Lab Exercise 5

Lab Title: Refactoring User Data Export API to Asynchronous with AWS SQS, Lambda, DynamoDB, and Google Sign-In

Duration:
2–3 hours

Overview:

In this lab, students will create an authenticated and secure asynchronous data export system using Next.js and AWS. Initially, students will create an API that sends messages to a queue using AWS SQS (Simple Queue Service). They will then create an endpoint in this API to process data synchronously by simulating a long running process in the endpoints code.

Following this, they will then refactor this API to handle export requests asynchronously by creating a Lambda that also simulates a long running process, triggering each time a message is added in the queue. The exported data will be displayed on a front-end that the student will develop, and all actions will be logged in DynamoDB. Students will also be required to pass OAuth tokens when testing the API locally. Integration of Google Sign-In for OAuth 2.0 authentication will ensure that only authorised users can access the API.

Lab Overview:

Detailed Steps: (Please fill in here)

Project Setup

1. **Installing Node.js**
    - As we are creating a project using Next.js, we will need Node installed on our machines
    - **Setup of Next.js Application:**
        - Create a new project using Node
        - Run the newly created project locally

2. **Editing your Node.Js Application:**
    - **Create an AWS Queue:**
        - Set up a queue using AWS SQS (Simple Queue Service)
    - **Create an asynchronous endpoint:**
        - Create an asynchronous endpoint in our project API
        - Install the AWS-SDK
        - Ensure that this endpoint pushes messages to the AWS queue
    - **Set up AWS Credentials:**
        - Obtain your AWS Root User access keys
        - Create a .env file to store your access keys
    - **Testing the Endpoint:**
        - Test the asynchronous endpoint in Postman
    - **Initial Frontend Implementation:**
        - Modify your existing project front-end to display a button that will invoke our asynchronous API endpoint
3. **Authentication Integration:**
    - **Create OAuth Consent Screen:**
        - Develop data consent screen for users who want to sign in with Google
    - **Create OAuth Credentials:**
        - Set up your OAuth Client ID and Client Secret
    - **Install & Configure NextAuth:**
        - Install NextAuth
        - Add your OAuth Client ID and Client Secret to your .env file
    - **Add Components for Authentication:**
        - Create a Sign In component for the front-end
        - Create a Providers component & import the providers to the layout file of our front-end
        - Create an AppBar component & import it into the layout file of our front-end
    - **Protect our Task Manager:**
        - Modify the page component of our front-end to hide the task manager when a user is not signed in
4. **Create a Synchronous Data Export API:**
    - **Create the SendMessageSync Endpoint:**
        - Create a new endpoint in the project API that exports data synchronously
    - **Test the sendMessageSync Endpoint**
        - Test the sendMessageSync endpoint in Postman
    - **Hook this endpoint to the front-end**
        - Edit the taskmanager file to add another button to invoke out synchronous API endpoint
5. **Set up a Lambda to Process Messages in our SQS Queue:**
    - **Create a Lambda in the AWS Console:**
        - Create a new Lambda function in your AWS console
    - **Change the Default Timeout of the Lambda:**

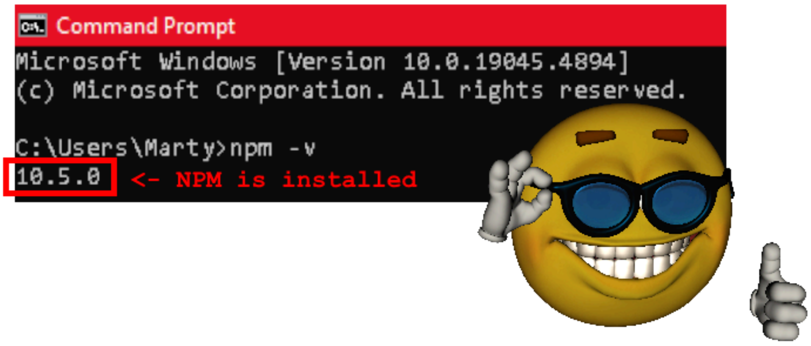
        - Change the timeout from 3 seconds to 15 seconds to accommodate long running processes
    - **Add SQS Permissions to the Lambda:**
        - Allow the Lambda to read from & write to the queue created earlier
    - **Add DynamoDB Permissions to the Lambda:**
        - Create a DynamoDB table in your AWS console
        - Allow the Lambda to read from & write to your DynamoDB table
    - **Add our Lambda as a trigger to SQS:**
        - Add a new trigger to your queue that calls your lambda function each time a message is added to the queue
    - **Add code to the Lambda:**
        - Add code to your lambda function that will..
            1. Log events and the queue message to CloudWatch
            2. Create an entry in your DynamoDB table with “In-Progress” status
            3. Simulate a long running process
            4. Update the previous DynamoDB entries status to “Complete”
            5. Return a successful response

6. **Test the Lambda, view CloudWatch logs & scan DynamoDB Table:**
    - **Test the Lambda from the front-end:**
        - Invoke the asynchronous API endpoint from the front-end
    - **View CloudWatch Logs:**
        - Observe the execution of the lambda function in the CloudWatch logs of the Lambda function
    - **View new entries in DynamoDB Table**
        - Run a scan on your DynamoDB table to see the entries added by the Lambda function
7. **Deliver Data to User on the Front-end:**
    - **Front-end Setup for Polling:**
        - Edit your taskmanager file to display the processing & complete statuses on the front-end
    - **Create Polling Endpoint:**
        - Create a checkStatus endpoint to check when the status of your dynamoDB entry
        - Return either “Processing” or “Completed” to the front-end
8. **Secure your API:**
    - **Generate a NextAuth token:**
        - Create a random Base64 string to act as our .env variable for NextAuth
    - **Create a New .Env variable:**
        - Create a NextAuth_Secret variable in the .env file
    - **Update your NextAuth Route:**
        - Modify the route.ts file in the [...nextauth] file to use the nextauth secret
    - **Update the sendMessageAsync endpoint:**
        - Modify the route.ts file of the asynchronous endpoint to check the AuthOptions for the server session
    - **Test the updated sendMessageAsync endpoint:**

- Test the endpoint in Postman before logging in (unauthorised response) and then after you login (successful response)

Step 1 Installing Node.js (5 mins):

- Head to https://nodejs.org/en to download node.js Once the installation is finished,
- Run the `npm -v` command in a terminal to ensure you have node.js installed correctly.
- The version of Node.js should print following this command. If this does not happen, Node.js has not been properly installed.



Command Prompt

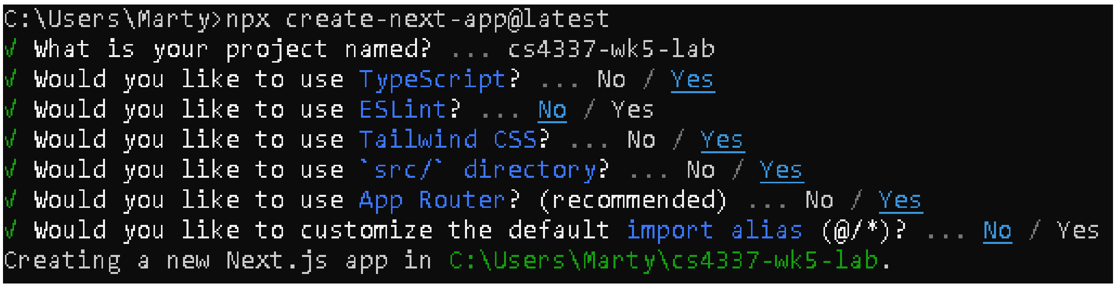
```
Microsoft Windows [Version 10.0.19045.4894]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Marty>npm -v
10.5.0  <- NPM is installed
```

Step 2 Setup of Next.js application:

- We will now set up a next.js application. To do this, open your command prompt once more and navigate to a directory where you want to set up the project.
- Then, Run the following command

```
npx create-next-app@latest
```

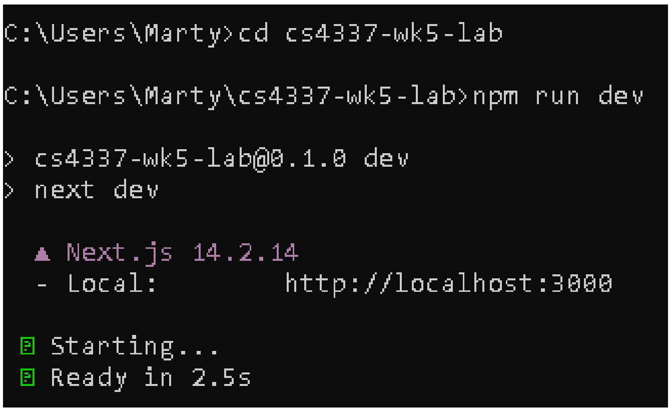
- After running this command, you will first be asked to give it a name. After that you will need to select the configuration of your next.js project. Answer the questions as follows:



```
C:\Users\Marty>npx create-next-app@latest
√ What is your project named? ... cs4337-wk5-lab
√ Would you like to use TypeScript? ... No / Yes
√ Would you like to use ESLint? ... No / Yes
√ Would you like to use Tailwind CSS? ... No / Yes
√ Would you like to use `src/` directory? ... No / Yes
√ Would you like to use App Router? (recommended) ... No / Yes
√ Would you like to customize the default import alias (@/*)? ... No / Yes
Creating a new Next.js app in C:\Users\Marty\cs4337-wk5-lab.
```

- Next, we will run the development server to ensure that we set the project up correctly. We do this by navigating into the project directory cd to your project and run

```
npm run dev
```



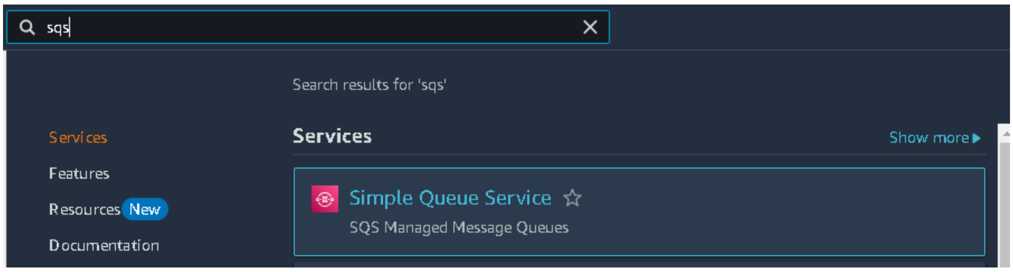
```
C:\Users\Marty>cd cs4337-wk5-lab

C:\Users\Marty\cs4337-wk5-lab>npm run dev

> cs4337-wk5-lab@0.1.0 dev
> next dev

  ▲ Next.js 14.2.14
  - Local:        http://localhost:3000

 □ Starting...
 □ Ready in 2.5s
```

- We can now run http://localhost:3000 to see the default Next.js welcome page.
- This indicates that next.js is running successfully

Step 2 Editing your Node.Js Application (10 mins):

2.1:

- Head to AWS, we will need to set up an SQS queue.



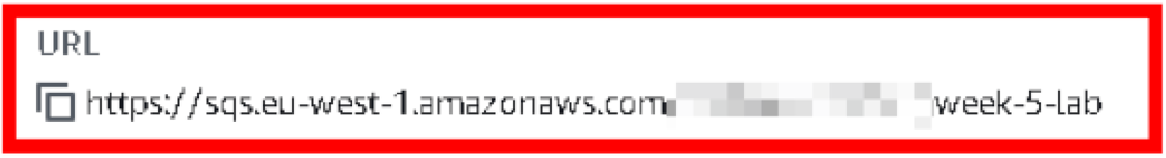
Next, Select create queue

Create a name for this queue, take note of it. Leave the rest of the settings to the default ones provided by aws. Then scroll to the bottom, create the queue.

On the next screen, you will get info about your queue, take note of the URL associated with your queue. You will need this for the next step.

Type

Standard

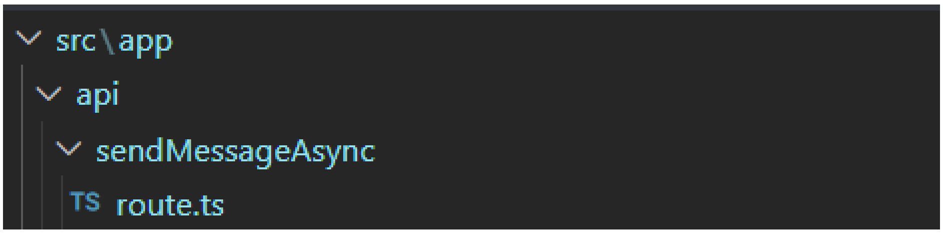


URL

https://sqs.eu-west-1.amazonaws.com[redacted]week-5-lab

2.2:

- Under the app folder, we will make an api folder, within this we will create folders for the various apis we wish to integrate. For now, we will create a sendMessage folder to house the endpoint for our first api.
  src/app/api/sendMessageAsync



```
src\app
  api
    sendMessageAsync
      TS route.ts
```

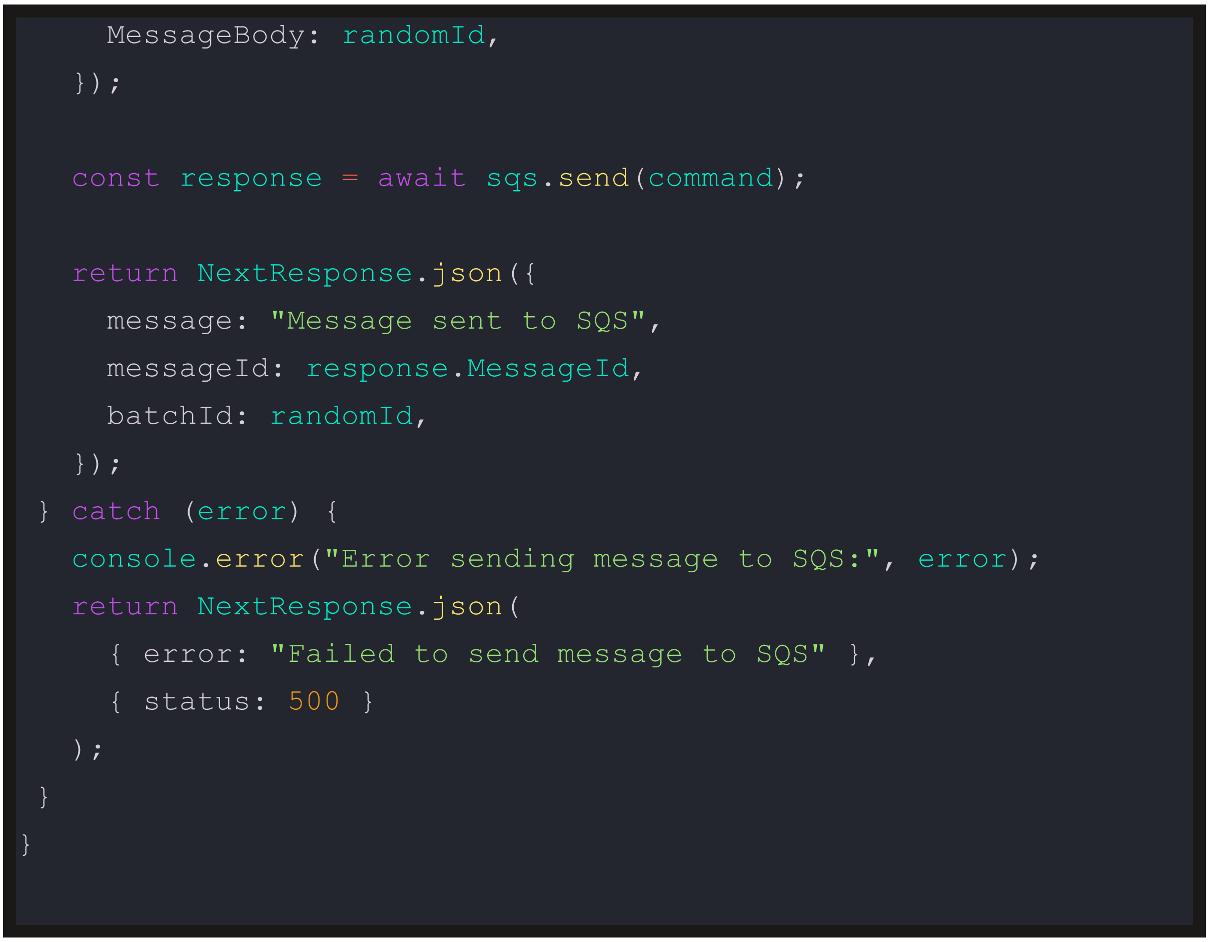
- Within this sendMessageAsync folder, we will create a new typescript file. **route.ts**

- We need to install the amazon sdk for our project to work with SQS. **Run the following command in a terminal:**

```
npm install @aws-sdk/client-sqs
```

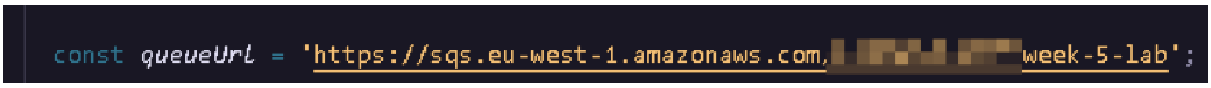
- Within the **route.ts** file, we will create the post request. Code on the next page.

```
import { SQSClient, SendMessageCommand } from
"@aws-sdk/client-sqs";
import { NextRequest, NextResponse } from "next/server";

const sqs = new SQSClient({ region: process.env.AWS_REGION});

const generateRandomLongID = () => {
 return Math.random().toString(36).substring(2, 15) +
Math.random().toString(36).substring(2, 15);
}

export async function POST(req: NextRequest) {
 try {
   const queueUrl = process.env.QUEUE_URL
   const randomId = generateRandomLongID()

   const command = new SendMessageCommand({
     QueueUrl: queueUrl,
```



```
      MessageBody: randomId,
    });

    const response = await sqs.send(command);

    return NextResponse.json({
      message: "Message sent to SQS",
      messageId: response.MessageId,
      batchId: randomId,
    });
  } catch (error) {
    console.error("Error sending message to SQS:", error);
    return NextResponse.json(
      { error: "Failed to send message to SQS" },
      { status: 500 }
    );
  }
}
```

Make sure to include the QueueURL, this is the URL for your queue from your last step. Paste it within this file.

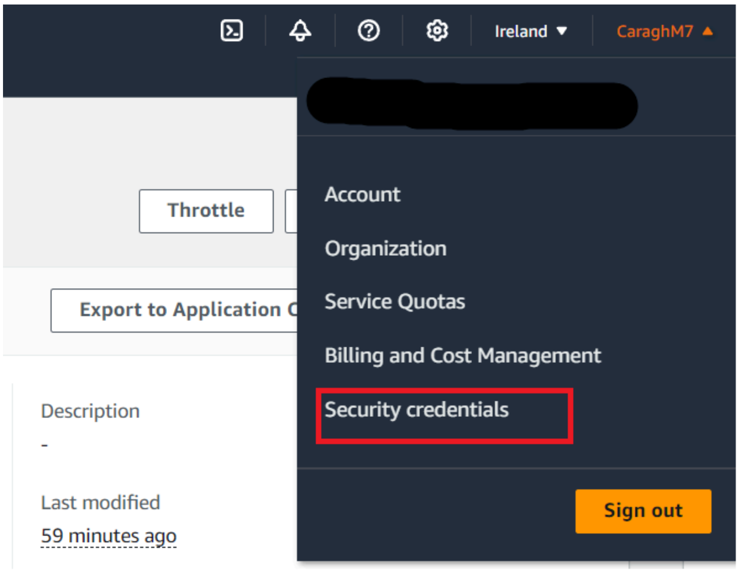


```
const queueUrl = 'https://sqs.eu-west-1.amazonaws.com/[redacted]week-5-lab';
```

2.3 Setting Up AWS Credentials

To use our queue, we must add some **credentials to our local environment** to access the resources on our AWS account.

In your AWS Management console, navigate to your account options drop down and click **Security Credentials**



Ireland ▼ CaraghM7 ▲

[redacted]

Account

Organization

Service Quotas

Billing and Cost Management

Security credentials

Sign out

Throttle

Export to Application C

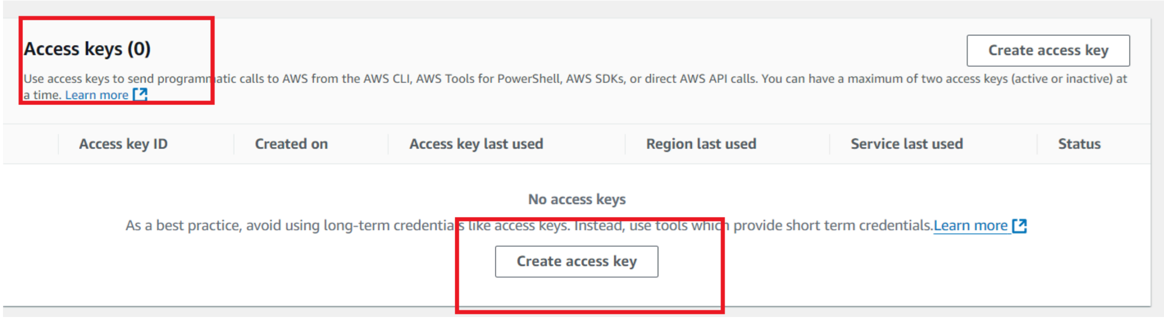
Description

-

Last modified

59 minutes ago

Then scroll to the **Access Keys section**, and create an access key



Access keys (0)

Use access keys to send programmatic calls to AWS from the AWS CLI, AWS Tools for PowerShell, AWS SDKs, or direct AWS API calls. You can have a maximum of two access keys (active or inactive) at a time. Learn more

Create access key

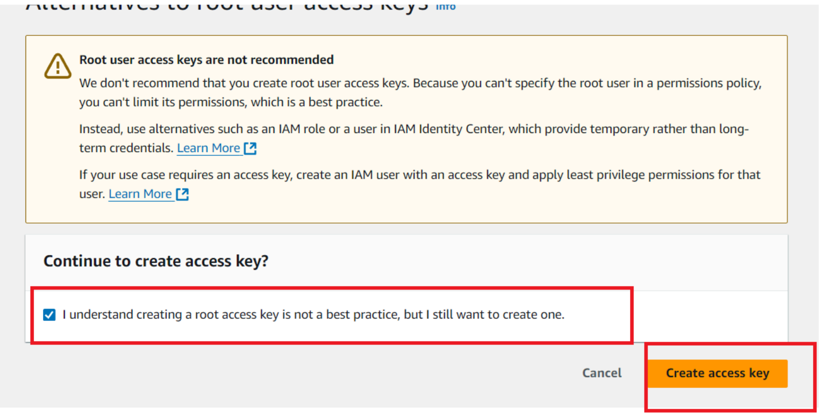
Access key ID	Created on	Access key last used	Region last used	Service last used	Status

No access keys

As a best practice, avoid using long-term credentials like access keys. Instead, use tools which provide short term credentials. Learn more

Create access key

Ignore the warnings regarding using the root user access keys, as we will delete these at the end of the lab!



Alternatives to root user access keys Info

Root user access keys are not recommended

We don't recommend that you create root user access keys. Because you can't specify the root user in a permissions policy, you can't limit its permissions, which is a best practice.

Instead, use alternatives such as an IAM role or a user in IAM Identity Center, which provide temporary rather than long-term credentials. Learn More

If your use case requires an access key, create an IAM user with an access key and apply least privilege permissions for that user. Learn More

Continue to create access key?

☑ I understand creating a root access key is not a best practice, but I still want to create one.

Cancel Create access key

Now we will see our **Access Key** and our **Secret Access key**

Don’t close this page!

- Now **create a .env file** in the **root directory** of the project (the same folder as **README.md**)

```
QUEUE_URL=
AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
AWS_REGION=eu-west-1
```

- Change the **first two values** to the **Access Key** and **Secret Access Key** you have open in your browser
- Fill in the **SQS QUEUE URL** from above into the first line

Now you are authenticated and can access your AWS Resources programmatically!

2.4 Testing your endpoint

Once that is set up, SQS should be ready to work with our next.js app. We can ensure our api is set up properly by doing the following steps:

1. Running our application

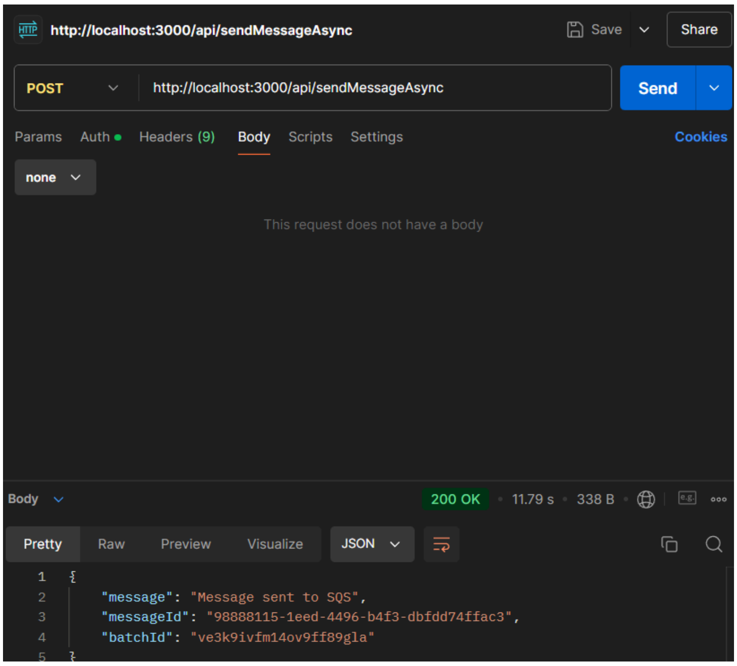
   `npm run dev` In the terminal

2. Over to postman

   Create a new **POST** request with this URL

   `http://localhost:3000/api/sendMessageAsync`

   No body required, and hit send!



http://localhost:3000/api/sendMessageAsync

POST http://localhost:3000/api/sendMessageAsync

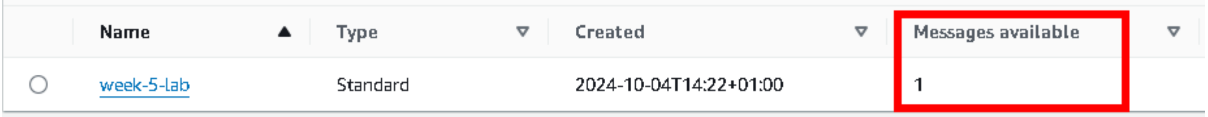
Body: none

This request does not have a body

200 OK • 11.79 s • 338 B

```
{
    "message": "Message sent to SQS",
    "messageId": "98888115-1eed-4496-b4f3-dbfdd74ffac3",
    "batchId": "ve3k9ivfm14ov9ff89gla"
}
```

You can also see the message sent to your SQS queue by heading back to SQS on AWS.

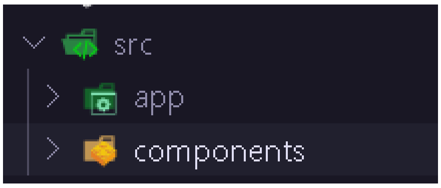


	Name	Type	Created	Messages available
○	week-5-lab	Standard	2024-10-04T14:22+01:00	1

2.5 Initial Frontend Implementation

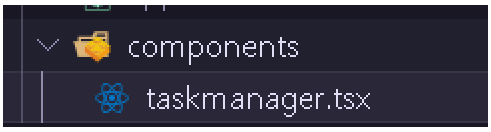
Great, we are ready to create the frontend implementation of this post request.

Lets first create a components folder within our src folder:



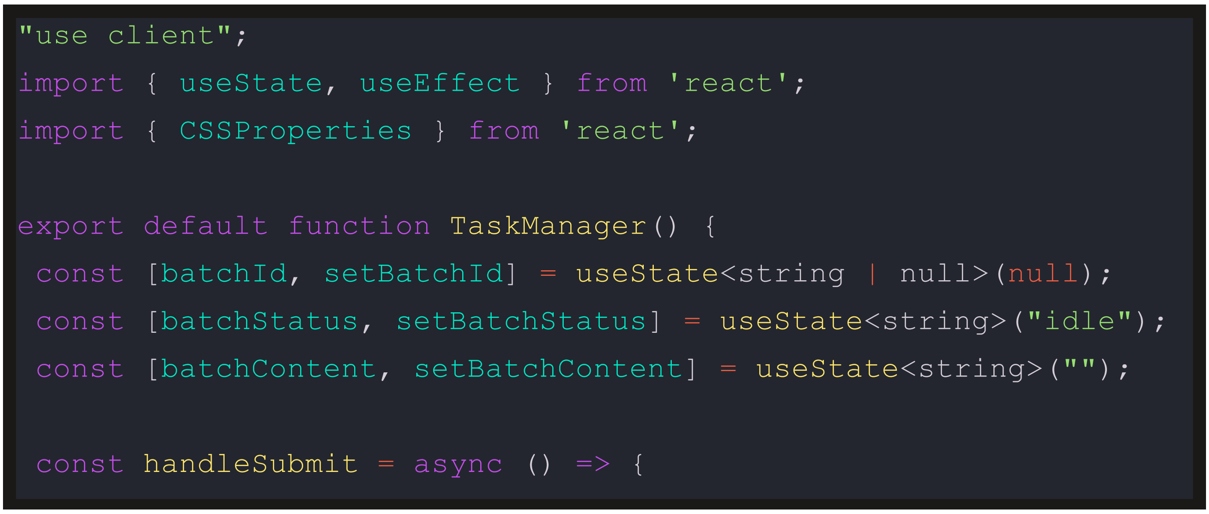
```
src
  app
  components
```

Next, create a taskmanager.tsx file within the components folder.

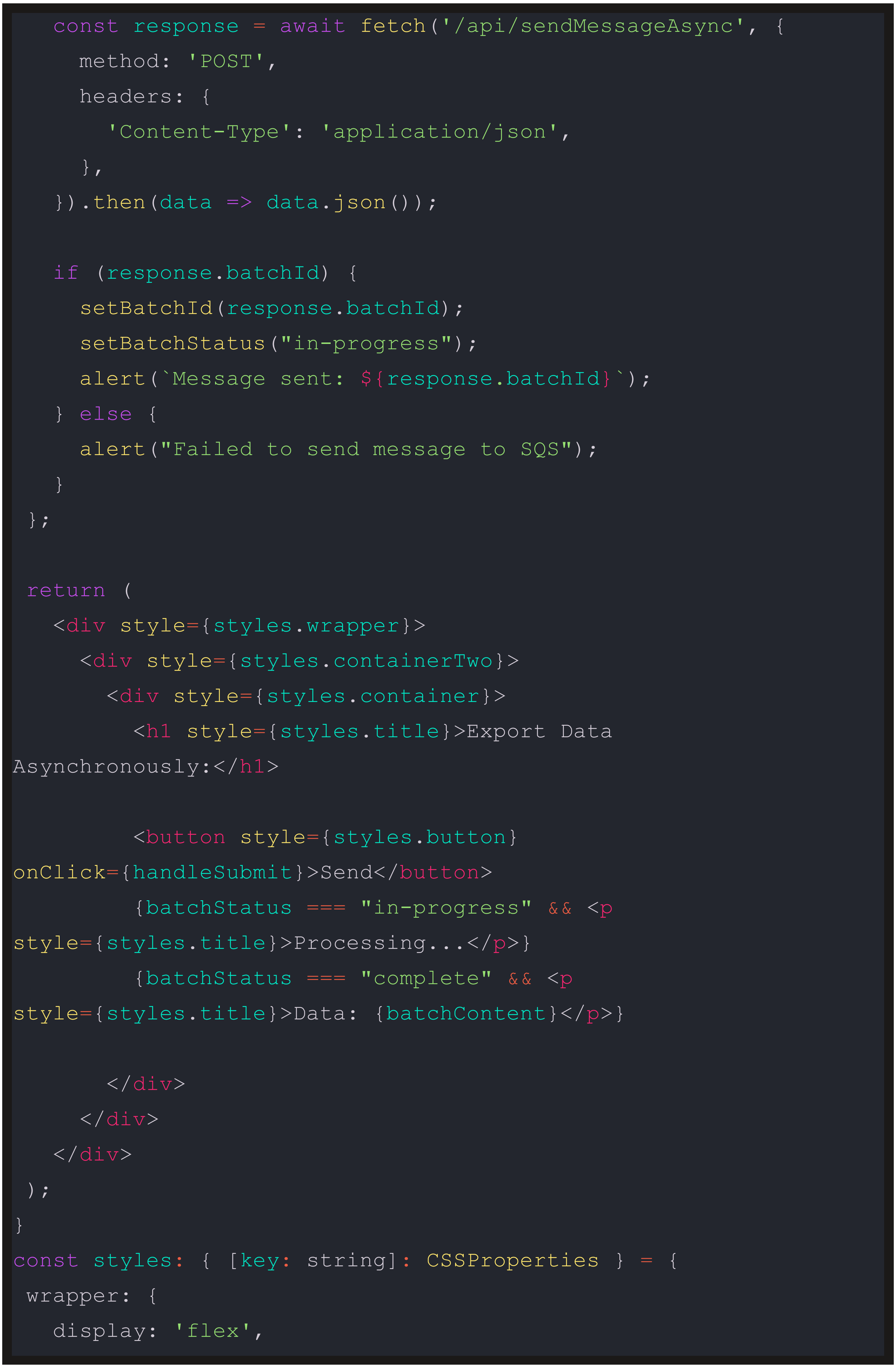


```
components
  taskmanager.tsx
```

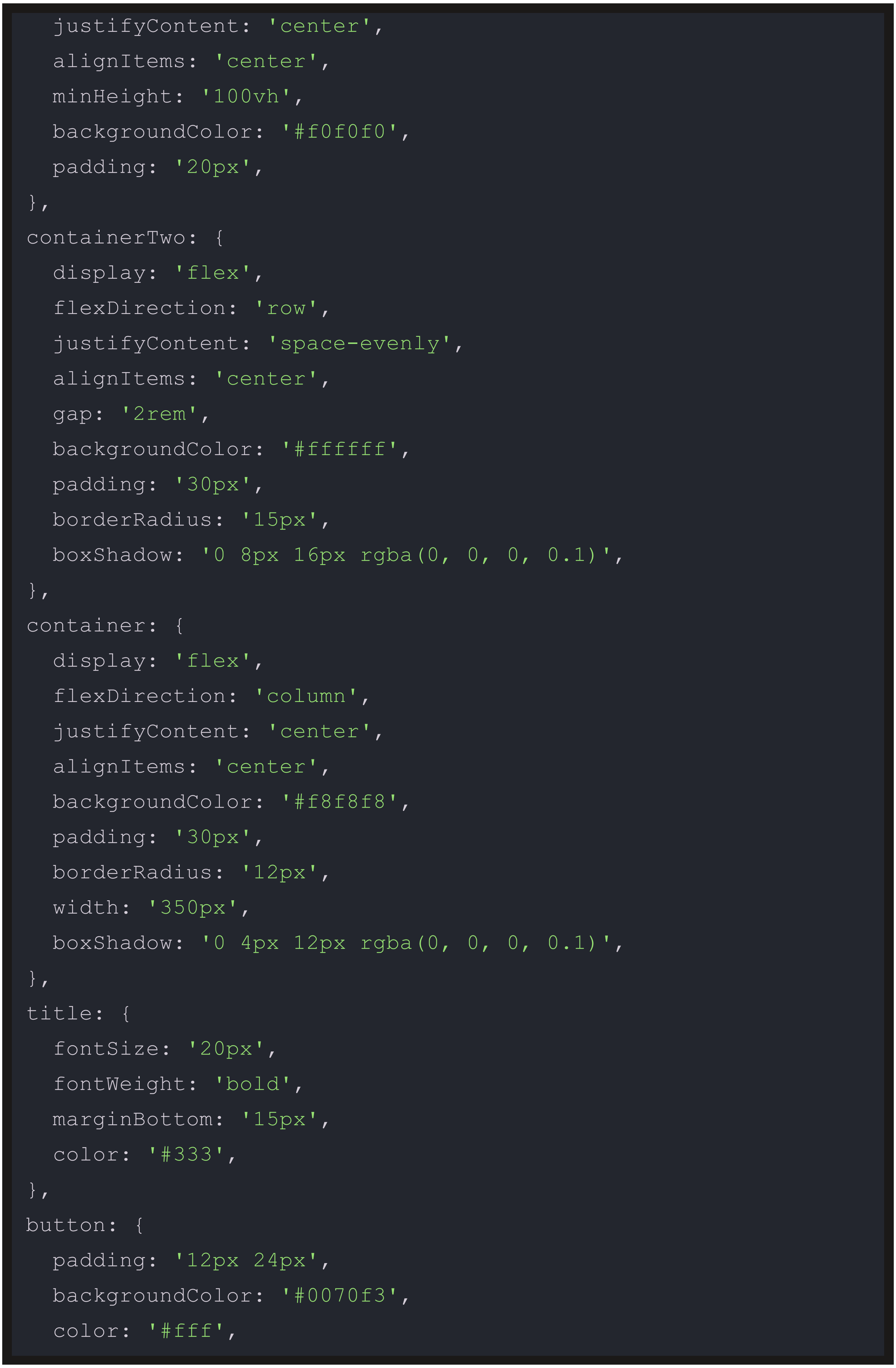
We can then create our taskmanager.tsx file:



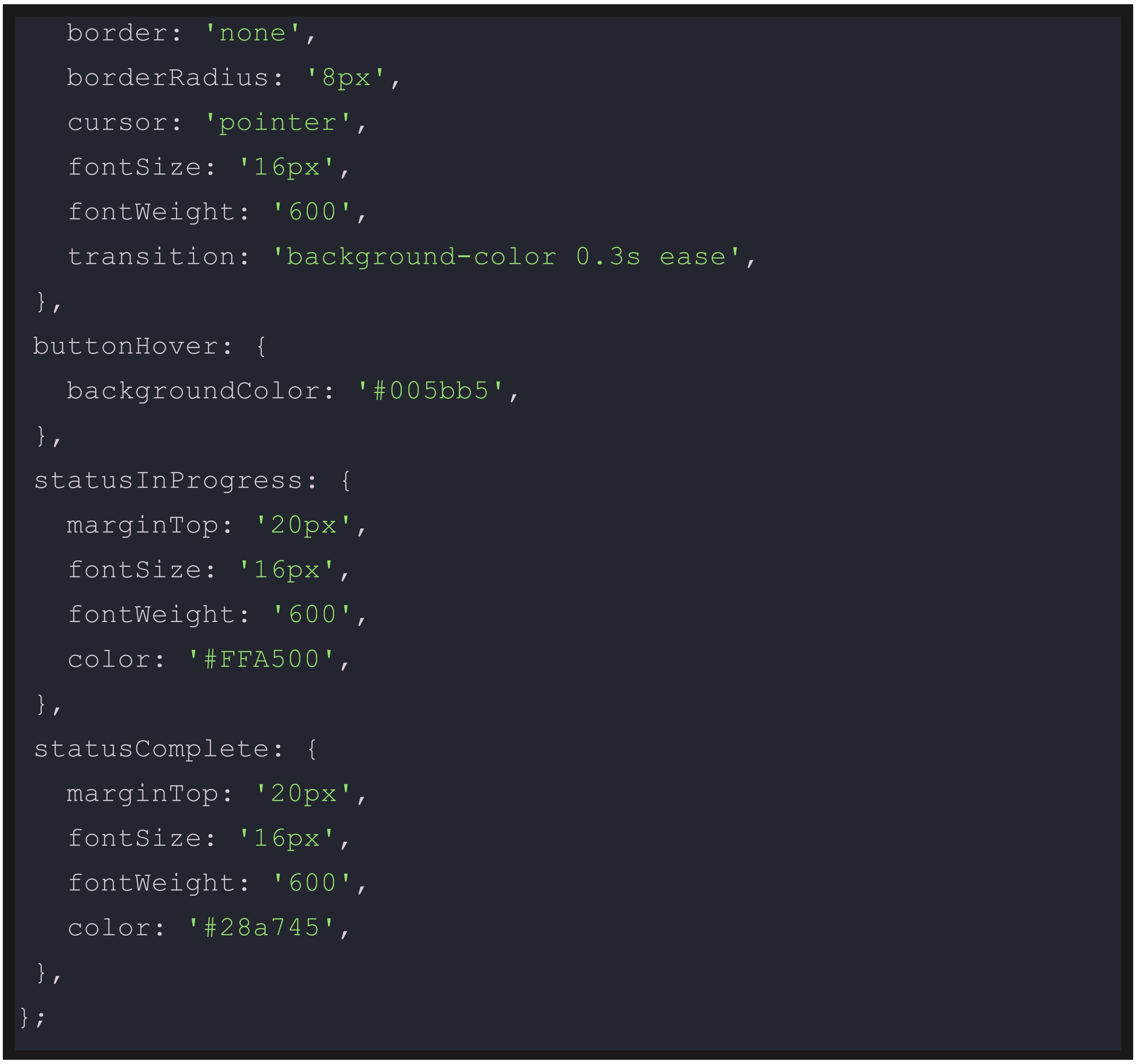
```
"use client";
import { useState, useEffect } from 'react';
import { CSSProperties } from 'react';

export default function TaskManager() {
 const [batchId, setBatchId] = useState<string | null>(null);
 const [batchStatus, setBatchStatus] = useState<string>("idle");
 const [batchContent, setBatchContent] = useState<string>("");

 const handleSubmit = async () => {
```



```
   const response = await fetch('/api/sendMessageAsync', {
     method: 'POST',
     headers: {
       'Content-Type': 'application/json',
     },
   }).then(data => data.json());

   if (response.batchId) {
     setBatchId(response.batchId);
     setBatchStatus("in-progress");
     alert(`Message sent: ${response.batchId}`);
   } else {
     alert("Failed to send message to SQS");
   }
 };

 return (
   <div style={styles.wrapper}>
     <div style={styles.containerTwo}>
       <div style={styles.container}>
         <h1 style={styles.title}>Export Data
Asynchronously:</h1>

         <button style={styles.button}
onClick={handleSubmit}>Send</button>
         {batchStatus === "in-progress" && <p
style={styles.title}>Processing...</p>}
         {batchStatus === "complete" && <p
style={styles.title}>Data: {batchContent}</p>}

       </div>
     </div>
   </div>
 );
}
const styles: { [key: string]: CSSProperties } = {
 wrapper: {
   display: 'flex',
```



```
  justifyContent: 'center',
  alignItems: 'center',
  minHeight: '100vh',
  backgroundColor: '#f0f0f0',
  padding: '20px',
},
containerTwo: {
  display: 'flex',
  flexDirection: 'row',
  justifyContent: 'space-evenly',
  alignItems: 'center',
  gap: '2rem',
  backgroundColor: '#ffffff',
  padding: '30px',
  borderRadius: '15px',
  boxShadow: '0 8px 16px rgba(0, 0, 0, 0.1)',
},
container: {
  display: 'flex',
  flexDirection: 'column',
  justifyContent: 'center',
  alignItems: 'center',
  backgroundColor: '#f8f8f8',
  padding: '30px',
  borderRadius: '12px',
  width: '350px',
  boxShadow: '0 4px 12px rgba(0, 0, 0, 0.1)',
},
title: {
  fontSize: '20px',
  fontWeight: 'bold',
  marginBottom: '15px',
  color: '#333',
},
button: {
  padding: '12px 24px',
  backgroundColor: '#0070f3',
  color: '#fff',
```



```
    border: 'none',
    borderRadius: '8px',
    cursor: 'pointer',
    fontSize: '16px',
    fontWeight: '600',
    transition: 'background-color 0.3s ease',
  },
  buttonHover: {
    backgroundColor: '#005bb5',
  },
  statusInProgress: {
    marginTop: '20px',
    fontSize: '16px',
    fontWeight: '600',
    color: '#FFA500',
  },
  statusComplete: {
    marginTop: '20px',
    fontSize: '16px',
    fontWeight: '600',
    color: '#28a745',
  },
};
```

Our first component for the frontend is complete.

To insert this into our webpage we need to insert this into the **page.tsx** file. **Src\app\page.tsx**

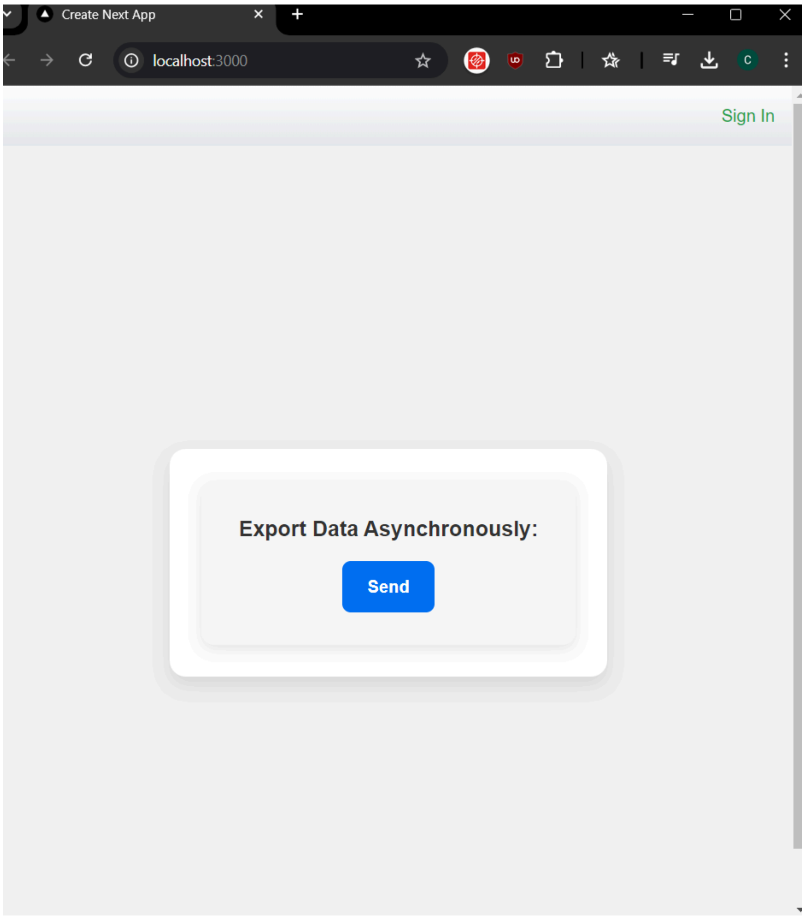
Delete the existing contents of the page.tsx file, this is just a placeholder from next.js. And insert the code below. Ensure that the import is correct to your specific filename.

page.tsx

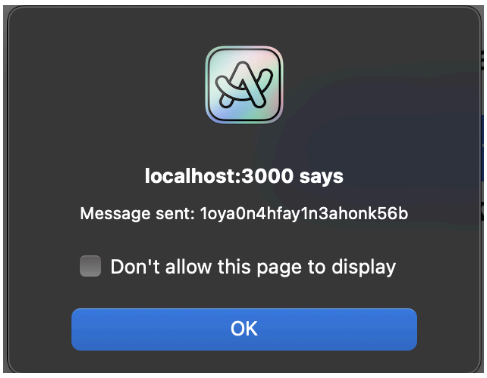
```
"use client";
import TaskManager from '../components/taskmanager';
```

```
export default function Home() {

  return (
        <TaskManager />
  );
}
```

You are now ready to `npm run dev` once more to look at your frontend.



Click Send and you should get a similar response to the one below.



localhost:3000 says

Message sent: 1oya0n4hfay1n3ahonk56b

□ Don't allow this page to display

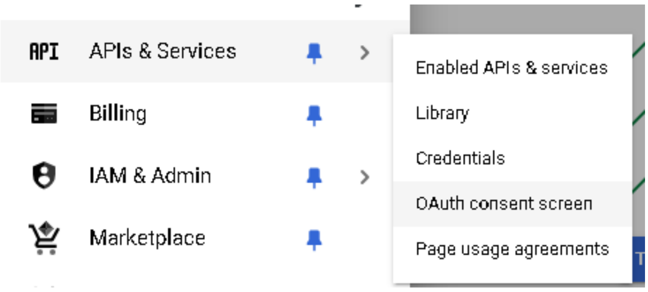
Step 3: Authentication Integration (15 mins)

Step 3.1 Creating OAuth Consent Screen:

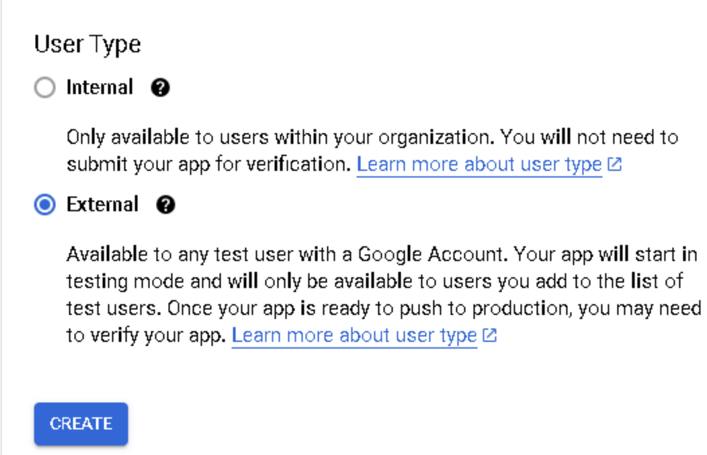
1. Go to the Google Cloud Console and create a new project.

   https://console.cloud.google.com/

2. Select your project and navigate to the OAuth consent screen from the left hand menu.



3. Set the **User Type** to **External**



User Type

○ Internal

Only available to users within your organization. You will not need to submit your app for verification. Learn more about user type

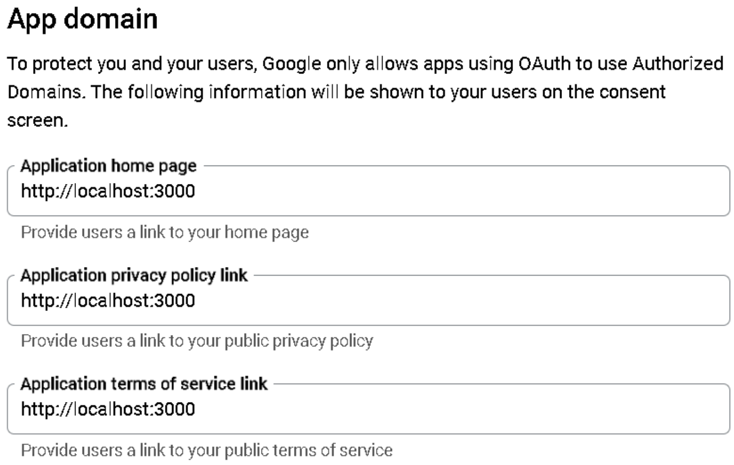
● External

Available to any test user with a Google Account. Your app will start in testing mode and will only be available to users you add to the list of test users. Once your app is ready to push to production, you may need to verify your app. Learn more about user type

4. Fill out the required fields such as
   a. App Name

b. User Support Email (Your email)
c. Developer Contact Information (Your email)

5. For the app domain fields set it up as below.



App domain

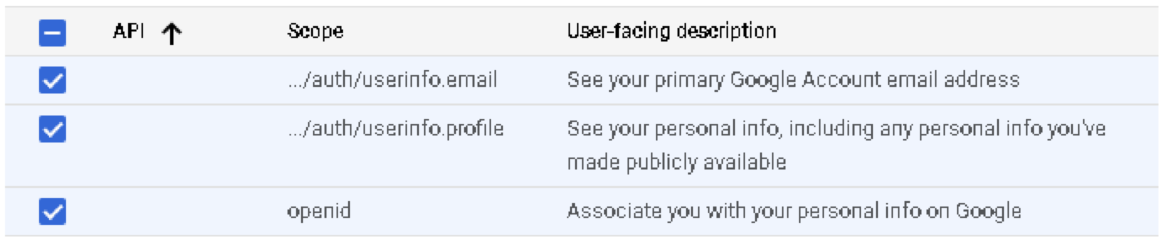
To protect you and your users, Google only allows apps using OAuth to use Authorized Domains. The following information will be shown to your users on the consent screen.

Application home page: http://localhost:3000
Provide users a link to your home page

Application privacy policy link: http://localhost:3000
Provide users a link to your public privacy policy

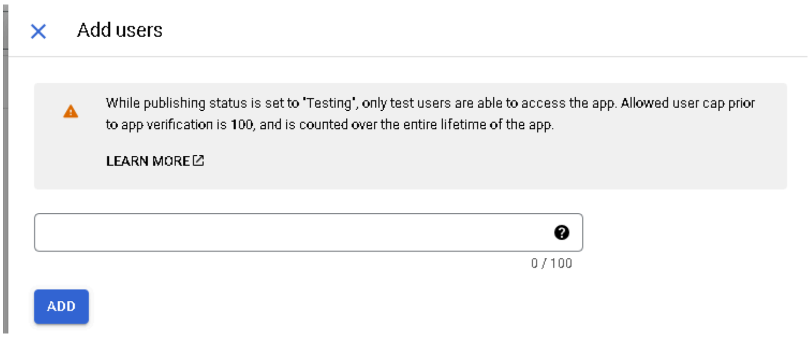
Application terms of service link: http://localhost:3000
Provide users a link to your public terms of service

6. Under Scopes, click **Add or Remove scopes**, select the three following scopes and then hit **update:**



☐	API ↑	Scope	User-facing description
☑		.../auth/userinfo.email	See your primary Google Account email address
☑		.../auth/userinfo.profile	See your personal info, including any personal info you've made publicly available
☑		openid	Associate you with your personal info on Google

7. Save and continue
8. On the test users page click + Add Users and add your email as a test user. Ensure you **click out of the text box** before pressing **Add.**



While publishing status is set to 'Testing', only test users are able to access the app. Allowed user cap prior to app verification is 100, and is counted over the entire lifetime of the app.

LEARN MORE

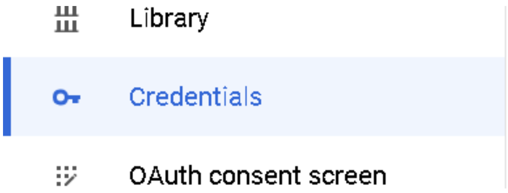
0 / 100

NOTE Ensure that the email you wish to sign in with is added to the above.

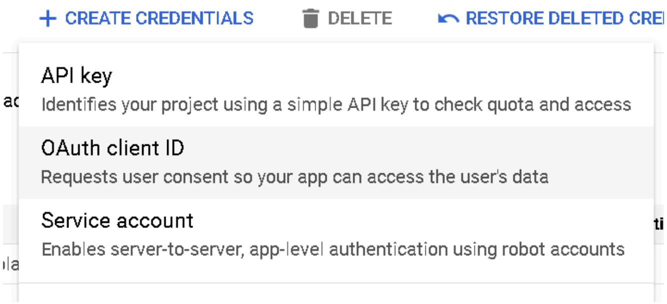
9. Save and continue

Step 3.2 Creating OAuth Credentials:

1. Head over to the Credentials tab on the left hand side.



2. Click on Create Credentials, create and OAuth Client ID.



API key
Identifies your project using a simple API key to check quota and access

OAuth client ID
Requests user consent so your app can access the user's data

Service account
Enables server-to-server, app-level authentication using robot accounts

3. Set the Application type to a Web Application.
4. Set the Authorised Javascript Origins to:

   http://localhost:3000

5. Set the Authorised Redirect URIs to:

http://localhost:3000/api/auth/callback/google

6. **Save your Oauth Client ID and Client secret as you will need this for the next step.**

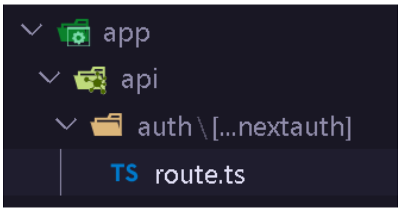
Step 3.3 Installing & Configuring NextAuth:

1. You will need to install NextAuth.js. Run the following command:

```
npm install next-auth
```

2. Create a file within the App Directory.

```
src/app/api/auth/[...nextauth]/route.ts
```

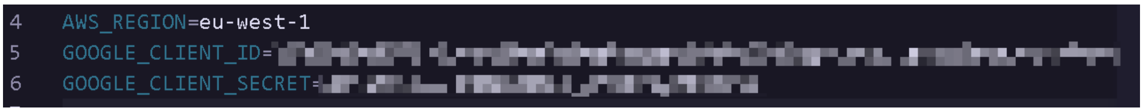


3. Within this file we will set up NextAuth.js with the Google provider. The code for this is below:

```
import NextAuth from "next-auth/next";
import GoogleProvider from "next-auth/providers/google";

const handler = NextAuth({
  providers: [
    GoogleProvider({
      clientId: process.env.GOOGLE_CLIENT_ID ?? "",
      clientSecret: process.env.GOOGLE_CLIENT_SECRET ?? "",
    }),
  ],
});
```

```
export { handler as GET, handler as POST };
```

4. Navigate to the .env file you created in your root directory and add your google client ID and secret:



```
AWS_REGION=eu-west-1
GOOGLE_CLIENT_ID=[redacted]
GOOGLE_CLIENT_SECRET=[redacted]
```

Step 3.4 Adding Components for Authentication:

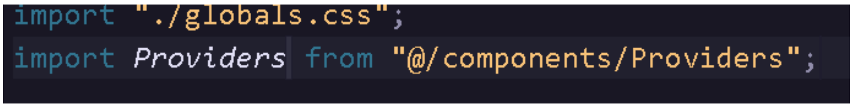
1. We will create a **Sign In button Component**
   a. Create a file called **SigninButton.tsx** in the **src/app/components/** directory:
   b. The code within this is on the next page:

```
"use client";
import React from "react";
import { signIn, signOut, useSession } from "next-auth/react";
const SigninButton = () => {
  const { data: session } = useSession();

  if (session && session.user) {
    return (
      <div className="flex gap-4 ml-auto">
        <p className="text-sky-600">{session.user.name}</p>
        <button onClick={() => signOut()}
className="text-red-600">
          Sign Out
        </button>
      </div>
    );
  }
  return (
    <button onClick={() => signIn()} className="text-green-600
ml-auto">
      Sign In
    </button>
  );
};

export default SigninButton;
```

2. We will now create a Providers component.
    a. Create a file called Providers.tsx in the **src/app/components/** directory.
    b. Insert the code:

```
"use client";
import { SessionProvider } from "next-auth/react";
import React, { ReactNode } from "react";

interface Props {
  children: ReactNode;
}

const Providers = (props: Props) => {
  return <SessionProvider>{props.children}</SessionProvider>;
};

export default Providers;
```

3. Now that we have this, we can head over to the **layout.tsx** file and import the providers file.
    a. First, Import the providers file into the file.



```
import "./globals.css";
import Providers from "@/components/Providers";
```

    b. Then scrolling down to the body of this file, we will wrap the providers around the app components and add a <Appbar />.

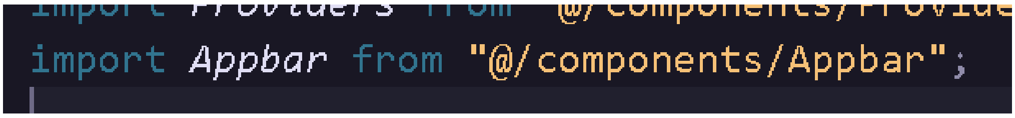
    NOTE You may get an error. Don't worry, we will make the Appbar component next!

```
return (
    <html lang="en">
    <body
    className={`${geistSans.variable} ${geistMono.variable}
antialiased`}
    > <Providers>
      <Appbar />
      {children}
    </Providers>
    </body>
    </html>
  );
```

4. We will now create the Appbar component. Create a new tsx file under components
   src\app\components\Appbar.tsx
   a. Within this file we will insert the code:

```
import React from "react";
import SigninButton from "./SigninButton";

const Appbar = () => {
  return (
    <header className="flex gap-4 p-4 bg-gradient-to-b
from-white to-gray-200 shadow">
      <SigninButton />
    </header>
  );
};

export default Appbar;
```

You will need to head back to layout.tsx and import the AppBar.



```
import Providers from "@/components/Provide
import Appbar from "@/components/Appbar";
```

Step 3.5 Protecting our Task Manager:

1. We will modify the page.tsx file to hide the Task Manager component when a user is not signed in:
    a. **Copy the following code into your page.tsx file.**

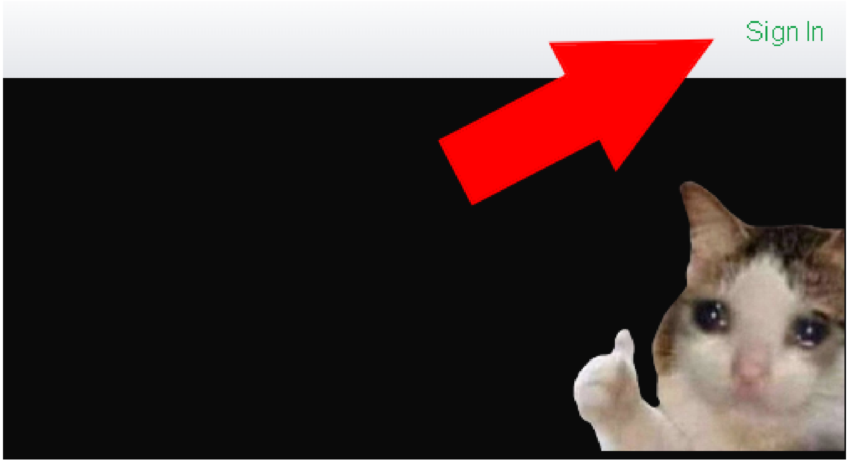
```
"use client";
import TaskManager from '../components/taskmanager';
import { signIn, signOut, useSession } from "next-auth/react";


export default function Home() {

  const { data: session } = useSession();

  if (session && session.user) {
    return (
      <TaskManager />
    );
  }
}
```

2. Run your application and head to localhost:3000

```
npm run dev
```

3. Test that Auth works properly, you should see a sign in button at the top right.

Congrats, your frontend is secure!



Step 4: Creating a Synchronous Data Export API (10 mins)

Step 4.1 Create the SendMessageSync Endpoint

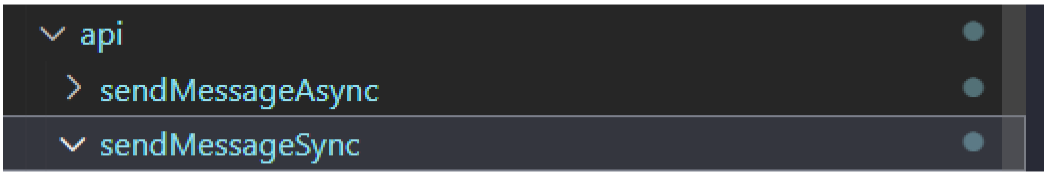
To begin,we are going to create a new folder in our API folder, where we previously added the **sendMessageAsync** endpoint.

NOTE: If you keep your local server running we don’t need to re-run the

```
Npm run dev
```

command each time we make a change to our code , as next.js will re-compile and make the changes for you!

Call this folder **sendMessageSync**



```
api
  sendMessageAsync
  sendMessageSync
```

In this folder **create a route.ts** file and paste in the following code.
This file describes the POST request that will be called when we want to **export our data synchronously.**

```
import { NextRequest, NextResponse } from "next/server";

async function sleep(ms: number): Promise<void> {
 return new Promise((resolve) => setTimeout(resolve, ms));
}

export async function POST(req: NextRequest) {
 try {
   console.log("simulating a long process");
   console.log("exporting data...");

   const start = new Date().getTime();
   await sleep(10000);
   let elapsed = new Date().getTime() - start;
   const elapsedInt = Math.floor(elapsed / 1000);

   console.log("data exported successfully!");
   console.log("time taken to process: " + elapsedInt + " seconds");

   return NextResponse.json(
     {
       message: "data exported successfully in " + elapsedInt + " seconds",
     },
     { status: 200 }
   );
 } catch (error) {
   console.error("error exporting data: ", error);
   return NextResponse.json(
     { error: "failed to export data" },
     { status: 500 }
   );
 }
}
```

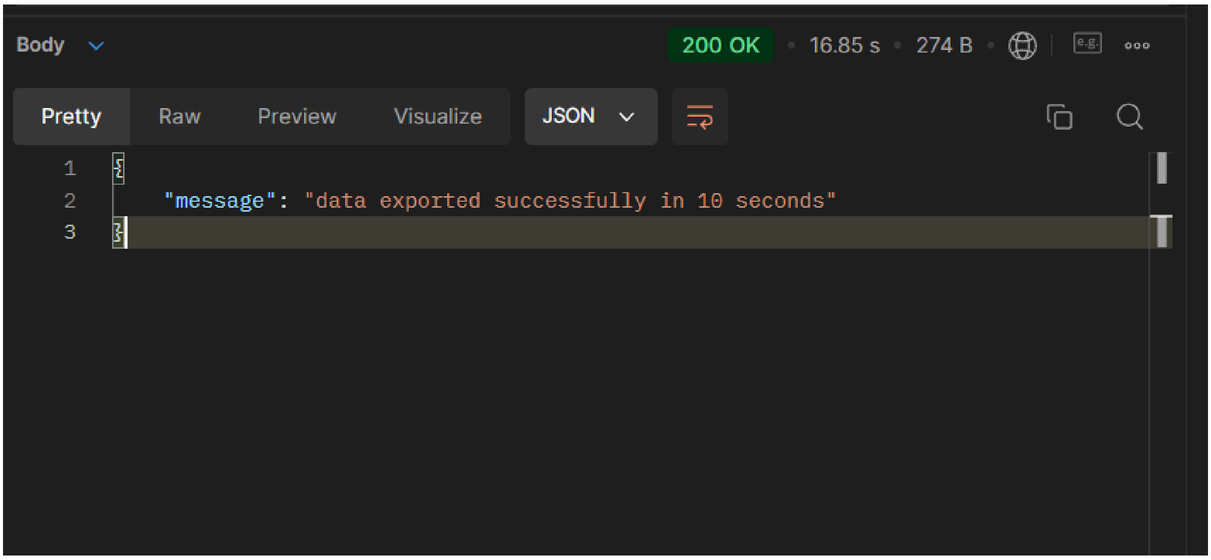
What does this code do?

- It creates an endpoint, that when called with some data, will simulate a long running process using *Await sleep(10000);*
- After "processing" for 10 seconds, this endpoint will then return successfully.

Step 4.2 Testing the sendMessageSync endpoint we just made In Postman

Go back to Postman and create a new **POST** request and add the URL `http://localhost:3000/api/sendMessageSync` and now send - **no body required.**

After 10 seconds you should get a response, indicating that the data was exported successfully.



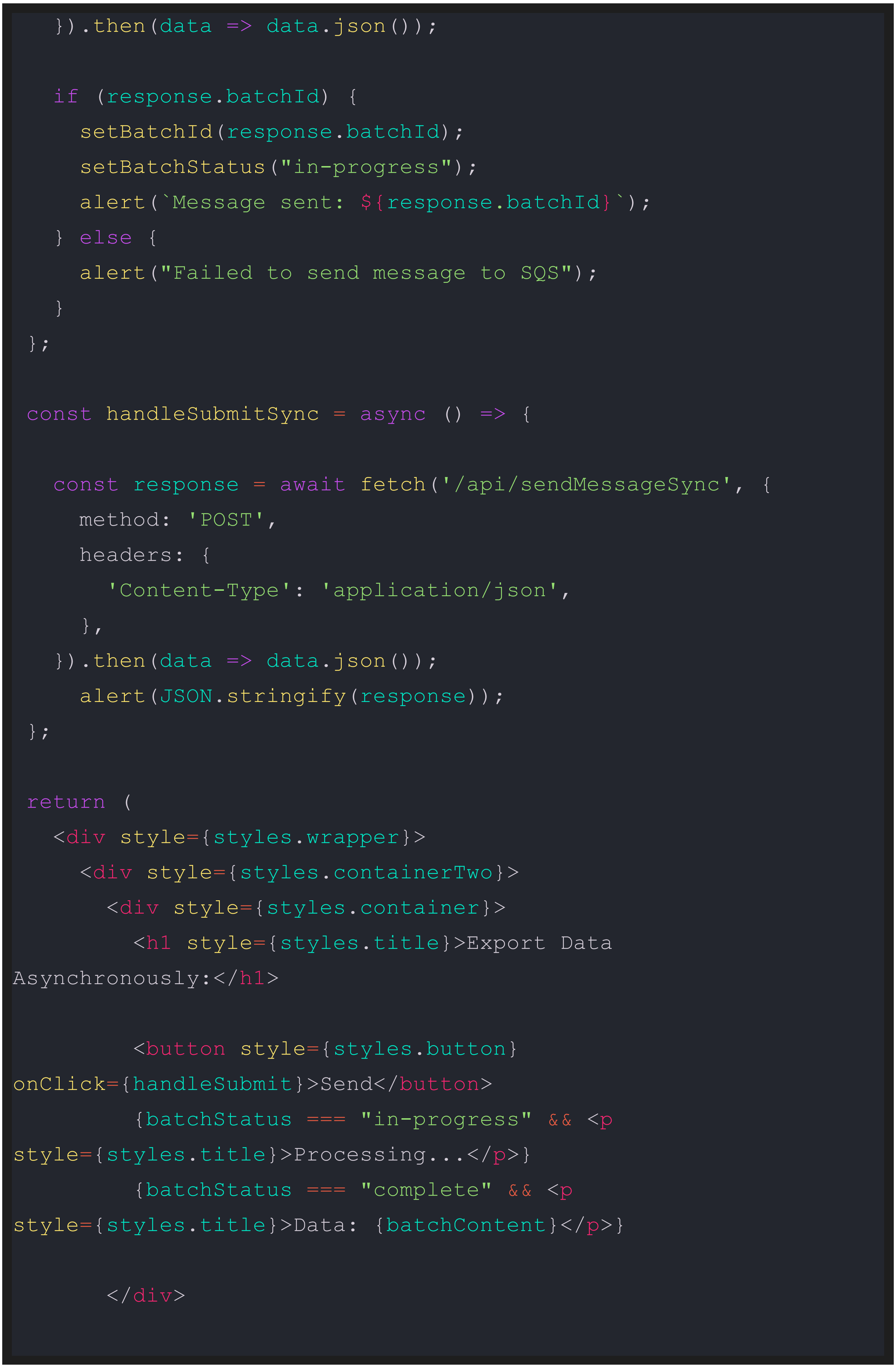
```
{
    "message": "data exported successfully in 10 seconds"
}
```

Step 4.3 Hooking this endpoint up to our front-end

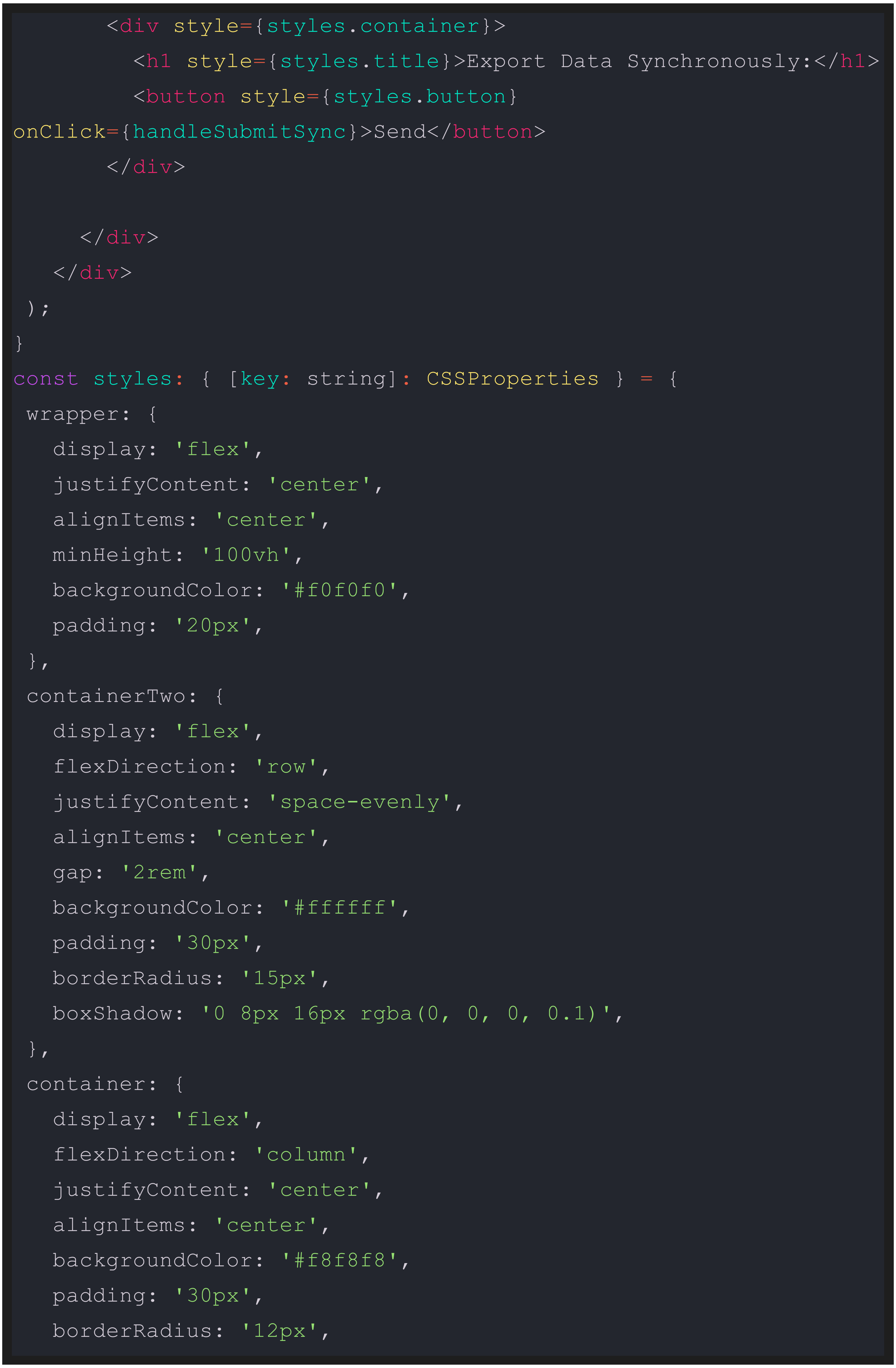
Now we want to hook this endpoint up to our front-end, so we are going to make some changes to the **taskmanager.tsx** file.

First, **remove everything** you had in that file previously and paste in the following code.

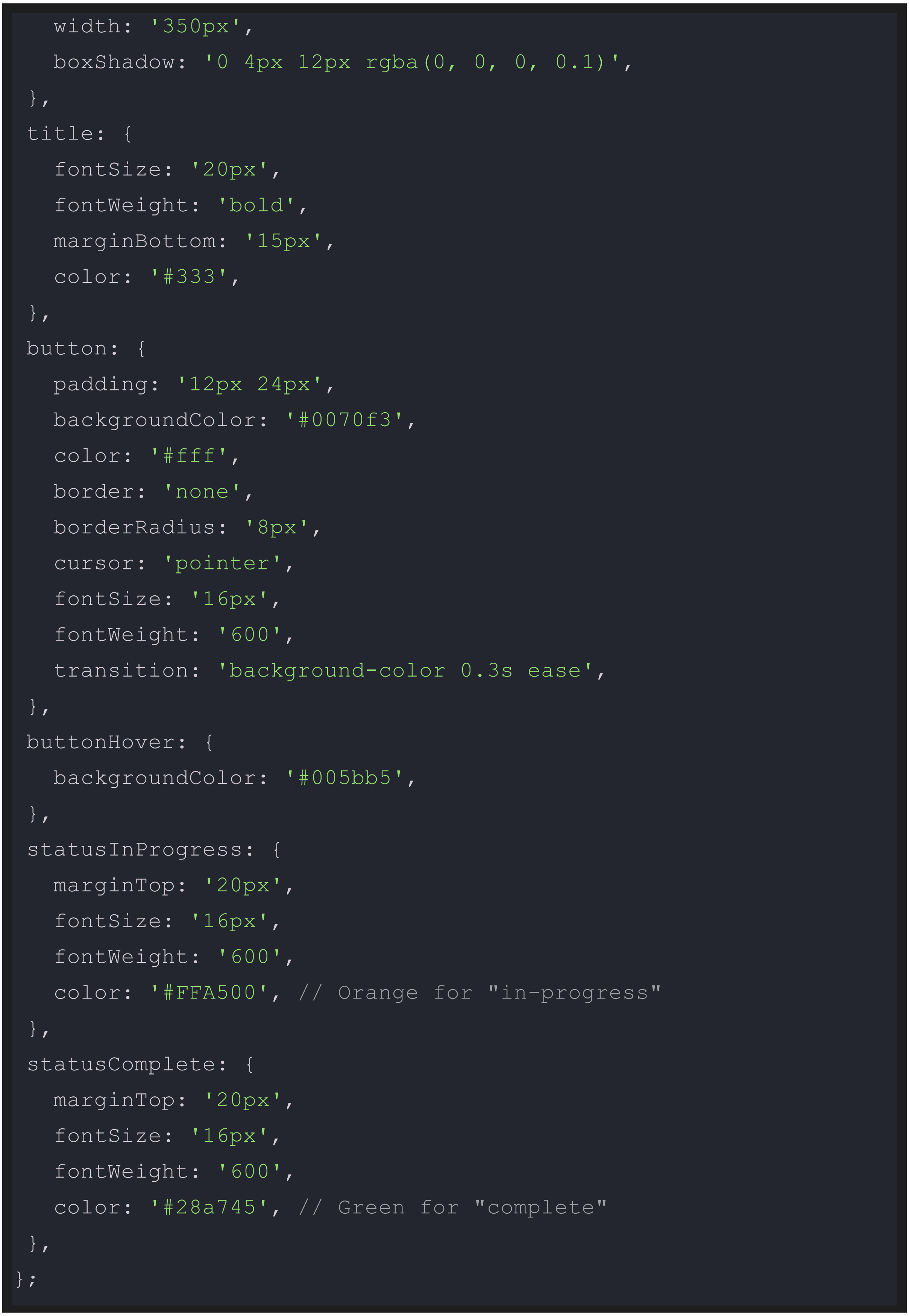
```
"use client";
import { useState, useEffect } from 'react';
import { CSSProperties } from 'react';

export default function TaskManager() {
 const [batchId, setBatchId] = useState<string | null>(null);
 const [batchStatus, setBatchStatus] = useState<string>("idle");
 const [batchContent, setBatchContent] = useState<string>("");

 const handleSubmit = async () => {
   const response = await fetch('/api/sendMessageAsync', {
     method: 'POST',
     headers: {
       'Content-Type': 'application/json',
     },
```



```
    }).then(data => data.json());

    if (response.batchId) {
      setBatchId(response.batchId);
      setBatchStatus("in-progress");
      alert(`Message sent: ${response.batchId}`);
    } else {
      alert("Failed to send message to SQS");
    }
  };

  const handleSubmitSync = async () => {

    const response = await fetch('/api/sendMessageSync', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
    }).then(data => data.json());
      alert(JSON.stringify(response));
  };

  return (
    <div style={styles.wrapper}>
      <div style={styles.containerTwo}>
        <div style={styles.container}>
          <h1 style={styles.title}>Export Data
Asynchronously:</h1>

          <button style={styles.button}
onClick={handleSubmit}>Send</button>
          {batchStatus === "in-progress" && <p
style={styles.title}>Processing...</p>}
          {batchStatus === "complete" && <p
style={styles.title}>Data: {batchContent}</p>}

        </div>
```



```
       <div style={styles.container}>
         <h1 style={styles.title}>Export Data Synchronously:</h1>
         <button style={styles.button}
onClick={handleSubmitSync}>Send</button>
       </div>

     </div>
   </div>
 );
}
const styles: { [key: string]: CSSProperties } = {
 wrapper: {
   display: 'flex',
   justifyContent: 'center',
   alignItems: 'center',
   minHeight: '100vh',
   backgroundColor: '#f0f0f0',
   padding: '20px',
 },
 containerTwo: {
   display: 'flex',
   flexDirection: 'row',
   justifyContent: 'space-evenly',
   alignItems: 'center',
   gap: '2rem',
   backgroundColor: '#ffffff',
   padding: '30px',
   borderRadius: '15px',
   boxShadow: '0 8px 16px rgba(0, 0, 0, 0.1)',
 },
 container: {
   display: 'flex',
   flexDirection: 'column',
   justifyContent: 'center',
   alignItems: 'center',
   backgroundColor: '#f8f8f8',
   padding: '30px',
   borderRadius: '12px',
```

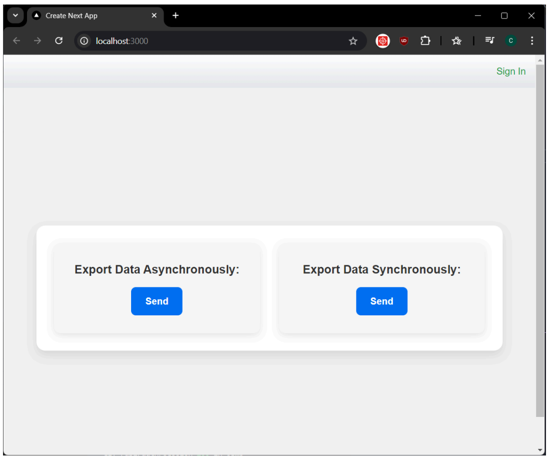


```
    width: '350px',
    boxShadow: '0 4px 12px rgba(0, 0, 0, 0.1)',
  },
  title: {
    fontSize: '20px',
    fontWeight: 'bold',
    marginBottom: '15px',
    color: '#333',
  },
  button: {
    padding: '12px 24px',
    backgroundColor: '#0070f3',
    color: '#fff',
    border: 'none',
    borderRadius: '8px',
    cursor: 'pointer',
    fontSize: '16px',
    fontWeight: '600',
    transition: 'background-color 0.3s ease',
  },
  buttonHover: {
    backgroundColor: '#005bb5',
  },
  statusInProgress: {
    marginTop: '20px',
    fontSize: '16px',
    fontWeight: '600',
    color: '#FFA500', // Orange for "in-progress"
  },
  statusComplete: {
    marginTop: '20px',
    fontSize: '16px',
    fontWeight: '600',
    color: '#28a745', // Green for "complete"
  },
};
```

What does this code do?

- Adds a new button to our front-end, so now we have two options for exporting our data
  1. Asynchronously - using the **sendMessageAsync** endpoint we made earlier
  2. Synchronously - using the **sendMessageSync** endpoint we just made!

Now our front-end should look like this:



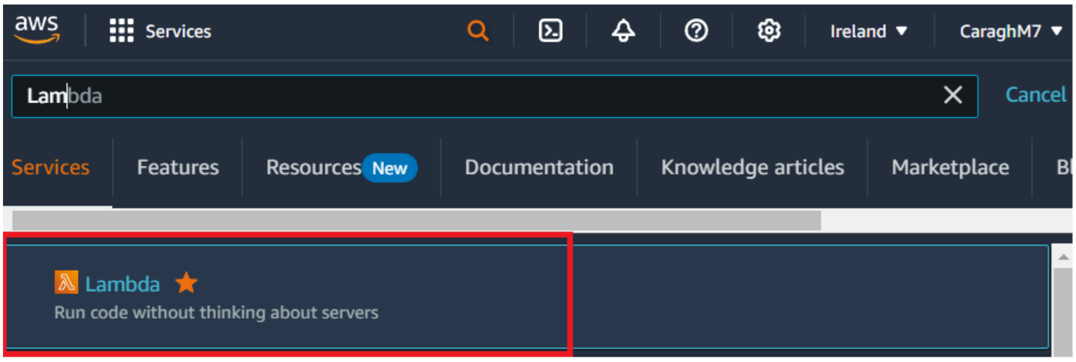
We are now finished with **Step 4.** 🫡

Step 5: Setting up a Lambda to process Messages in our SQS Queue (25 mins)

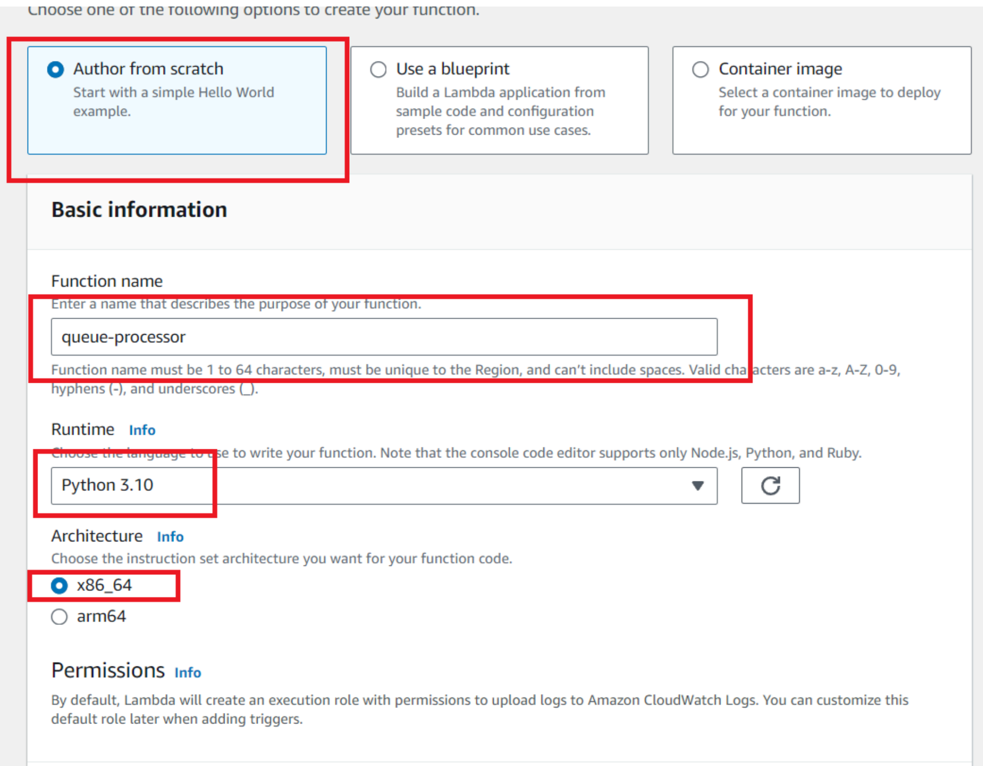
Now that we have an endpoint that we can call to process our data synchronously, we can see that waiting 10 seconds for a response from the API is a **really long wait!** So we should use the Asynchronous endpoint so we get a response straight away, but at the minute all that endpoint is doing is **sending a message to the queue, not processing it!** So this is why we add a Lambda.

Step 5.1 Creating a Lambda in the AWS Console

Go back to your AWS console and look up Lambda in the search bar.



Once on the lambda page, click the **Create function** button. When creating the lambda, select the following options for config



Choose one of the following options to create your function.

● Author from scratch
Start with a simple Hello World example.

○ Use a blueprint
Build a Lambda application from sample code and configuration presets for common use cases.

○ Container image
Select a container image to deploy for your function.

Basic information

Function name
Enter a name that describes the purpose of your function.
queue-processor
Function name must be 1 to 64 characters, must be unique to the Region, and can't include spaces. Valid characters are a-z, A-Z, 0-9, hyphens (-), and underscores (_).

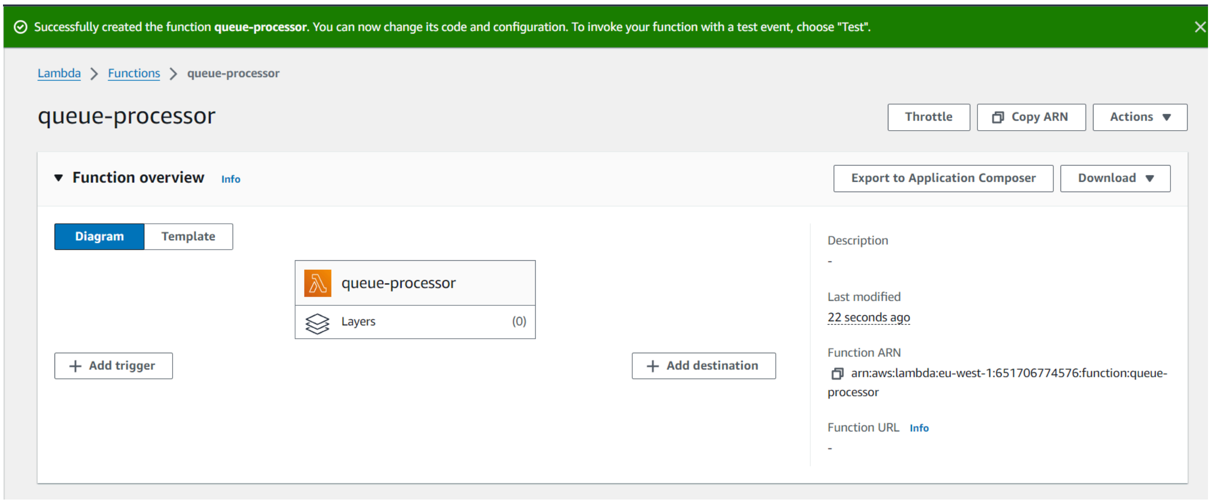
Runtime Info
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.
Python 3.10 ▼

Architecture Info
Choose the instruction set architecture you want for your function code.
● x86_64
○ arm64

Permissions Info
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

Once you have these selected, hit **create function** at the bottom of the page.

Now you should be presented with this page to let you know it has been created successfully



Successfully created the function **queue-processor**. You can now change its code and configuration. To invoke your function with a test event, choose "Test".

Lambda > Functions > queue-processor

queue-processor

Description
-

Last modified
22 seconds ago

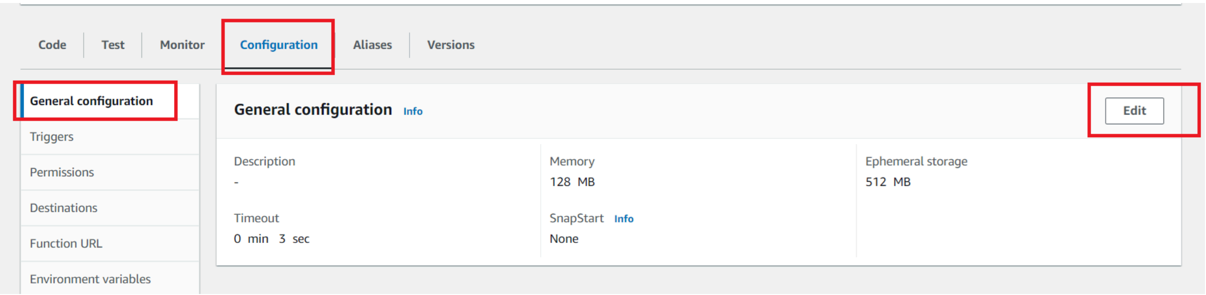
Function ARN
arn:aws:lambda:eu-west-1:651706774576:function:queue-processor

Function URL Info
-

Step 5.2 Changing default timeout of the lambda

We need to do this to allow the lambda to run for longer as we will be running long processing tasks in the lambda

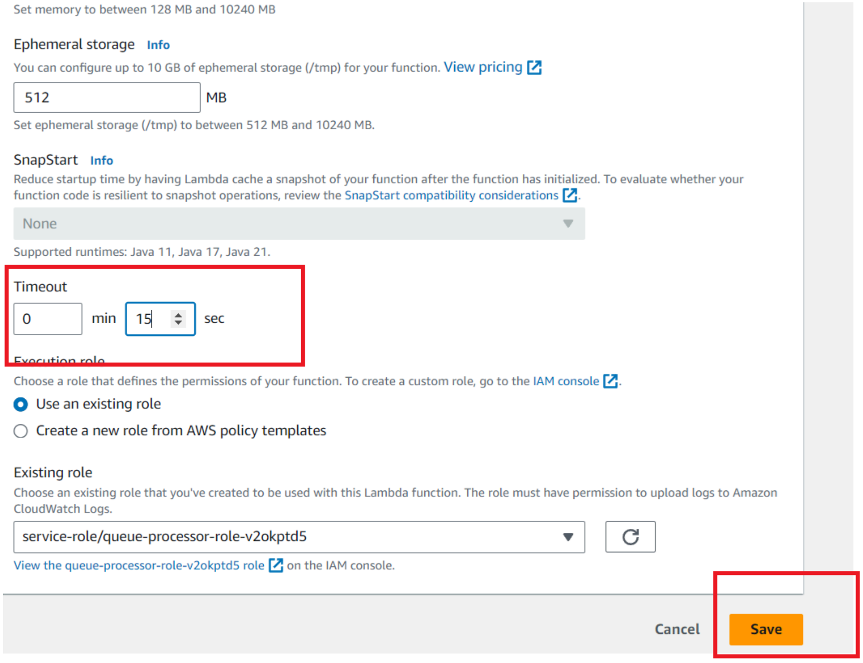
Scroll down to the lambda options bar and click on **Configuration.** Then click on General Configuration and then **edit**



General configuration

Description	Memory	Ephemeral storage
-	128 MB	512 MB
Timeout	SnapStart Info	
0 min 3 sec	None	

Then **only** change the **timeout** attribute to be 15 seconds and click save



Set memory to between 128 MB and 10240 MB

Ephemeral storage Info

You can configure up to 10 GB of ephemeral storage (/tmp) for your function. View pricing

512 MB

Set ephemeral storage (/tmp) to between 512 MB and 10240 MB.

SnapStart Info

Reduce startup time by having Lambda cache a snapshot of your function after the function has initialized. To evaluate whether your function code is resilient to snapshot operations, review the SnapStart compatibility considerations.

None

Supported runtimes: Java 11, Java 17, Java 21.

Timeout

0 min 15 sec

Execution role

Choose a role that defines the permissions of your function. To create a custom role, go to the IAM console.

● Use an existing role

○ Create a new role from AWS policy templates

Existing role

Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

service-role/queue-processor-role-v2okptd5

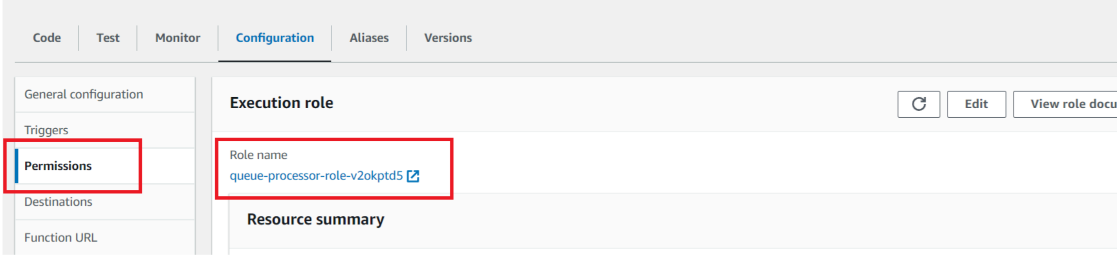
View the queue-processor-role-v2okptd5 role on the IAM console.

Done!

Step 5.2 Adding SQS permissions to the lambda

Now we have to add some permissions to the lambda to read the messages from our queue

Click on **Configuration** again, then we are going to click on **Permissions** on the left, and then click on **role name.**



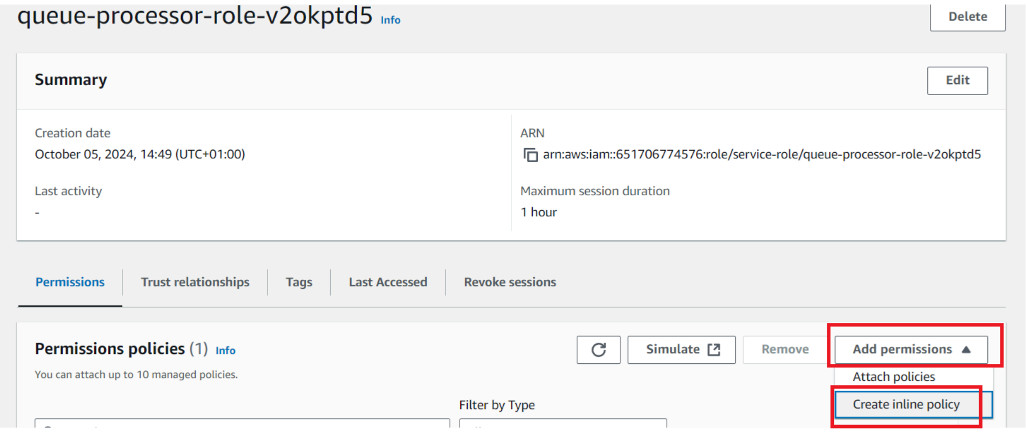
Execution role

Role name

queue-processor-role-v2okptd5

Resource summary

This will bring you to the Access management page, where you have to scroll down and click **add permissions,** and then **create inline policy**



queue-processor-role-v2okptd5 Info

Summary

Creation date
October 05, 2024, 14:49 (UTC+01:00)

ARN
arn:aws:iam::651706774576:role/service-role/queue-processor-role-v2okptd5

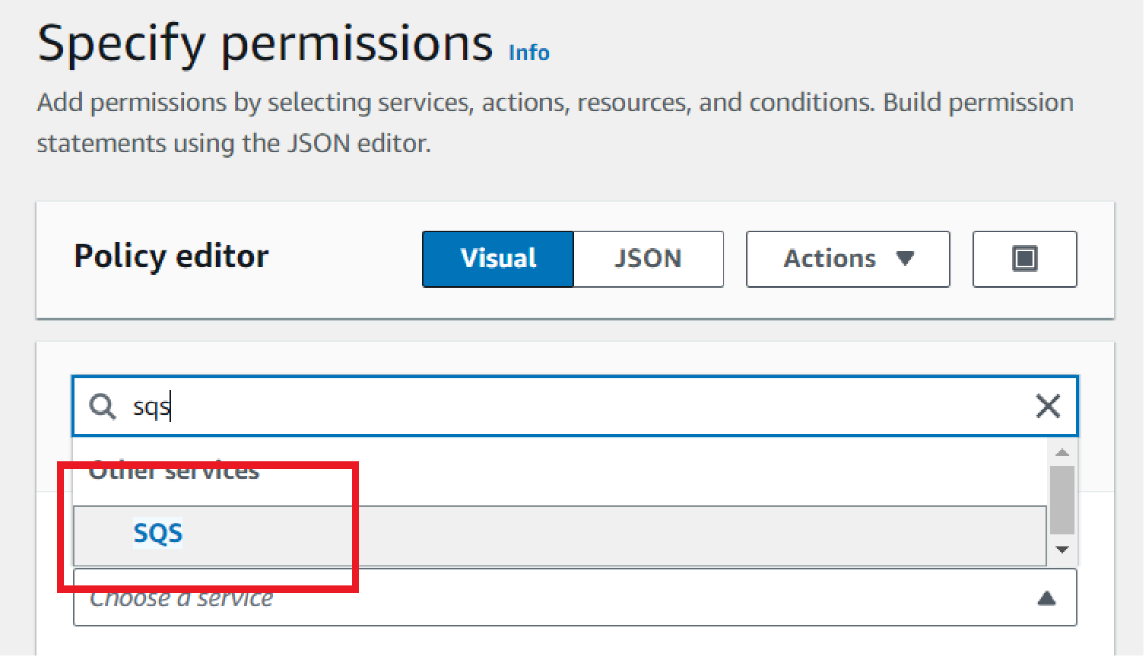
Last activity
-

Maximum session duration
1 hour

Permissions policies (1) Info

You can attach up to 10 managed policies.

You will be brought to the **create policy** page, where you choose **SQS** as the service in the **Select a service** dropdown.



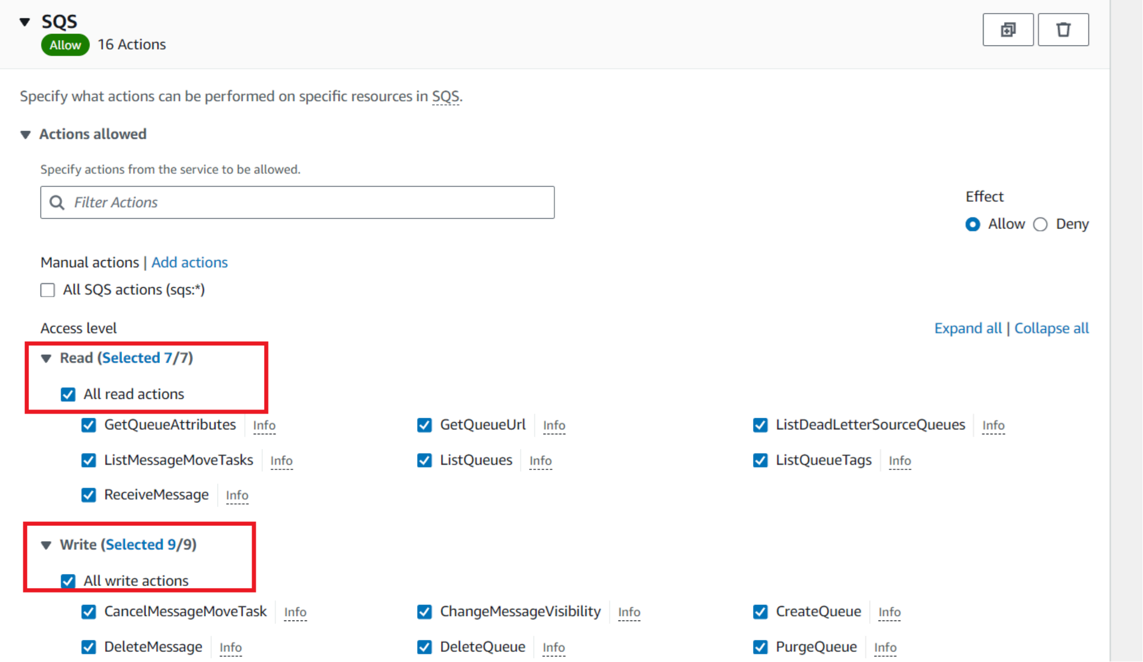
Specify permissions Info

Add permissions by selecting services, actions, resources, and conditions. Build permission statements using the JSON editor.

Policy editor

sqs

Once SQS is selected, scroll down and expand the **read** section and select **all read actions**, and do the same for the **write actions**



▼ SQS

Allow 16 Actions

Specify what actions can be performed on specific resources in SQS.

▼ Actions allowed

Specify actions from the service to be allowed.

Filter Actions

Effect

● Allow ○ Deny

Manual actions | Add actions

□ All SQS actions (sqs:*)

Access level

Expand all | Collapse all

▼ Read (Selected 7/7)

☑ All read actions

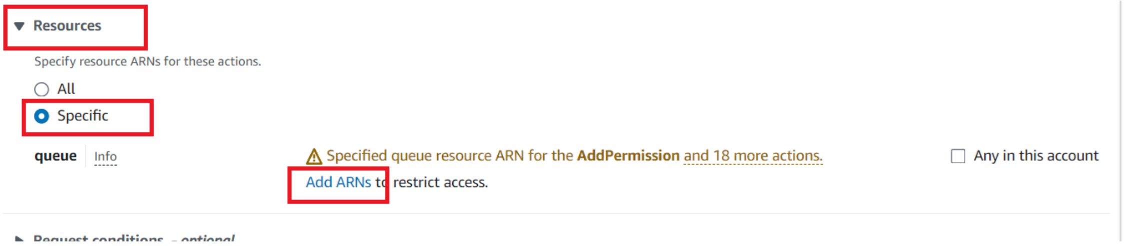
☑ GetQueueAttributes Info	☑ GetQueueUrl Info	☑ ListDeadLetterSourceQueues Info
☑ ListMessageMoveTasks Info	☑ ListQueues Info	☑ ListQueueTags Info
☑ ReceiveMessage Info		

▼ Write (Selected 9/9)

☑ All write actions

☑ CancelMessageMoveTask Info	☑ ChangeMessageVisibility Info	☑ CreateQueue Info
☑ DeleteMessage Info	☑ DeleteQueue Info	☑ PurgeQueue Info

Then scroll down to where the **resources** section is and and select **specific** if not auto-selected, and click **Add ARNs**



▼ Resources

Specify resource ARNs for these actions.

○ All

● Specific

queue Info

⚠ Specified queue resource ARN for the **AddPermission** and 18 more actions.

Add ARNs to restrict access.

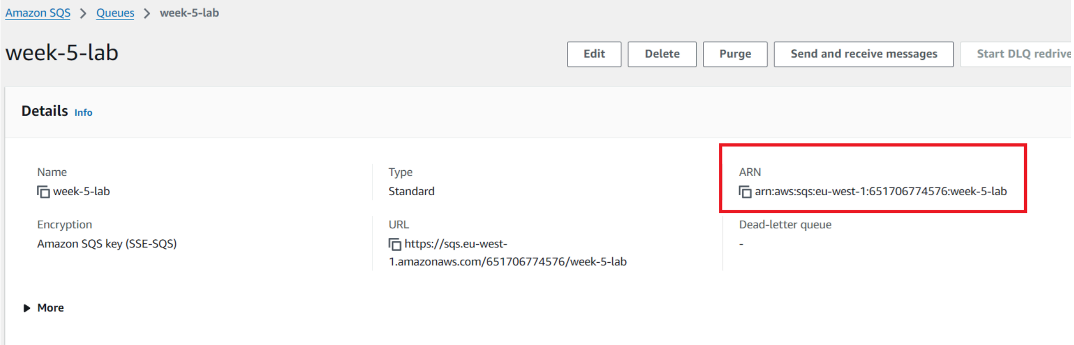
□ Any in this account

▸ Request conditions - optional

Now in here, you are going to paste the ARN of your SQS queue. So..

- Open a new tab in your browser, search AWS management console log in, then once in the console search **SQS** in the services search bar. (If we don't do this in a new tab we will lose our progress!)
- Once in the **SQS page** click on the **queue we made earlier**, and then copy the **ARN** displayed in the

details section.



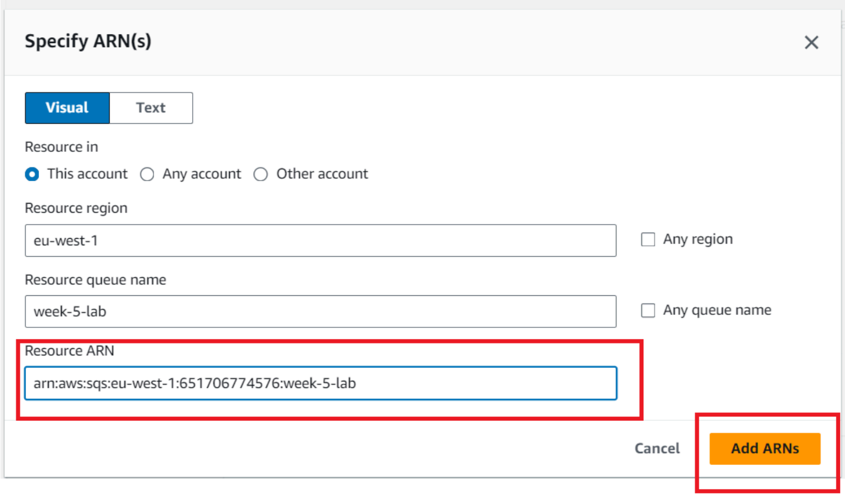
Amazon SQS > Queues > week-5-lab

week-5-lab

Details Info

Name: week-5-lab
Type: Standard
ARN: arn:aws:sqs:eu-west-1:651706774576:week-5-lab
Encryption: Amazon SQS key (SSE-SQS)
URL: https://sqs.eu-west-1.amazonaws.com/651706774576/week-5-lab
Dead-letter queue: -

Once copied, **go back to the previous tab** where we were editing the policy, and paste into the **bottom input section** and click **Add ARN**



Specify ARN(s)

Resource in: ● This account ○ Any account ○ Other account

Resource region: eu-west-1 □ Any region

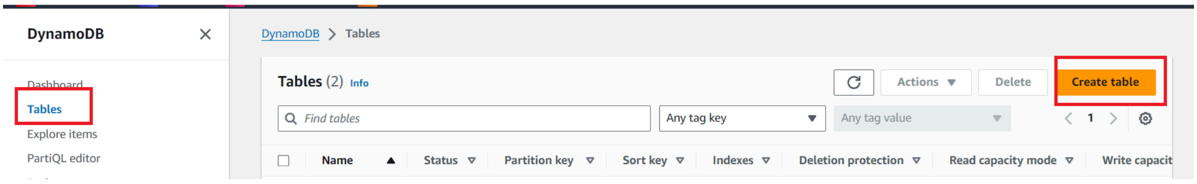
Resource queue name: week-5-lab □ Any queue name

Resource ARN: arn:aws:sqs:eu-west-1:651706774576:week-5-lab

That's it for adding permissions for our queue, but don't hit next at the bottom of the page, as we now need to add permissions for the dynamo table!

Step 5.3 Adding DynamoDB Permissions to our Lambda

In the tab where we got the SQS ARN, now look up **DynamoDB** in the services search bar, and go into the **Tables** section, and click **create table**.



Name	Status	Partition key	Sort key	Indexes	Deletion protection	Read capacity mode	Write capacit

Then configure the table as shown below and hit **create table** at the bottom of the page.

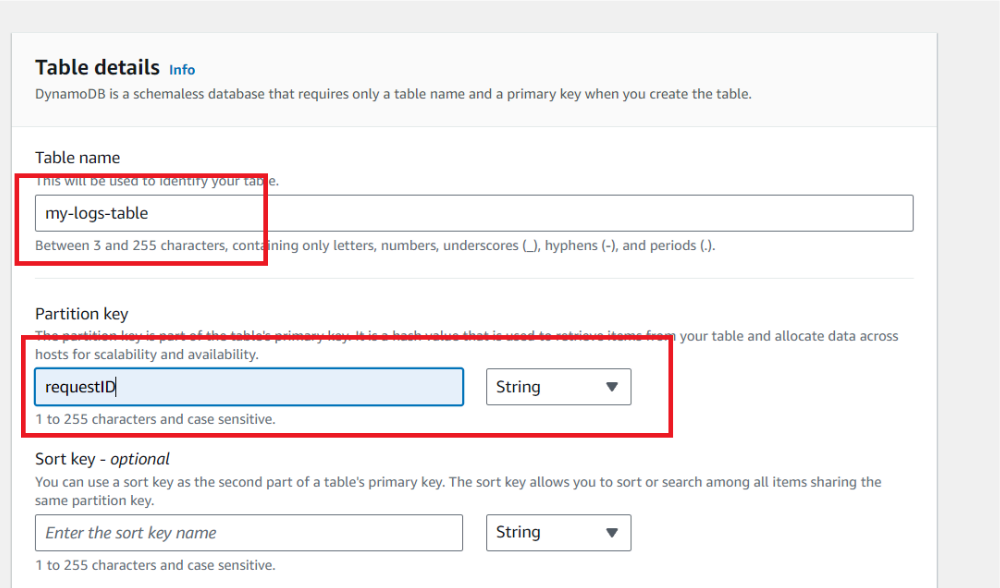


Table details Info

DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name

This will be used to identify your table.

my-logs-table

Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key

The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

requestID | String ▼

1 to 255 characters and case sensitive.

Sort key - *optional*

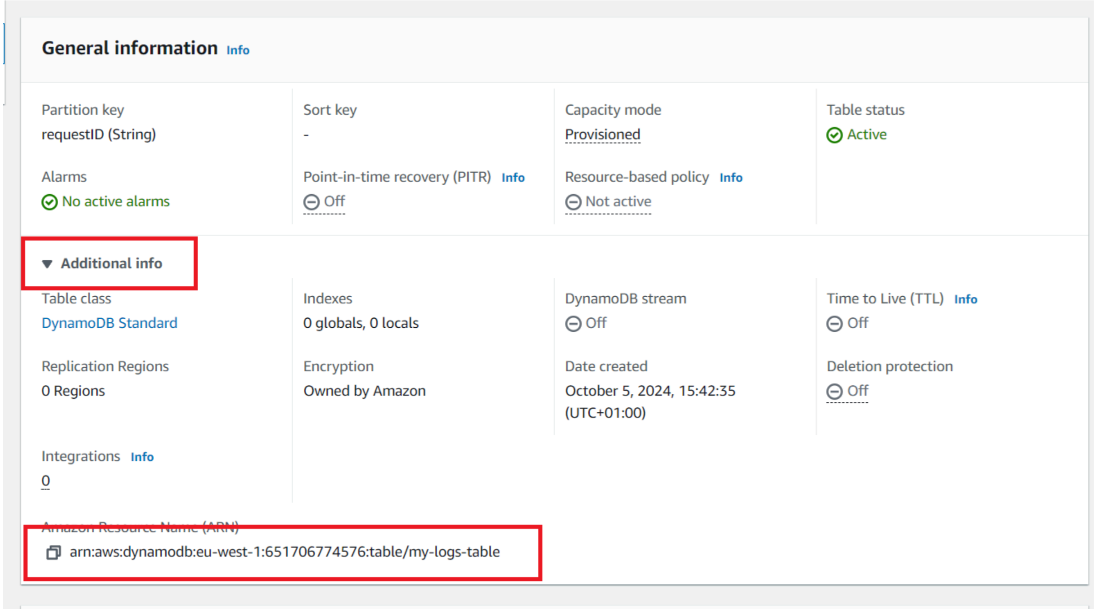
You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

Enter the sort key name | String ▼

1 to 255 characters and case sensitive.

We will be brought back to the tables page and now click on the **table you just created.**

In your table page scroll down to **General Information,** click **additional information** and **copy that ARN**!



General information Info

Partition key
requestID (String)

Sort key
-

Capacity mode
Provisioned

Table status
✓ Active

Alarms
✓ No active alarms

Point-in-time recovery (PITR) Info
⊖ Off

Resource-based policy Info
⊖ Not active

▼ Additional info

Table class
DynamoDB Standard

Indexes
0 globals, 0 locals

DynamoDB stream
⊖ Off

Time to Live (TTL) Info
⊖ Off

Replication Regions
0 Regions

Encryption
Owned by Amazon

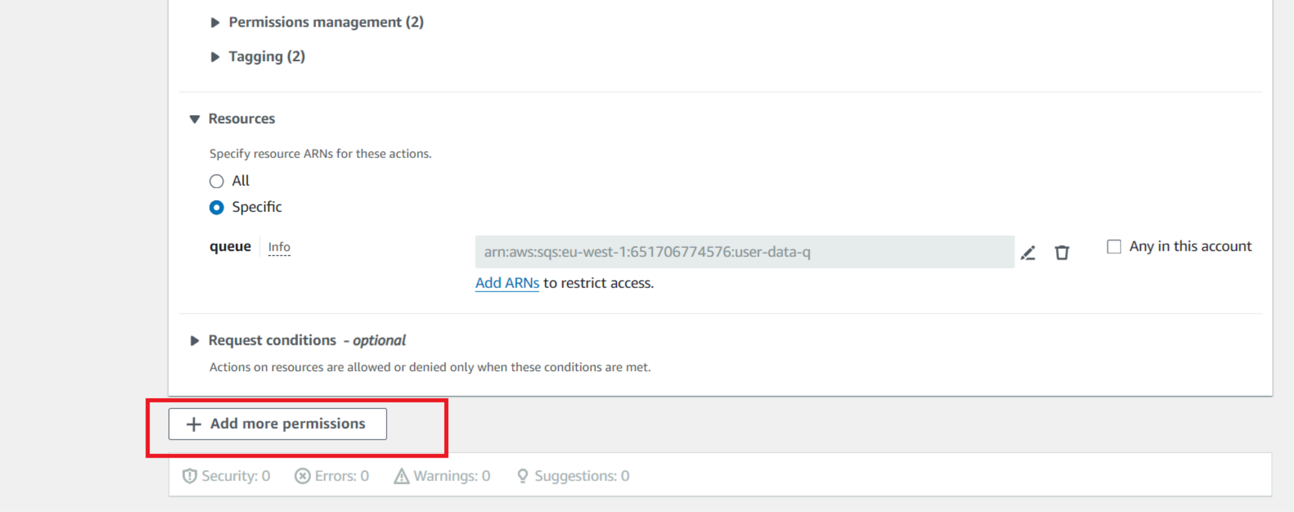
Date created
October 5, 2024, 15:42:35 (UTC+01:00)

Deletion protection
⊖ Off

Integrations Info
0

Amazon Resource Name (ARN)
arn:aws:dynamodb:eu-west-1:651706774576:table/my-logs-table

Now back again to the tab where we were editing the permissions (**last time I promise!!**) Under where you filled out all the options for the SQS permissions, click **add more permissions.**



▶ Permissions management (2)

▶ Tagging (2)

▼ Resources

Specify resource ARNs for these actions.

○ All

● Specific

queue Info — arn:aws:sqs:eu-west-1:651706774576:user-data-q — □ Any in this account

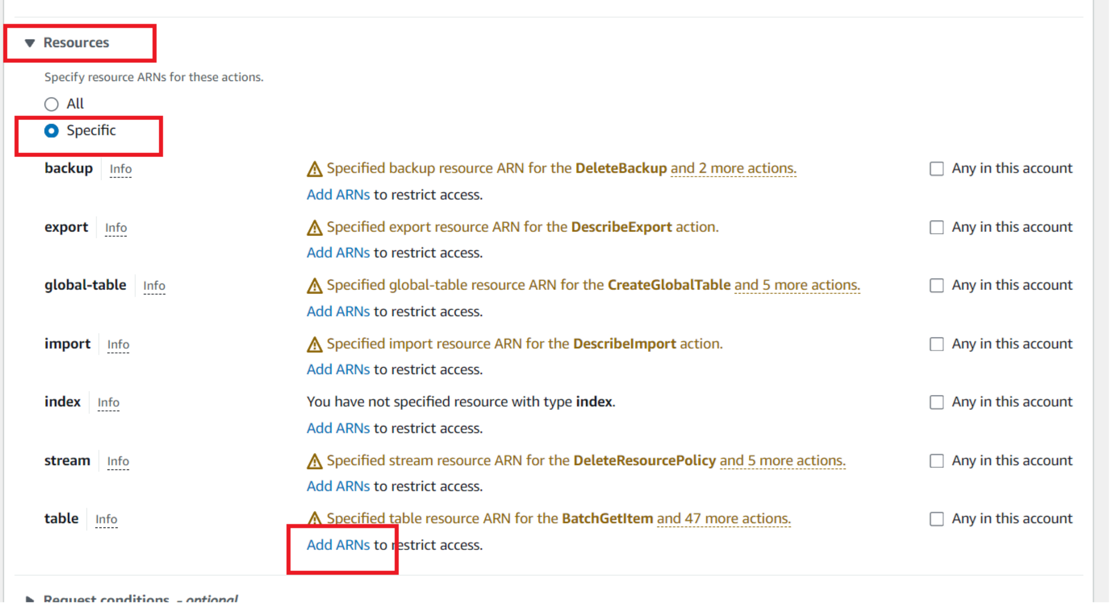
Add ARNs to restrict access.

▶ Request conditions - *optional*

Actions on resources are allowed or denied only when these conditions are met.

Now we are going to **recreate** the steps we took to add permissions for SQS, so in short

- Select **DynamoDB** as the service
- Expand the **read section** and select **All read actions**
- Expand the **write section** and select **All read actions**
- Scroll down to the **Resources** section, click **specific** if not auto-selected, and then go to the **table section and add the ARN** (this is the one you copied when creating the Dynamo table)



▼ Resources

Specify resource ARNs for these actions.

○ All

● Specific

backup Info — ⚠ Specified backup resource ARN for the **DeleteBackup** and 2 more actions. — □ Any in this account
Add ARNs to restrict access.

export Info — ⚠ Specified export resource ARN for the **DescribeExport** action. — □ Any in this account
Add ARNs to restrict access.

global-table Info — ⚠ Specified global-table resource ARN for the **CreateGlobalTable** and 5 more actions. — □ Any in this account
Add ARNs to restrict access.

import Info — ⚠ Specified import resource ARN for the **DescribeImport** action. — □ Any in this account
Add ARNs to restrict access.

index Info — You have not specified resource with type **index**. — □ Any in this account
Add ARNs to restrict access.

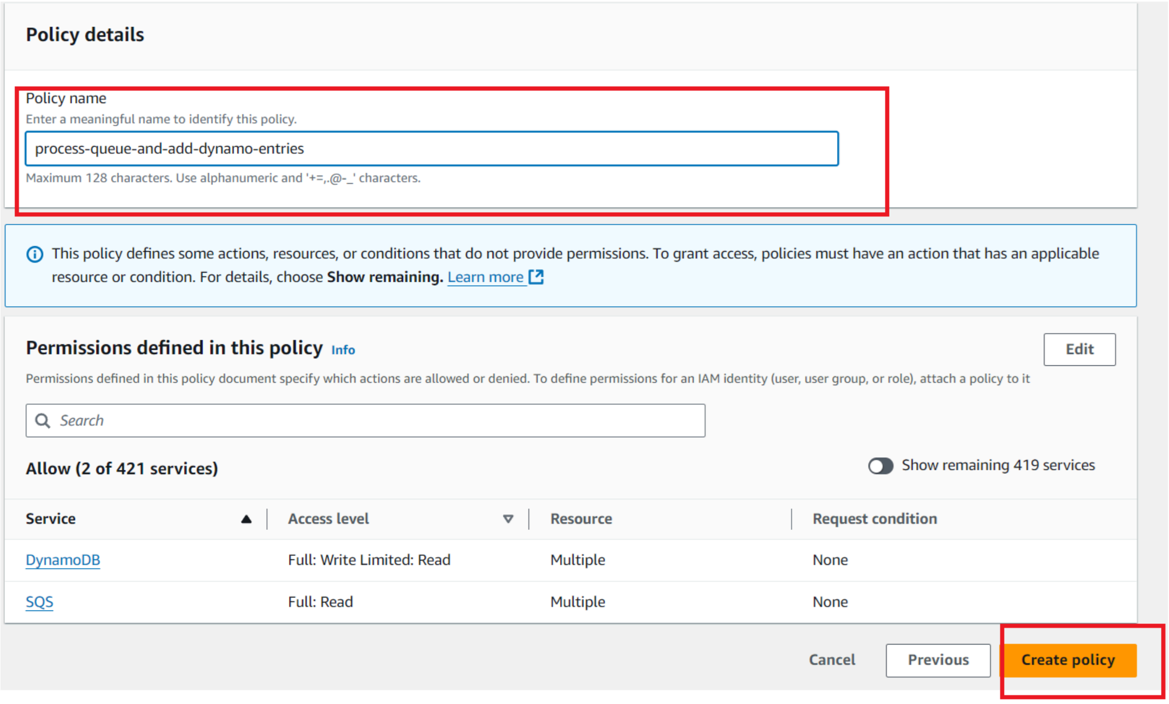
stream Info — ⚠ Specified stream resource ARN for the **DeleteResourcePolicy** and 5 more actions. — □ Any in this account
Add ARNs to restrict access.

table Info — ⚠ Specified table resource ARN for the **BatchGetItem** and 47 more actions. — □ Any in this account
Add ARNs to restrict access.

▶ Request conditions - *optional*

After **pasting the ARN** in the bottom input of the Add ARN window, scroll to the bottom of the permissions page and **click next.**

This will bring us to the **review and create page,** where we will add a name for this policy.



Policy details

Policy name
Enter a meaningful name to identify this policy.
process-queue-and-add-dynamo-entries
Maximum 128 characters. Use alphanumeric and '+=,.@-_' characters.

ⓘ This policy defines some actions, resources, or conditions that do not provide permissions. To grant access, policies must have an action that has an applicable resource or condition. For details, choose **Show remaining.** Learn more

Permissions defined in this policy Info

Permissions defined in this policy document specify which actions are allowed or denied. To define permissions for an IAM identity (user, user group, or role), attach a policy to it

Allow (2 of 421 services)

Show remaining 419 services

Service	Access level	Resource	Request condition
DynamoDB	Full: Write Limited: Read	Multiple	None
SQS	Full: Read	Multiple	None

Now your lambda has permission to do cool stuff!

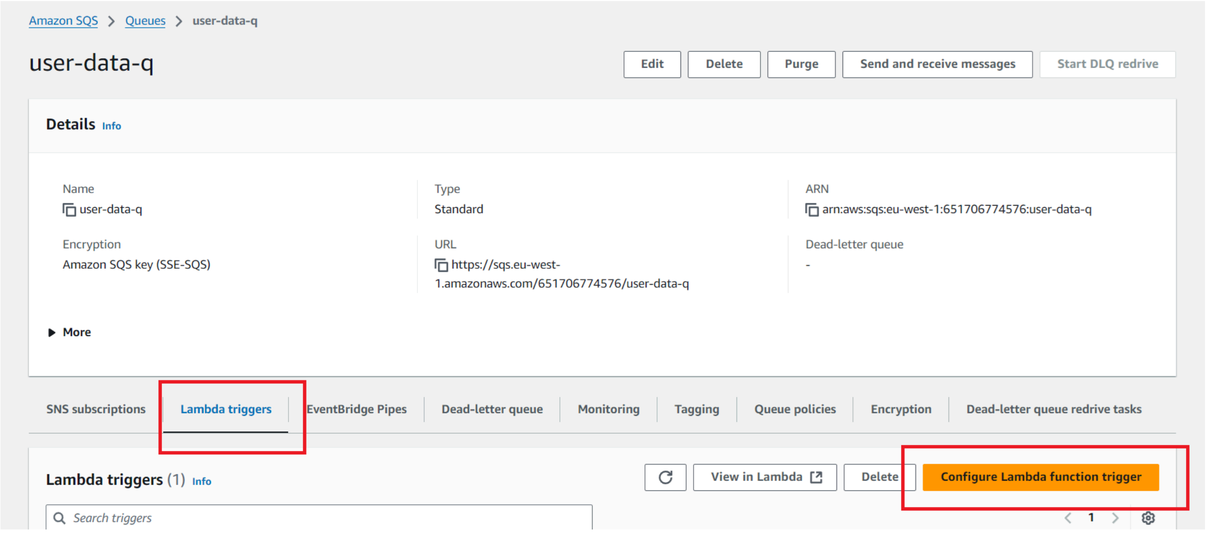
Step 5.4 Adding our Lambda as a trigger to SQS

Now that our lambda has permission to read messages from our queue and also to write entries to our table, **we can add it as a trigger** on our queue.

Why?

- So that when we add a message to our queue, this lambda will trigger and do some cool stuff automatically.

Now search again for SQS in the services search bar and click on your queue.
In here click on **Lambda triggers** and then on **Configure lambda function trigger**



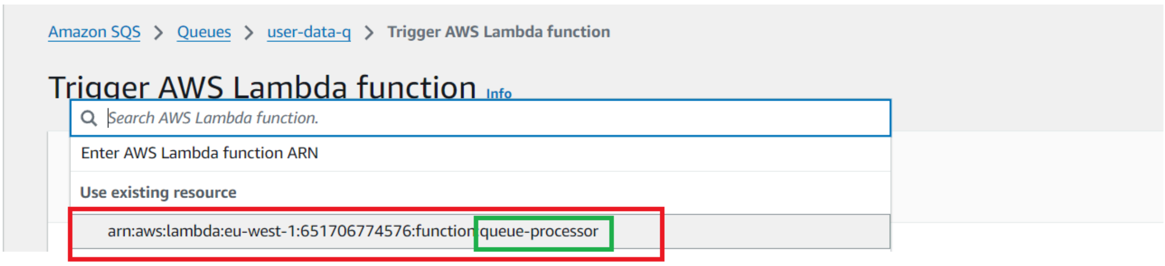
user-data-q

Details Info

Name: user-data-q
Type: Standard
ARN: arn:aws:sqs:eu-west-1:651706774576:user-data-q
Encryption: Amazon SQS key (SSE-SQS)
URL: https://sqs.eu-west-1.amazonaws.com/651706774576/user-data-q
Dead-letter queue: -

Lambda triggers (1) Info

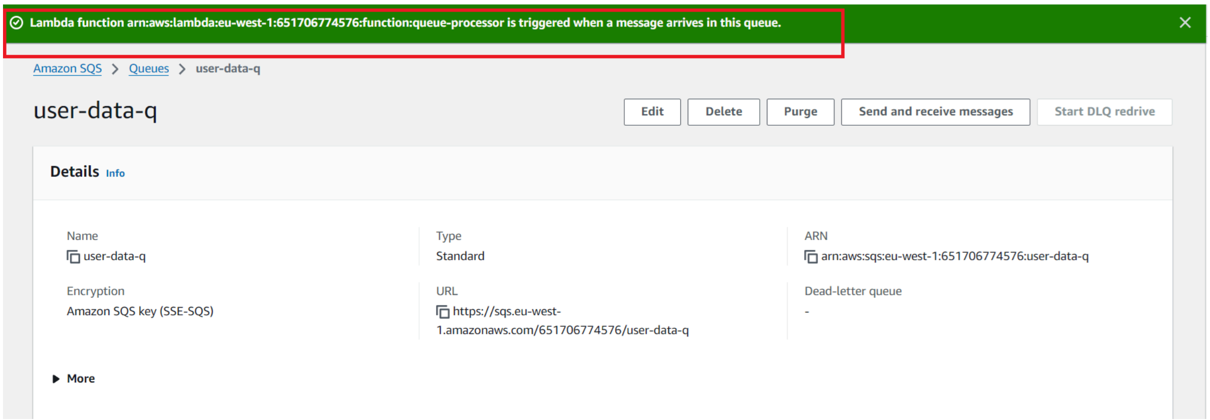
Click the **choose a function** drop down, and select the function we made earlier, and then hit **save.**


Amazon SQS > Queues > user-data-q > Trigger AWS Lambda function
Search AWS Lambda function.
Enter AWS Lambda function ARN
Use existing resource

Trigger AWS Lambda function Info

arn:aws:lambda:eu-west-1:651706774576:function:queue-processor

When we hit save you are brought back to your queue overview page and should see this message!



Lambda function arn:aws:lambda:eu-west-1:651706774576:function:queue-processor is triggered when a message arrives in this queue.

user-data-q

Details Info

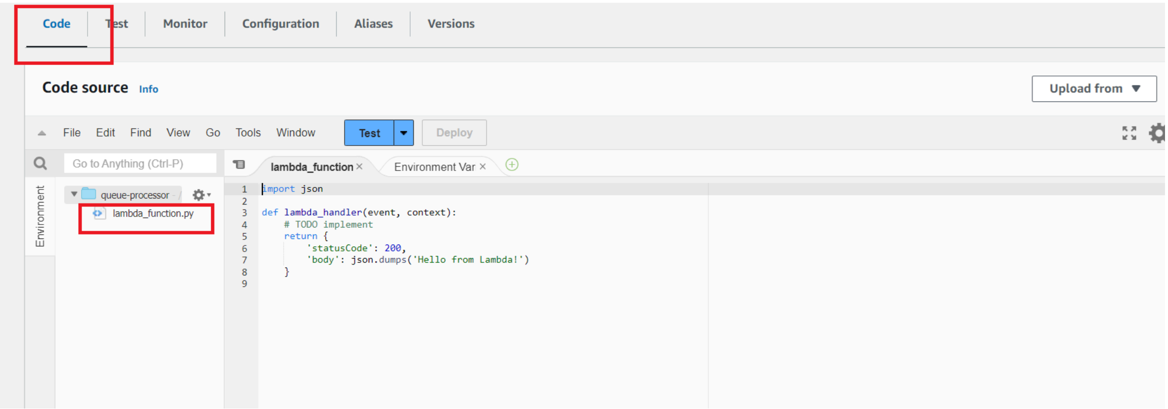
Name: user-data-q
Type: Standard
ARN: arn:aws:sqs:eu-west-1:651706774576:user-data-q
Encryption: Amazon SQS key (SSE-SQS)
URL: https://sqs.eu-west-1.amazonaws.com/651706774576/user-data-q
Dead-letter queue: -

Trigger done!

Step 5.4 Adding code to our Lambda

Now that our lambda triggers when there is a message in the queue, our lambda actually has to do some stuff to process the message.

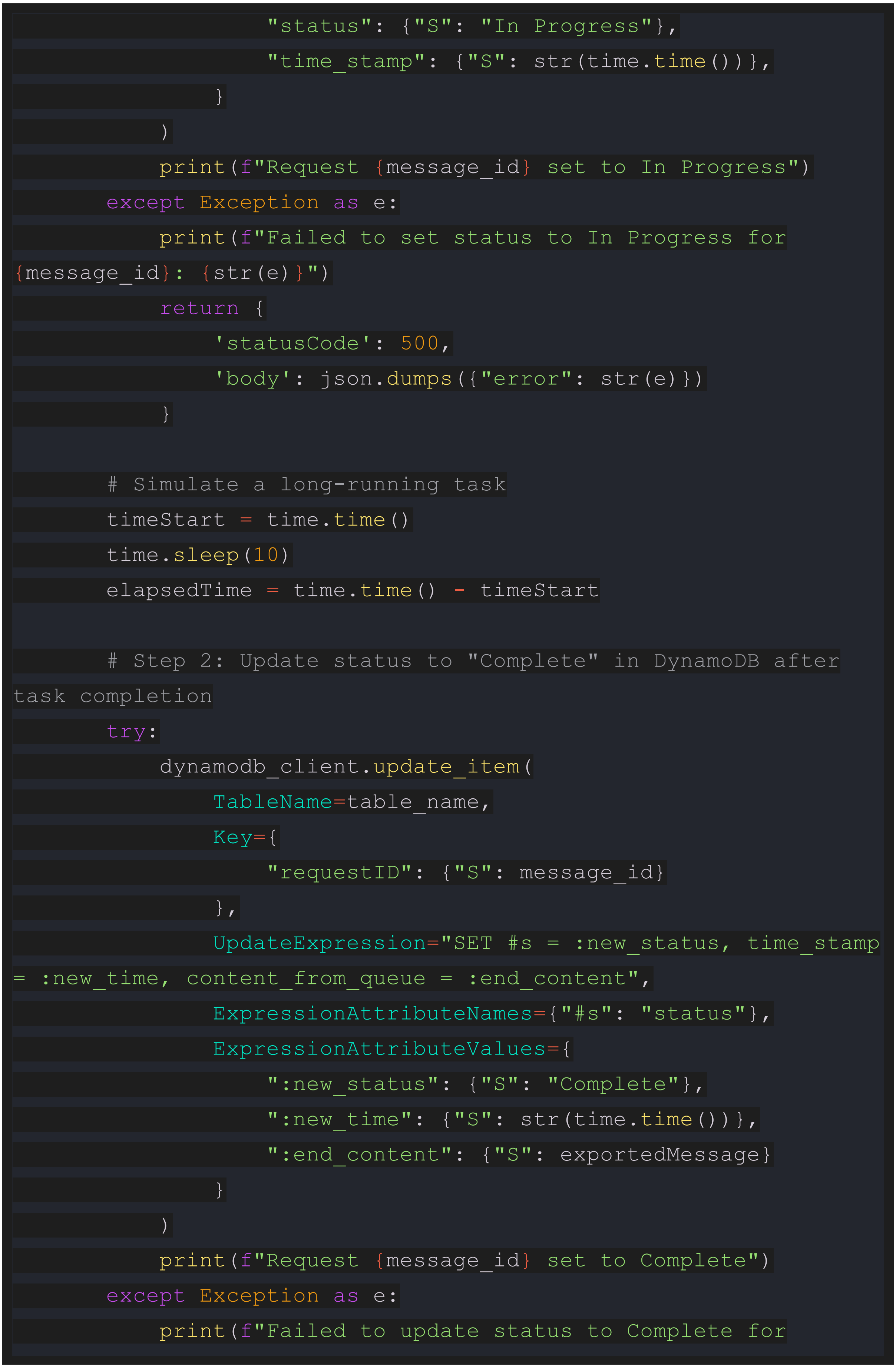
So go to **your Lambda**, scroll down to the **code** section where you will see the code displayed for the **lambda_function.py** file.



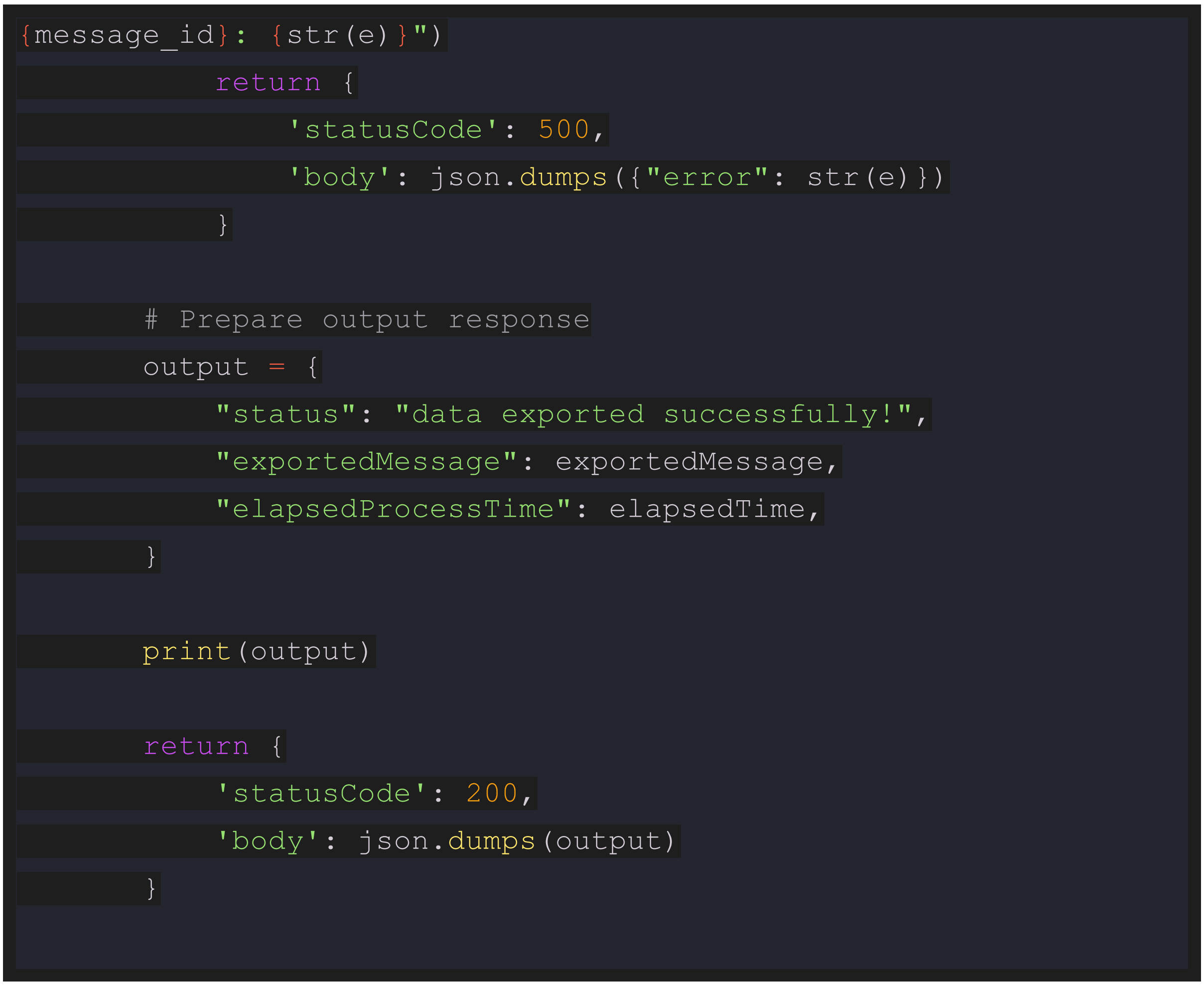
```python
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }

```

In here paste the following code

```python
import json, time, boto3, random

def lambda_handler(event, context):
    # Initialize DynamoDB client
    dynamodb_client = boto3.client("dynamodb")
    table_name = "my-logs-table"

    for record in event['Records']:
        exportedMessage = record['body']
        message_id = exportedMessage
        # Log consumption of the message
        print(f"Consuming from queue: {exportedMessage}")

        # Step 1: Set status to "In Progress" in DynamoDB
        try:
            dynamodb_client.put_item(
                TableName=table_name,
                Item={
                    "requestID": {"S": message_id},
                    "content_from_queue": {"S": exportedMessage},
```

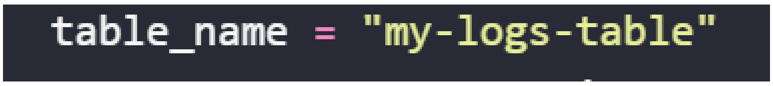


```
                    "status": {"S": "In Progress"},
                    "time_stamp": {"S": str(time.time())},
                }
            )
            print(f"Request {message_id} set to In Progress")
        except Exception as e:
            print(f"Failed to set status to In Progress for
{message_id}: {str(e)}")
            return {
                'statusCode': 500,
                'body': json.dumps({"error": str(e)})
            }

        # Simulate a long-running task
        timeStart = time.time()
        time.sleep(10)
        elapsedTime = time.time() - timeStart

        # Step 2: Update status to "Complete" in DynamoDB after
task completion
        try:
            dynamodb_client.update_item(
                TableName=table_name,
                Key={
                    "requestID": {"S": message_id}
                },
                UpdateExpression="SET #s = :new_status, time_stamp
= :new_time, content_from_queue = :end_content",
                ExpressionAttributeNames={"#s": "status"},
                ExpressionAttributeValues={
                    ":new_status": {"S": "Complete"},
                    ":new_time": {"S": str(time.time())},
                    ":end_content": {"S": exportedMessage}
                }
            )
            print(f"Request {message_id} set to Complete")
        except Exception as e:
            print(f"Failed to update status to Complete for
```



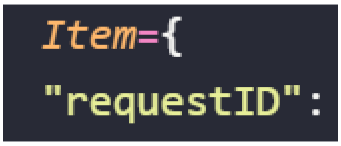
```
{message_id}: {str(e)}")
            return {
                'statusCode': 500,
                'body': json.dumps({"error": str(e)})
            }

        # Prepare output response
        output = {
            "status": "data exported successfully!",
            "exportedMessage": exportedMessage,
            "elapsedProcessTime": elapsedTime,
        }

        print(output)

        return {
            'statusCode': 200,
            'body': json.dumps(output)
        }
```

IMPORTANT: Make sure the table_name variable in the code is set to the name of your dynamo table.



```
table_name = "my-logs-table"
```

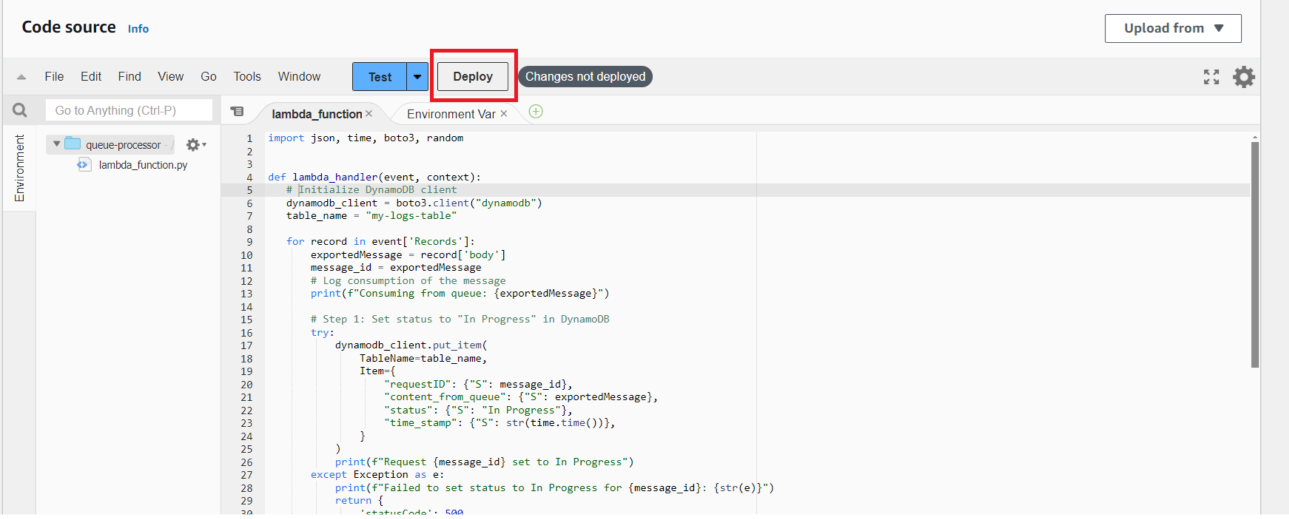
And that the first key in the Item variable is the name of the partition key you created when setting up your dynamo table



```
Item={
"requestID":
```

(if you used the names in the screenshots for your table & partition key you won’t have to make any changes😁)

Now the code is in there, we click **Deploy**!



```python
import json, time, boto3, random


def lambda_handler(event, context):
    # Initialize DynamoDB client
    dynamodb_client = boto3.client("dynamodb")
    table_name = "my-logs-table"

    for record in event['Records']:
        exportedMessage = record['body']
        message_id = exportedMessage
        # Log consumption of the message
        print(f"Consuming from queue: {exportedMessage}")

        # Step 1: Set status to "In Progress" in DynamoDB
        try:
            dynamodb_client.put_item(
                TableName=table_name,
                Item={
                    "requestID": {"S": message_id},
                    "content_from_queue": {"S": exportedMessage},
                    "status": {"S": "In Progress"},
                    "time_stamp": {"S": str(time.time())},
                }
            )
            print(f"Request {message_id} set to In Progress")
        except Exception as e:
            print(f"Failed to set status to In Progress for {message_id}: {str(e)}")
            return {
                'statusCode': 500
```

What does this code do?

- Prints the message sent in the queue to CloudWatch Logs
- Adds an entry to your dynamoDB table with fields such as the requestID, content_from_queue, timestamp and status. Status here is set to "In progress"
- Then the code then simulates a long running process (sleeping for 10 seconds)
- After the task is completed, the code updates the entry that was just made in your dynamoDB table with a status of "Complete"
- Then prints the lambda function output to the CloudWatch logs.

LAMBDA DONE!

Step 6: Testing that our Lambda works, viewing logs in CloudWatch and Viewing entries in Dynamodb Table (10 mins)

This section is simple despite the big heading don't worry!

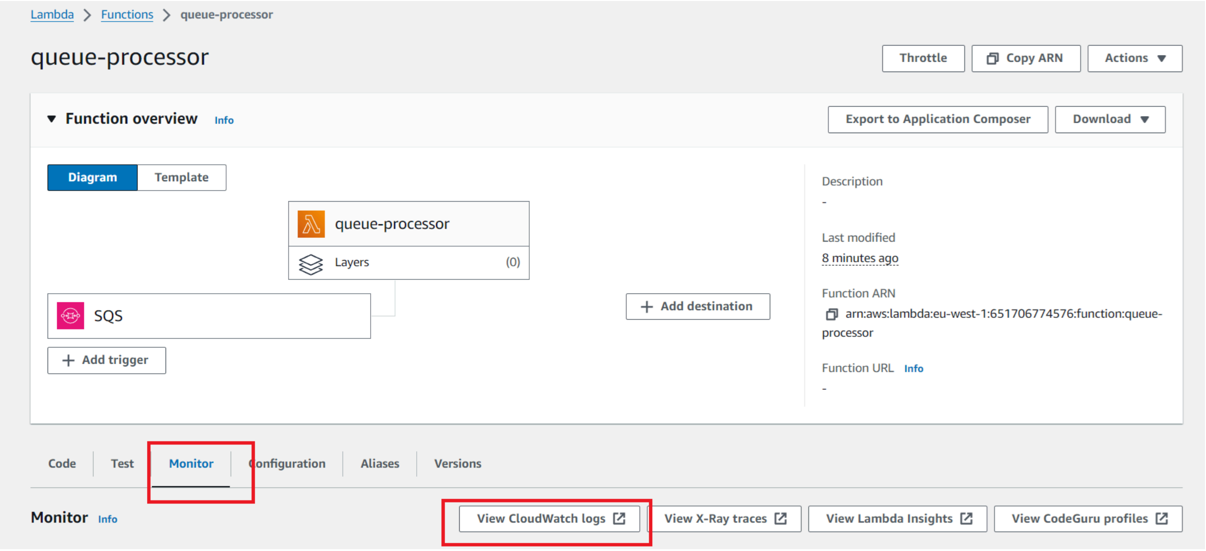
Step 6.1 Testing Lambda from the front end

Head back to the localhost page in your browser - if you stopped running your local server, just head back to your IDE and run the command `npm run dev`

Now click the **Send** button in the **Export Data Asynchronously** box.
You should get back the success message from SQS, now to check that the lambda ran!

Step 6.2 Viewing CloudWatch Logs

Head back to your Lambda page after sending a message using the Async Export Option front end and scroll down to the **Monitor Section**



queue-processor

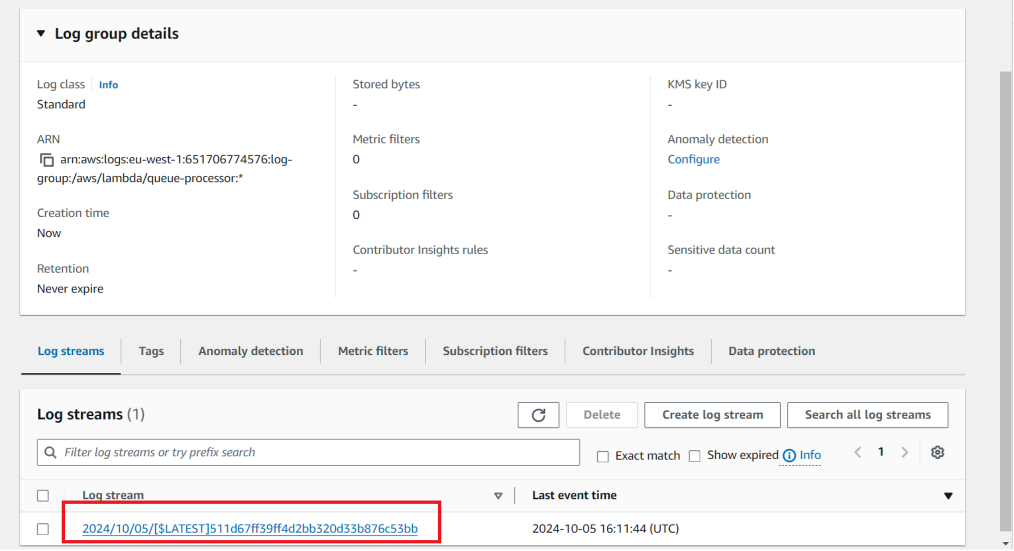
Description
-

Last modified
8 minutes ago

Function ARN
arn:aws:lambda:eu-west-1:651706774576:function:queue-processor

Function URL Info
-

This will bring you to the CloudWatch Log groups page. Scroll down and click the **very top log** (the one at the top is a log from your most recent message send to the queue)



Log class Info
Standard

ARN
arn:aws:logs:eu-west-1:651706774576:log-group:/aws/lambda/queue-processor:*

Creation time
Now

Retention
Never expire

Stored bytes
-

Metric filters
0

Subscription filters
0

Contributor Insights rules
-

KMS key ID
-

Anomaly detection
Configure

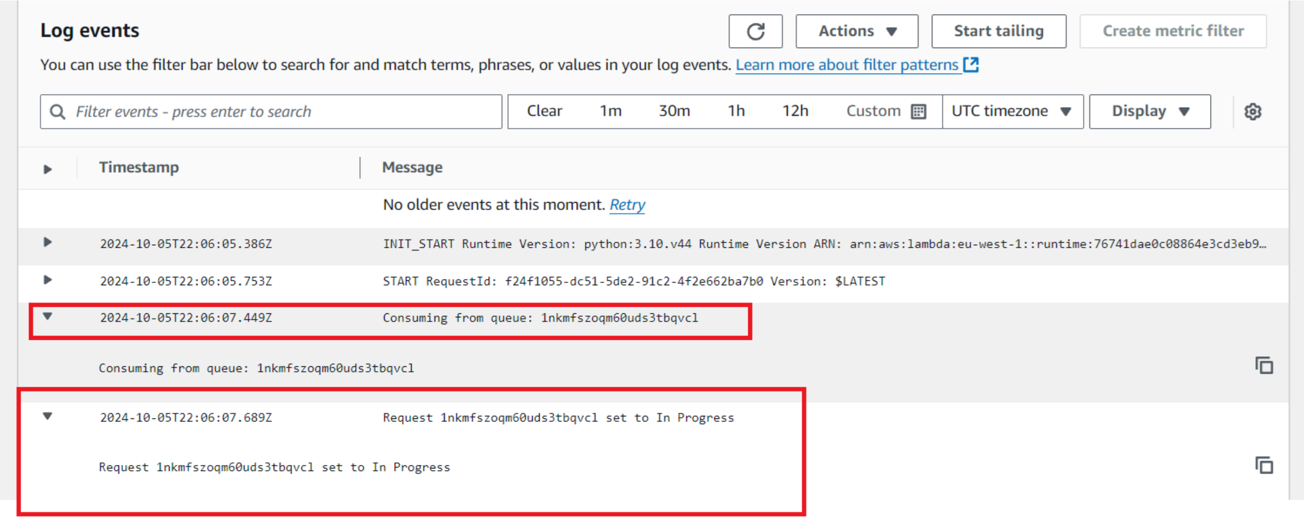
Data protection
-

Sensitive data count
-

Log streams (1)

Log stream	Last event time
2024/10/05/[$LATEST]511d67ff39ff4d2bb320d33b876c53bb	2024-10-05 16:11:44 (UTC)

Once you click into this log you will see “**consuming from queue**” that we printed in the lambda code.
Now you will have to **wait 10 seconds** for the rest of the logs to come through, as this lambda is running a long process! So **click refresh a few times**

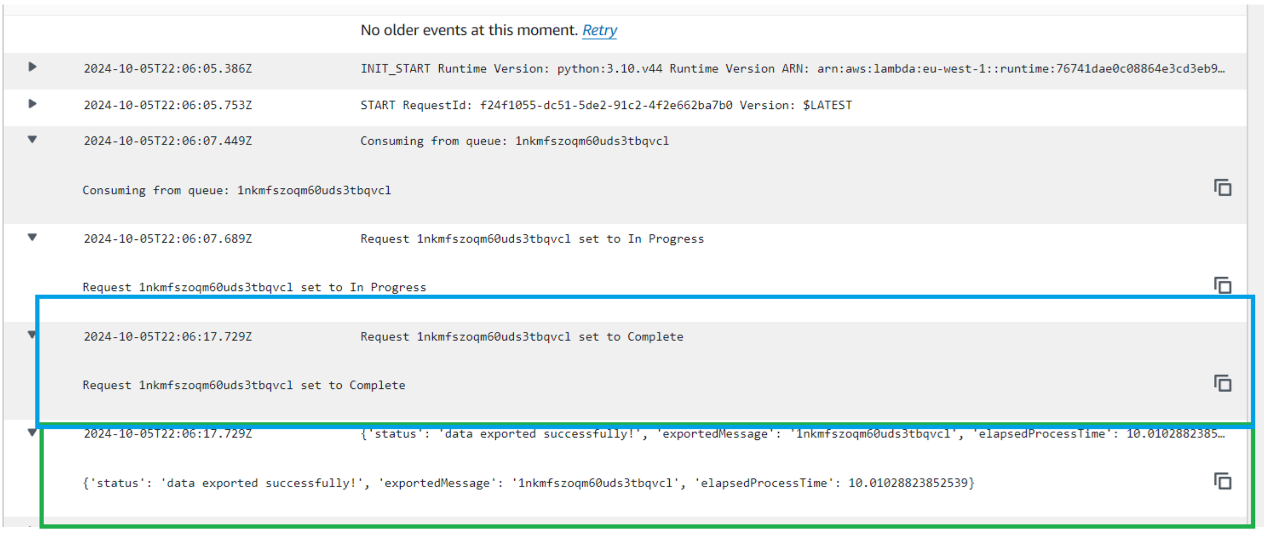


You can use the filter bar below to search for and match terms, phrases, or values in your log events. Learn more about filter patterns

```
   Timestamp                  Message
                              No older events at this moment. Retry
▶  2024-10-05T22:06:05.386Z   INIT_START Runtime Version: python:3.10.v44 Runtime Version ARN: arn:aws:lambda:eu-west-1::runtime:76741dae0c08864e3cd3eb9…
▶  2024-10-05T22:06:05.753Z   START RequestId: f24f1055-dc51-5de2-91c2-4f2e662ba7b0 Version: $LATEST
▼  2024-10-05T22:06:07.449Z   Consuming from queue: 1nkmfszoqm60uds3tbqvcl

   Consuming from queue: 1nkmfszoqm60uds3tbqvcl
▼  2024-10-05T22:06:07.689Z   Request 1nkmfszoqm60uds3tbqvcl set to In Progress

   Request 1nkmfszoqm60uds3tbqvcl set to In Progress
```

Then after 10 seconds, you **should see the results** from the long running process

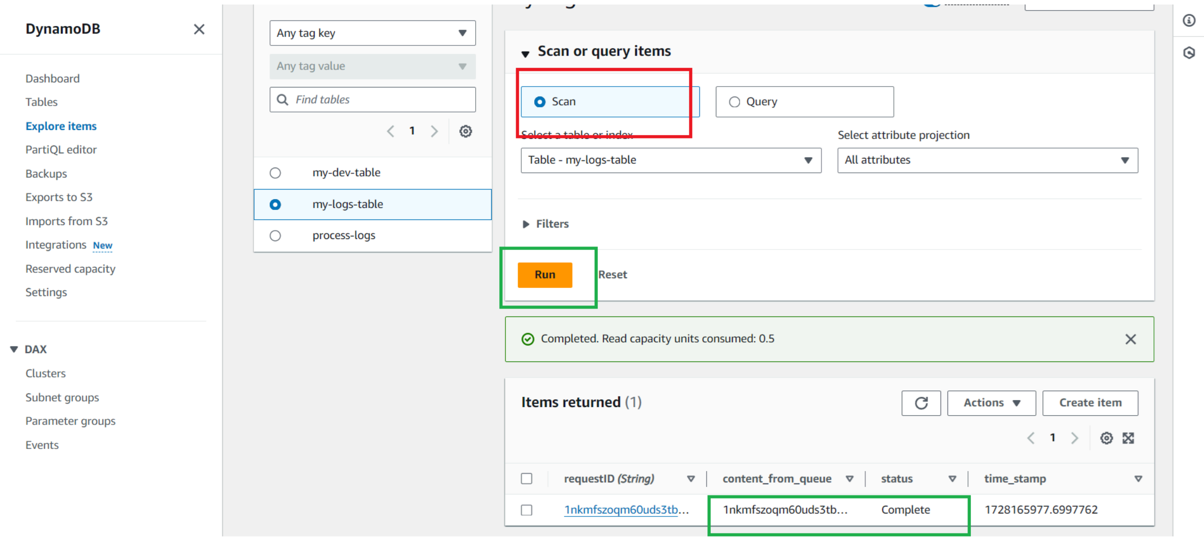
- The status of the export being updated to “complete” (blue)
- The success response from the lambda itself (green)



```
                              No older events at this moment. Retry
▶  2024-10-05T22:06:05.386Z   INIT_START Runtime Version: python:3.10.v44 Runtime Version ARN: arn:aws:lambda:eu-west-1::runtime:76741dae0c08864e3cd3eb9…
▶  2024-10-05T22:06:05.753Z   START RequestId: f24f1055-dc51-5de2-91c2-4f2e662ba7b0 Version: $LATEST
▼  2024-10-05T22:06:07.449Z   Consuming from queue: 1nkmfszoqm60uds3tbqvcl

   Consuming from queue: 1nkmfszoqm60uds3tbqvcl
▼  2024-10-05T22:06:07.689Z   Request 1nkmfszoqm60uds3tbqvcl set to In Progress

   Request 1nkmfszoqm60uds3tbqvcl set to In Progress
▼  2024-10-05T22:06:17.729Z   Request 1nkmfszoqm60uds3tbqvcl set to Complete

   Request 1nkmfszoqm60uds3tbqvcl set to Complete
▼  2024-10-05T22:06:17.729Z   {'status': 'data exported successfully!', 'exportedMessage': '1nkmfszoqm60uds3tbqvcl', 'elapsedProcessTime': 10.0102882385…

   {'status': 'data exported successfully!', 'exportedMessage': '1nkmfszoqm60uds3tbqvcl', 'elapsedProcessTime': 10.01028823852539}
```

Step 6.3 Viewing entries in the Dynamo table

Head to your Dynamo Table, and then click **explore table items**, then run a quick **scan** to see the items in your table.
You should see an entry with the **same message** as the one **from Cloudwatch**, with a **status of complete**



● Scan ○ Query

Select a table or index: Table - my-logs-table

Select attribute projection: All attributes

Completed. Read capacity units consumed: 0.5

Items returned (1)

requestID (String)	content_from_queue	status	time_stamp
1nkmfszoqm60uds3tb...	1nkmfszoqm60uds3tb...	Complete	1728165977.6997762

STEP 6 DONE!

Step 7: Delivering the data to the user (10 mins)

Step 7.1 Let’s set up our frontend to poll for the completion of our request

Replace your **taskmanager.tsx file** in its entirety with the following

```
"use client";
import { useState, useEffect } from 'react';
import { CSSProperties } from 'react';

export default function TaskManager() {
 const [batchId, setBatchId] = useState<string | null>(null);
 const [batchStatus, setBatchStatus] = useState<string>("idle");
 const [batchContent, setBatchContent] = useState<string>("");

 const checkStatus = async () => {
   if (batchId) {
     const response = await
fetch(`/api/checkStatus?batchId=${batchId}`);
     const data = await response.json();

     if (data.status === "Complete") {
       alert("Done!");
       setBatchStatus("complete");
       setBatchContent(data.content_from_queue);
     }
   }
 };

 const handleSubmit = async () => {
   const response = await fetch('/api/sendMessageAsync', {
     method: 'POST',
     headers: {
       'Content-Type': 'application/json',
     },
   }).then(data => data.json());

   if (response.batchId) {
     setBatchId(response.batchId);
     setBatchStatus("in-progress");
     alert(`Message sent: ${response.batchId}`);
   } else {
     alert("Failed to send message to SQS");
   }
 };

 const handleSubmitSync = async () => {

   const response = await fetch('/api/sendMessageSync', {
     method: 'POST',
     headers: {
       'Content-Type': 'application/json',
     },
   }).then(data => data.json());
     alert(JSON.stringify(response));
 };

 useEffect(() => {
   let interval: NodeJS.Timeout | null = null;

   if (batchStatus === "in-progress") {
     interval = setInterval(checkStatus, 2500);
   }

   return () => {
     if (interval) {
       clearInterval(interval);
     }
   };
 }, [batchId, batchStatus]);

 return (
   <div style={styles.wrapper}>
     <div style={styles.containerTwo}>
       <div style={styles.container}>
         <h1 style={styles.title}>Export Data Asynchronously:</h1>

         <button style={styles.button}
onClick={handleSubmit}>Send</button>
         {batchStatus === "in-progress" && <p
style={styles.title}>Processing...</p>}
         {batchStatus === "complete" && <p
style={styles.title}>Data: {batchContent}</p>}

       </div>

       <div style={styles.container}>
         <h1 style={styles.title}>Export Data Synchronously:</h1>
         <button style={styles.button}
onClick={handleSubmitSync}>Send</button>
       </div>
     </div>
   </div>
 );
}
const styles: { [key: string]: CSSProperties } = {
 wrapper: {
   display: 'flex',
   justifyContent: 'center',
   alignItems: 'center',
   minHeight: '100vh',
   backgroundColor: '#f0f0f0',
   padding: '20px',
 },
 containerTwo: {
   display: 'flex',
   flexDirection: 'row',
   justifyContent: 'space-evenly',
   alignItems: 'center',
   gap: '2rem',
   backgroundColor: '#ffffff',
   padding: '30px',
   borderRadius: '15px',
   boxShadow: '0 8px 16px rgba(0, 0, 0, 0.1)',
 },
 container: {
   display: 'flex',
   flexDirection: 'column',
   justifyContent: 'center',
   alignItems: 'center',
   backgroundColor: '#f8f8f8',
   padding: '30px',
   borderRadius: '12px',
   width: '350px',
   boxShadow: '0 4px 12px rgba(0, 0, 0, 0.1)',
 },
 title: {
   fontSize: '20px',
   fontWeight: 'bold',
   marginBottom: '15px',
   color: '#333',
 },
 button: {
   padding: '12px 24px',
   backgroundColor: '#0070f3',
   color: '#fff',
   border: 'none',
   borderRadius: '8px',
   cursor: 'pointer',
   fontSize: '16px',
   fontWeight: '600',
   transition: 'background-color 0.3s ease',
 },
 buttonHover: {
   backgroundColor: '#005bb5',
 },
 statusInProgress: {
   marginTop: '20px',
   fontSize: '16px',
   fontWeight: '600',
   color: '#FFA500', // Orange for "in-progress"
 },
 statusComplete: {
   marginTop: '20px',
   fontSize: '16px',
   fontWeight: '600',
   color: '#28a745', // Green for "complete"
 },
};
```

Step 7.2 Creating our polling endpoint

And let’s create our **checkStatus** route in **/api/checkStatus/route.tsx**

This route checks DynamoDB to see when the status has been set to **Complete**

We will need to run the following command to integrate dynamo into our project.

```
npm install @aws-sdk/client-dynamodb
```

```
import { DynamoDBClient, GetItemCommand } from
"@aws-sdk/client-dynamodb";
import { NextRequest, NextResponse } from "next/server";

const dynamodb = new DynamoDBClient({ region: "eu-west-1" });

export async function GET(req: NextRequest) {
   const { searchParams } = new URL(req.url);
   const batchId = searchParams.get("batchId");

   if (!batchId) {
       return NextResponse.json(
           { error: "batchId is required" },
           { status: 400 }
       );
   }

   try {
       const command = new GetItemCommand({
           TableName: "my-logs-table",
           Key: {
               requestID: { S: batchId },
           },
       });

       const response = await dynamodb.send(command);

       if (!response.Item) {
           return NextResponse.json({ error: "Batch not found" },
{ status: 404 });
       }
```

```
        const status = response.Item.status.S;
        const data = {
            status: status,
            content_from_queue: response.Item.content_from_queue.S,
        };

        return NextResponse.json(data);
    } catch (error) {
        console.error("Error fetching batch status:", error);
        return NextResponse.json(
            { error: "Failed to fetch batch status" },
            { status: 500 }
        );
    }
}
```

With this, we are now able to observe the data going from a **Processing…** to a **Completed** state.

Step 8: Securing our APIs (10 mins)

Our APIs are secured on the frontend as we are using the Nextauth library to control what our end user can and cannot see. However, **our backend remains vulnerable** as any malicious user could bypass this and send unauthorised requests directly to our api.

To ensure that our apis are protected, we need to ensure that all **API requests check for a session** and that the backend can verify the request before any data is processed.

Step 8.1 Generating a NextAuth token

Run the following command in your terminal to generate a random Base64 string.

```
node -e
```

```
"console.log(require('crypto').randomBytes(32).toString('base64'))"
```

Next, copy the string from the terminal, we will be using this as an env variable for NextAuth.

Step 8.2 Creating a new env variable

Head over to the .env file from before, create a new variable

```
NEXTAUTH_SECRET=[YOUR RANDOM BASE64 STRING]
```

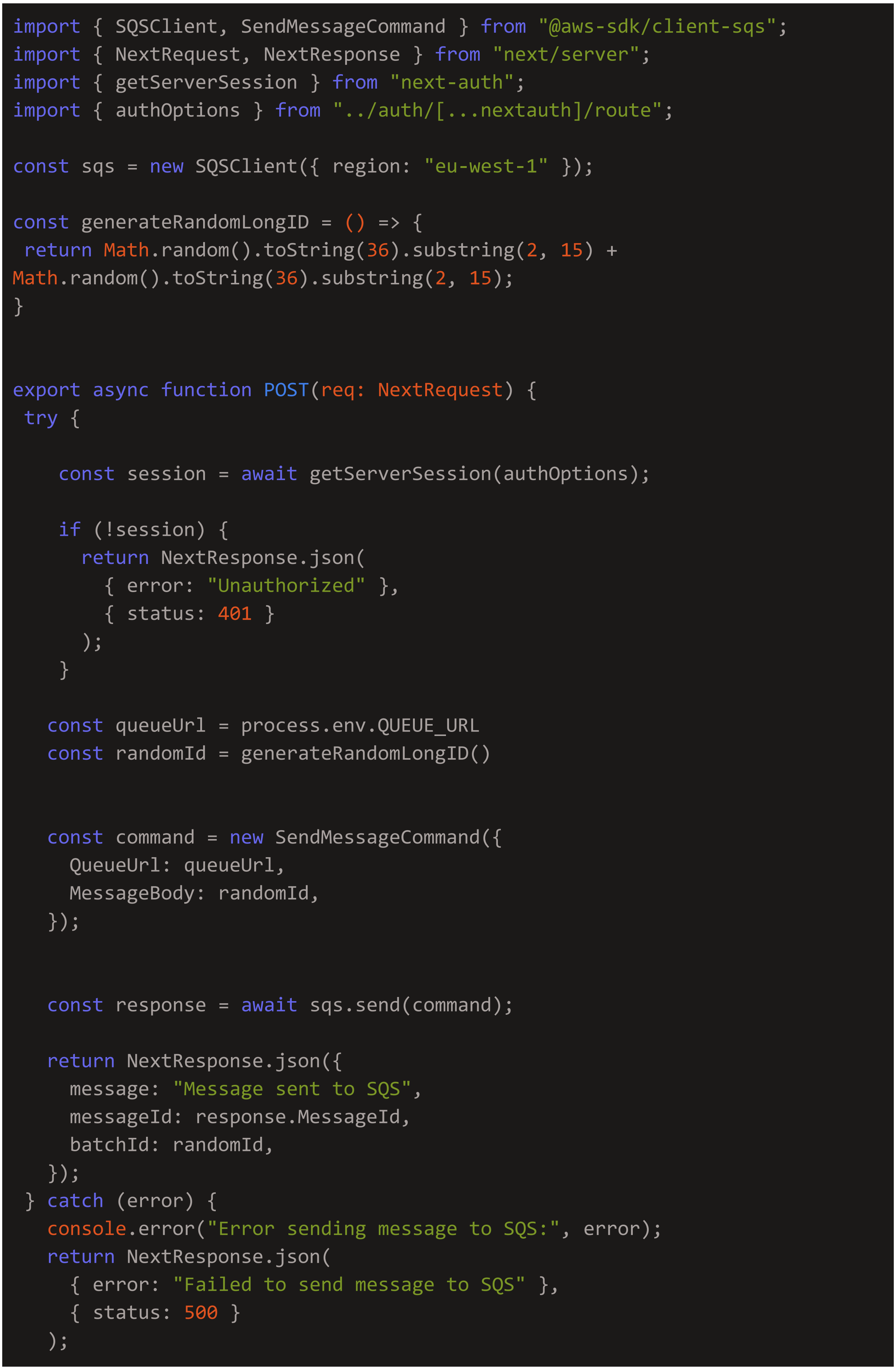
Step 8.3 Updating your nextAuth route

1. Head back to the file at
   src\app\api\auth\[...nextauth]\route.ts
2. Modify the file to use the nextauth secret.

```
import NextAuth from "next-auth/next";
import GoogleProvider from "next-auth/providers/google";

export const authOptions = {
  providers: [
    GoogleProvider({
      clientId: process.env.GOOGLE_CLIENT_ID ?? "",
      clientSecret: process.env.GOOGLE_CLIENT_SECRET ?? "",
    }),
  ],
  secret: process.env.NEXTAUTH_SECRET,
};

const handler = NextAuth(authOptions);

export { handler as GET, handler as POST };
```

Step 8.4 Updating sendMessageAsync

1. We will head back into the route.ts for the sendMessageAsync and modify it to check the authOptions for the server session.
2. Copy the following code into your file at the path: **src\app\api\sendMessageAsync\route.ts**

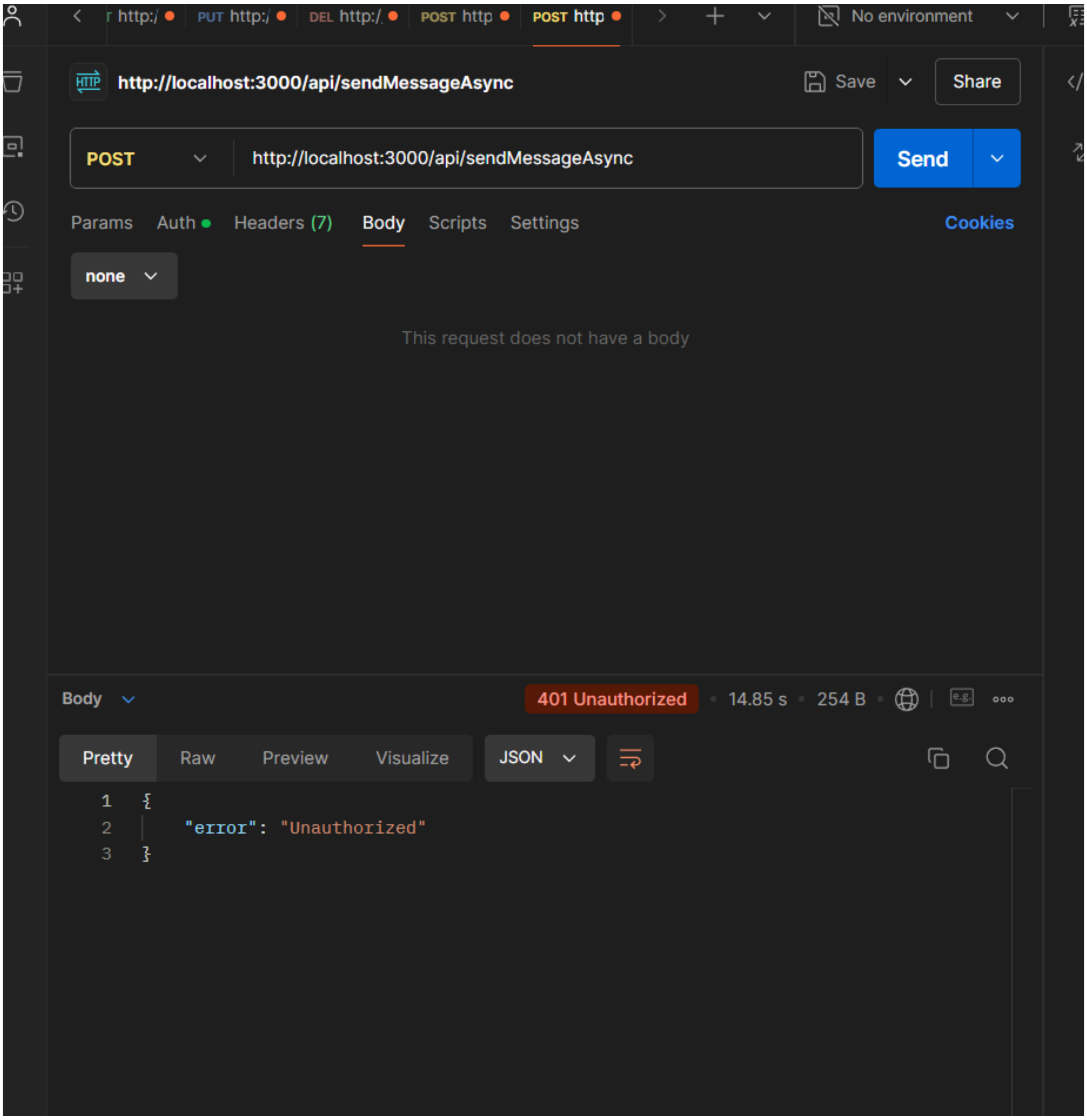


```
import { SQSClient, SendMessageCommand } from "@aws-sdk/client-sqs";
import { NextRequest, NextResponse } from "next/server";
import { getServerSession } from "next-auth";
import { authOptions } from "../auth/[...nextauth]/route";

const sqs = new SQSClient({ region: "eu-west-1" });

const generateRandomLongID = () => {
 return Math.random().toString(36).substring(2, 15) +
Math.random().toString(36).substring(2, 15);
}


export async function POST(req: NextRequest) {
 try {

    const session = await getServerSession(authOptions);

    if (!session) {
      return NextResponse.json(
        { error: "Unauthorized" },
        { status: 401 }
      );
    }

   const queueUrl = process.env.QUEUE_URL
   const randomId = generateRandomLongID()


   const command = new SendMessageCommand({
     QueueUrl: queueUrl,
     MessageBody: randomId,
   });


   const response = await sqs.send(command);

   return NextResponse.json({
     message: "Message sent to SQS",
     messageId: response.MessageId,
     batchId: randomId,
   });
 } catch (error) {
   console.error("Error sending message to SQS:", error);
   return NextResponse.json(
     { error: "Failed to send message to SQS" },
     { status: 500 }
   );
```

```
  }
}
```

Step 8.5 Testing Endpoint

1. Now that our api backend is checking for a session before proceeding with the request, we can test this in postman. Where we should expect to get an unauthorised request.



http://localhost:3000/api/sendMessageAsync

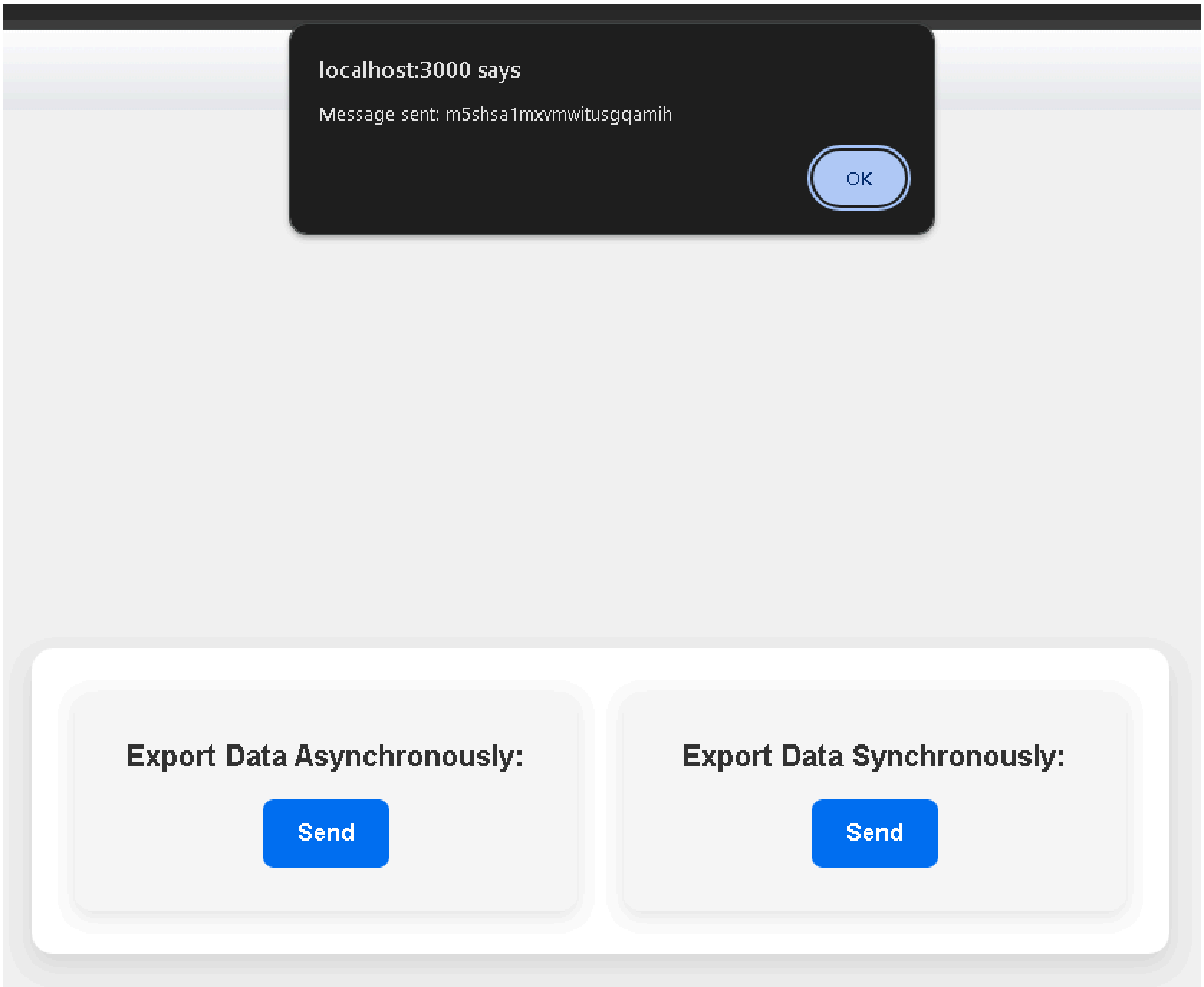
POST http://localhost:3000/api/sendMessageAsync

This request does not have a body

401 Unauthorized • 14.85 s • 254 B

```
{
    "error": "Unauthorized"
}
```

And if we test this endpoint after logging into our webpage:



localhost:3000 says

Message sent: m5shsa1mxvmwitusgqamih

Export Data Asynchronously:

Export Data Synchronously:

FINAL STEP ⚠

Please make sure to DELETE the access keys you made on AWS console (look at step 2.3 to find where they are)

Why?

- Because if anyone were to get these keys somehow, they would be able to access literally every resource on your AWS account programmatically, and could cost you thousands 😃

Tools/Software Required:

- **Development:**
    - An IDE of your choice
    - Typescript, Python & Node
    - Next.js
    - AWS SDK for Java
- **Containers & Testing:**
    - Postman (for API testing)
- **AWS Services:**
    - AWS SQS
    - AWS Lambda
    - AWS DynamoDB
- **Authentication:**
    - Google Developer Console (for OAuth 2.0)
- **Front-End Development:**
    - HTML/CSS/JavaScript or a front-end framework (e.g., React, Angular)
- **Additional Tools:**
    -

Expected Outcomes:

By the end of this lab, students will:

- **Authentication & Authorization:**
    - Successfully implement OAuth 2.0 using Google Sign-In.
    - Securely pass OAuth tokens to authorise requests to the data export API.
- **API Development:**
    - Develop a synchronous data export API that simulates long-running processes.
    - Refactor the synchronous API into an asynchronous one using AWS SQS and Lambda.
- **AWS Integration:**
    - Utilise AWS SQS for queuing export requests.
    - Implement AWS Lambda functions to process export requests.
    - Log all requests and results in DynamoDB.
- **Front-End Development:**
    - Develop a front-end UI that displays export options only to authenticated users.
    - Successfully pass OAuth tokens from the UI to the back-end.
- **Performance Analysis:**
    - Understand and compare the performance differences between synchronous and asynchronous API handling.

- Manage long-running tasks securely and efficiently in a real-world scenario.