

University of Westminster

# Project Report Online Travel Agency Software Solution

Cloud Computing Applications (7BUIS027)

Coursework 2

**Team of the Group 07**

B Gayan Kalinga w2027429 20231057 [gayan.20231057@iit.ac.lk](mailto:gayan.20231057@iit.ac.lk)

R A D V K Ranasinghe w2027419 20231017 [voshani.20231017@iit.ac.lk](mailto:voshani.20231017@iit.ac.lk)

W A R Sandeepa w2029463 20231044 [reshika.20231044@iit.ac.lk](mailto:reshika.20231044@iit.ac.lk)

K T R Greru w2027413 20230651 [tharaka.20230651@iit.ac.lk](mailto:tharaka.20230651@iit.ac.lk)

**Table of Contents**

[***Project Report  
Online Travel Agency Software Solution 1***](#_heading=h.3znysh7)

[**Introduction 1**](#_heading=h.tyjcwt)

[Solution Architecture Description 1](#_heading=h.tew0b6bgk4v7)

[System Components 1](#_heading=h.pukcrrcwnes9)

[Interactions and Interfaces 1](#_heading=h.njlwoqeh6h1l)

[Data Architecture 2](#_heading=h.q4pf38erj7x3)

[Security Considerations 3](#_heading=h.tthxcqpc0u6r)

[Reliability and Availability 3](#_heading=h.k7inljl9n6w7)

[Maintenance and Support 4](#_heading=h.gxub8lm1j9kj)

[Solution Architecture Diagram 5](#_heading=h.ku0epcu2r0vc)

[Deployment Architecture Diagram 7](#_heading=h.1hm0umgfaqd3)

[**Implementation Details 10**](#_heading=h.lnxbz9)

[AWS Infrastructure 10](#_heading=h.kc3467z0er9)

[Code Repositories and CI/CD 16](#_heading=h.5ywxx430yixv)

[Result Validations 18](#_heading=h.o9rqx5l4kdlz)

[Challenges Faced and how we solved 19](#_heading=h.6fm4w0v3gvc2)

# Introduction

We have selected the option two provided in the module coursework to design and implement a secure solution for an online travel agency to manage flight bookings.

This system will provide the main functionality such as flight search, ticket booking, user management, and managing reservations.

We have decided to use Amazon Web Services (AWS) as our cloud provider and Terraform as our Infrastructure as Code (IaC) tool. The common pattern we will be using is Terraform as a provisioning tool and Amazon ECS as an orchestration tool. Backend services are built using the programming language Java 17.0 and SpringBoot 3.2.

We have decided to use Github for version management and Github actions as our CI/CD tool to deploy the services to Amazon ECS and we can handle infrastructure deployment also in the same manner via Github actions.

For Terraform we have decided to use a remote backend to manage the terraform state so that multiple DevOps engineers can work on the same Terraform project and AWS instances.

The purpose of this report is to delve into the proposed solution architecture, elaborate on design decisions along with justifications, present the solution deployment diagram, and furnish implementation details

# Solution Architecture Description and Justification

Solution architecture was designed to provide a scalable and robust infrastructure for the application. The main goal is to have security and flexibility on the scaling given the high seasonal based demand. The following are the main components of the solution. We have summarized the decision and the justification to use components within this section.

### System Components

###### Route 53:

Route 53 offers various features, including domain registration, DNS routing, health checking, and traffic management, So this is good for hosting and managing domain-related functionalities.

###### API Gateway

Amazon API Gateway supports RESTful APIs, integrates with various backend services, with features such as security controls, traffic management, monitoring, and logging.

###### Web Application Firewall (WAF)

AWS Web Application Firewall (WAF) is a security service that protects web applications from common threats. It offers customizable rules, managed rule sets, rate limiting, IP reputation lists, and integration with AWS services.

###### Virtual Private Cloud (VPC)

VPC provides secure networking environment for the resources, allowing our resources to be deployed in a customized and isolated network infrastructure to host and manage the other provided services and functions.

###### Internet Gateway (IGW)

IGW enables communication between instances in a Virtual Private Cloud (VPC) and the internet. It serves as a gateway for outbound and inbound traffic.

###### NAT Gateways (NATGW)

NAT Gateway allows private subnet instances in a VPC to initiate outbound traffic to the internet while preventing inbound traffic from reaching them. This is mainly for software updates to be carried out without exposing their private IP addresses.

###### Application Load Balancer (ALB)

ALB distributes incoming application traffic across multiple instances, within one or more availability zones.

###### Private & public subnets

This separation is based on their internet access. Public subnets allow direct internet access, while private subnets route their traffic through a NAT gateway deployed in a public subnet.. This helps enhance security by isolating sensitive resources in private subnets from direct exposure to the public internet

###### Elastic Container Service (ECS) Cluster

ECS Cluster acts as the central management and orchestration unit for deploying and running Docker containers on a group of EC2 or Fargate. It simplifies the scaling of containerized applications. this is equal to Kubernetes.

###### Fargate

Fargate is a serverless compute engine for containers, allowing you to run Docker containers without managing the underlying infrastructure.

###### Elastic Container Registry (ECR)

ECR is a container registry service that allows developers to store, manage, and deploy Docker container images and streamlined the container deployment workflows.

###### Relational Database Service (RDS) MySQL

Amazon RDS for MySQL is a fully managed relational database service to simplify the deployment, management, and scaling of MySQL databases which offers high availability, security, and automation features to streamline database administration tasks.

###### S3

Amazon S3 is an object storage service that offers scalable, durable, and secure storage for a multiple static data types..

###### GitHub

###### GitHub is a web-based git version management platform. It provides a collaborative environment for our team to work.

###### GitHub Action

GitHub Actions is a CI/CD (Continuous Integration/Continuous Deployment) provided by GitHub. It allows our team to define workflows as YAML code.

###### CloudWatch

CloudWatch enables users to collect, analyze, and act on statistical data from the deployed other resources. This can help us maintain the reliability, availability, and performance and do future optimizations.

### Interactions and Interfaces

Route 53 is used as a Domain Name System (DNS) where we can use user-friendly domain names to call our backend APIs where Route 53 will convert it to the IP address. Also, Route 53 is scalable to support required traffic by our system and will provide regional based nearest routing as an additional function. so all the public traffic will first come to Route 53. However, in our solution, we have decided to deploy this for only one region. So this allows us to have future scalability. An alternative option is to use the AWS Simple DNS or CloudFront instead of the Route 53 however based on the functions and features offered by the Route 53 makes it suitable for our application.

We have used the AWS API gateway to provide additional security by attaching a WAF, while API GW will provide more advanced control for the backend application rather than the CloudFront, CloudFront does not have capabilities such as API Rate limiting, etc…

We have provisioned the Internet Gateway to provide inbound and outbound internet for resources under VPC and NAT gateways are deployed to access the internet for private subnets.

### Data Architecture

We have decided to deploy RDS MySQL database version 8.0.33 for the persistent data handling, SQL was preferred against NoSQL as we have to store relational data which is structured given the solution requirement.

We could have used an EC2 instance with databases setup as an alternative solution, However given the RDS is a managed service which could reduce the operation overhead for the team and provide more robust security and failover protection to the solution.

However, deploying RDS in a private subnet will reduce the control we have over the DB for maintenance tasks, Hence we have deployed the Bastion Host to manage our application requirements.

In order to solve the availability and data recovery, We are using a real time standby database in the second availability zone. Also, we are maintaining hourly snapshots from the secondary database, the reason behind having snapshots from the secondary database is to have the main/master database available without any slowness. In the future, if we want to move the databases to different regions we can use the snapshots to migrate. However during the implementation we found out that Amazon RDS Multi-AZ is a paid service. In that case for the demonstration purpose we have used only one RDS which is further explained in the implementation section.

We have used the Amazon S3 as object storage to store the user profile images and other static file requirements that may arise in the future. For this, we have disabled public access since these are confidential data. Generally, database read/write operations are slower. If files are stored in DB, the DB backup size will increase drastically which will result in slower replications and restorations. On the other hand, if we used file storage or EBS we would have certain scalability issues and global distribution-related issues.

### Security Considerations

We have isolated the network by utilizing Virtual Private Cloud (VPC); given the VPC is deployed we can use the subnets and security groups to manage the connection requirements in a more controllable way for our application.

We are using WAF with API GW mainly to address security related enhancements such as mitigating DDoS attacks, BOT control, web traffic filtering, automated threat detection, IP reputation list and rate limiting etc...

We have designed our subnets within the VPC to increase security by having Public and Private Subnets separately for critical services. so the communication is more standardized and controlled as per our requirements.

Security groups associated with instances control ingress and egress traffic. So configuring security groups, you can control which resources can communicate with each other and with external entities.

We have isolated the NGW to a different public subnet to reduce the privilege requirements and increase the security of the application.

### Reliability and Availability

We are using ECS as our container orchestration services, our decision was mainly skewed toward ECS rather than Kubernetes due to the tight integration of the AWS services, simplicity and cost factors. We had two choices to deploy the capacity providers. Either EC2 instances or Fargate serverless functions, Even though EC2 offers more control over the infrastructure we have decided to use fargate for its simplicity and lower cost to the hardware, Utilizing ECS over Kubernetes will reduce our ability to move to another cloud platform but given the current considerations we have made this decisions.

We are using AutoScaleGroup (ASG) to trigger deployment of more instances based on the memory usage, However we can use more parameters as we require to enable the high availability of services. Since we are deploying our services in two availability zones it would increase the availability of our infrastructure.

In order to have high availability and disaster recovery for the databases we have deployed RDS with Standby Mode and Snapshots from the standby databases. As we have previously talked about this under the data architecture, this will increase our ability to recover fast without any data loss.

### Maintenance and Support

Based on the solution design most of the infrastructure is maintained by the AWS and application related maintenance needs to be fully owned by the development team.

We have decided to deploy Bastion host for easy Database related maintenance and query execution. This is due to RDS being deployed on the private subnet so the Bastion host is connecting through the NAT gateway on the public subnet. However since the bastion host is deployed using EC2 we have to maintain the patches and OS related operations and this may increase the cost as well. so in future we can deploy Lambda function to auto shutdown the Bastion Host to use when required. Also we may need to deploy the same for the standby RDS at a later stage.

As an alternative to bastion host, we can use site to site VPN to increase the security, however since we do not have a static public endpoint we have used the above implementation for the demonstration purposes.

When it comes to maintenance and support it is paramount to deploy logging and tracing capabilities to identify production issues and incidents, and matrices will provide insights into performances and resource utilization for the deployed services.

Micro-services architecture is heavily dependent on these logging and tracing services. So we have decided to use the Amazon Cloud Watch for these purposes. Cloud Watch will initially collect and track metrics, collect and monitor log files, and moving forward we can set alarms based on certain event triggers. Further down the road, we can deploy dashboards for the Operation team to monitor the overall health.

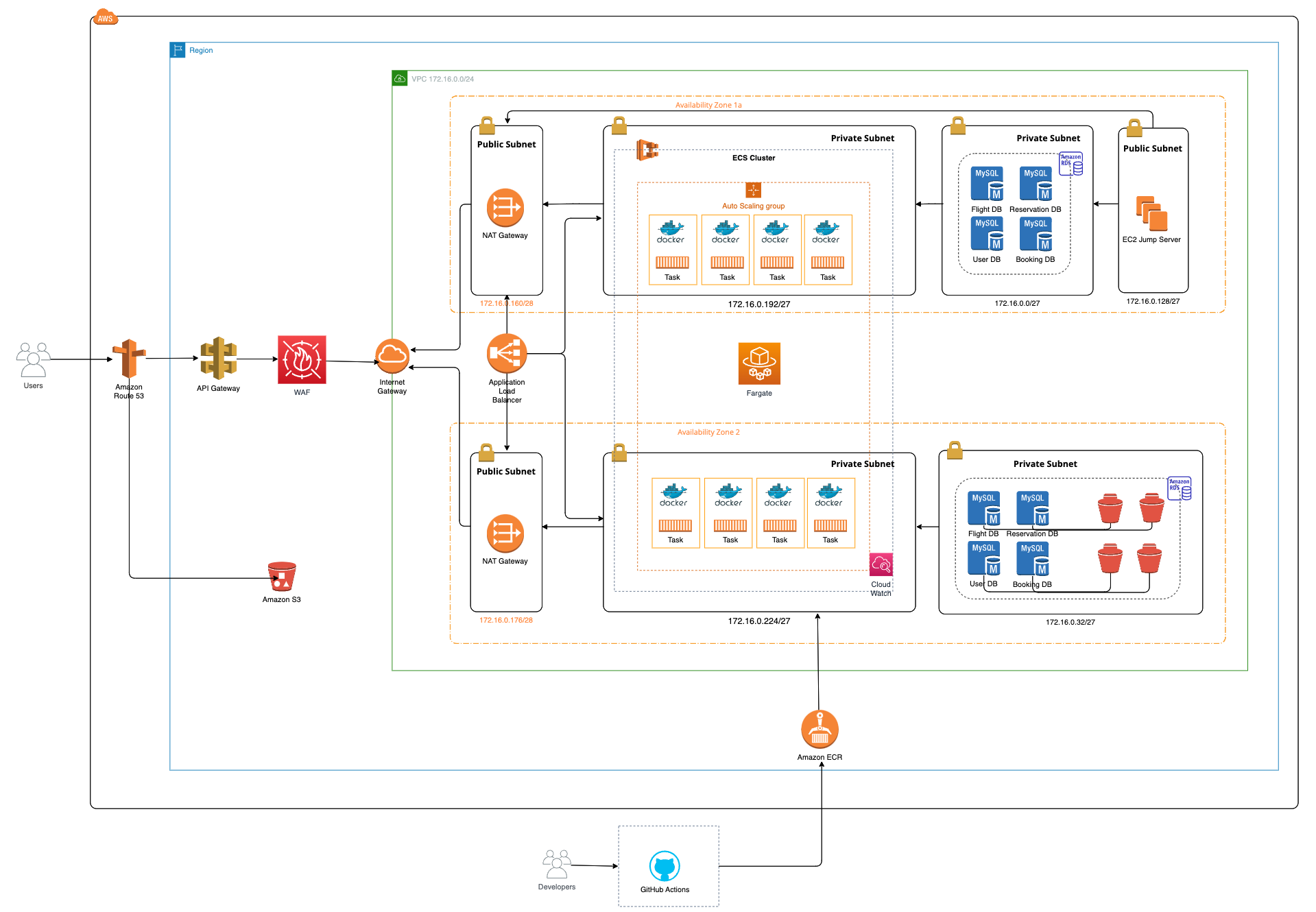
We could have used open-source alternatives such as Prometheus and Grafana but we will have to set up these manually and maintain them ourselves which will add additional operation overhead also Cloud Watch is tightly integrated with the AWS ecosystem is another deciding factor.

Our implementation supports all 3 Feature deployment methods as we have a Load Balancer in place we can use the Blue Green method, Rolling Deployment, and Canary Deployment methods.

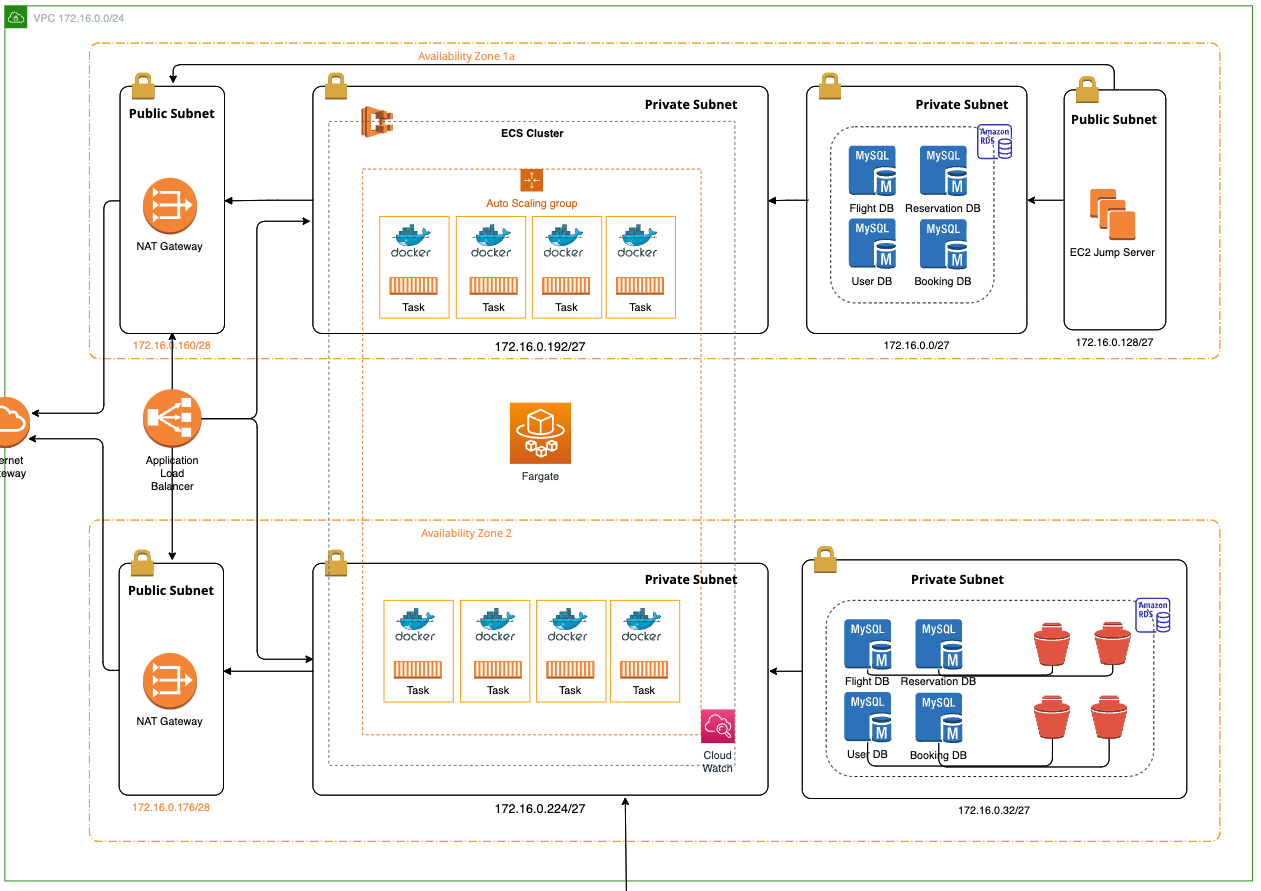
# Solution Architecture Diagram

SVG file attached in the zip folder to view the diagram

Complete View



VPC View



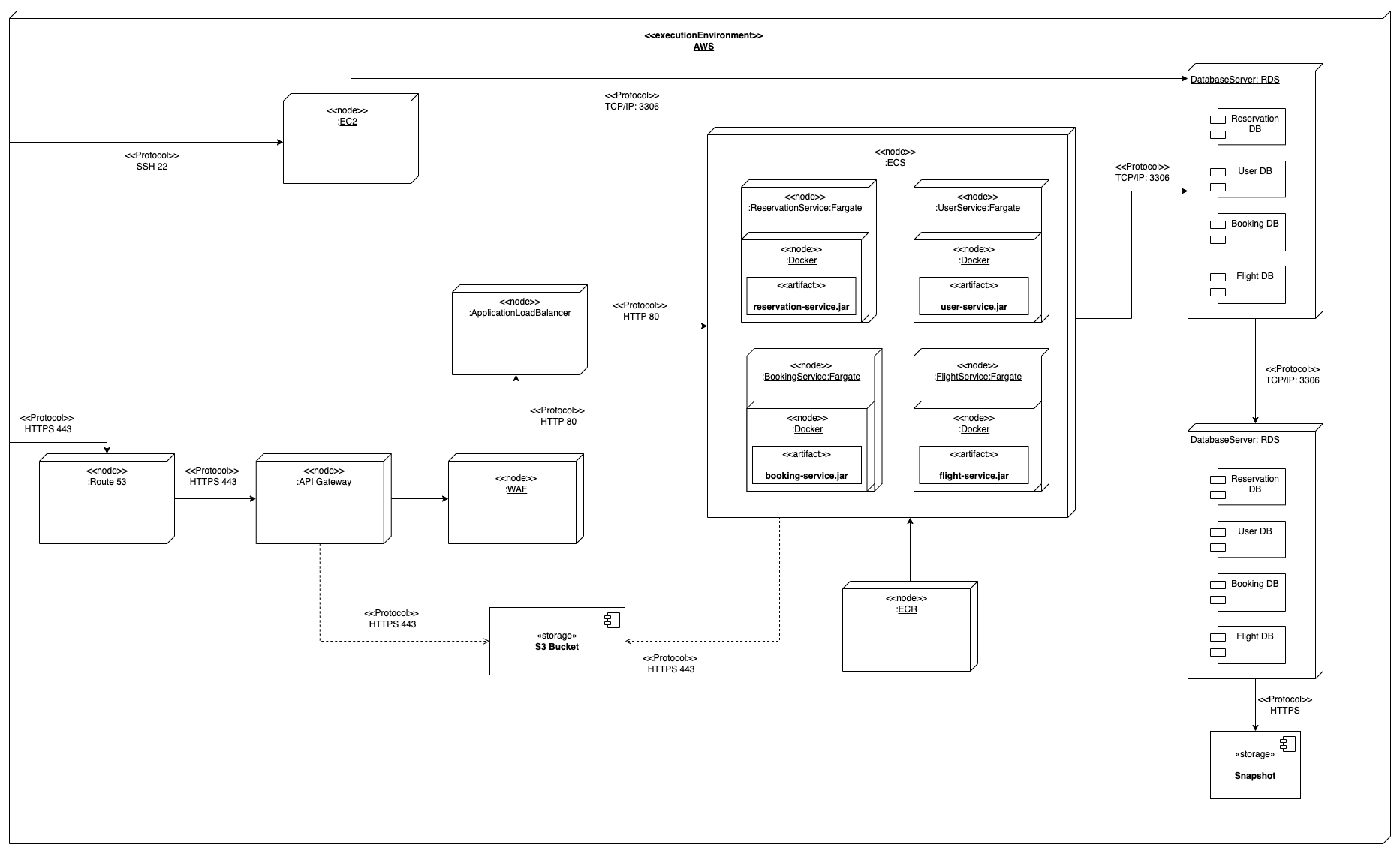
Single Availability Zone

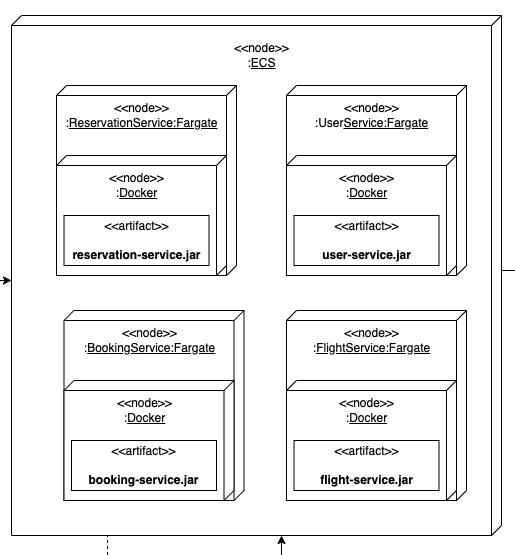
# 

# 

# Deployment Architecture Diagram

SVG file is attached in the zip folder to view the diagram





# Implementation Details

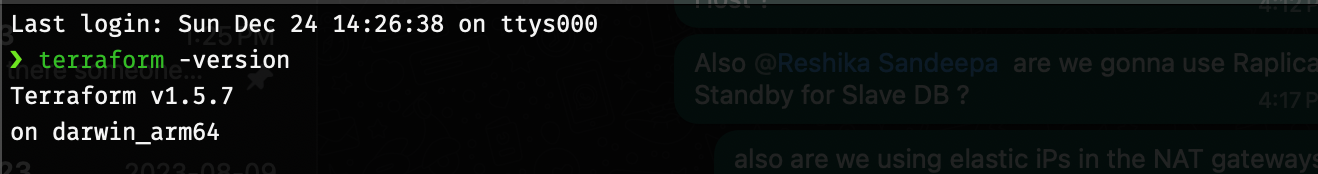
For the implementation we have used Java 17 and SpringBoot 3.0 as the backend development language and framework respectively. Four services are developed to have the core features of the solution. We have chosen Terraform v1.5.7 as our Infrastructure as Code tool (IaC).

We developed the terraform code in a modular way with each module representing the main resources that will be created in the AWS. Variables are separately declared to increase the code reusability and reduce the errors. In order to manage the multi user development for IaC we have opted for a remote backend to manage the terraform state. Once the terraform code is updated and changes are pushed to the git repository, GitHub action will be initiated and infrastructure will be deployed in the AWS.

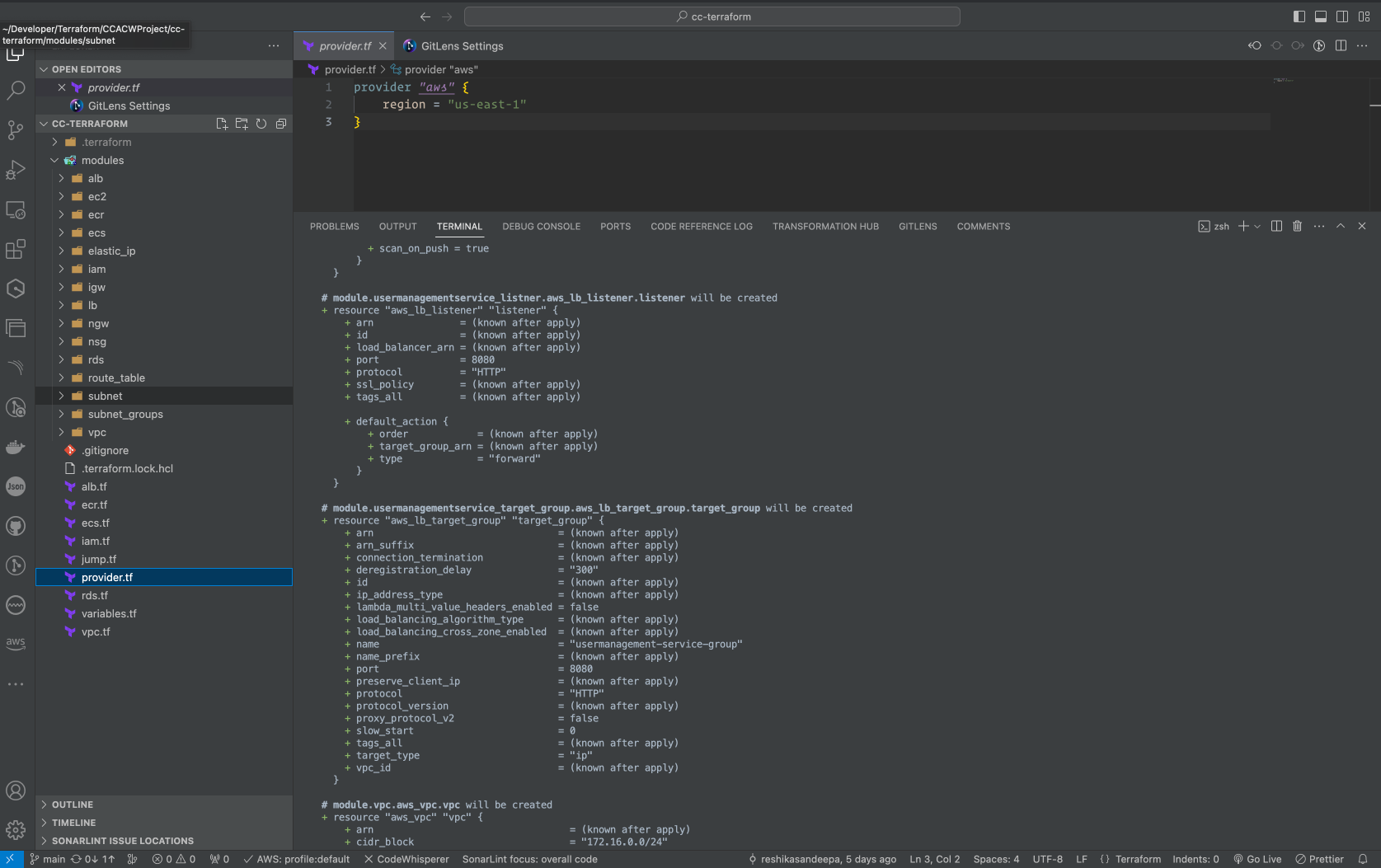
## AWS Infrastructure

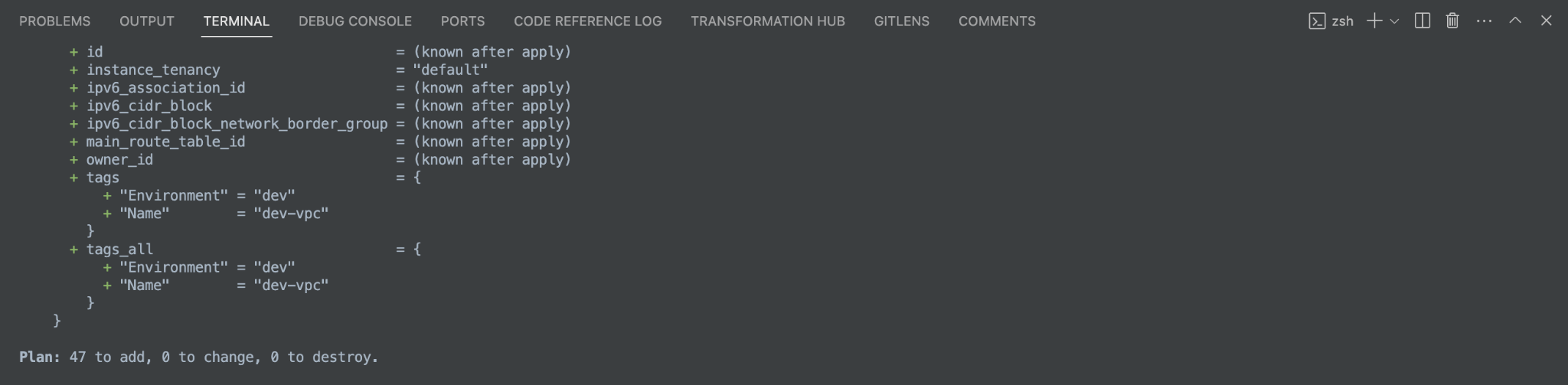
AWS IAM User details to check and verify the already deployed solution.

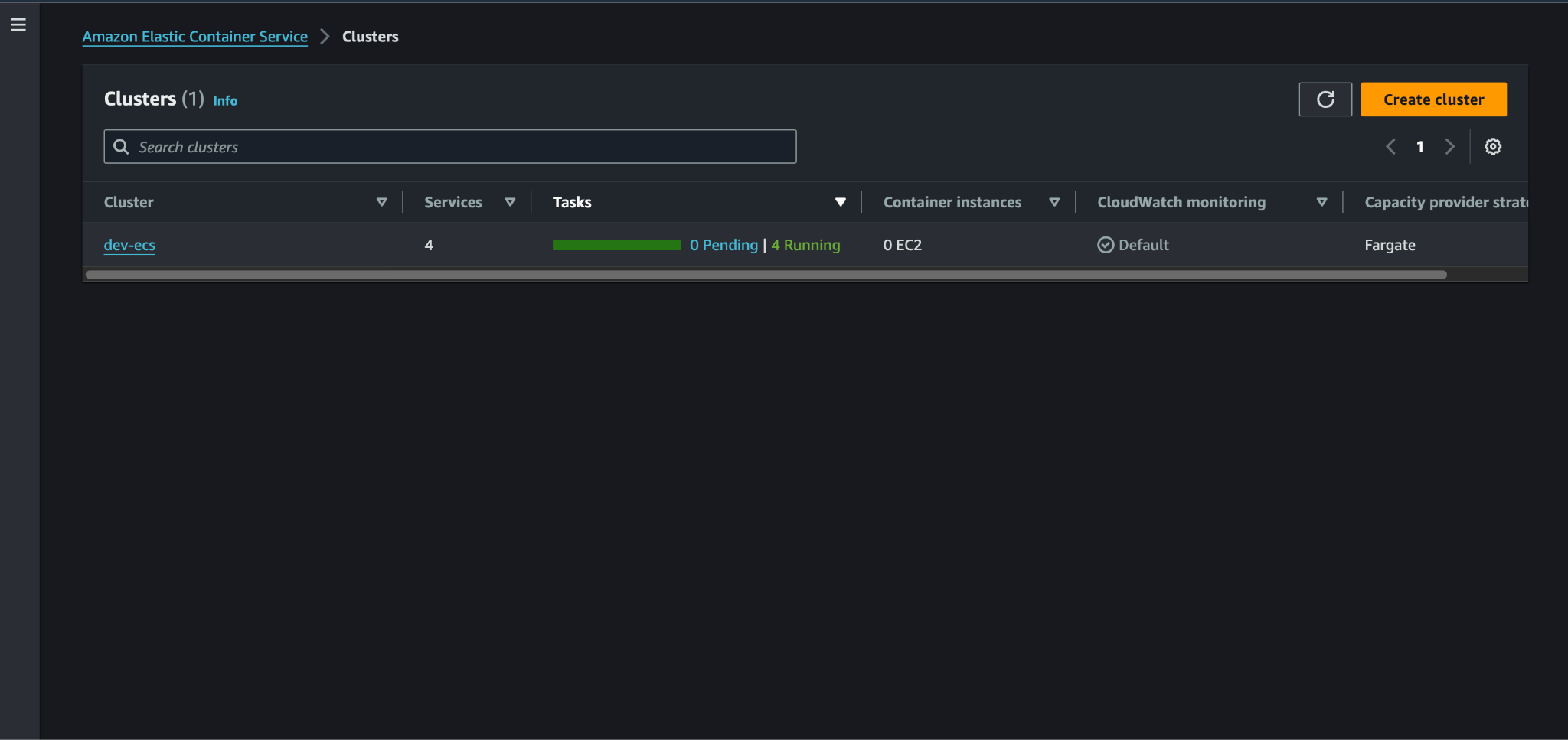
Tools and Versions Used.

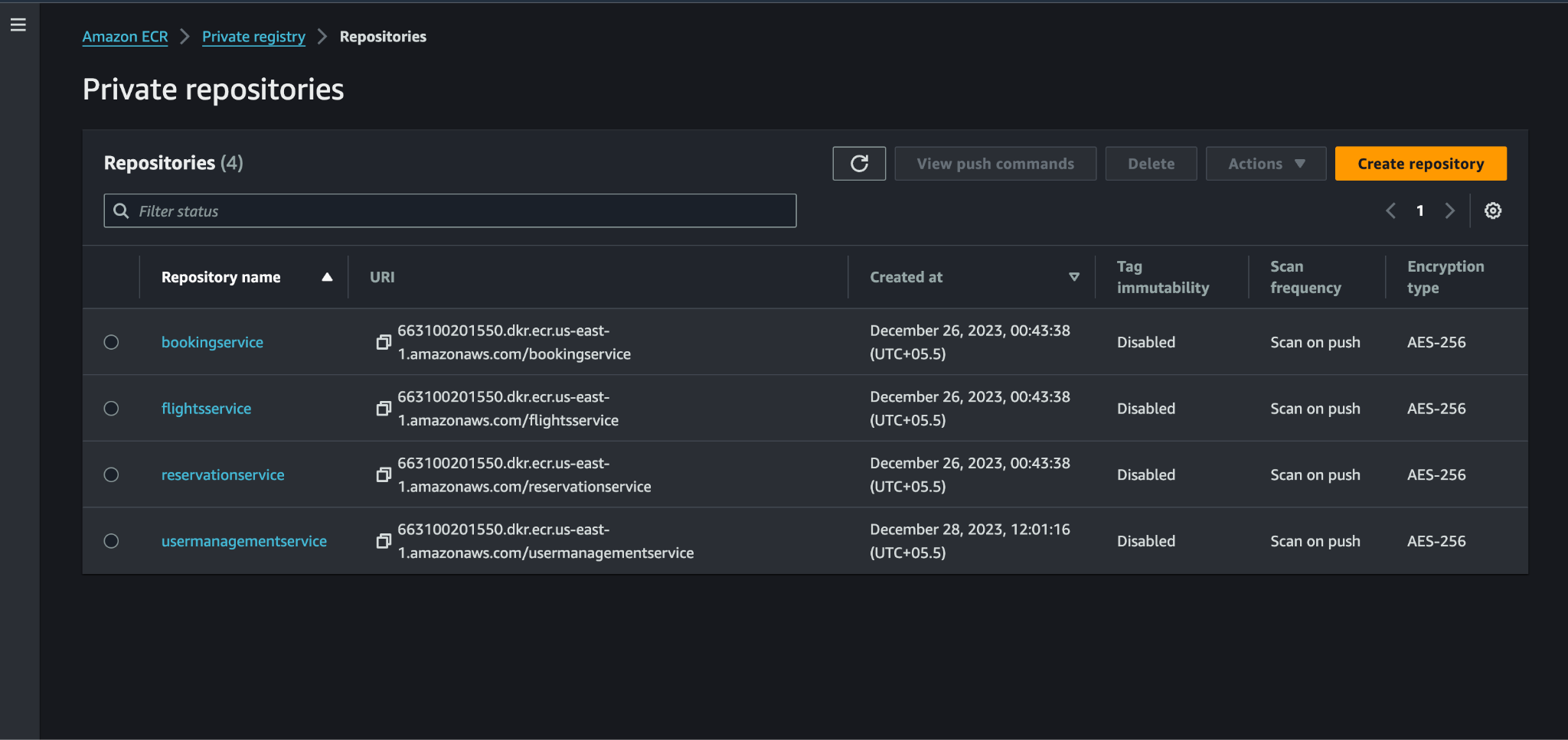


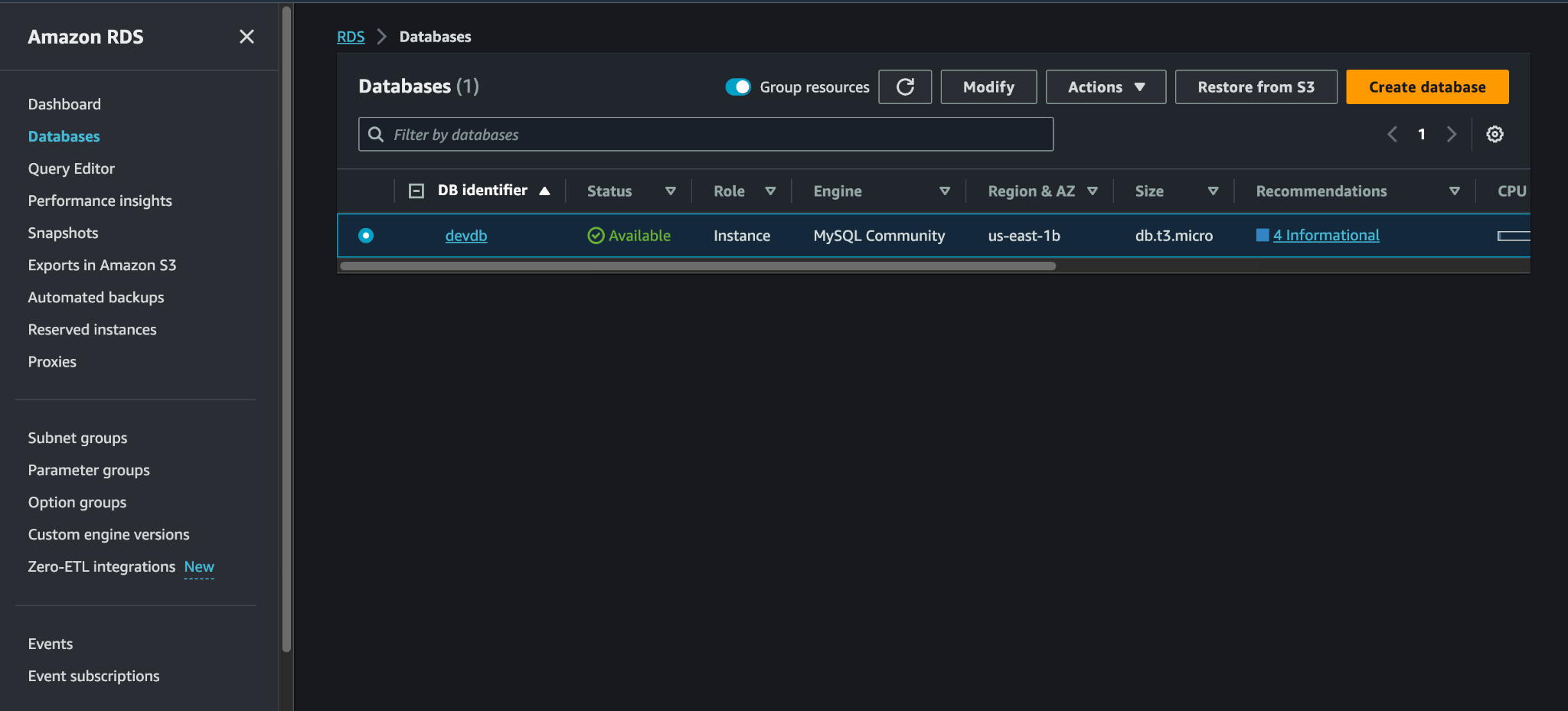


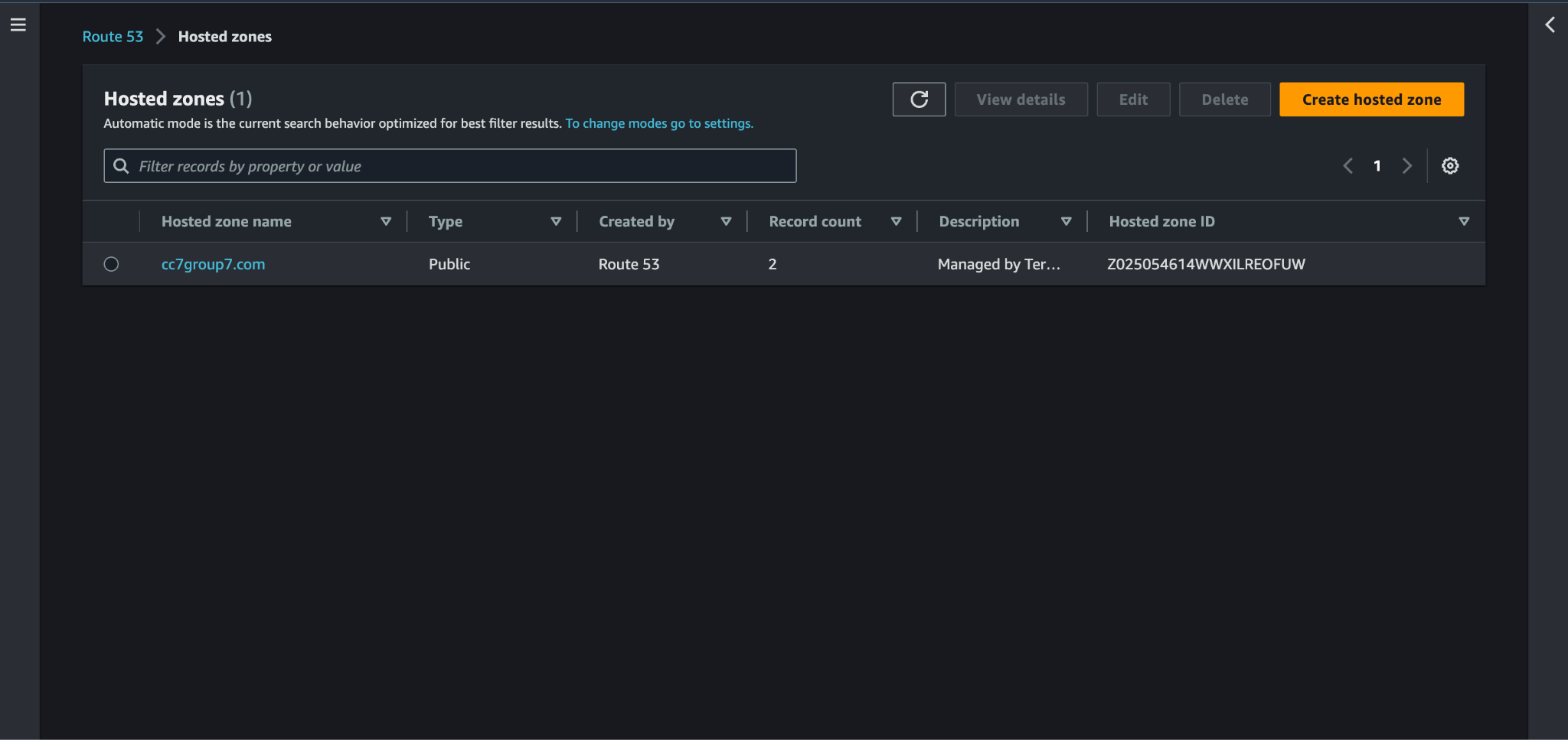


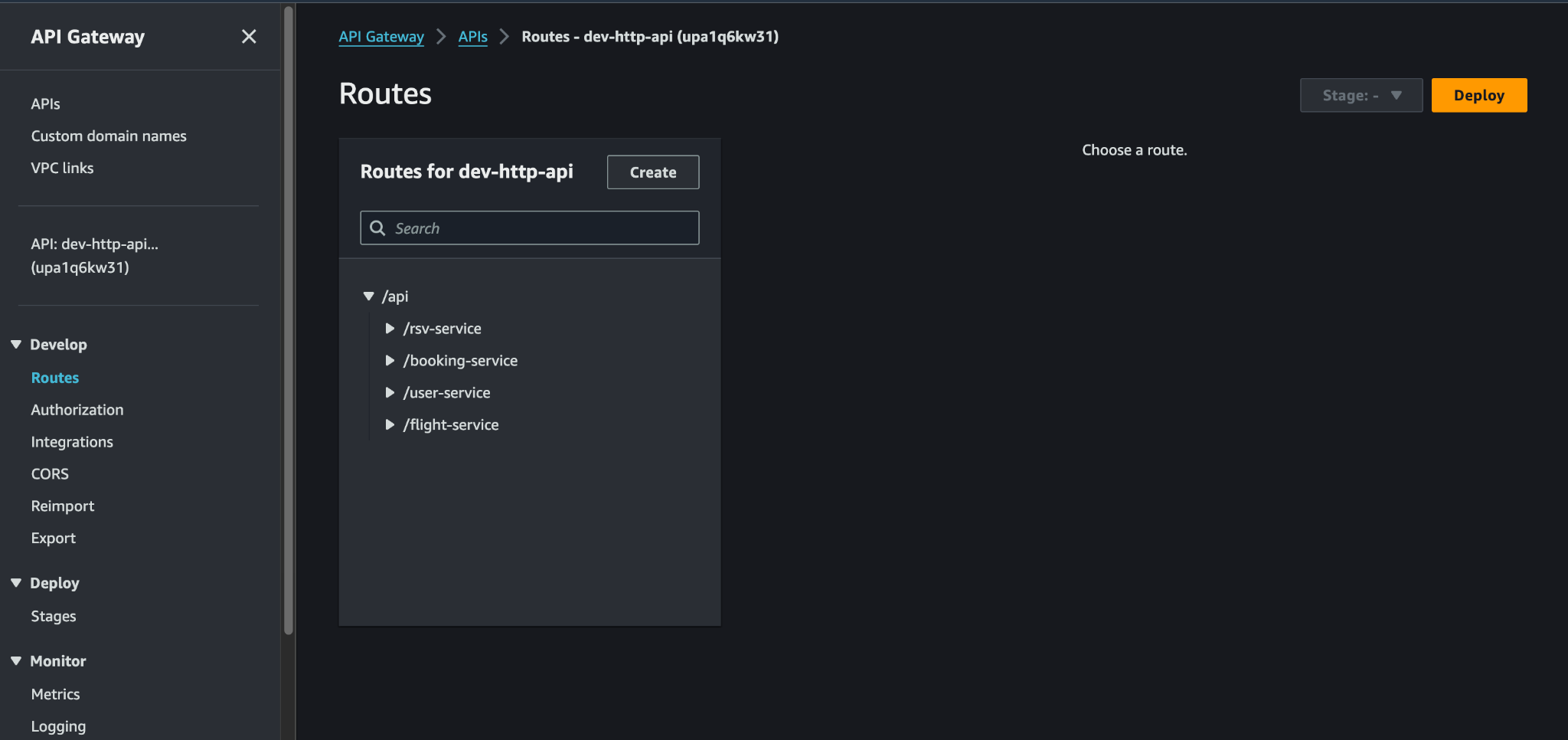




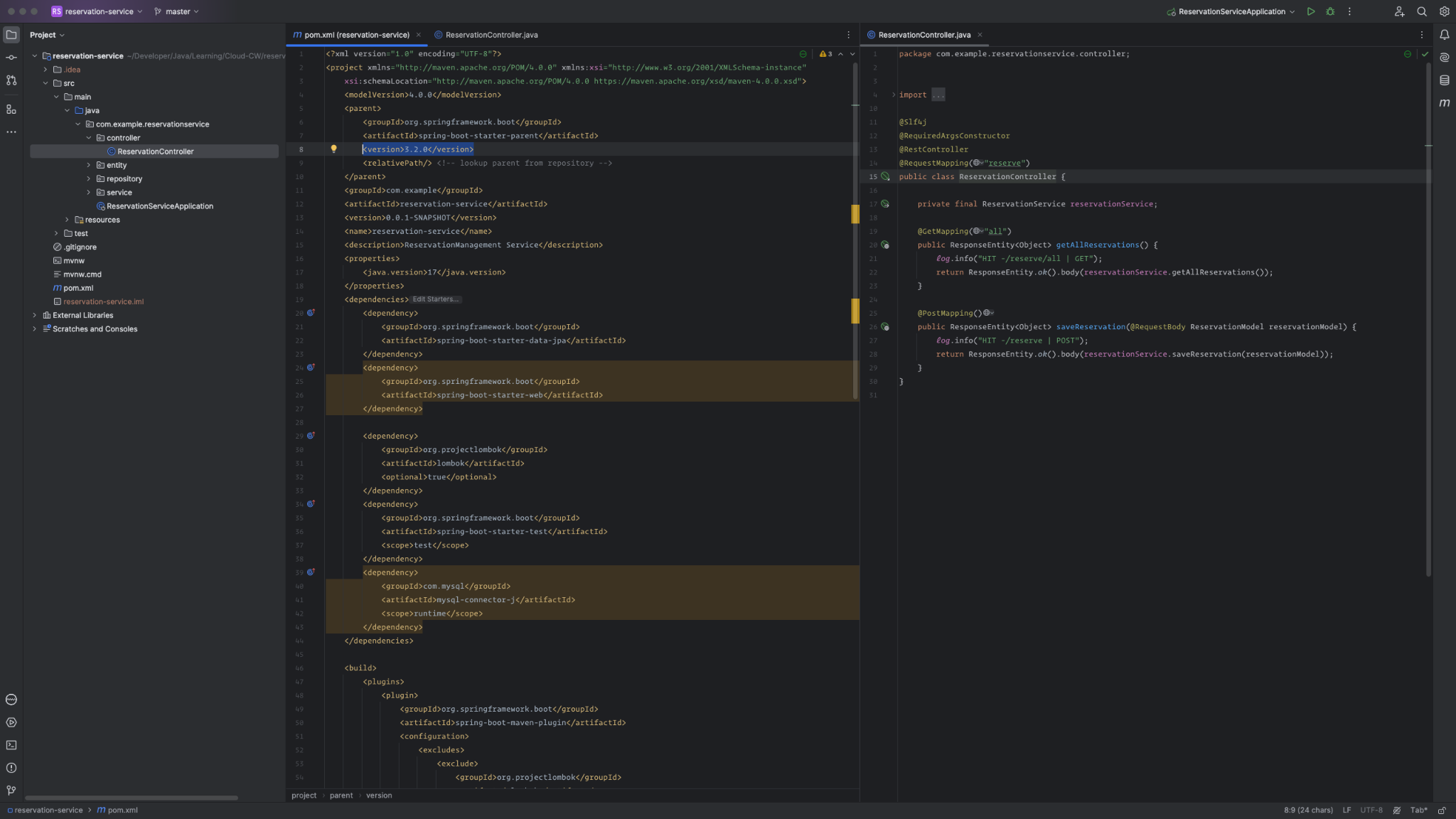


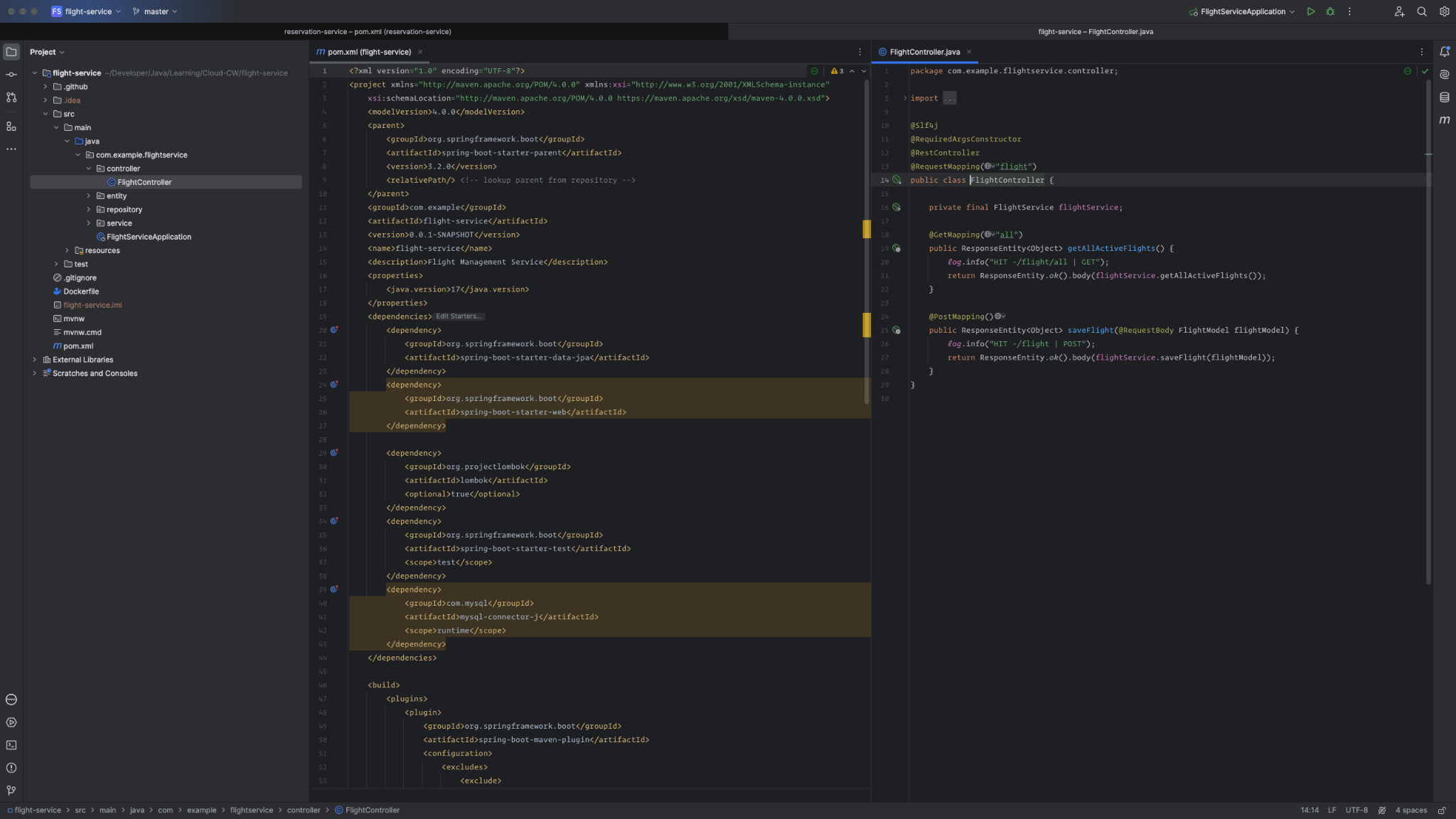


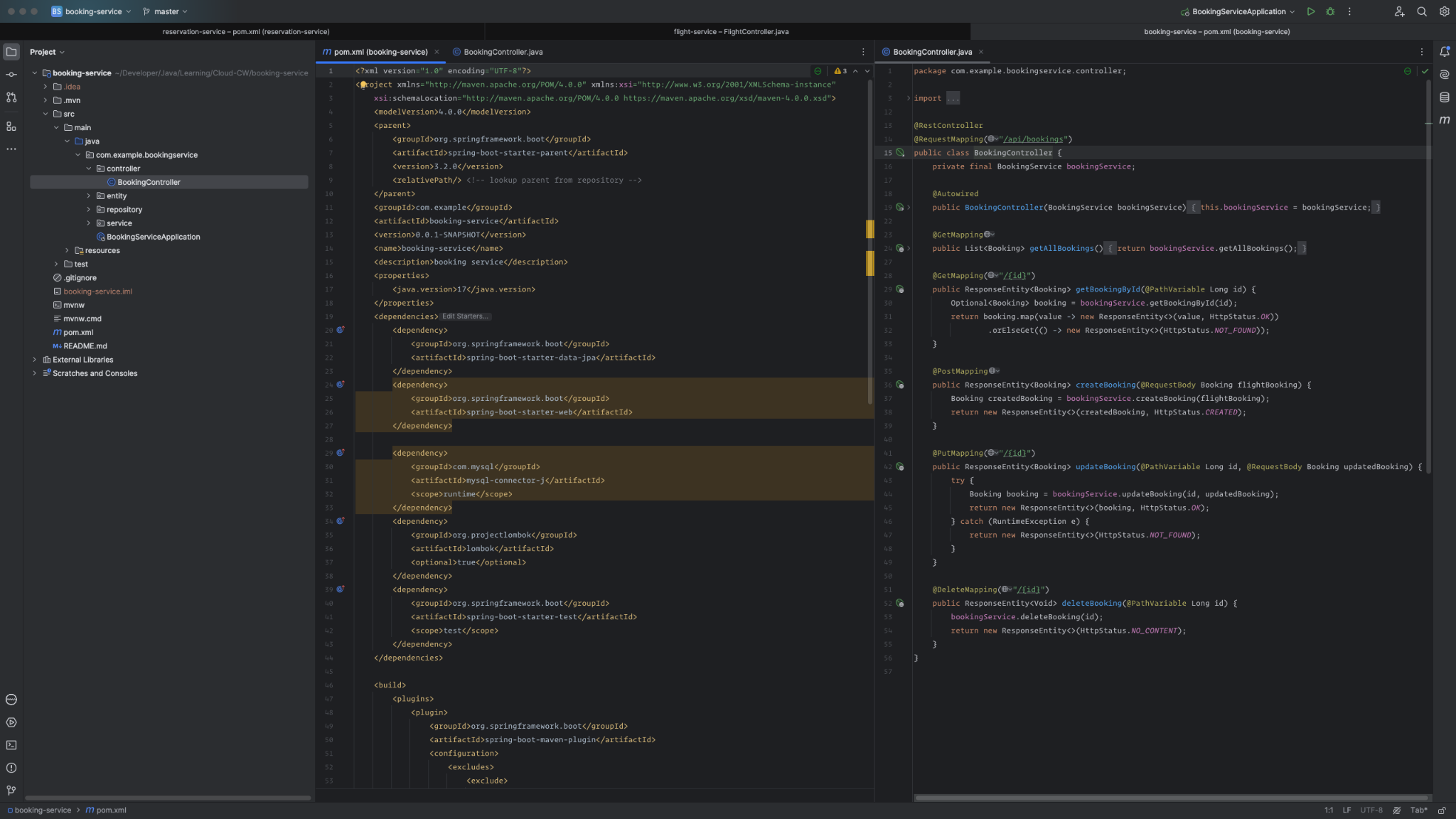


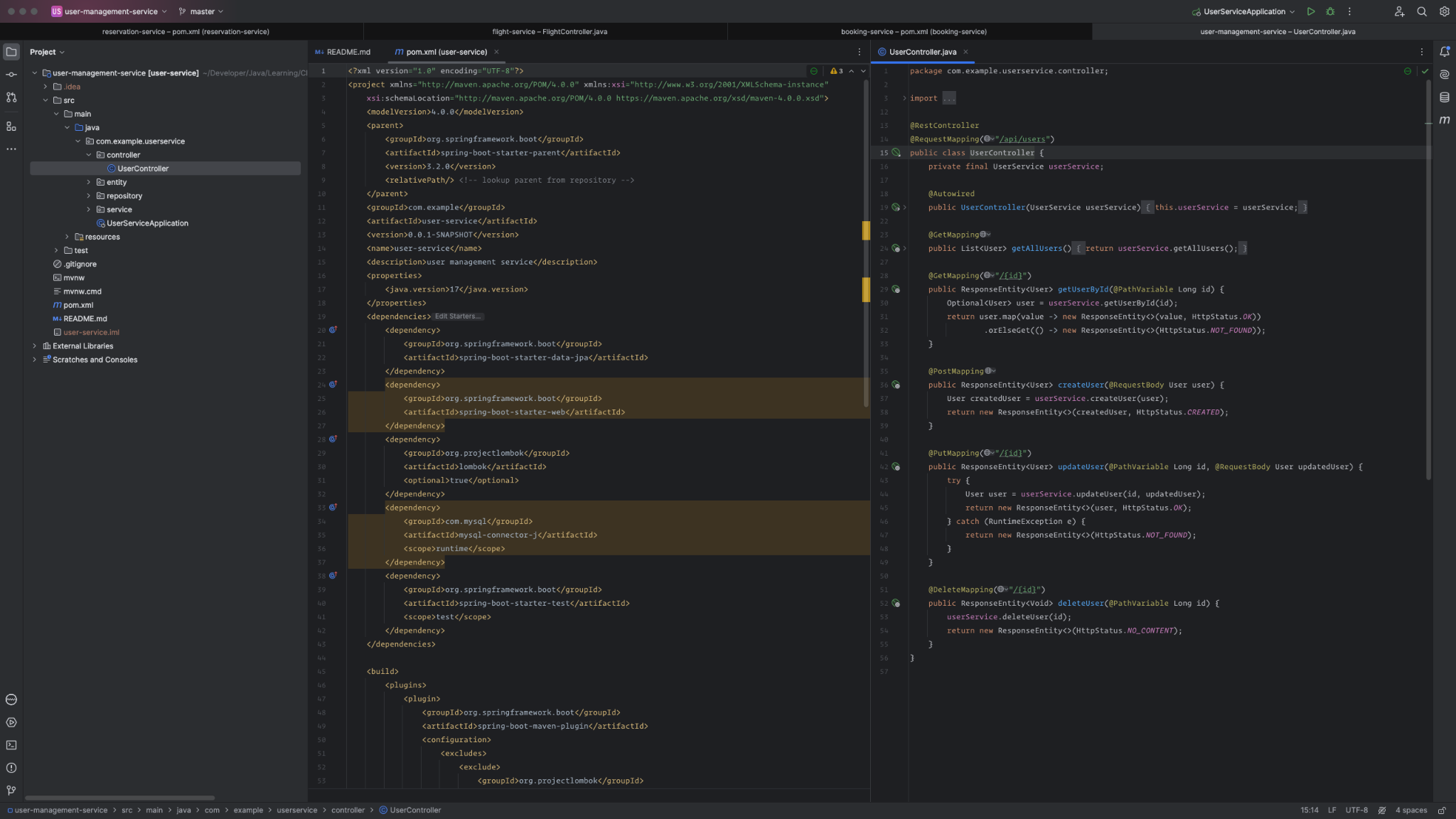








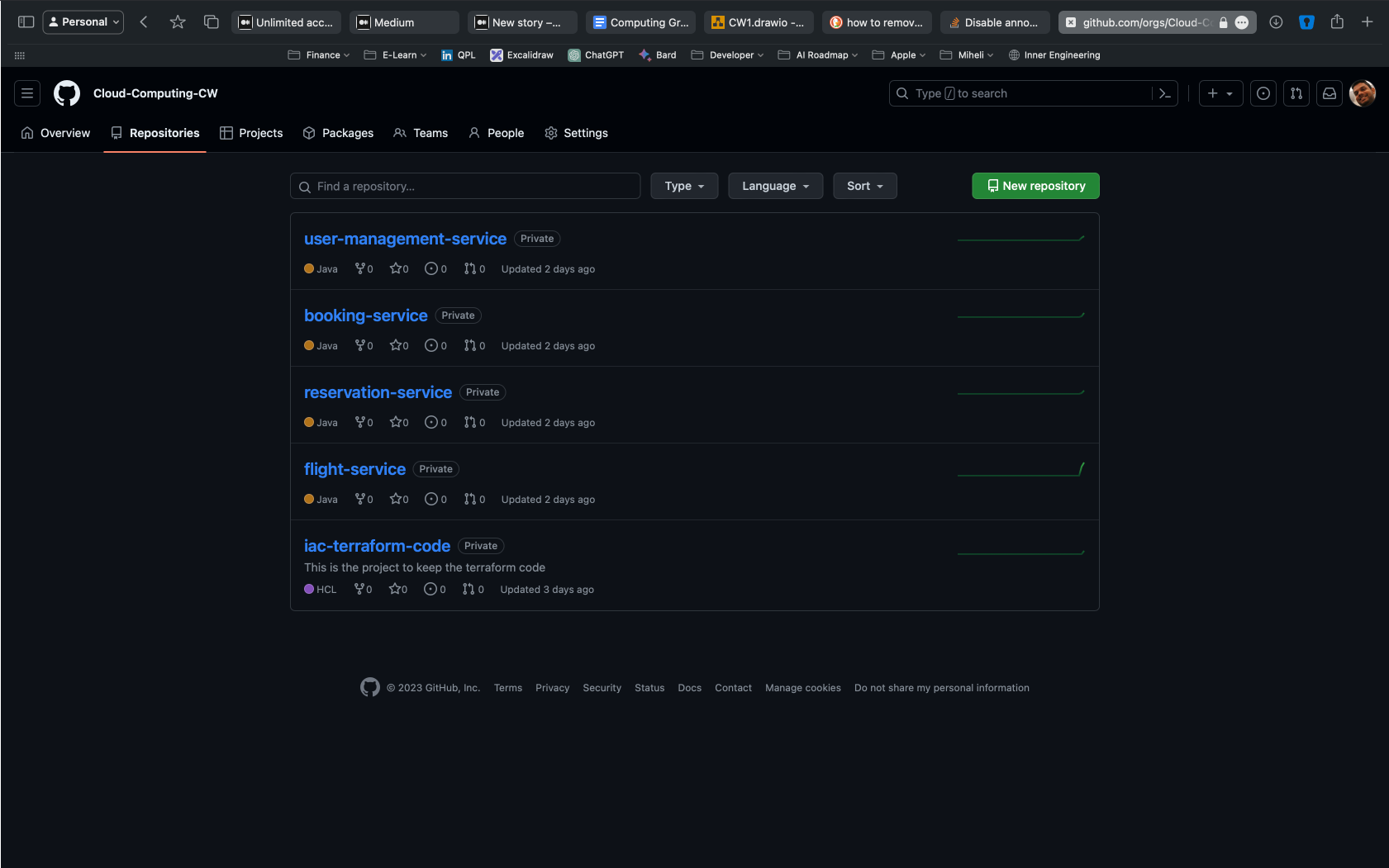


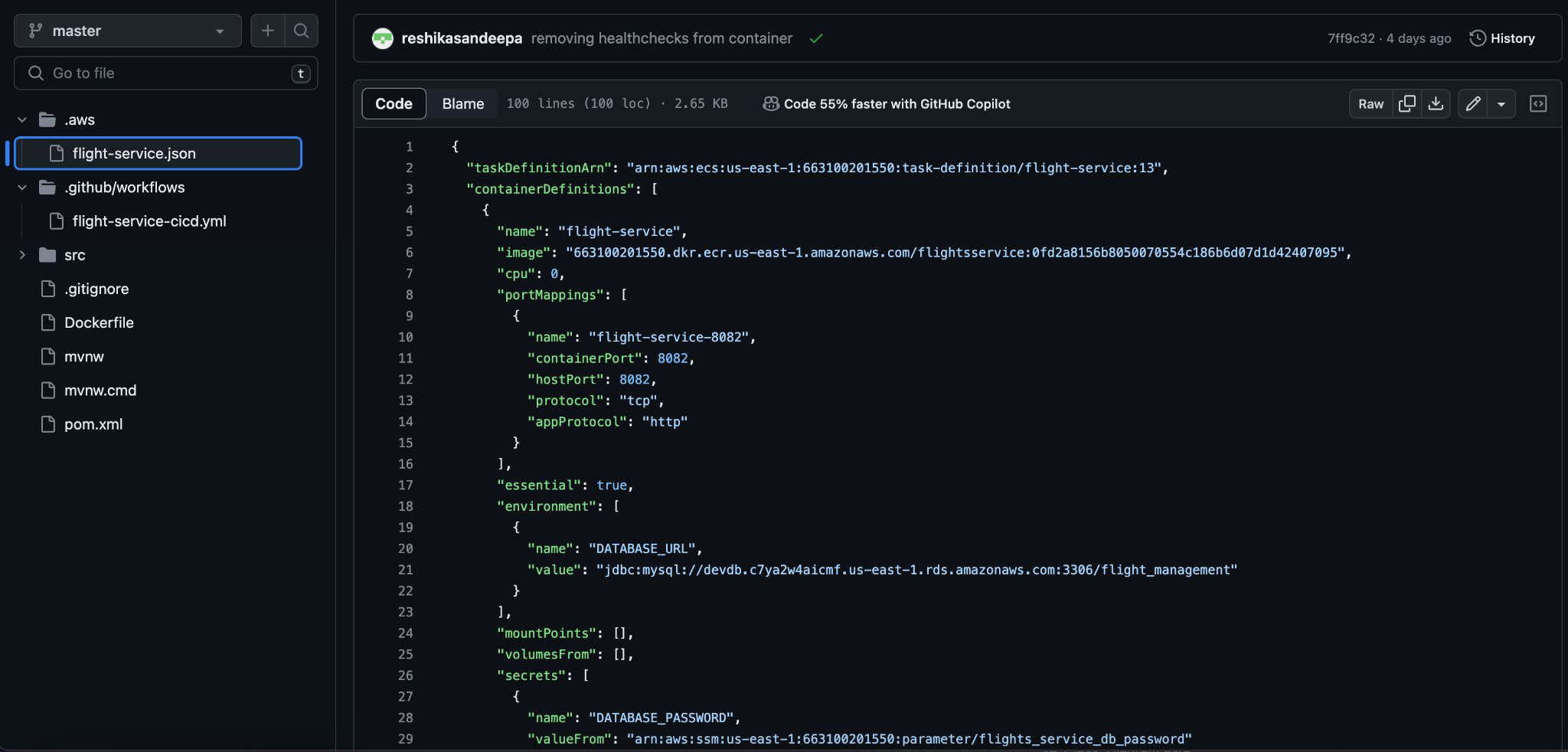


## Code Repositories and CI/CD

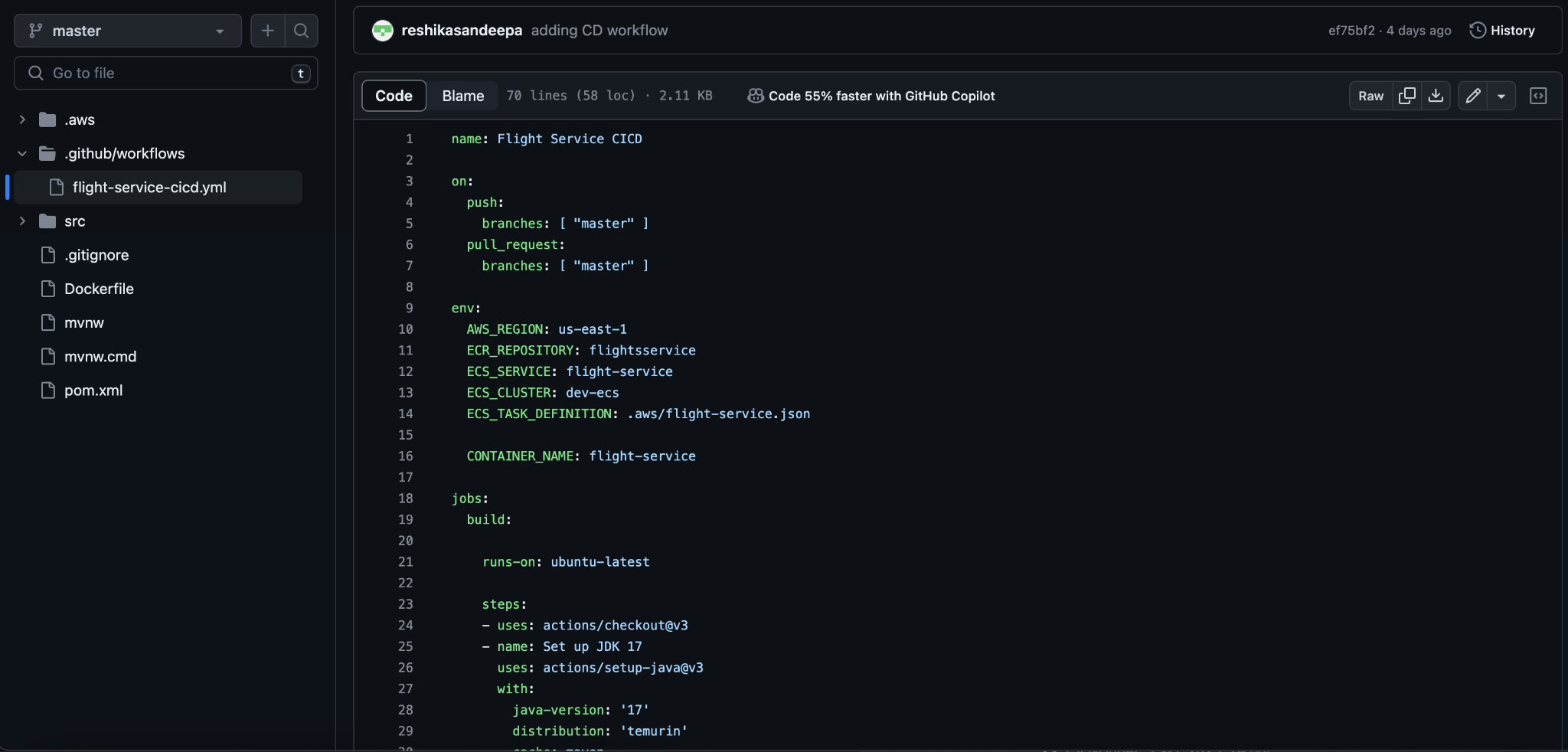
<https://github.com/orgs/Cloud-Computing-CW/repositories>

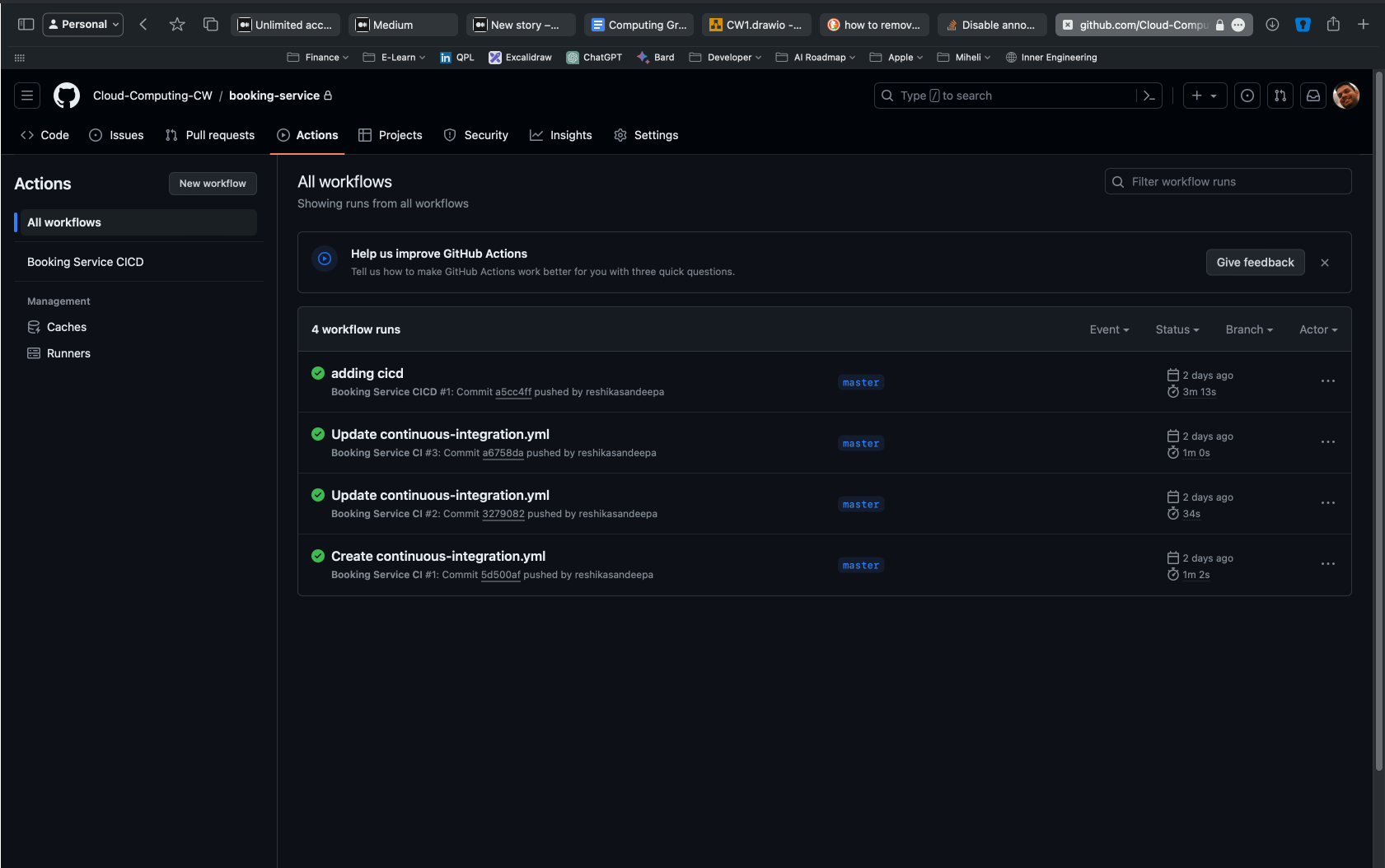
We have implemented the task definition for the container services, docker file and CI/CD YAML files in the github, above link will provide access to verify the implementation.









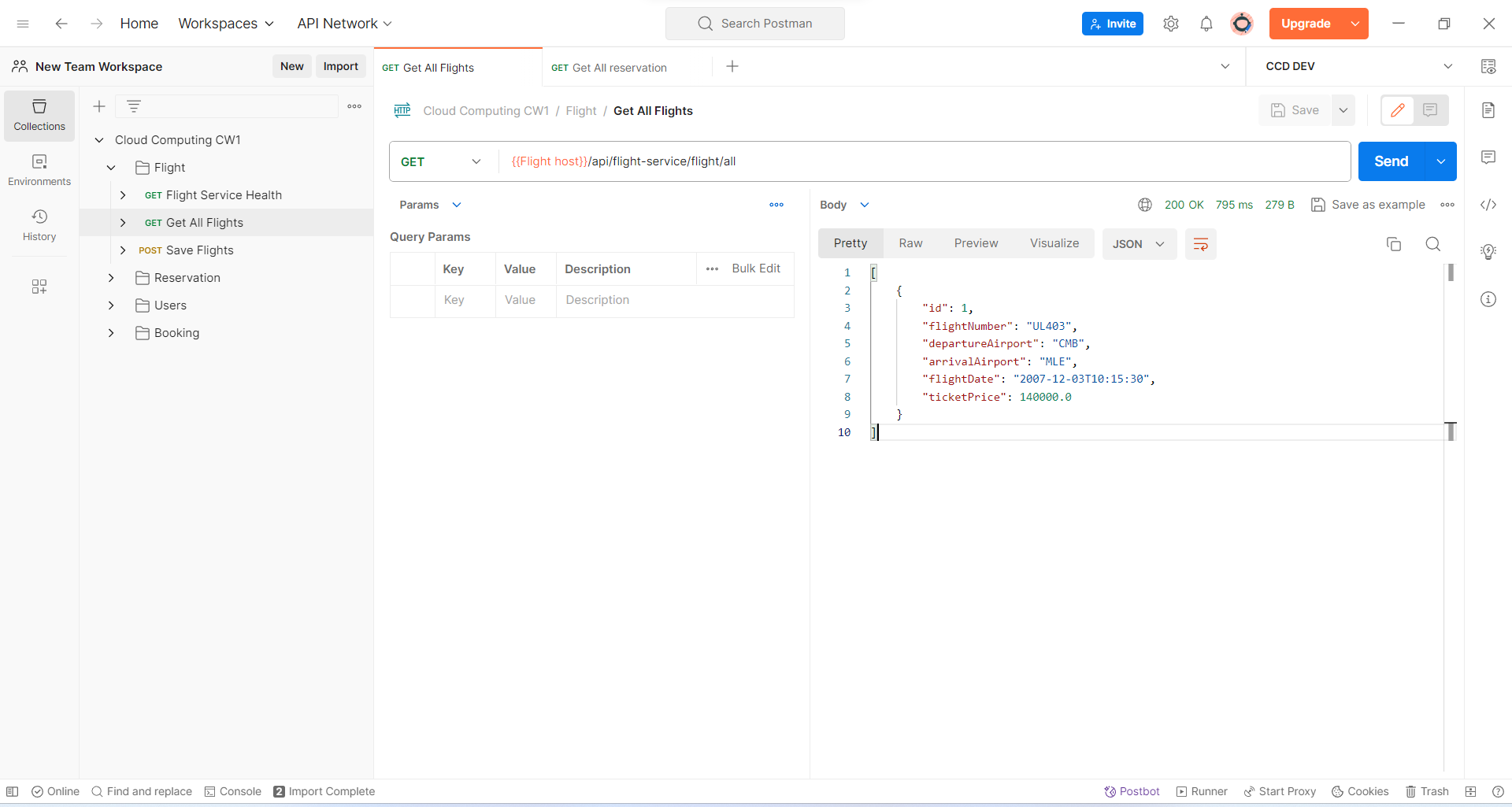


## Result Validations

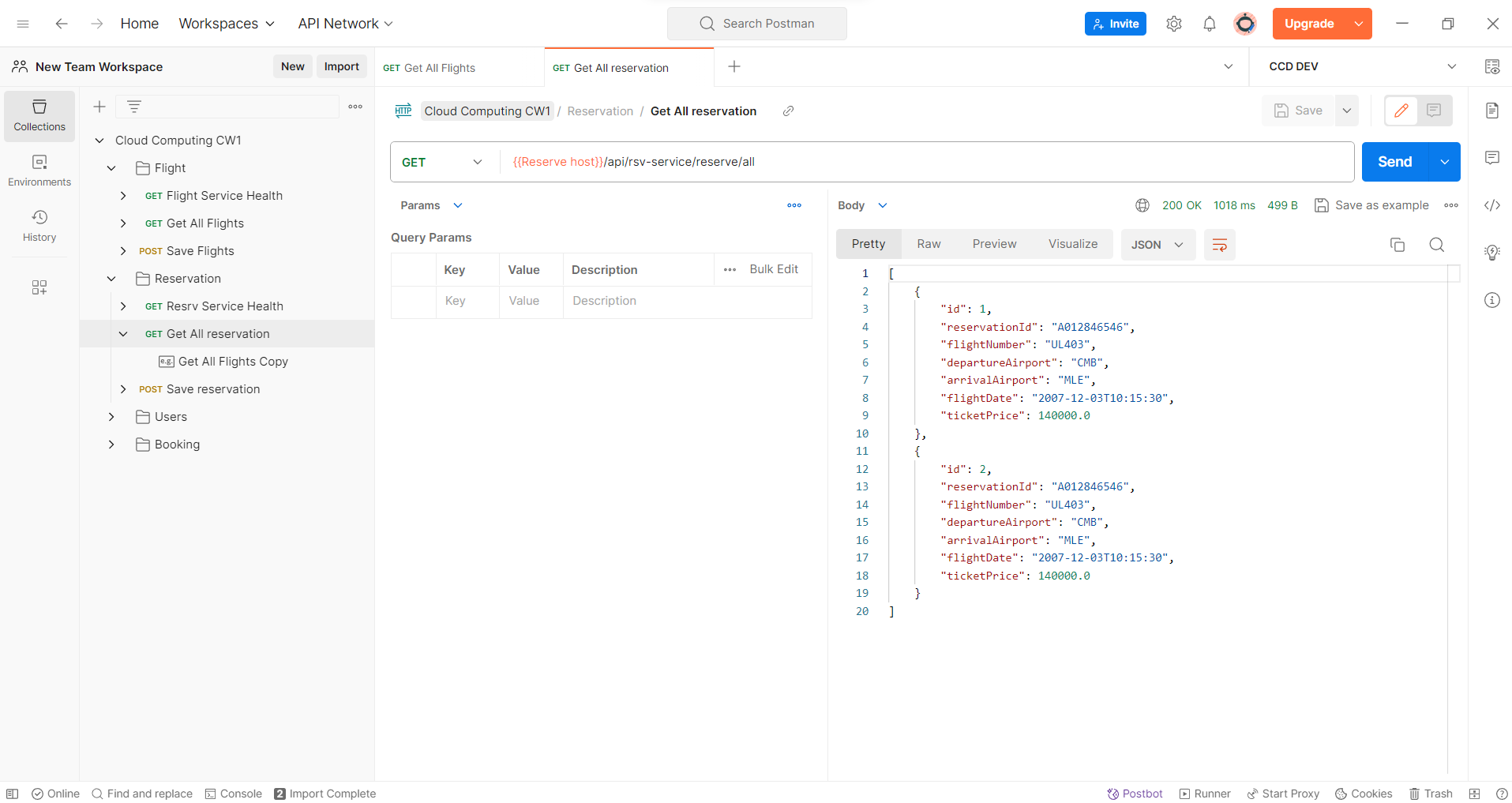
We have shown below the API response on the deployed infrastructure using Postman output. Following postman public URL can be used to access the deployed services to try out.

<https://documenter.getpostman.com/view/12087065/2s9YsDmFW7>

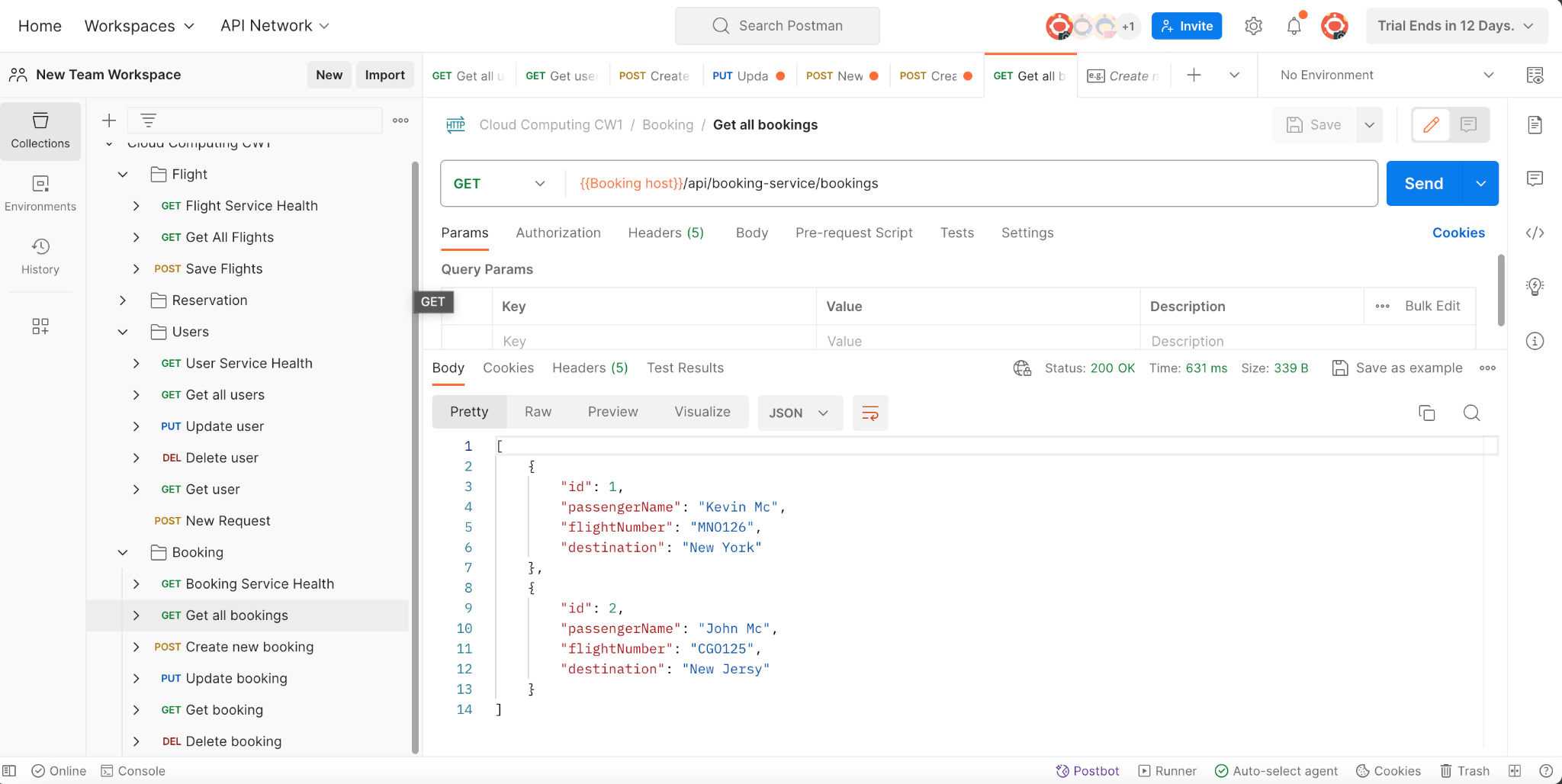
Flight Service



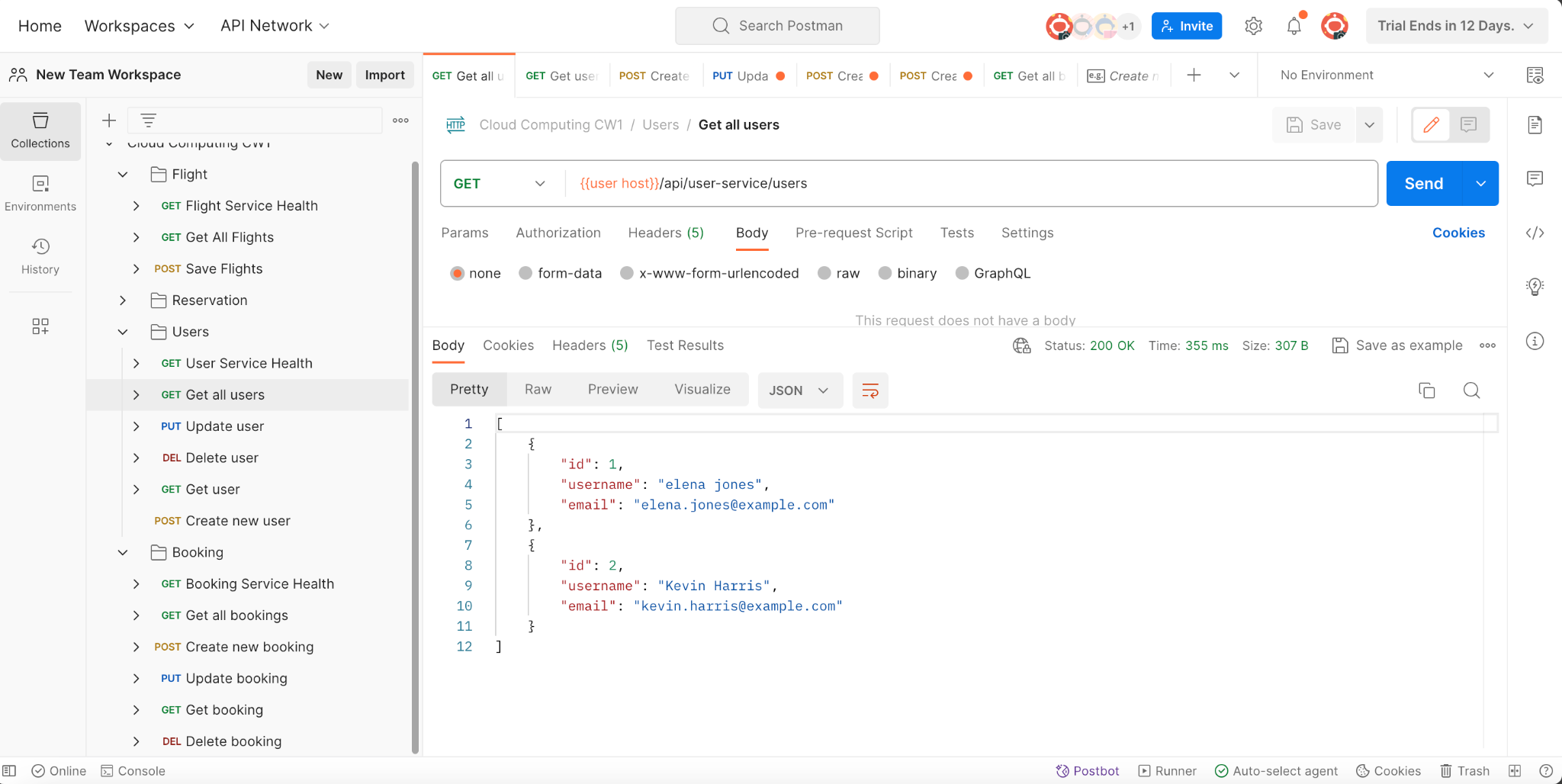
Reservation Service



Booking Service



User Service



## Challenges Faced and how we solved

**Challenge 01**

We are required to get the user id and password of the database to application via the AWS secrets to make the system more secure, However in order to do this we are required to implement the IAM Custom policy. This was identified by looking at the default AWS policy which is with full read and write so we removed the write part of one of the policies and created this new IAM policy to overcome this challenge (“Restricting access to Systems Manager parameters using IAM policies - AWS Systems Manager”).

**Challenge 02**

We were having doubts to choose between CloudFront and API Gateway, however given the nature of our solution we thought using API gateway will add more value due to API GW can be configured with WAF to protect against attacks and more advanced matrices and features availability of the API GW.

**Challenge 03**

When we were trying our code in AWS due to the Database Delete prevention we were required to manually delete the RDS and re-do the terraform destroy command, so for demonstration purposes we have removed this prevention and this needs to be enabled in the production environment.

**Challenge 04**

When we configured the load balancers we deployed public load balancers, however API GW cannot connect to LB without VPC link according to the AWS documentation as we found out later so we changed the LB to private subnets and pointed the network through the NAT Gateways (“Tutorial: Building an HTTP API with a private integration to an Amazon ECS service - Amazon API Gateway”).

**Challenge 05**

When we work with 4 different team members in the terraform code, we were facing issues in managing the terraform state files and we had to delete every time we pull from the Github so later found out that we can use remote backend with terraform which allows to use Hashi Corps own cloud services or AWS S3 with the DynamoDB table which made the process easier.

**Challenge 06**

At the beginning we were thinking of using Azure due to our team members being more familiar with Azure than AWS, however when we tried to create free Azure accounts it was blocking us from creating free accounts with the same credit/debit cards, emails and phone numbers. Since we all have already used what we have, we changed the cloud environment to AWS (“I can't create a free account in Azure - Microsoft Q&A”).

**Challenge 07**

According to our architecture diagram we have to create standby RDS with snapshot enabled for reliability, however this service was having cost attached to it in that case we decided to go with one RDS for the demo purposes.

**Challenge 08**

During the initial application setup we required to execute some queries in the RDS without our services running, however this was not possible due to the RDS was created in the private subnet, hence while keeping the same level of security we decided to deploy bastion host (EC2) to do the required maintenance tasks.