lander">  
      <h1>Scratch</h1>  
      <p>A simple note taking app</p>  
      <div>  
        <Link to="/login" className="btn btn-info btn-lg">  
          Login  
        </Link>  
        <Link to="/signup" className="btn btn-success btn-lg">  
          Signup  
        </Link>  
      </div>  
    </div>  
  );  
}
```

◆ CHANGE And import the `Link` component from React-Router in the header.

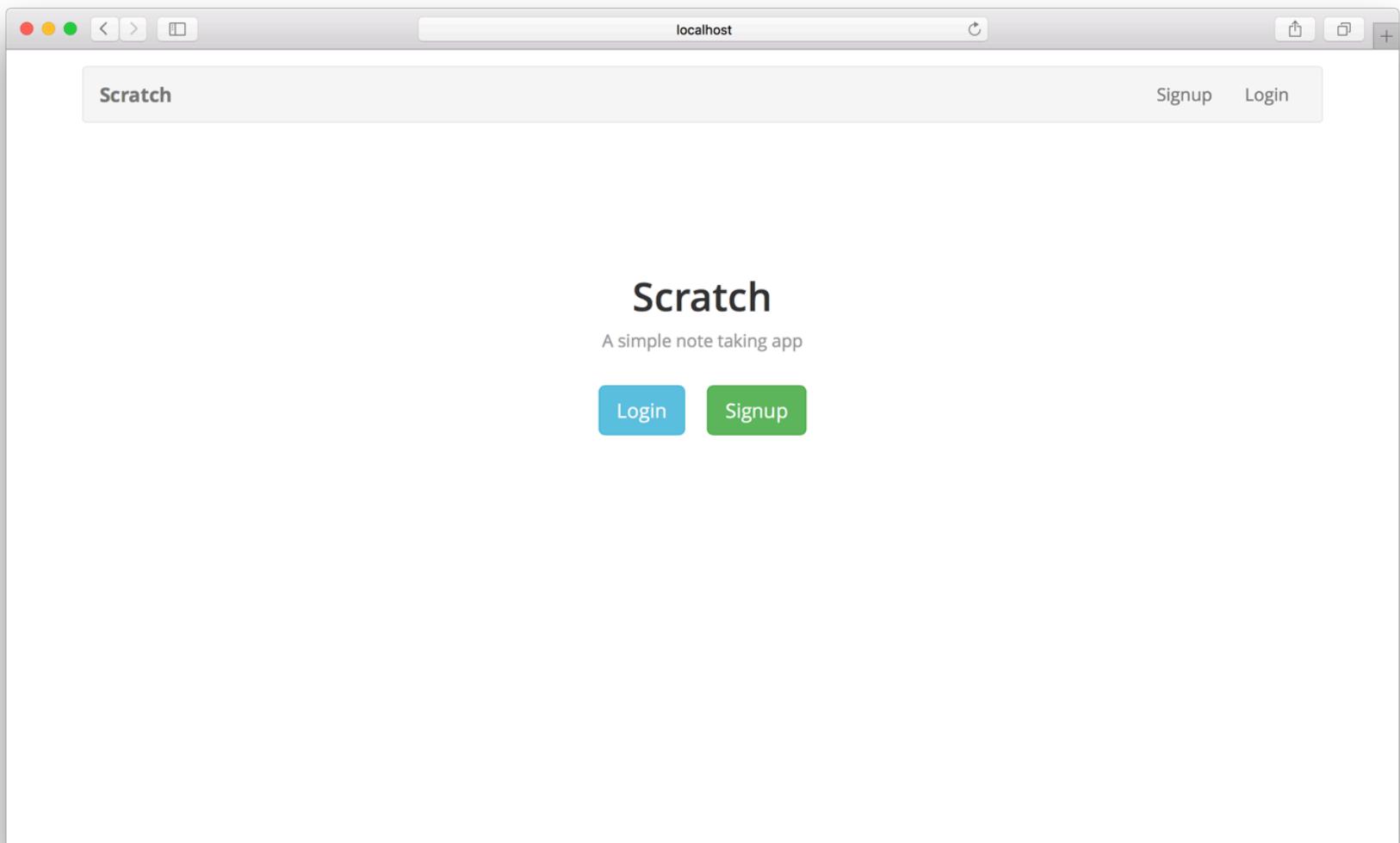
```
import { Link } from "react-router-dom";
```

◆ CHANGE Also, add a couple of styles to `src/containers/Home.css`.

```
.Home .lander div {  
  padding-top: 20px;  
}
```

```
.Home .lander div a:first-child {  
  margin-right: 20px;  
}
```

And our lander should look something like this.



Next, let's deploy these updates.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/69>)

For reference, here is the code so far

🌐 Frontend Source :update-the-app

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/update-the-app>)

Deploy Again

Now that we've made some changes to our app, let's deploy the updates. This is the process we are going to repeat every time we need to deploy any updates.

Build Our App

First let's prepare our app for production by building it. Run the following in your working directory.

```
$ npm run build
```

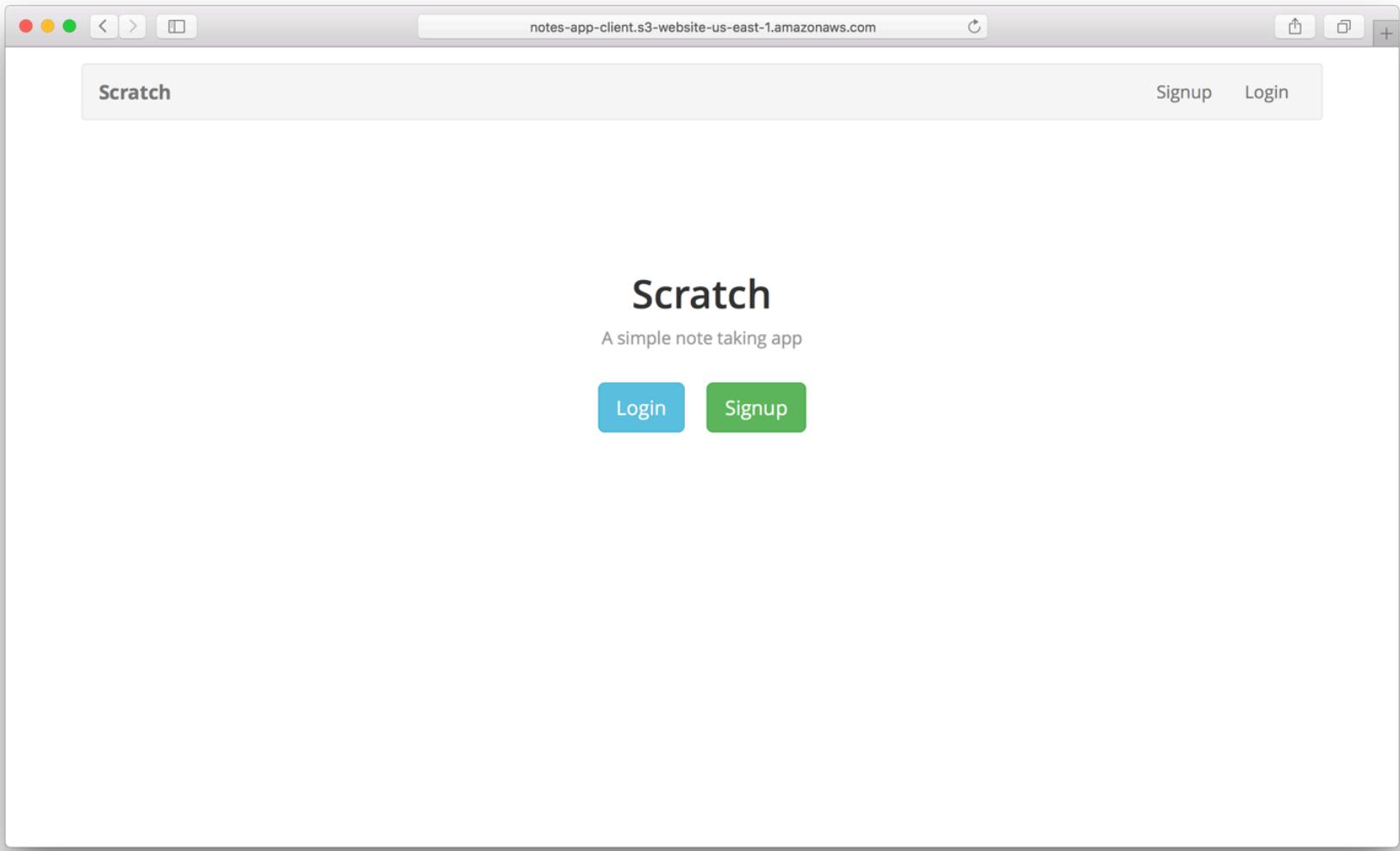
Now that our app is built and ready in the `build/` directory, let's deploy to S3.

Upload to S3

Run the following from our working directory to upload our app to our main S3 Bucket. Make sure to replace `YOUR_S3_DEPLOY_BUCKET_NAME` with the S3 Bucket we created in the Create an S3 bucket (/chapters/create-an-s3-bucket.html) chapter.

```
$ aws s3 sync build/ s3://YOUR_S3_DEPLOY_BUCKET_NAME
```

Our changes should be live on S3.



Now to ensure that CloudFront is serving out the updated version of our app, let's invalidate the CloudFront cache.

Invalidate the CloudFront Cache

CloudFront allows you to invalidate objects in the distribution by passing in the path of the object. But it also allows you to use a wildcard (`//*`) to invalidate the entire distribution in a single command. This is recommended when we are deploying a new version of our app.

To do this we'll need the **Distribution ID** of **both** of our CloudFront Distributions. You can get it by clicking on the distribution from the list of CloudFront Distributions.

Distributions

CloudFront Distributions > E1KTCKT9SOAHBW

General Origins Behaviors Error Pages Restrictions Invalidations Tags

What's New *

Reports & Analytics

Cache Statistics Monitoring and Alarms Popular Objects Top Referrers Usage Viewers

Private Content

How-to Guide Origin Access Identity

Distribution ID E1KTCKT9SOAHBW

ARN arn:aws:cloudfront::232771856781:distribution/E1KTCKT9SOAHBW

Log Prefix -

Delivery Method Web

Cookie Logging Off

Distribution Status Deployed

Comment -

Price Class Use All Edge Locations (Best Performance)

AWS WAF Web ACL -

State Enabled

Alternate Domain Names (CNAMEs)

SSL Certificate	demo.serverless-stack.com (93691abe-f8be-4020-905e-af94bc00913)
Domain Name	d1r8102xi6mdx3.cloudfront.net
Custom SSL Client Support	Only Clients that Support Server Name Indication (SNI)
Supported HTTP Versions	HTTP/2, HTTP/1.1, HTTP/1.0
IPv6	Enabled
Default Root Object	-
Last Modified	2017-02-17 19:12 UTC-5
Log Bucket	-

Feedback English © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Now we can use the AWS CLI to invalidate the cache of the two distributions. As of writing this, the CloudFront portion of the CLI is in preview and needs to be enabled by running the following. This only needs to be run once and not every time we deploy.

```
$ aws configure set preview.cloudfront true
```

And to invalidate the cache we run the following. Make sure to replace

YOUR_CF_DISTRIBUTION_ID and YOUR_WWW_CF_DISTRIBUTION_ID with the ones from above.

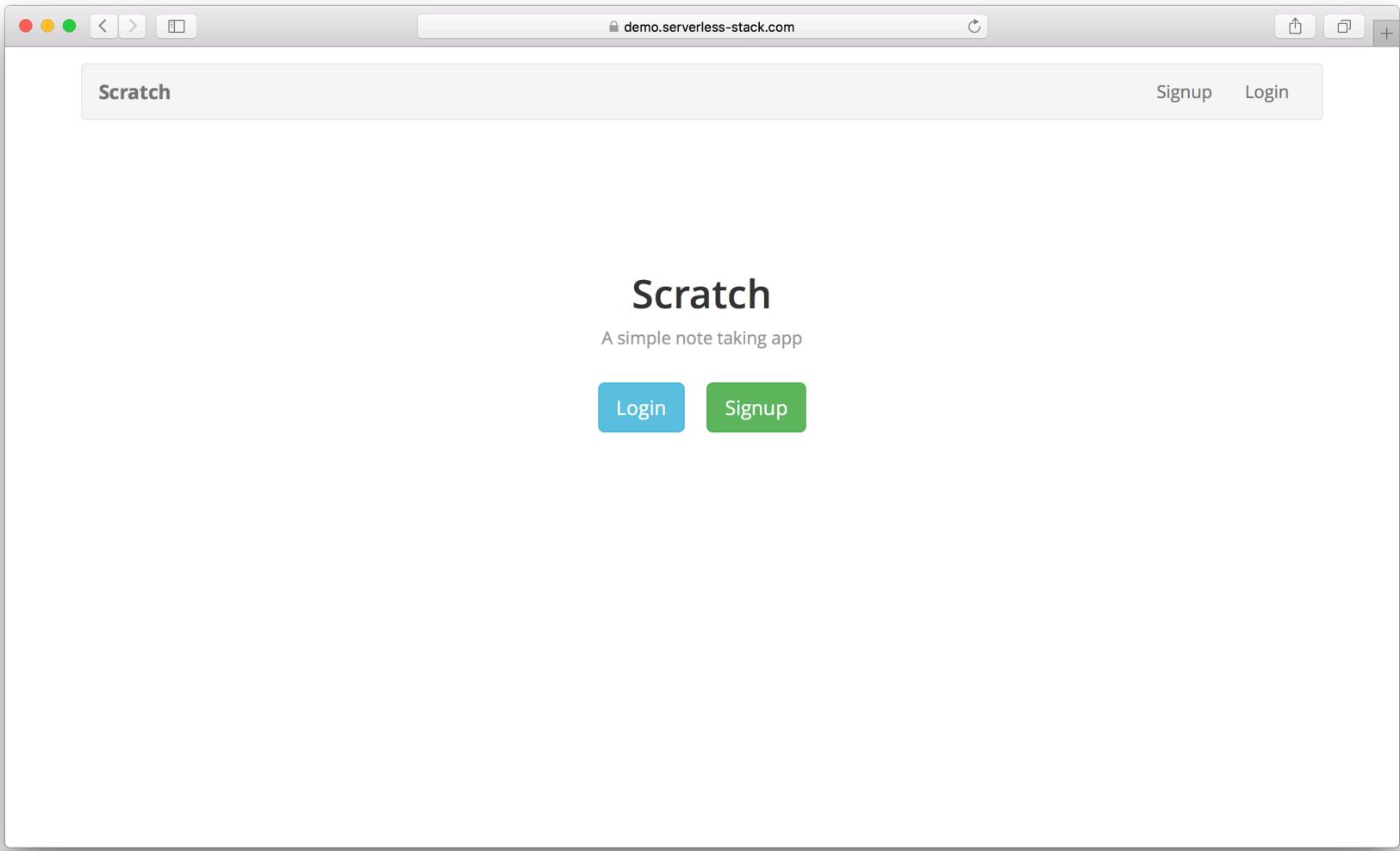
```
$ aws cloudfront create-invalidation --distribution-id  
YOUR_CF_DISTRIBUTION_ID --paths "/*"  
$ aws cloudfront create-invalidation --distribution-id  
YOUR_WWW_CF_DISTRIBUTION_ID --paths "/*"
```

This invalidates our distribution for both the www and non-www versions of our domain. If you click on the **Invalidations** tab, you should see your invalidation request being processed.

A screenshot of the AWS CloudFront Distributions page for distribution E3MQXGQ47VCJB0. The 'Invalidations' tab is selected. A red arrow points to the status column of the first item in the table, which shows 'In Progress'. The table has columns for 'Invalidation ID' (IA60DS11ATXA7), 'Status' (In Progress), and 'Date' (2017-02-18 13:58 UTC-5). The status cell contains a circular icon with a refresh symbol.

	Invalidation ID	Status	Date
<input type="checkbox"/>	IA60DS11ATXA7	In Progress	2017-02-18 13:58 UTC-5

It can take a few minutes to complete. But once it is done, the updated version of our app should be live.



And that's it! We now have a set of commands we can run to deploy our updates. Let's quickly put them together so we can do it with one command.

Add a Deploy Command

NPM allows us to add a `deploy` command in our `package.json`.

◆ CHANGE Add the following in the `scripts` block above `eject` in the `package.json`.

```
"predeploy": "npm run build",
"deploy": "aws s3 sync build/ s3://YOUR_S3_DEPLOY_BUCKET_NAME",
"postdeploy": "aws cloudfront create-invalidation --distribution-id
YOUR_CF_DISTRIBUTION_ID --paths '/*' && aws cloudfront create-
invalidation --distribution-id YOUR_WWW_CF_DISTRIBUTION_ID --paths
'/*'",
```

Make sure to replace `YOUR_S3_DEPLOY_BUCKET_NAME`, `YOUR_CF_DISTRIBUTION_ID`, and `YOUR_WWW_CF_DISTRIBUTION_ID` with the ones from above.

For Windows users, if `postdeploy` returns an error like.

An error occurred (InvalidArgumentException) when calling the CreateInvalidation operation: Your request contains one or more invalid invalidation paths.

Make sure that there is no quote in the `/*`.

```
"postdeploy": "aws cloudfront create-validation --distribution-id YOUR_CF_DISTRIBUTION_ID --paths /* && aws cloudfront create-validation --distribution-id YOUR_WWW_CF_DISTRIBUTION_ID --paths /*",
```

Now simply run the following command from your project root when you want to deploy your updates. It'll build your app, upload it to S3, and invalidate the CloudFront cache.

```
$ npm run deploy
```

Our app is now complete. And we have an easy way to update it!

For help and discussion

 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/70>)

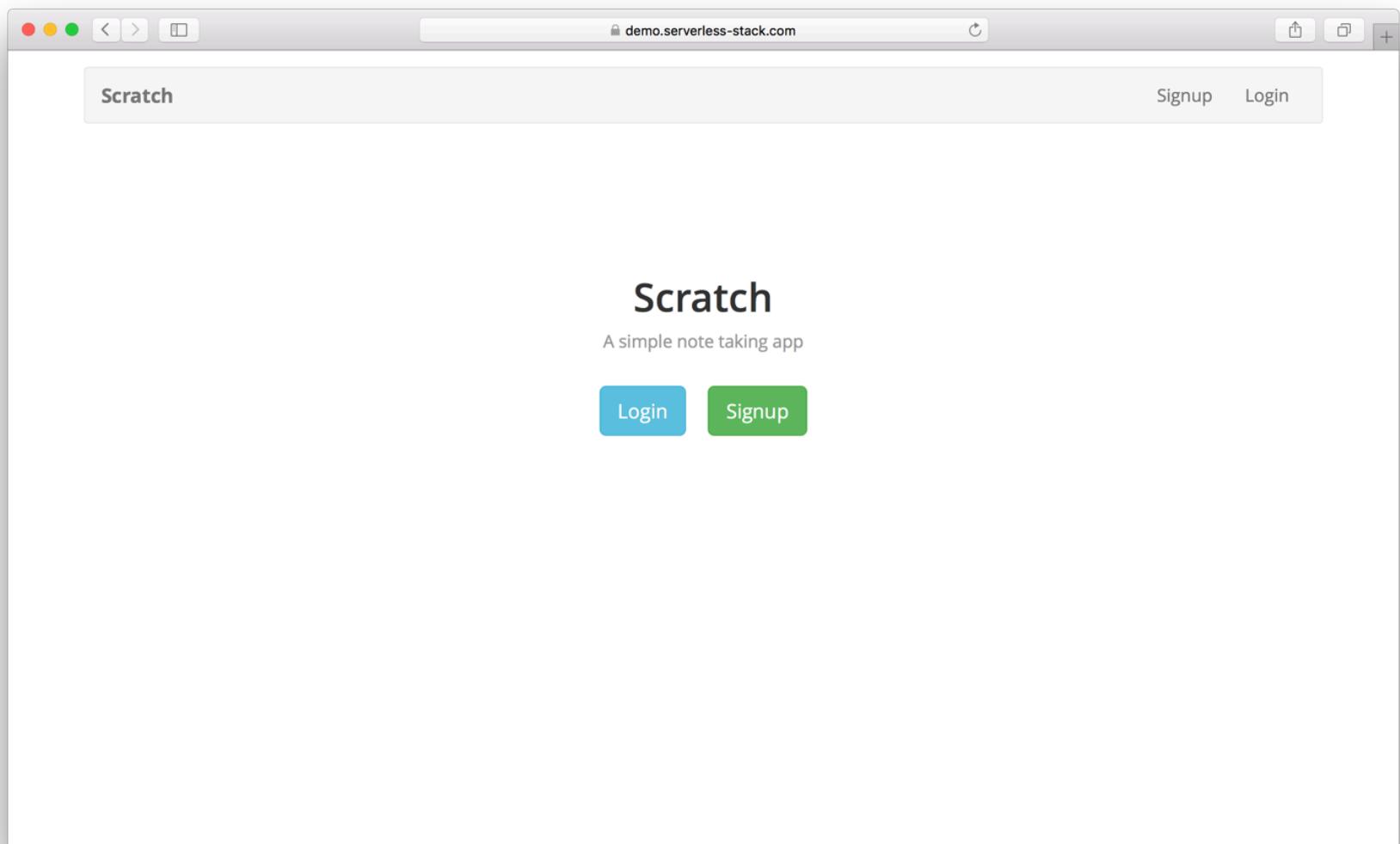
For reference, here is the complete code

 [Frontend Source](#)

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client>)

Wrapping Up

So we've covered how to build and deploy our backend serverless API and our frontend serverless app. And not only does it work well on the desktop.



It's mobile optimized as well!



The process of creating our app might have been quite long, but this guide is meant to be more of a learning process. You should be able to take what you've learned from here and adapt it to better suit your workflow. We are also going to be adding tools to help you work on your projects. You can find them in the tools section ([/#tools](#)) in our homepage.

We'd love to hear from you about your experience following this guide. Please send us any comments or feedback you might have, via email (<mailto:contact@anoma.ly>). We'd love to feature your comments here. Also, if you'd like us to cover any of the chapters or concepts in a

bit more detail, feel free to let us know (<mailto:contact@anomaly.ly>).

Thank you and we hope you found this guide helpful!

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/71>)

Giving Back

If you've found this guide helpful please consider helping us out by doing the following.

- **Helping others in the comments**

If you've found yourself using the GitHub issues (<https://github.com/AnomalyInnovations/serverless-stack-com/issues?q=is%3Aissue+is%3Aopen+label%3ADiscussion+sort%3Aupdated-desc>) to get help, please consider helping anybody else with issues that you might have run into.

- **Fixing typos and errors**

The content on this site is kept up to date thanks in large part to our community and our readers. Submit a Pull Request (<https://github.com/AnomalyInnovations/serverless-stack-com/compare>) to fix any typos or errors you might find.

- **Give us a Star on GitHub** (<https://github.com/AnomalyInnovations/serverless-stack-com>)

We rely on our GitHub repo for everything from hosting this site to code samples and comments. Starring our repo (<https://github.com/AnomalyInnovations/serverless-stack-com>) helps us get the word out.

- **Sharing this guide**

Share this guide via Twitter (<https://twitter.com/intent/tweet?text=Serverless Stack&url=https://serverless-stack.com>) or Facebook ([https://www.facebook.com/sharer/sharer.php?u=https://serverless-stack.com&p\[title\]=Serverless%20Stack](https://www.facebook.com/sharer/sharer.php?u=https://serverless-stack.com&p[title]=Serverless%20Stack)) with others that might find this helpful.

Also, if you have any other ideas on how to contribute; feel free to let us know via email (<mailto:contact@anoma.ly>).

 Comments on this chapter
[\(https://github.com/AnomalyInnovations/serverless-stack-com/issues/71\)](https://github.com/AnomalyInnovations/serverless-stack-com/issues/71)

Changelog

As we continue to update Serverless Stack, we want to make sure that we give you a clear idea of all the changes that are being made. This is to ensure that you won't have to go through the entire tutorial again to get caught up on the updates. We also want to leave the older versions up in case you need a reference. This is also useful for readers who are working through the tutorial while it gets updated.

Below are the updates we've made to Serverless Stack, each with:

- Each update has a link to an **archived version of the tutorial**
- Updates to the tutorial **compared to the last version**
- Updates to the **API and Client repos**

While the hosted version of the tutorial and the code snippets are accurate, the sample project repo that is linked at the bottom of each chapter is unfortunately not. We do however maintain the past versions of the completed sample project repo. So you should be able to use those to figure things out. All this info is also available on the releases page

(<https://github.com/AnomalyInnovations/serverless-stack-com/releases>) of our GitHub repo (<https://github.com/AnomalyInnovations/serverless-stack-com>).

You can get these updates emailed to you via our newsletter (<http://eepurl.com/cEaBlf>).

Changes

v1.2.5: Using specific Bootstrap CSS version (<https://branchv125--serverless-stack.netlify.com>) (Current)

Feb 5, 2018: Using specific Bootstrap CSS version since `latest` now points to Bootstrap v4. But React-Bootstrap uses v3.

- Tutorial changes (<https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2.4...v1.2.5>)
- Client (<https://github.com/AnomalyInnovations/serverless-stack-demo>)

v1.2.4: Updating to React 16 (<https://5a4993f3a6188f5a88e0c777--serverless-stack.netlify.com/>)

Dec 31, 2017: Updated to React 16 and fixed `sigv4Client.js` IE11 issue (<https://github.com/AnomalyInnovations/serverless-stack-com/issues/114#issuecomment-349938586>).

- Tutorial changes (<https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2.3...v1.2.4>)
- Client (<https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v1.2...v1.2.4>)

v1.2.3: Updating to babel-preset-env

(<https://5a4993898198761218a1279f--serverless-stack.netlify.com/>)

Dec 30, 2017: Updated serverless backend to use babel-preset-env plugin and added a note to the Deploy to S3 chapter on reducing React app bundle size.

- Tutorial changes (<https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2.2...v1.2.3>)
- API (<https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v1.2...v1.2.3>)

v1.2.2: Adding new chapters (<https://5a499324a6188f5a88e0c76d--serverless-stack.netlify.com/>)

Dec 1, 2017: Added the following *Extra Credit* chapters.

1. Customize the Serverless IAM Policy
 2. Environments in Create React App
- Tutorial changes (<https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2.1...v1.2.2>)

v1.2.1: Adding new chapters (<https://5a4992e70b79b76fb0948300--serverless-stack.netlify.com/>)

Oct 7, 2017: Added the following *Extra Credit* chapters.

1. API Gateway and Lambda Logs
 2. Debugging Serverless API Issues
 3. Serverless environment variables
 4. Stages in Serverless Framework
 5. Configure multiple AWS profiles
- Tutorial changes (<https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.2...v1.2.1>)

v1.2: Upgrade to Serverless Webpack v3

(<https://59caac9bcf321c5b78f2c3e2--serverless-stack.netlify.com/>)

Sep 16, 2017: Upgrading serverless backend to using serverless-webpack plugin v3. The new version of the plugin changes some of the commands used to test the serverless backend. You can read more about it here (<https://github.com/AnomalyInnovations/serverless-stack-com/issues/130>).

- Tutorial changes (<https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.1...v1.2>)
- API (<https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v1.1...v1.2>)

v1.1: Improved Session Handling (<https://59caae1e6f4c50416e86701d--serverless-stack.netlify.com/>)

Aug 30, 2017: Fixing some issues with session handling in the React app. A few minor updates bundled together. You can read more about it here (<https://github.com/AnomalyInnovations/serverless-stack-com/issues/123>).

- Tutorial changes (<https://github.com/AnomalyInnovations/serverless-stack-com/compare/v1.0...v1.1>)
- Client (<https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v1.0...v1.1>)

v1.0: IAM as authorizer (<https://59caae01424ef20727c342ce--serverless-stack.netlify.com/>)

July 19, 2017: Switching to using IAM as an authorizer instead of the authenticating directly with User Pool. This was a major update to the tutorial. You can read more about it here (<https://github.com/AnomalyInnovations/serverless-stack-com/issues/108>).

- Tutorial changes (<https://github.com/AnomalyInnovations/serverless-stack-com/compare/v0.9...v1.0>)
- API (<https://github.com/AnomalyInnovations/serverless-stack-demo-api/compare/v0.9...v1.0>)
- Client (<https://github.com/AnomalyInnovations/serverless-stack-demo-client/compare/v0.9...v1.0>)

v0.9: Cognito User Pool as authorizer

(<https://59caadbd424ef20abdc342b4--serverless-stack.netlify.com/>)

- API (<https://github.com/AnomalyInnovations/serverless-stack-demo-api/releases/tag/v0.9>)
- Client (<https://github.com/AnomalyInnovations/serverless-stack-demo-client/releases/tag/v0.9>)

For help and discussion

 **Comments on this chapter**

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/124>)

Staying up to date

We made this guide open source to make sure that the content is kept up to date and accurate with the help of the community. We are also adding new chapters based on the needs of the community and the feedback we receive.

To help people stay up to date with the changes, we run the Serverless Stack newsletter (<http://eepurl.com/cEaBlf>). The newsletter is a:

- Short plain text email
- Outlines the recent updates to Serverless Stack
- Never sent out more than once a week
- One click unsubscribe
- And you get the entire guide as a 300 page PDF

We hope you subscribe to it and find it useful.

For help and discussion

 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/102>)

API Gateway and Lambda Logs

Logging is an essential part of building backends and it is no different for a serverless API. It gives us visibility into how we are processing and responding to incoming requests.

Types of Logs

There are 2 types of logs we usually take for granted in a monolithic environment.

- Server logs

Web server logs maintain a history of requests, in the order they took place. Each log entry contains the information about the request, including client IP address, request date/time, request path, HTTP code, bytes served, user agent, etc.

- Application logs

Application logs are a file of events that are logged by the web application. It usually contains errors, warnings, and informational events. It could contain everything from unexpected function failures, to key events for understanding how users behave.

In the serverless environment, we have lesser control over the underlying infrastructure, logging is the only way to acquire knowledge on how the application is performing. Amazon CloudWatch (<https://aws.amazon.com/cloudwatch/>) is a monitoring service to help you collect and track metrics for your resources. Using the analogy of server logs and application logs, you can roughly think of the API Gateway logs as your server logs and Lambda logs as your application logs.

In the chapter we are going to look to do the following:

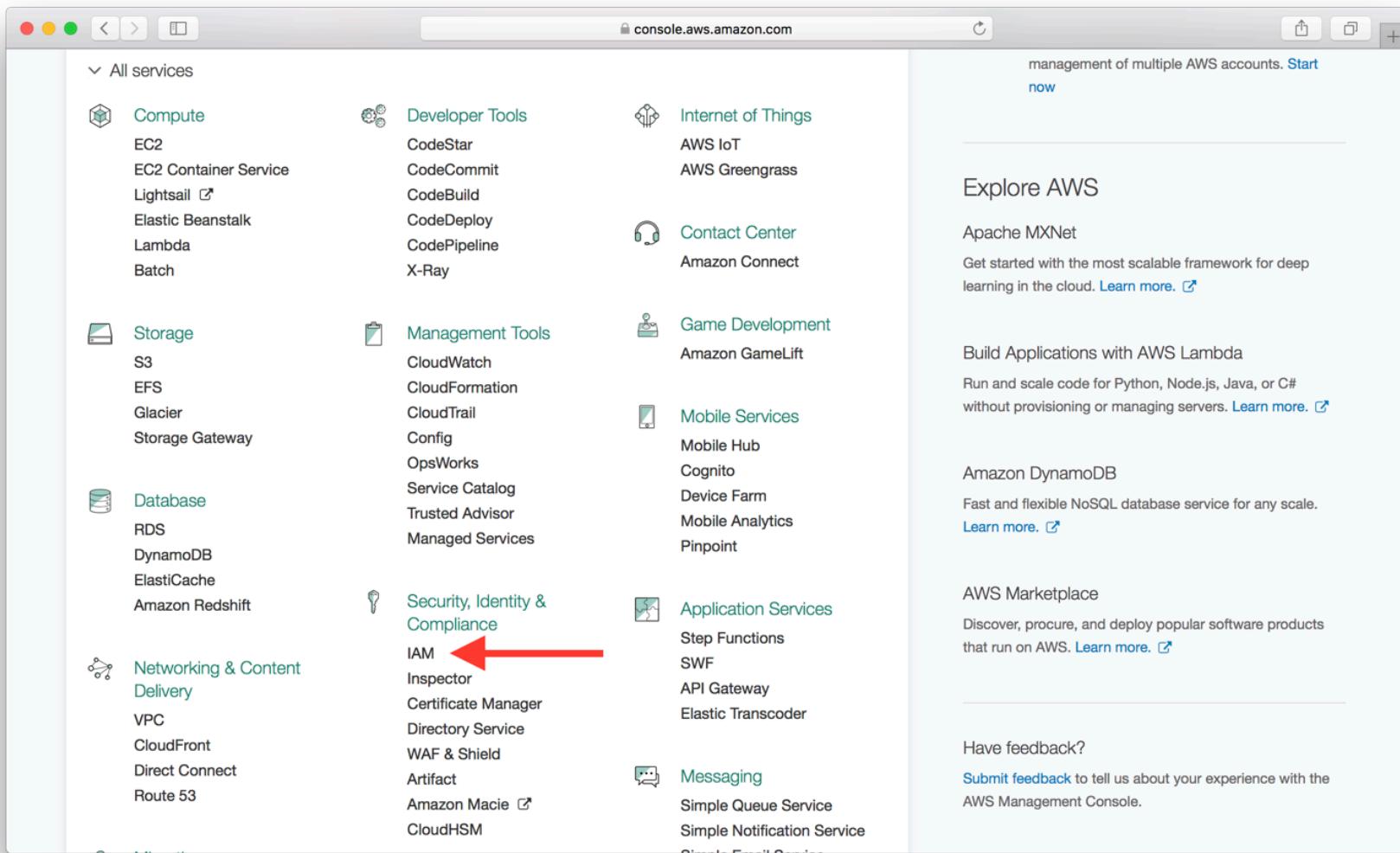
- Enable API Gateway CloudWatch Logs (#enable-api-gateway-cloudwatch-logs)
- Enable Lambda CloudWatch Logs (#enable-lambda-cloudwatch-logs)
- Viewing API Gateway CloudWatch Logs (#viewing-api-gateway-cloudwatch-logs)
- Viewing Lambda CloudWatch Logs (#viewing-lambda-cloudwatch-logs)

Let's get started.

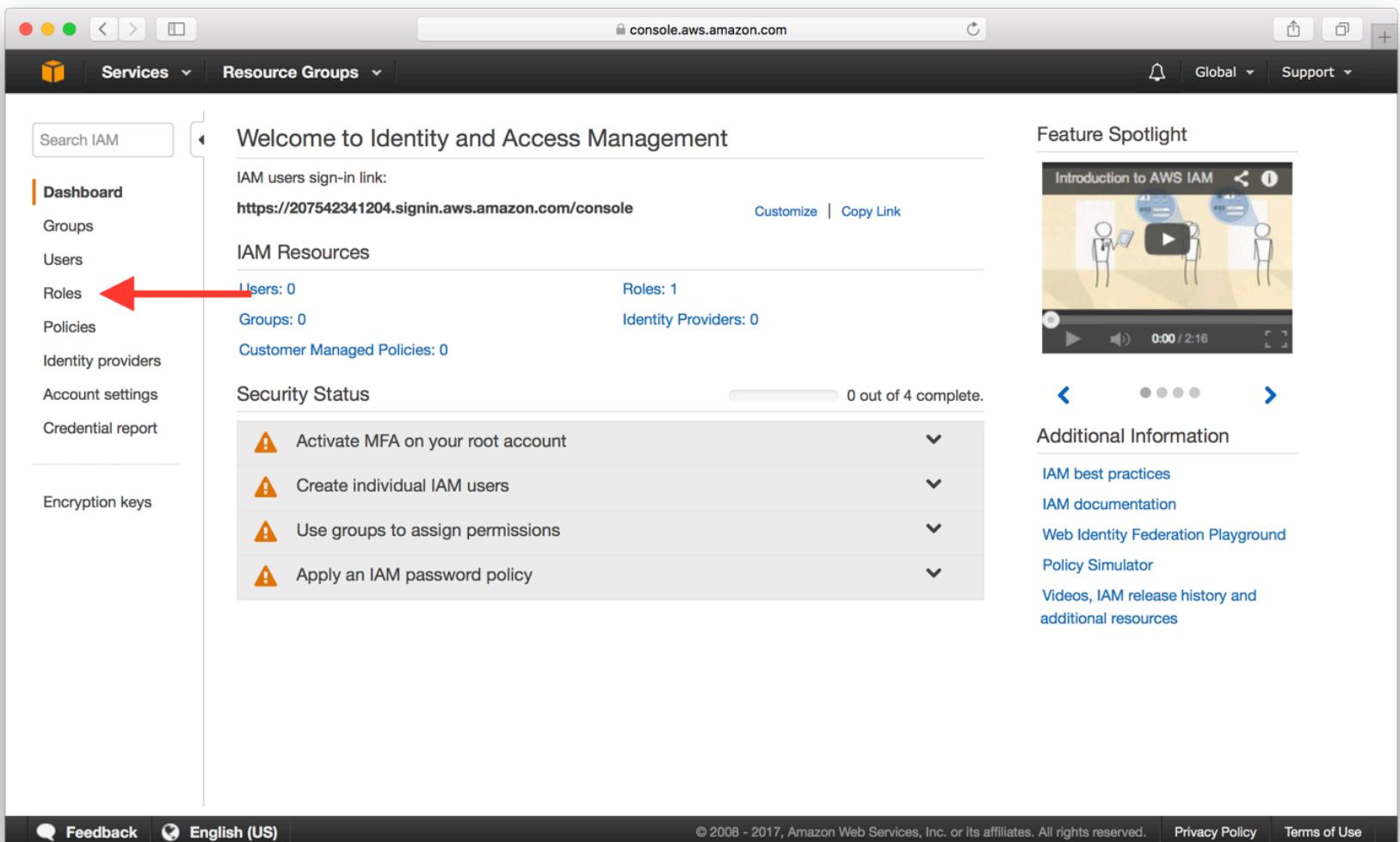
Enable API Gateway CloudWatch Logs

This is a two step process. First, we need to create an IAM role that allows API Gateway to write logs in CloudWatch. Then we need to turn on logging for our API project.

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



Select Roles on the left menu.



Welcome to Identity and Access Management

IAM users sign-in link:
<https://207542341204.signin.aws.amazon.com/console>

Customize | Copy Link

IAM Resources

Users: 0	Roles: 1
Groups: 0	Identity Providers: 0
Customer Managed Policies: 0	

Security Status

Activate MFA on your root account
⚠ Create individual IAM users
⚠ Use groups to assign permissions
⚠ Apply an IAM password policy

0 out of 4 complete.

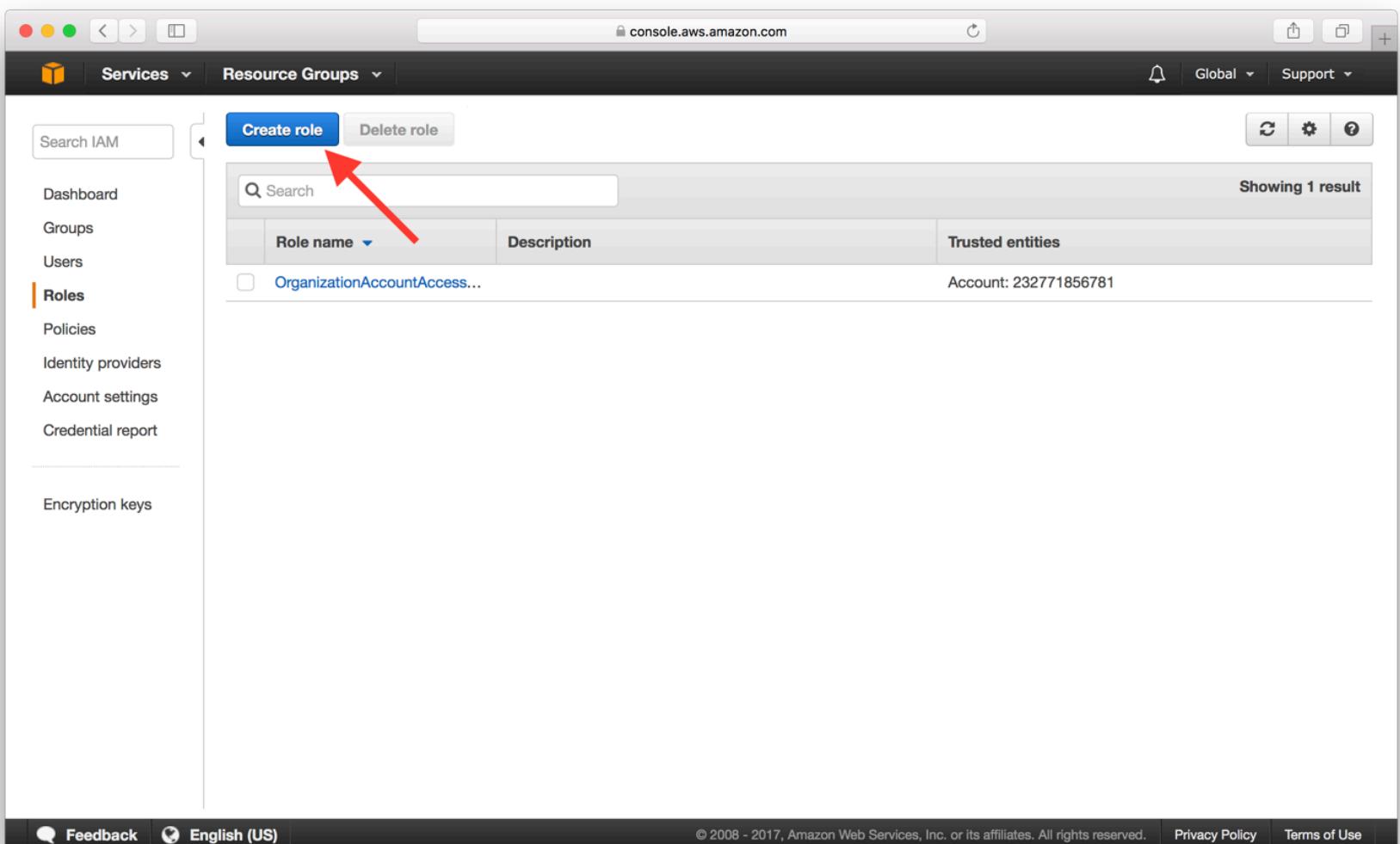
Feature Spotlight

Introduction to AWS IAM

Additional Information

- IAM best practices
- IAM documentation
- Web Identity Federation Playground
- Policy Simulator
- Videos, IAM release history and additional resources

Select **Create Role**.



Services ▾ Resource Groups ▾

Create role Delete role

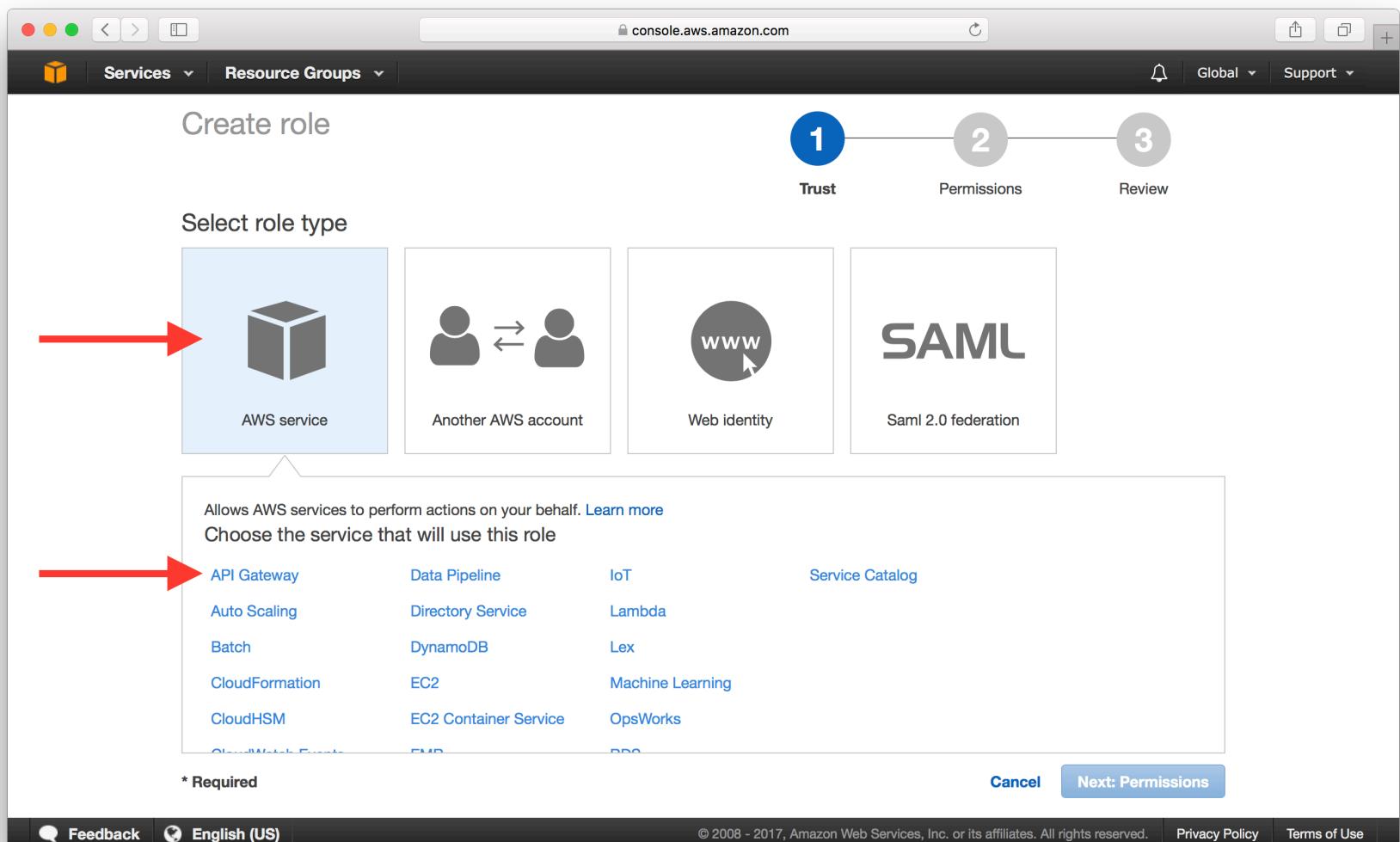
Search

Showing 1 result

Role name	Description	Trusted entities
OrganizationAccountAccess...		Account: 232771856781

Feedback English (US) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Under AWS service, select API Gateway.



Click **Next: Permissions**.

Screenshot of the AWS IAM 'Create role' wizard Step 1: Trust

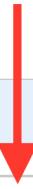
The page shows a list of AWS services that can use this role:

API Gateway	Data Pipeline	IoT	Service Catalog
Auto Scaling	Directory Service	Lambda	
Batch	DynamoDB	Lex	
CloudFormation	EC2	Machine Learning	
CloudHSM	EC2 Container Service	OpsWorks	
CloudWatch Events	EMR	RDS	
CodeBuild	Elastic Beanstalk	Redshift	
CodeDeploy	Elastic Transcoder	SMS	
Config	Glue	SNS	
DMS	Greengrass	SWF	

Select your use case:

API Gateway
Allows API Gateway to push logs to CloudWatch Logs.

* Required Cancel **Next: Permissions**



Click **Next: Review**.

Screenshot of the AWS IAM 'Create role' wizard Step 2: Permissions

Attached permissions policy:

The type of role that you selected requires the following policy.

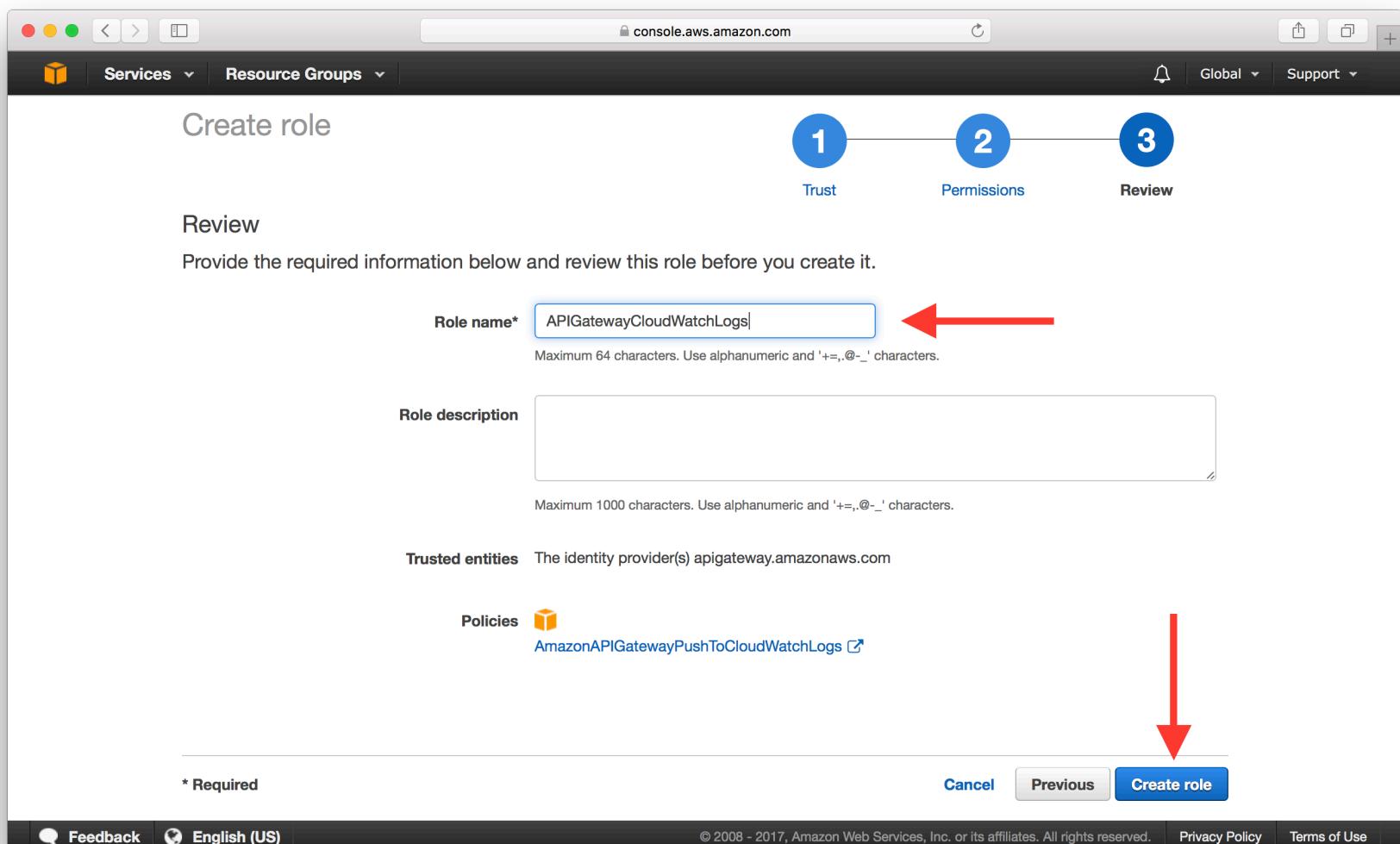
Policy name	Type	Attachments	Description
AmazonAPIGatewayPush...	AWS managed	0	Allows API Gateway to push logs to user's account.

* Required Cancel Previous **Next: Review**

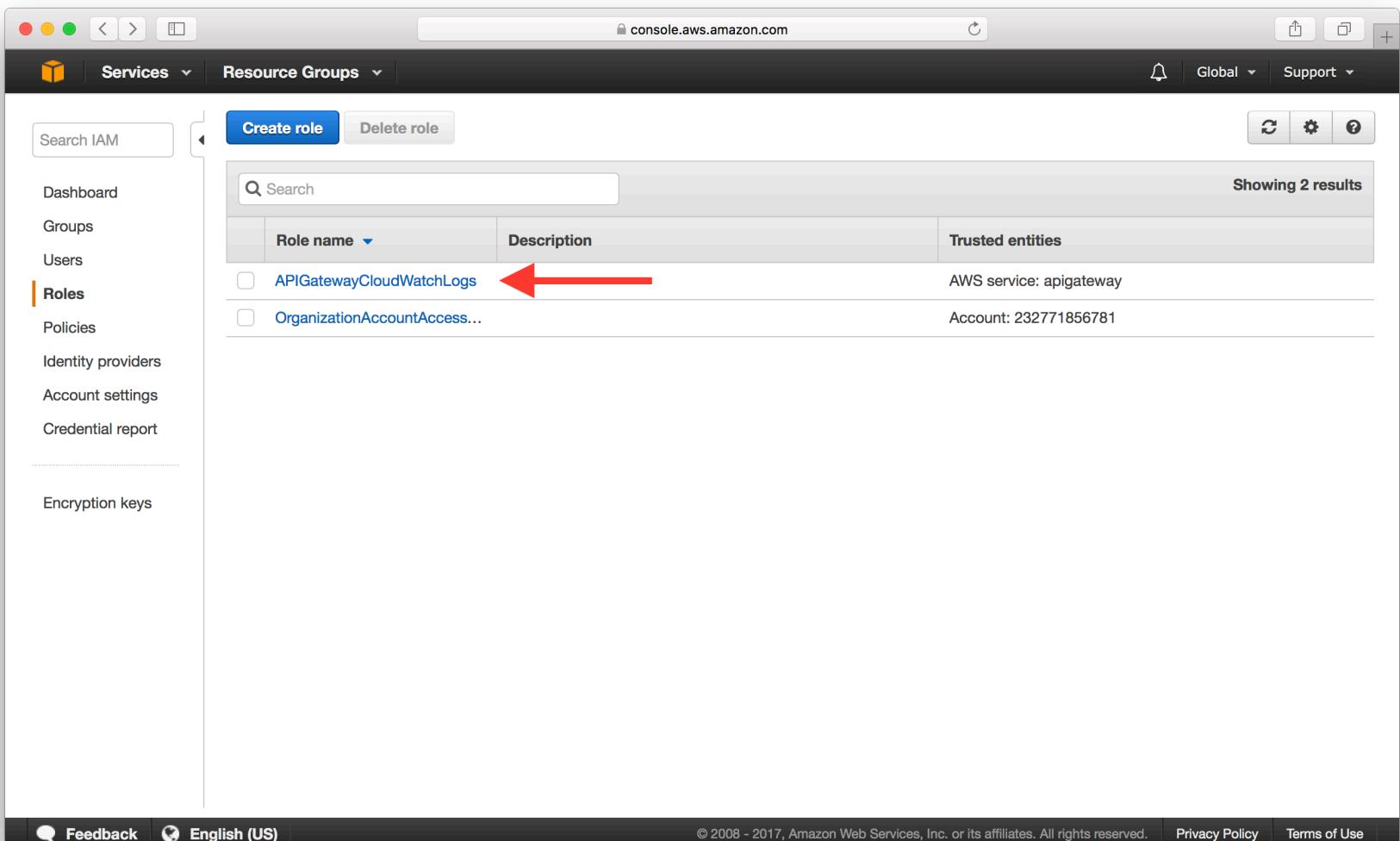


Enter a Role name and select **Create role**. In our case, we called our role

APIGatewayCloudWatchLogs .

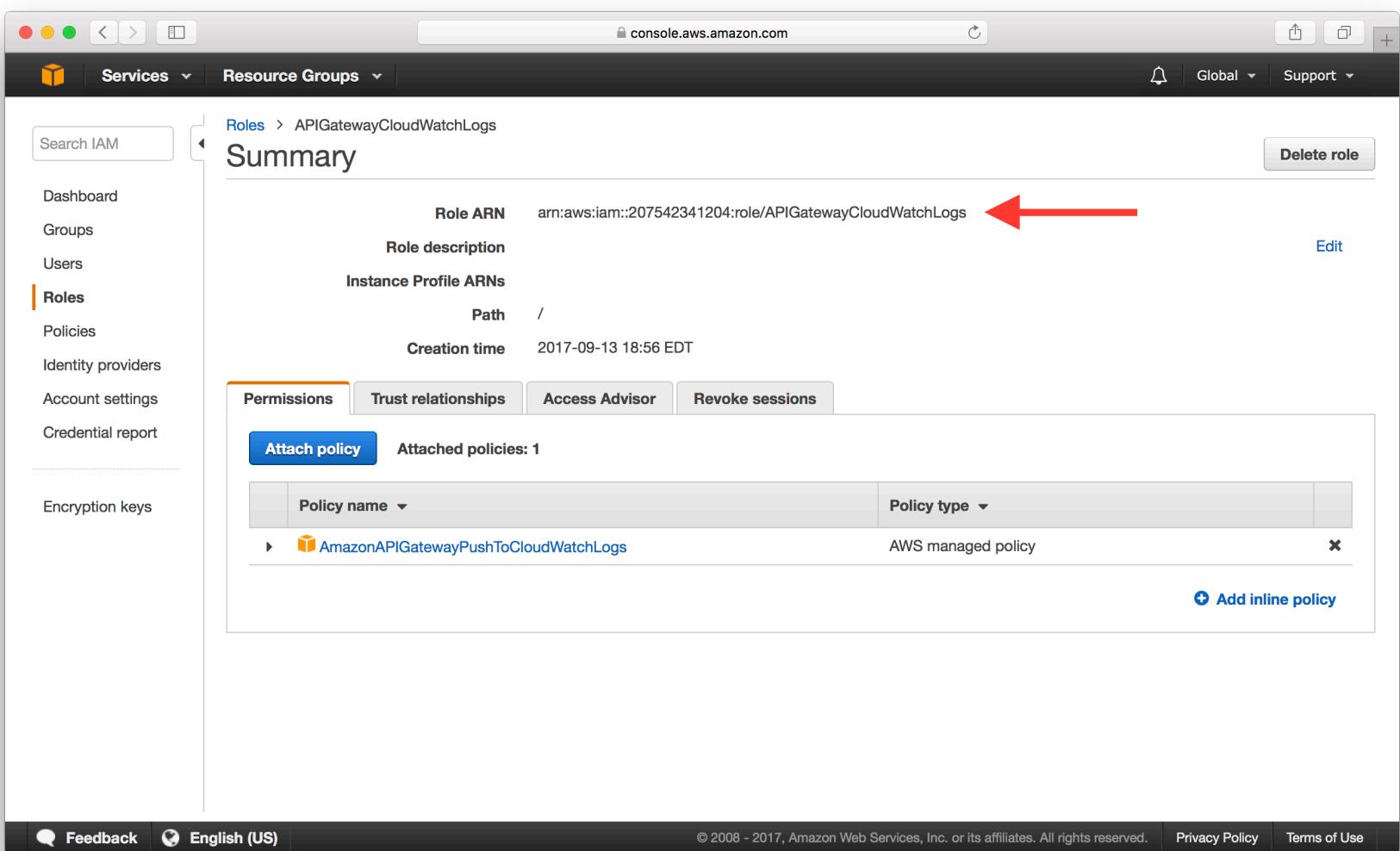


Click on the role we just created.



The screenshot shows the AWS IAM Roles page. On the left, there's a sidebar with links like Dashboard, Groups, Users, Roles (which is selected), Policies, Identity providers, Account settings, Credential report, and Encryption keys. The main area has a search bar and two buttons: 'Create role' and 'Delete role'. A table lists roles with columns for Role name, Description, and Trusted entities. Two roles are listed: 'APIGatewayCloudWatchLogs' and 'OrganizationAccountAccess...'. A red arrow points to the first role. At the bottom, there are links for Feedback, English (US), and some legal notices.

Take a note of the **Role ARN**. We will be needing this soon.



The screenshot shows the 'Summary' page for the 'APIGatewayCloudWatchLogs' role. The top navigation bar includes 'Roles > APIGatewayCloudWatchLogs'. The main content area is titled 'Summary' and contains the following details:

Role ARN	arn:aws:iam::207542341204:role/APIGatewayCloudWatchLogs
Role description	(empty)
Instance Profile ARNs	(empty)
Path	/
Creation time	2017-09-13 18:56 EDT

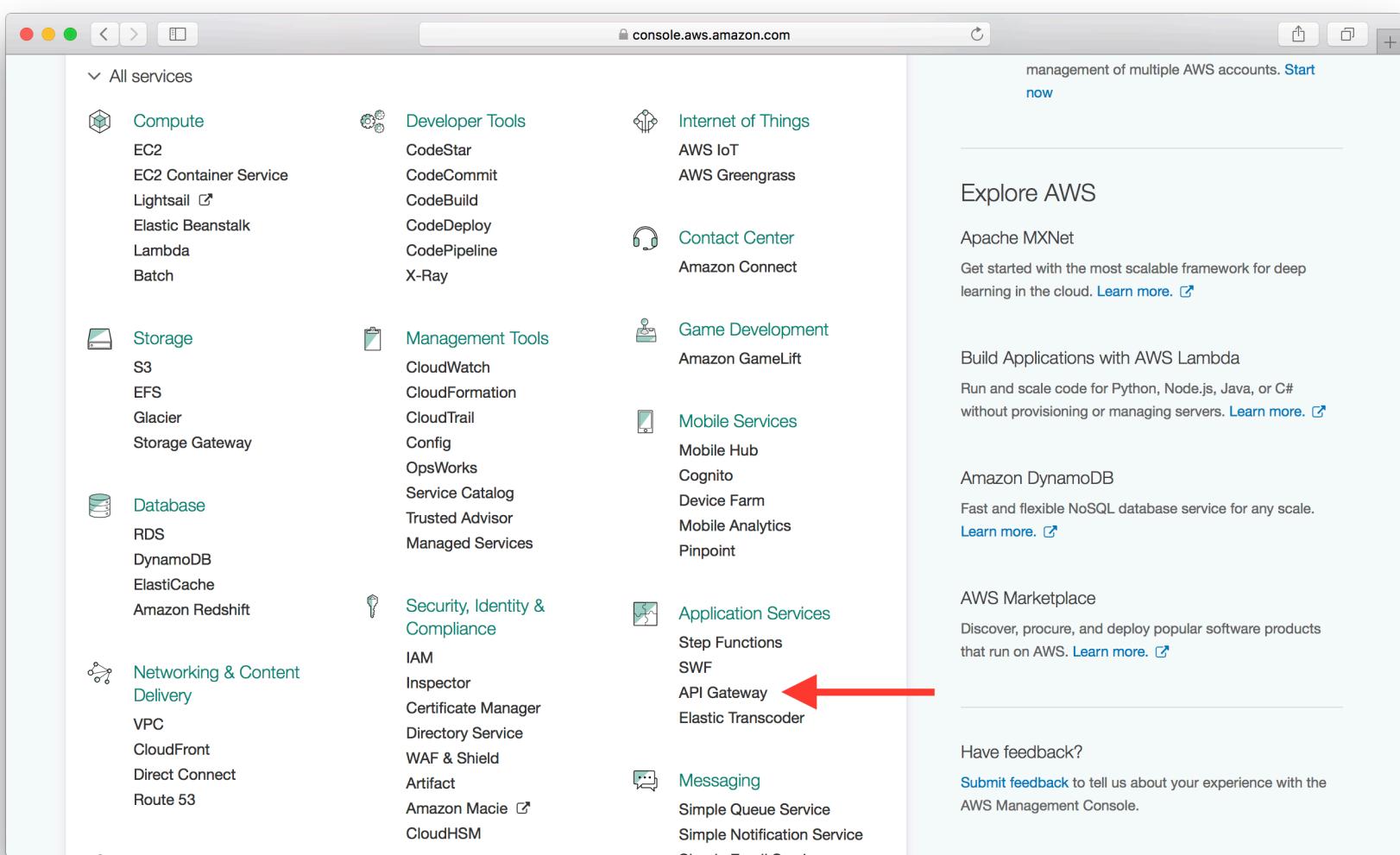
Below this, there are tabs for 'Permissions', 'Trust relationships', 'Access Advisor', and 'Revoke sessions'. The 'Permissions' tab is active, showing 'Attached policies: 1'. A table lists the attached policy:

Policy name	Policy type
AmazonAPIGatewayPushToCloudWatchLogs	AWS managed policy

At the bottom right of the permissions section, there's a link 'Add inline policy'.

Now that we have created our IAM role, let's turn on logging for our API Gateway project.

Go back to your AWS Console (<https://console.aws.amazon.com>) and select API Gateway from the list of services.



Select **Settings** from the left panel.

The screenshot shows the AWS API Gateway console at console.aws.amazon.com. The top navigation bar includes 'Services' (with a dropdown for 'Resource Groups'), 'Amazon API Gateway', 'APIs', and links for 'Show all hints' and '?'. On the left, a sidebar titled 'APIs' lists 'prod-notes-app-api', 'Usage Plans', 'API Keys', 'Custom Domain Names', 'Client Certificates', and 'Settings'. A red arrow points to the 'Settings' link. The main content area shows a card for 'prod-notes-app-api' with the placeholder text 'No description.' A blue 'Create API' button is visible at the top left of the main content area.

Enter the ARN of the IAM role we just created in the **CloudWatch log role ARN** field and hit **Save**.

The screenshot shows the AWS API Gateway Settings page for the 'prod-notes-app-api' stage. The left sidebar lists various API-related options like APIs, Usage Plans, API Keys, etc. The 'Settings' option is selected. The main panel is titled 'Settings' and contains a note about providing a CloudWatch log role ARN. A text input field contains the ARN 'arn:aws:iam::207542341204:role/APIGatewayCloudWatchLogs'. Below this is a section for 'Account level throttling' with a note about the current rate. At the bottom, there's a 'Required' label and a prominent blue 'Save' button. Red arrows point from the text input field and the 'Save' button towards the right.

Select your API project from the left panel, select **Stages**, then pick the stage you want to enable logging for. For the case of our Notes App API (<https://github.com/AnomalyInnovations/serverless-stack-demo-api>), we deployed to the **prod** stage.

The screenshot shows the 'prod Stage Editor' interface in the Amazon API Gateway console. The left sidebar lists various API resources like APIs, Resources, Authorizers, etc., with 'Stages' selected. The breadcrumb navigation shows 'APIs > prod-notes-app-api (5by75p4gn3) > Stages > prod'. The main area is titled 'prod Stage Editor' and contains tabs for Settings, Stage Variables, SDK Generation, Export, Deployment History, and Documentation History. Under the Settings tab, there are sections for Cache Settings (with 'Enable API cache' unchecked), CloudWatch Settings (with 'Enable CloudWatch Logs' and 'Enable Detailed CloudWatch Metrics' both unchecked), and Default Method Throttling (with 'Enable throttling' checked and a rate of 1000 requests per second). An 'Invoke URL' is provided at the top right.

In the **Settings** tab:

- Check **Enable CloudWatch Logs**.
- Select **INFO** for **Log level** to log every request.
- Check **Log full requests/responses** data to include entire request and response body in the log.
- Check **Enable Detailed CloudWatch Metrics** to track latencies and errors in CloudWatch metrics.

APIs > prod-notes-app-api (5by75p4gn3) > Stages > prod

prod Stage Editor

Invoke URL: <https://5by75p4gn3.execute-api.us-east-1.amazonaws.com/prod>

Settings Stage Variables SDK Generation Export Deployment History Documentation History

Configure the metering and caching settings for the **prod** stage.

Cache Settings

Enable API cache

CloudWatch Settings

Enable CloudWatch Logs ↗

Log level **INFO** ↗

Log full requests/responses data ↗

Enable Detailed CloudWatch Metrics ↗

Default Method Throttling

Choose the default throttling level for the methods in this stage. Each method in this stage will respect these rate and burst settings. Your current account level throttling rate is **10000** requests per second with a burst of **5000** requests. ↗

Feedback English (US) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Scroll to the bottom of the page and click **Save Changes**.

APIs > prod-notes-app-api (5by75p4gn3) > Stages > prod

CloudWatch Settings

Enable CloudWatch Logs ↗

Log level **INFO** ↗

Log full requests/responses data ↗

Enable Detailed CloudWatch Metrics ↗

Default Method Throttling

Choose the default throttling level for the methods in this stage. Each method in this stage will respect these rate and burst settings. Your current account level throttling rate is **10000** requests per second with a burst of **5000** requests. ↗

Enable throttling ↗

Rate **1000** requests per second ↗

Burst **2000** requests ↗

Client Certificate

Select the client certificate that API Gateway will use to call your integration endpoints in this stage.

Certificate **None** ↗

Save Changes

Feedback English (US) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Now our API Gateway requests should be logged via CloudWatch.

Enable Lambda CloudWatch Logs

Lambda CloudWatch logs are enabled by default. It tracks the duration and max memory usage for each execution. You can write additional information to CloudWatch via `console.log`.

For example:

```
export function main(event, context, callback) {  
    console.log('Hello world');  
    callback(null, { body: '' });  
}
```

Viewing API Gateway CloudWatch Logs

CloudWatch groups log entries into **Log Groups** and then further into **Log Streams**. Log Groups and Log Streams can mean different things for different AWS services. For API Gateway, when logging is first enabled in an API project's stage, API Gateway creates 1 log group for the stage, and 300 log streams in the group ready to store log entries. API Gateway picks one of these streams when there is an incoming request.

To view API Gateway logs, log in to your AWS Console (<https://console.aws.amazon.com>) and select CloudWatch from the list of services.

The screenshot shows the AWS Management Console homepage. The left sidebar lists services under 'All services' including Compute, Storage, Database, Networking & Content Delivery, and Management Tools. A red arrow points from the 'Storage' section to the 'Management Tools' section, which contains CloudWatch, CloudFormation, CloudTrail, Config, OpsWorks, Service Catalog, Trusted Advisor, and Managed Services. The right panel features sections like 'Explore AWS', 'Build Applications with AWS Lambda', 'Amazon DynamoDB', 'AWS Marketplace', and a feedback section.

Select **Logs** from the left panel.

The screenshot shows the AWS CloudWatch Metrics console. The left sidebar has a 'Logs' link highlighted with a red arrow. The main area displays the 'Metric Summary' section, which shows 1,100 metrics available in the US East (N. Virginia) region. It includes a search bar and a 'Browse Metrics' button. Below it is the 'Alarm Summary' section, which shows 66 alarms in the 'INSUFFICIENT' state. It includes a 'Create Alarm' button and a 'See top 20 alarms' section with four line charts. The right sidebar provides 'Additional Info' links.

Select the log group prefixed with **API-Gateway-Execution-Logs_** followed by the API Gateway id.

The screenshot shows the AWS CloudWatch Log Groups page. On the left sidebar, under the 'Logs' section, there is a red arrow pointing to the 'Metrics' link. The main content area displays a table of log groups:

Log Groups	Expire Events After	Metric Filters	Subscriptions
/aws/lambda/notes-app-api-prod-create	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-delete	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-get	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-list	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-update	Never Expire	0 filters	None
API-Gateway-Execution-Logs_5by75p4gn3/prod	Never Expire	0 filters	None

You should see 300 log streams ordered by the last event time. This is the last time a request was recorded. Select the first stream.

This screenshot shows the AWS CloudWatch Log Streams page. The left sidebar is titled 'Logs' and lists various CloudWatch services with their status: ALARM (0), INSUFFICIENT (66), OK (64), Billing, Events, Rules, Event Buses (NEW), Metrics. The main content area shows a list of log streams under 'Streams for API-Gateway-Execution-Logs_...'. The list is filtered by 'Log Stream Name Prefix' and sorted by 'Last Event Time'. A red arrow points to the first log stream in the list, which has a timestamp of '2017-09-13 11:08 UTC-4'. The table columns are 'Log Streams' and 'Last Event Time'.

Log Streams	Last Event Time
ec8956637a99787bd197eacd77acce5e	2017-09-13 11:08 UTC-4
8f53295a73878494e9bc8dd6c3c7104f	2017-09-13 11:08 UTC-4
49182f81e6a13cf5eaa496d51fea6406	2017-09-13 11:08 UTC-4
0a09c8844ba8f0936c20bd791130d6b6	2017-09-13 11:08 UTC-4
0777d5c17d4066b82ab86dff8a46af6f	2017-09-13 11:08 UTC-4
045117b0e0a11a242b9765e79cbf113f	2017-09-13 11:08 UTC-4
a4a042cf4fd6fb47701cbc8a1653ada	2017-09-13 11:08 UTC-4
a1d0c6e83f027327d8461063f4ac58a6	2017-09-13 11:08 UTC-4
02522a2b2726fb0a03bb19f2d8d9524d	2017-09-13 11:08 UTC-4
45c48cce2e2d7fbdea1afc51c7c6ad26	2017-09-13 11:08 UTC-4
2b44928ae11fb9384c4cf38708677c48	2017-09-13 11:08 UTC-4
07e1cd7dca89a1678042477183b7ac3f	2017-09-13 11:08 UTC-4
6364d3f0f495b6ab9dcf8d3b5c6e0b01	2017-09-13 11:08 UTC-4
182be0c5cdcd5072bb1864cdee4d3d6e	2017-09-13 11:08 UTC-4
42a0e188f5033bc65bf8d78622277c4e	2017-09-13 11:07 UTC-4
ef0d3930a7b6c95bd2b32ed45989c61f	2017-09-13 11:07 UTC-4
65b9eea6e1cc6bb9f0cd2a47751a186f	2017-09-13 11:07 UTC-4
f90f2aca5c640289d0a29417bcb63a37	2017-09-13 11:07 UTC-4

This shows you the log entries grouped by request.

This screenshot shows the AWS CloudWatch Log Events page. The left sidebar is identical to the previous screenshot. The main content area shows the details for the first log stream from the previous screenshot, specifically for the entry on '2017-09-13'. The log entries are grouped by time (UTC +00:00). A red arrow points to the timestamp '2017-09-13' in the header of the second group of log entries. The table columns are 'Time (UTC +00:00)' and 'Message'.

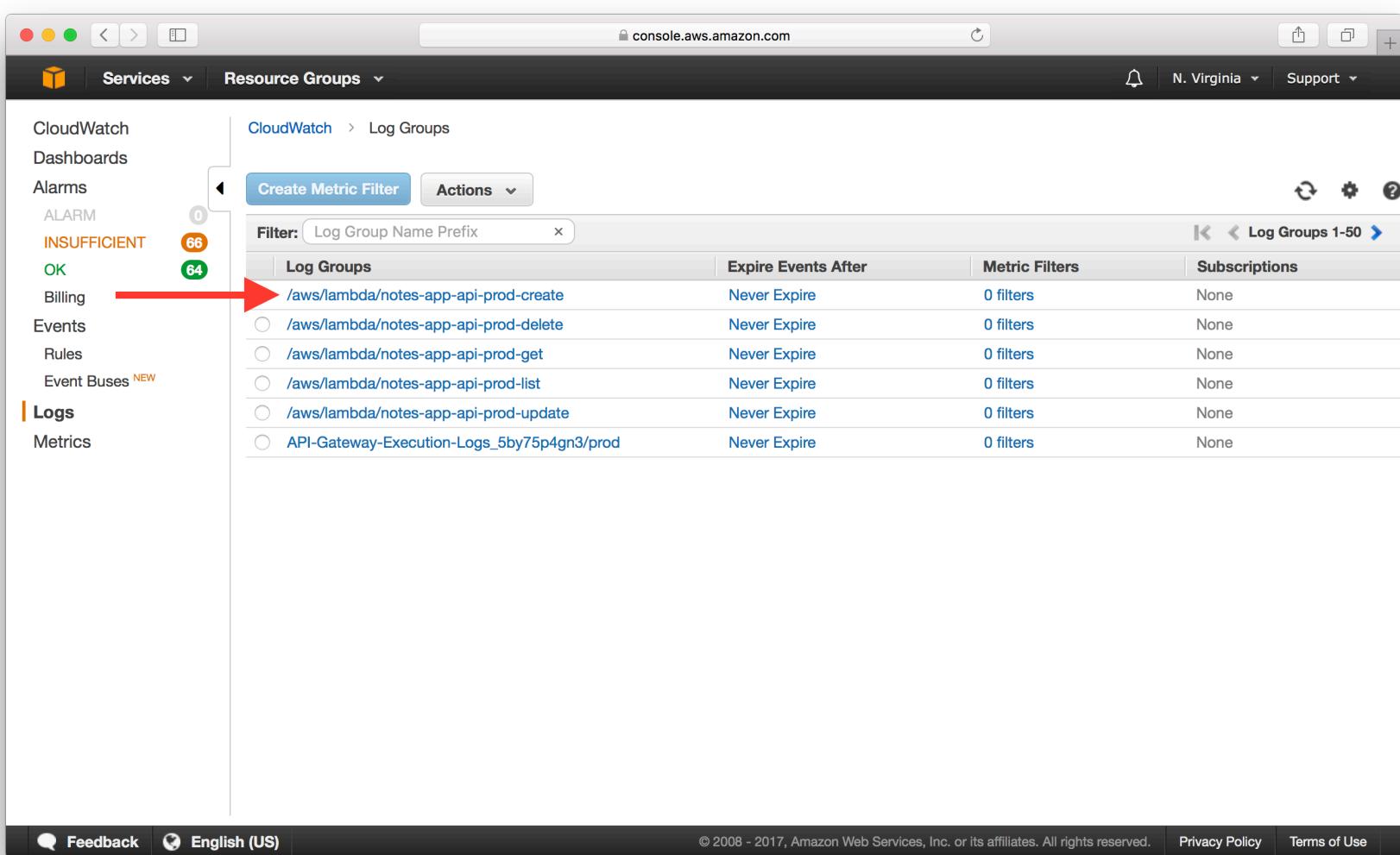
Time (UTC +00:00)	Message
2017-09-05	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method request path: {id=de1b2bf0-926e-11e7-9c19-35de5cc401ec}
19:18:00	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method request query string: {}
19:18:00	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method request headers: {X-AMZ-Date=20170905T191756Z, Accept=application/json, C}
19:18:00	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method request body before transformations:
19:18:00	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Endpoint request URI: https://lambda.us-east-1.amazonaws.com/2015-03-31/functions/a
19:18:00	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Endpoint request headers: {x-amzn-lambda-integration-tag=eeb3ad80-926e-11e7-ac8d-
19:18:00	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Endpoint request body after transformations: {"resource":"/notes/{id}", "path":"/notes/de1
19:18:01	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Endpoint response body before transformations: {"statusCode":200, "headers":{ "Access-I
19:18:01	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Endpoint response headers: {x-amzn-Remapped-Content-Length=0, x-amzn-RequestId=
19:18:01	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method response body after transformations: {"attachment":"https://notes-app-uploads.s
19:18:01	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method response headers: {Access-Control-Allow-Origin=*, Access-Control-Allow-Crede
19:18:01	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Successfully completed execution
19:18:01	(eeb3ad80-926e-11e7-ac8d-b7ef47cf7d48) Method completed with status: 200
2017-09-13	(6622b95c-9895-11e7-87c5-772559cc6127) Verifying Usage Plan for request: 6622b95c-9895-11e7-87c5-772559cc6127. API Key: A
15:08:28	(6622b95c-9895-11e7-87c5-772559cc6127) API Key authorized because method 'GET /notes' does not require API Key. Request wil
15:08:28	(6622b95c-9895-11e7-87c5-772559cc6127) Usage Plan check succeeded for API Key and API Stage 5by75p4gn3/prod
15:08:28	(6622b95c-9895-11e7-87c5-772559cc6127) Starting execution for request: 6622b95c-9895-11e7-87c5-772559cc6127
15:08:28	(6622b95c-9895-11e7-87c5-772559cc6127) HTTP Method: GET, Resource Path: /notes
15:08:28	(6622b95c-9895-11e7-87c5-772559cc6127) Method request path: {}
15:08:28	(6622b95c-9895-11e7-87c5-772559cc6127) Method request query string: {}

Note that two consecutive groups of logs are not necessarily two consecutive requests in real time. This is because there might be other requests that are processed in between these two that were picked up by one of the other log streams.

Viewing Lambda CloudWatch Logs

For Lambda, each function has its own log group. And the log stream rotates if a new version of the Lambda function has been deployed or if it has been idle for some time.

To view Lambda logs, select **Logs** again from the left panel. Then select the first log group prefixed with **/aws/lambda/** followed by the function name.



The screenshot shows the AWS CloudWatch Log Groups page. On the left sidebar, under the 'Logs' section, the 'CloudWatch' link is highlighted with a red arrow. The main content area displays a table of log groups:

Log Groups	Expire Events After	Metric Filters	Subscriptions
/aws/lambda/notes-app-api-prod-create	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-delete	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-get	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-list	Never Expire	0 filters	None
/aws/lambda/notes-app-api-prod-update	Never Expire	0 filters	None
API-Gateway-Execution-Logs_5by75p4gn3/prod	Never Expire	0 filters	None

Select the first stream.

The screenshot shows the AWS CloudWatch Log Groups interface. On the left sidebar, under the 'Logs' section, there are several status indicators: ALARM (0), INSUFFICIENT (66), OK (64), Billing, Events, Rules, and Event Buses (NEW). The 'Logs' section is currently selected. In the main content area, the path 'CloudWatch > Log Groups > Streams for /aws/lambda/notes-app-api-p...' is displayed. Below this, there are three buttons: 'Search Log Group', 'Create Log Stream', and 'Delete Log Stream'. A search bar labeled 'Filter: Log Stream Name Prefix' is present. The main table lists 'Log Streams' with columns for 'Last Event Time'. A red arrow points to the first row of the table, which contains the timestamp '2017-09-13 [\$LATEST]218f5330a344456ea7960fc105d1656d'. The table also includes rows for various other log stream names and their last event times.

Last Event Time
2017-09-13 11:08 UTC-4
2017-09-13 05:15 UTC-4
2017-09-13 01:21 UTC-4
2017-09-13 00:05 UTC-4
2017-09-12 12:55 UTC-4
2017-09-12 10:04 UTC-4
2017-09-12 09:12 UTC-4
2017-09-12 05:13 UTC-4
2017-09-12 04:04 UTC-4
2017-09-12 03:26 UTC-4
2017-09-11 23:28 UTC-4
2017-09-11 16:49 UTC-4
2017-09-11 15:10 UTC-4
2017-09-11 09:57 UTC-4
2017-09-11 08:27 UTC-4
2017-09-10 20:36 UTC-4
2017-09-10 20:13 UTC-4
2017-09-10 09:06 UTC-4

You should see **START**, **END** and **REPORT** with basic execution information for each function invocation. You can also see content logged via `console.log` in your Lambda code.

The screenshot shows the AWS CloudWatch Logs interface. On the left, a sidebar lists services: CloudWatch, Dashboards, Alarms, ALARM (INSUFFICIENT: 66, OK: 64), Billing, Events, Rules, Event Buses (NEW), Logs (selected), and Metrics. The main area shows a log group path: CloudWatch > Log Groups > /aws/lambda/notes-app-api-prod-create > 2017/09/13/[LATEST]218f5330a344456ea7960fc105d1656d. A table displays log events with columns for Time (UTC +00:00) and Message. The table header includes a 'Filter events' input and time range controls (all, 30s, 5m, 1h, 6h, 1d, 1w, custom). The log entries show requests starting at 15:07:34 and ending at 15:08:03, with details like RequestId and Duration. A message at the bottom states 'No newer events found at the moment. Retry.'

You can also use the Serverless CLI to view CloudWatch logs for a Lambda function.

From your project root run the following.

```
$ serverless logs -f <func-name>
```

Where the `<func-name>` is the name of the Lambda function you are looking for.

Additionally, you can use the `--tail` flag to stream the logs automatically to your console.

```
$ serverless logs -f <func-name> --tail
```

This can be very helpful during development when trying to debug your functions using the `console.log` call.

Hopefully, this has helped you set up CloudWatch logging for your API Gateway and Lambda projects. And given you a quick idea of how to read your serverless logs using the AWS Console.

Comments on this chapter
(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/146>)

Debugging Serverless API Issues

In this chapter we are going to take a brief look at some common API Gateway and Lambda issues we come across and how to debug them.

When a request is made to your serverless API, it starts by hitting API Gateway and makes its way through to Lambda and invokes your function. It takes quite a few hops along the way and each hop can be a point of failure. And since we don't have great visibility over each of the specific hops, pinpointing the issue can be a bit tricky. We are going to take a look at the following issues:

- Invalid API Endpoint (#invalid-api-endpoint)
- Missing IAM Policy (#missing-iam-policy)
- Lambda Function Error (#lambda-function-error)
- Lambda Function Timeout (#lambda-function-timeout)

This chapter assumes you have turned on CloudWatch logging for API Gateway and that you know how to read both the API Gateway and Lambda logs. If you have not done so, start by taking a look at the chapter on API Gateway and Lambda Logs (/chapters/api-gateway-and-lambda-logs.html).

Invalid API Endpoint

The first and most basic issue we see is when the API Gateway endpoint that is requested is invalid. An API Gateway endpoint usually looks something like this:

```
https://API_ID.execute-api.REGION.amazonaws.com/STAGE/PATH
```

- **API_ID** - a unique identifier per API Gateway project
- **REGION** - the AWS region in which the API Gateway project is deployed to
- **STAGE** - the stage of the project (defined in your `serverless.yml` or passed in through the `serverless deploy -stage` command)
- **PATH** - the path of an API endpoint (defined in your `serverless.yml` for each function)

An API request will fail if:

- The **API_ID** is not found in the specified **REGION**
- The API Gateway project does not have the specified **STAGE**
- API endpoint invoked does not match a pre-defined **PATH**

In all of these cases, the error does not get logged to CloudWatch since the request does not hit your API Gateway project.

Missing IAM Policy

This happens when your API endpoint uses **aws_iam** as the authorizer, and the IAM role assigned to the Cognito Identity Pool has not been granted the **execute-api:Invoke** permission for your API Gateway resource.

This is a tricky issue to debug because the request still has not reached API Gateway, and hence the error is not logged in the API Gateway CloudWatch logs. But we can perform a check to ensure that our Cognito Identity Pool users have the required permissions, using the IAM policy Simulator (<https://policysim.aws.amazon.com>).

Before we can use the simulator we first need to find out the name of the IAM role that we are using to connect to API Gateway. We had created this role back in the Create a Cognito identity pool (/chapters/create-a-cognito-identity-pool.html) chapter.

Select **Cognito** from your AWS Console (<https://console.aws.amazon.com>).

Screenshot of the AWS Management Console homepage. A red arrow points to the 'Cognito' service icon in the 'Recently visited services' section.

AWS services

Find a service by name or feature (for example, EC2, S3 or VM, storage).

Recently visited services: Cognito (highlighted with a red arrow), IAM, CloudWatch, S3, Route 53.

All services:

- Compute**: EC2, EC2 Container Service, Lightsail, Elastic Beanstalk, Lambda, Batch.
- Storage**: S3, EFS, Glacier, Storage Gateway.
- Database**: RDS.
- Developer Tools**: CodeStar, CodeCommit, CodeBuild, CodeDeploy, CodePipeline, X-Ray.
- Management Tools**: CloudWatch, CloudFormation, CloudTrail, Config, OpsWorks, Service Catalog, Trusted Advisor, Managed Services.
- Internet of Things**: AWS IoT, AWS Greengrass.
- Contact Center**: Amazon Connect.
- Game Development**: Amazon GameLift.
- Mobile Services**: Mobile Hub, Cognito, Device Farm, Mobile Analytics, Pinpoint.

Helpful tips

- Manage your costs**: Get real-time billing alerts based on your cost and usage budgets. [Start now](#).
- Create an organization**: Use AWS Organizations for policy-based management of multiple AWS accounts. [Start now](#).

Explore AWS

- Apache MXNet**: Get started with the most scalable framework for deep learning in the cloud. [Learn more](#).
- Build Applications with AWS Lambda**: Run and scale code for Python, Node.js, Java, or C# without provisioning or managing servers. [Learn more](#).
- Amazon DynamoDB**: Fast and flexible NoSQL database service for any scale. [Learn more](#).

Next hit the **Manage Federated Identities** button.

Screenshot of the Amazon Cognito service page.

The 'Manage Federated Identities' button is highlighted with a red arrow.

Amazon Cognito

Amazon Cognito makes it easy for you to have users sign up and sign in to your apps, federate identities from social identity providers, secure access to AWS resources and synchronize data across multiple devices, platforms, and applications.

[Manage your User Pools](#) [Manage Federated Identities](#)

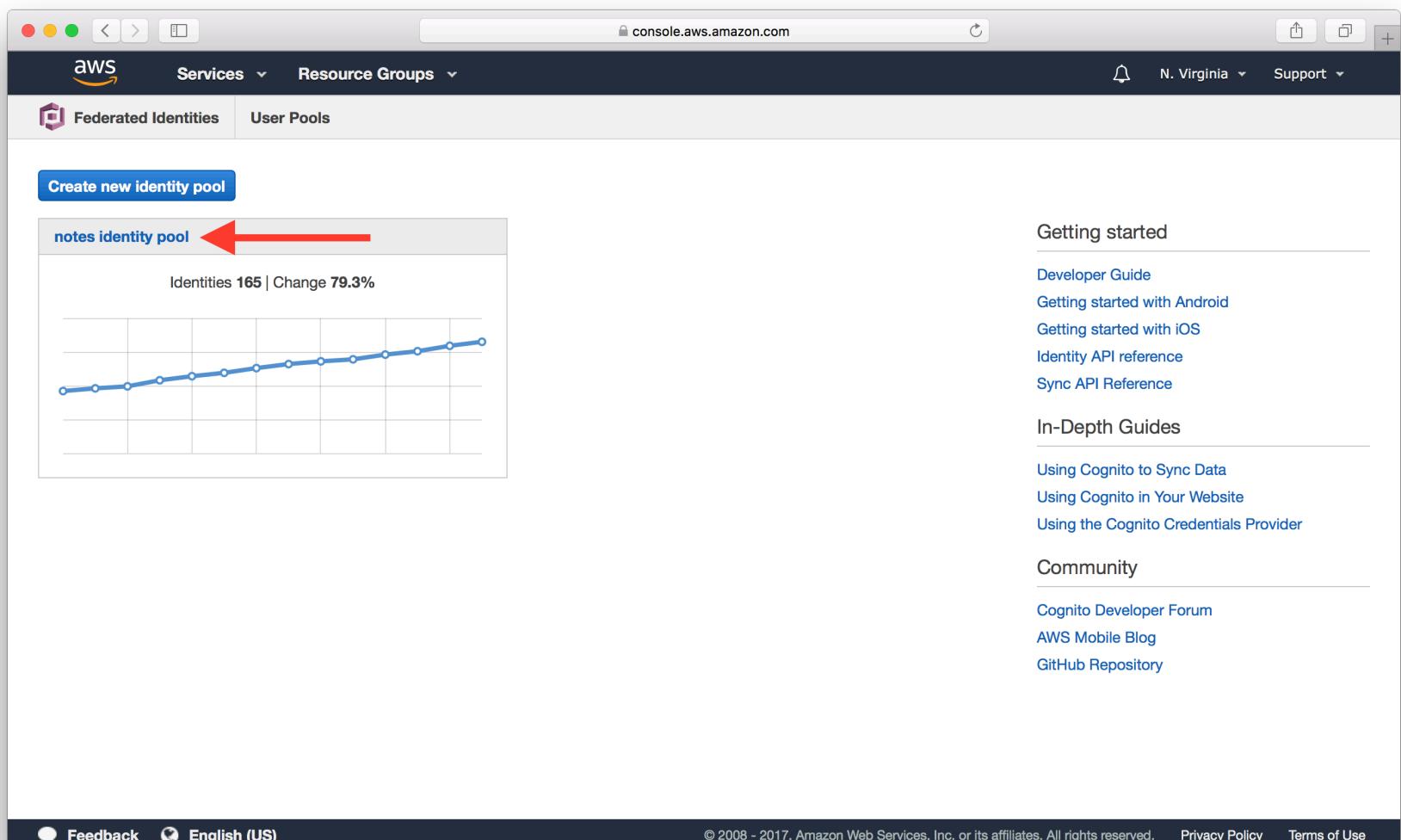
Add Sign-up and Sign-in

Federate User Identities

Synchronize Data Across Devices

With Cognito User Pools, you can easily and securely ...
With Cognito Federated Identities, your users can sign-in ...
With Cognito Sync, your app can save user data, such as ...

And select your Identity Pool. In our case it's called `notes identity pool`.



The screenshot shows the AWS Cognito console interface. At the top, there are navigation links for AWS, Services (with a dropdown for Resource Groups), and Resource Groups. On the far right, there are links for Support and N. Virginia (region). Below the navigation bar, there are two tabs: 'Federated Identities' (selected) and 'User Pools'. A prominent blue button labeled 'Create new identity pool' is visible. To its right, a list displays an identity pool named 'notes identity pool'. This entry includes a small icon, the pool name, a count of 'Identities 165', and a percentage change of '79.3%'. Below this information is a line graph showing the growth of identities over time. To the right of the main content area, there is a sidebar with several sections: 'Getting started' (Developer Guide, Getting started with Android, Getting started with iOS, Identity API reference, Sync API Reference), 'In-Depth Guides' (Using Cognito to Sync Data, Using Cognito in Your Website, Using the Cognito Credentials Provider), and 'Community' (Cognito Developer Forum, AWS Mobile Blog, GitHub Repository). At the bottom of the page, there are links for Feedback, English (US) language selection, and legal notices: © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved., Privacy Policy, and Terms of Use.

Click **Edit identity pool** at the top right.

Screenshot of the AWS Cognito Federated Identities service dashboard for the 'notes identity pool'.

The dashboard shows the following key metrics:

- Identity pool**:
 - Dashboard** (selected)
 - Sample code
 - Identity browser
- Identities this month**: 150
- Total identities**: 165
- Cognito Sync** help message: "Cognito Sync helps you sync user data across devices. Get started using the Mobile SDK: [Android](#), [iOS](#)".
- Authentication methods**: us-east-1_udmFFSb92 (100.0% completion, 165 users)
- Graph**: A line graph showing the trend of total identities over the past 14 days. The Y-axis ranges from 0 to 200, and the X-axis shows dates from Sep 14 to Sep 27. The line starts at approximately 90 on Sep 14 and rises steadily to about 170 by Sep 27.
- Resources**:
 - Getting started with Amazon Cognito**: User authentication is just the beginning. Learn how to store data in a user's profile, synchronize it across all of their devices, and
 - Learn about the AWS Mobile SDKs**: Amazon Cognito is one of the services that comes bundled with the AWS Mobile SDK. Clients for S3, DynamoDB, and many other services are also included in the AWS Mobile SDK. Browse the documentation to learn how
 - Connect with the community**:
 - [Cognito community forum](#)
 - [AWS Mobile Blog](#)
 - [GitHub repository](#)

Here make a note of the name of the **Authenticated role**. In our case it is

`Cognito_notesidentitypoolAuth_Role`.

Screenshot of the AWS Federated Identities console showing the 'Edit identity pool' page for 'notes identity pool'. The page displays the identity pool name ('notes identity pool'), ID ('us-east-1:ceef8ccc-0a19-4616-9067-854dc69c2d82'), and two roles: 'Unauthenticated role' (selected) and 'Authenticated role'. A red arrow points upwards from the bottom of the page towards the 'Unauthenticated identities' section.

Identity pool name* notes identity pool

Identity pool ID ⓘ us-east-1:ceef8ccc-0a19-4616-9067-854dc69c2d82 (Show ARN)

Unauthenticated role ⓘ Cognito_notesidentitypoolUnauth_Role Create new role

Authenticated role ⓘ Cognito_notesidentitypoolAuth_Role Create new role

▶ Unauthenticated identities ⓘ

▶ Authentication providers ⓘ

▶ Push synchronization

▶ Cognito Streams

Now that we know the IAM role we are testing, let's open up the IAM Policy Simulator (<https://policysim.aws.amazon.com>).

The screenshot shows the IAM Policy Simulator interface. On the left, there is a sidebar titled "Users, Groups, and Roles" with a dropdown menu set to "Users" and a "Filter" input field. Below the dropdown is a list containing the entry "admin". On the right, the main area is titled "Policy Simulator" and contains a "Global Settings" section with a link to "Action Settings and Results" (0 actions selected). Below this is a table header with columns: Service, Action, Resource Type, Simulation Resource, and Permission.

Select Roles.

This screenshot is similar to the first one, but the "Select service" dropdown in the sidebar has been changed from "Users" to "Roles". A red arrow points to the "Roles" dropdown button. The list of roles now includes "APIGatewayLogging", "Cognito_notesidentitypoolAuth_Role", and "Cognito_notesidentitypoolUnauth_Role". The rest of the interface remains the same, showing the "Policy Simulator" area with its settings and table header.

Select the IAM role that we made a note of in the steps above. In our case it is

Cognito_notesidentitypoolAuth_Role .

The screenshot shows the IAM Policy Simulator interface. On the left, a sidebar titled "Policies" shows a selected role: "Cognito_notesidentitypoolAuth_Role". A red arrow points upwards from the bottom of this sidebar towards the main "Policy Simulator" area. The main area has a header "Policy Simulator" with buttons for "Mode : Existing Policies", "Run Simulation", and others. Below the header is a section for "Global Settings" and a table titled "Action Settings and Results" which is currently empty. The table has columns: Service, Action, Resource Type, Simulation Resource, and Permission.

Select API Gateway as the service and select the Invoke action.

The screenshot shows the IAM Policy Simulator interface. On the left, there's a sidebar with 'Policies' and 'Selected role: Cognito_notesidentitypoolAuth_Role'. The main area is titled 'Policy Simulator' and shows 'Amazon API G...' selected. Under 'Global Settings', the 'Invoke' checkbox is checked (highlighted by a red arrow). Below this is a table titled 'Action Settings and Requests' with one row: 'Amazon API Gateway' has 'Action' set to 'Invoke' and 'Resource Type' to 'not required'. Buttons at the top include 'Select All', 'Deselect All', 'Reset Contexts', 'Clear Results', and 'Run Simulation'.

Expand the service and enter the API Gateway endpoint ARN, then select **Run Simulation**. The format here is the same one we used back in the Create a Cognito identity pool (/chapters/create-a-cognito-identity-pool.html) chapter; `arn:aws:execute-api:YOUR_API_GATEWAY_REGION:*:YOUR_API_GATEWAY_ID/*`. In our case this looks like `arn:aws:execute-api:us-east-1:ly55wbovq4/*`.

The screenshot shows the IAM Policy Simulator interface. On the left, there's a sidebar with 'Policies' and 'IAM Policies' sections. The 'Selected role' dropdown is set to 'Cognito_notesidentitypoolAuth_Role'. In the main area, under 'Global Settings', the 'Action' dropdown is set to 'Invoke' and the 'Service' dropdown is set to 'Amazon API Gateway'. The 'Resource' field contains 'arn:aws:execute-api:us-east-1:*:ly55wbovq4/*'. Below this, a table shows a single row: 'Amazon API Gateway' with 'Action' 'Invoke', 'Resource Type' 'execute-api-general', 'Simulation Resource' 'execute-api-general', and 'Permission' 'Not simulated'. A red arrow points to the 'Run Simulation' button at the top right of the table.

If your IAM role is configured properly you should see **allowed** under **Permission**.

This screenshot shows the same IAM Policy Simulator interface as the previous one, but the simulation has succeeded. The 'Resource' field still contains 'arn:aws:execute-api:us-east-1:*:ly55wbovq4/*'. The table now shows a single row: 'Amazon API Gateway' with 'Action' 'Invoke', 'Resource Type' 'execute-api-general', 'Simulation Resource' 'execute-api-general', and 'Permission' 'allowed' with the note '1 matching statements.' A red arrow points to the 'Run Simulation' button at the top right of the table.

But if something is off, you'll see denied.

The screenshot shows the IAM Policy Simulator interface. On the left, there's a sidebar titled 'Policies' with a 'Back' button. It lists a selected role: 'Cognito_notesidentitypoolAuth_Role'. Below it is a 'Filter' input field. Under 'Resource Policies', there is no content. The main area is titled 'Policy Simulator' and contains a 'Global Settings' section with a link to 'Action Settings and Results'. This section displays a table with one row:

Service	Action	Resource Type	Simulation Resource	Permission
Amazon API Gateway	Invoke	execute-api-general	execute-api-general	denied Implicitly denied (no mat...)

A red arrow points upwards from the bottom right towards the 'denied' status in the table.

To fix this and edit the role we need to go back to the AWS Console (<https://console.aws.amazon.com>) and select IAM from the list of services.

The screenshot shows the AWS Management Console homepage. The left sidebar contains several service categories: Lambda, Batch, Storage (with S3, EFS, Glacier, Storage Gateway), Database (with RDS, DynamoDB, ElastiCache, Amazon Redshift), Networking & Content Delivery (with VPC, CloudFront, Direct Connect, Route 53), Migration (with AWS Migration Hub, Application Discovery Service, Database Migration Service, Server Migration Service, Snowball), and Analytics (with Athena, EMR, CloudSearch, Elasticsearch Service, Kinesis, Data Pipeline). A red arrow points to the 'IAM' link under the 'Security, Identity & Compliance' section.

Select Roles on the left menu.

The screenshot shows the AWS IAM service dashboard. The left sidebar has a 'Dashboard' section with links for Groups, Users, Roles (which is highlighted with a red arrow), Policies, Identity providers, Account settings, Credential report, and Encryption keys. The main content area displays the 'Welcome to Identity and Access Management' page. It includes a 'Search IAM' bar, a 'Feature Spotlight' video player, and sections for IAM Resources (Users: 0, Groups: 0, Customer Managed Policies: 0) and Security Status (with four items: 'Activate MFA on your root account', 'Create individual IAM users', 'Use groups to assign permissions', and 'Apply an IAM password policy').

And select the IAM role that our Identity Pool is using. In our case it's called `Cognito_notesidentitypoolAuth_Role`.

The screenshot shows the AWS IAM Roles page. The search bar at the top contains the text "Cognito_notesidentitypoolAuth". Below the search bar, there is a table with three columns: "Role name", "Description", and "Trusted entities". A single result is listed: "Cognito_notesidentitypoolAuth..." with the description "Identity Provider: cognito-identity.amazonaws.com". A red arrow points to the "Role name" column of this result. The left sidebar has a "Roles" section selected, which is highlighted with an orange border. Other options in the sidebar include "Dashboard", "Groups", "Users", "Policies", "Identity providers", "Account settings", "Credential report", and "Encryption keys". At the bottom of the page, there are links for "Feedback", "English (US)", and copyright information: "© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved." followed by "Privacy Policy" and "Terms of Use".

Expand the policy under the list of policies.

The screenshot shows the AWS IAM Roles Summary page for the role 'Cognito_notesidentitypoolAuth_Role'. The 'Attached policies' section displays one policy: 'oneClick_Cognito_notesidentitypoolAuth_Role_1506534839331', which is an inline policy. A red arrow points upwards from the bottom of the attached policy table towards the 'Edit' link in the top right corner of the summary section.

Role ARN: arn:aws:iam::232771856781:role/Cognito_notesidentitypoolAuth_Role

Role description:

Instance Profile ARNs:

Path: /

Creation time: 2017-09-27 13:54 EDT

Attached policies: 1

Policy name	Policy type
oneClick_Cognito_notesidentitypoolAuth_Role_1506534839331	Inline policy

Click Edit policy.

The screenshot shows the same AWS IAM Roles Summary page as the previous one, but with a red arrow pointing to the 'Edit policy' button located at the bottom of the 'Attached policies' section. This button is used to view or modify the JSON-based policy document.

Role ARN: arn:aws:iam::232771856781:role/Cognito_notesidentitypoolAuth_Role

Role description:

Instance Profile ARNs:

Path: /

Creation time: 2017-09-27 13:54 EDT

Attached policies: 1

Policy name	Policy type
oneClick_Cognito_notesidentitypoolAuth_Role_1506534839331	Inline policy

Policy summary { } JSON **Edit policy** Simulate policy

```
1 {  
2   "Version": "2012-10-17",  
3   "Statement": [  
4     {  
5       "Effect": "Allow",  
6       "Action": [  
7         "mobileanalytics:PutEvents",  
8         "cognito-sync:*",  
9       ]  
10    }  
11  ]  
12}  
13
```

Here you can edit the policy to ensure that it has the right permission to invoke API Gateway. Ensure that there is a block in your policy like the one below.

```
...
{
    "Effect": "Allow",
    "Action": [
        "execute-api:Invoke"
    ],
    "Resource": [
        "arn:aws:execute-
api:YOUR_API_GATEWAY_REGION:*:YOUR_API_GATEWAY_ID/*"
    ]
}
...
...
```

Finally, hit **Save** to update the policy.

The screenshot shows the AWS IAM Policy Editor interface. At the top, the URL is console.aws.amazon.com. The policy name is "oneClick_Cognito_notesidentitypoolAuth_Role_15065348". The policy document is a JSON object:

```
1 {  
2     "Version": "2012-10-17",  
3     "Statement": [  
4         {  
5             "Effect": "Allow",  
6             "Action": [  
7                 "mobileanalytics:PutEvents",  
8                 "cognito-sync:*",  
9                 "cognito-identity:*"  
10            ],  
11            "Resource": [  
12                "*"  
13            ]  
14        },  
15        {  
16            "Effect": "Allow",  
17            "Action": [  
18                "s3:*"  
19            ],  
20            "Resource": [  
21                "*"  
22            ]  
23        }  
24    ]  
25}
```

At the bottom, there is a checkbox "Use autoformatting for policy editing" and three buttons: "Cancel", "Validate policy", and a blue "Save" button. A large red arrow points from the text above to the "Save" button.

Now if you test your policy, it should show that you are allowed to invoke your API Gateway endpoint.

Lambda Function Error

Now if you are able to invoke your Lambda function but it fails to execute properly due to uncaught exceptions, it'll error out. These are pretty straightforward to debug. When this happens, AWS Lambda will attempt to convert the error object to a string, and then send it to CloudWatch along with the stacktrace. This can be observed in both Lambda and API Gateway CloudWatch log groups.

Lambda Function Timeout

Sometimes we might run into a case where the Lambda function just times out. Normally, a Lambda function will end its execution by invoking the **callback** function that was passed in. By default, the callback will wait until the Node.js runtime event loop is empty before returning the results to the caller. If the Lambda function has an open connection to, let's say a database server, the event loop is not empty, and the callback will wait indefinitely until the connection is closed or the Lambda function times out.

To get around this issue, you can set this **callbackWaitsForEmptyEventLoop** property to false to request AWS Lambda to freeze the process as soon as the callback is called, even if there are events in the event loop.

```
export async function handler(event, context, callback) {  
  
    context.callbackWaitsForEmptyEventLoop = false;  
  
    ...  
};
```

This effectively allows a Lambda function to return its result to the caller without requiring that the database connection be closed. This allows the Lambda function to reuse the same connection across calls, and it reduces the execution time as well.

These are just a few of the common issues we see folks running into while working with serverless APIs. Feel free to let us know via the comments if there are any other issues you'd like us to cover.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/147>)

Serverless Environment Variables

In Node.js we use the `process.env` to get access to environment variables of the current process. In AWS Lambda, we can set environment variables that we can access via the `process.env` object.

Let's take a quick look at how to do that.

Defining Environment Variables

We can define our environment variables in our `serverless.yml` in two separate places. The first is in the `functions` section:

```
service: service-name

provider:
  name: aws
  stage: dev

functions:
  hello:
    handler: handler.hello
    environment:
      SYSTEM_URL: http://example.com/api/v1
```

Here `SYSTEM_URL` is the name of the environment variable we are defining and `http://example.com/api/v1` is its value. We can access this in our `hello` Lambda function using `process.env.SYSTEM_URL`, like so:

```
export function hello(event, context, callback) {
  callback(null, { body: process.env.SYSTEM_URL });
}
```

We can also define our environment variables globally in the `provider` section:

```
service: service-name

provider:
  name: aws
  stage: dev
  environment:
    SYSTEM_ID: jdoe

functions:
  hello:
    handler: handler.hello
    environment:
      SYSTEM_URL: http://example.com/api/v1
```

Just as before we can access the environment variable `SYSTEM_ID` in our `hello` Lambda function using `process.env.SYSTEM_ID`. The difference being that it is available to **all** the Lambda functions defined in our `serverless.yml`.

In the case where both the `provider` and `functions` section has an environment variable with the same name, the function specific environment variable takes precedence. As in, we can override the environment variables described in the `provider` section with the ones defined in the `functions` section.

Custom Variables in Serverless Framework

Serverless Framework builds on these ideas to make it easier to define and work with environment variables in our `serverless.yml` by generalizing the idea of variables (<https://serverless.com/framework/docs/providers/aws/guide/variables/>).

Let's take a quick look at how these work using an example. Say you had the following `serverless.yml`.

```
service: service-name

provider:
```

```
name: aws
stage: dev

functions:
  helloA:
    handler: handler.helloA
    environment:
      SYSTEM_URL: http://example.com/api/v1/pathA

  helloB:
    handler: handler.helloB
    environment:
      SYSTEM_URL: http://example.com/api/v1/pathB
```

In the case above we have the environment variable `SYSTEM_URL` defined in both the `helloA` and `helloB` Lambda functions. But the only difference between them is that the url ends with `pathA` or `pathB`. We can merge these two using the idea of variables.

A variable allows you to replace values in your `serverless.yml` dynamically. It uses the `${variableName}` syntax, where the value of `variableName` will be inserted.

Let's see how this works in practice. We can rewrite our example and simplify it by doing the following:

```
service: service-name

custom:
  systemUrl: http://example.com/api/v1/

provider:
  name: aws
  stage: dev

functions:
  helloA:
    handler: handler.helloA
    environment:
      SYSTEM_URL: ${self:custom.systemUrl}pathA
```

```
helloB:  
  handler: handler.helloB  
  environment:  
    SYSTEM_URL: ${self:custom.systemUrl}pathB
```

This should be pretty straightforward. We started by adding this section first:

```
custom:  
  systemUrl: http://example.com/api/v1/
```

This defines a variable called `systemUrl` under the section `custom`. We can then reference the variable using the syntax `${self:custom.systemUrl}`.

We do this in the environment variables `SYSTEM_URL`:

`${self:custom.systemUrl}pathA`. Serverless Framework parses this and inserts the value of `self:custom.systemUrl` and that combined with `pathA` at the end gives us the original value of `http://example.com/api/v1/pathA`.

Variables can be referenced from a lot of different sources including CLI options, external YAML files, etc. You can read more about using variables in your `serverless.yml` here (<https://serverless.com/framework/docs/providers/aws/guide/variables/>).

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/148>)

Stages in Serverless Framework

Serverless Framework allows you to create stages for your project to deploy to. Stages are useful for creating environments for testing and development. Typically you create a staging environment that is an independent clone of your production environment. This allows you to test and ensure that the version of code that you are about to deploy is good to go.

In this chapter we will take a look at how to configure stages in Serverless. Let's first start by looking at how stages can be implemented.

How Is Staging Implemented?

There are a couple of ways to set up stages for your project:

- Using the API Gateway built-in stages

You can create multiple stages within a single API Gateway project. Stages within the same project share the same endpoint host, but have a different path. For example, say you have a stage called `prod` with the endpoint:

```
https://abc12345.execute-api.us-east-1.amazonaws.com/prod
```

If you were to add a stage called `dev` to the same API Gateway API, the new stage will have the endpoint:

```
https://abc12345.execute-api.us-east-1.amazonaws.com/dev
```

The downside is that both stages are part of the same project. You don't have the same level of flexibility to fine tune the IAM policies for stages of the same API, when compared to tuning different APIs. This leads to the next setup, each stage being its own API.

- Separate APIs for each stage

You create an API Gateway project for each stage. Let's take the same example, your `prod` stage has the endpoint:

```
https://abc12345.execute-api.us-east-1.amazonaws.com/prod
```

To create the `dev` stage, you create a new API Gateway project and add the `dev` stage to the new project. The new endpoint will look something like:

```
https://xyz67890.execute-api.us-east-1.amazonaws.com/dev
```

Note that the `dev` stage carries a different endpoint host since it belongs to a different project. This is the approach Serverless Framework takes when configuring stages for your Serverless project. We will look at this in detail below.

- Separate AWS account for each stage

Just like how having each stage being separate APIs give us more flexibility to fine tune the IAM policy. We can take it a step further and create the API project in a different AWS account. Most companies don't keep their production infrastructure in the same account as their development infrastructure. This helps reduce any cases where developers accidentally edit/delete production resources. We go in to more detail on how to deploy to multiple AWS accounts using different AWS profiles in the [Configure Multiple AWS Profiles](#) (/chapters/configure-multiple-aws-profiles.html) chapter.

Deploying to a Stage

Let's look at how the Serverless Framework helps us work with stages. As mentioned above, a new stage is a new API Gateway project. To deploy to a specific stage, you can either specify the stage in the `serverless.yml`.

```
service: service-name

provider:
  name: aws
  stage: dev
```

Or you can specify the stage by passing the `--stage` option to the `serverless deploy` command.

```
$ serverless deploy --stage dev
```

Stage Variables in Serverless Framework

Deploying to stages can be pretty simple but now let's look at how to configure our environment variables so that they work with our various stages. We went over the concept of environment variables in the chapter on Serverless Environment Variables (/chapters/serverless-environment-variables.html). Let's extend that to specify variables based on the stage we are deploying to.

Let's take a look at a sample `serverless.yml` below.

```
service: service-name

custom:
  myStage: ${opt:stage, self:provider.stage}
  myEnvironment:
    MESSAGE:
      prod: "This is production environment"
      dev: "This is development environment"

provider:
  name: aws
  stage: dev
  environment:
    MESSAGE:
      ${self:custom.myEnvironment.MESSAGE.${self:custom.myStage}}
```

There are a couple of things happening here. We first defined the `custom.myStage` variable as `${opt:stage, self:provider.stage}`. This is telling Serverless Framework to use the `--stage` CLI option if it exists. And if it does not, then use the default stage specified by `provider.stage`. We also define the `custom.myEnvironment` section. This contains the value for `MESSAGE` defined for each stage. Finally, we set the environment variable `MESSAGE` as `${self:custom.myEnvironment.MESSAGE.${self:custom.myStage}}` . This sets the variable to pick the value of `self.custom.myEnvironment` depending on the current stage defined in `custom.myStage`.

You can easily extend this format to create separate sets of environment variables for the stages you are deploying to.

And we can access the `MESSAGE` in our Lambda functions via `process.env` object like so.

```
export function main(event, context, callback) {  
  callback(null, { body: process.env.MESSAGE } );  
}
```

Hopefully, this chapter gives you a quick idea on how to set up stages in your Serverless project.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/149>)

Configure Multiple AWS Profiles

When we configured our AWS CLI in the [Configure the AWS CLI](#) (/chapters/configure-the-aws-cli.html) chapter, we used the `aws configure` command to set the IAM credentials of the AWS account we wanted to use to deploy our serverless application to.

These credentials are stored in `~/.aws/credentials` and are used by the Serverless Framework when we run `serverless deploy`. Behind the scenes Serverless uses these credentials and the AWS SDK to create the necessary resources on your behalf to the AWS account specified in the credentials.

There are cases where you might have multiple credentials configured in your AWS CLI. This usually happens if you are working on multiple projects or if you want to separate the different stages of the same project.

In this chapter let's take a look at how you can work with multiple AWS credentials.

Create a New AWS Profile

Let's say you want to create a new AWS profile to work with. Follow the steps outlined in the [Create an IAM User](#) (/chapters/create-an-iam-user.html) chapter to create an IAM user in another AWS account and take a note of the **Access key ID** and **Secret access key**.

To configure the new profile in your AWS CLI use:

```
$ aws configure --profile newAccount
```

Where `newAccount` is the name of the new profile you are creating. You can leave the **Default region name** and **Default output format** the way they are.

Set a Profile on Local

We mentioned how the Serverless Framework uses your AWS profile to deploy your resources on your behalf. But while developing on your local using the `serverless invoke local`

command things are a little different.

In this case your Lambda function is run locally and has not been deployed yet. So any calls made in your Lambda function to any other AWS resources on your account will use the default AWS profile that you have. You can check your default AWS profile in `~/.aws/credentials` under the `[default]` tag.

To switch the default AWS profile to a new profile for the `serverless invoke local` command, you can run the following:

```
$ AWS_PROFILE=newAccount serverless invoke local --function hello
```

Here `newAccount` is the name of the profile you want to switch to and `hello` is the name of the function that is being invoked locally. By adding `AWS_PROFILE=newAccount` at the beginning of our `serverless invoke local` command we are setting the variable that the AWS SDK will use to figure out what your default AWS profile is.

If you want to set this so that you don't add it to each of your commands, you can use the following command:

```
$ export AWS_PROFILE=newAccount
```

Where `newAccount` is the profile you want to switch to. Now for the rest of your shell session, `newAccount` will be your default profile.

You can read more about this in the AWS Docs here (<http://docs.aws.amazon.com/cli/latest/userguide/cli-multiple-profiles.html>).

Set a Profile While Deploying

Now if we want to deploy using this newly created profile we can use the `--aws-profile` option for the `serverless deploy` command.

```
$ serverless deploy --aws-profile newAccount
```

Again, `newAccount` is the AWS profile Serverless Framework will be using to deploy.

If you don't want to set the profile every time you run `serverless deploy`, you can add it to

your `serverless.yml`.

```
service: service-name

provider:
  name: aws
  stage: dev
  profile: newAccount
```

Note the `profile: newAccount` line here. This is telling Serverless to use the `newAccount` profile while running `serverless deploy`.

Set Profiles per Stage

There are cases where you would like to specify a different AWS profile per stage. A common scenario for this is when you have a completely separate staging environment than your production one. Each environment has its own API endpoint, database tables, and more importantly, the IAM policies to secure the environment. A simple yet effective way to achieve this is to keep the environments in separate AWS accounts. AWS Organizations (<https://aws.amazon.com/organizations/>) was in fact introduced to help teams to create and manage these accounts and consolidate the usage charges into a single bill.

Let's look at a quick example of how to work with multiple profiles per stage. So following the examples from before, if you wanted to deploy to your production environment, you would:

```
$ serverless deploy --stage prod --aws-profile prodAccount
```

And to deploy to the staging environment you would:

```
$ serverless deploy --stage dev --aws-profile devAccount
```

Here, `prodAccount` and `devAccount` are the AWS profiles for the production and staging environment respectively.

To simplify this process you can add the profiles to your `serverless.yml`. So you don't have to specify them in your `serverless deploy` commands.

```

service: service-name

custom:
  myStage: ${opt:stage, self:provider.stage}
  myProfile:
    prod: prodAccount
    dev: devAccount

provider:
  name: aws
  stage: dev
  profile: ${self:custom.myProfile.${self:custom.myStage}}

```

There are a couple of things happening here.

- We first defined `custom.myStage` as `${opt:stage, self:provider.stage}`. This is telling Serverless Framework to use the value from the `--stage` CLI option if it exists. If not, use the default stage specified in `provider.stage`.
- We also defined `custom.myProfile`, which contains the AWS profiles we want to use to deploy for each stage. Just as before we want to use the `prodAccount` profile if we are deploying to stage `prod` and the `devAccount` profile if we are deploying to stage `dev`.
- Finally, we set the `provider.profile` to `${self:custom.myProfile.${self:custom.myStage}}`. This picks the value of our profile depending on the current stage defined in `custom.myStage`.

We used the concept of variables in Serverless Framework in this example. You can read more about this in the chapter on Serverless Environment Variables ([/chapters/serverless-environment-variables.html](#)).

Now, when you deploy to production, Serverless Framework is going to use the `prodAccount` profile. And the resources will be provisioned inside `prodAccount` profile user's AWS account.

```
$ serverless deploy --stage prod
```

And when you deploy to staging, the exact same set of AWS resources will be provisioned inside `devAccount` profile user's AWS account.

```
$ serverless deploy --stage dev
```

Notice that we did not have to set the `--aws-profile` option. And that's it, this should give you a good understanding of how to work with multiple AWS profiles and credentials.

For help and discussion

💬 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/150>)

Customize the Serverless IAM Policy

Serverless Framework deploys using the policy attached to the IAM credentials in your AWS CLI profile. Back in the Create an IAM User (/chapters/create-an-iam-user.html) chapter we created a user that the Serverless Framework will use to deploy our project. This user was assigned **AdministratorAccess**. This means that Serverless Framework and your project has complete access to your AWS account. This is fine in trusted environments but if you are working as a part of a team you might want to fine-tune the level of access based on who is using your project.

In this chapter we will take a look at how to customize the IAM Policy that Serverless Framework is going to use.

The permissions required can be categorized into the following areas:

- Permissions required by Serverless Framework
- Permissions required by your Serverless Framework plugins
- Permissions required by your Lambda code

Granting **AdministratorAccess** policy ensures that your project will always have the necessary permissions. But if you want to create an IAM policy that grants the minimal set of permissions, you need to customize your IAM policy.

A basic Serverless project needs permissions to the following AWS services:

- **CloudFormation** to create change set and update stack
- **S3** to upload and store Serverless artifacts and Lambda source code
- **CloudWatch Logs** to store Lambda execution logs
- **IAM** to manage policies for the Lambda IAM Role
- **API Gateway** to manage API endpoints
- **Lambda** to manage Lambda functions
- **EC2** to execute Lambda in VPC
- **CloudWatch Events** to manage CloudWatch event triggers

A simple IAM Policy template

These can be defined and granted using a simple IAM policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "cloudformation:*",  
                "s3:*",  
                "logs:*",  
                "iam:*",  
                "apigateway:*",  
                "lambda:*",  
                "ec2:DescribeSecurityGroups",  
                "ec2:DescribeSubnets",  
                "ec2:DescribeVpcs",  
                "events:*"  
            ],  
            "Resource": [  
                "*"  
            ]  
        }  
    ]  
}
```

We can attach this policy to the IAM user we are creating by continuing from the **Attach existing policies directly** step in the Create an IAM User ([/chapters/create-an-iam-user.html](#)) chapter.

Hit the **Create policy** button.

Add user

1 Details 2 Permissions 3 Review 4 Complete

Set permissions for new-user

1. Add user to group
2. Copy permissions from existing user
3. Attach existing policies directly

Attach one or more existing policies directly to the users or create a new policy. [Learn more](#)

Create policy Refresh

Filter: Policy type ▾ Search Showing 299 results

	Policy name ▾	Type	Attachments ▾	Description
<input type="checkbox"/>	AdministratorAccess	Job function	2	Provides full access to AWS services and resources.
<input type="checkbox"/>	AmazonAPIGatewayAdministrator	AWS managed	0	Provides full access to create/edit/delete APIs in Amazon API Gateway via ...

Feedback English (US) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

And hit **Select** in the **Create Your Own Policy** section.

Create Policy

Step 1 : Create Policy
Step 2 : Set Permissions
Step 3 : Review Policy

Create Policy

A policy is a document that formally states one or more permissions. Create a policy by copying an AWS Managed Policy, using the Policy Generator, or typing your own custom policy.

Copy an AWS Managed Policy

Start with an AWS Managed Policy, then customize it to fit your needs. [Select](#)

Policy Generator

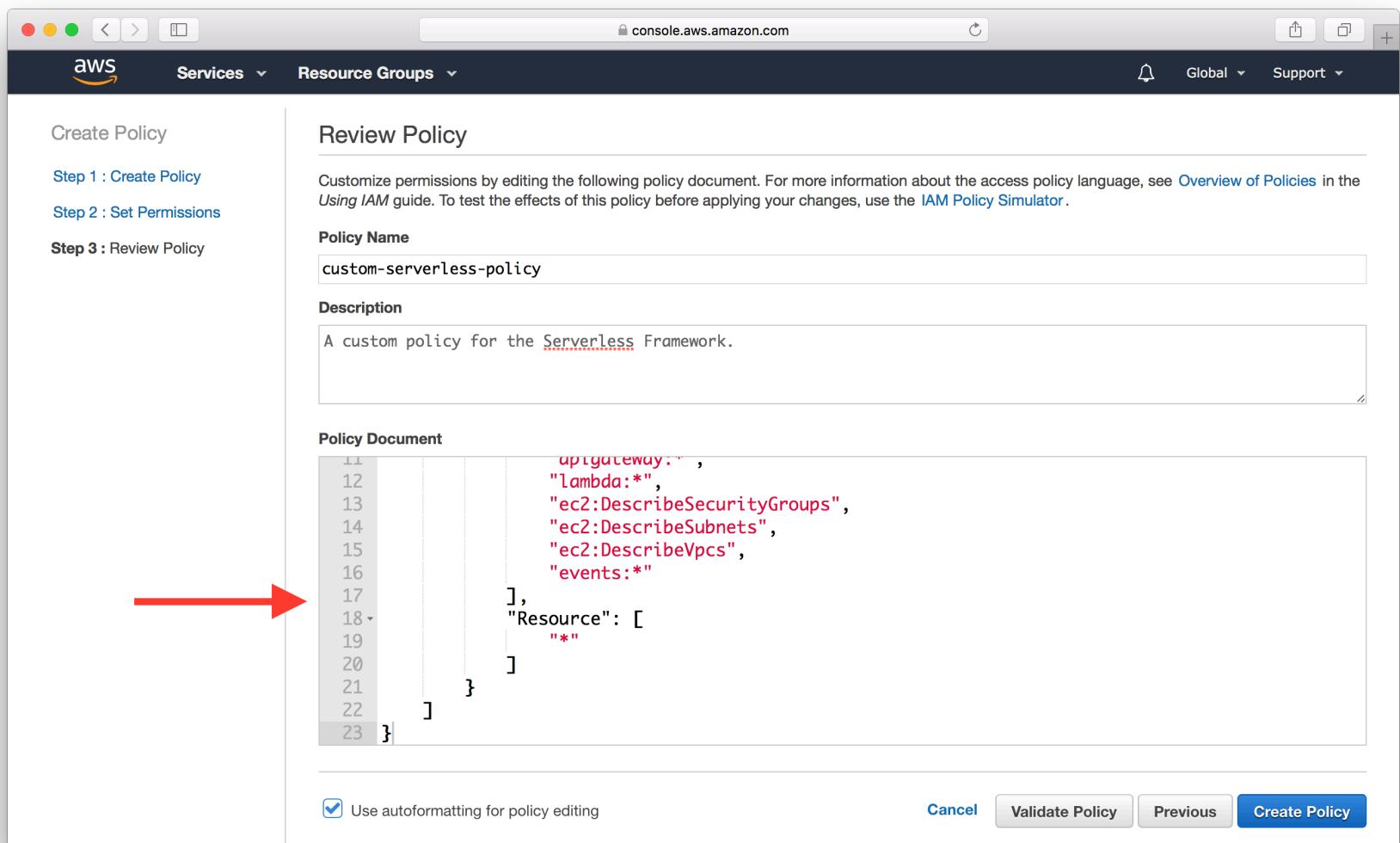
Use the policy generator to select services and actions from a list. The policy generator uses your selections to create a policy. [Select](#)

Create Your Own Policy

Use the policy editor to type or paste in your own policy. [Select](#)

Cancel

Here fill pick a name for your new policy and paste the policy created above in the **Policy Document** field.



Finally, hit **Create Policy**. You can now chose this policy while creating your IAM user instead of the **AdministratorAccess** one that we had used before.

This policy grants your Serverless Framework project access to all the resources listed above. But we can narrow this down further by restricting them to specific **Actions** for the specific **Resources** in each AWS service.

An advanced IAM Policy template

Below is a more nuanced policy template that restricts access to the Serverless project that is being deployed. Make sure to replace `<region>`, `<account_no>` and `<service_name>` for your specific project.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
```

```
"Effect": "Allow",
"Action": [
    "cloudformation:Describe*",
    "cloudformation>List*",
    "cloudformation:Get*",
    "cloudformation:PreviewStackUpdate",
    "cloudformation>CreateStack",
    "cloudformation:UpdateStack",
    "cloudformation>DeleteStack"
],
"Resource": "arn:aws:cloudformation:<region>:<account_no>:stack/<service_name>*//*"
},
{
    "Effect": "Allow",
    "Action": [
        "cloudformation:ValidateTemplate"
    ],
    "Resource": "*"
},
{
    "Effect": "Allow",
    "Action": [
        "s3>CreateBucket",
        "s3>DeleteBucket",
        "s3:Get*",
        "s3>List*"
    ],
    "Resource": [
        "arn:aws:s3:::/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "s3:/*"
    ],
    "Resource": [

```

```
"arn:aws:s3:::*/*"
]

} ,
{

  "Effect": "Allow",
  "Action": [
    "logs:DescribeLogGroups"
  ],
  "Resource": "arn:aws:logs:<region>:<account_no>:log-group::log-
stream:/*"
},
{
  "Action": [
    "logs>CreateLogGroup",
    "logs>CreateLogStream",
    "logs>DeleteLogGroup",
    "logs>DeleteLogStream",
    "logs>DescribeLogStreams",
    "logs>FilterLogEvents"
  ],
  "Resource": "arn:aws:logs:<region>:<account_no>:log-
group:/aws/lambda/<service_name>*:log-stream:/*",
  "Effect": "Allow"
},
{
  "Effect": "Allow",
  "Action": [
    "iam:GetRole",
    "iam:PassRole",
    "iam>CreateRole",
    "iam>DeleteRole",
    "iam:DetachRolePolicy",
    "iam:PutRolePolicy",
    "iam:AttachRolePolicy",
    "iam>DeleteRolePolicy"
  ],
  "Resource": [
    "arn:aws:iam::<account_no>:role/<service_name>*-lambdaRole"
  ]
}
```

```
        ],
    },
{
    "Effect": "Allow",
    "Action": [
        "apigateway:GET",
        "apigateway:POST",
        "apigateway:PUT",
        "apigateway:DELETE"
    ],
    "Resource": [
        "arn:aws:apigateway:<region>:::/restapis"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "apigateway:GET",
        "apigateway:POST",
        "apigateway:PUT",
        "apigateway:DELETE"
    ],
    "Resource": [
        "arn:aws:apigateway:<region>:::/restapis/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "lambda:GetFunction",
        "lambda>CreateFunction",
        "lambda>DeleteFunction",
        "lambda:UpdateFunctionConfiguration",
        "lambda:UpdateFunctionCode",
        "lambda>ListVersionsByFunction",
        "lambda>PublishVersion",
        "lambda>CreateAlias",
        "lambda>DeleteAlias"
    ]
}
```

```

    "lambda:UpdateAlias",
    "lambda:GetFunctionConfiguration",
    "lambda:AddPermission",
    "lambda:RemovePermission",
    "lambda:InvokeFunction"
],
"Resource": [
    "arn:aws:lambda:*:<account_no>:function:<service_name>*"
]
},
{
    "Effect": "Allow",
    "Action": [
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs"
    ],
    "Resource": [
        "*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "events:Put*",
        "events:Remove*",
        "events:Delete*",
        "events:Describe*"
    ],
    "Resource": "arn:aws:events::<account_no>:rule/<service_name>*"
}
]
}

```

The `<account_no>` is your AWS Account ID and you can follow these instructions (http://docs.aws.amazon.com/IAM/latest/UserGuide/console_account-alias.html) to look it up.

Also, recall that the `<region>` and `<service_name>` are defined in your

`serverless.yml` like so.

```
service: my-service

provider:
  name: aws
  region: us-east-1
```

In the above `serverless.yml`, the `<region>` is `us-east-1` and the `<service_name>` is `my-service`.

The above IAM policy template restricts access to the AWS services based on the name of your Serverless project and the region it is deployed in.

It provides sufficient permissions for a minimal Serverless project. However, if you provision any additional resources in your `serverless.yml`, or install Serverless plugins, or invoke any AWS APIs in your application code; you would need to update the IAM policy to accommodate for those changes. If you are looking for details on where this policy comes from; here is an in-depth discussion on the minimal Serverless IAM Deployment Policy (<https://github.com/serverless/serverless/issues/1439>) required for a Serverless project.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/167>)

Code Splitting in Create React App

Code Splitting is not a necessary step for building React apps. But feel free to follow along if you are curious about what Code Splitting is and how it can help larger React apps.

Code Splitting

While working on React.js single page apps, there is a tendency for apps to grow quite large. A section of the app (or route) might import a large number of components that are not necessary when it first loads. This hurts the initial load time of our app.

You might have noticed that Create React App will generate one large `.js` file while we are building our app. This contains all the JavaScript our app needs. But if a user is simply loading the login page to sign in; it doesn't make sense that we load the rest of the app with it. This isn't a concern early on when our app is quite small but it becomes an issue down the road. To address this, Create React App has a very simple built-in way to split up our code. This feature unsurprisingly, is called Code Splitting.

Create React App (from 1.0 onwards) allows us to dynamically import parts of our app using the `import()` proposal. You can read more about it here (<https://facebook.github.io/react/blog/2017/05/18/whats-new-in-create-react-app.html#code-splitting-with-dynamic-import>).

While, the dynamic `import()` can be used for any component in our React app; it works really well with React Router. Since, React Router is figuring out which component to load based on the path; it would make sense that we dynamically import those components only when we navigate to them.

Code Splitting and React Router v4

The usual structure used by React Router to set up routing for your app looks something like this.

```
/* Import the components */
```

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

/* Use components to define routes */
export default () =>
<Switch>
  <Route path="/" exact component={Home} />
  <Route path="/posts/:id" exact component={Posts} />
  <Route component={NotFound} />
</Switch>;
```

We start by importing the components that will respond to our routes. And then use them to define our routes. The `Switch` component renders the route that matches the path.

However, we import all of the components in the route statically at the top. This means, that all these components are loaded regardless of which route is matched. To implement Code Splitting here we are going to want to only load the component that responds to the matched route.

Create an Async Component

To do this we are going to dynamically import the required component.

◆ CHANGE Add the following to `src/components/AsyncComponent.js`.

```
import React, { Component } from "react";

export default function asyncComponent(importComponent) {
  class AsyncComponent extends Component {
    constructor(props) {
      super(props);

      this.state = {
        component: null
      };
    }
  }
}
```

```

async componentDidMount() {
  const { default: component } = await importComponent();

  this.setState({
    component: component
  });
}

render() {
  const C = this.state.component;

  return C ? <C {...this.props} /> : null;
}
}

return AsyncComponent;
}

```

We are doing a few things here:

1. The `asyncComponent` function takes an argument; a function (`importComponent`) that when called will dynamically import a given component. This will make more sense below when we use `asyncComponent`.
2. On `componentDidMount`, we simply call the `importComponent` function that is passed in. And save the dynamically loaded component in the state.
3. Finally, we conditionally render the component if it has completed loading. If not we simply render `null`. But instead of rendering `null`, you could render a loading spinner. This would give the user some feedback while a part of your app is still loading.

Use the Async Component

Now let's use this component in our routes. Instead of statically importing our component.

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

We are going to use the `asyncComponent` to dynamically import the component we want.

```
const AsyncHome = asyncComponent(() => import("./containers/Home"));
```

It's important to note that we are not doing an import here. We are only passing in a function to `asyncComponent` that will dynamically `import()` when the `AsyncHome` component is created.

Also, it might seem weird that we are passing a function here. Why not just pass in a string (say `./containers/Home`) and then do the dynamic `import()` inside the `AsyncComponent`? This is because we want to explicitly state the component we are dynamically importing. Webpack splits our app based on this. It looks at these imports and generates the required parts (or chunks). This was pointed out by @wSokra (<https://twitter.com/wSokra/status/866703557323632640>) and @dan_abramov (https://twitter.com/dan_abramov/status/866646657437491201).

We are then going to use the `AsyncHome` component in our routes. React Router will create the `AsyncHome` component when the route is matched and that will in turn dynamically import the `Home` component and continue just like before.

```
<Route path="/" exact component={AsyncHome} />
```

Now let's go back to our Notes project and apply these changes.

👉 **CHANGE** Your `src/Routes.js` should look like this after the changes.

```
import React from "react";
import { Route, Switch } from "react-router-dom";
import asyncComponent from "./components/AsyncComponent";
import AppliedRoute from "./components/AppliedRoute";
import AuthenticatedRoute from "./components/AuthenticatedRoute";
import UnauthenticatedRoute from "./components/UnauthenticatedRoute";

const AsyncHome = asyncComponent(() => import("./containers/Home"));
const AsyncLogin = asyncComponent(() => import("./containers/Login"));
const AsyncNotes = asyncComponent(() => import("./containers/Notes"));
const AsyncSignup = asyncComponent(() =>
  import("./containers/Signup"));
const AsyncNewNote = asyncComponent(() =>
```

```
import("./containers/NewNote"));
const NotFound = asyncComponent(() =>
import("./containers/NotFound"));

export default ({ childProps }) =>
<Switch>
  <AppliedRoute
    path="/"
    exact
    component={AsyncHome}
    props={childProps}
  />
  <UnauthenticatedRoute
    path="/login"
    exact
    component={AsyncLogin}
    props={childProps}
  />
  <UnauthenticatedRoute
    path="/signup"
    exact
    component={AsyncSignup}
    props={childProps}
  />
  <AuthenticatedRoute
    path="/notes/new"
    exact
    component={AsyncNewNote}
    props={childProps}
  />
  <AuthenticatedRoute
    path="/notes/:id"
    exact
    component={AsyncNotes}
    props={childProps}
  />
  {/* Finally, catch all unmatched routes */}
  <Route component={AsyncNotFound} />
```

```
</Switch>
```

```
;
```

It is pretty cool that with just a couple of changes, our app is all set up for code splitting. And without adding a whole lot more complexity either! Here is what our `src/Routes.js` looked like before.

```
import React from "react";
import { Route, Switch } from "react-router-dom";
import AppliedRoute from "./components/AppliedRoute";
import AuthenticatedRoute from "./components/AuthenticatedRoute";
import UnauthenticatedRoute from "./components/UnauthenticatedRoute";

import Home from "./containers/Home";
import Login from "./containers/Login";
import Notes from "./containers/Notes";
import Signup from "./containers/Signup";
import NewNote from "./containers/NewNote";
import NotFound from "./containers/NotFound";

export default ({ childProps }) =>
  <Switch>
    <AppliedRoute
      path="/"
      exact
      component={Home}
      props={childProps}
    />
    <UnauthenticatedRoute
      path="/login"
      exact
      component={Login}
      props={childProps}
    />
    <UnauthenticatedRoute
      path="/signup"
      exact
      component={Signup}
    />
```

```
    props={childProps}  
  />  
  
<AuthenticatedRoute  
  path="/notes/new"  
  exact  
  component={NewNote}  
  props={childProps}  
/>  
  
<AuthenticatedRoute  
  path="/notes/:id"  
  exact  
  component={Notes}  
  props={childProps}  
/>  
{/* Finally, catch all unmatched routes */}  
<Route component={NotFound} />  
</Switch>  
;
```

Notice that instead of doing the static imports for all the containers at the top, we are creating these functions that are going to do the dynamic imports for us when necessary.

Now if you build your app using `npm run build`; you'll see the code splitting in action.

```
ServerlessStackDemoClient — ServerlessStackDemoClient — -bash — 90x25
```

```
> react-scripts build
```

```
Creating an optimized production build...
Compiled successfully.
```

```
File sizes after gzip:
```

314.66 KB	build/static/js/main.92eaa171.js
2.98 KB	build/static/js/0.501c9213.chunk.js
2.67 KB	build/static/js/2.69156576.chunk.js
2.56 KB	build/static/js/1.424b6dfc.chunk.js
2.46 KB	build/static/js/3.919af17b.chunk.js
2.02 KB	build/static/js/4.e243d17a.chunk.js
547 B	build/static/js/5.de81e9a1.chunk.js
374 B	build/static/css/main.dc12301d.css

```
The project was built assuming it is hosted at the server root.
To override this, specify the homepage in your package.json.
For example, add this to build it for GitHub Pages:
```

```
"homepage" : "http://myname.github.io/myapp",
```

```
The build folder is ready to be deployed.
You may serve it with a static server:
```

Each of those `.chunk.js` files are the different dynamic `import()` calls that we have. Of course, our app is quite small and the various parts that are split up are not significant at all. However, if the page that we use to edit our note included a rich text editor; you can imagine how that would grow in size. And it would unfortunately affect the initial load time of our app.

Now if we deploy our app using `npm run deploy`; you can see the browser load the different chunks on-demand as we browse around in the demo (<https://demo.serverless-stack.com>).

Name	Domain	Type	Met...	Scheme	Status	Cached	Size	Transferred	St...	Late...	Duration
demo.serverless-st...	demo.serverless-st...	Doc...	GET	HTTPS	200	No	943 B	1.31 KB	0.03...	4.24s	15.83ms
css	fonts.googleapis.c...	Styl...	GET	HTTPS	200	No	12.85 KB	13.23 KB	4.26s	137....	6.433ms
bootstrap.min.css	maxcdn.bootstrapcdn.c...	Styl...	GET	HTTPS	304	Yes	118.42...	380 B	4.26s	154....	104.0ms
main.dc12301d.css	demo.serverless-st...	Styl...	GET	HTTPS	304	Yes	800 B	306 B	4.26s	140....	3.036ms
main.92eaaf71.js	demo.serverless-st...	Script	GET	HTTPS	304	Yes	1.72 MB	287 B	4.26s	129....	11.86ms
2.69156576.chunk.js	demo.serverless-st...	Script	GET	HTTPS	304	Yes	6.79 KB	287 B	4.51s	193....	7.472ms
blob:https://demo.serverless-stack.co...	—	Styl...	GET	—	200	No	1.75 KB	1.81 KB	4.71s	3.10...	0.094ms
3.919af17b.chunk.js	demo.serverless-st...	Script	GET	HTTPS	304	Yes	6.67 KB	287 B	8.16s	6.70s	36.83ms
blob:https://demo.serverless-stack.co...	—	Styl...	GET	—	200	No	941 B	1003 B	14.8...	33.1...	0.086ms
4.e243d17a.chunk.js	demo.serverless-st...	Script	GET	HTTPS	304	Yes	4.44 KB	287 B	20.1...	13.4...	20.58ms
blob:https://demo.serverless-stack.co...	—	Styl...	GET	—	200	No	668 B	730 B	33.5...	17.3...	0.161ms

That's it! With just a few simple changes our app is completely set up to use the code splitting feature that Create React App has.

Next Steps

Now this seems really easy to implement but you might be wondering what happens if the request to import the new component takes too long, or fails. Or maybe you want to preload certain components. For example, a user is on your login page about to login and you want to preload the homepage.

It was mentioned above that you can add a loading spinner while the import is in progress. But we can take it a step further and address some of these edge cases. There is an excellent higher order component that does a lot of this well; it's called **react-loadable** (<https://github.com/thejameskyle/react-loadable>).

All you need to do to use it is install it.

```
$ npm install --save react-loadable
```

Use it instead of the `asyncComponent` that we had above.

```
const AsyncHome = Loadable({
  loader: () => import("./containers/Home"),
  loading: MyLoadingComponent
});
```

And `AsyncHome` is used exactly as before. Here the `MyLoadingComponent` would look something like this.

```
const MyLoadingComponent = ({isLoading, error}) => {
  // Handle the loading state
  if (isLoading) {
    return <div>Loading...</div>;
  }
  // Handle the error state
  else if (error) {
    return <div>Sorry, there was a problem loading the page.</div>;
  }
  else {
    return null;
  }
};
```

It's a simple component that handles all the different edge cases gracefully.

To add preloading and to further customize this; make sure to check out the other options and features that react-loadable (<https://github.com/thejameskyle/react-loadable>) has. And have fun code splitting!

For help and discussion

💬 [Comments on this chapter](#)

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/90>)

For reference, here is the code so far

⌚ [Frontend Source :code-splitting-in-create-react-app](#)

(<https://github.com/AnomalyInnovations/serverless-stack-demo-client/tree/code-splitting-in-create-react-app>)

Environments in Create React App

While developing your frontend React app and working with an API backend, you'll often need to create multiple environments to work with. For example, you might have an environment called dev that might be connected to the dev stage of your serverless backend. This is to ensure that you are working in an environment that is isolated from your production version.

Aside from isolating the resources used, having a separate environment that mimics your production version can really help with testing your changes before they go live. You can take this idea of environments further by having a staging environment that can even have snapshots of the live database to give you as close to a production setup as possible. This type of setup can sometimes help track down bugs and issues that you might run into only on our live environment and not on local.

In this chapter we will look at some simple ways to configure multiple environments in our React app. There are many different ways to do this but here is a simple one based on what we have built so far in this guide.

Custom Environment Variables

Create React App (<https://github.com/facebookincubator/create-react-app/blob/master/packages/react-scripts/template/README.md#adding-custom-environment-variables>) has support for custom environment variables baked into the build system. To set a custom environment variable, simply set it while starting the Create React App build process.

```
$ REACT_APP_TEST_VAR=123 npm start
```

Here `REACT_APP_TEST_VAR` is the custom environment variable and we are setting it to the value `123`. In our app we can access this variable as `process.env.REACT_APP_TEST_VAR`. So the following line in our app:

```
console.log(process.env.REACT_APP_TEST_VAR);
```

Will print out `123` in our console.

Note that, these variables are embedded during build time. Also, only the variables that start with `REACT_APP_` are embedded in our app. All the other environment variables are ignored.

Configuring Environments

We can use this idea of custom environment variables to configure our React app for specific environments. Say we used a custom environment variable called `REACT_APP_STAGE` to denote the environment our app is in. And we wanted to configure two environments for our app:

- One that we will use for our local development and also to test before pushing it to live. Let's call this one `dev`.
- And our live environment that we will only push to, once we are comfortable with our changes. Let's call it `production`.

The first thing we can do is to configure our build system with the `REACT_APP_STAGE` environment variable. Currently the `scripts` portion of our `package.json` looks something like this:

```
"scripts": {  
  "start": "react-scripts start",  
  "build": "react-scripts build",  
  "test": "react-scripts test --env=jsdom",  
  "predeploy": "npm run build",  
  "deploy": "aws s3 sync build/ s3://YOUR_S3_DEPLOY_BUCKET_NAME",  
  "postdeploy": "aws cloudfront create-invalidation --distribution-id  
  YOUR_CF_DISTRIBUTION_ID --paths '/*' && aws cloudfront create-  
  invalidation --distribution-id YOUR_WWW_CF_DISTRIBUTION_ID --paths  
  '/*'",  
  "eject": "react-scripts eject"  
}
```

Recall that the `YOUR_S3_DEPLOY_BUCKET_NAME` is the S3 bucket we created to host our React app back in the Create an S3 bucket (/chapters/create-an-s3-bucket.html) chapter. And `YOUR_CF_DISTRIBUTION_ID` and `YOUR_WWW_CF_DISTRIBUTION_ID` are the CloudFront

Distributions for the apex (/chapters/create-a-cloudfront-distribution.html) and www (/chapters/setup-www-domain-redirect.html) domains.

Here we only have one environment and we use it for our local development and on live. The `npm start` command runs our local server and `npm run deploy` command deploys our app to live.

To set our two environments we can change this to:

```
"scripts": {  
  "start": "REACT_APP_STAGE=dev react-scripts start",  
  "build": "react-scripts build",  
  "test": "react-scripts test --env=jsdom",  
  
  "predeploy": "REACT_APP_STAGE=dev npm run build",  
  "deploy": "aws s3 sync build/ s3://YOUR_DEV_S3_DEPLOY_BUCKET_NAME",  
  "postdeploy": "aws cloudfront create-invalidation --distribution-id  
YOUR_DEV_CF_DISTRIBUTION_ID --paths '/*' && aws cloudfront create-  
invalidation --distribution-id YOUR_DEV_WWW_CF_DISTRIBUTION_ID --paths  
'/*'",  
  
  "predeploy:prod": "REACT_APP_STAGE=production npm run build",  
  "deploy:prod": "aws s3 sync build/  
s3://YOUR_PROD_S3_DEPLOY_BUCKET_NAME",  

```

We are doing a few things of note here:

1. We use the `REACT_APP_STAGE=dev` for our `npm start` command.
2. We also have dev versions of our S3 and CloudFront Distributions called

`YOUR_DEV_S3_DEPLOY_BUCKET_NAME`, `YOUR_DEV_CF_DISTRIBUTION_ID`, and
`YOUR_DEV_WWW_CF_DISTRIBUTION_ID`.

3. We default `npm run deploy` to the dev environment and dev versions of our S3 and CloudFront Distributions. We also build using the `REACT_APP_STAGE=dev` environment variable.
4. We have production versions of our S3 and CloudFront Distributions called `YOUR_PROD_S3_DEPLOY_BUCKET_NAME`, `YOUR_PROD_CF_DISTRIBUTION_ID`, and `YOUR_PROD_WWW_CF_DISTRIBUTION_ID`.
5. Finally, we create a specific version of the deploy script for the production environment with `npm run deploy:prod`. And just like the dev version of this command, it builds using the `REACT_APP_STAGE=production` environment variable and the production versions of the S3 and CloudFront Distributions.

Note that you don't have to replicate the S3 and CloudFront Distributions for the dev version. But it does help if you want to mimic the live version as much as possible.

Using Environment Variables

Now that we have our build commands set up with the custom environment variables, we are ready to use them in our app.

Currently, our `src/config.js` looks something like this:

```
export default {  
  MAX_ATTACHMENT_SIZE: 500000,  
  s3: {  
    BUCKET: "YOUR_S3_UPLOADS_BUCKET_NAME"  
  },  
  apiGateway: {  
    REGION: "YOUR_API_GATEWAY_REGION",  
    URL: "YOUR_API_GATEWAY_URL"  
  },  
  cognito: {  
    REGION: "YOUR_COGNITO_REGION",  
    USER_POOL_ID: "YOUR_COGNITO_USER_POOL_ID",  
    APP_CLIENT_ID: "YOUR_COGNITO_APP_CLIENT_ID",  
    IDENTITY_POOL_ID: "YOUR_IDENTITY_POOL_ID"  
  }  
};
```

To use the `REACT_APP_STAGE` variable, we are just going to set the config conditionally.

```
const dev = {
  s3: {
    BUCKET: "YOUR_DEV_S3_UPLOADS_BUCKET_NAME"
  },
  apiGateway: {
    REGION: "YOUR_DEV_API_GATEWAY_REGION",
    URL: "YOUR_DEV_API_GATEWAY_URL"
  },
  cognito: {
    REGION: "YOUR_DEV_COGNITO_REGION",
    USER_POOL_ID: "YOUR_DEV_COGNITO_USER_POOL_ID",
    APP_CLIENT_ID: "YOUR_DEV_COGNITO_APP_CLIENT_ID",
    IDENTITY_POOL_ID: "YOUR_DEV_IDENTITY_POOL_ID"
  }
};

const prod = {
  s3: {
    BUCKET: "YOUR_PROD_S3_UPLOADS_BUCKET_NAME"
  },
  apiGateway: {
    REGION: "YOUR_PROD_API_GATEWAY_REGION",
    URL: "YOUR_PROD_API_GATEWAY_URL"
  },
  cognito: {
    REGION: "YOUR_PROD_COGNITO_REGION",
    USER_POOL_ID: "YOUR_PROD_COGNITO_USER_POOL_ID",
    APP_CLIENT_ID: "YOUR_PROD_COGNITO_APP_CLIENT_ID",
    IDENTITY_POOL_ID: "YOUR_PROD_IDENTITY_POOL_ID"
  }
};

const config = process.env.REACT_APP_STAGE === 'production'
? prod
: dev;
```

```
export default {  
  // Add common config values here  
  MAX_ATTACHMENT_SIZE: 5000000,  
  ...config  
};
```

This is pretty straightforward. We simply have a set of configs for dev and for production. The configs point to a separate set of resources for our dev and production environments. And using `process.env.REACT_APP_STAGE` we decide which one to use.

Again, it might not be necessary to replicate the resources for each of the environments. But it is pretty important to separate your live resources from your dev ones. You do not want to be testing your changes directly on your live database.

So to recap:

- The `REACT_APP_STAGE` custom environment variable is set to either `dev` or `production`.
- While working locally we use the `npm start` command which uses our dev environment.
- The `npm run deploy` command then deploys by default to dev.
- Once we are comfortable with the dev version, we can deploy to production using the `npm run deploy:prod` command.

This entire setup is fairly straightforward and can be extended to multiple environments. You can read more on custom environment variables in Create React App here (<https://github.com/facebookincubator/create-react-app/blob/master/packages/react-scripts/template/README.md#adding-custom-environment-variables>).

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/168>)

Serverless Node.js Starter

Now that we know how to set our Serverless projects up, it makes sense that we have a good starting point for our future projects. For this we created a couple of Serverless starter projects that you can use called, Serverless Node.js Starter

(<https://github.com/AnomalyInnovations/serverless-nodejs-starter>). We also have a Python version called Serverless Python Starter (<https://github.com/AnomalyInnovations/serverless-python-starter>). Our starter projects also work really well with Seed (<https://seed.run>); a fully-configured CI/CD pipeline for Serverless Framework.

Serverless Node.js Starter (<https://github.com/AnomalyInnovations/serverless-nodejs-starter>) uses the serverless-webpack (<https://github.com/serverless-heaven/serverless-webpack>) plugin, Babel (<https://babeljs.io>), and Jest (<https://facebook.github.io/jest/>). It supports:

- **Use async/await in your handler functions**
- **Support for unit tests**
 - Run `npm test` to run your tests
- **Sourcemaps for proper error messages**
 - Error message show the correct line numbers
 - Works in production with CloudWatch
- **Automatic support for multiple handler files**
 - No need to add a new entry to your `webpack.config.js`

Demo

A demo version of this service is hosted on AWS - <https://cvps1pt354.execute-api.us-east-1.amazonaws.com/dev/hello> (<https://cvps1pt354.execute-api.us-east-1.amazonaws.com/dev/hello>).

And here is the ES7 source behind it.

```
export const hello = async (event, context, callback) => {
  const response = {
    statusCode: 200,
```

```
body: JSON.stringify({
  message: `Go Serverless v1.0! ${await message({ time: 1, copy: 'Your function executed successfully!' })}`,
  input: event,
}),
};

callback(null, response);
};

const message = ({ time, ...rest }) => new Promise((resolve, reject) =>
  setTimeout(() => {
    resolve(`$${rest.copy} (with a delay)`);
  }, time * 1000)
);
```

Requirements

- Configure your AWS CLI (/chapters/configure-the-aws-cli.html)
- Install the Serverless Framework `npm install serverless -g`

Installation

To create a new Serverless project with ES7 support.

```
$ serverless install --url
https://github.com/AnomalyInnovations/serverless-nodejs-starter --name
my-project
```

Enter the new directory.

```
$ cd my-project
```

Install the Node.js packages.

```
$ npm install
```

Usage

To run a function on your local

```
$ serverless invoke local --function hello
```

Run your tests

```
$ npm test
```

We use Jest to run our tests. You can read more about setting up your tests here (<https://facebook.github.io/jest/docs/en/getting-started.html#content>).

Deploy your project

```
$ serverless deploy
```

Deploy a single function

```
$ serverless deploy function --function hello
```

So give it a try and send us an email (<mailto:contact@anoma.ly>) if you have any questions or open a new issue (<https://github.com/AnomalyInnovations/serverless-nodejs-starter/issues/new>) if you've found a bug.

For help and discussion

💬 Comments on this chapter

(<https://github.com/AnomalyInnovations/serverless-stack-com/issues/72>)