BUILDING SERVERLESS WEBAPPS

Using AWS Cognito, Lambda, Lambda@Edge, S3, DynamoDB and Cloud Front

AYMEN EL AMRI

Introduction and Requirements

Serverless web applications draw a lot of attention these days. They are simple and easy to develop and deploy. Serverless reduces the costs of infrastructure dramatically and give you a set of tools and an ecosystem to grow your applications. Many cloud providers offer Serverless runtimes and services.

In this course, we will dive deep into building a serverless web application on AWS. The web application is a prototype of serving private files to authenticated users.

Let's say you want to share a paid course (or any private document) with only a few users from your database. These users should authenticate to the application.

This is the file tree we are going to use:

```
.
├── private-1
│   ├── private-11.html
│   └── private-12.html
├── private-2
│   └── private-21.html
├── private-3
│   └── private-31.html
├── public
│   └── public-1.html
├── resources.html
├── signin.html
└── signup.html
```

The private folders are called `private-x` and the private files inside these private folders and are called `private-xx.html`.

We also have a `public` folder, accessible to all users.

The `resources.html` is accessible only when the user is authenticated and it contains the private files the same connected user can access to.

If a given user X has only access to `private-1` and `private-3` folders, they will not be able to see files in other protected folders.

The `signing.html` and `signup.html` pages are the forms used to manage users authentication.

Once a user is authenticated, we should check (from the database) which files they have access to; then you show it to them.

To build this, you will need a language like Python, PHP or Java to create the basics of the app, an authentication library (e.g. Allauth for Django, Spring Social for Java Spring or SocialConnect for PHP), a server to host the private files, cache server for the static files to make your user experience better, a reverse proxy if you host many web apps on the same server, an API gateway to expose your app if you wish and other components like a SQL database where you store information about users..etc.

In this course, you'll discover that you will optimize your time and resources and enhance your user experience using S3, CloudFront, Lambda, Cognito, and other AWS services.

We are not focusing on the application architecture here, so our goal is not to learn how to architect your data or how to optimize your code calls, but how to deploy and integrate a serverless application running on Lambda with other services such as S3, CloudFront and Lambda@Edge.

This course is not about theories and definitions. It's a practical step-by-step course to help you understand how to do things. If you are a doer or a maker, you will certainly like this approach.

Whenever you need to read more about a service, we recommend the AWS official documentation along with this course. This course is not documentation, as you may now understand.

In other words, we are going to demonstrate how the following items can be implemented on AWS :

- Setting up applications, files, source code, and databases.
- Onboarding users using Identity Providers such as Google authentication.
- Allowing users to register and create user accounts.
- Giving users access to some static files based on permissions
- Implementing the functionality of the web application (backend and frontend)
- Exposing the applications to the public and access it through the Internet.

To achieve the above points, we will be exploring and using some managed services from AWS, below is a brief description of these services and the use case for each of them:

- **AWS S3**: This service will be used to store the static assets (HTML, Images, and Javascript) of the web application.
- **AWS DynamoDB**: is a service that will be used to store the application data such as files a user has permissions to view. It's the AWS NoSQL database.
- **AWS Cognito**: is a service that helps in implementing singing, signup, and access control functionalities for web applications. It's the AWS alternative for Auth0.
- **AWS Lambda**: is a service that provides an easy interface to implement "function as a service" or FaaS applications. The application code will be implemented using this service. In this course, we will be building Lambda functions using NodeJs and Python. Therefore you need to make yourself familiar with the basics of these languages.
- **AWS API Gateway**: is a managed service that allows us easily deploy an API and expose it to the Internet. When you deploy a Serverless application, exposing it to the Internet via a public API is a good practice. Therefore, this service allows mapping API endpoints to the functions defined using AWS Lambda.
- **AWS CloudFront**: is a content delivery network offered by Amazon Web Services. This service helps in serving and cashing the static application files and assets (images/videos/scripts/etc..). This service uses S3 but replicate the static files stored on a bucket to global endpoints and servers around the world. This way, your app users will download and use these static files rapidly.

Before proceeding with the next lectures, please make sure that you already have a valid AWS account and can use the above services.

Learning by doing is one of the best ways to learn, so let's start working on building our serverless application.

This diagram describes the architecture of the application we are going to build in this course.

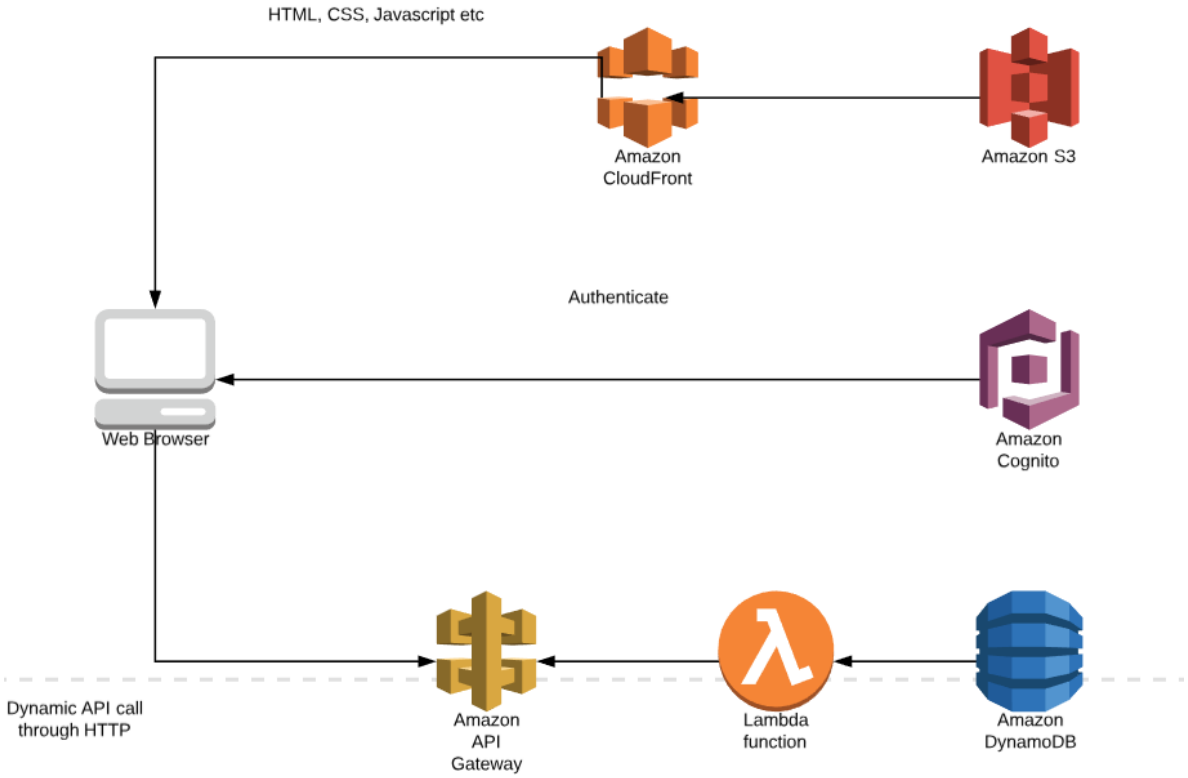
It is organized into two parts, namely the backend and frontend.

The backend part includes:

- Authentication with AWS Cognito

- A Restful API with AWS API Gateway, AWS Lambda, and AWS DynamoDB
- The website with AWS S3 and AWS CloudFront.

The frontend part includes all the HTML and JS files, which will be shown on the user web browser.



All of our resources will be located in one region. In my case, I will be using the FRANKFURT region, namely eu-central-1.

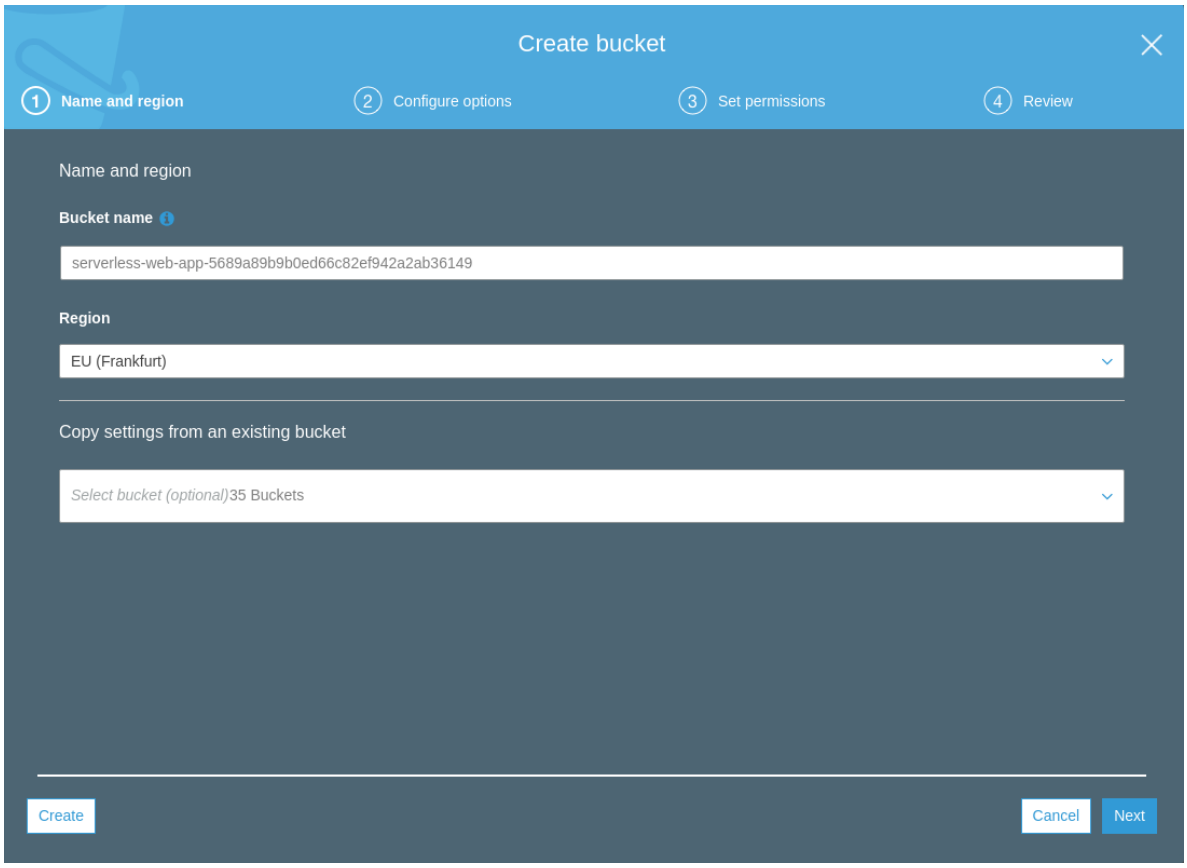
In AWS, some services are global like S3, IAM, and CloudFront, so there is no need to choose a specific location for these resources.

Building the Application Backend

Setting up our S3 Bucket

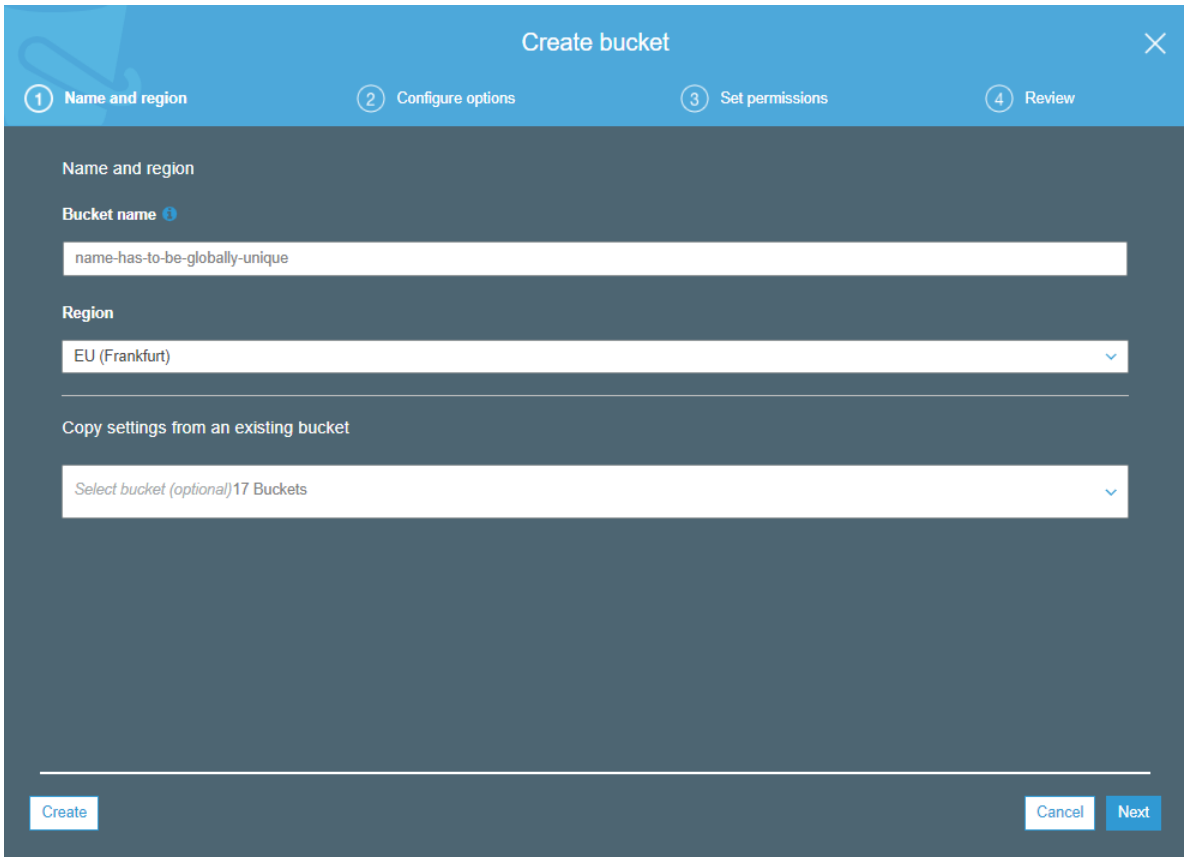
Enter AWS service S3 then click on “Create bucket”. Then give in the bucket name.

Note: The bucket name has to be globally unique for everyone! So you can not copy-paste the name used in this tutorial.

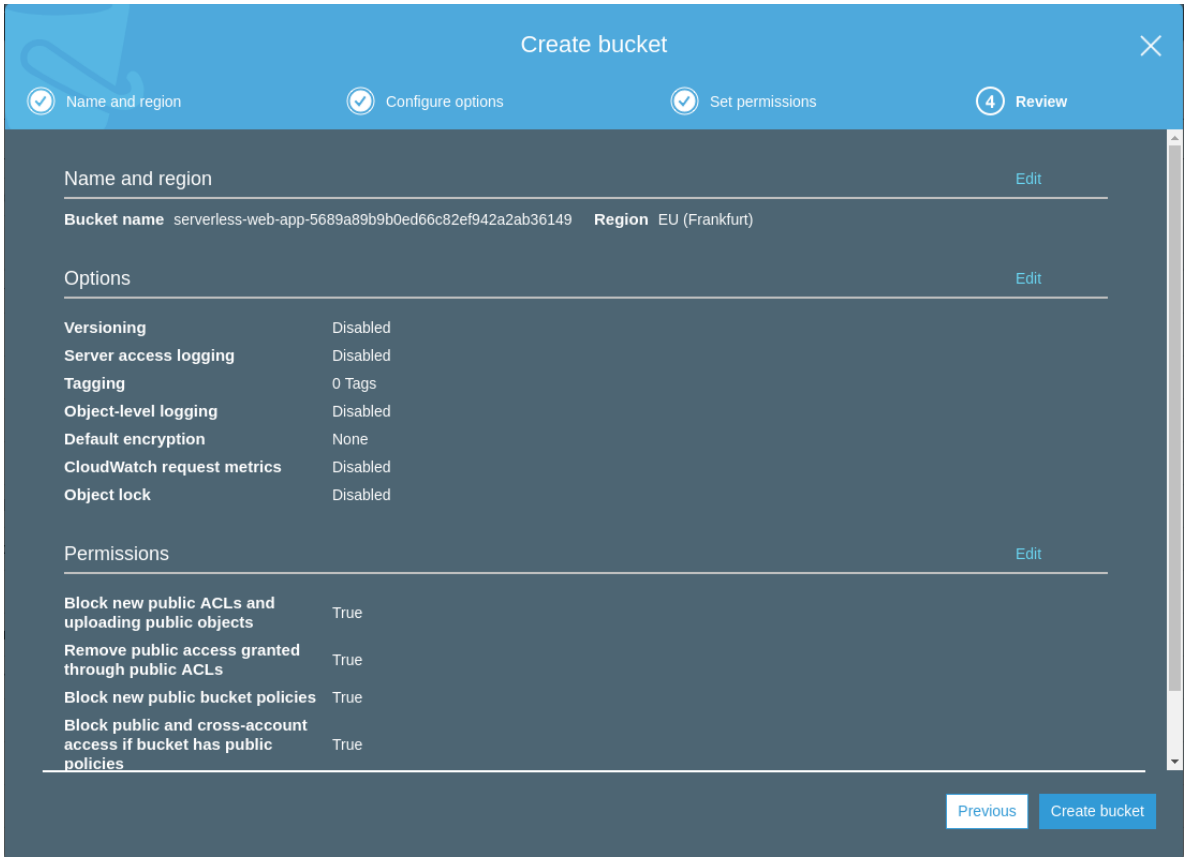


Note: To get a unique name, I usually append the output of `date|md5sum|awk '{print $1}'` to the name of the bucket.

Accept defaults for “Configure options”

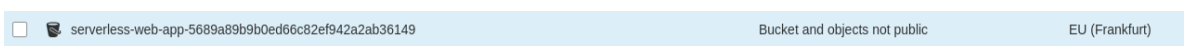


Accept the default for “Set permissions”



After “Review”, create the bucket.

In S3 dashboard, you will see:



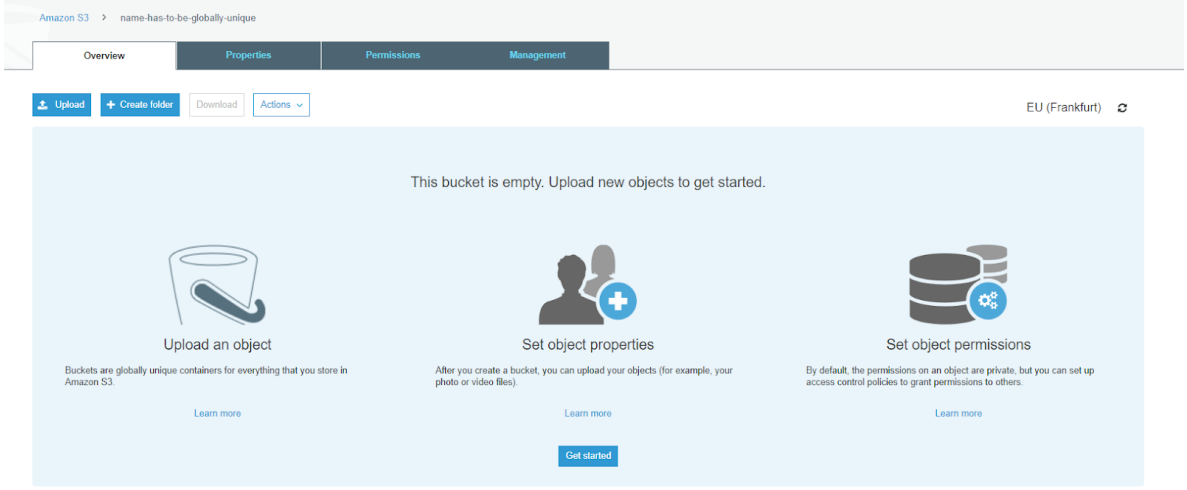
Note: Do not activate S3 Bucket static web hosting, because Cloudfront will consider the bucket as "Custom Origin" and this will leads to a different setting other than what described in this tutorial. This is even mentioned in the Cloud Front Developer Guide.

Building the Application Backend

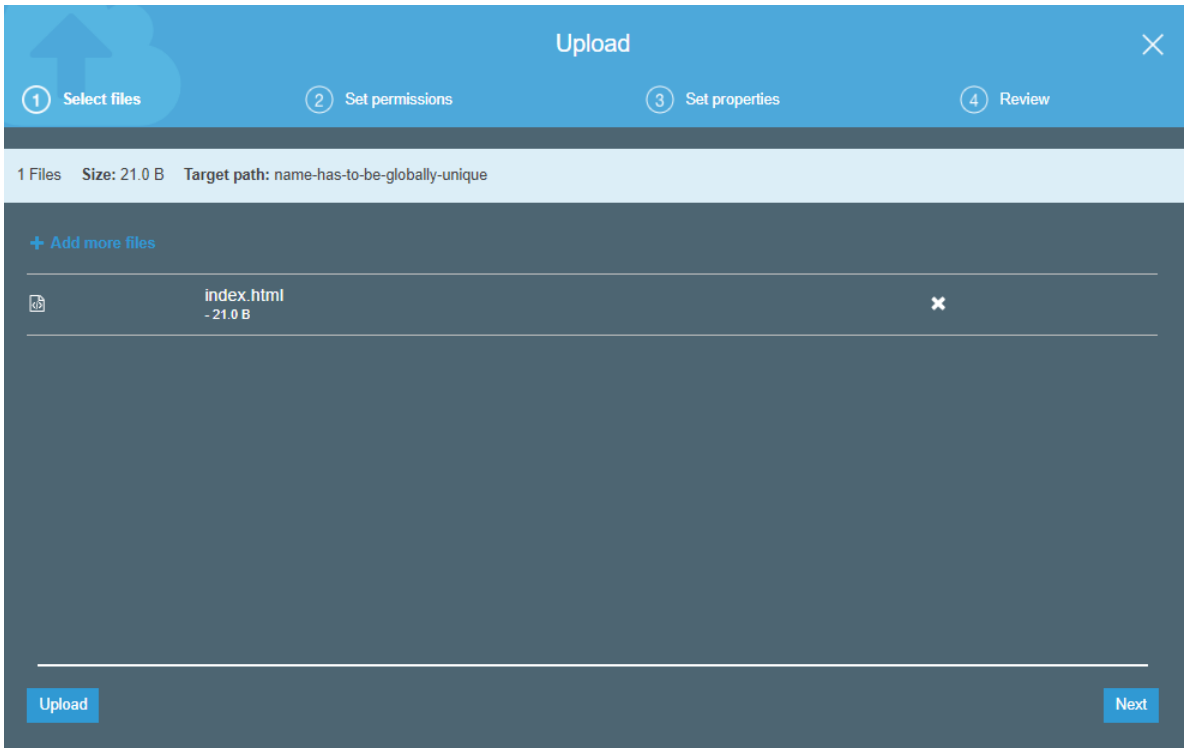
Uploading our Static Files

In this course, we will need to upload HTML, JS and other types of files to S3. We are going to upload all of them using the following routine:

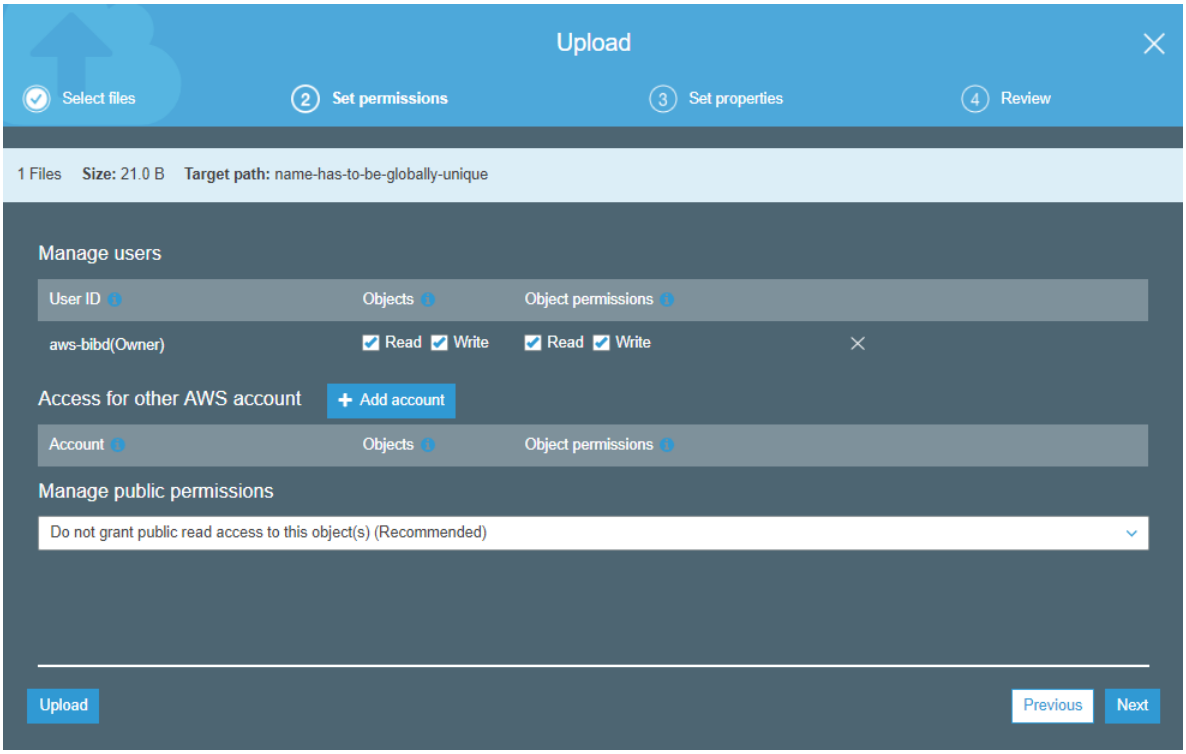
- Enter into the bucket, click on “Upload”



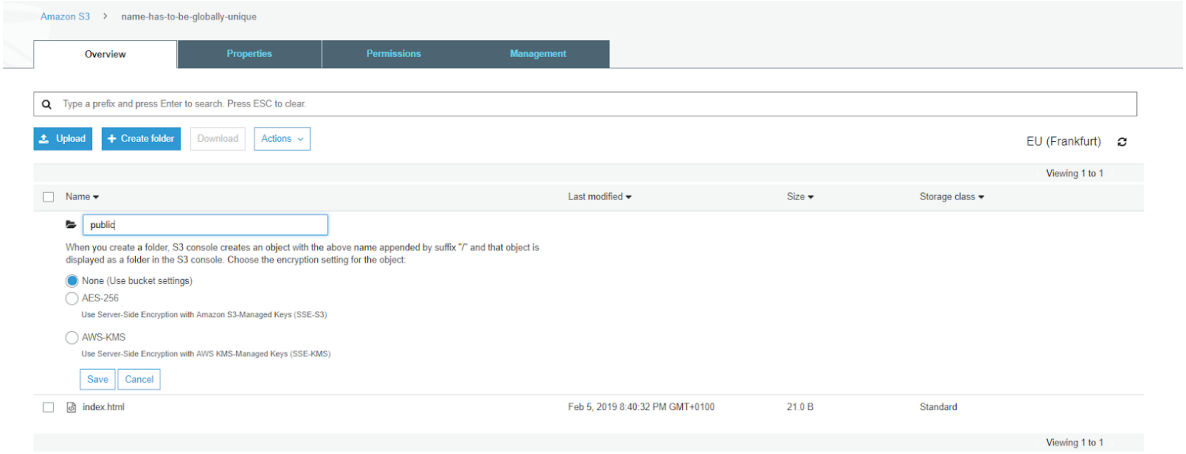
- Choose the file you want to upload



- Grant full access to Owner, do not grant public read access



- Keep all other settings default
- If you have to upload things that happen to be not in the root folder. You can create a folder structure by clicking on “Create folder”



- Click on save, then you have a normal folder created

Note: Uploading files one by one can be very slow, you can upload all of your files in one-shot using AWS CLI.

This is an example:

```
cd serverlesswebapp/code/s3
aws s3 cp . s3://serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149 --recursive
```

Note: You can copy all the files from the code folder to the bucket but it is recommended to go through all of them first and change some variables by their new values, then upload to S3. We don't really need to upload the code right now.

Building the Application Backend

Introduction to AWS Cognito

Cognito is described by AWS as a simple and secure user sign-up, sign-in, and access control managed service.

In other words, it lets you add user sign-up, sign-in, and access control to your web and mobile apps quickly and easily. It can scale to millions of users and supports sign-in with social identity providers, such as Facebook, Google, and Amazon, and enterprise identity providers via SAML 2.0. All of this without the need of setting up your infrastructure.

If you are using Cognito Identity to create a User Pool, you pay based on your **m**onthly **a**ctive **u**sers (MAUs) only.

A user is counted in the MAU if, within a calendar month, there is an identity operation related to that user, such as sign-up, sign-in, token refresh, or password change. You are not charged for subsequent sessions or for inactive users within that calendar month.

The first 50,000 MAU are included in the free tier.

There are two types of pools in AWS Cognito:

- `User Pool`
- `Identity Pool`

These 2 services are separate but related to each other. Let's see how:

User Pools act like a user directory for your application, including all the operations and transactions that come with user management, like signing it in and out.. etc.

On the other hand, Identity Pools allow you to delegate the authorization for AWS resources to AWS itself by mapping a user from an Identity Provider to an IAM role. In other words, Identity Pools are used to assign IAM roles to users who authenticate through a separate Identity Provider.

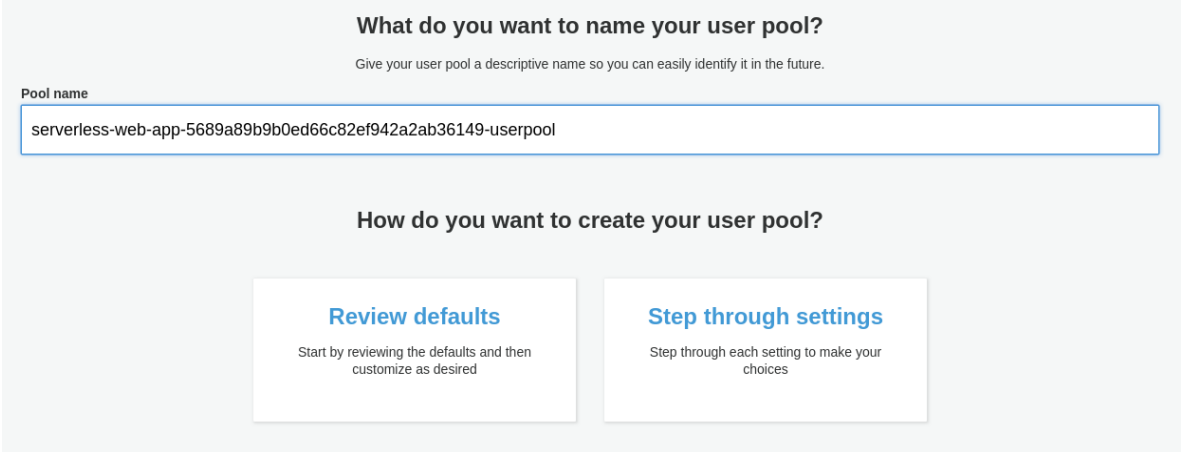
We will use `User Pool` in this tutorial.

Building the Application Backend

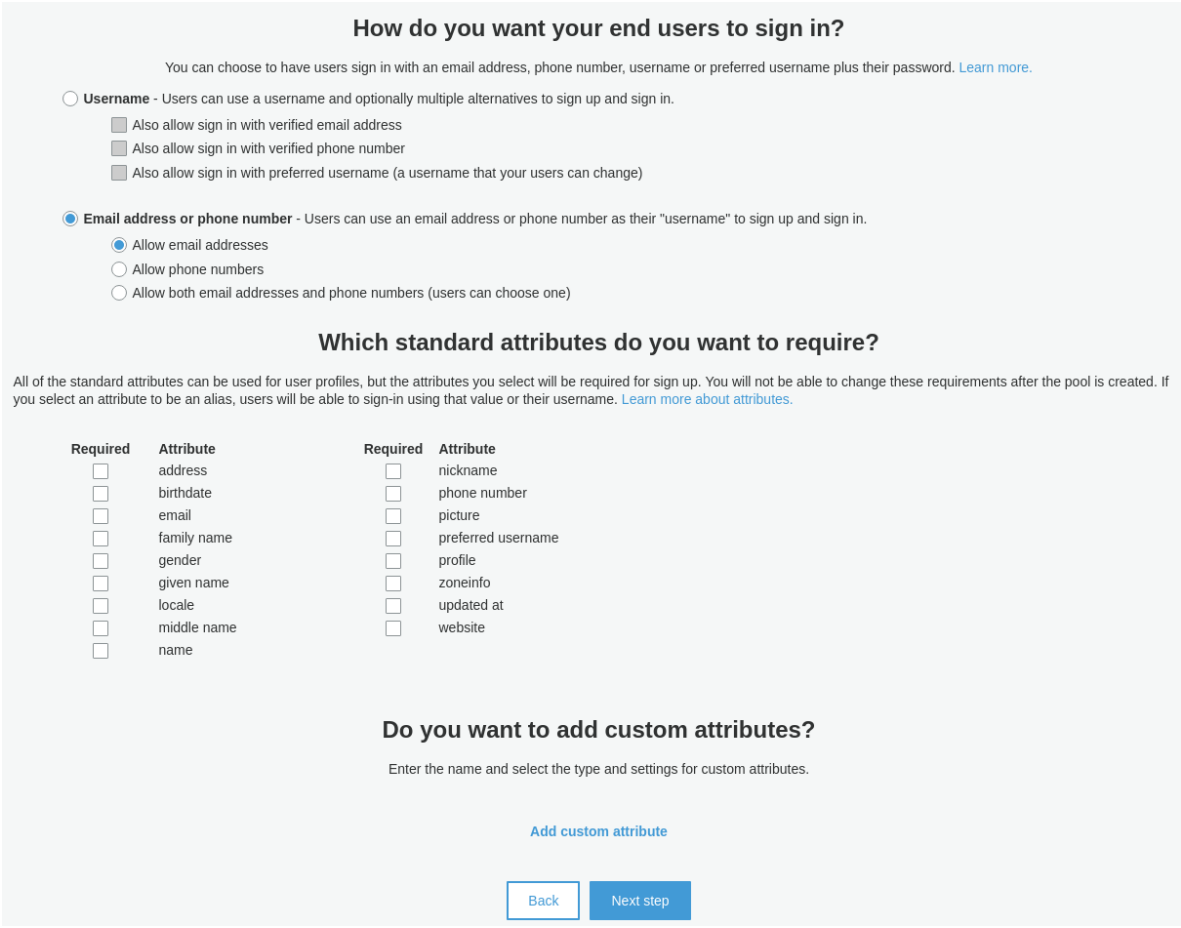
Setting up AWS Cognito

To do this, follow these steps:

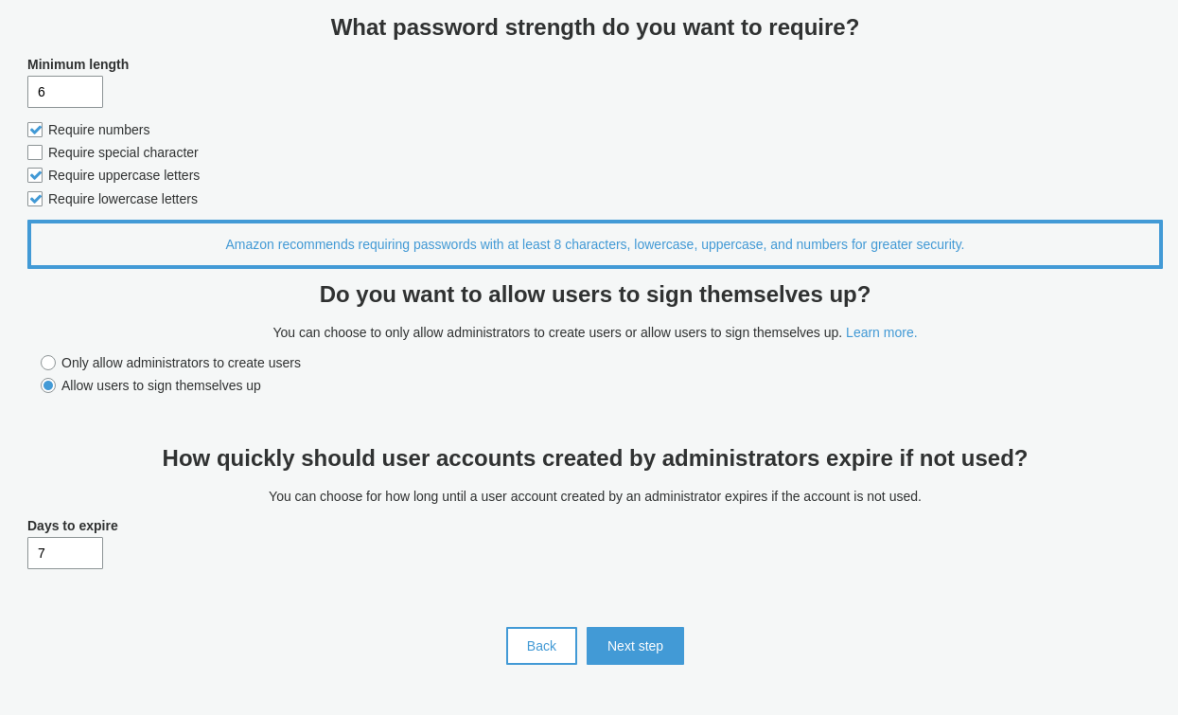
- Enter into AWS service Cognito
- Choose “Manage User Pools”
- Choose “Create a user pool” and type in a “Pool name”



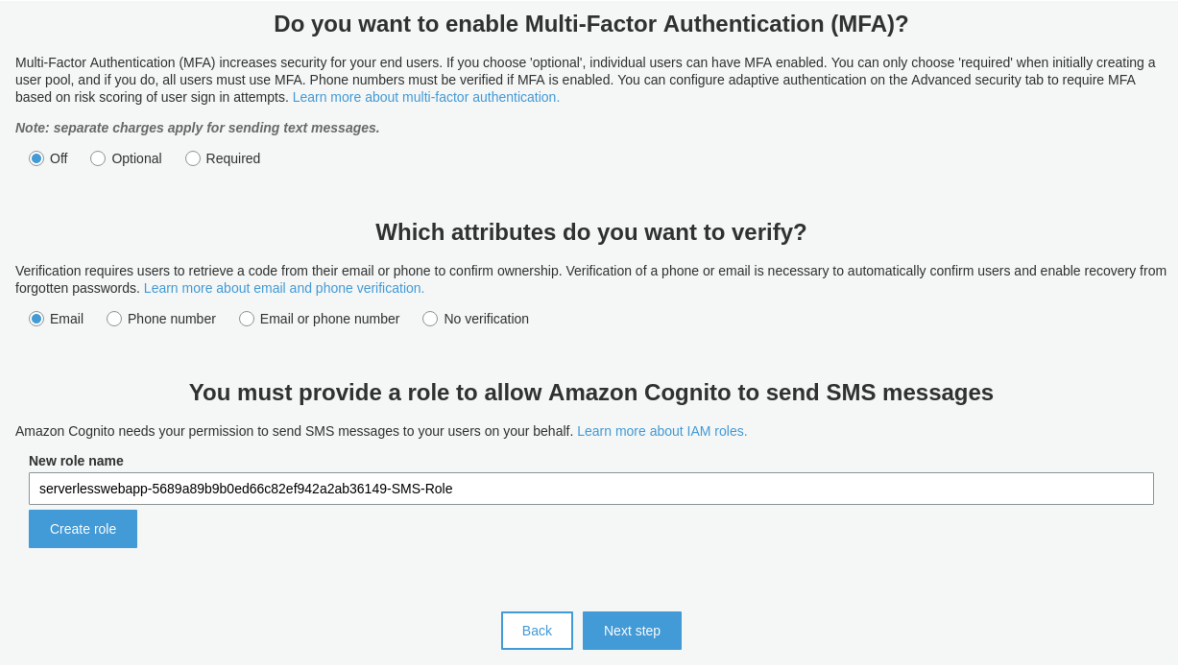
- Click “Step through settings”. For this case, we want the user to sign in with an email address and only an email address.



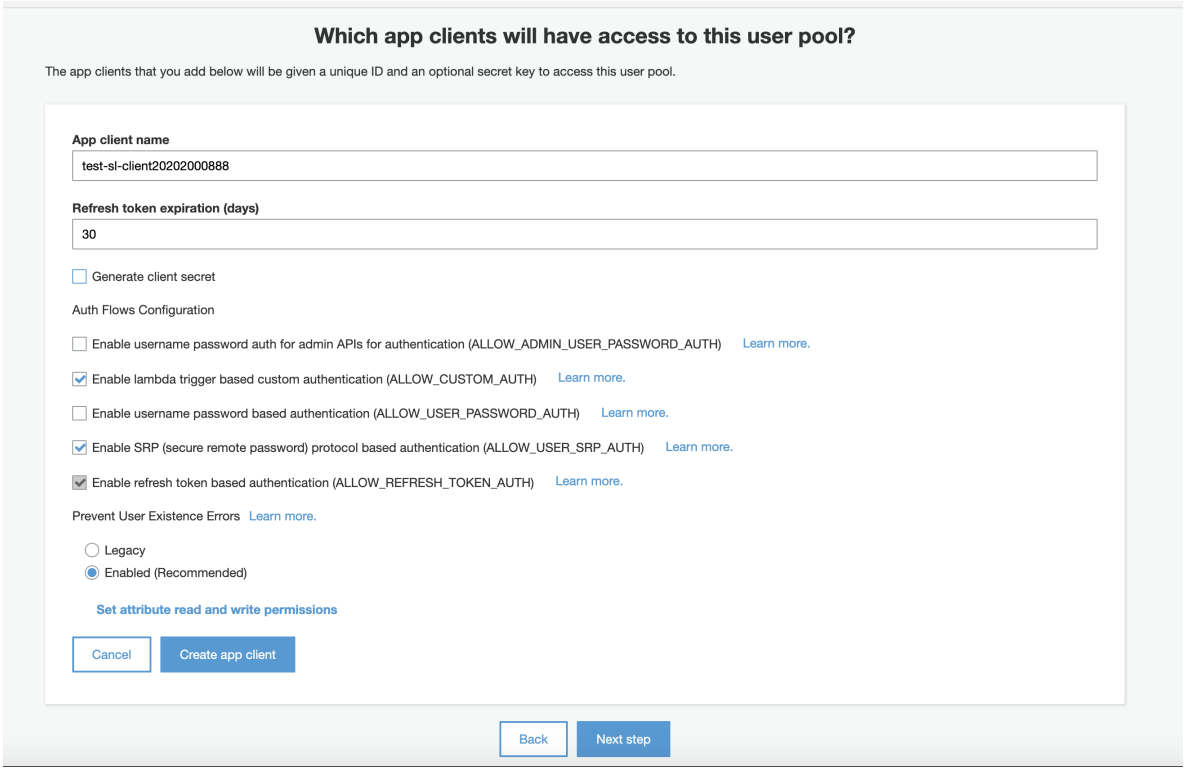
- Accept the default password requirements and “allow users to sign themselves up”



- Deactivate MFA and enforce Email verification (verification through email address will be used to accept reset password token, for example). In addition, for the next section, choose the verification method. I made the choice of confirming the user signup using a link for the sake of simplicity. Click on save and continue. We are not going to use the SMS for the account creation validation.



- Leave everything in the tab “Message customizations”, “Tags” and “Devices” as default
- Add an app client
- Fill in “App client name”
- Untick the “Generate client secret”, tick "USER_PASSWORD_AUTH"
- Click on “Create app client”
- Click on “Next step”



Note: Since we have enabled email verification, you will get an email sent to verify the signing up. If you are developing an application and do not want to confirm each time you use a fake email address like test@test.com for instance, we will deactivate temporarily the account confirmation by using the "Pre sign-up" feature and create a Lambda function that will deactivate this. We will demonstrate how to do it.

Building the Application Backend

Creating the AWS Lambda Pre Sign-up Function

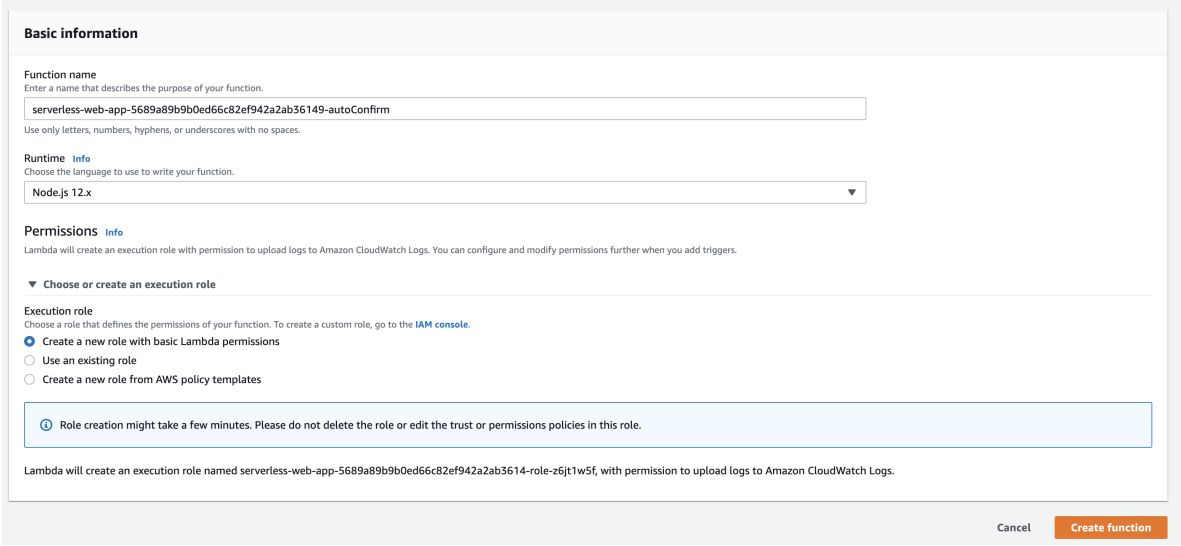
As its name describes it, the pre sign-up function is a Lambda function. It is triggered just before Cognito signs up a new user. It allows us to perform custom validation to accept or deny the registration request as part of the sign-up process.

We are going to create a pre sign-up function.

Open another window now and go to the AWS Lambda console and create a new function with Runtime Node.js 12.x , name this lambda function `serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-autoConfirm`.

- Use the default role "lambda_basic_execution"
- Choose runtime Node.js

Note: An AWS Lambda function's execution role (namely "lambda_basic_execution") grants it permission to access AWS services and resources. You provide this role when you create a function, and Lambda assumes the role when your function is invoked.

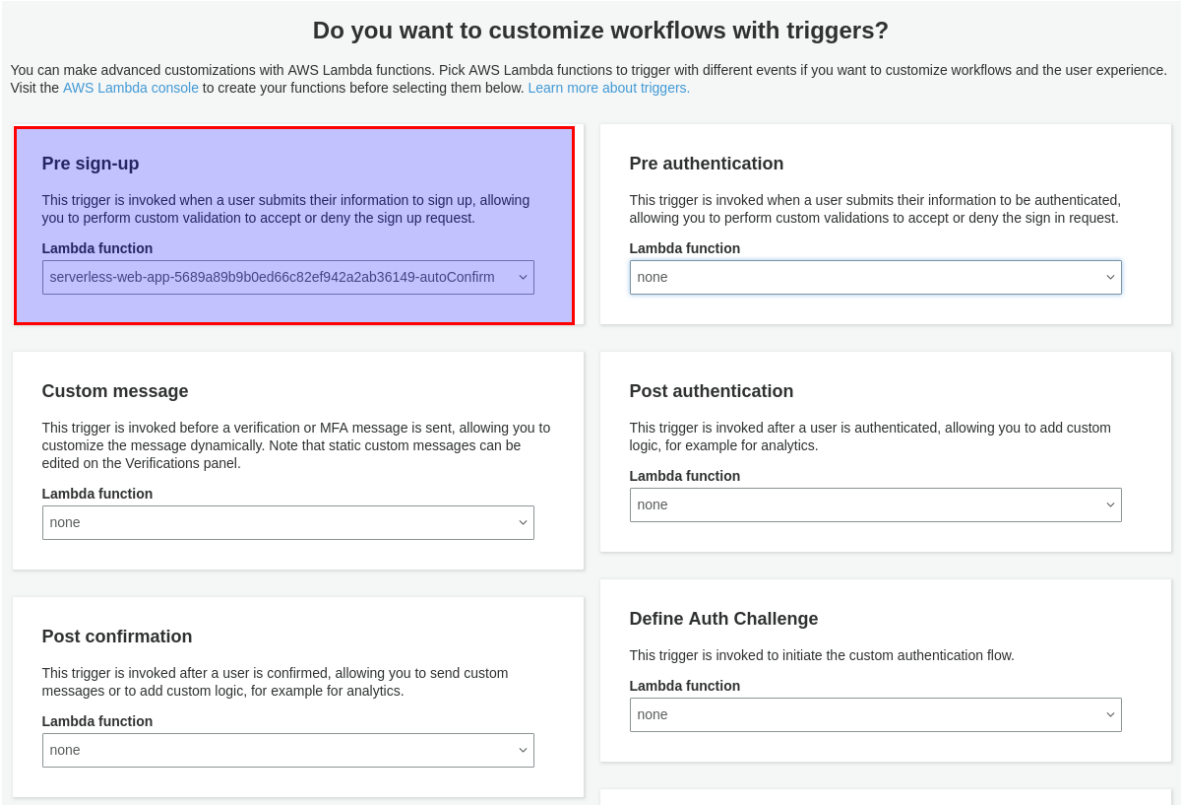


- Fill the following source code

```
exports.handler = (event, context, callback) => {
    event.response.autoConfirmUser = true;
    event.response.autoVerifyEmail = true;  // this is NOT needed if e-mail is
not in attributeList
    context.done(null, event);
};
```

- Click on “Save” to preserve the change
- On the AWS Cognito window, add the created lambda to “Pre sign-up” hook

Note: If the Lambda function does not show in the list of functions under the hook, go through all of the next steps then come back to the created pool and modify it to add the function.



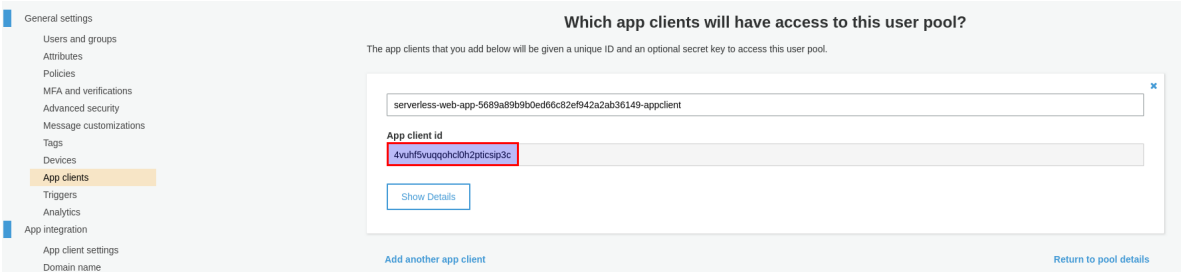
- After adding Lambda, proceed with the next steps, accept the default configurations and create the user pool.

Note: Two pieces of information here that we are going to use and reuse in the next steps:

- User Pool ID: eu-central-1_A0xCdzSim
- Cognito Application ID: 4vuhf5vuqqohcl0h2pticsip3c

You need to change these values by yours.

Note: Application ID can be found under the `App clients` menu, on the other hand the User Pool ID can be found under `General settings` menu.



Building the Application Backend

Creating a CloudFront Distribution to Deliver S3 Content

Go to AWS CloudFront, list the distributions from CloudFront dashboard, then click on `Create Distribution`.

There are two steps to create a distribution:

- Selecting the delivery method
- Creating the distribution

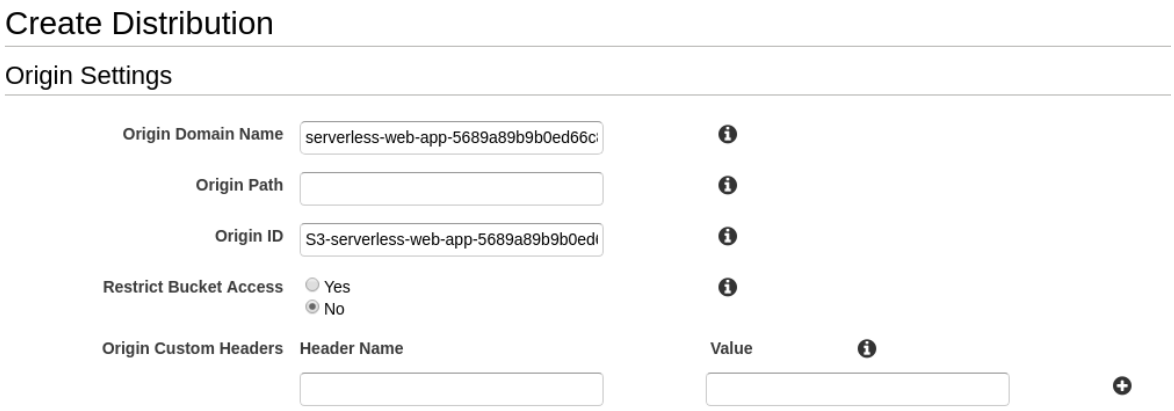
There are two options for the delivery method (`Web` and `RTMP`). We will use the `web` one.

Note: RTMP distributions stream media files using Adobe Media Server and the Adobe Real-Time Messaging Protocol (RTMP).

Now, click on “Get Started”.

You will be led to a page where the distribution settings should be filled.

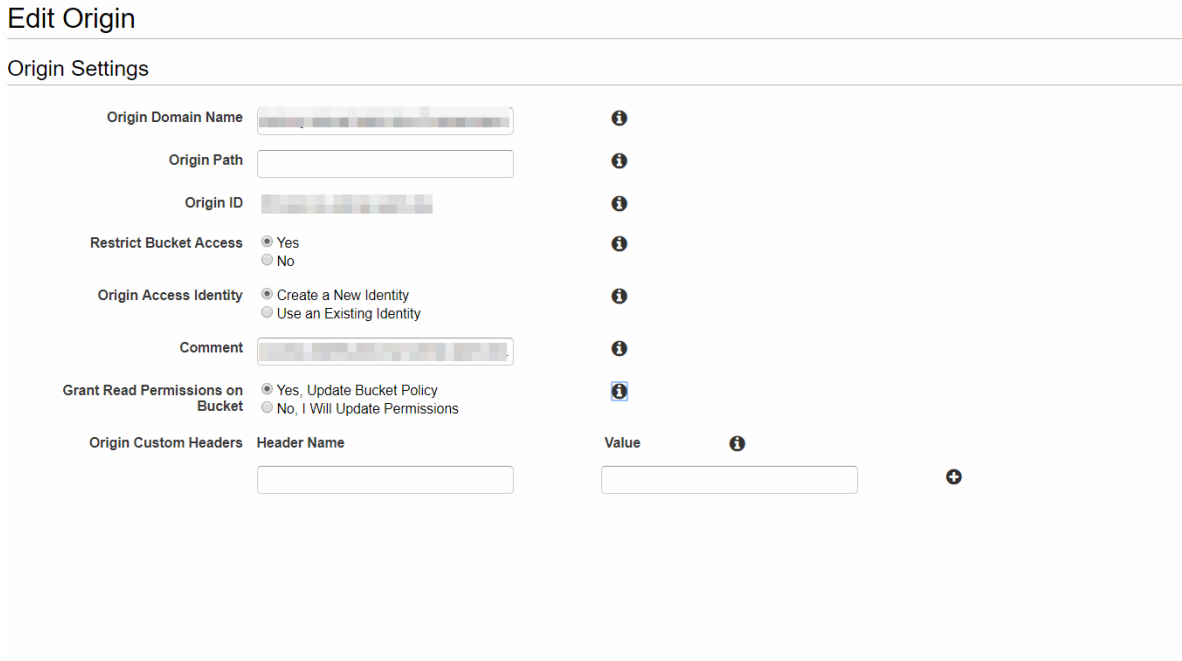
- Fill in `Origin Domain Name` with the domain name of S3 bucket (Origin ID will be filled in automatically according to Origin Domain Name)
- Leave everything else in default for the moment
- Click on `Create Distribution` button (you will see the distribution going into status `in Progress`)



The creation will take up to a few minutes until the deployment of the distribution and edge locations finish.

We can access our website (the static files HTML / JS script) in the S3 bucket through both the distribution domain and the S3 bucket domain. If we want the static files can only be accessed through the distribution domain, we should follow these steps:

- Enter into "Distribution" through AWS service dashboard
- Choose the “Origins and Origin Group” tab
- Tick the S3 bucket origin and click “Edit”
- Configure “Restrict Bucket Access” to “yes”
- Configure “Origin Access Identity” to “Create a New Identity”
- Configure “Grant Read Permission on Bucket” to “Yes, Update Bucket Policy”



Note: An origin access identity is a special user that you can use to give access to your Amazon S3 bucket. This is useful when you are using signed URLs or signed cookies to restrict access to private content in Amazon S3. To change the comment for an origin access identity, enter the new value and click Yes, Edit.

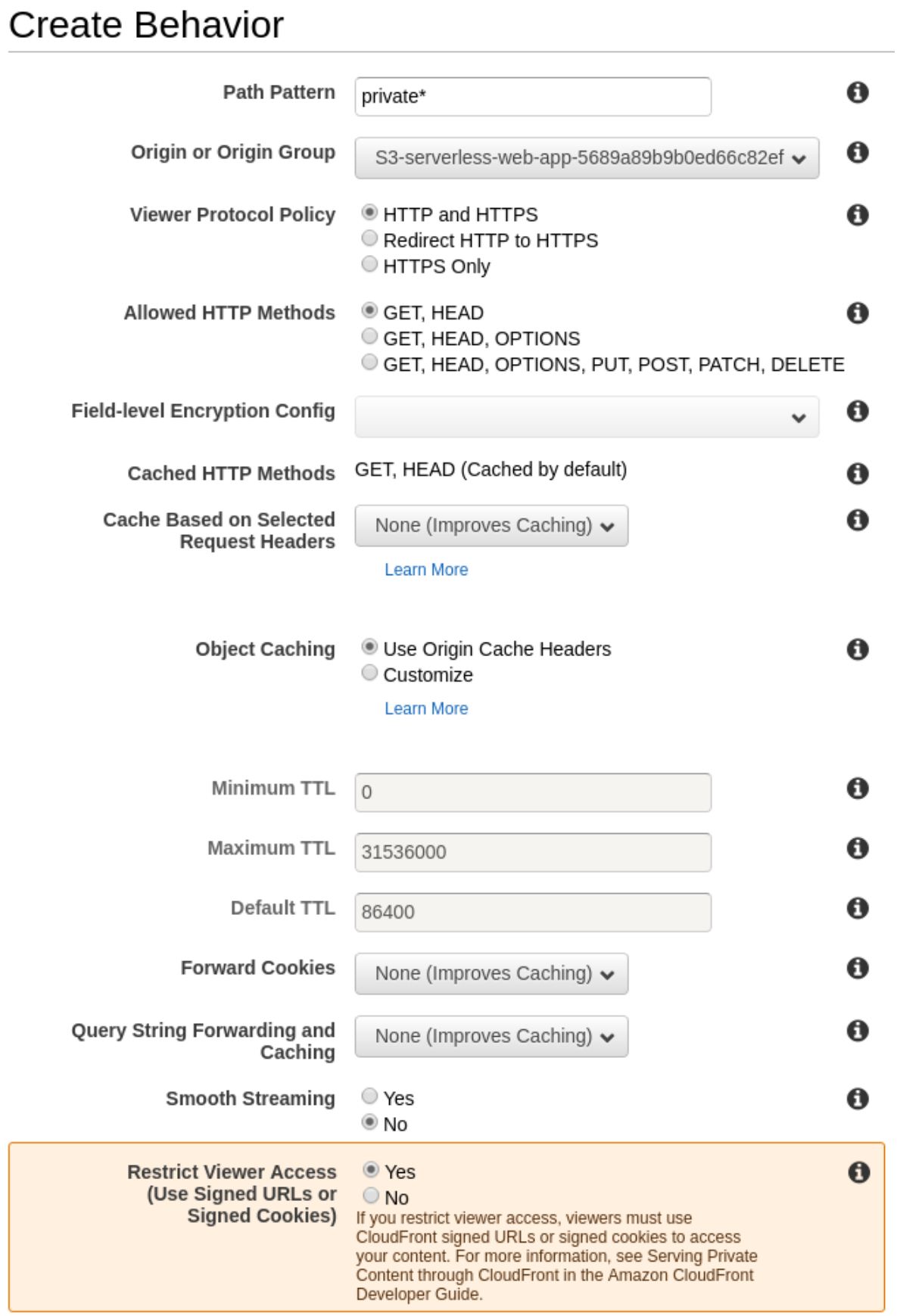
Note: An AWS account can have up to 100 CloudFront origin access identities. Nevertheless, you can add an origin access identity to as many distributions as you want, because in most case one identity is enough.

Let's now move to create a "Behavior". Click on the "Behaviors" tab and create a new one. Since we need to protect the access to the private folders called `private*` (private1/ private2/ ..privateN/), we use the pattern `private*` in the Path Pattern.

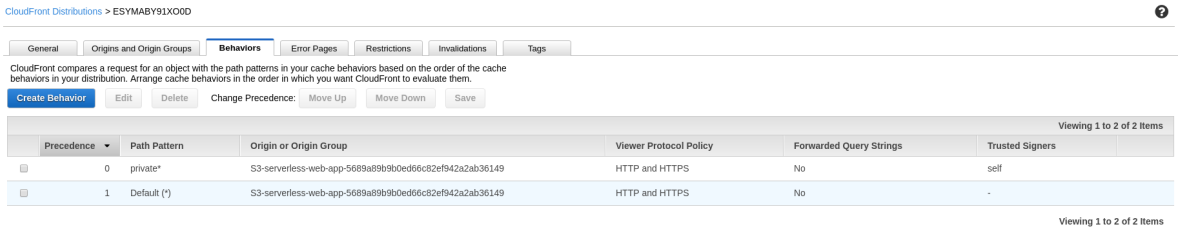
Note: This is a reminder about an example we can use to store the private files:

```
.
├── private-1
│   ├── private-11.html
│   └── private-12.html
├── private-2
│   └── private-21.html
├── private-3
│   └── private-31.html
```

Tick the "Restrict Viewer Access" option.



After creating the behavior, you will notice that we have two behaviors right now, the "Default" one and the newly created one.



Now our path `<bucket>/private*` is secured which means that it can not be accessed publicly.

We want it to be open for the rightful user who authenticate themselves with Cognito `idToken`. This authentication is implemented with a lambda function and triggered by the behavior "private*".

In the next steps, we are going to create the Lambda function to help users authenticate, but since are using Cognito, it's important to know how JWT works even briefly.

Building the Application Backend

Introduction to JWT and Lambda@edge

When a client authenticates to our application using the Cognito user pool that we created, the Cognito service sends back a JWT message which is a Base64-encoded JSON string that containss useful information about the user like name and email (called claims or the ID Token).

Cognito returns 2 other tokens:
The Access Token, used to grant access to authorized resources and the Refresh token containing the information necessary to obtain a new ID or access token

In all there are 3 tokens:

- ID token
- Access token
- Refresh token

A token includes 3 main sections:

- a header
- a payload
- a signature

Typically, a JWT looks like:

```
xxxxx.yyyyy.zzzzz
```

Where `xxxxx` is the header, `yyyyy` is the payload and `zzzzz` is the signature.

Example:

```
iIsInR5cCI6IeyJhbGciOiJIUzI1NkpXVCJ9.
eyJpc3MiOiJ0wiZXhwIjoxNDI2NDIwODAwLCJodHRwOi8vdG9wdGFsLmNvbS9qd3RfY2xhaW1zL2lzX2
FkbWluIjp0cnVlLb3B0YWwuY29tIiCJjb21wYW55IjoiVG9wdGFsIPwaQupWkJ1ZX0.
yRQYnWzskCiwiYXdlc29tZSI6dHZUxiUzKELZ49eM7oWxAQK_ZFw
```

JWT or JSON Web Tokens is used to securely transmit information between two or more parties, but it's mainly used to manage authorizations: Once one of our app users is logged in, each subsequent request should include the JWT. This will allows them to access resources that are permitted with that token.

In order to implement this, we are going to create a new Lambda@edge function.

Lambda@Edge lets us run a Lambda function to customize content that CloudFront delivers.

From a geographical point of view, Lambda@Edge, unlike the common Lambda function, has the advantage of executing the function in AWS locations closer to the viewer.

A Lambda function will run in response to a CloudFront event and it can be useful for some use cases like changing CloudFront requests and responses.

Building the Application Backend

Creating the Verification Lambda Function

First, let's create the verification Lambda, we will call it "serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-verifyToken".

Note: I'm using long names inlcuding "5689a89b9b0ed66c82ef942a2ab36149" because some resources on AWS like S3 don't allow the creation of objects that exist with the same name even if this object is created by another user.

Example: If a user before you used "my-bucket" to create their S3 bucket, you will not be able to create another bucket with the same name.

Because this Lambda must include several packages used for Cognito token verification, we can not use code entry type `code inline`. It has to be uploaded from an S3 bucket.

Note: We will not use our website S3 bucket here, namely "serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149" but we are going to create a new bucket for the code package.

```
mkdir serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-verifyToken

cd serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-verifyToken
```

We are going to use Node.js v12. You can install `n`, a Node version manager if you need to manage multiple versions in the same machine.

```
npm install -g n
```

Now select the version we need to use:

```
n 12
```

Verification:

```
node --version

v12.18.0
```

Create lambda function entry point (index.js):

```
'use strict';
var jwt = require('jsonwebtoken');
var jwkToPem = require('jwk-to-pem');
var USERPOOLID = 'eu-central-1_A0xCdzSim';
```

```
var JWKS = '{"keys":
[{"alg":"RS256","e":"AQAB","kid":"I/1BcIZQIr18RW+SDDXTKNGodWG0yMrPBqYE7c2TH70=",
"kty":"RSA","n":"mJgDyAbYCJFCQmsJDsDYn7XBbycClsXEB1KGQ3MNWjasH-
cZ7yFlhFW3MRj8LYBZmZHg1FSG2dXe6pvp4S8RTGwI5oRFdT4BeS1JLRbwyMKL1vOKFaJgjhxEhKUDAx
1sskcGwaRyMtgvL7MHOCdnHjp-QL9_Hnkbo-Hj4mrNW8CNkR6aozEDE4yFQfjcQukF9dXYnJOGbm-L-
fGuqQYCxYcxvC0o7tzQlb_c7u4RJfZx1SprZQHgmbGhjbc9s3h6p6jUnlTSmDjyD0LgYLgPUrewxrO00
_dJUeam4lJQrFjnR-xh9Eim1EhWE0ad-rHHyMrKl2I7ZB1mSW3n-fgUiQ","use":"sig"},
{"alg":"RS256","e":"AQAB","kid":"5AVUJSRIyc/ZoOHTpgrPqokM1ZUKZoD2j5NLFu/hI5w=","
kty":"RSA","n":"jBkJxhvaPqhIg6B6Yz-
WnpjITC63vIETNU2BOrid5Y6y6_Nls3195Ki8ZZGujNgRoHM92LNBzL_RsonnoomF7OIGpKv4WkIJkZD
aJ8n-
WSkDvyw0GWsv_kBjGDb01Cgz_esO6zJjsJzGFauEREIiSLRppxat3PnIujCXr9XBLsH82JzjimVIhuXd
WnmOafeX1I73FQl9aB60zpos_qgAmQLuaUGg5VRaXSJpzziCQ6SDEDwX4_sMRhnkAoNI-
hxtlJNejpzCJY01sFIf5IuRfHwtp21zo5Is_ZbirgNvJlKFx6oZxG1CGES2trEaKr73Qx0HhWttSjUmZ
9s8TYVq4w","use":"sig"}]}';


var region = 'eu-central-1';
var iss = 'https://cognito-idp.' + region + '.amazonaws.com/' + USERPOOLID;
var pems;

pems = {};
var keys = JSON.parse(JWKS).keys;
for (var i = 0; i < keys.length; i++) {
    //Convert each key to PEM
    var key_id = keys[i].kid;
    var modulus = keys[i].n;
    var exponent = keys[i].e;
    var key_type = keys[i].kty;
    var jwk = {
        kty: key_type,
        n: modulus,
        e: exponent
    };
    var pem = jwkToPem(jwk);
    pems[key_id] = pem;
}

const response401 = {
    status: '401',
    statusDescription: 'Unauthorized'
};

exports.handler = (event, context, callback) => {
    const cfrequest = event.Records[0].cf.request;
    const headers = cfrequest.headers;
    console.log('getting started');
    console.log('USERPOOLID=' + USERPOOLID);
    console.log('region=' + region);
    console.log('pems=' + pems);

    //Fail if no authorization header found
    if (!headers.authorization) {
        console.log("no auth header");
        callback(null, response401);
        return false;
    }
```

```
    //strip out "Bearer " to extract JWT token only
    var jwtToken = headers.authorization[0].value.slice(7);
    console.log('jwtToken=' + jwtToken);

    //Fail if the token is not jwt
    var decodedJwt = jwt.decode(jwtToken, {
        complete: true
    });
    if (!decodedJwt) {
        console.log("Not a valid JWT token");
        callback(null, response401);
        return false;
    }

    //Fail if token is not from your UserPool
    if (decodedJwt.payload.iss != iss) {
        console.log("invalid issuer");
        callback(null, response401);
        return false;
    }

    //Reject the jwt if it's not an 'Access Token'
    if (decodedJwt.payload.token_use != 'access') {
        console.log("Not an access token");
        callback(null, response401);
        return false;
    }

    //Get the kid from the token and retrieve corresponding PEM
    var kid = decodedJwt.header.kid;
    var pem = pems[kid];
    if (!pem) {
        console.log('Invalid access token');
        callback(null, response401);
        return false;
    }

    //Verify the signature of the JWT token to ensure it's really coming from
your User Pool
    jwt.verify(jwtToken, pem, {
        issuer: iss
    }, function (err, payload) {
        if (err) {
            console.log('Token failed verification');
            callback(null, response401);
            return false;
        } else {
            //Valid token.
            console.log('Successful verification');
            //remove authorization header
            delete cfrequest.headers.authorization;
            //CloudFront can proceed to fetch the content from origin
            callback(null, cfrequest);
            return true;
        }
    });
};
```

The code above is self-explanatory. You will certainly need some basic JS knowledge. If you don't, some exercices will not hurt.

Also, make sure to change the values of `USERPOOLID`, `JWKS` and `region` by your values.

You can get your JWKS by visiting this page:

```
https://cognito-idp.{region}.amazonaws.com/{userPoolId}/.well-known/jwks.json
```

You should change {region} and {userPoolId} by their respective values then install the dependencies using npm.

Note: If you are not sure about your region, this guide may help.

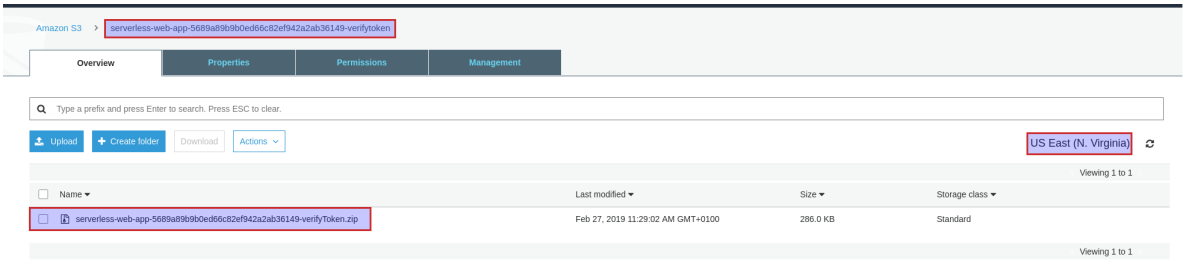
```
npm install jsonwebtoken
npm install jwk-to-pem
```

Now create a ZIP file containing the index.js file as well as all the dependencies:

```
zip -r ../serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-verifyToken.zip .
```

Then upload it to an S3 bucket that you can create. We used the US East region in this example.

```
aws s3 cp  \
../serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-verifyToken.zip \
s3://serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-verifytoken
```



Since we are building an Edge function, our new function should be deployed to the region US-EAST-1 otherwise CloudFront (at least for the moment), will not accept it. Check the Lambda limitation in the official documentation.

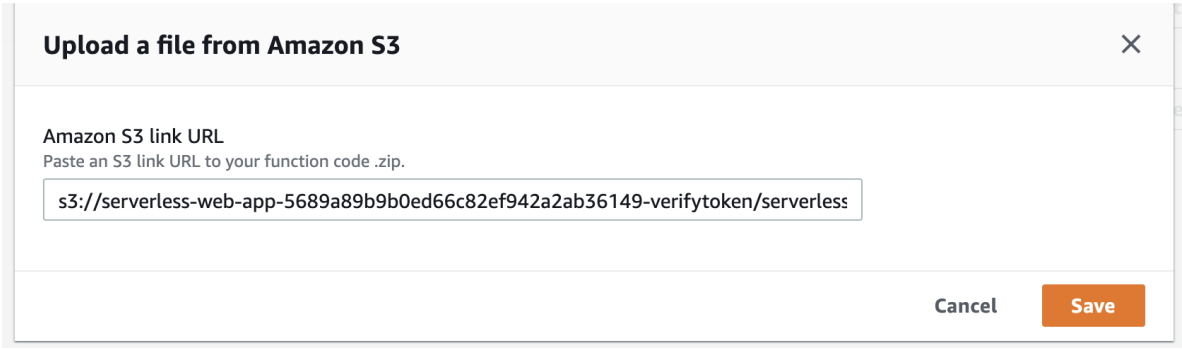
Open you Lambda console in the good region: https://console.aws.amazon.com/lambda/home?region=us-east-1

Create a new function. In my example, the function is also called `serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-verifyToken` .

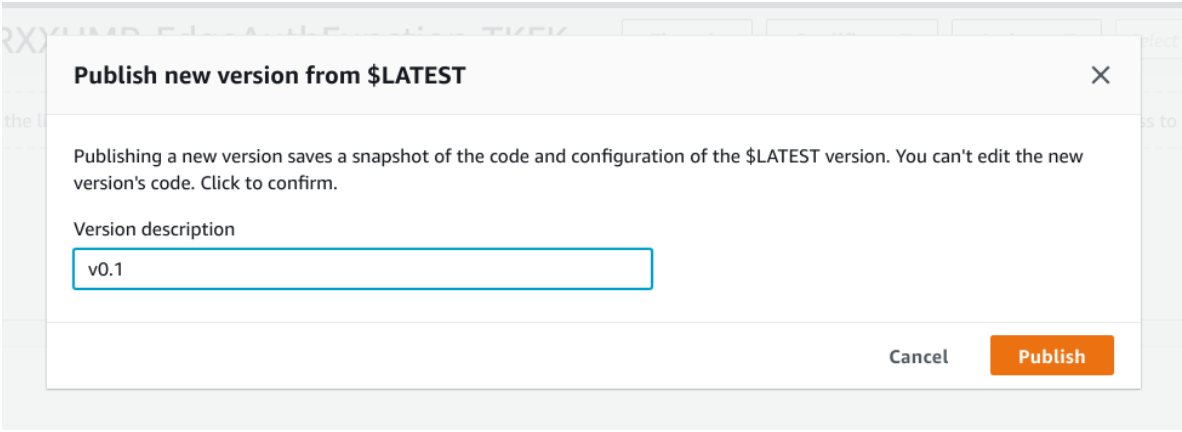
In the Code entry type, choose S3. Choose Node.js as a runtime and `index.handler` as a Handler.

Make sure to choose the right Node.js version.

Do not forget to add the S3 file URL that you can get by clicking on the archive file from the S3 bucket.



From the action menu on the top of the page, click on publish a version of your function by clicking on Action > Publish a new version.

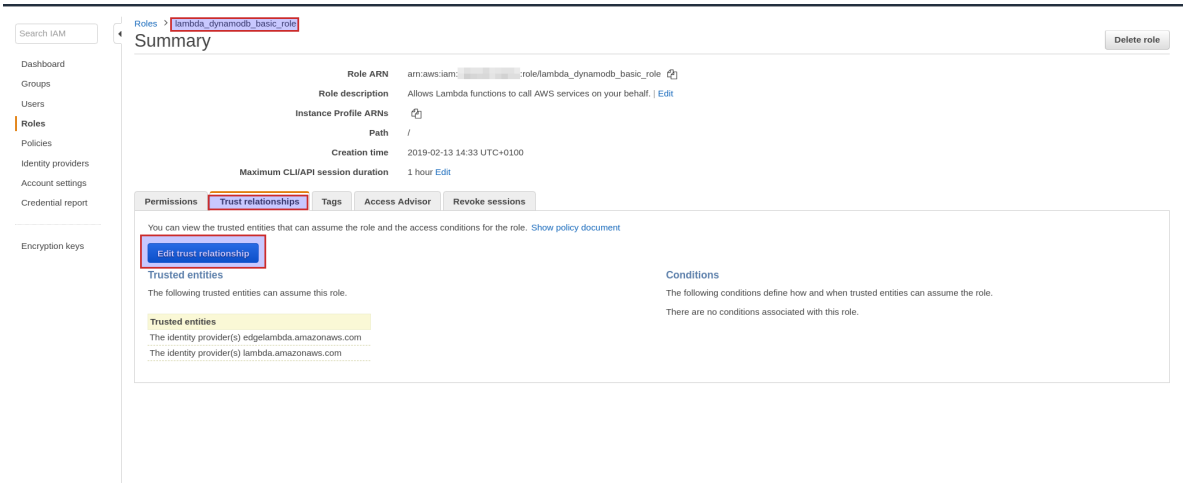


Now copy the ARN since we are going to use it later in CloudFront.

Note: ARN is the abbreviation of Amazon Resource Name, a unique identifier of a resource like a bucket or a Lambda function.

The next step is updating the IAM role for the lambda function with the needed permisions.

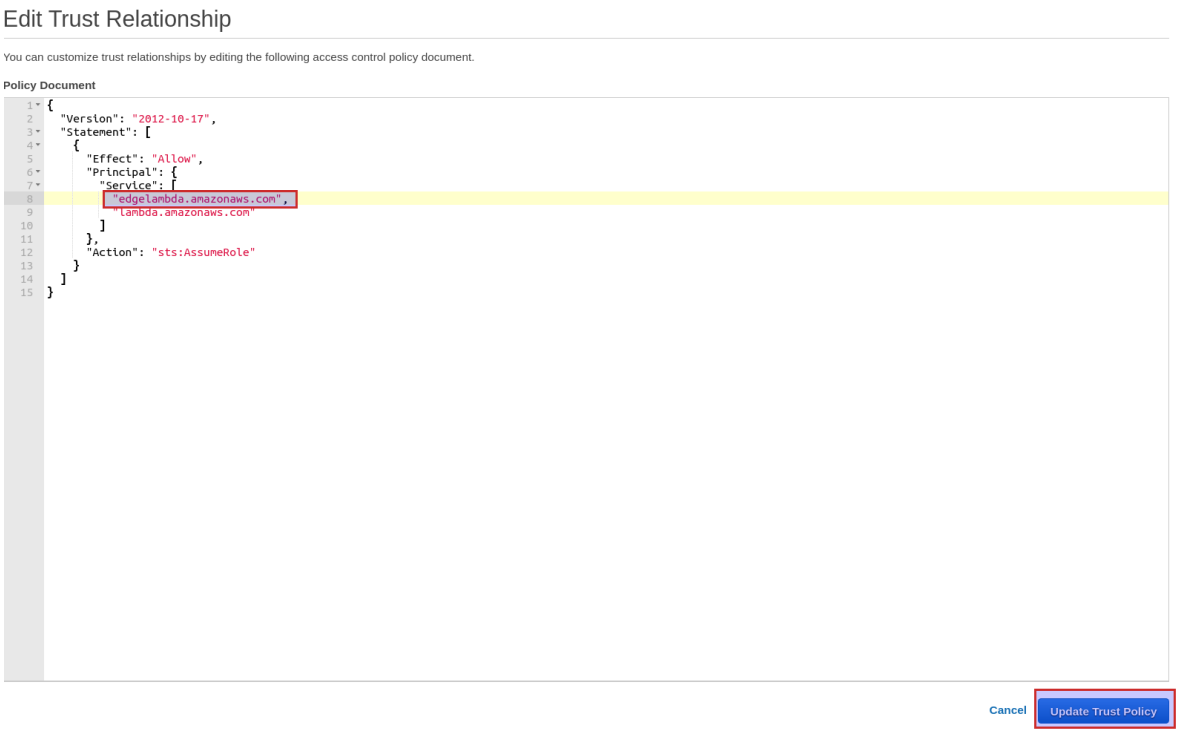
Choose the basic execution role we previously created then click on Trust Relationships > Edit trust relationships and add the principal `edgelambda.amazonaws.com`.



This can be done by pasting this JSON file instead of the old one.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": [
```

```
            "edgelambda.amazonaws.com",
            "lambda.amazonaws.com"
          ]
        },
        "Action": "sts:AssumeRole"
      }
    ]
}
```

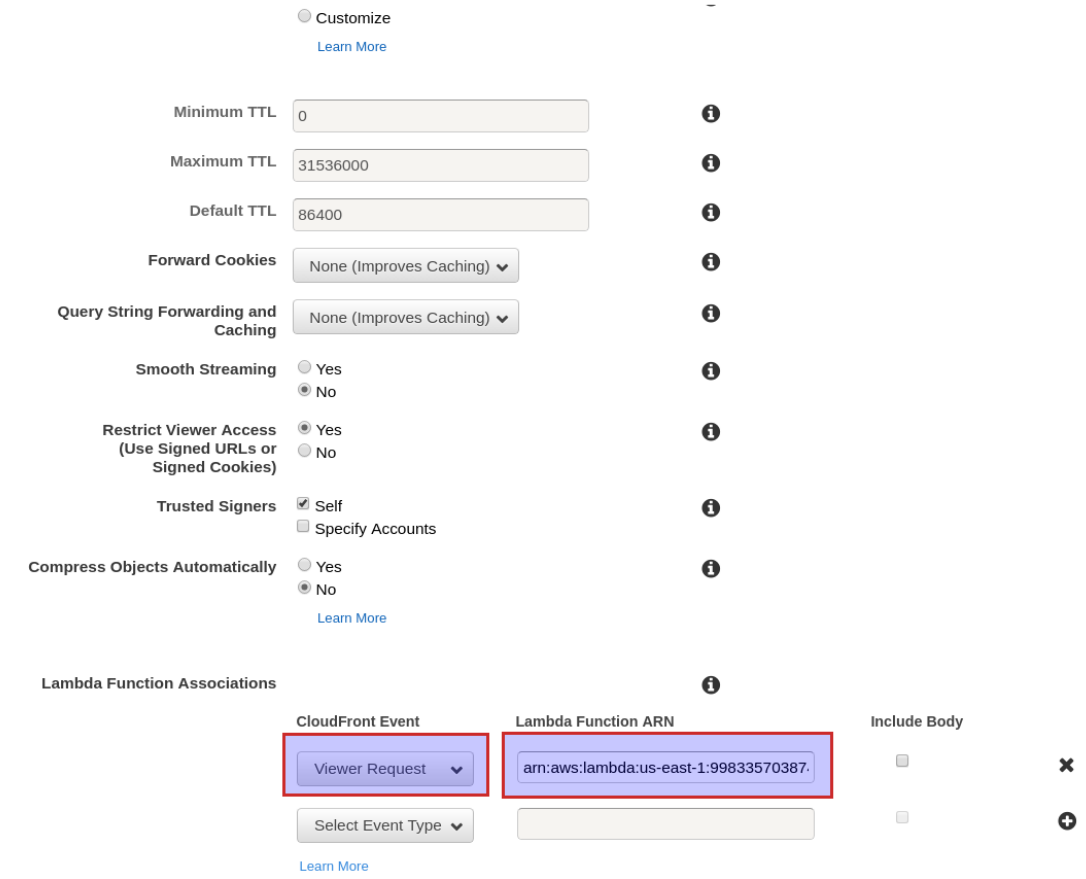


This role is assumed by the service principals when they execute your function and it is required for the execution of our Lambda function.

The last step is to update the CloudFront distribution, and modify the behavior with the pattern "private*".

Choose the CloudFront Event "Viewer Request" and paste the lambda function ARN next to it and then save the changes.

Edit Behavior



Building the Application Backend

Introduction to Social IdP with Cognito

Instead of letting users sign-up to your Cognito user pool, you can also allow users to sign-in with their current Google account through Cognito Social Identity Provider. This is what we are going to do in this part.

This feature can be also used on mobile apps (as well as web apps), so users can sign-in through social identity providers (IdP) like Facebook, Google, and Github.

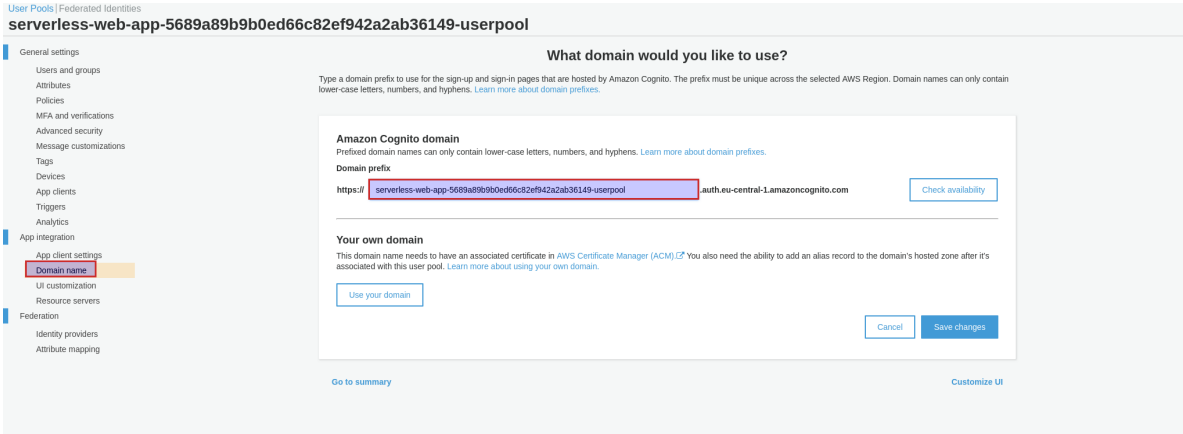
Cognito has a built-in token handling and management features for all authenticated users, so our backend system can standardize on one set of user pool tokens.

Building the Application Backend

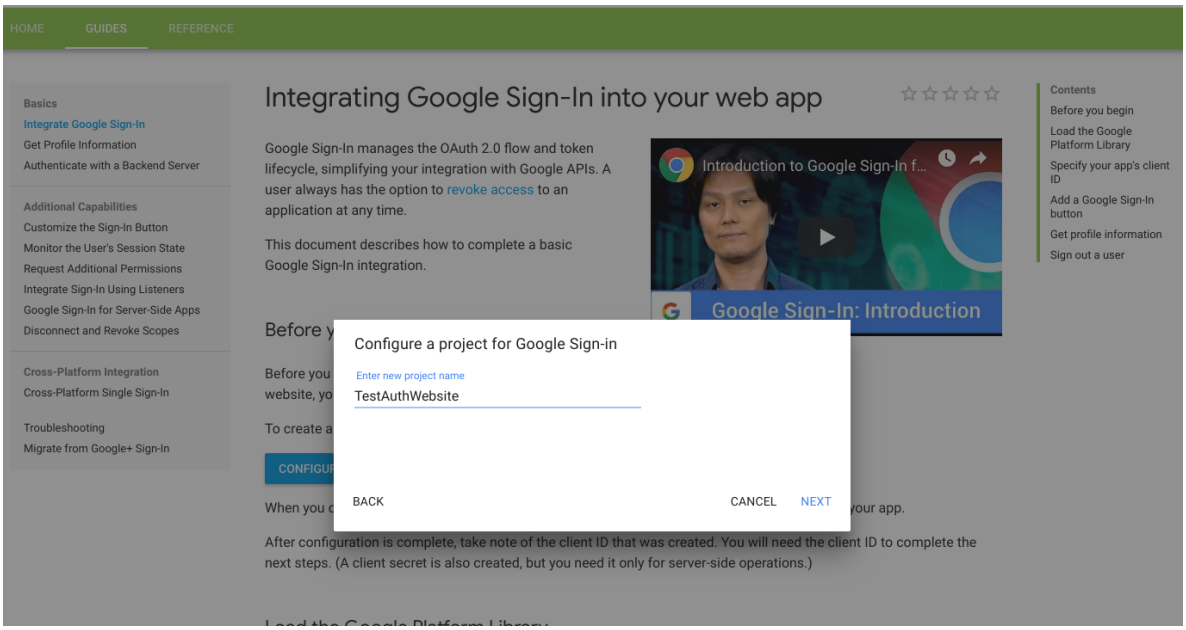
Adding Social Identity Provider to a User Pool

Before proceeding with this step, if you did not add a domain to the Cognito pool you created in the previous steps, go back to your Cognito dashboard and add a domain.

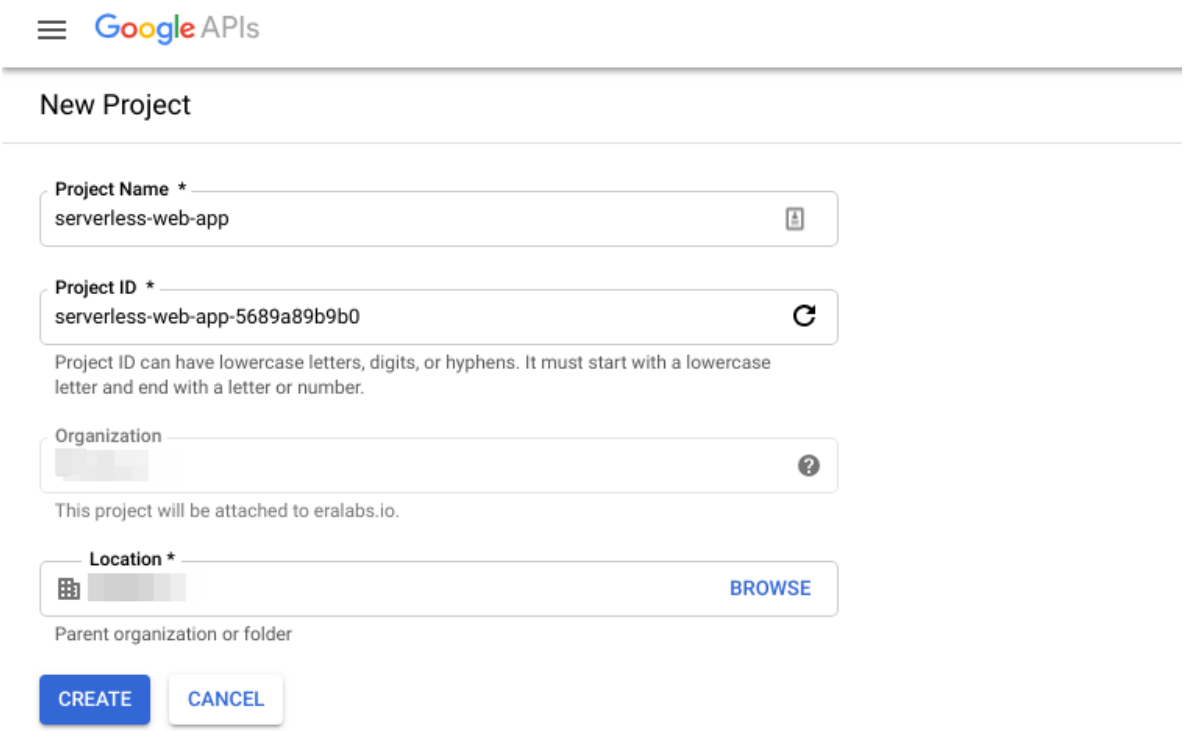
I will be using a subdomain here:



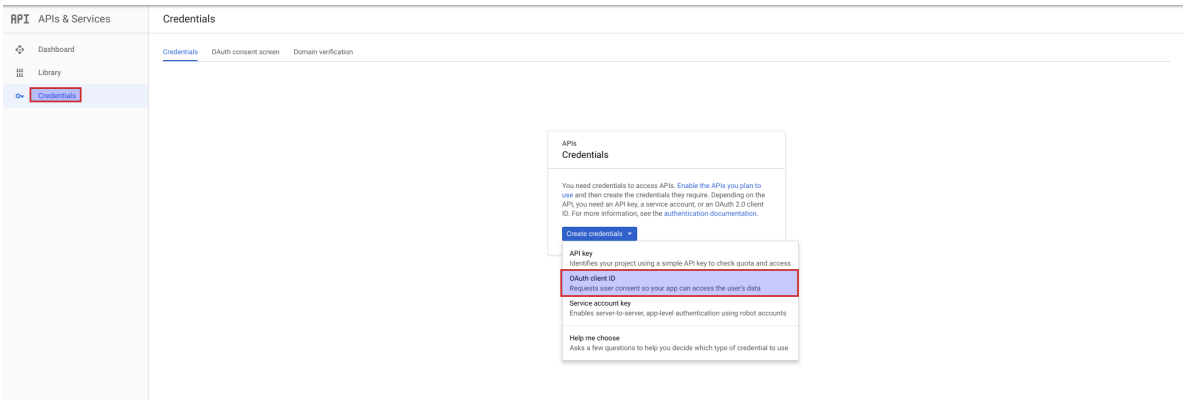
[Create a developer account with Google](#) if you do not have one and start [by configuring a project](#) with name TestAuthWebsite and product with name TestAuthWebsite. Configure your OAuth client “Web browser” and use the Amazon Cognito Domain as an Authorized JavaScript origin.



Now, if you do not have a Google Cloud account, optionally create a new project.

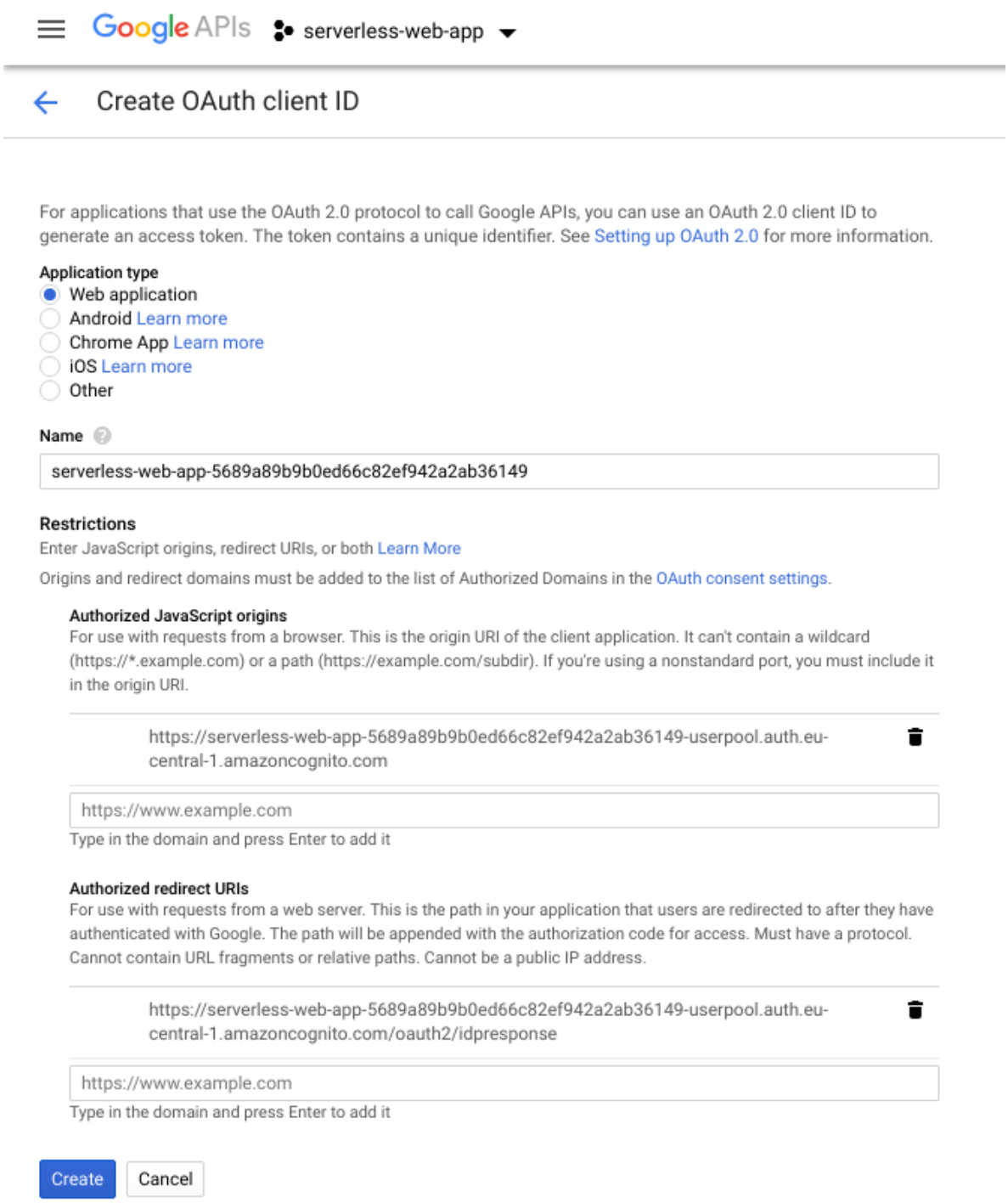


Create your OAuth2.0 credentials:



To create an OAuth client ID, you must first set a product name on the consent screen "Configure consent screen", add the application name then the authorized domain (which is your pool domain).

After finishing the configuration of the consent screen, proceed by creating the OAuth2 credentials.

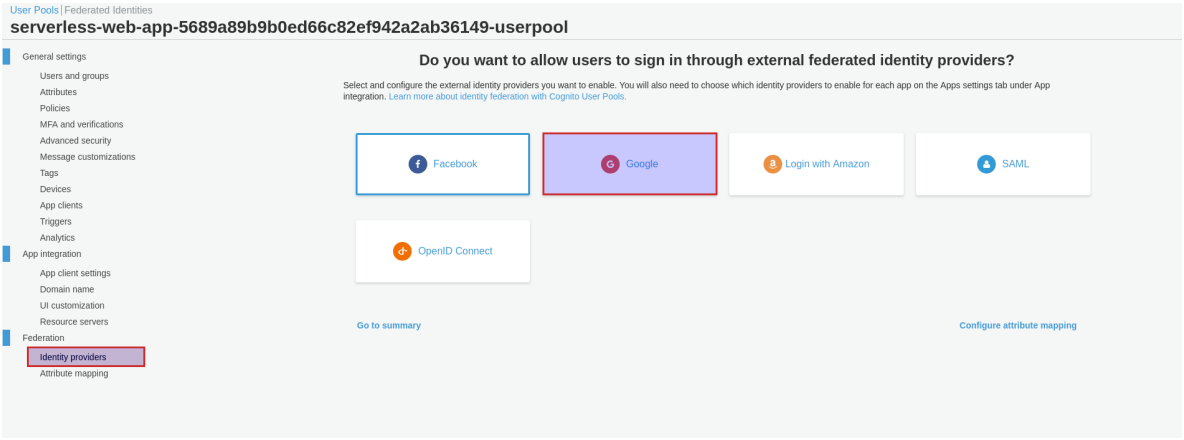


Note that the "Authorized redirect URIs" should be filled with `https://<your_pool_domain>/oauth2/idpresponse`

After finishing the creation of the client ID, Google will generate a client ID and a client secret:

- client ID: `456711906645-pe91g2skhrivo1pfmaa5845h3f3kubd1.apps.googleusercontent.com`
- Client Secret: `FtlA56Z5GerijkXLIQnU0FFP`

Go back to the AWS Cognito dashboard and add a social IdP to the user pool, choose Google.



Fill in the information with your Client ID and Client Secret then type the names of the scopes that you want to authorize.

Scopes define which user attributes; like `name`, `profile` and `email`; you want to access your application.

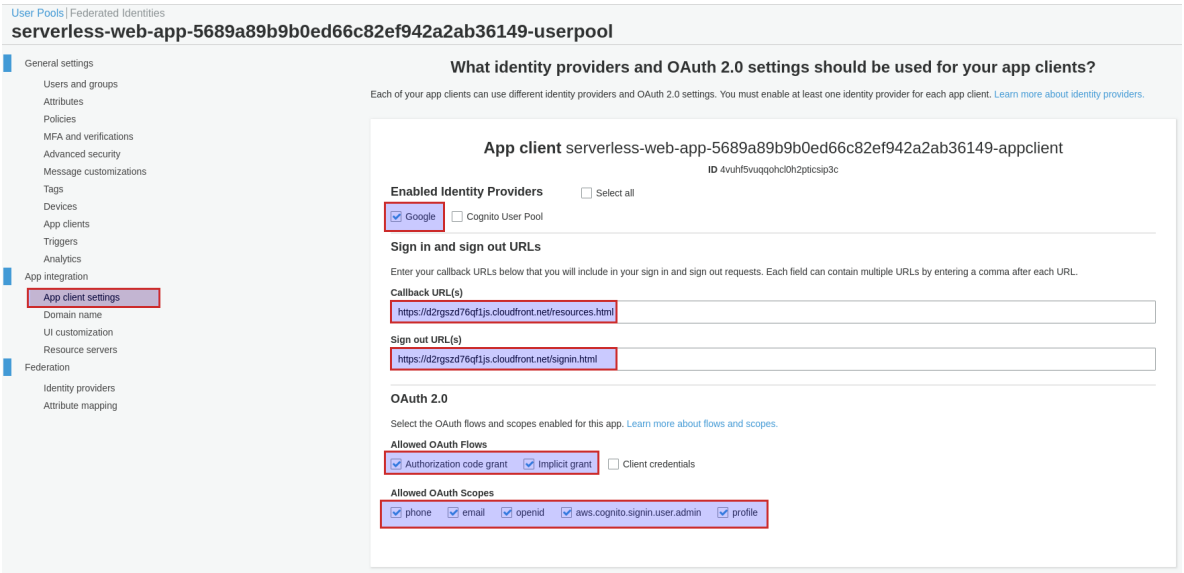
Note: With Facebook, the scopes should be separated by commas while for Google and Amazon, they should be separated by spaces.

Let's keep `profile`, `email`, and `openid` as the scopes for our application.

Enter the tab “App client settings", then add the callback and the sign-out URLs.

In the example we are building, when the user will sign in they should be redirected to the private resources page.

When they sign out, the user should be redirected to the sign-in page.



For the moment, allow Google to access all the scopes and for the flows, be careful to tick both “Authorization code grant” and “implicit grant”.

Note: The OAuth 2.0 specification describes five grants for acquiring an access token:

- **Client credentials grant:** Used if the application is the resource owner
- **Authorization code grant:** Used if the application is a web app executing on the server
- **Resource owner credentials grant:** Used if the application is trusted with user credentials
- **Implicit grant:** Used if the client is user-agent-based like single page web apps and when the client can’t keep a client secret because its code and storage are easily accessible.
- **Refresh token grant:** Used to enable the client to get a new access token without requiring the user to be redirected when the access tokens expire.

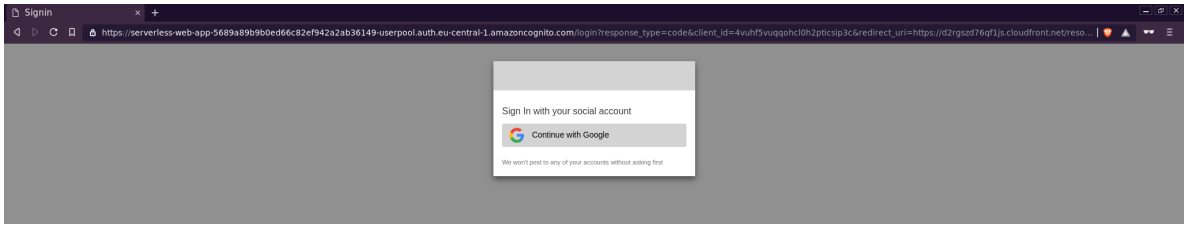
Save and test your setup with following link:

```
https://<your_user_pool_domain>/login?response_type=code&client_id=
<your_cognito_client_id>&redirect_uri=<callback_URL>
```

In the actual example, the URL will look like the following:

```
serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-userpool**.auth.eu-central-
1.amazoncognito.com/login?
response_type=code&client_id=**4vuhf5vuqqohcl0h2pticsip3c**&redirect_uri=**https
://d2rgszd76qf1js.cloudfront.net/resources.html
```

If you do not have any problem and if there is nothing missing out here, you will be redirected to the login page:



If you authenticate using your Google profile, you will be redirected to the resources page:

```
https://d2rgszd76qf1js.cloudfront.net/resources.html?code=<auto_generated_code>
```

Building the Application Backend

DynamoDB at a Glance

DynamoDB is the fully managed key-value and document database on AWS. You can deploy a multi-region and multi-master database with features like security, backup, and restore. It can be used in several use cases like serverless web apps, microservices datastore, mobile backend, and IoT.

DynamoDB will not work well when you need to make cross-table transactions, or complex queries like joins or when you want to time series database.

Every database technology has its use cases and DynamoDB is optimized when you need to use a document oriented data model, when you want easy operations and scalability, when auto-sharding and auto-scaling is required, and especially when you want to integrate with other AWS services such as Lambda.

Building the Application Backend

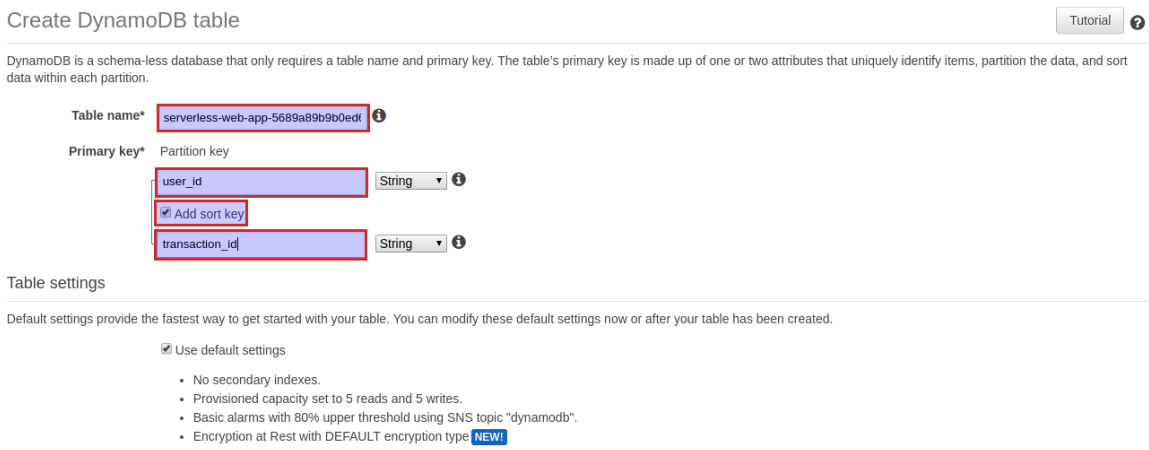
Creating the Database

Enter into AWS service DynamoDB and click on “Create Table”

Fill in Table name `serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149`, which means we create a DynamoDB table.

Note: The table name should be unique.

Fill in Primary key `user_id`, tick on `Add sort key`, fill in `transaction_id`, leave the default for the type.



Click on “Create”, and wait for the creation of the table.

Building the Application Backend

Populating the Database

The next step is adding records/rows (called items in DynamoDB) to the table.

For example, we want to add items like the following:

```
Item: {
    partition_key: 1,
    sort_key: 10,
    name: "Jane Doe",
    email: "janedoe@mail.com",
    status: "active",
    folders: "private-1"
}

Item: {
    partition_key: 2,
    sort_key: 11,
    name: "Jane Doe",
    email: "janedoe@mail.com",
    status: "active",
    folders: "private-2"
}

Item: {
    partition_key: 3,
    sort_key: 12,
    name: "Jane Doe",
    email: "janedoe@mail.com",
    status: "active",
    folders: "private-3"
}
```

Select the table, click on the “Items” tab. After that, you can fill in attributes of the item by clicking on “+” symbol then select “append”. Make every attribute of type “String”. Click on “Save”.

You can also load our items into a DynamoDB table using the following AWS command line

```
aws dynamodb batch-write-item  --request-items file://table_items.json
```

The loaded data must be in JSON format. More information can be found on the following page.

Building the Application Backend

Using Lambda to Serve Data From DynamoDB

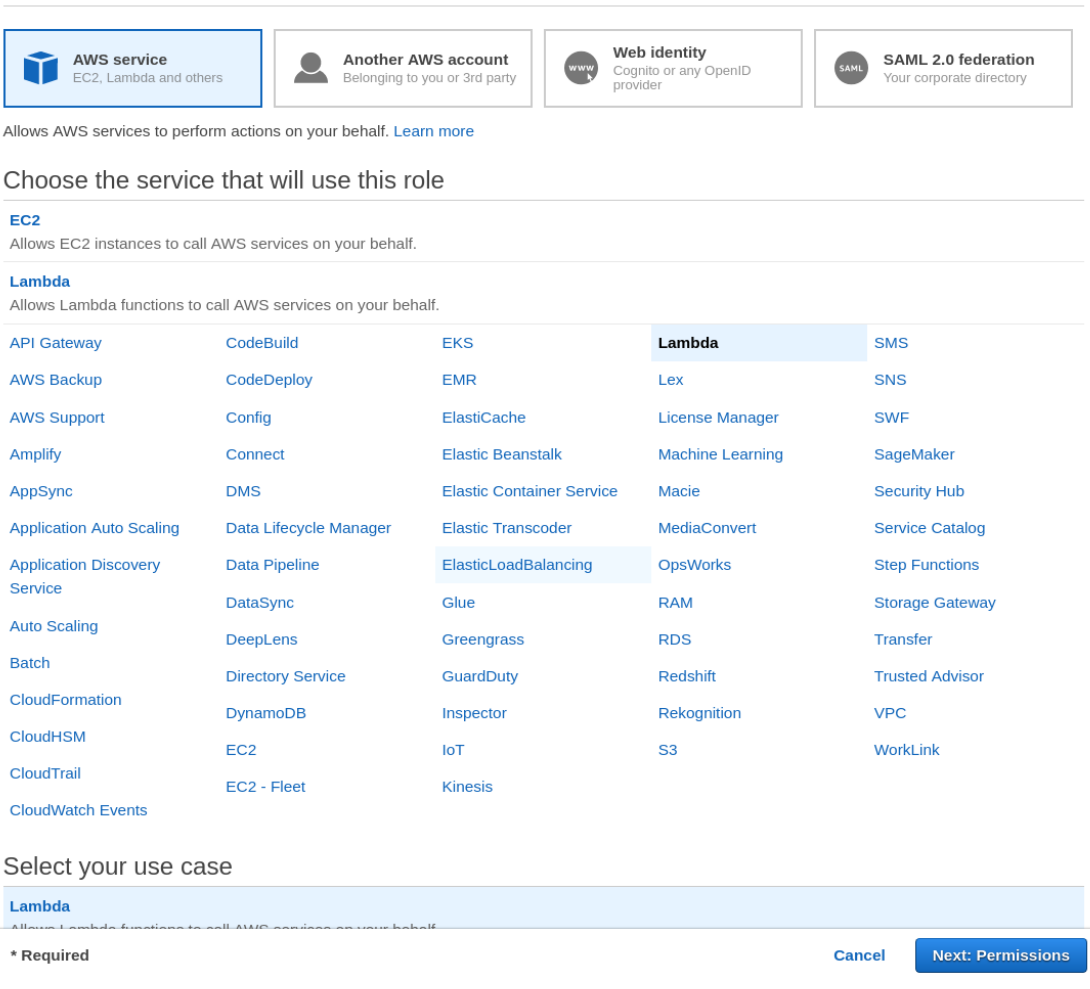
In this part, we are going to create a Lambda function to search for user folder access rights stored in the DynamoDB users table.

For instance, in the following example, Jane Does has access to the folder "private-1" since the user is active and the "private-1" folder appears in the folders list.

```
Item: {
    partition_key: 1,
    sort_key: 10,
    name: "Jane Doe",
    email: "janedoe@mail.com",
    status: "active",
    folders: "private-1"
}
```

First, let's create a role for Lambda to execute our code.

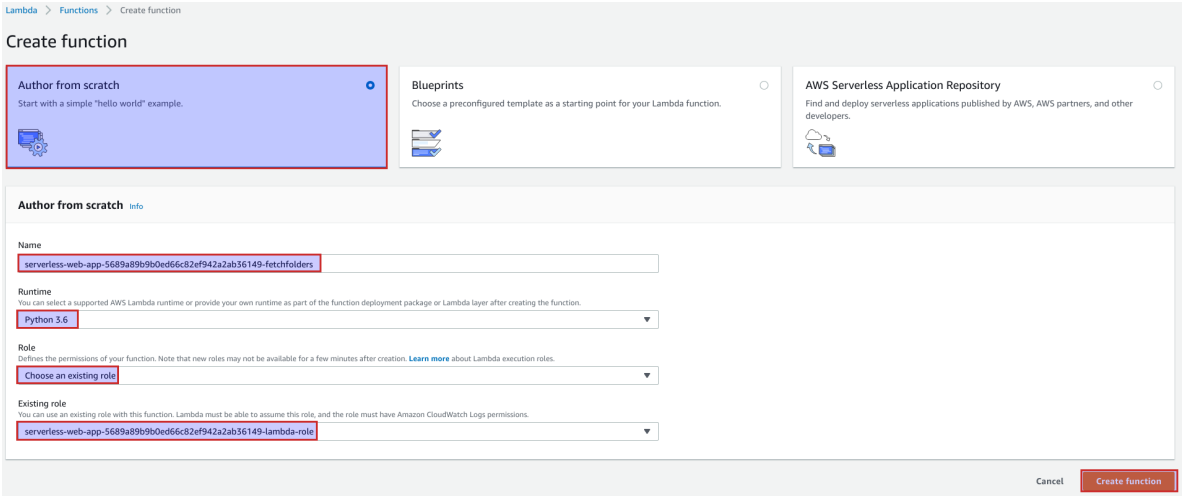
- Enter AWS service IAM
- Select “Roles” and click on “Create role”
- Choose “Lambda” as a service to use this role by clicking on it



- Click on “Next: Permission”
- Choose two policies “AmazonDynamoDBReadOnlyAccess” and “AWSLambdaBasicExecutionRole” by ticking the box in front of both of them
- Click on “Next: Tags”
- We do not set tags, so directly click on “Next: Review”
- Fill in the role name, give it a meaningful name. In this example, we will call it "serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-lambda-role"
- Click on “Create role”

We are done with IAM. Go to the Lambda dashboard.

- Enter into AWS service Lambda
- Click on “Create Function”
- Select “Author from scratch” tab
- Name the lambda function “serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-fetchfolders”
- Set Runtime “Python 3.6”
- Set role “Choosing an existing role”, which is “lambda_dynamodb_basic_role”
- Click on “Create function”



- Fill in the source code
- Set Handler to “lambda_function.lambda_handler”
- Set timeout to “10s”
- Leave the other default settings
- Click on “Save”

This is the Python Serverless function we are going to use:

```
import boto3
import json
from botocore.exceptions import ClientError
from boto3.dynamodb.conditions import Key

# Perform a scan operation on the table.
# Can specify filter_key (col name) and its value to be filtered.
# This gets all pages of results. Returns list of items.
def scan_table_allpages(table, filter_key=None, filter_value=None):
    if filter_key and filter_value:
        filtering_exp = Key(filter_key).eq(filter_value)
        response = table.scan(FilterExpression=filtering_exp)
    else:
        response = table.scan()
```

```
    items = response['Items']
    while True:
        print(len(response['Items']))
        if response.get('LastEvaluatedKey'):
            response =
table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
            items += response['Items']
        else:
            break

    return items


def lambda_handler(event, context):
    # create dynamodb client
    dynamodb = boto3.resource('dynamodb', region_name='eu-central-1')
    # get Users table. This needs to be the same as the table you aready created
    users = dynamodb.Table('Users')
    result = ""
    try:
      # get user_name information from event lambda input
      name = event["queryStringParameters"]["email"]
      response = scan_table_allpages(users, filter_key="email",
filter_value=name)
      # get a list of private folders that are active
      result = [item["folders"] for item in response if
item["status"]=="active"]
      print(result)

    except ClientError as e:
      print("Error")

    # include Access-Control-Allow-Origin header to allow CORS
    return {
        'statusCode': 200,
        'headers': {"Access-Control-Allow-Origin":"*"},
        'body': json.dumps(result)
    }
```

Remember that this is the structure of an item from our database:

```
Item: {
    partition_key: 1,
    sort_key: 10,
    name: "Jane Doe",
    email: "janedoe@mail.com",
    status: "active",
    folders: "private-1"
}
```

Note: We are using "status" to tell our app if the access is active or no longer active for the corresponding user.

The above function will scan all the pages of a table to return the good records in the function of the filter key name. It will return a 200 response with `Access-Control-Allow-Origin` header that allows the response to be publically accessed. The data returned in the response contains the folder that a user can access.

Building the Application Backend

The API Gateway

The AWS API Gateway is a managed service that allows us to easily deploy an API. All the phases of the API lifetime can be managed using this service like creating, publishing, maintaining, monitoring, and securing.

We have also the choice of creating a REST API or WebSocket APIs that act as a gateway to applications and data deployed on the cloud or on-premise.

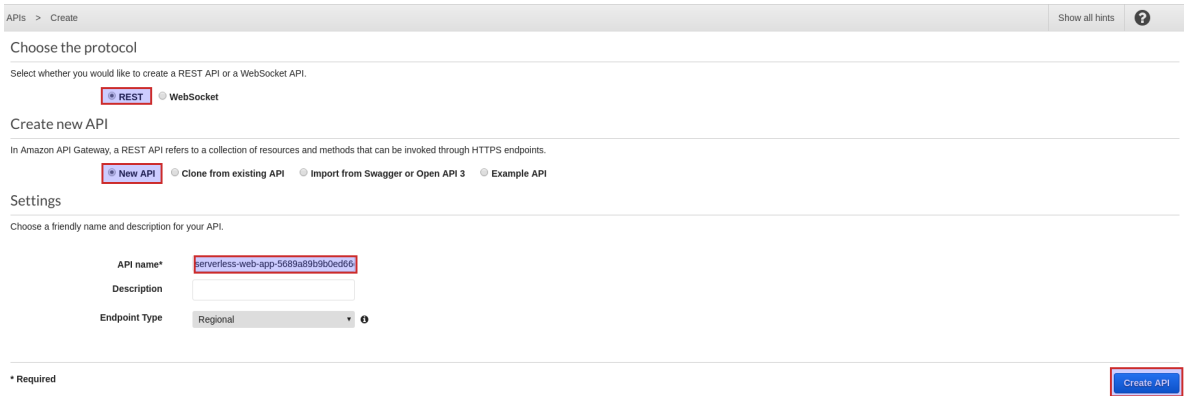
Since AWS API Gateway can be integrated with many AWS services like EC2, Lambda, Kinesis, DynamoDB, the choice of exposing our app via AWS API Gateway is obvious.

Building the Application Backend

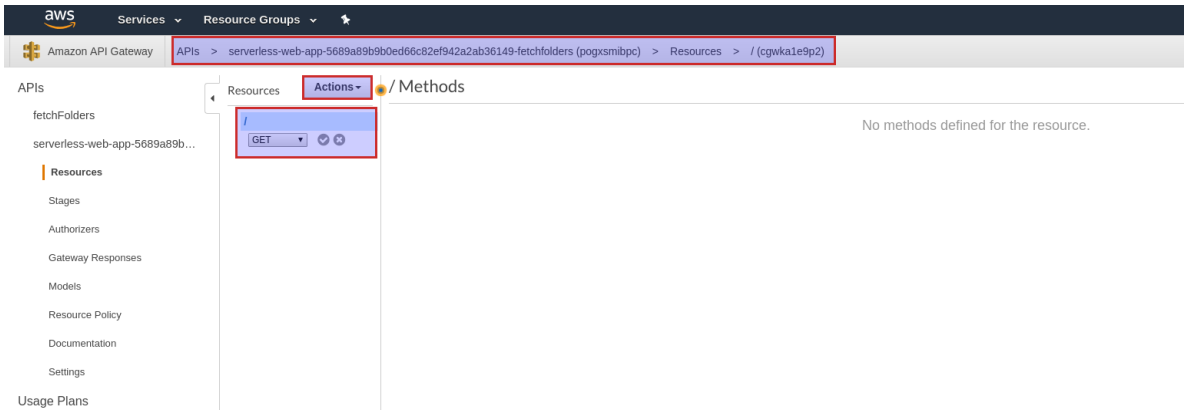
Creating a Restful API and Connecting to Lambda

To create an API, follow these steps:

- Enter into AWS service API Gateway
- Select “REST” then “New API”, fill in API name “fetchFolders”
- Click on “Create API”. In my example, the API is called: `serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-fetchfolders`



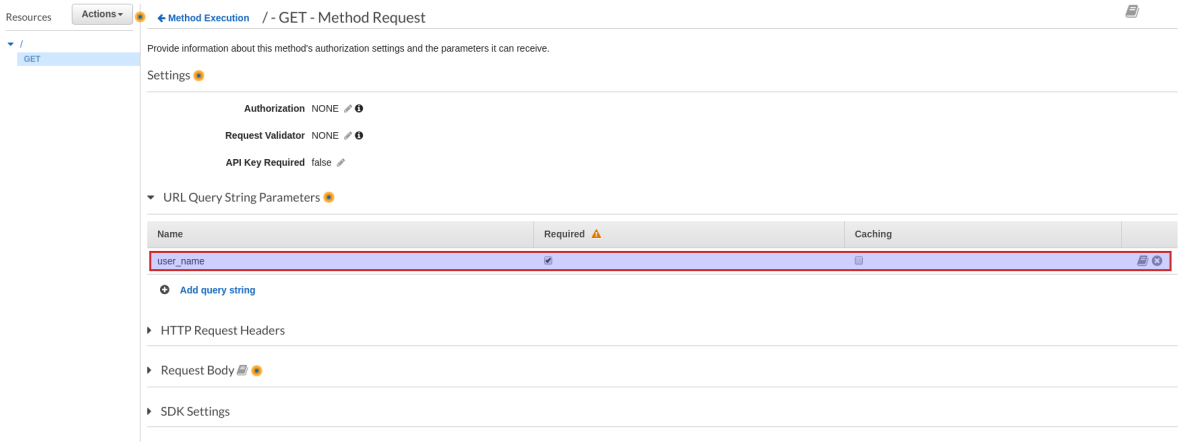
- After the creation of the REST API, let's create a Method
- Add a "GET" operation under the root path



For the integration of our "GET" operation, use the Lambda function we created before this step. Do not forget to tick "Use Lambda Proxy Integration".

- Click on save and confirm the lambda permissions.

- Add the URL Query String Parameter that we will call `user_name`: Click on Method `GET` -> Method Request -> URL Query String Parameters -> Add query string name -> tick `Required` checkbox.



Building the Application Backend

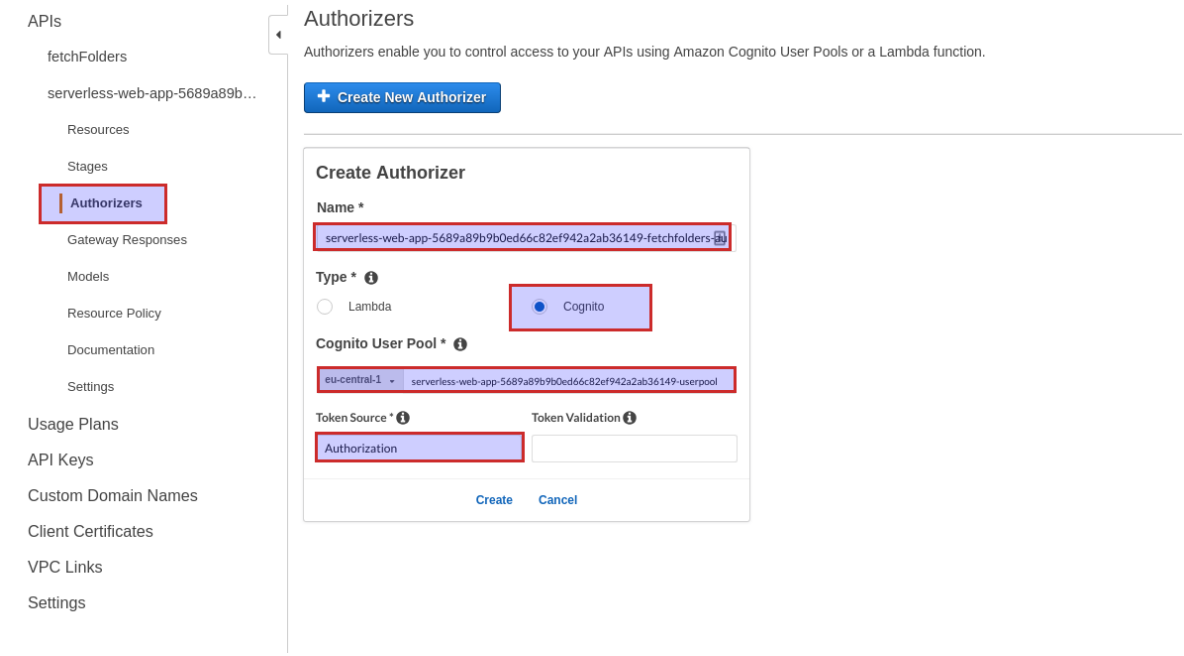
Securing the API with Cognito Authorizer

Let's secure access to the API by adding an authorizer.

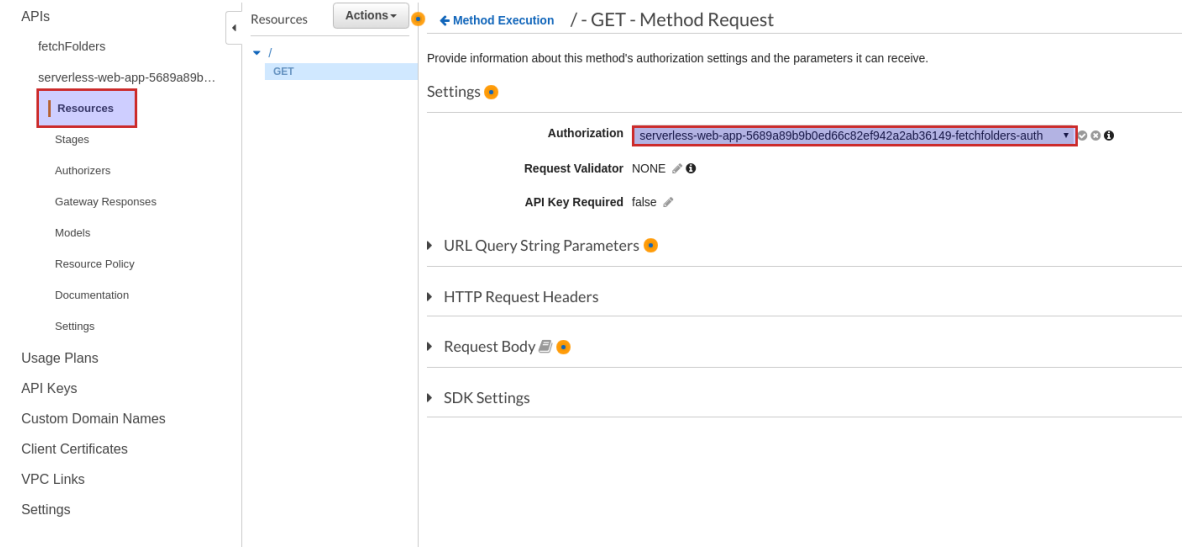
- Select “Authorizers” -> “Create Authorizer”
- Fill in Name, Cognito User Pool Id, Token Source

In my example, the name is `serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-fetchfolders-auth`.

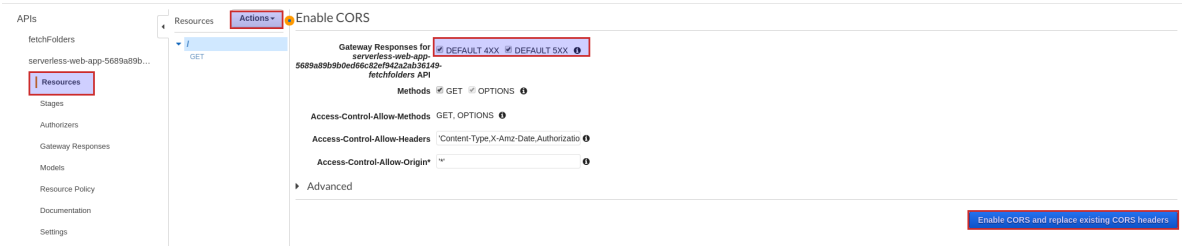
Choose the good Cognito pool and add `Authorization` as a Token Source. The `Authorization` header will be sent from the request to the Cognito user pool.



Now, add the authorizer to Resource method:



Enable CORS by going to Resources -> Actions -> Enable CORS.



The latest operation will perform the following operations:

✔ Create **OPTIONS** method

✔ Add **200 Method Response** with **Empty Response Model** to **OPTIONS** method

✔ Add **Mock Integration** to **OPTIONS** method

✔ Add **200 Integration Response** to **OPTIONS** method

✔ Add **Access-Control-Allow-Headers, Access-Control-Allow-Methods, Access-Control-Allow-Origin Method Response Headers** to **OPTIONS** method

✔ Add **Access-Control-Allow-Headers, Access-Control-Allow-Methods, Access-Control-Allow-Origin Integration Response Header Mappings** to **OPTIONS** method

✔ Add **Access-Control-Allow-Headers, Access-Control-Allow-Methods, Access-Control-Allow-Origin Response Headers** to **DEFAULT 4XX Gateway Response**

✔ Add **Access-Control-Allow-Headers, Access-Control-Allow-Methods, Access-Control-Allow-Origin Response Headers** to **DEFAULT 5XX Gateway Response**

✔ Add **Access-Control-Allow-Origin Method Response Header** to **GET** method

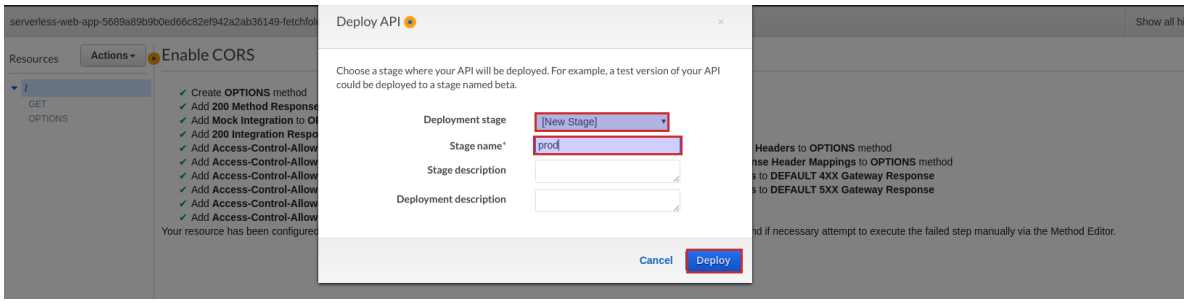
✔ Add **Access-Control-Allow-Origin Integration Response Header Mapping** to **GET** method

Building the Application Backend

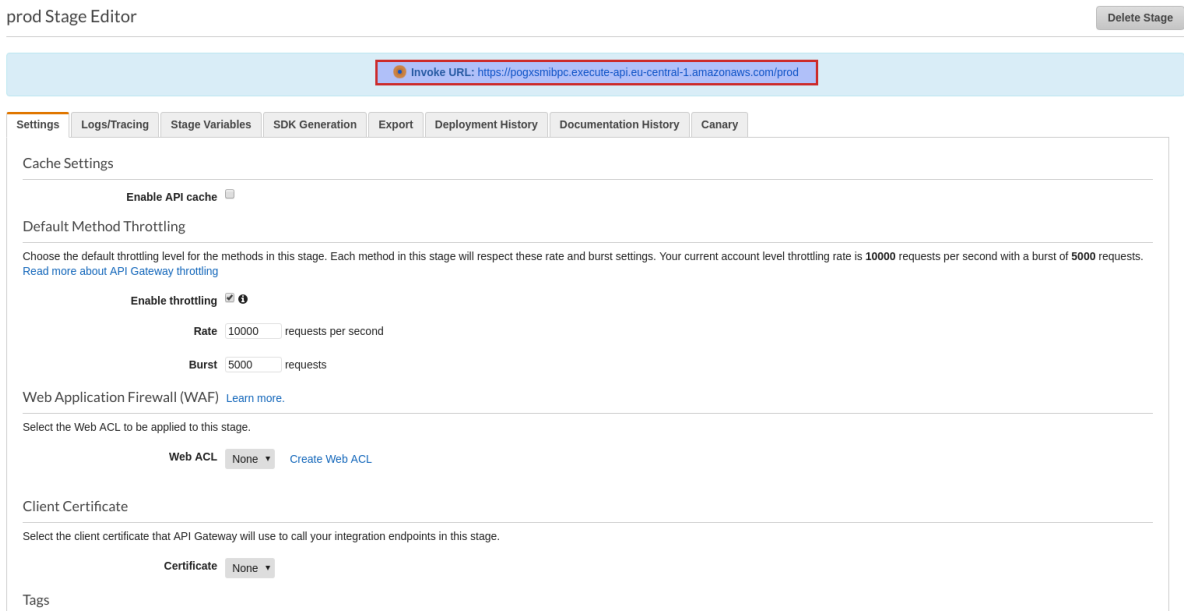
Deploying the API

We should now deploy our API, otherwise it will not be accessible. To deploy it, follow the next steps:

- Click on “Resources” tab -> Click on “Actions” -> Deploy API
- Deploy to “prod” stage



After the deployment, get the invoke URL of the Rest API through which the API can be visited from outside.



In my case, the URL is: `https://pogxsmibpc.execute-api.eu-central-1.amazonaws.com/prod`

If you visit your URL using your browser, you will get a similar response to:

```
{"message": "Internal server error"}
```

Building the Application Frontend

Folder Structure

To understand how the API Gateway + the Cognito pool and all of our Lambda functions work, we will create a simple frontend application that will call these resources using the flow we set up.

This is the folder structure of the HTML files and JS libraries we use:

```
-- signup.html
-- signin.html
-- resources.html
-- lib
------ amazon-cognito-auth.min.js
------ amazon-cognito-identity.min.js
------ aws-cognito-sdk.min.js
-- scripts
------ get-resources.js
------ sign-in.js
------ sign-up.js
```

In folder `lib/`, we store the public JS libraries.

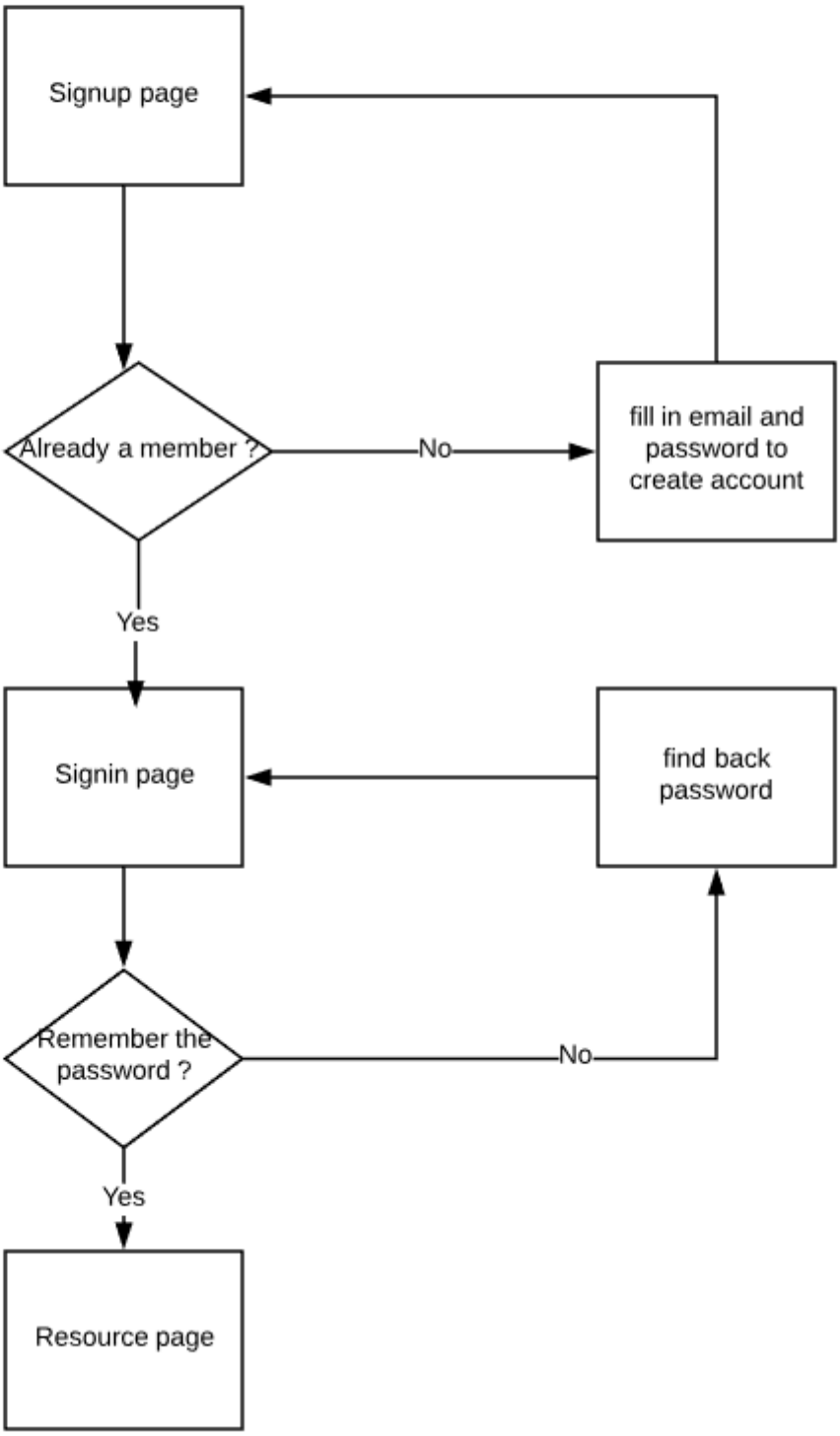
In folder `scripts/`, we store the user-defined JS.

All three HTML pages will be shown when the user-type in the domain name of our service in the browser.

Building the Application Frontend

The Frontend Logic

The logic is quite simple. The user who wants to access private resources should be authenticated. Cognito manages the different operations related to authentication with the help of Lambda as seen in the backend part of this course.


Signup page
Already a member ?
No
fill in email and password to create account
Yes
Signin page
find back password
Remember the password ?
No
Yes
Resource page

Building the Application Frontend

The Signup Page

In this part as well as the following one, we are going to get a quick look at the frontend source code.

This is the basic HTML page for the signup:

```
<html>
  <header>
    <script src="lib/aws-cognito-sdk.min.js"></script>
    <script src="lib/amazon-cognito-identity.min.js"></script>
  </header>

  <body>
    <h1>Sign Up</h1>
    <form>
      Email:<br>
      <input type="text" name="email" id="email"><br>
      Password:<br>
      <input type="password" name="password" id="password"><br>
      </br>
      </br>
      <input type="submit" value="Create Account" id="signup">
    </form>
    <a href="signin.html">Already a Member ?</a>
    <script src="scripts/sign-up.js"></script>
  </body>
</html>
```

We are going to use Bootstrap to enhance the UI. Our new code will become slightly different :

```
<!DOCTYPE html>
<html>
<header>
  <script src="lib/aws-cognito-sdk.min.js"></script>
  <script src="lib/amazon-cognito-identity.min.js"></script>
  <!-- Latest compiled and minified CSS -->
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css">
  <!-- jQuery library -->
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
  <!-- Latest compiled JavaScript -->
  <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js">
</script>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</header>
```

```
<body>
  <div class="container-fluid">
    <h1>Sign Up</h1>
    <form>
      Email:<br>
      <input type="text" name="email" id="email"><br>
      Password:<br>
      <input type="password" name="password" id="password"><br>
      </br>
      </br>
      <input type="button" value="Create Account" id="signup">
    </form>
    <a href="signin.html">Already a Member ?</a>
  </div>
    <script src="scripts/sign-up.js"></script>
  </body>
</html>
```

Note: You should substitute AWS resource id and links with your own, for example, use variables `user_pool_id` and `client_app_id`.

To make JS work correctly on the HTML page, you have to download aws-cognito-sdk.min.js, and amazon-cognito-identity.min.js.

Note: In order to avoid any compatibility issues, use the same files and versions mentioned above. However, you can test the latest versions, using NPM.

These libraries are stored in the `lib` directory and called from the HTML page:

```
  <script src="lib/aws-cognito-sdk.min.js"></script>
  <script src="lib/amazon-cognito-identity.min.js"></script>
```

Create a `lib` directory and download the libraries:

```
mkdir lib && cd lib

wget https://github.com/amazon-archives/amazon-cognito-identity-
js/raw/master/dist/aws-cognito-sdk.min.js \
https://raw.githubusercontent.com/amazon-archives/amazon-cognito-identity-
js/master/dist/amazon-cognito-identity.min.js
```

In `<form></form>` block, two input boxes are defined to fetch user input from the web browser and use them as information for the sign-up operation.

`<input type="button" value="Create Account" id="signup">` is a button on which event listener is registered through:

```
document.getElementById('signup').onclick = function () {
  //code here
};
```

This means as long as the element with id "signup" (our “Create Account”) is clicked, the function on the right side of the equation will be triggered.

```
<a href="signin.html">Already a Member ?</a>
```

`<a></a>` block is a hyperlink, it will jump to the “sign-in page” when clicked.

Do not forget to create the `scripts/sign-up.js` script and change the AWS resources.

```
var user_pool_id = 'eu-central-1_A0xCdzSim';
var client_id = '4vuhf5vuqqohcl0h2pticsip3c';

// call function if signup button on signup.html page is clicked
document.getElementById('signup').onclick = function () {
  // cognito user pool data
  var poolData = {
    UserPoolId : user_pool_id,
    ClientId : client_id
};

// You can find the javascript SDK cognito authentication example here
https://docs.aws.amazon.com/cognito/latest/developerguide/using-amazon-cognito-
user-identity-pools-javascript-examples.html
// create an cognito user pool object
var userPool = new AmazonCognitoIdentity.CognitoUserPool(poolData);

// get user input from <input> box
var attributeList = [];
var email = document.getElementById('email').value ;
var password = document.getElementById('password').value ;
// build attributeList which is required parameter for signUp function
var dataEmail = {
    Name : 'email',
    Value : email
};

var attributeEmail = new AmazonCognitoIdentity.CognitoUserAttribute(dataEmail);
attributeList.push(attributeEmail);

// sign up the user
// if fail an alert will be shown
// if success user name will be outputed in console
var cognitoUser;
userPool.signUp(email, password, attributeList, null, function(err, result){
      // pop up alert if signup error
      if (err) {
          alert(err);
          return;
      }

      // log the signed up username
      cognitoUser = result.user;
      console.log('user name is ' + cognitoUser.getUsername());
  });
}
```

Building the Application Frontend

The Sign-in Page

Open a HTML file called `signin.html` and add the following code:

```
<!DOCTYPE html>
<html>
<header>
  <script src="lib/aws-cognito-sdk.min.js"></script>
  <script src="lib/amazon-cognito-identity.min.js"></script>
  <!-- Latest compiled and minified CSS -->
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css">
  <!-- jQuery library -->
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
  <!-- Latest compiled JavaScript -->
  <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js">
</script>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</header>

<body>
  <div class="container-fluid">
    <h1>Sign In</h1>
    <form>
      Email:<br>
      <input type="text" name="email" id="email"><br>
      Password:<br>
      <input type="password" name="password" id="password"><br>
      </br>
      </br>
      <input type="button" value="Sign In" id='signin'>
    </form> <br>
    <button type="button" id="forgetpassword">Forget Password</button> <br>
    <a href="https://serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149-
userpool.auth.eu-central-1.amazoncognito.com/login?
response_type=code&client_id=4vuhf5vuqqohcl0h2pticsip3c&redirect_uri=https://d2r
gszd76qf1js.cloudfront.net/resources.html">Use Google Auth</a><br>
    <a href="signup.html">Sign Up</a>
  </div>
    <script src="scripts/sign-in.js"></script>
   </body>
 </html>
```

Now let's create the `scripts/sign-in.js` file called by our sign-in page.

```
document.getElementById('signin').onclick = function (){
  // fetch user input from form
```

```
  var email = document.getElementById('email').value ;
  var password = document.getElementById('password').value ;
  // construct authenticationData object
  var authenticationData = {
      Username : email,
      Password : password,
  };
  // construct authenticationDetails object
  var authenticationDetails = new
AmazonCognitoIdentity.AuthenticationDetails(authenticationData);
  // construct cognito user pool data
  var poolData = { UserPoolId : 'eu-central-1_A0xCdzSim',
      ClientId : '4vuhf5vuqqohcl0h2pticsip3c'
  };
  var userPool = new AmazonCognitoIdentity.CognitoUserPool(poolData);
  // construct cognito user object
  var userData = {
      Username : email,
      Pool : userPool
  };
  localStorage.setItem("name", email);

  // authenticating user process
  var cognitoUser = new AmazonCognitoIdentity.CognitoUser(userData);
  cognitoUser.authenticateUser(authenticationDetails, {
      onSuccess: function (result) {
          var accessToken = result.getAccessToken().getJwtToken();
          console.log(accessToken)
          /* Use the idToken for Logins Map when Federating User Pools with
identity pools or when passing through an Authorization Header to an API Gateway
Authorizer*/
          var idToken = result.idToken.jwtToken;
          // store token information locally
          localStorage.setItem("accessToken", accessToken);
          localStorage.setItem("idToken", idToken)
          // by success show a list of resources
          window.location.href="resources.html";
      },
      onFailure: function(err) {
          alert(err);
      }
  });
}

// forget password process
// verification code will be sent through registered email
// two prompt to confirm password reset
document.getElementById("forgetpassword").onclick = function(){

  var email = document.getElementById('email').value;
  // construct cognito user pool data
  var poolData = { UserPoolId : 'eu-central-1_A0xCdzSim',
      ClientId : '4vuhf5vuqqohcl0h2pticsip3c'};
  var userPool = new AmazonCognitoIdentity.CognitoUserPool(poolData);
  var userData = {
      Username : email,
      Pool : userPool
  };
```

```
    var cognitoUser = new AmazonCognitoIdentity.CognitoUser(userData);

    cognitoUser.forgotPassword({
      onSuccess: function (result) {
          console.log('call result: ' + result);
      },
      onFailure: function(err) {
          alert(err);
      },
    inputVerificationCode() {
          var verificationCode = prompt('Please input verification code ' ,'');
          var newPassword = prompt('Enter new password ' ,'');
          cognitoUser.confirmPassword(verificationCode, newPassword, this);
      }
});
}
```

Don't forget to substitute AWS resource IDs and links with your own in the sign-in script `scripts/sign-in.js`.

As we have already learned, the “Sign-in page” will take the email and the password from `<input></input>` form box.

Click on “Forget Password” and “Sign In” button will trigger the process of `forgotPassword` and `authenticateUser` defined in `signin.js`.

A successful login leads to a resource page on which a list of resources that the user has access to will be shown.

Building the Application Frontend

The Resources Page

Create a file called `resources.html` and add the following code:

```
<!DOCTYPE html>
<html>
<header>
  <!-- Latest compiled and minified CSS -->
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css">
  <!-- jQuery library -->
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
  <!-- Latest compiled JavaScript -->
  <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js">
</script>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</header>

<body>
  <div class="container-fluid">
    <h1>Private Resources</h1>
    <ul id="myItemList"> </ul>
  </div>
    <script src="scripts/get-resources.js"></script>
  </body>
</html>
```

This is the code of the `scripts/get-resources.js` file:

```
// output what stored in local Storage
console.log(localStorage.getItem("idToken"));
console.log(localStorage.getItem("accessToken"));
console.log(localStorage.getItem("name"));

var name = localStorage.getItem("name");
const url = 'https://pogxsmibpc.execute-api.eu-central-1.amazonaws.com/prod?
email='+name;

// make http requst to get folders list
fetch(url, {
  method: "GET",
  headers: {
    "Authorization" : localStorage.getItem("idToken"),
    "Content-Type": "application/json",
  },
}).then(function(response) {
    console.log("Success");
```

```
    return response.json();
})
.then(function(items) {
    // get list object on resources.html
    var ul = document.getElementById('myItemList');

    items.forEach(function (item) {
      var a = document.createElement('a');
      var linkText = document.createTextNode(item);
      a.appendChild(linkText);
      a.title = item;
      a.href = "https://d2rgszd76qf1js.cloudfront.net/private/" + item;

      var node = document.createElement('li');
      node.appendChild(a);
      document.getElementById("myItemList").appendChild(node);
    });
})
.catch(function() {
    console.log("Failed")
});
```

Note: Do not forget to adapt the above code with your own configurations.

Building the Application Frontend

Serving Private Files From S3

The goal is always serving the private files to the users who have access to. Remember that users are stored in the Cognito database and have access rights to specific folders based on the DynamoDB datastore.

Create some public and private folders to test this. This is the file tree of the final project:

```
.
├── lib
│   ├── amazon-cognito-identity.min.js
│   └── aws-cognito-sdk.min.js
├── private-1
│   ├── private-11.html
│   └── private-12.html
├── private-2
│   └── private-21.html
├── private-3
│   └── private-31.html
├── public
│   └── public-1.html
├── resources.html
├── scripts
│   ├── get-resources.js
│   └── sign-up.js
├── signin.html
└── signup.html
```

Then upload everything to the S3 bucket we created for the web app.

```
cd <your code folder>
aws s3 sync . s3://serverless-web-app-5689a89b9b0ed66c82ef942a2ab36149
```

Visit the CloudFront distribution URL and will be able to test the sign-in page, login and start playing with your app:

```
https://d2rgszd76qf1js.cloudfront.net/signin.html
```

To test your application, add your own gmail or google account email to the database, add certain private folders you have the right to access to then login using the front page and check if you are able to access these files.

Conclusion

AWS is a leader in Serverless services hence our choice to create a course about Lambda and Lambda@Edge.

As a developer, you can easily prototype and deploy your application then deploy it by following simple steps in a few minutes.

The fact that integrating your Serverless app with other AWS services is easy, makes Lambda a popular service.

However, Serverless is not the technology you can use in every use case. For instance, if you have long-running functions then the Serverless architecture will not help you: Each serverless instance of Lambda can run for a limited time. AWS Lambda can run for 5 minutes.

Other use case may not suitable, so beware of the pitfalls of Serverless computing.