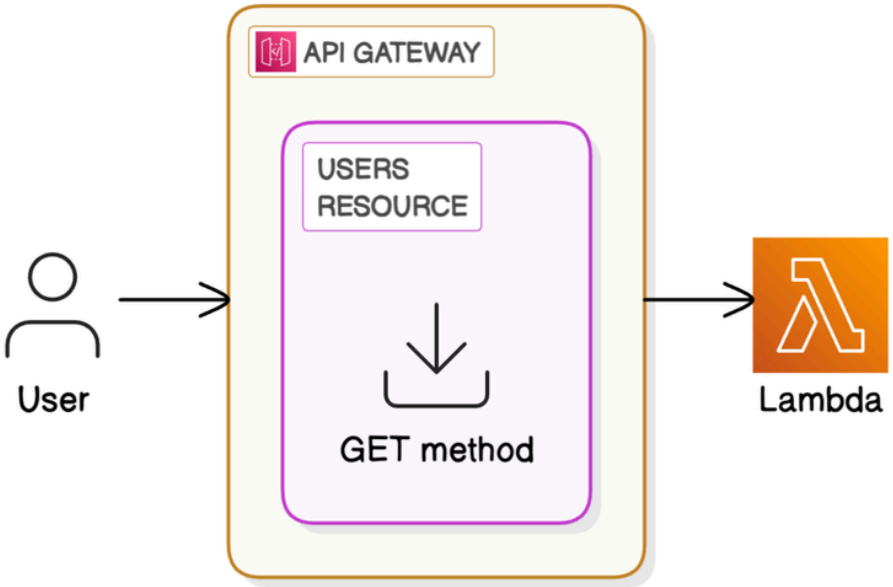
Kareshma
Rajaananthapadmanaban

AWS Three-tier series

Part 2

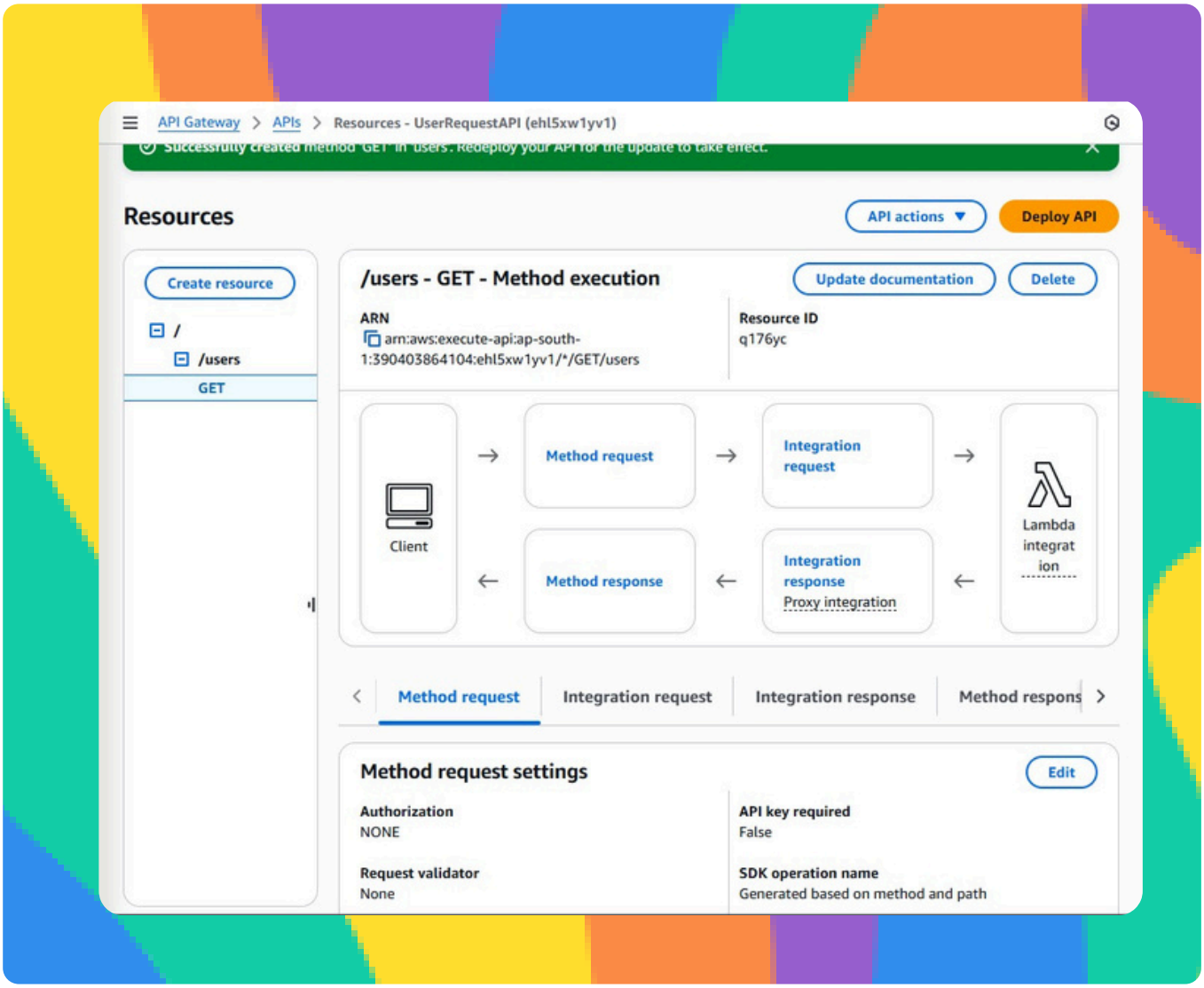
To build a serverless API using AWS Lambda and API Gateway, enabling scalable backend logic without managing servers.



nextwork.org

APIs with Lambda + API Gateway

Kareshma Rajaananthapadmanaban



Introducing Today's Project!

In this project, I will demonstrate how to build a **serverless API** using AWS Lambda and API Gateway. I'm doing this project to learn how **backend logic** is implemented in a cloud-native way, **without managing servers**, and to understand how APIs are securely configured and integrated in a three-tier architecture.

Tools and concepts

Services I used were **AWS Lambda** and **Amazon API Gateway**. Key concepts I learnt include Lambda functions, API creation and deployment, **RESTful** resource handling, HTTP methods like **GET**, and how Lambda and API Gateway work together to create serverless APIs. I also explored API stages, invoke URLs, and writing API documentation using **OpenAPI/Swagger standards**.

Project reflection

This project took me approximately 1 hour to complete. The most challenging part was understanding how **API Gateway integrates** with **Lambda** and configuring the GET method correctly. It was most rewarding to see my serverless API work and to write and publish proper **API documentation** an essential skill for real-world backend and cloud development.

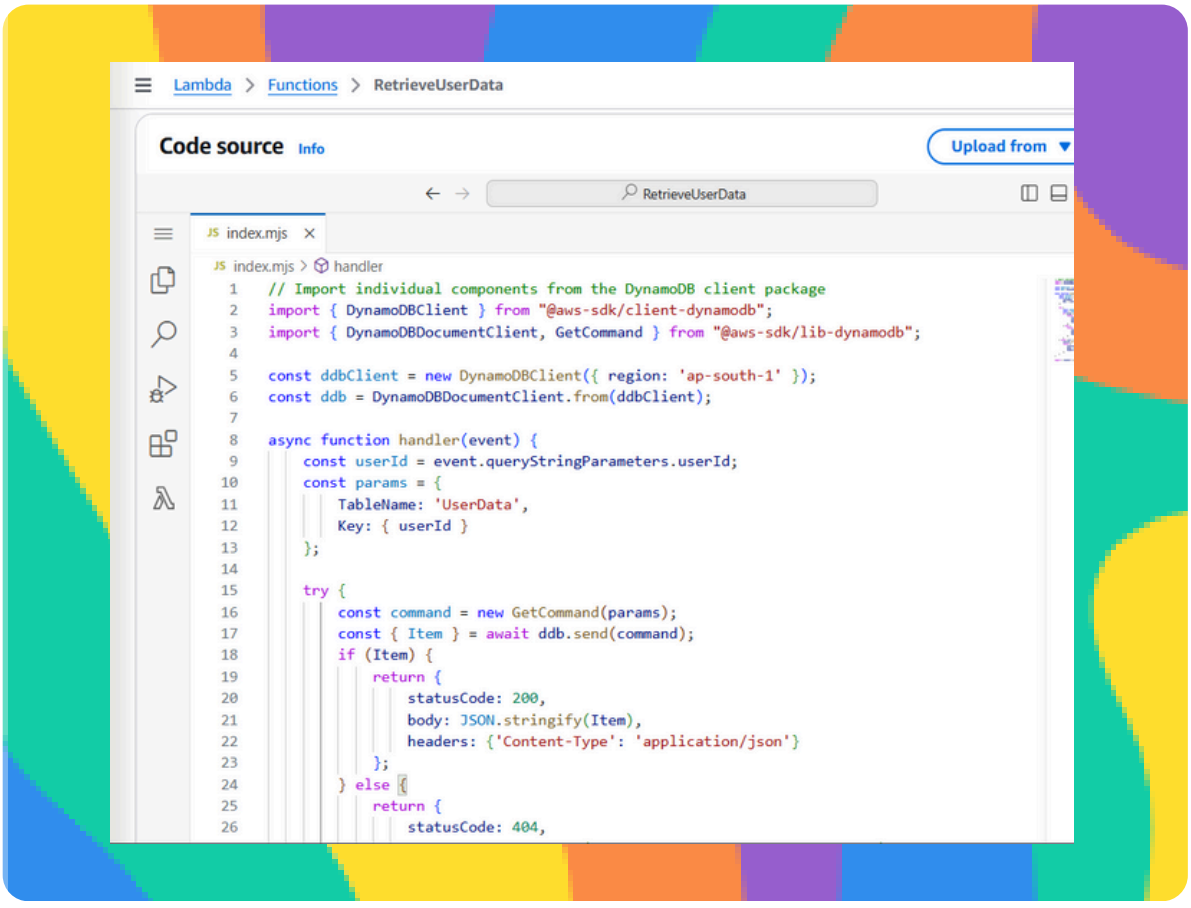
I did this project today to deepen my understanding of **serverless architecture** and learn how to **build** and **expose backend logic** using AWS Lambda and API Gateway. This project met my goals by helping me set up a working API, connect it to a Lambda function, and document it properly valuable skills for backend and cloud roles.

Lambda functions

In this step, I will create and configure a **new AWS Lambda function** because it serves as the backend logic of my application. This function will simulate **fetching user data** just like how a real-world app retrieves information from a database in response to user actions, forming the core of the **logic tier** in a three-tier architecture.

AWS Lambda is a **serverless compute service** that runs my code in response to events, without the need to manage servers. I'm using **Lambda** in this project to handle backend logic that responds to API requests specifically, to simulate fetching user data as part of the logic tier in a three-tier web architecture.

The code I added to my function will **connect** to a DynamoDB table named **UserData** and retrieve user information based on a **userID** passed through an **API query string.** It returns the users' data if found, or an appropriate error message otherwise simulating a typical backend lookup in a serverless setup.



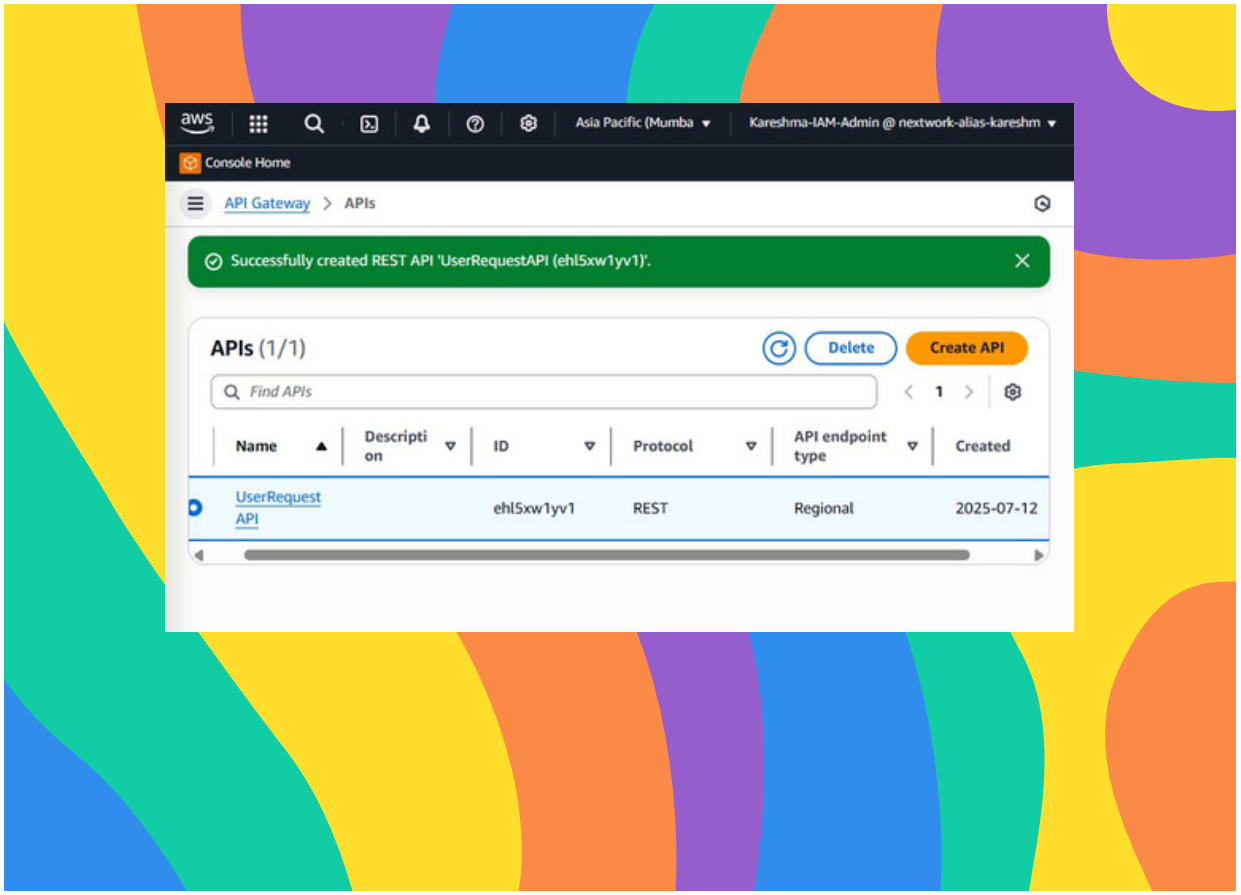
API Gateway

In this step, I will set up a **new API** using Amazon API Gateway because I need a secure and scalable way to expose my Lambda function to external users. This API will act as the **communication layer** between the user's request and the serverless function running in AWS Lambda.

APIs are interfaces that allow different software systems to communicate and exchange data. There are different types of APIs, like **REST** for standard **web operations**, WebSocket for real-time communication, and HTTP for simplified routing. My API is a REST API, ideal for linking **user requests** to my **Lambda function** in a scalable, language-agnostic way.

Amazon API Gateway is a fully managed service that helps me create and manage APIs at scale. I'm using **API Gateway** in this project to securely expose my Lambda function to users, handle incoming requests, and manage traffic, security, and routing between clients and backend logic.

When a user makes a request, API Gateway receives it and forwards it to the Lambda function. Lambda then processes the request such as retrieving data and sends the response back to API Gateway, which returns it to the user. This setup creates a **secure and serverless way** to run backend logic for web or mobile applications.

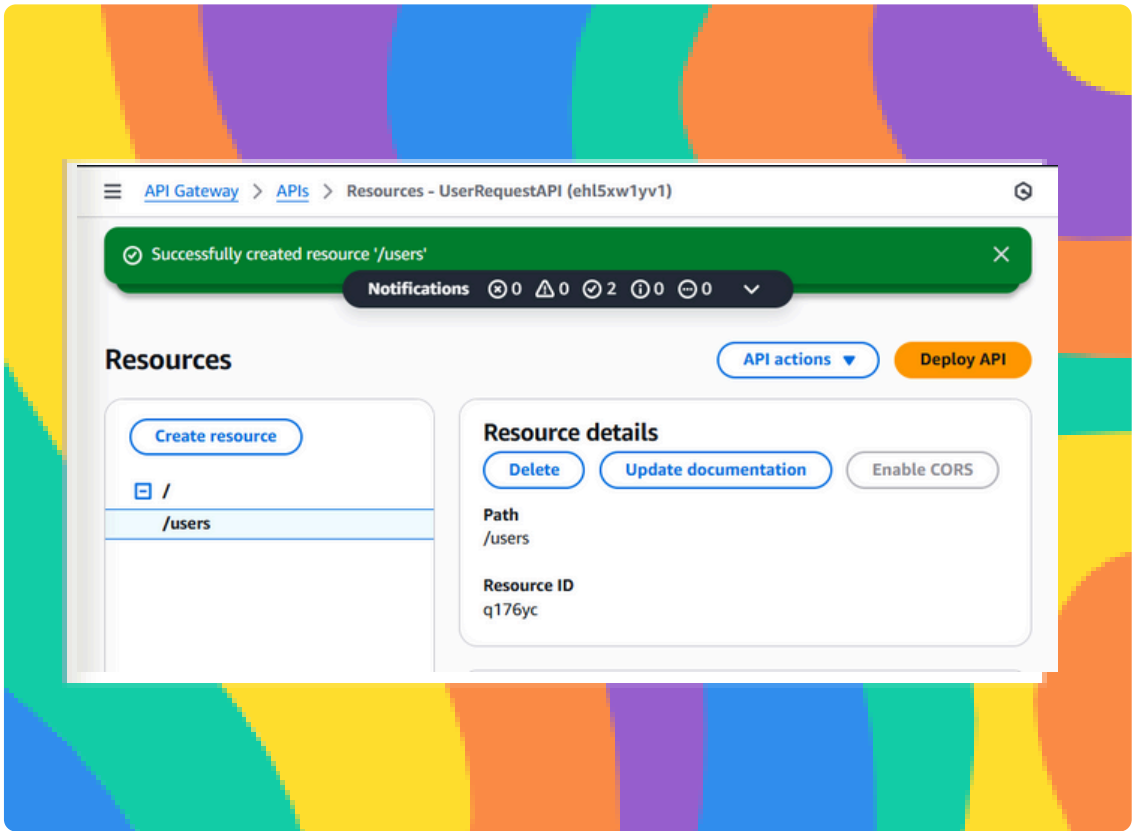


Newly created API.

API Resources and Methods

In this step, I will create a **new resource** in my API because resources define the different endpoints users can access. Each resource helps **organize specific types of requests** (like /users or /products) so my API remains structured, scalable, and easy to maintain.

An API is made up of resources, which represent **specific endpoints** that handle different functions of your application. Each resource maps to a part of the data or service, like /users or /messages, helping to organize and route incoming requests effectively.



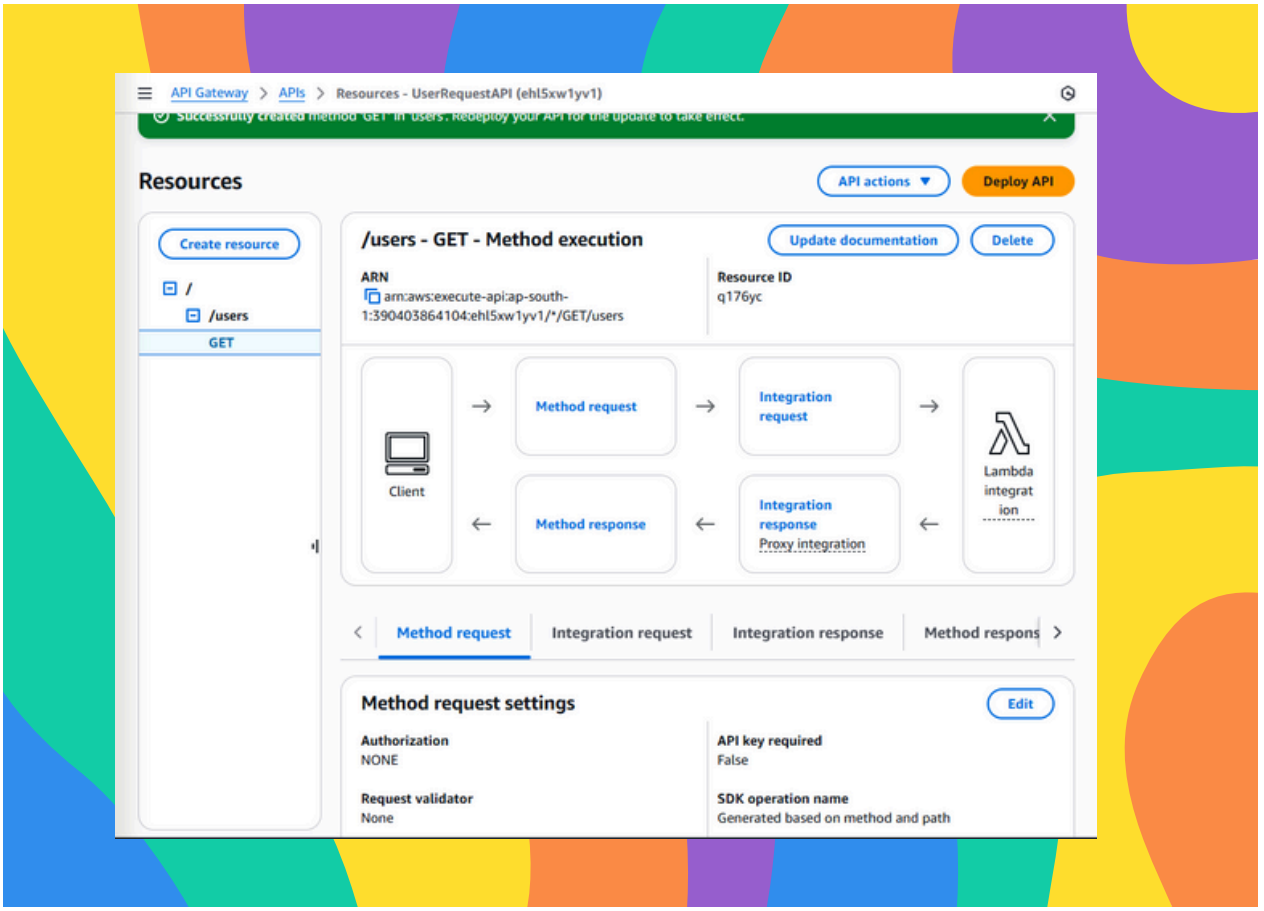
Newly created 'users' resource.

API Resources and Methods

In this step, I will create a **GET method** for the /users resource and connect it to my Lambda function because this allows users to retrieve data from the backend through the API. The GET method will **forward the request** to Lambda, which processes it and returns the appropriate user data.

Each resource consists of methods, which are actions that **define how clients** can **interact** with that **resource** using standard **HTTP commands**. For example, GET retrieves data, POST adds new data, PUT updates existing data, and DELETE removes data. These methods control how your API behaves and responds.

I created a GET method for the /users resource and connected it to my RetrieveUserData Lambda function. This allows API Gateway to forward user requests **directly** to the Lambda function, which then retrieves and **returns** user data from DynamoDB.



GET method configuration

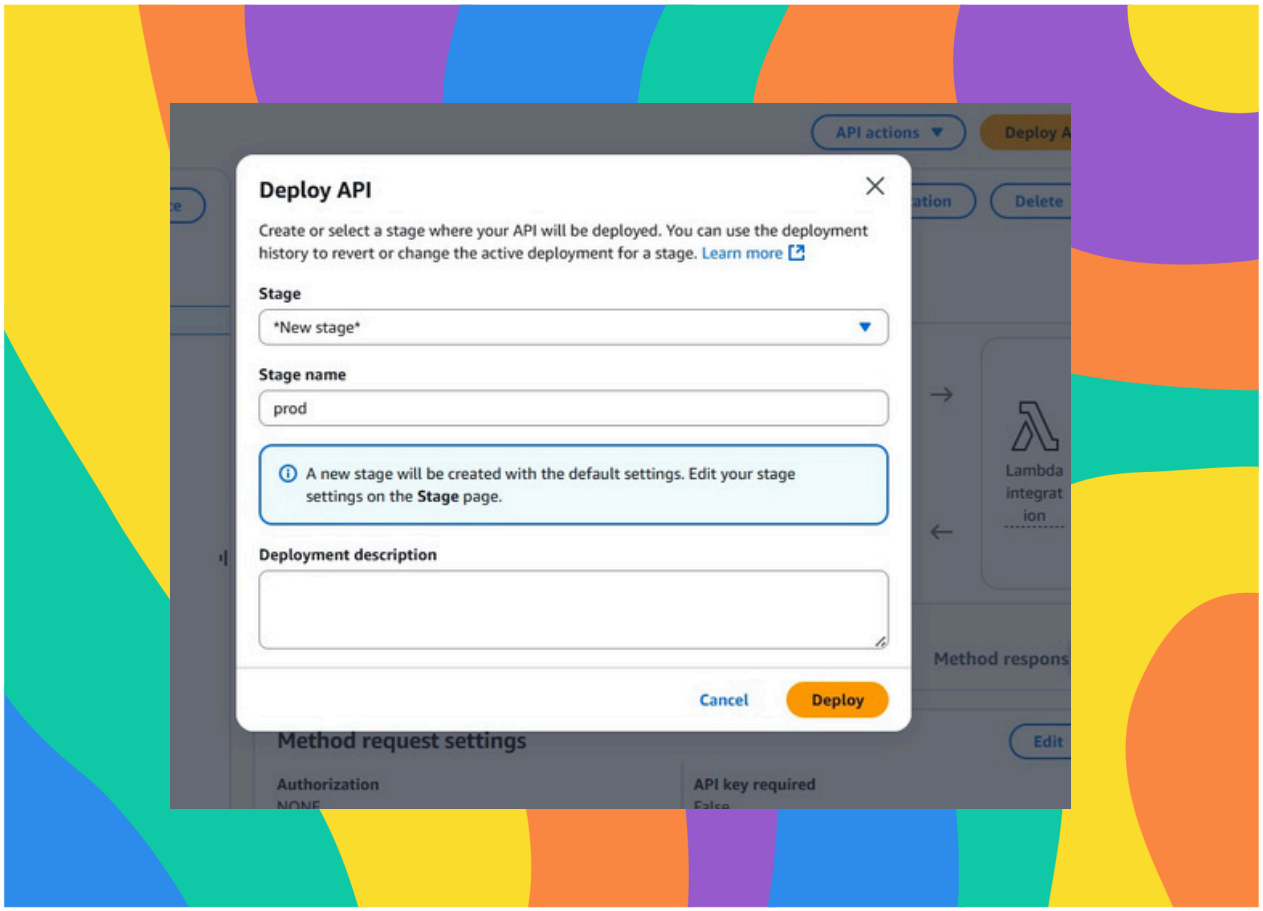
Deploy API

In this step, I will deploy my API to a **new stage** because deployment makes the API publicly accessible. This allows users to send real requests to the API endpoint and get responses from the connected Lambda function.

API Deployment

When you deploy an API, you deploy it to a **specific stage**. A stage is like a **versioned environment** (e.g., dev, test, or prod) that holds your deployed API configuration. I deployed to the prod stage, which is the live environment where real users can **access and interact** with the API.

To visit my API, I copied the **Invoke URL** from the prod stage page and opened it in a new browser tab. The API displayed an error because I haven't set up the DynamoDB table yet so there's no data for the Lambda function to retrieve.



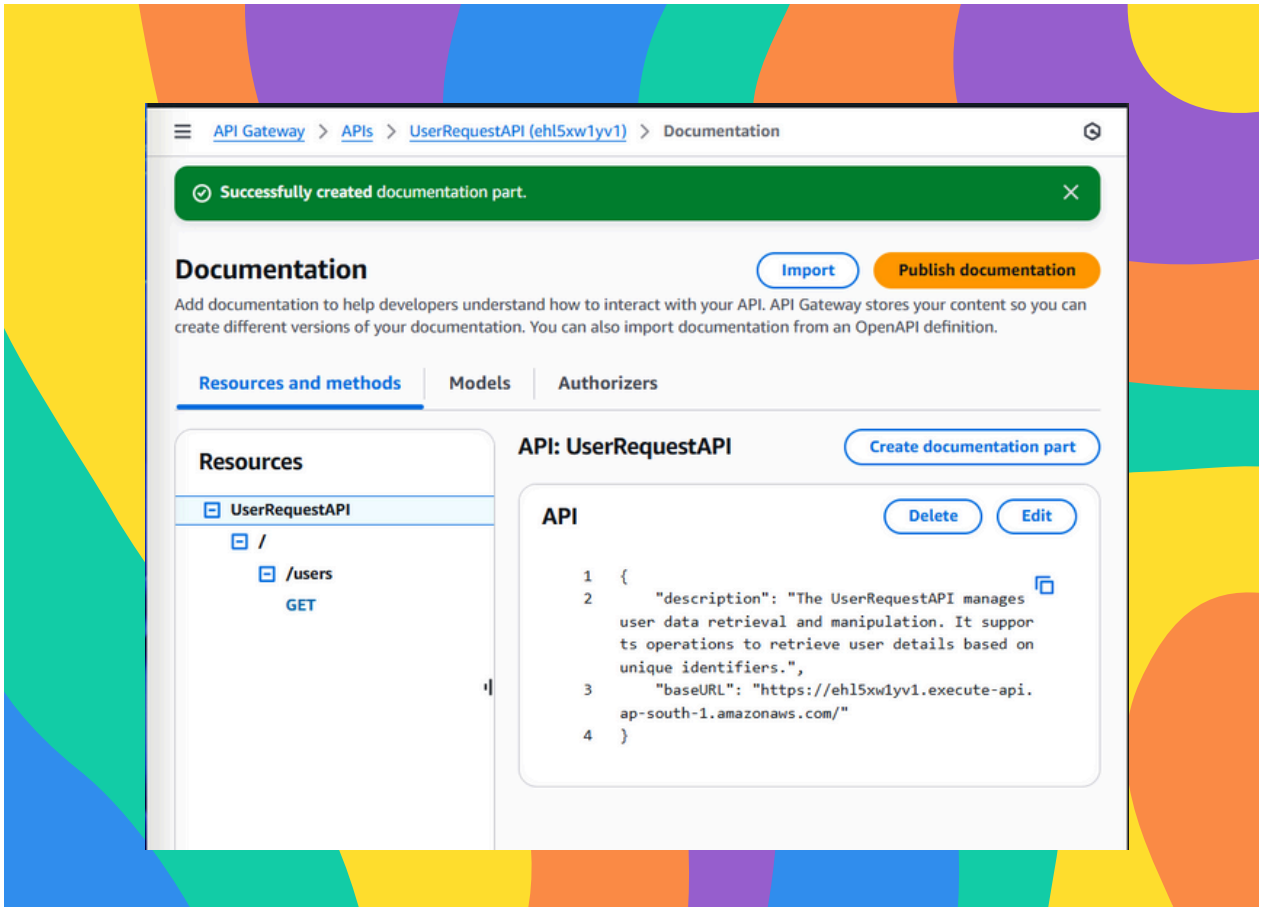
API Documentation

In this secret mission, I will write **API documentation** in JSON format using API Gateway because it helps developers understand how to use my API effectively. Documenting endpoints, methods, and parameters ensures clarity, improves collaboration, and makes it easier to maintain or extend the API in the future.

For my project's extension, I am writing API documentation because it helps other developers understand how to use my API, what endpoints and parameters to use, and what responses to expect. You can do this in API Gateways' Documentation section, where each method and resource can be clearly described in **JSON format**.

Once I prepared my documentation, I could publish it to the prod stage by selecting **Publish documentation**, choosing the correct stage, and entering a version number. You have to **publish** your **API** to a **specific stage** because each stage represents a version of the API, ensuring the right documentation matches the deployed API.

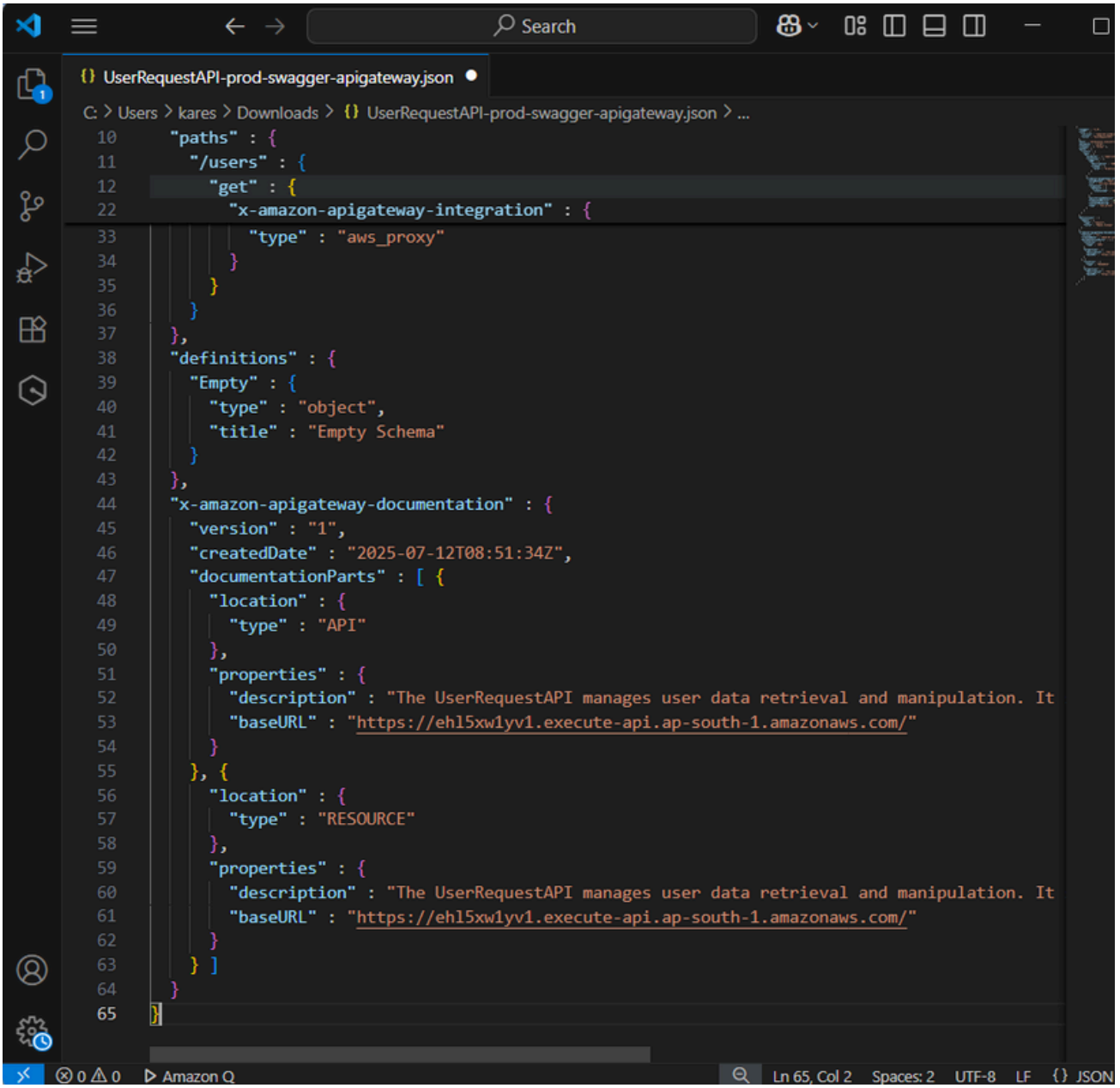
My published and downloaded documentation showed me a detailed JSON file containing **both** the high-level description I **manually** wrote and the **automatically generated** sections from API Gateway. It included my API's title, version, base URL, resource paths like /users, supported methods like GET, and a x-amazon-apigateway- documentation section capturing descriptions, parameters, and responses.



JSON documentation


Kareshma
Rajaananthapadmanaban

API Documentation



Delete the Resources

To avoid unnecessary AWS charges, it's important to clean up the resources created in this project once testing is complete.

Resources to Delete:

- API Gateway: UserRequestAPI
- Lambda Function: RetrieveUserData

1) Delete the API Gateway

1. Go to API Gateway in the AWS Console.
2. Select your API: UserRequestAPI.
3. Click Actions > Delete API.
4. Confirm deletion.

2) Delete the Lambda Function

1. Go to Lambda in the AWS Console.
2. Select your function: RetrieveUserData.
3. Click Actions > Delete.
4. Confirm deletion.

Deleting unused resources is a best practice for cost optimization and a tidy cloud environment.

Kareshma
Rajaananthapadmanaban

Follow up for more interesting projects !!!

Check out nextwork.org / my profile for more projects