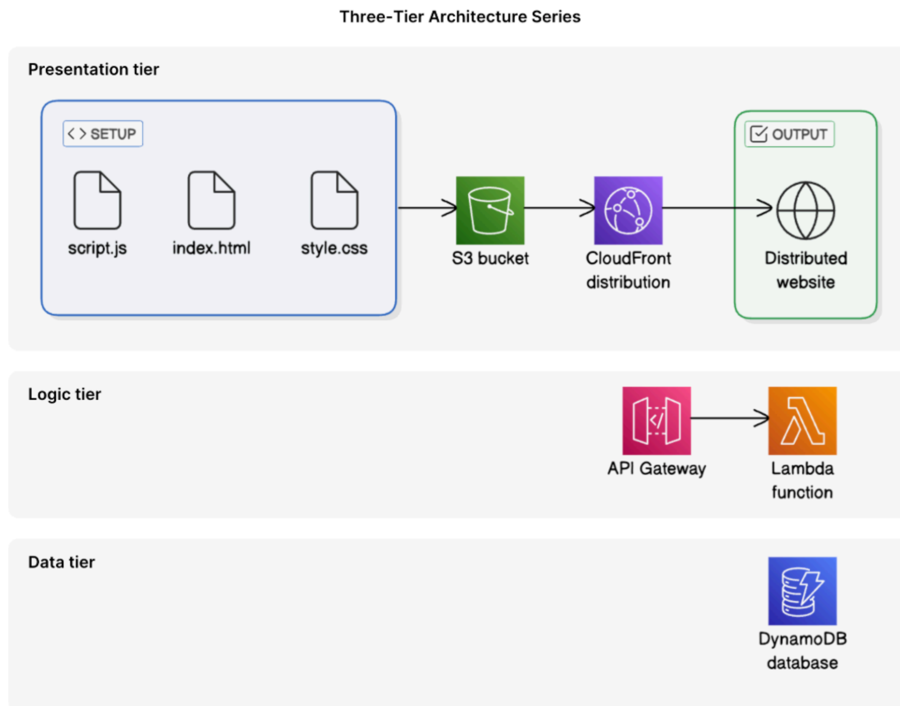
Build a Three-Tier Web App

By Chisom Uketui

In this project, I will be demonstrating how to set up a three-tier web app from scratch! I will start with the presentation layer, then set up the logic tier and finally set up the data tier before tying them all together. Below diagram shows the architecture of this project.



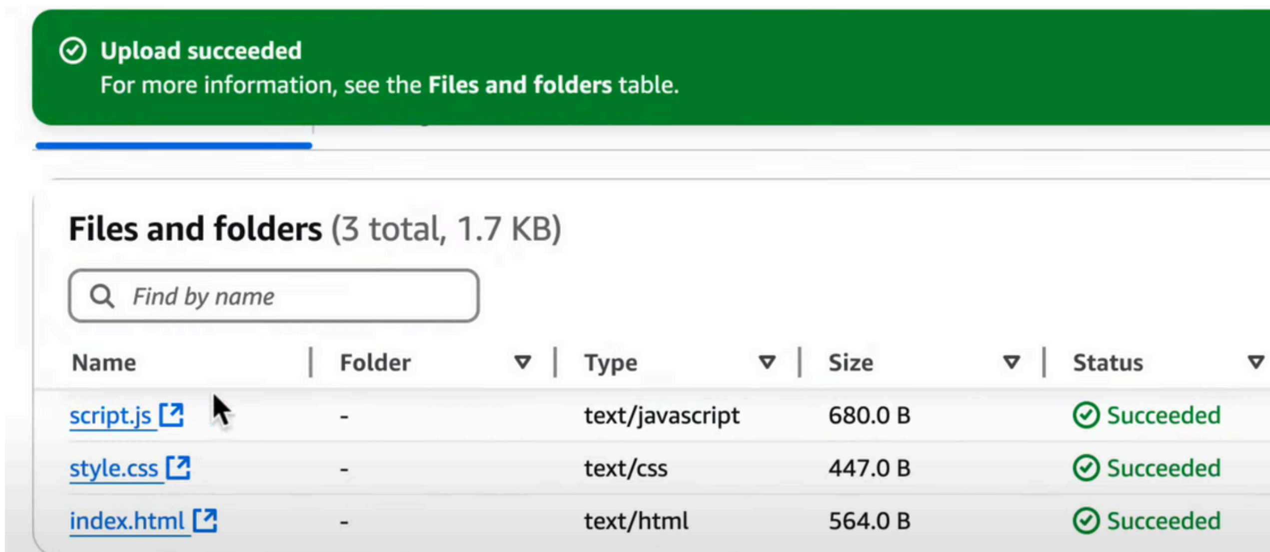
Set Up the Presentation Tier

In this step, for the presentation tier, I will set up how my website will be displayed and availabe to my end users. This is because the presentation tier is responsible for storing my website's files(Amazon S3) + website distribution (Amazon CloudFront).

Create an s3 bucket and upload my files (index.html, style.css and script.js.) in it.

Here is my html, just a simple webpage 🙂

Successfully uploaded my files onto my s3 bucket.



Create a CloudFront Distribution

Now that my website files are uploaded, I will then go ahead to create cloudfront distribution. Amazon CloudFront is a **Content Delivery Network (CDN),** which means it speeds up the distribution of your static and dynamic web content, such as .html, .css, .js, and image files.

For default root object, I’m using my index.html

Supported HTTP versions
Add support for additional HTTP versions. HTTP/1.0 and HTTP/1.1 are supported by default.
- [x] HTTP/2
- [ ] HTTP/3

Default root object - *optional*
The object (file name) to return when a viewer requests the root URL (/) instead of a specific object.
index.html

IPv6
- ( ) Off
- (•) On

CloudFront > Distributions > E1RKEAYAGOND4S

Successfully created new distribution.
To get in-depth monitoring information for your distribution's internet traffic, create an Internet Monitor

E1RKEAYAGOND4S View metrics

General | Security | Origins | Behaviors | Error pages | Invalidations | Tags | Logging

Details

Distribution domain name	ARN	Last modified
d1awba178apc9a.cloudfront.net	arn:aws:cloudfront::471112976395:distribution/E1RKEAYAGOND4S	Deploying

My s3 bucket policy now needs to be updated using cloudfront policy statement. This is to allow read access to CloudFront origin access control in our s3 bucket.

Amazon S3 > Buckets > nextwork-three-tier-name-numbers

Amazon S3
- General purpose buckets
- Directory buckets
- Table buckets New
- Access Grants
- Access Points
- Object Lambda Access Points
- Multi-Region Access Points
- Batch Operations
- IAM Access Analyzer for S3
- Block Public Access settings for this account
- Storage Lens

nextwork-three-tier-name-numbers Info

ta - Preview | Properties | Permissions | Metrics | Management | Access Poi

Permissions overview
Access finding
Access findings are provided by IAM external access analyzers. Learn more about How IAM analyzer findings work
View analyzer for us-east-1

Block public access (bucket settings) Edit
Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to all your S3 buckets and objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to your buckets or objects within, you can customize the individual settings below to suit your specific storage use cases. Learn

Amazon S3 > Buckets > nextwork-three-tier-name-numbers > Edit bucket policy

Bucket ARN

arn:aws:s3:::nextwork-three-tier-name-numbers

Policy

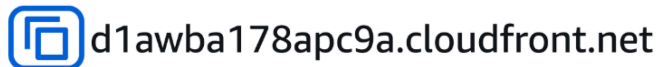
```
{
    "Version": "2008-10-17",
    "Id": "PolicyForCloudFrontPrivateContent",
    "Statement": [
        {
            "Sid": "AllowCloudFrontServicePrincipal",
            "Effect": "Allow",
            "Principal": {
                "Service": "cloudfront.amazonaws.com"
            },
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::nextwork-three-tier-name-numbers/*",
            "Condition": {
                "StringEquals": {
                  "AWS:SourceArn": "arn:aws:cloudfront::471112976395:distribution/E1RKEAYAG0ND4S"
                }
            }
        }
    ]
}
```

I will try to access my delivered website using the cloudfront distribution's URL. This distribution shoud because because I have also set up an origin access control that lets my S3 bucket restrict access to only my CloudFront distribution.

Cloudfront URL:

Details

Distribution domain name



Perfect!!! it works. This ticks off the presentation tier, which is all about the interface that my users will see and interact with.

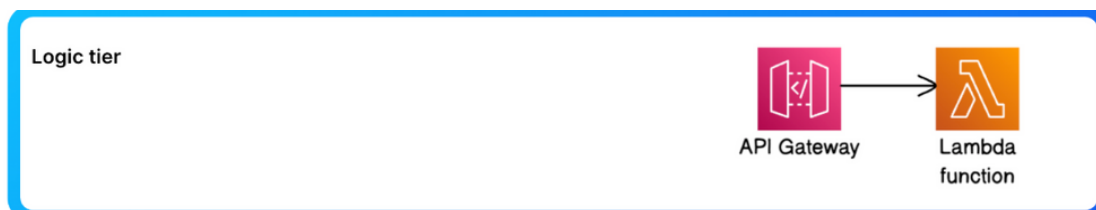
User Information

Enter User ID

Get User Data

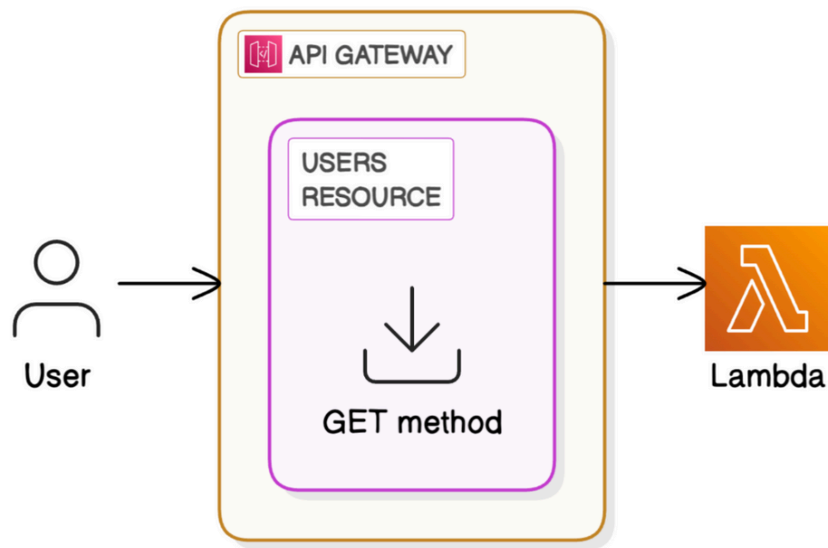
Set Up the Logic Tier

The logic tier is responsible for handling the brains of the application, such as fetching data from a database and performing calculations. In this project, my logic will be a simple **Lambda function** that retrieves user data from a DynamoDB table. I need a way to expose that functionality to the outside world, so I will use **API Gateway** to handle requests and route them to the right place.



In this step, I'm going to:

- Create a Lambda function to fetch data from a DynamoDB table.
- Write the code for my Lambda function.
- Create an API Gateway REST API.
- Create a resource and method to handle GET requests.
- Deploy the API to make it accessible.



Here is the code for my Lambda function:

Code source Info

Upload from

RetrieveUserData

EXPLORER

RETRIEVEUSERDATA

index.mjs

DEPLOY [UNDEPLOYED CHANGES]

You have undeployed changes.

Deploy (⇧⌘U)

Test (⇧⌘I)

TEST EVENTS [NONE SELECTED]

Create new test event

index.mjs > ...

```
1  // Load the AWS SDK for Node.js
2  const AWS = require('aws-sdk');
3  // Set the region
4  AWS.config.update({region: 'us-east-2'});  // Example: 'us-west-2'
5
6  // Create the DynamoDB service object
7  const ddb = new AWS.DynamoDB.DocumentClient();
8
9  exports.handler = async (event) => {
10     // Assume the incoming event is an API Gateway event with user ID passed as a
11     const userId = event.queryStringParameters.userId;
12
13     const params = {
14         TableName: 'UserData',
15         Key: {
16             'userId': userId
17         }
```

The Lambda function retrieves data by looking up the user ID(that our user will enter over the webpage) in dynamoDB. The AWS SDK is used in the function code so we can use template and libraries that lets us find the correct DynamoDB table + request data.

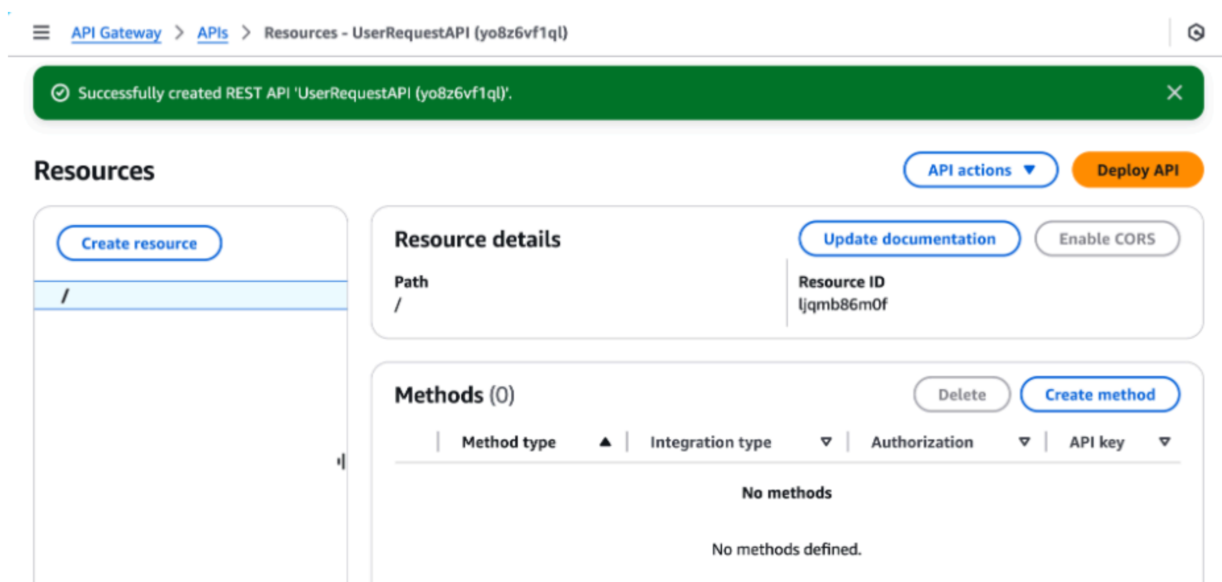
```
JS index.mjs ×
JS index.mjs > ...
 9  exports.handler = async (event) => {
23      const response = {
25          body: JSON.stringify(data.Item),
26          headers: {
27              'Content-Type': 'application/json'
28          }
29      };
30      return response;
31  } catch (err) {
32      console.error("Unable to retrieve data: ", err);
33      return {
34          statusCode: 500,
35          body: JSON.stringify({ message: "Failed to retrieve user data" }),
36          headers: {
37              'Content-Type': 'application/json'
38          }
39      };
40  }
```

Deployment successful

Set up API Gateway

Now that I have my Lambda function ready, I need a way to access it. This is where API Gateway comes in. An **API**, or Application Programming Interface, is a way for different software systems to talk to each other. It's like a messenger that carries requests and responses between systems.

In this project, I'm creating an API that carries requests from my user's browser to my Lambda function.



I will now create a resouce. API resources are endpoints that handle different parts of your API's functionalities.

For example, an API for a messaging app might have separate resources for retrieving messages and for retrieving user profiles.

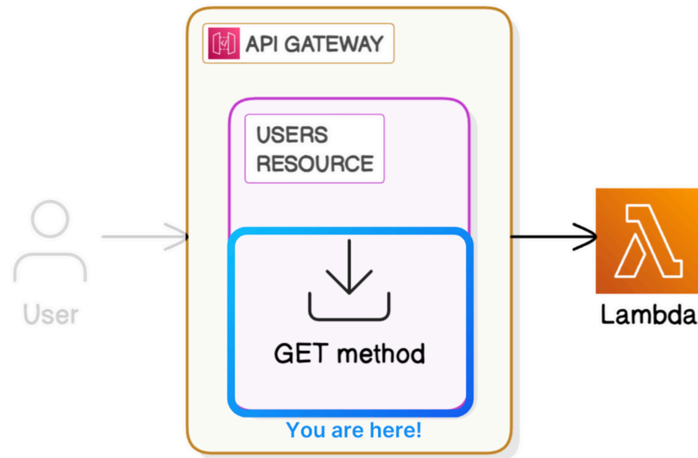
Resource details

Proxy resource Info

Proxy resources handle requests to all sub-resources. To create a proxy resource use a path parameter that ends with a plus sign, for example {proxy+}.

Resource path	Resource name
/	users

Set up an API Method



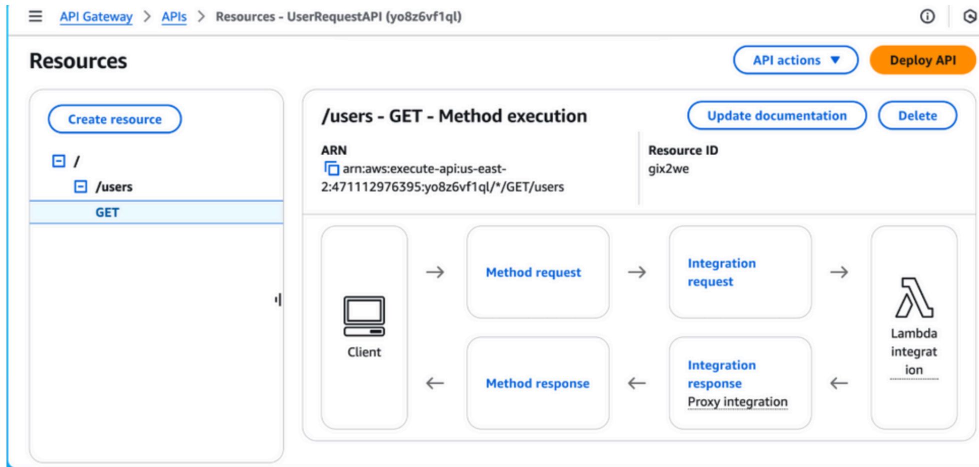
API methods are actions you can do in a resource.

They are based on standard HTTP methods, which are different commands that let you interact with data over the internet. E.g GET to retrieve, DELETE to remove data, POST to add etc. For this project, I will go with the GET method and Lambda integration option.

Method type

GET

- ANY
- DELETE
- GET ✓
- HEAD
- OPTIONS



Deploy the API in Prod stage:

In API Gateway, a stage is a snapshot of your API at a specific point in time.

API Gateway lets you deploy different versions of your API to different stages. This way, you can easily control who accesses what version of your API and when.

Deploy API

Create or select a stage where your API will be deployed. You can use the deployment history to revert or change the active deployment for a stage. Learn more

Stage

New stage

Stage name

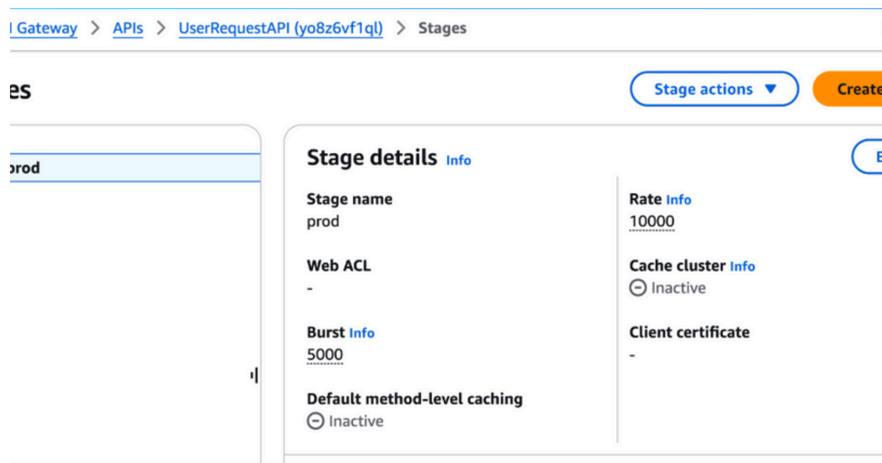
prod

A new stage will be created with the default settings. Edit your stage settings on the **Stage** page.

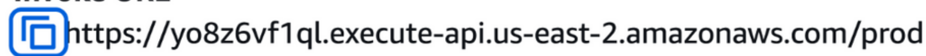
Deployment description

Cancel Deploy

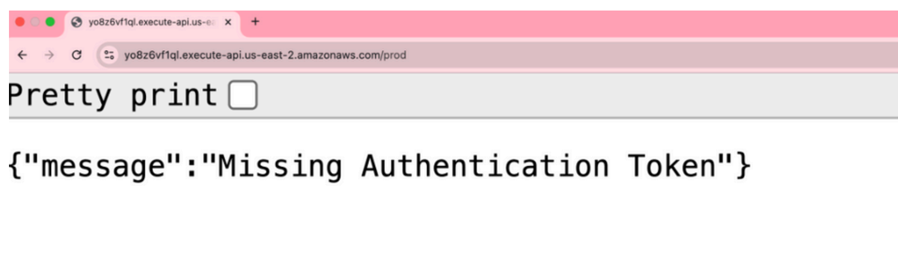
Deployment success!



Let’s visit my API using the invoke URL. In real world scenarios, developers use the prod stage's invoke URL into their live application's code, so users are using the live/production version of the API.



But, I got an error because I haven't set up my DynamoDB table yet. That's okay! We're getting to that next 😉



Set Up the Data Tier

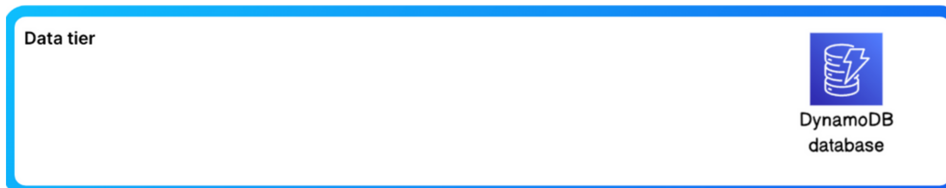
I've got website files distributed through CloudFront, and a Lambda function that's ready to retrieve data.

Now, let's put my API to use. The **data tier** is where I store all the data that my application uses.

I'll use DynamoDB to store some user data. Therefore:

In this step, I’m going to:

- Create a DynamoDB table.
- Add user data into my table.



DynamoDB(DDB) is my NoSQL database. It's fast, flexible, and perfect for storing user data.

Creating DDB table:

Partition key

The partition key is part of the table's primary key. It is a hash

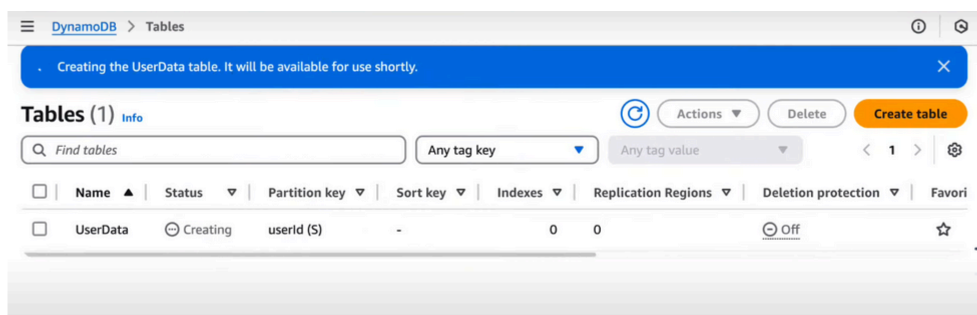
userId

1 to 255 characters and case sensitive.

The partition key for my DDB table is 'userId'. This means that when my table looks up for user data, it will look it up based on userId. Then, it can return all data(values) related to the item with that ID.

A partition key is the heart of how DynamoDB organizes data. Think of it as a label that you can use to group similar items. Under the hood, the partition key is how DynamoDB spreads out your data across different servers for quick access and efficient querying.

Every item in your table *must* have a unique partition key.



Create an item:

json Copy code

```json
{
  "userId": "1",
  "name": "Test User",
  "email": "test@example.com"
}
```

This JSON code defines a new item for my UserData table.

DynamoDB is **schemaless**, meaning you can add attributes as you need, and every item in your database can have a different set of attributes. This flexibility is one of the key benefits of using a NoSQL database like DynamoDB.

Create item

You can add, remove, or edit the attributes of an item. You can r

Attributes View DynamoDB JSON

```
1 {
2   "userId": {
3     "S": "1"
4   },
5   "name": {
6     "S": "Test User"
7   },
8   "email": {
9     "S": "test@example.com"
10  }
11 }
```

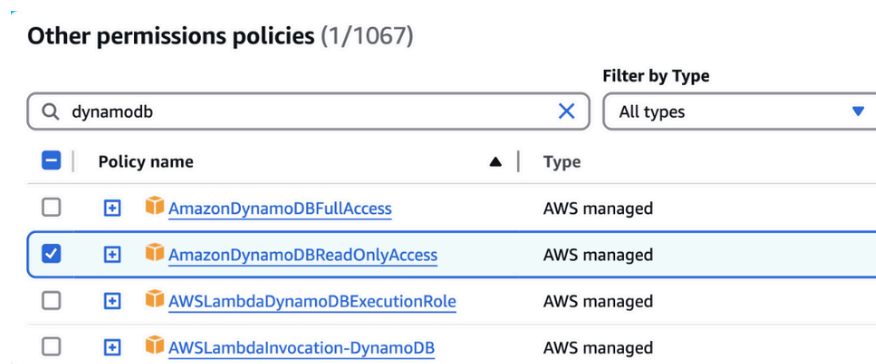
Items returned (1)

Actions ▼ Create item

< 1 >

	userId (String)	email	name
	1	test@example.com	Test User

Grant DynamoDB read only access to Lambda



Yay! Permissions added. This means my Lambda function should be able to read DynamoDB table items.

With the data tier ticked off, I'm officially ready to merge the three layers!

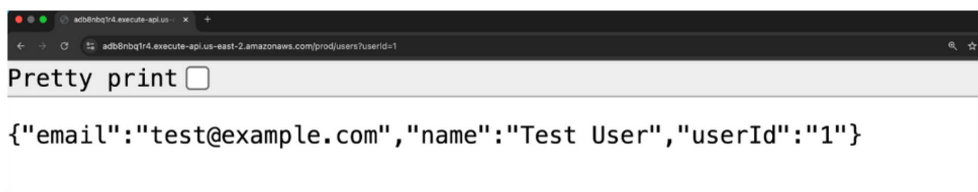
Integrate the Tiers

I've built all three tiers of my application!

Now, it's time to connect the presentation and logic tier together. Currently, there is no way for my API to catch requests that users make through my distributed site.

I will Update my script.js file with JavaScript code to make an API request.

To Verify API Functionality, paste the API invoke URL but appended with " /users?userId=1" to the end of the URL . I will run the new edited url in my browser. The results were some user data in JSON which proofed a logic + data tier connection.

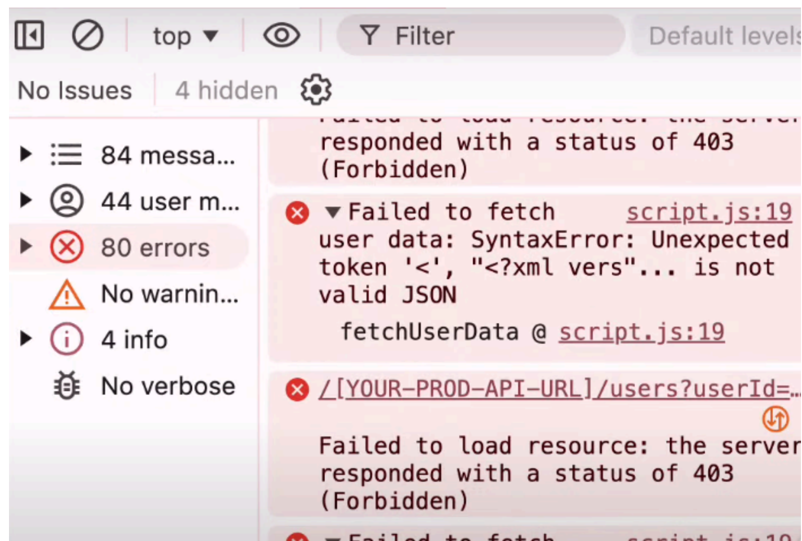


That's the logic and data tier's integration verified ✅

Now let's check my distributed website on CloudFront by typing in 1.

User Information

1 | Get User Data




```
async function fetchUserData() {
    const userId = document.getElementById('userId').value;
    if (!userId) {
        alert('Please enter a User ID');
        return;
    }

    try {
        const response = await fetch(`https://[YOUR-PROD-API-URL]/users?userId=${userId}`);
        const data = await response.json();
        const userDetails = document.getElementById('userDetails');

        if (response.ok) {
            userDetails.innerHTML = `<pre>${JSON.stringify(data, null, 2)}</pre>`;
        } else {
            userDetails.innerHTML = `<p>${data.message}</p>`;
        }
    } catch (error) {
        console.error('Failed to fetch user data:', error);
    }
}
```

Trouble shooting from browser’s developer tool, I got an error because there was an error within my script.js (one of the files i uploaded into s3). This file is referencing a prod stage API placeholder and not my API’s actual URL. To resove the error, I reuploaded script.js into s3 because s3 is still storing the last uploaded version with error.

Upload succeeded
For more information, see the **Files and folders** table.

Upload: status

Close

After you navigate away from this page, the following information is no longer available.

Summary

Destination	Succeeded	Failed
s3://nextwork-three-tier-name-123	1 file, 720.0 B (100.00%)	0 files, 0 B (0%)

Files and folders | Configuration

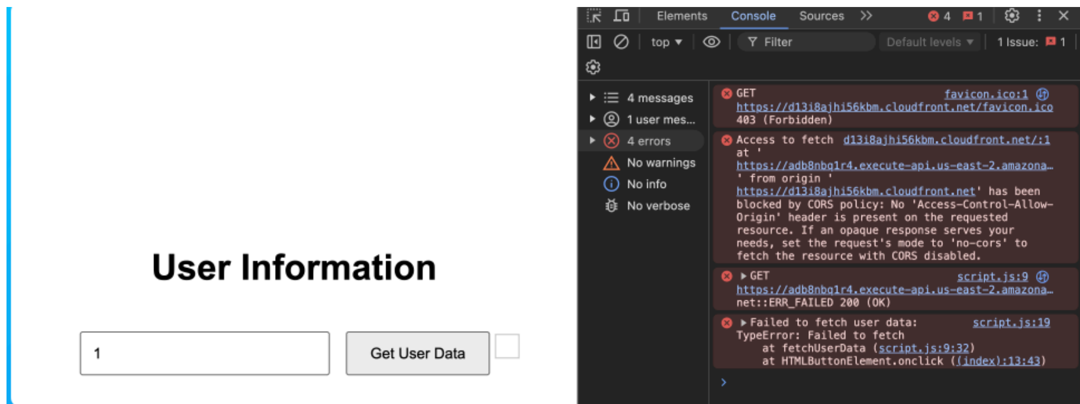
Files and folders (1 total, 720.0 B)

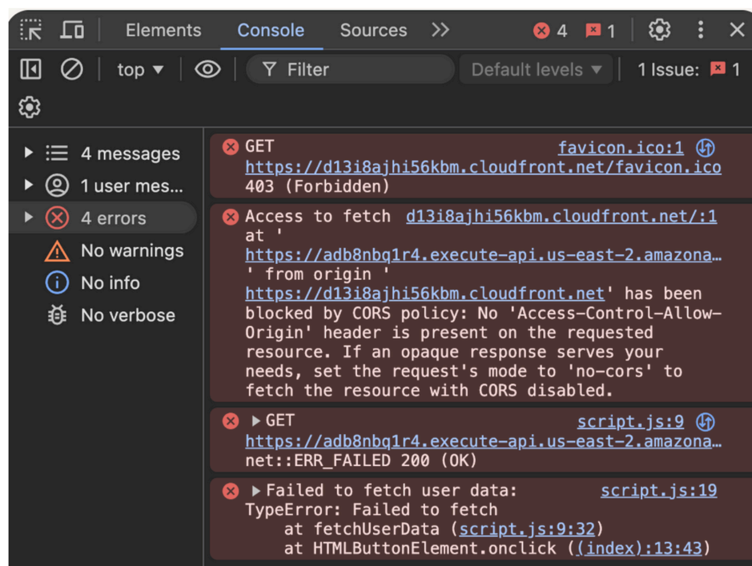
Find by name

Name	Folder	Type	Size	Status	Error
script.js	-	text/javascript	720.0 B	Succeeded	-

Vaidate a Fully Functioning Web App

I ran into another error when I tried to access my website through the CloudFront URL again.

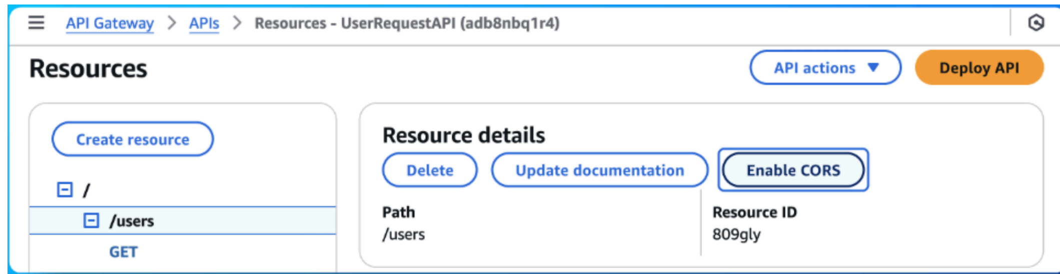




The **CORS (Cross-Origin Resource Sharing)** error I encountered happened because my API Gateway is not configured to allow requests from my CloudFront URL.

API Gateway is only allowing requests directly from its **Invoke URL**!

To resolve this, I'll need to enable CORS on my API Gateway so that it can accept requests from the domain where my frontend is hosted.



Using my CloudFront distribution domain name as the Access-Control-Allow-Origin value. This will allow requests from my CloudFront domain to my API. Afterwards, I will redeply my API.

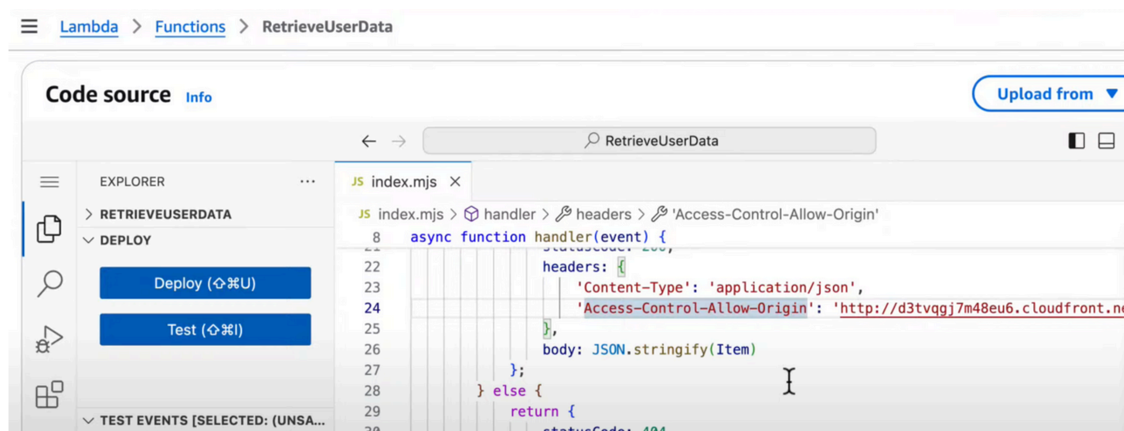
Access-Control-Allow-Origin

Enter an origin that can access the resource. Use a wildcard '*' to allow any origin to access the resource.

https://d13i8ajhi56kbm.cloudfront.net

In this step I will also add CORS Headers in my Lambda Function response. I updated Lambda function because it needs to be able to return CORS headers to show that it has the permission to invoke the API's invoke URL and return a response.

```
index.mjs
index.mjs > handler > headers
7   async function handler(event) {
30              headers: {
32                  'Access-Control-Allow-Origin': 'https://d13i8ajhi56kbm.cloudfront.net'
33              },
34              body: JSON.stringify({ message: "No user data found" })
35          };
36      }
37  } catch (err) {
38      console.error("Unable to retrieve data:", err);
39      return {
40          statusCode: 500,
41          headers: {
42              'Content-Type': 'application/json',
43              'Access-Control-Allow-Origin': 'https://d13i8ajhi56kbm.cloudfront.net'
```



The Final Test...

- Let's do one more refresh of my CloudFront domain name.
- WOAHH, I can now see the data fetched from DynamoDB displayed on my website!