Technology Research Document

Name: Tony Jiang

Semester: 6

Class: RB04

Contents

Introduction

This document contains all the research I've done on the technology and methods I use, along with the reasons for my choices. This way, I can showcase to the teachers that I have explored the options and justified my decisions. This research can be for the individual project or the group project.

Technology for backend (individual)

This semester, I have the opportunity to explore a new programming language. A friend recommended that I try Golang, so I did some research. Initially, I was planning to use Java because of its vast library options, built-in security features, high scalability, and strong support for microservices.

However, Golang (Go), which was designed by Google, also offers several appealing features. It has a simple and easy-to-read design, fast compilation, strong support for microservices, and is excellent for cloud-native development. Additionally, it can easily scale horizontally and is highly favored for DevOps and infrastructure tasks. The only problem is that I need to learn how to code in Golang.

The obvious choice is Golang because it adheres to all the learning outcome of this semester.

Database selection (individual)

Choosing the right database for a project can be challenging. To make this decision, I use the CAP theorem and it is reference Canvas.

CAP theorem

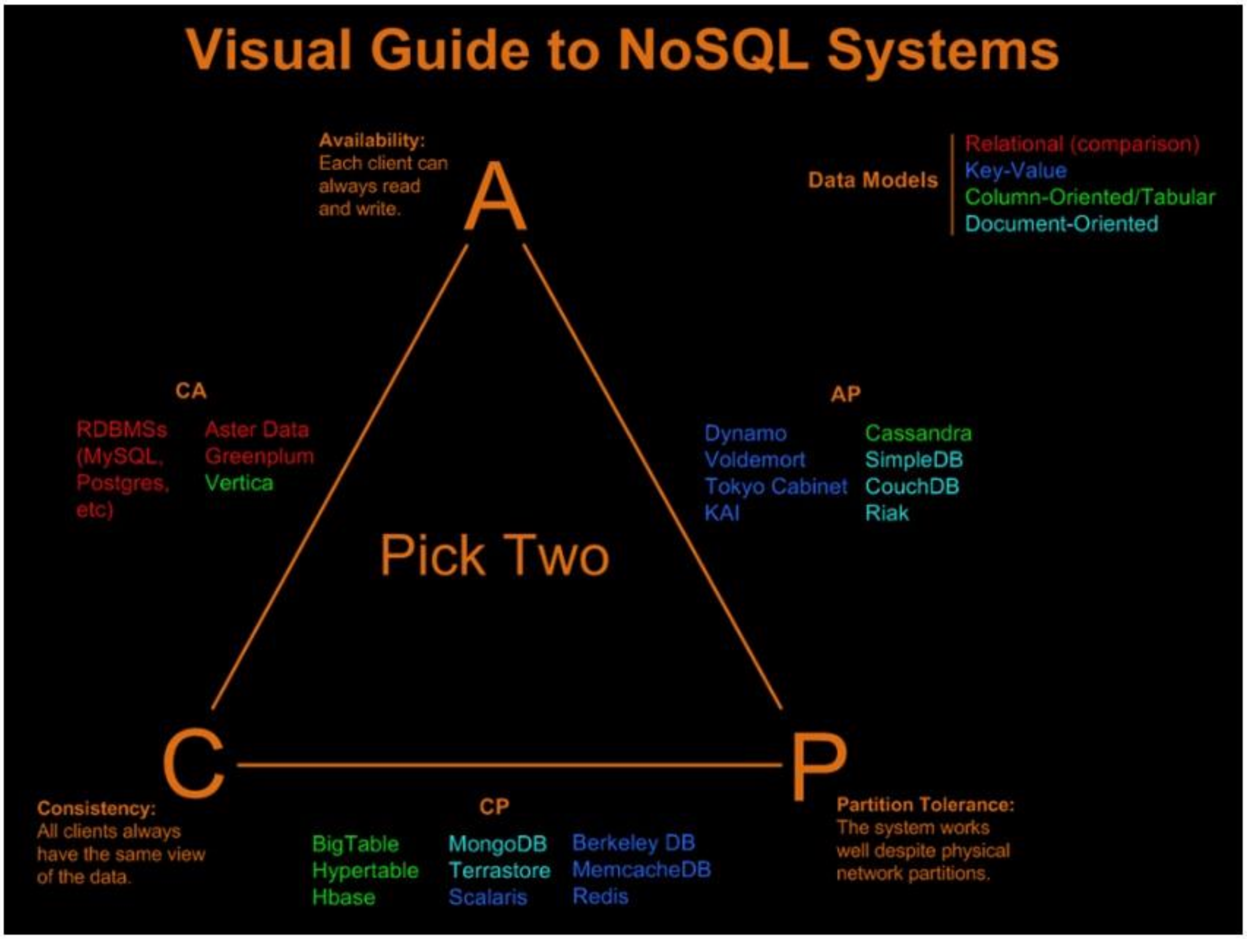
The CAP theorem is a fundamental concept in distributed database systems. It states that it is impossible for a distributed data store to simultaneously guarantee all three of the following properties:

- Consistency (C): Every read receives the most recent, correct, and up-to-date information, regardless of which server (or node) you ask.
- Availability (A): Every request (read or write) receives a response, even if it doesn't guarantee that the response contains the most recent write.
- Partition Tolerance (P): The system remains operational despite network partitions or communication breakdowns between nodes.

According to the CAP theorem, you can only guarantee two out of these three:

- CP (Consistency + Partition Tolerance): The system remains consistent and tolerates network partitions, but this comes at the cost of availability—some requests may fail.

- AP (Availability + Partition Tolerance): The system prioritizes responsiveness and handles network issues, but consistency may be sacrificed, meaning you might not always receive the latest data.
- CA (Consistency + Availability): The system ensures both accurate, up-to-date data and immediate responses, but it cannot tolerate network failures or partitions.



Based on the CAP theorem, my project should prioritize AP (Availability + Partition Tolerance) to ensure quick response times and resilience to network issues. My users prefer faster responses over strict data consistency since they are playing a game, and they value immediate feedback on their answers more than having perfectly consistent data. Therefore, my system will need an AP-focused database. A reference to AP databases is provided in the image above.

Polyglot persistence

In microservices architecture, it's common to use Polyglot Persistence, where each microservice can use the database best suited to its specific needs. You don't need to rely on a single type of database for the entire system. Different microservices can use different databases based on their requirements.

Database selection for service

Here is the database selection for the service that are currently implemented for the application.

User service

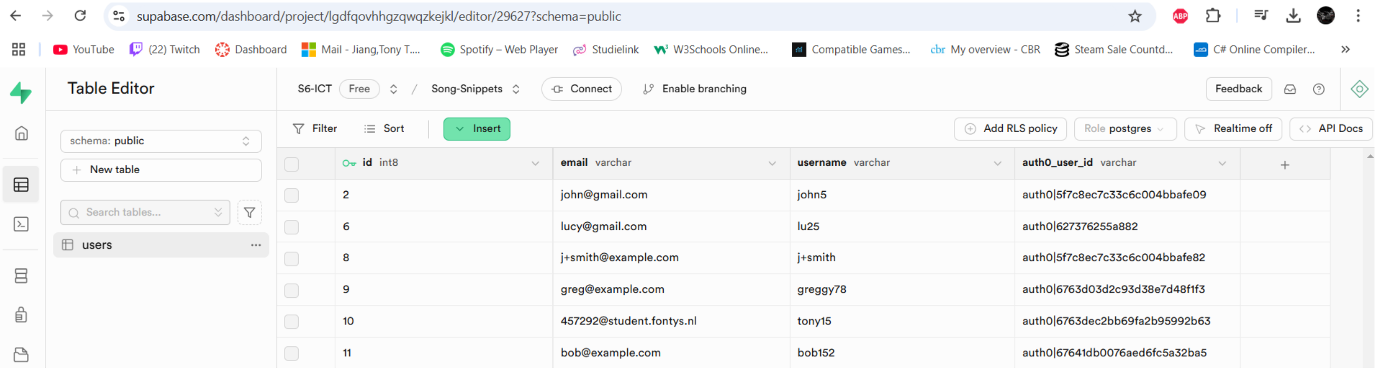
For the user service, I want to prioritize data accuracy and consistency. A relational database like PostgreSQL or MySQL is well-suited for this purpose, as these databases ensure that user data—such as account details, login credentials, and profile information—remains consistent and accurate.

PostgreSQL: Known for its robustness and support for complex queries, it is ideal if you anticipate needing to manage intricate relationships, such as user roles and permissions.

MySQL: A faster, reliable option for handling straightforward relational data and transactional workloads.

For my user service, I have chosen PostgreSQL because of its robustness and ability to handle complex relationships. Instead of hosting PostgreSQL locally, I will use a cloud-based solution to simplify future deployment. Using a cloud-hosted database eliminates the need to configure connections between my application and a remote PostgreSQL instance later on, making the setup more efficient from the start.

The PostgreSQL cloud provider I am using is Supabase, which offers a free tier. This includes 500 MB of storage, 2 GB of bandwidth, and 50 concurrent connections. While exceeding these limits incurs additional costs, the free tier is sufficient for my current needs.



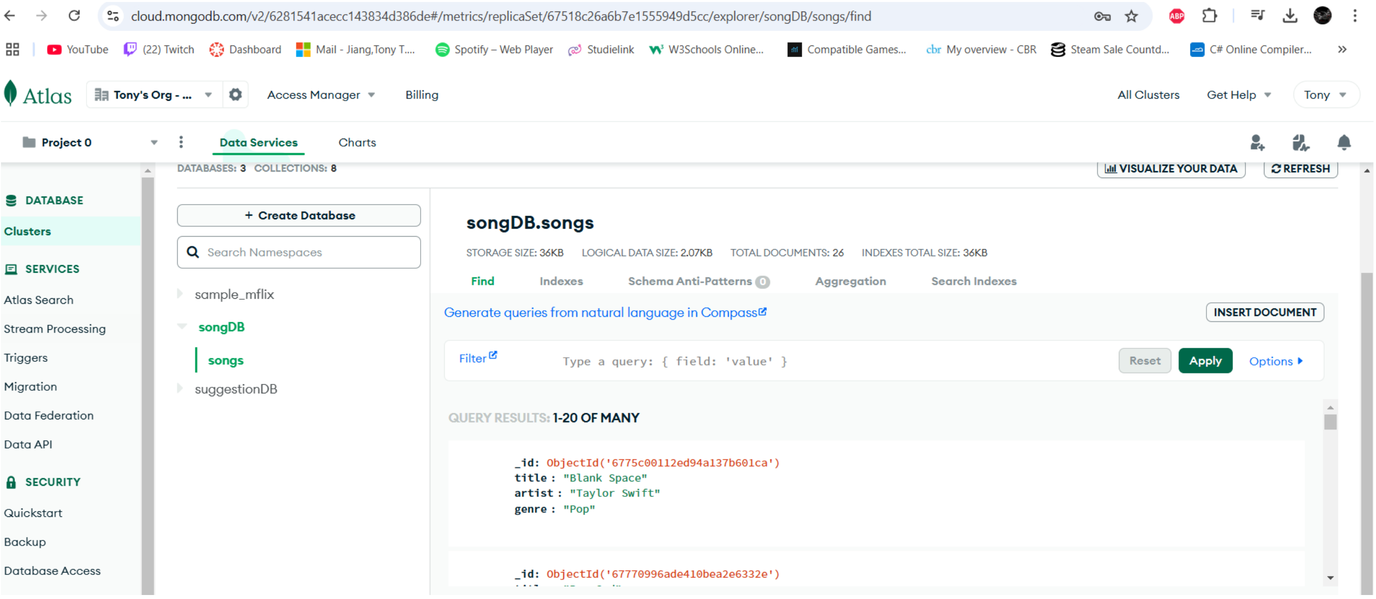
id int8	email varchar	username varchar	auth0_user_id varchar
2	john@gmail.com	john5	auth0|5f7c8ec7c33c6c004bbafe09
6	lucy@gmail.com	lu25	auth0|627376255a882
8	j+smith@example.com	j+smith	auth0|5f7c8ec7c33c6c004bbafe82
9	greg@example.com	greggy78	auth0|6763d03d2c93d38e7d48f1f3
10	457292@student.fontys.nl	tony15	auth0|6763dec2bb69fa2b95992b63
11	bob@example.com	bob152	auth0|67641db0076aed6fc5a32ba5

Song service

For the song service, MongoDB is an ideal database due to its flexible data structure and its ability to handle frequent read and write operations efficiently. MongoDB's document-based storage is well-suited for storing song metadata, such as the title, artist, genre, and other attributes, while also supporting frequent CRUD operations. Its schema flexibility allows for seamless modifications as the song data evolves over time.

MongoDB supports fast and flexible CRUD operations and scales horizontally, which is advantageous as the volume of song data grows. Additionally, it integrates easily with message brokers for sending events (e.g., CRUD operation events).

I have chosen MongoDB Atlas for cloud storage instead of hosting it locally. Using a cloud-based solution eliminates the need for future configuration and setup, saving time and effort. It's better to address this early in the project to streamline future development and deployment.



cloud.mongodb.com/v2/6281541acecc143834d386de#/metrics/replicaSet/67518c26a6b7e1555949d5cc/explorer/songDB/songs/find

DATABASES: 3 COLLECTIONS: 8

songDB.songs

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 2.07KB TOTAL DOCUMENTS: 26 INDEXES TOTAL SIZE: 36KB

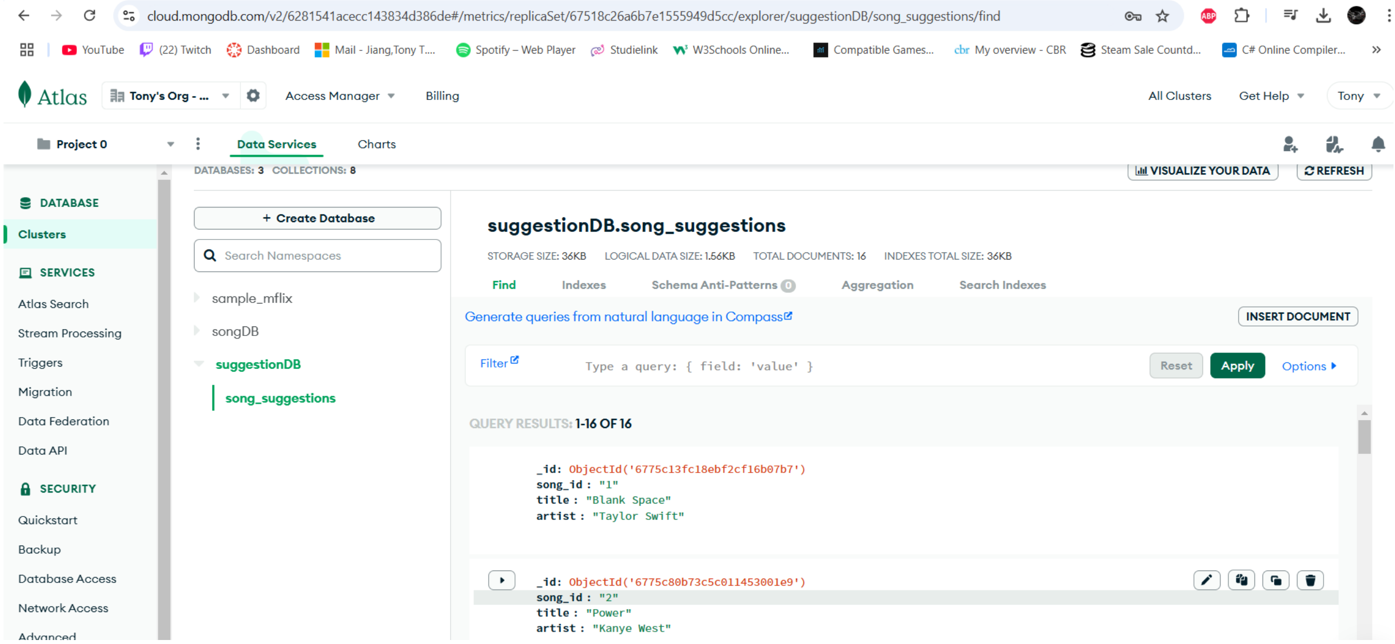
Type a query: { field: 'value' }

QUERY RESULTS: 1-20 OF MANY

```
_id: ObjectId('6775c00112ed94a137b601ca')
title : "Blank Space"
artist : "Taylor Swift"
genre : "Pop"

_id: ObjectId('67770996ade410bea2e6332e')
```

Song suggestion service

For the song suggestion service, it the same as the song service, which is using Mongo DB Atlas.



cloud.mongodb.com/v2/6281541acecc143834d386de#/metrics/replicaSet/67518c26a6b7e1555949d5cc/explorer/suggestionDB/song_suggestions/find

DATABASES: 3 COLLECTIONS: 8

suggestionDB.song_suggestions

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 1.56KB TOTAL DOCUMENTS: 16 INDEXES TOTAL SIZE: 36KB

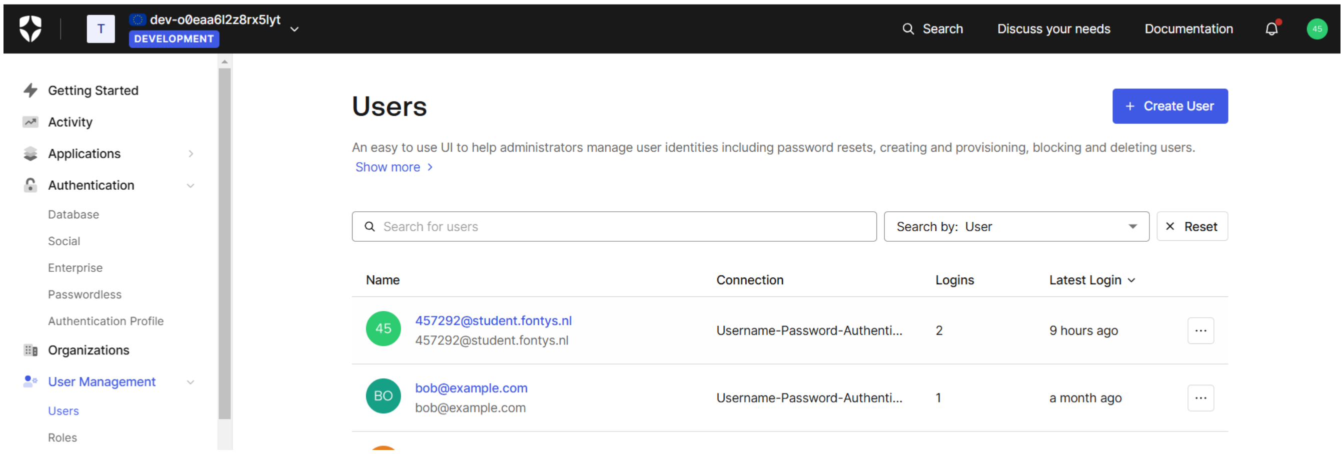
Type a query: { field: 'value' }

QUERY RESULTS: 1-16 OF 16

```
_id: ObjectId('6775c13fc18ebf2cf16b07b7')
song_id : "1"
title : "Blank Space"
artist : "Taylor Swift"

_id: ObjectId('6775c80b73c5c011453001e9')
song_id : "2"
title : "Power"
artist : "Kanye West"
```

Auth0

Auth0 is the external third party service which fulfill the authentication service, which I don’t have to configure any authentication and authorization and It also has it own database.



Users

An easy to use UI to help administrators manage user identities including password resets, creating and provisioning, blocking and deleting users.

Name	Connection	Logins	Latest Login
45 457292@student.fontys.nl 457292@student.fontys.nl	Username-Password-Authenti...	2	9 hours ago
BO bob@example.com bob@example.com	Username-Password-Authenti...	1	a month ago

Text file store (group)

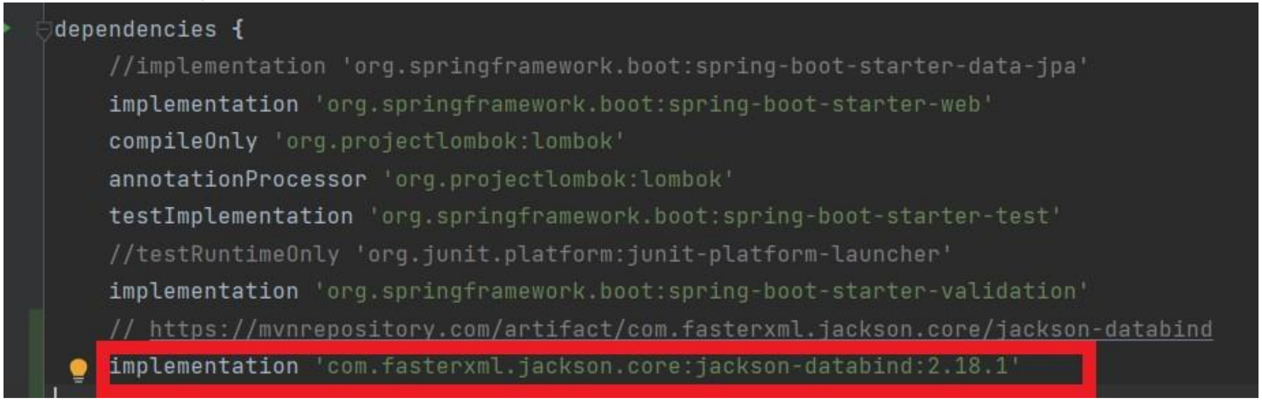
I was tasked with creating a text file storage solution for the botlist service for the group. Since I had no prior knowledge or experience in this area, I conducted some research on how to implement it. The language used to create the botlist service is Java, and the text file can be stored in JSON format.

I explored possible solutions for storing data in a text file using Java and identified two dependencies that could meet my requirements:

the Gson library and the Jackson library.

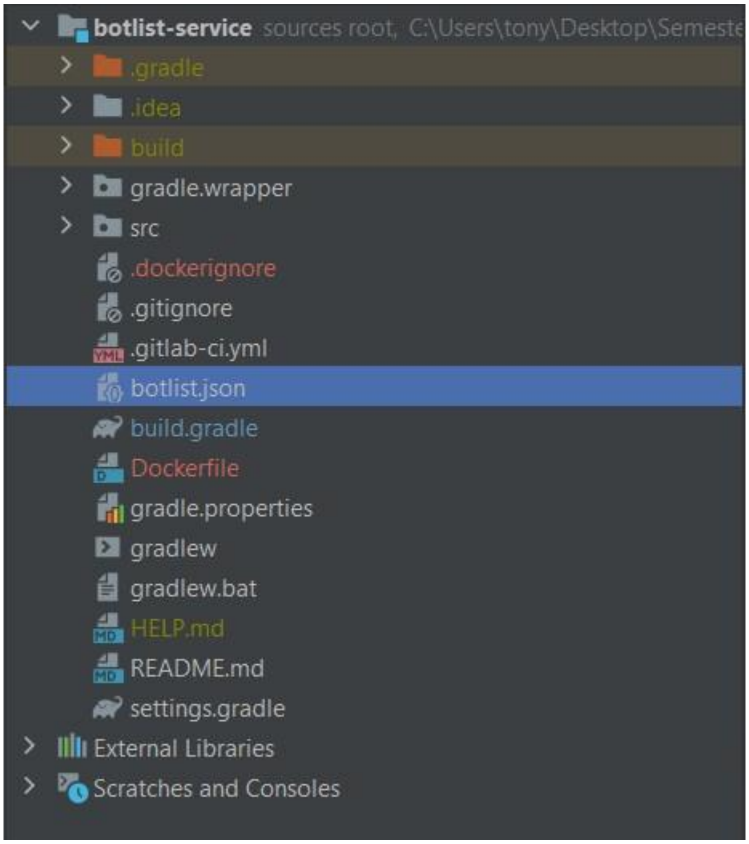
I chose the Jackson library due to its advanced features, better performance, and support for polymorphism. This choice also allows for future scalability, ensuring high performance and accommodating larger datasets if the company decides to expand.

While the Gson library is simpler and more beginner-friendly, I opted for Jackson to better suit the project’s needs. All you need to do is add the Jackson dependency in build.gradle, and you can start coding.

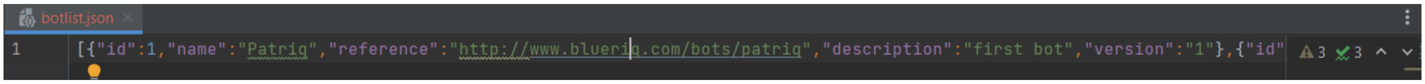


```
dependencies {
    //implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    //testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
    implementation 'org.springframework.boot:spring-boot-starter-validation'
    // https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.18.1'
```

The text file is stored in the root folder of the botlist service. If the file does not exist, the system will automatically create it when a botlist is added. However, the file does not track IDs, as it lacks an auto-increment mechanism like a database, so an auto-increment feature will need to be implemented manually.



```
botlist-service sources root, C:\Users\tony\Desktop\Semeste
  .gradle
  .idea
  build
  gradle.wrapper
  src
  .dockerignore
  .gitignore
  .gitlab-ci.yml
  botlist.json
  build.gradle
  Dockerfile
  gradle.properties
  gradlew
  gradlew.bat
  HELP.md
  README.md
  settings.gradle
External Libraries
Scratches and Consoles
```



```
[{"id":1,"name":"Patriq","reference":"http://www.blueriq.com/bots/patriq","description":"first bot","version":"1"},{"id"
```

Cloud technology (individual)

For the cloud technology, I’m planning to use cloud functions. While I have more experience with Kubernetes, I want to explore cloud functions as they are new to me. From my understanding, with cloud functions, instead of deploying an entire microservice to the cloud, you can deploy individual functions in a serverless manner. However, I’m not yet familiar with how they are deployed.

Reason for deploying to cloud

I am using the cloud for my project because it allows me to scale my application depending on the number of users. My application is a music guessing game, where users challenge themselves to guess the daily song or guess random songs. Since the application should handle a large number of users, deploying it to the cloud is ideal for scaling the project.

Cloud function vs Kubernetes cloud

The main difference between Kubernetes and cloud functions lies in their deployment and management models.

Kubernetes: With Kubernetes, you deploy and manage entire services, such as applications or containers, on the cloud. This typically involves setting up and maintaining infrastructure for scaling, availability, and load balancing. While Kubernetes provides robust control over the service lifecycle, it can lead to higher operational overhead and costs, as you're responsible for managing the infrastructure. The cost of hosting a service on Kubernetes begins as soon as the service is deployed and continues to accumulate based on how long your containers or pods are running.

Cloud Functions: In contrast, cloud functions are serverless. This means you deploy individual functions (specific pieces of code designed to perform tasks) rather than entire services. These functions are event-driven and run only when triggered—by user interactions, events, or other stimuli. Costs are determined by the number of invocations (triggers) and the function’s execution time, rather than continuous uptime like in Kubernetes. This pay-as-you-go model often makes cloud functions more cost-effective for smaller, event-driven tasks.

In summary: Kubernetes is designed for deploying and managing entire microservices on the cloud. For example, my song service, which handles CRUD operations for songs and manages song data, would be well-suited for Kubernetes. It allows me to manage multiple services, scale efficiently, and have fine-grained control over how my application is deployed and managed.

Cloud Functions are designed for simpler, event-driven tasks, such as my Song Suggestion service. This service only needs to process user input and suggest songs based on that input, making it ideal for a function-based architecture. With cloud functions, you pay for what you use, making it a more cost-effective option for smaller services.

While cloud functions can serve as an option for simple CRUD operations, deploying a full microservice (like a song service) with various interactions and data storage is often better suited to containers or Kubernetes for better scalability and management.

Cloud provider

For the cloud provider I can use Google, AWS or Azure. Azure is out of the question because I’m using Golang, I have to configure a lot of things for Golang. Google and AWS is ideal because of their ease of use, extensive support, and ecosystem integrations.

Pricing

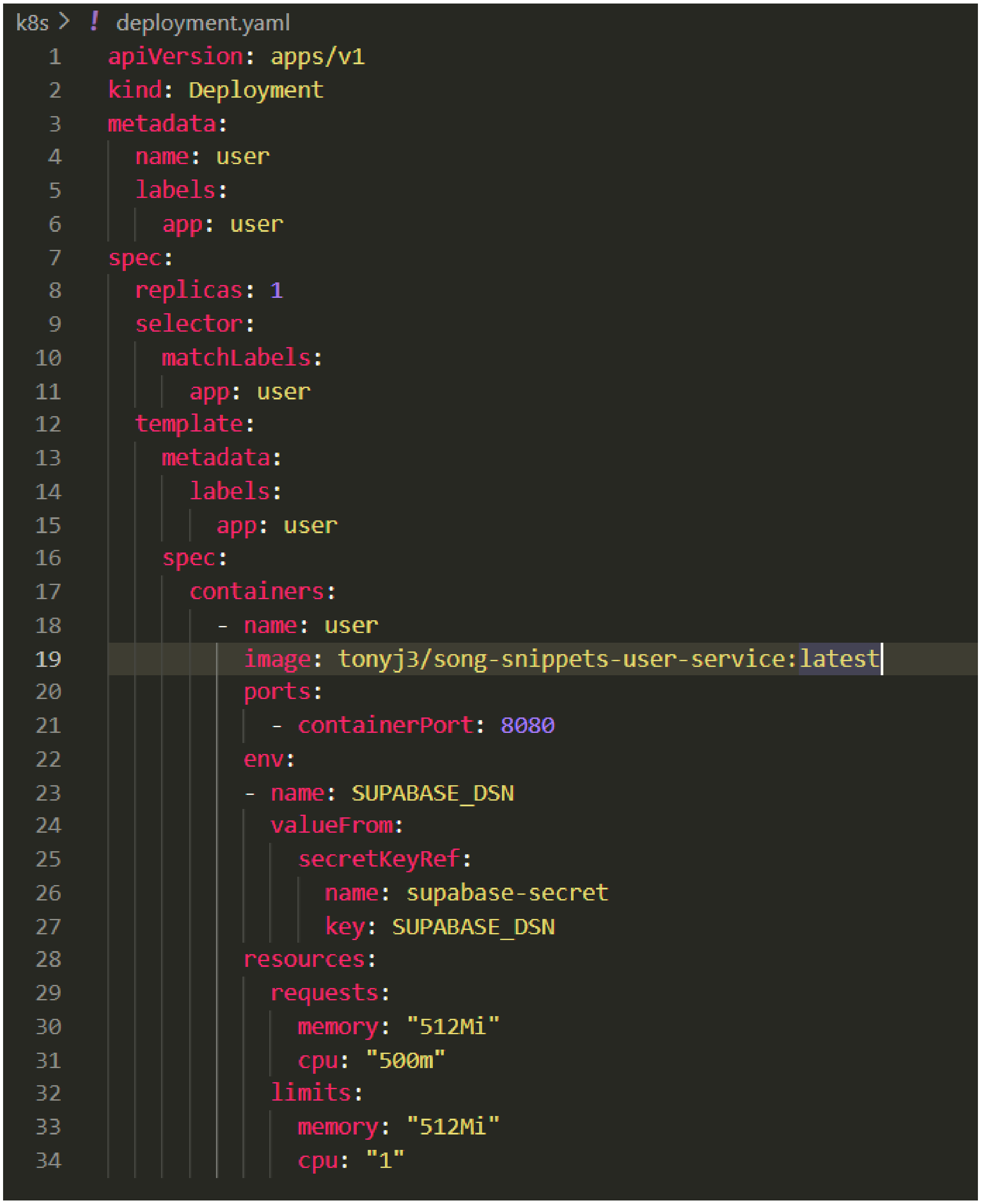
Google offers a free tier with $300 in credits valid for 90 days. After that, you are billed for usage. The free tier includes 180,000 vCPU-seconds and 360,000 GiB-seconds per month. Google bills execution time in vCPU-seconds and GiB-seconds. Beyond the free tier, the rates are $0.000018 per vCPU-second and $0.000002 per GiB-second. These rates can get more cost-effective with committed-use discounts for long-term, high-volume usage.

AWS provides a free tier with 7.5 billion GB-seconds per month, which is generous and suitable for low-cost startups. After exceeding the free tier, AWS bills based on GB-seconds (a combination of memory allocated and execution time). AWS charges $0.0000133334 per GB-second initially, with rates decreasing at higher usage volumes.

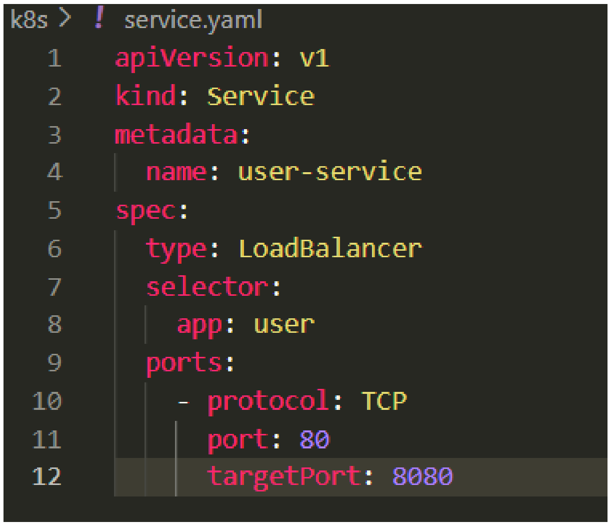
For testing purposes I'm using AWS, AWS is ideal due to its generous free tier. However, for long-term, high-volume scenarios, Google can be more cost-effective, especially with committed-use discounts.

Deploy manually to AWS EKS

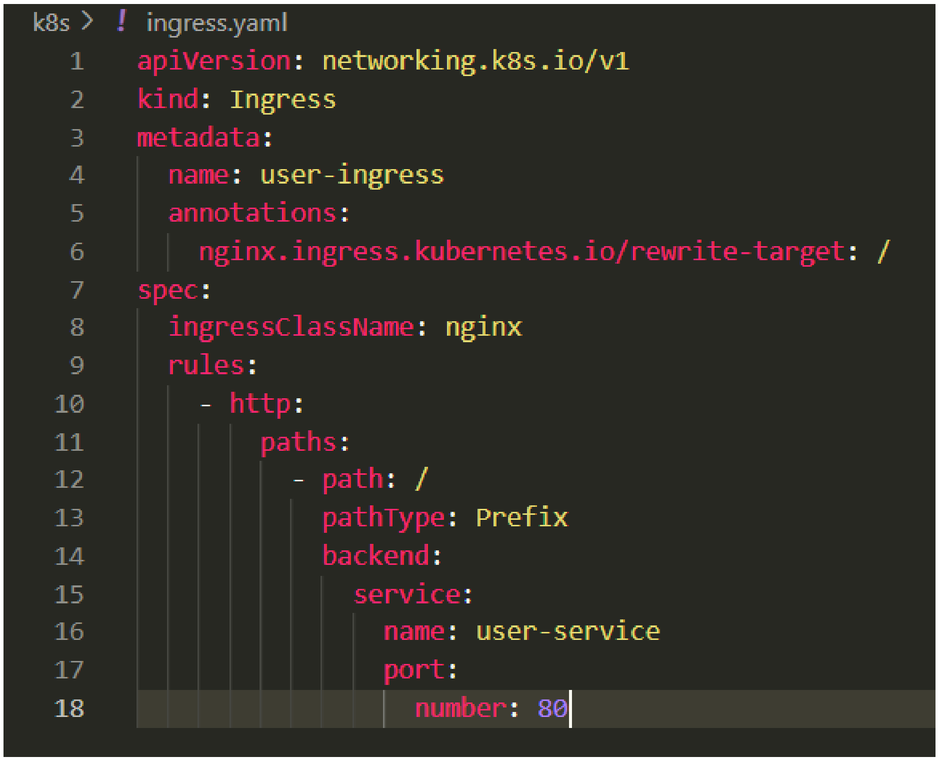
I've done this before, so I'll just summarize it. I use CloudShell on AWS to create an AWS EKS cluster, where I will deploy my user service. I need to create deployment.yaml, service.yaml, and ingress.yaml files. The deployment file is used to deploy the service to a pod, the service file defines the service for the pod (where you can add a load balancer), and the ingress file is used to expose the service through ingress.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user
  labels:
    app: user
spec:
  replicas: 1
  selector:
    matchLabels:
      app: user
  template:
    metadata:
      labels:
        app: user
    spec:
      containers:
        - name: user
          image: tonyj3/song-snippets-user-service:latest
          ports:
            - containerPort: 8080
          env:
          - name: SUPABASE_DSN
            valueFrom:
              secretKeyRef:
                name: supabase-secret
                key: SUPABASE_DSN
          resources:
            requests:
              memory: "512Mi"
              cpu: "500m"
            limits:
              memory: "512Mi"
              cpu: "1"
```

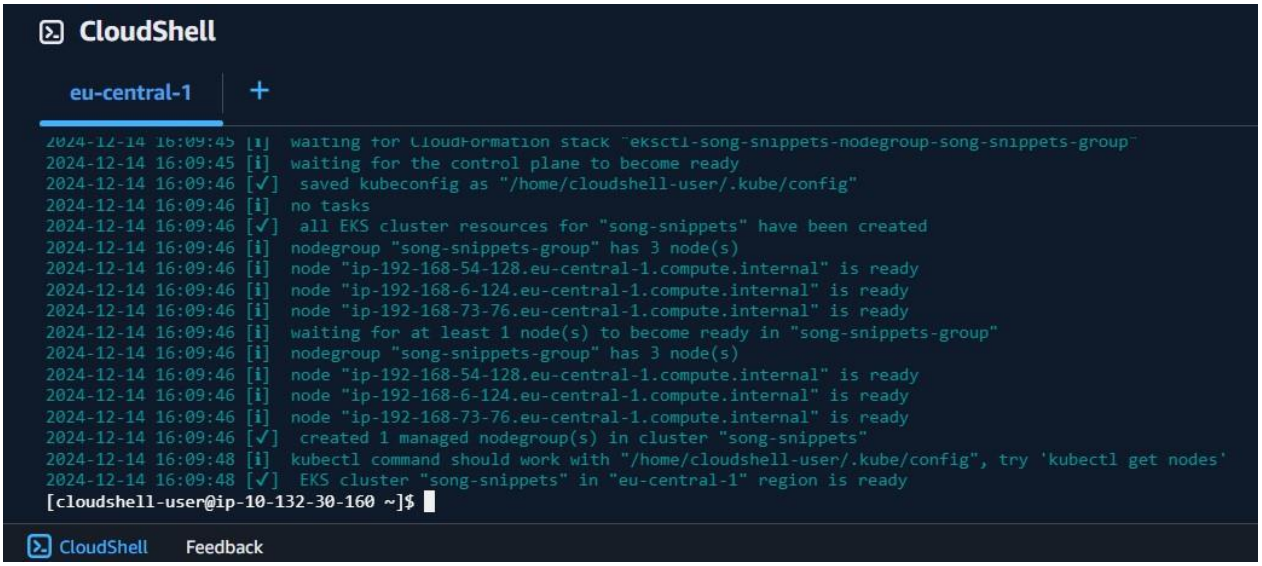


```yaml
apiVersion: v1
kind: Service
metadata:
  name: user-service
spec:
  type: LoadBalancer
  selector:
    app: user
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

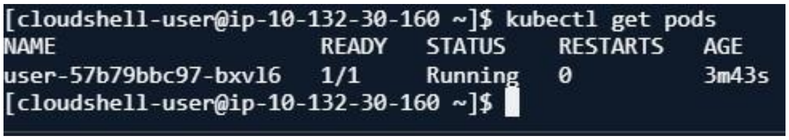


```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: user-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: user-service
                port:
                  number: 80
```

While creating the files, I also need to create the cluster before applying the deployment file. Once the cluster is created, I start by applying the deployment file. After that, I check if there are any issues with the pod using the following command: kubectl get pods.

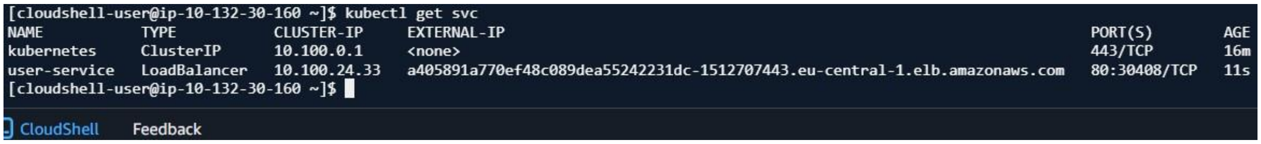


```
2024-12-14 16:09:45 [i]  waiting for CloudFormation stack "eksctl-song-snippets-nodegroup-song-snippets-group"
2024-12-14 16:09:45 [i]  waiting for the control plane to become ready
2024-12-14 16:09:46 [✓]  saved kubeconfig as "/home/cloudshell-user/.kube/config"
2024-12-14 16:09:46 [i]  no tasks
2024-12-14 16:09:46 [✓]  all EKS cluster resources for "song-snippets" have been created
2024-12-14 16:09:46 [i]  nodegroup "song-snippets-group" has 3 node(s)
2024-12-14 16:09:46 [i]  node "ip-192-168-54-128.eu-central-1.compute.internal" is ready
2024-12-14 16:09:46 [i]  node "ip-192-168-6-124.eu-central-1.compute.internal" is ready
2024-12-14 16:09:46 [i]  node "ip-192-168-73-76.eu-central-1.compute.internal" is ready
2024-12-14 16:09:46 [i]  waiting for at least 1 node(s) to become ready in "song-snippets-group"
2024-12-14 16:09:46 [i]  nodegroup "song-snippets-group" has 3 node(s)
2024-12-14 16:09:46 [i]  node "ip-192-168-54-128.eu-central-1.compute.internal" is ready
2024-12-14 16:09:46 [i]  node "ip-192-168-6-124.eu-central-1.compute.internal" is ready
2024-12-14 16:09:46 [i]  node "ip-192-168-73-76.eu-central-1.compute.internal" is ready
2024-12-14 16:09:46 [✓]  created 1 managed nodegroup(s) in cluster "song-snippets"
2024-12-14 16:09:48 [i]  kubectl command should work with "/home/cloudshell-user/.kube/config", try 'kubectl get nodes'
2024-12-14 16:09:48 [✓]  EKS cluster "song-snippets" in "eu-central-1" region is ready
[cloudshell-user@ip-10-132-30-160 ~]$
```

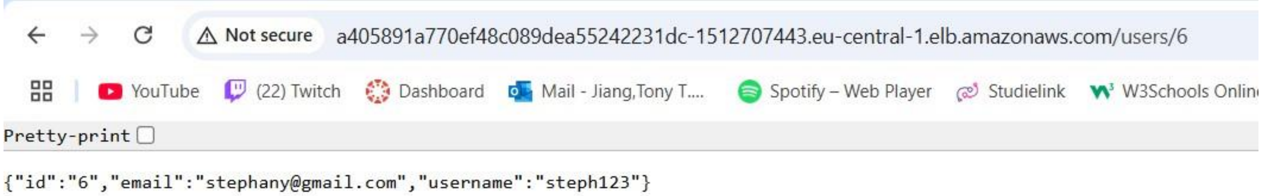


```
[cloudshell-user@ip-10-132-30-160 ~]$ kubectl get pods
NAME                   READY   STATUS    RESTARTS   AGE
user-57b79bbc97-bxvl6   1/1     Running   0          3m43s
[cloudshell-user@ip-10-132-30-160 ~]$
```

After that, I applied the service.yaml file, and I used kubectl get svc to check if it was applied. Additionally, if the service file includes a load balancer, it will display an external IP when you run kubectl get svc, which I can use to test it in Postman to see If it works.

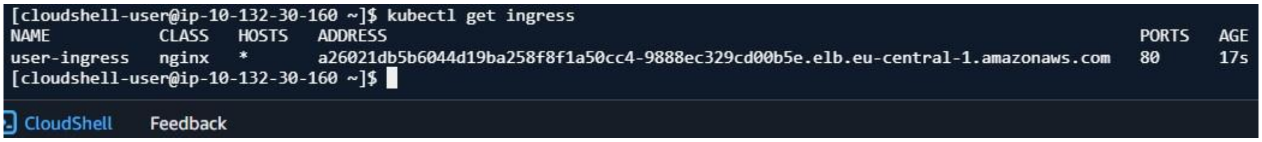


```
[cloudshell-user@ip-10-132-30-160 ~]$ kubectl get svc
NAME           TYPE           CLUSTER-IP     EXTERNAL-IP                                                                  PORT(S)        AGE
kubernetes     ClusterIP      10.100.0.1     <none>                                                                       443/TCP        16m
user-service   LoadBalancer   10.100.24.33   a405891a770ef48c089dea55242231dc-1512707443.eu-central-1.elb.amazonaws.com   80:30408/TCP   11s
[cloudshell-user@ip-10-132-30-160 ~]$
```

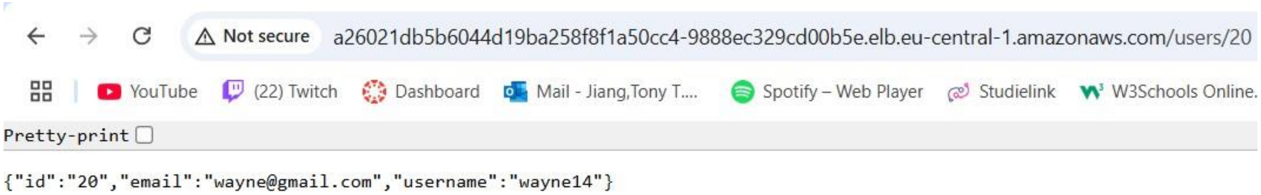


```
{"id":"6","email":"stephany@gmail.com","username":"steph123"}
```

For the ingress, I use ingress-nginx. I set up ingress-nginx and wait for it to be fully operational. After that, I applied the ingress file and retrieved the address for the ingress by running kubectl get ingress. I tested it in Postman to ensure it's working correctly, confirming that it is up and running and that the data is saved in my Supabase database.



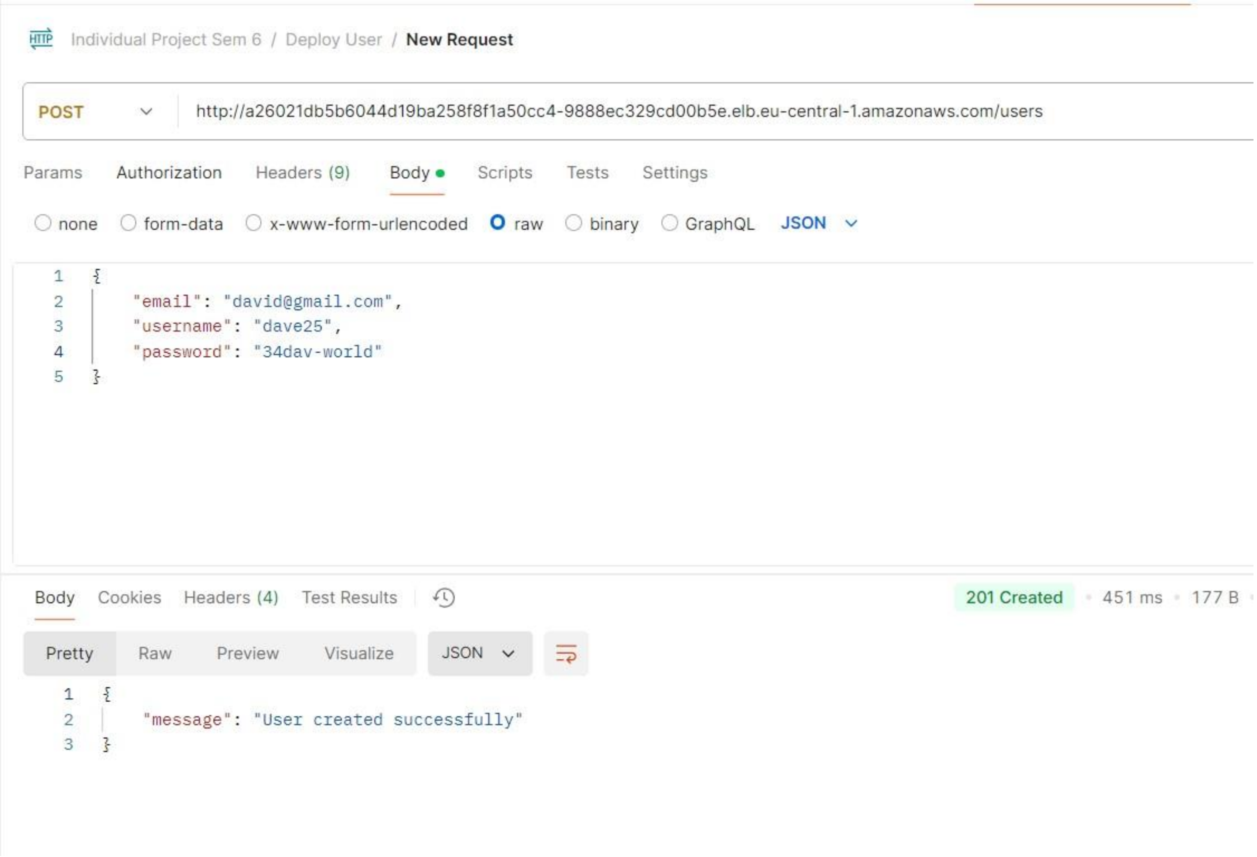
```
[cloudshell-user@ip-10-132-30-160 ~]$ kubectl get ingress
NAME           CLASS   HOSTS   ADDRESS                                                                           PORTS   AGE
user-ingress   nginx   *       a26021db5b6044d19ba258f8f1a50cc4-9888ec329cd00b5e.elb.eu-central-1.amazonaws.com   80      17s
[cloudshell-user@ip-10-132-30-160 ~]$
```



Not secure a26021db5b6044d19ba258f8f1a50cc4-9888ec329cd00b5e.elb.eu-central-1.amazonaws.com/users/20

Pretty-print □

```
{"id":"20","email":"wayne@gmail.com","username":"wayne14"}
```



Individual Project Sem 6 / Deploy User / **New Request**

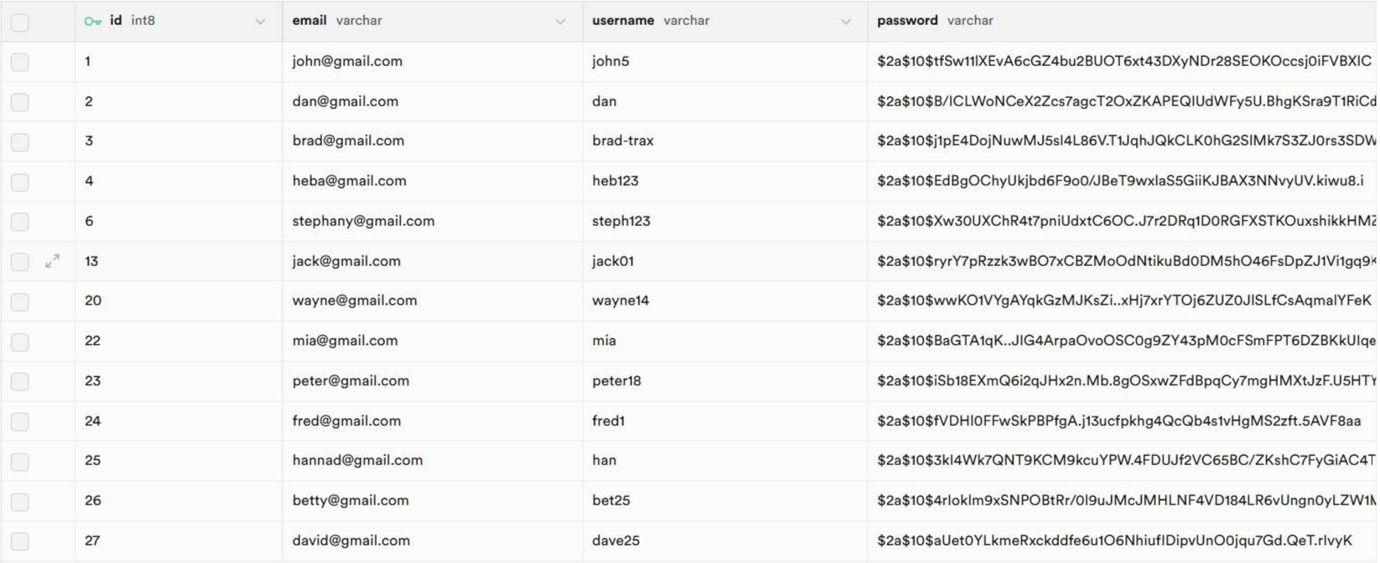
POST http://a26021db5b6044d19ba258f8f1a50cc4-9888ec329cd00b5e.elb.eu-central-1.amazonaws.com/users

○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

```
{
    "email": "david@gmail.com",
    "username": "dave25",
    "password": "34dav-world"
}
```

201 Created · 451 ms · 177 B

```
{
    "message": "User created successfully"
}
```

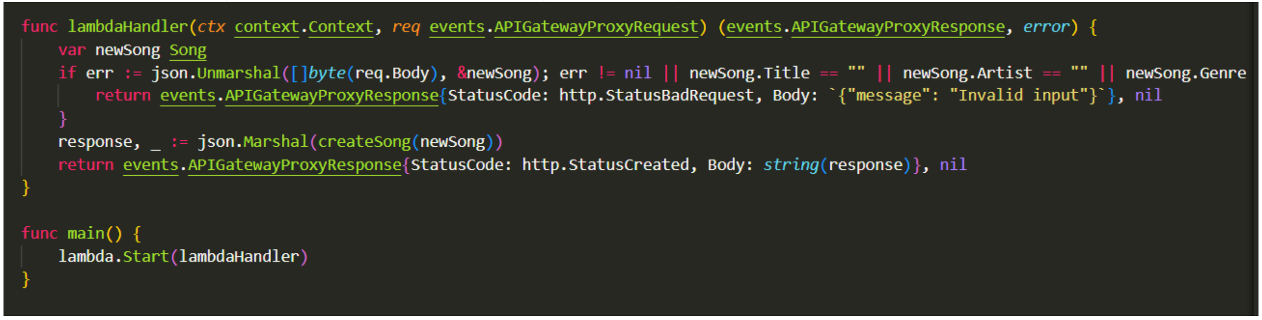


□	id int8	email varchar	username varchar	password varchar
□	1	john@gmail.com	john5	$2a$10$tfSw11lXEvA6cGZ4bu2BUOT6xt43DXyNDr28SEOKOccsj0iFVBXIC
□	2	dan@gmail.com	dan	$2a$10$B/ICLWoNCeX2Zcs7agcT2OxZKAPEQIUdWFy5U.BhgKSra9T1RiCd
□	3	brad@gmail.com	brad-trax	$2a$10$j1pE4DojNuwMJ5sl4L86V.T1JqhJQkCLK0hG2SIMk7S3ZJ0rs3SDW
□	4	heba@gmail.com	heb123	$2a$10$EdBgOChyUkjbd6F9o0/JBeT9wxlaS5GiiKJBAX3NNvyUV.kiwu8.i
□	6	stephany@gmail.com	steph123	$2a$10$Xw30UXChR4t7pniUdxtC6OC.J7r2DRq1D0RGFXSTKOuxshikkHMZ
□	13	jack@gmail.com	jack01	$2a$10$ryrY7pRzzk3wBO7xCBZMoOdNtikuBd0DM5hO46FsDpZJ1Vi1gq9
□	20	wayne@gmail.com	wayne14	$2a$10$wwKO1VYgAYqkGzMJKsZi..xHj7xrYTOj6ZUZ0JISLfCsAqmalYFeK
□	22	mia@gmail.com	mia	$2a$10$BaGTA1qK..JIG4ArpaOvoOSC0g9ZY43pM0cFSmFPT6DZBKkUlqe
□	23	peter@gmail.com	peter18	$2a$10$iSb18EXmQ6i2qJHx2n.Mb.8gOSxwZFdBpqCy7mgHMXtJzF.U5HTY
□	24	fred@gmail.com	fred1	$2a$10$fVDHI0FFwSkPBPfgA.j13ucfpkhg4QcQb4s1vHgMS2zft.5AVF8aa
□	25	hannad@gmail.com	han	$2a$10$3kI4Wk7QNT9KCM9kcuYPW.4FDUJf2VC65BC/ZKshC7FyGiAC4T
□	26	betty@gmail.com	bet25	$2a$10$4rloklm9xSNPOBtRr/0l9uJMcJMHLNF4VD184LR6vUngn0yLZW1M
□	27	david@gmail.com	dave25	$2a$10$aUet0YLkmeRxckddfe6u1O6NhiufIDipvUnO0jqu7Gd.QeT.rlvyK

This is how you deployed to AWS EKS.

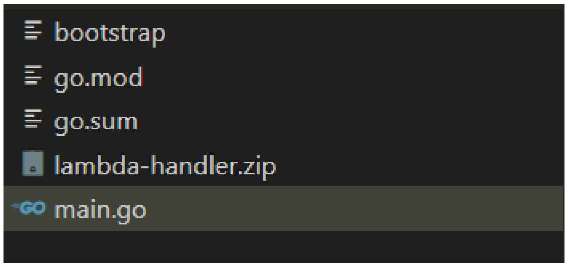
How to deploy manually for Golang

To test this, I use my song service to deploy a Golang function to AWS Lambda, I started by importing the AWS Lambda Go SDK into my project using the package "github.com/aws/aws-lambda-go". I then created a handler for Lambda, utilizing my CreateSong function as an example. To connect the handler to the Lambda runtime, I added it to the lambda.Start() function, ensuring that the code was ready for deployment.



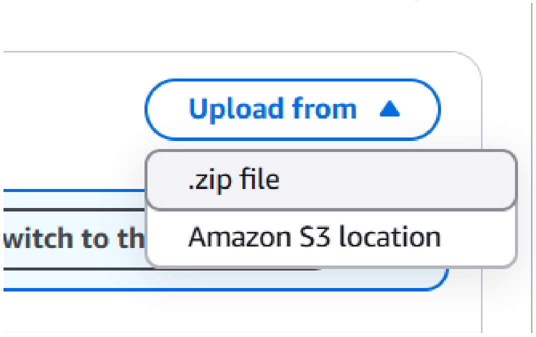
```go
func lambdaHandler(ctx context.Context, req events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse, error) {
    var newSong Song
    if err := json.Unmarshal([]byte(req.Body), &newSong); err != nil || newSong.Title == "" || newSong.Artist == "" || newSong.Genre
        return events.APIGatewayProxyResponse{StatusCode: http.StatusBadRequest, Body: `{"message": "Invalid input"}`}, nil
    }
    response, _ := json.Marshal(createSong(newSong))
    return events.APIGatewayProxyResponse{StatusCode: http.StatusCreated, Body: string(response)}, nil
}

func main() {
    lambda.Start(lambdaHandler)
}
```

Next, I built a binary file from my main.go file. Since AWS Lambda requires the binary to be named bootstrap, I made sure to name it accordingly. I used the following command to create the binary: $env:GOOS="linux"; $env:GOARCH="arm64"; $env:CGO_ENABLED="0"; go build -o bootstrap main.go. This step ensured that the binary was compatible with Lambda’s runtime environment. Additionally, I configured my environment file to match the environment instance I created in the Lambda UI. After creating the binary, I zipped it into a .zip archive and uploaded it to AWS Lambda.

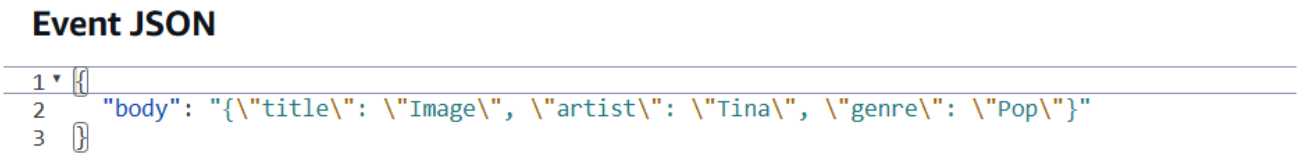


bootstrap
go.mod
go.sum
lambda-handler.zip
main.go

In the Lambda UI, I used the “Upload from” option and selected the .zip file for upload.

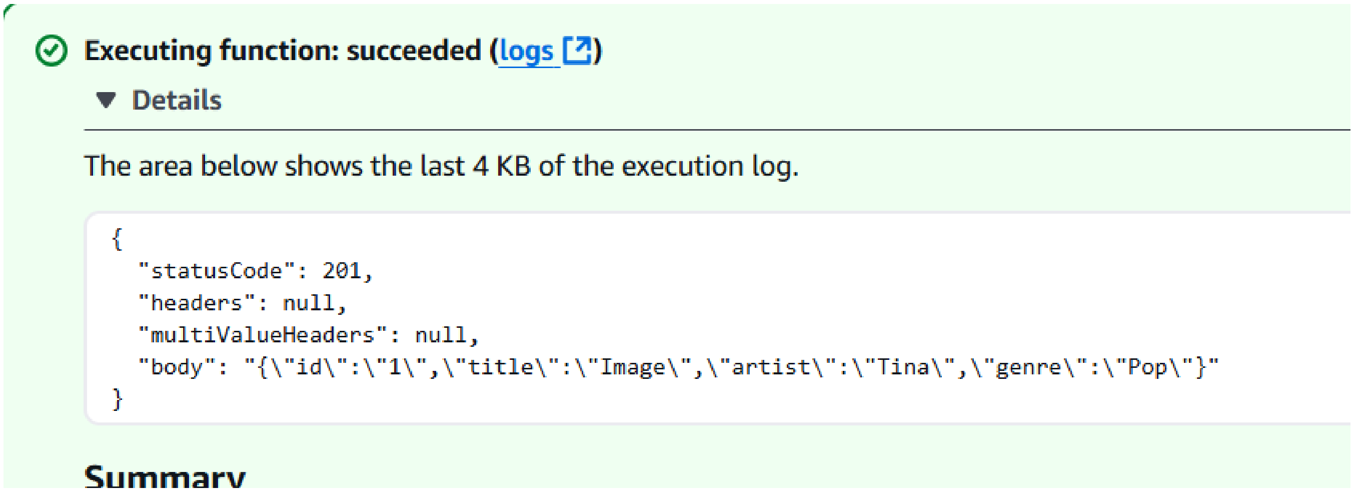


Once the file was uploaded, I tested the function within the Lambda UI. I noticed that the request body format for Lambda tests differed slightly from Postman, so I adjusted the test input accordingly. The function ran successfully, and I received a response confirming its execution.



Event JSON

```
{
  "body": "{\"title\": \"Image\", \"artist\": \"Tina\", \"genre\": \"Pop\"}"
}
```



Executing function: succeeded (logs)

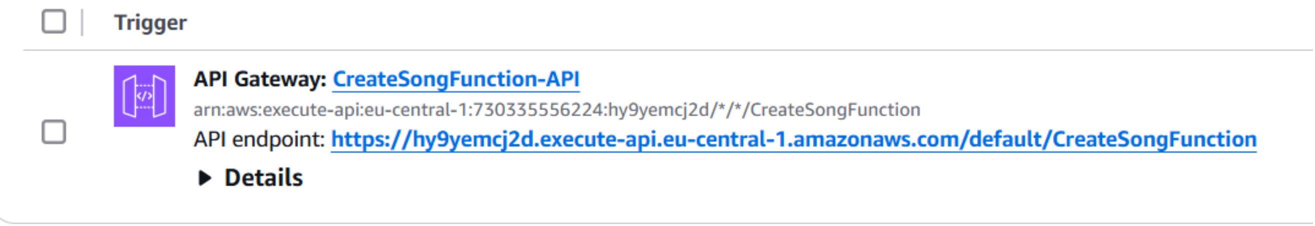
▼ Details

The area below shows the last 4 KB of the execution log.

```
{
  "statusCode": 201,
  "headers": null,
  "multiValueHeaders": null,
  "body": "{\"id\":\"1\",\"title\":\"Image\",\"artist\":\"Tina\",\"genre\":\"Pop\"}"
}
```

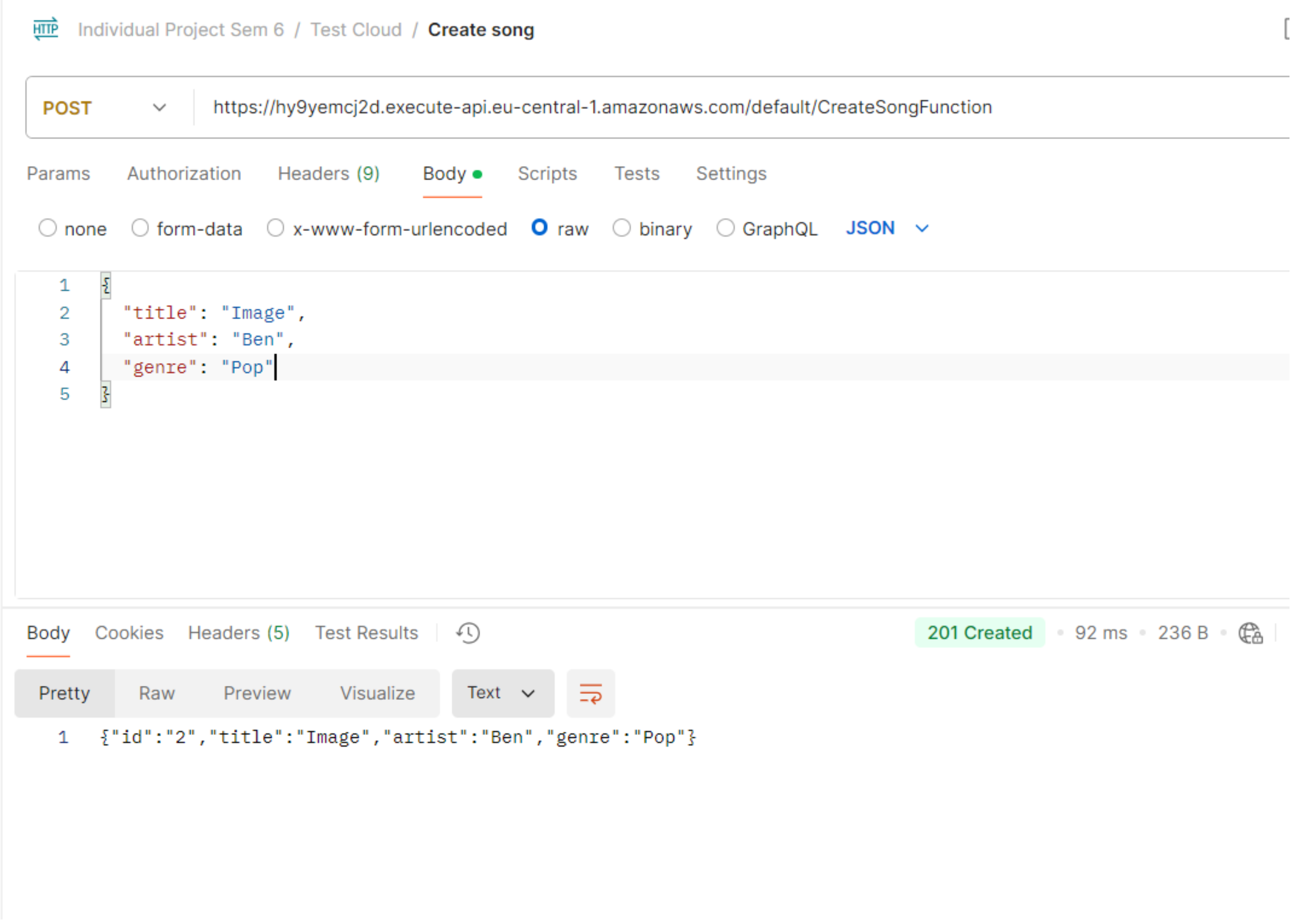
Summary

To make the function accessible, I added an API Gateway as a trigger. This setup generated a URL for the function, allowing me to send requests. In this case, I configured the API Gateway to handle POST requests. Finally, I tested the URL in Postman to verify that the function was triggered correctly and performed as expected.



□ Trigger

□ **API Gateway: CreateSongFunction-API**
arn:aws:execute-api:eu-central-1:730335556224:hy9yemcj2d/*/*/CreateSongFunction
API endpoint: https://hy9yemcj2d.execute-api.eu-central-1.amazonaws.com/default/CreateSongFunction
▶ **Details**



Individual Project Sem 6 / Test Cloud / **Create song**

POST https://hy9yemcj2d.execute-api.eu-central-1.amazonaws.com/default/CreateSongFunction

Params Authorization Headers (9) Body ● Scripts Tests Settings

○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

```
{
  "title": "Image",
  "artist": "Ben",
  "genre": "Pop"
}
```

Body Cookies Headers (5) Test Results

201 Created • 92 ms • 236 B

Pretty Raw Preview Visualize Text

```
{"id":"2","title":"Image","artist":"Ben","genre":"Pop"}
```

That’s the process I followed to deploy a Golang function to AWS Lambda. By using the Lambda UI and some manual steps, I successfully deployed, tested, and triggered the function.

How to automatically deploy from CI/CD

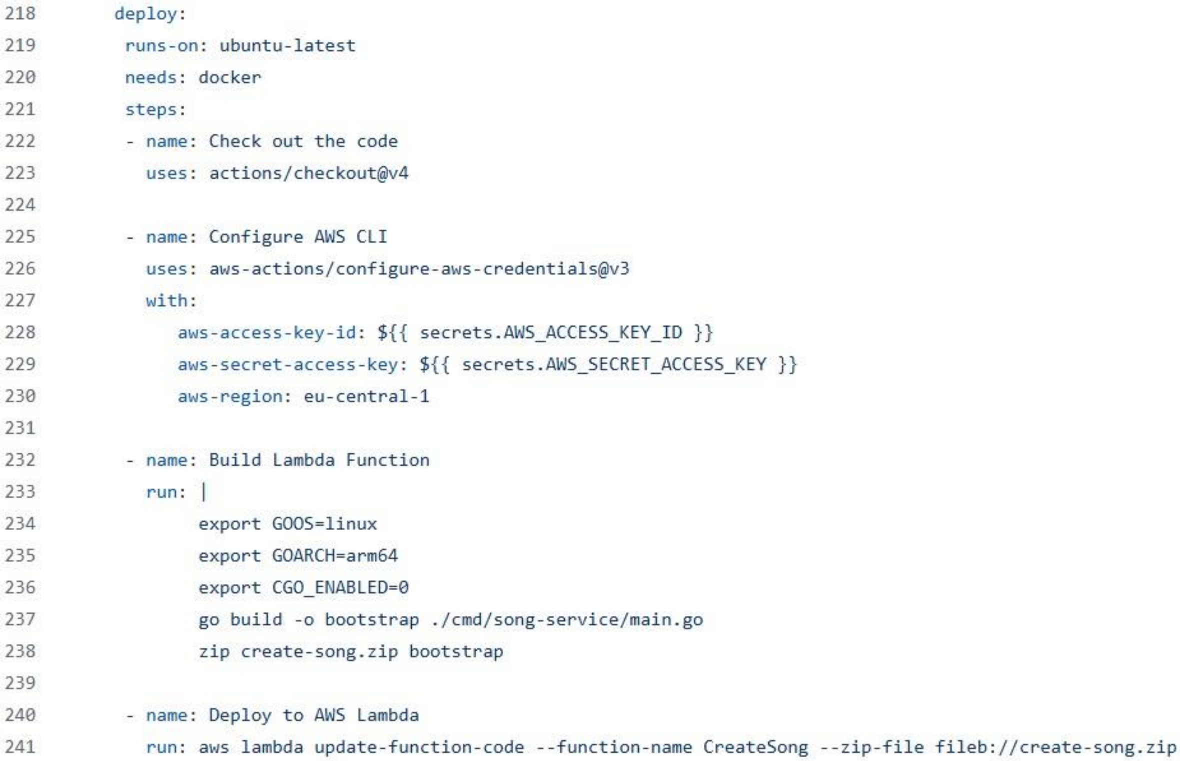
To automate deployment to an AWS Lambda function, here’s how I approached it:

First, I manually created the Lambda function in AWS. This step ensures that the function exists and can be updated later through the CI/CD pipeline. Next, I generated an access key and access key secret in AWS. These credentials are essential for logging in and interacting with AWS services through the CI/CD pipeline.

Since I use GitHub Actions for my CI/CD workflow, I integrated these credentials into the pipeline to automate the steps I used to perform manually. These included building the project, zipping the files, and deploying them to Lambda.

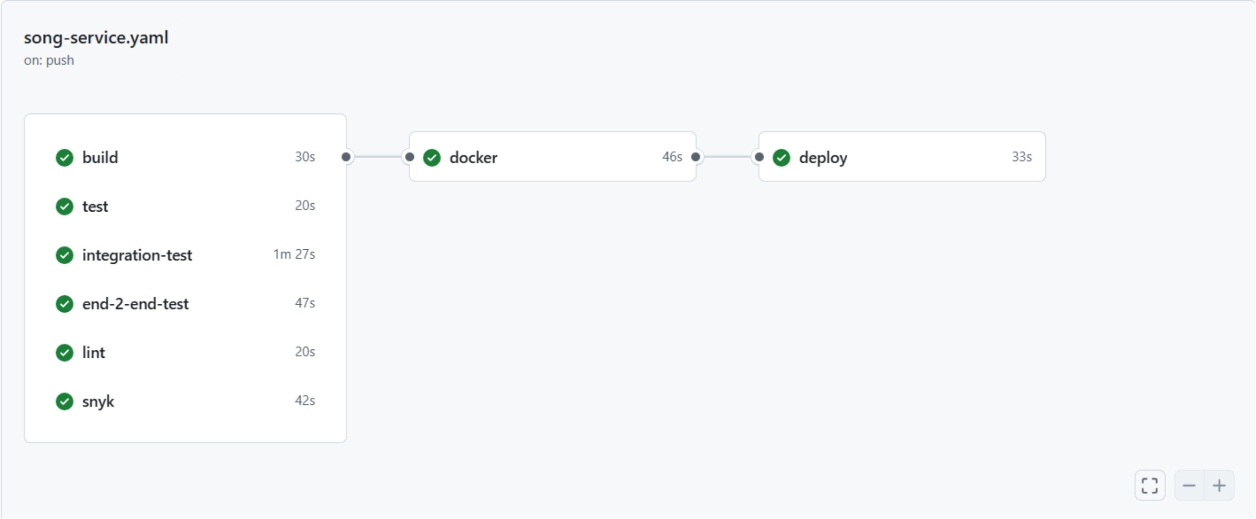
Steps I Followed to Configure CI/CD

- **Configuring the AWS CLI:** In the CI/CD script, I configured the AWS CLI by providing the access key and access key secret. This setup allows the pipeline to authenticate with AWS.
- **Building and Zipping the Project**: I automated the process of building the project and packaging it into a zip file. This replaced the manual step I used to do on my local machine.
- **Deploying the Update to Lambda:** Finally, I used the AWS CLI to deploy the zip package to the Lambda function, updating its code.



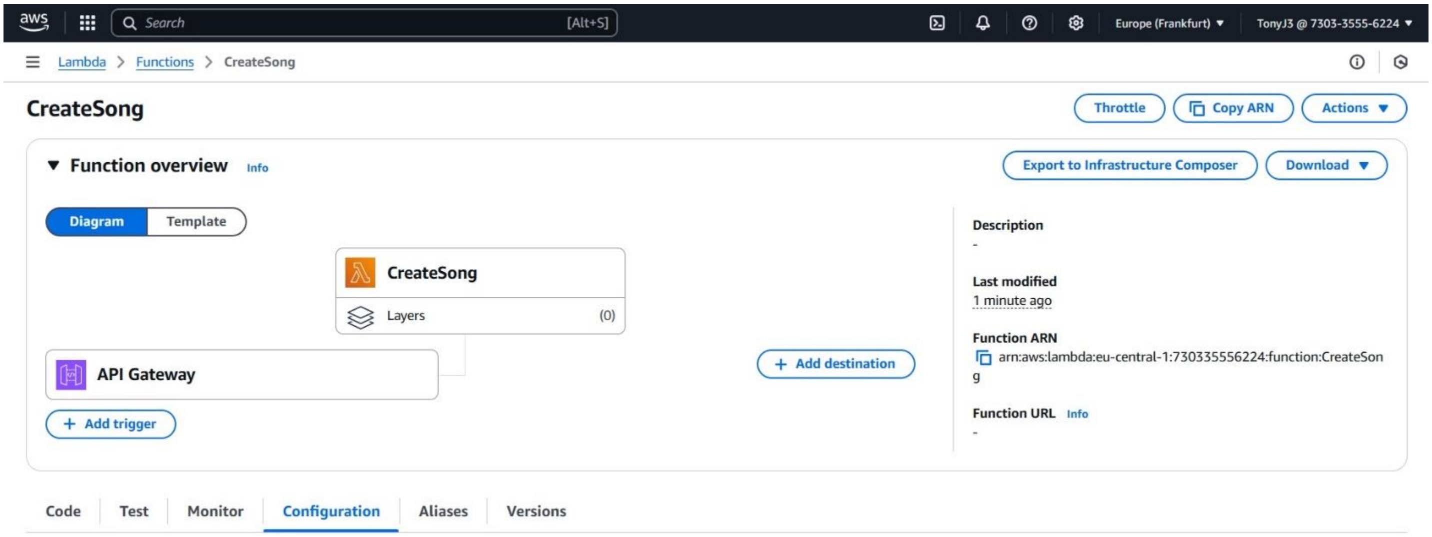
```
deploy:
 runs-on: ubuntu-latest
 needs: docker
 steps:
 - name: Check out the code
   uses: actions/checkout@v4

 - name: Configure AWS CLI
   uses: aws-actions/configure-aws-credentials@v3
   with:
      aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
      aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      aws-region: eu-central-1

 - name: Build Lambda Function
   run: |
        export GOOS=linux
        export GOARCH=arm64
        export CGO_ENABLED=0
        go build -o bootstrap ./cmd/song-service/main.go
        zip create-song.zip bootstrap

 - name: Deploy to AWS Lambda
   run: aws lambda update-function-code --function-name CreateSong --zip-file fileb://create-song.zip
```

Here is evidence that the create function is modified and showing that the API works.



song-service.yaml
on: push

✓ build	30s
✓ test	20s
✓ integration-test	1m 27s
✓ end-2-end-test	47s
✓ lint	20s
✓ snyk	42s
✓ docker	46s
✓ deploy	33s



CreateSong

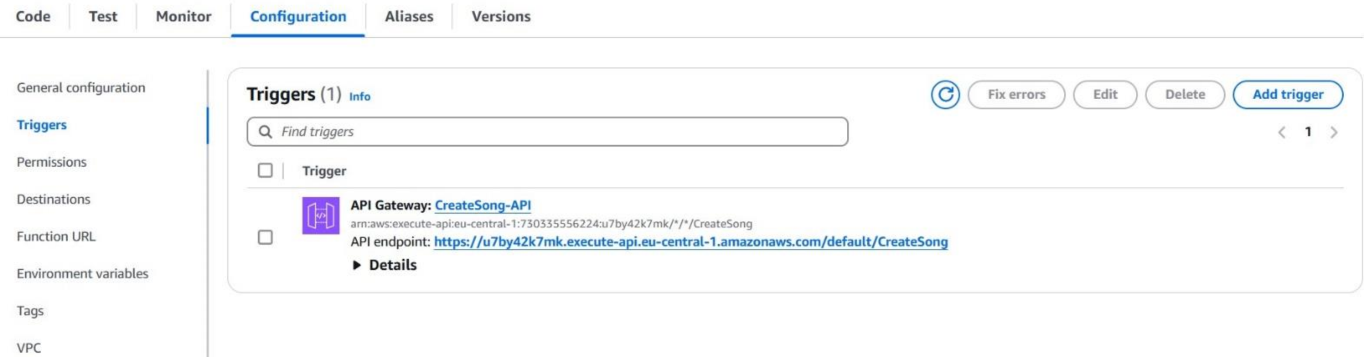
Function overview Info

Description
-

Last modified
1 minute ago

Function ARN
arn:aws:lambda:eu-central-1:730335556224:function:CreateSong

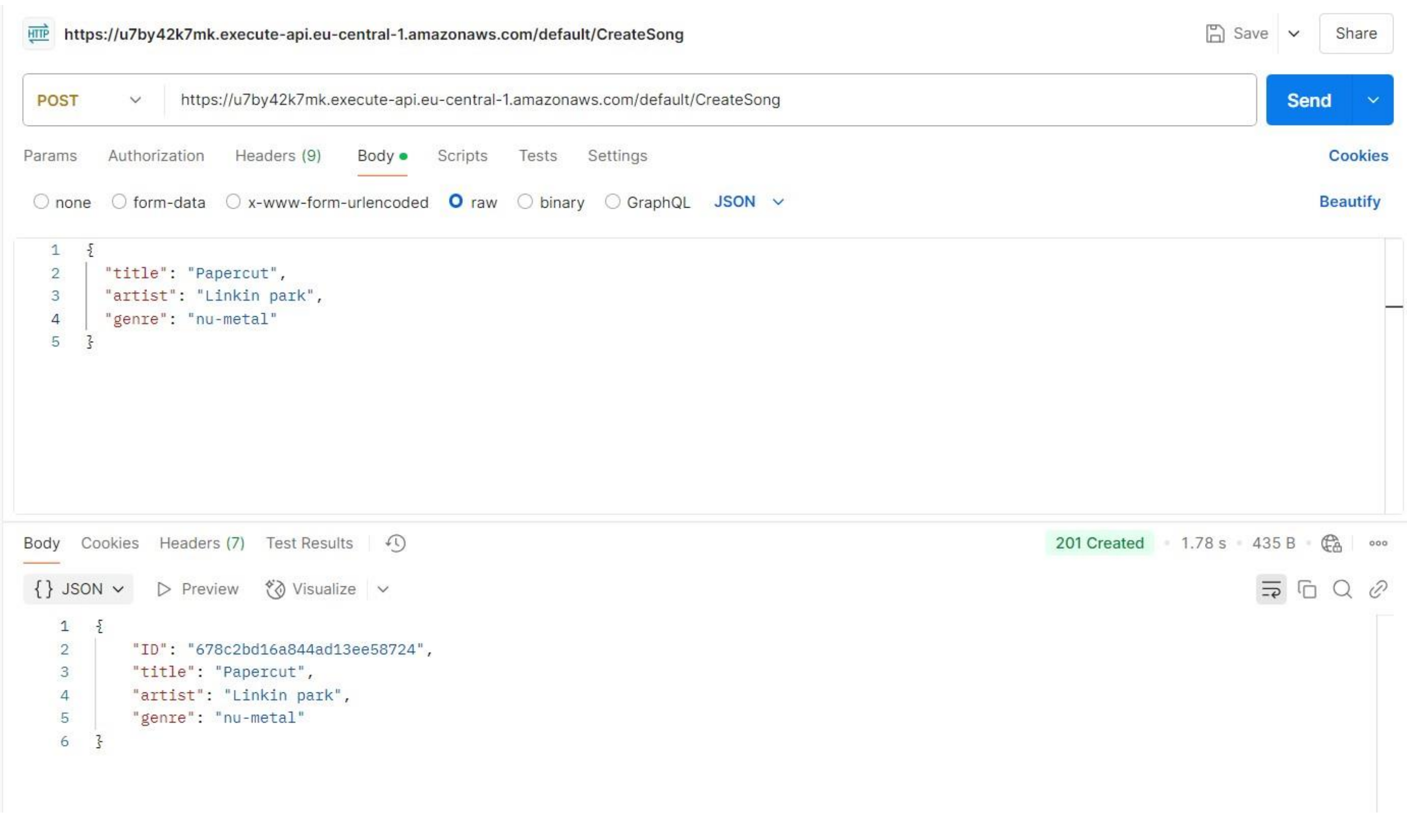
Function URL Info
-



Triggers (1) Info

□ Trigger

□ API Gateway: CreateSong-API
arn:aws:execute-api:eu-central-1:730335556224:u7by42k7mk/*/*/CreateSong
API endpoint: https://u7by42k7mk.execute-api.eu-central-1.amazonaws.com/default/CreateSong
▶ Details



https://u7by42k7mk.execute-api.eu-central-1.amazonaws.com/default/CreateSong

POST https://u7by42k7mk.execute-api.eu-central-1.amazonaws.com/default/CreateSong

```
{
  "title": "Papercut",
  "artist": "Linkin park",
  "genre": "nu-metal"
}
```

201 Created · 1.78 s · 435 B

```
{
    "ID": "678c2bd16a844ad13ee58724",
    "title": "Papercut",
    "artist": "Linkin park",
    "genre": "nu-metal"
}
```

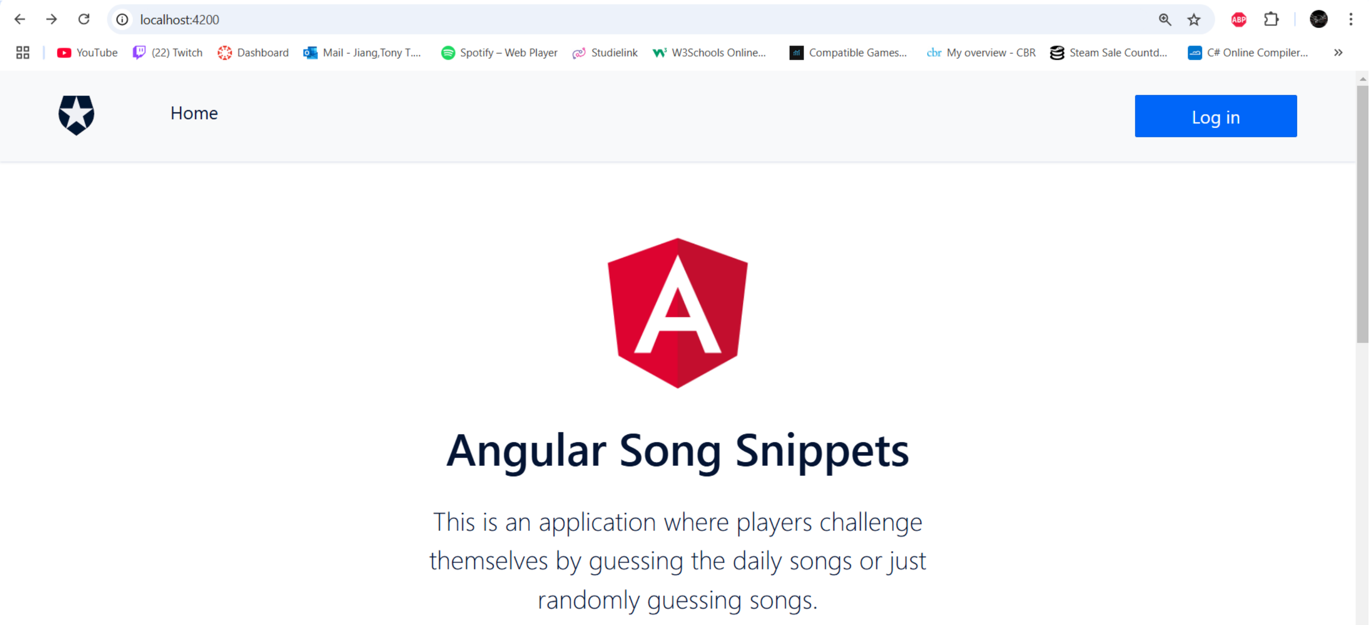
Third-party authentication and authorization tool (individual)

I use Auth0 for third-party authentication and authorization. Auth0 handles all aspects of authentication and authorization, eliminating the need for manual configuration. Instead of building and managing an authentication service myself, I rely on Auth0 to handle these functionalities. Additionally, Auth0 provides a database to store user data, with passwords securely encrypted. It also logs user activity during login events.

Setup Auth0

I use the Single Page Application (SPA) template provided by Auth0. It offers a frontend template that includes login and sign-up functionality. To use it, you simply need to configure the domain, client ID, and audience in the auth_config.json file. For the frontend, I use Angular. You can find the domain, client ID, and audience in your Auth0 account. The audience specifies the API that the frontend calls to authenticate the user and determine whether the user has authorization to access certain resources based on their role.

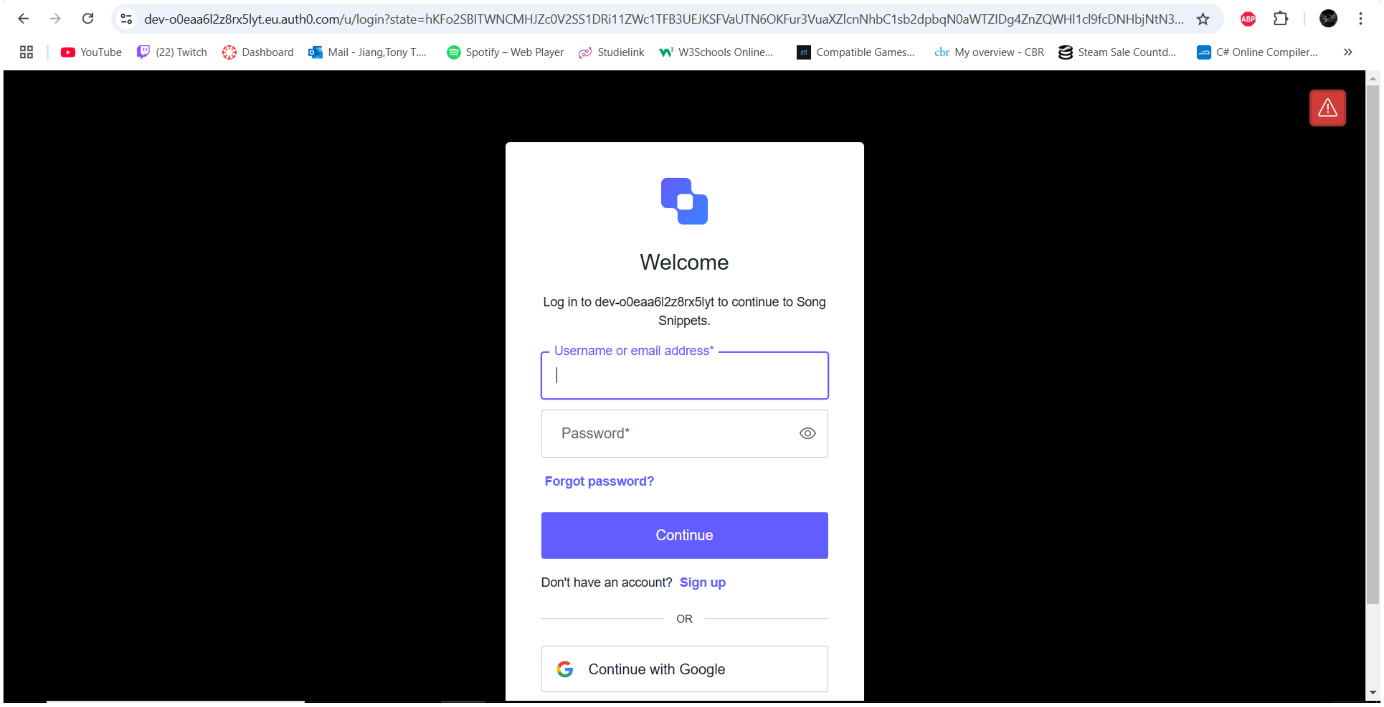
Once everything is configured, you can run the application using the ng serve command to start Angular. The frontend will be available at http://localhost:4200. My frontend is slightly modified I removed some elements and added some text, but the overall structure remains the same. You are free to edit the frontend as you like; I simply kept mine as is.



Angular Song Snippets

This is an application where players challenge themselves by guessing the daily songs or just randomly guessing songs.

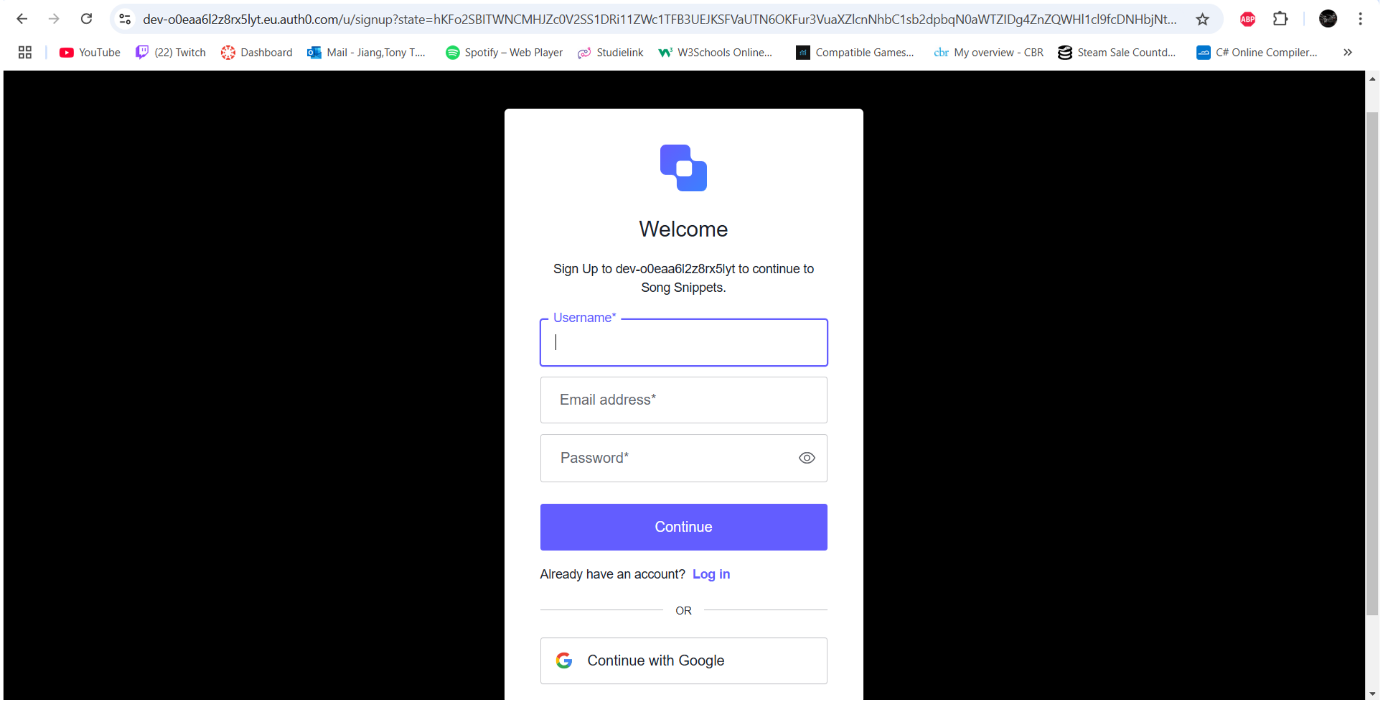
You can also configure which fields are required during sign-up and login. This can be done in Auth0. When you press the login button at the top right, it sends the audience API I mentioned earlier and calls the Auth0 login API. The login frontend and the sign up frontend is configured in Auth0. Additionally, you can allow users to log in using their Google or Facebook accounts; you just need to configure those options in Auth0.



Welcome

Log in to dev-o0eaa6l2z8rx5lyt to continue to Song Snippets.

Don't have an account? Sign up

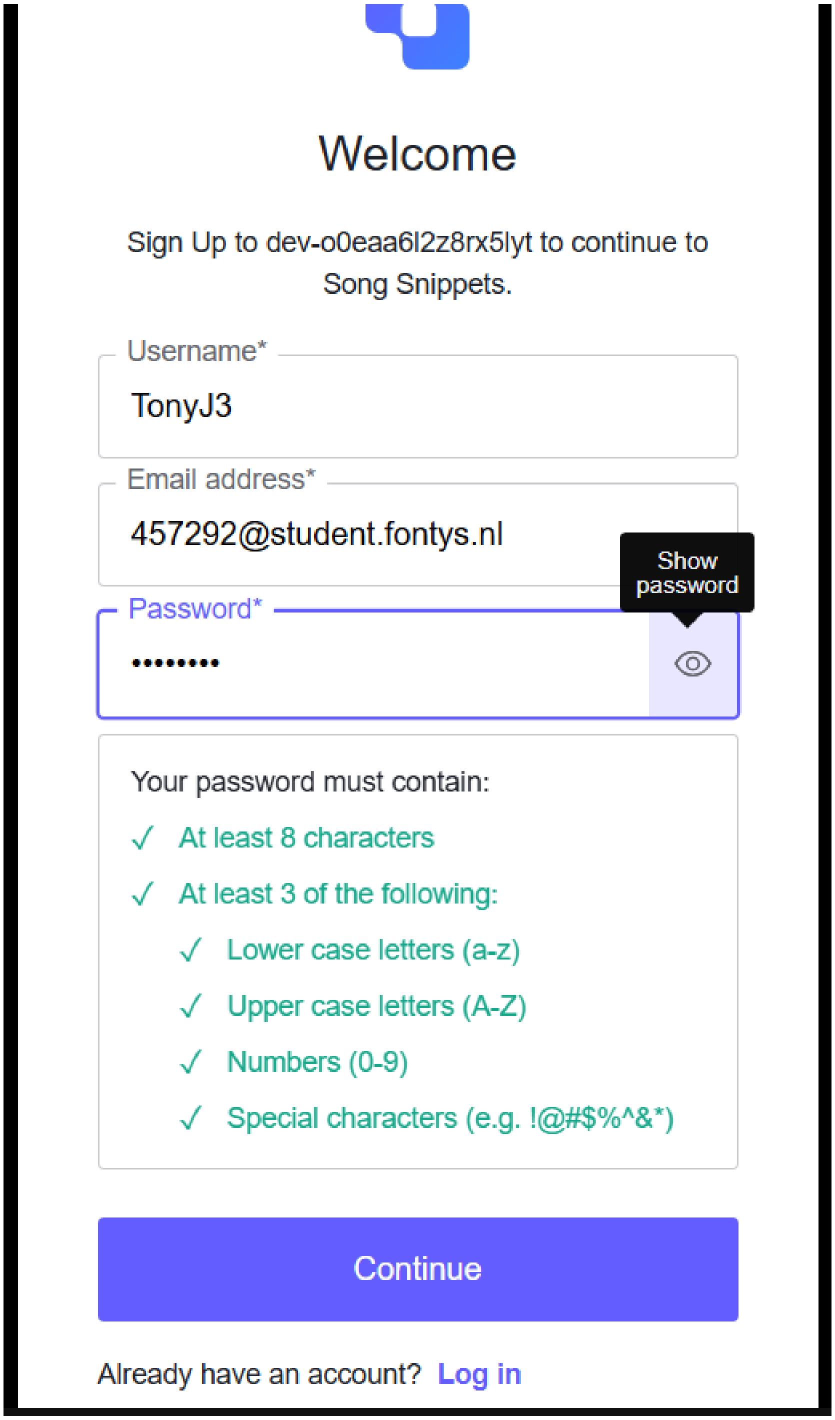


Welcome

Sign Up to dev-o0eaa6l2z8rx5lyt to continue to Song Snippets.

Already have an account? Log in

If you don't have an account, you can click the sign-up link to register. The fields are validated automatically, and I can configure the validation rules in Auth0. When you sign up, you will also receive a verification email to confirm your identity. Additionally, you have a profile page, which you can configure yourself on the frontend.



Welcome

Sign Up to dev-o0eaa6l2z8rx5lyt to continue to Song Snippets.

Username*

TonyJ3

Email address*

457292@student.fontys.nl

Show password

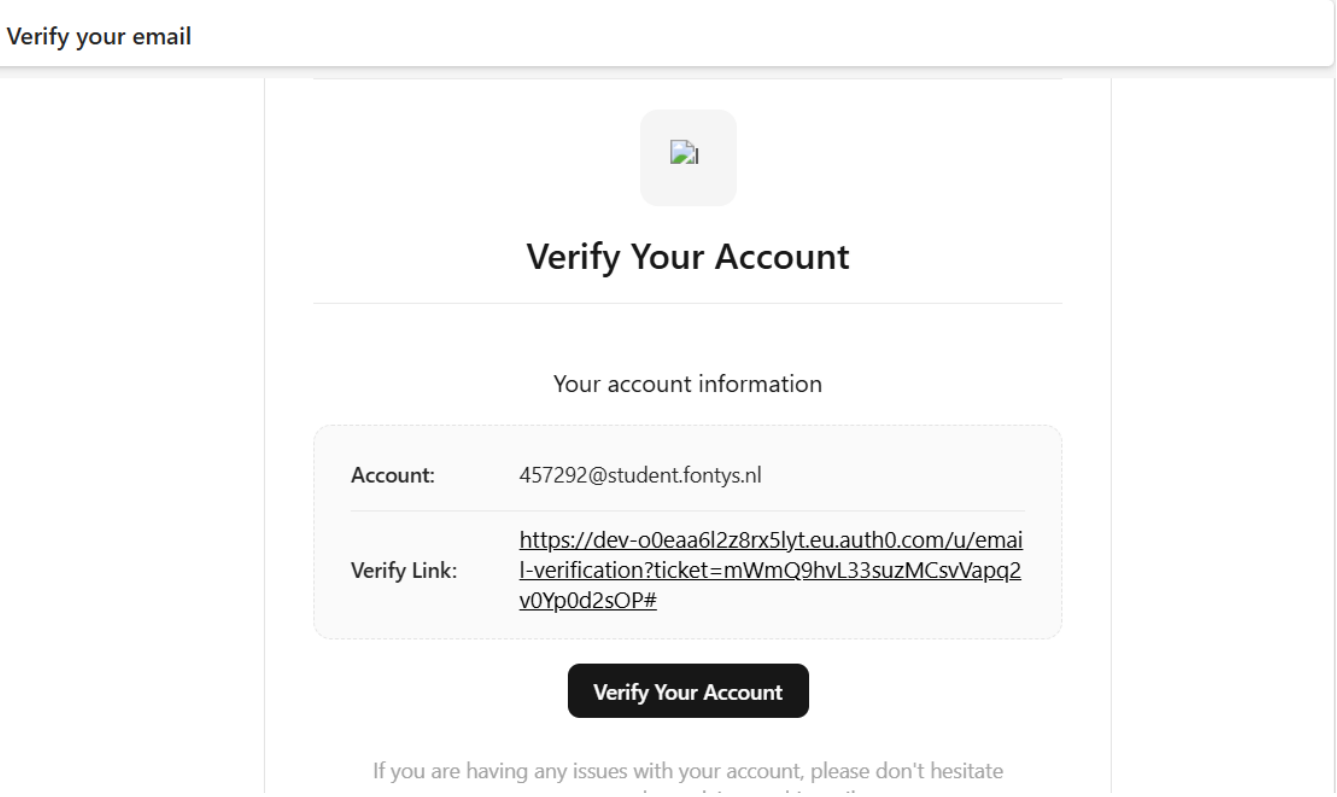
Password*

••••••••

Your password must contain:

- ✓ At least 8 characters
- ✓ At least 3 of the following:
  - ✓ Lower case letters (a-z)
  - ✓ Upper case letters (A-Z)
  - ✓ Numbers (0-9)
  - ✓ Special characters (e.g. !@#$%^&*)

Already have an account? Log in



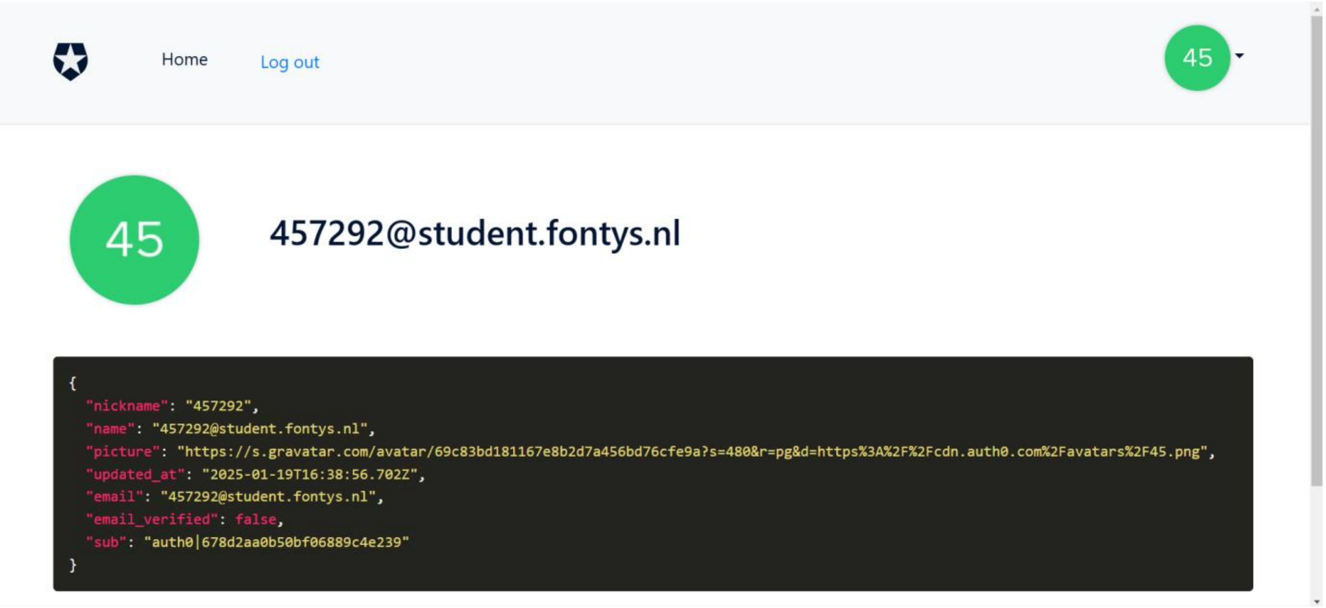
Verify your email

Verify Your Account

Your account information

Account:	457292@student.fontys.nl
Verify Link:	https://dev-o0eaa6l2z8rx5lyt.eu.auth0.com/u/email-verification?ticket=mWmQ9hvL33suzMCsvVapq2v0Yp0d2sOP#

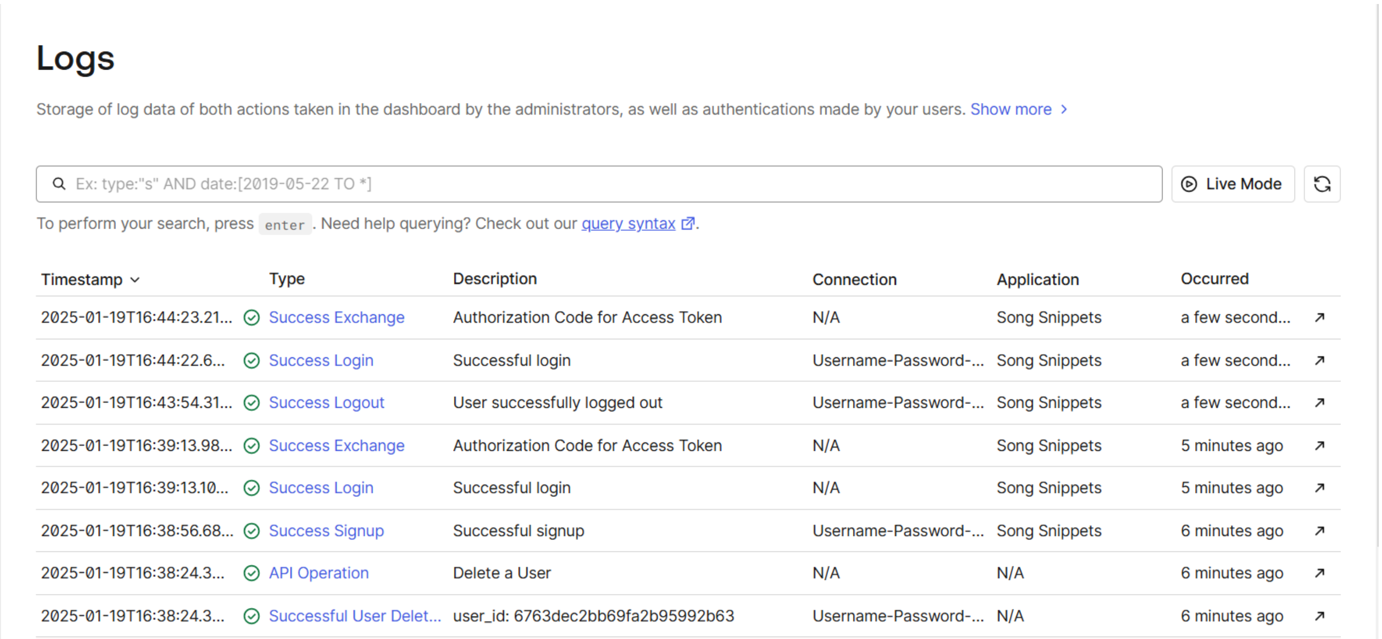
If you are having any issues with your account, please don't hesitate



457292@student.fontys.nl

```
{
  "nickname": "457292",
  "name": "457292@student.fontys.nl",
  "picture": "https://s.gravatar.com/avatar/69c83bd181167e8b2d7a456bd76cfe9a?s=480&r=pg&d=https%3A%2F%2Fcdn.auth0.com%2Favatars%2F45.png",
  "updated_at": "2025-01-19T16:38:56.702Z",
  "email": "457292@student.fontys.nl",
  "email_verified": false,
  "sub": "auth0|678d2aa0b50bf06889c4e239"
}
```

Here are the logs that track user sign-ups and logins. You can view them in your Auth0 account.



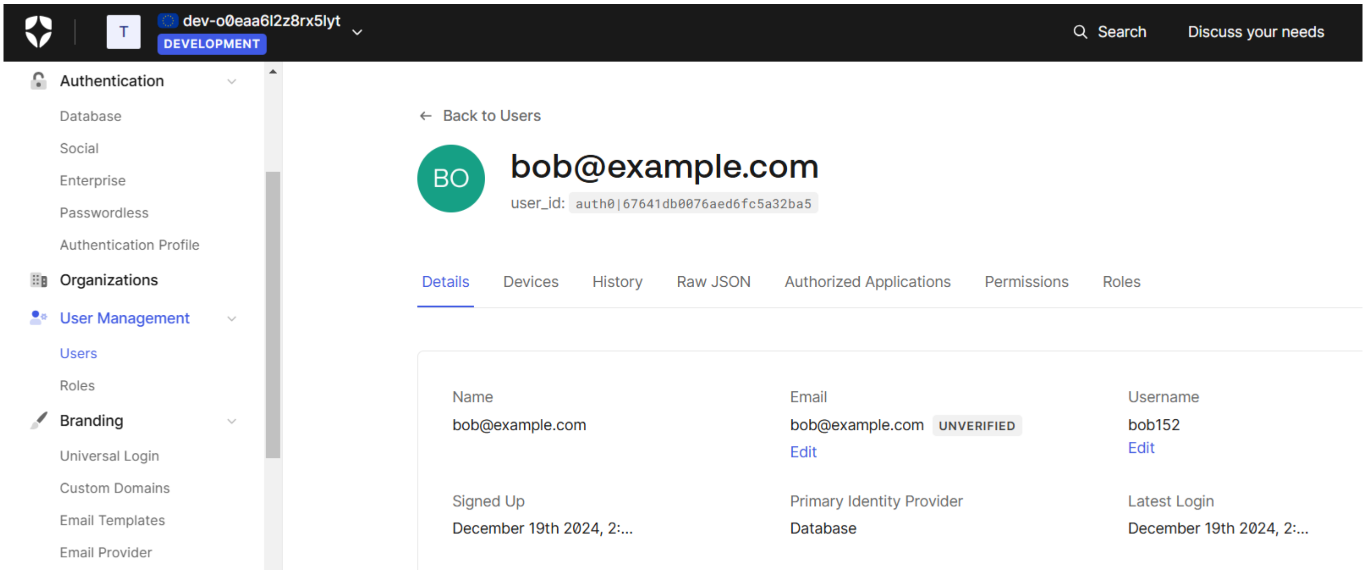
Logs

Storage of log data of both actions taken in the dashboard by the administrators, as well as authentications made by your users. Show more ›

To perform your search, press `enter`. Need help querying? Check out our query syntax.

Timestamp ⌄	Type	Description	Connection	Application	Occurred	
2025-01-19T16:44:23.21...	✓ Success Exchange	Authorization Code for Access Token	N/A	Song Snippets	a few second...	↗
2025-01-19T16:44:22.6...	✓ Success Login	Successful login	Username-Password-...	Song Snippets	a few second...	↗
2025-01-19T16:43:54.31...	✓ Success Logout	User successfully logged out	Username-Password-...	Song Snippets	a few second...	↗
2025-01-19T16:39:13.98...	✓ Success Exchange	Authorization Code for Access Token	N/A	Song Snippets	5 minutes ago	↗
2025-01-19T16:39:13.10...	✓ Success Login	Successful login	N/A	Song Snippets	5 minutes ago	↗
2025-01-19T16:38:56.68...	✓ Success Signup	Successful signup	Username-Password-...	Song Snippets	6 minutes ago	↗
2025-01-19T16:38:24.3...	✓ API Operation	Delete a User	N/A	N/A	6 minutes ago	↗
2025-01-19T16:38:24.3...	✓ Successful User Delet...	user_id: 6763dec2bb69fa2b95992b63	Username-Password-...	N/A	6 minutes ago	↗

You can also make a REST API call or send a message through a message queue from Auth0 to your other services. I implemented this with my user service. When a user signs up, Auth0 makes a REST API call to the user service to create the user. It passes the user's email, username, and auth0_user_id. I include the auth0_user_id because when a user is deleted in Auth0, it triggers a REST API call or message queue to delete the user from the user service using that auth0_user_id. This process is verified by checking the auth0_user_id in both the Auth0 user management and the user service database. For my user service, I use Supabase.

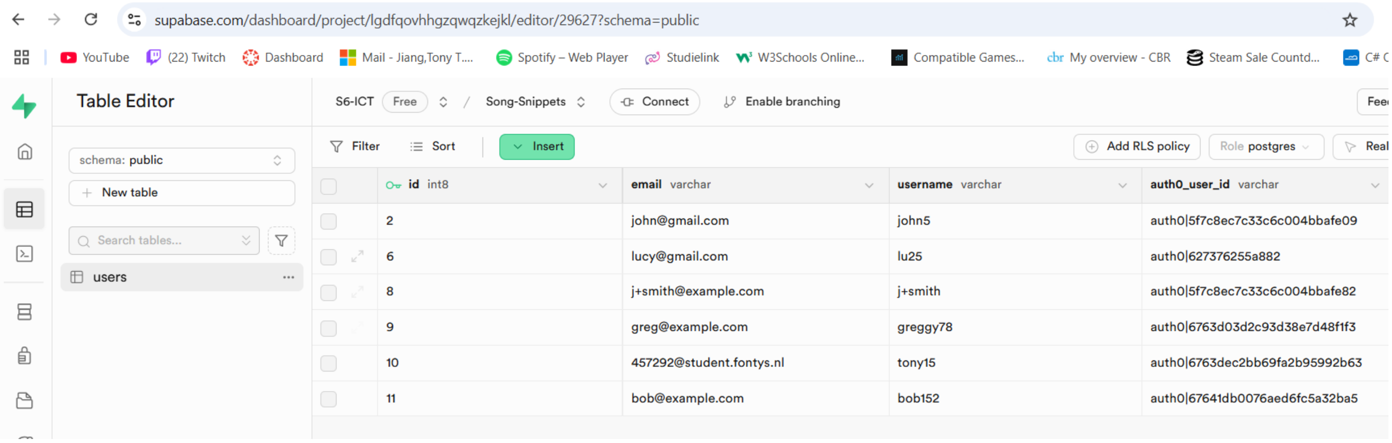


bob@example.com

user_id: `auth0|67641db0076aed6fc5a32ba5`

Name	Email	Username
bob@example.com	bob@example.com UNVERIFIED	bob152

Signed Up	Primary Identity Provider	Latest Login
December 19th 2024, 2:...	Database	December 19th 2024, 2:...



id int8	email varchar	username varchar	auth0_user_id varchar
2	john@gmail.com	john5	auth0|5f7c8ec7c33c6c004bbafe09
6	lucy@gmail.com	lu25	auth0|627376255a882
8	j+smith@example.com	j+smith	auth0|5f7c8ec7c33c6c004bbafe82
9	greg@example.com	greggy78	auth0|6763d03d2c93d38e7d48f1f3
10	457292@student.fontys.nl	tony15	auth0|6763dec2bb69fa2b95992b63
11	bob@example.com	bob152	auth0|67641db0076aed6fc5a32ba5

The initial configuration, such as setting up the template for login and sign-up, takes about 30 minutes to understand, including reading the documentation. Setting up REST API communication with other services also takes around 30 minutes. Auth0 is relatively easy to set up and includes all the necessary authentication, authorization, and user management features, eliminating the need to build an authentication or user service from scratch. However, I created my own user service because I may want to add custom fields in the future, such as a gamer tag, that Auth0 does not support.

Message queue tool (individual)

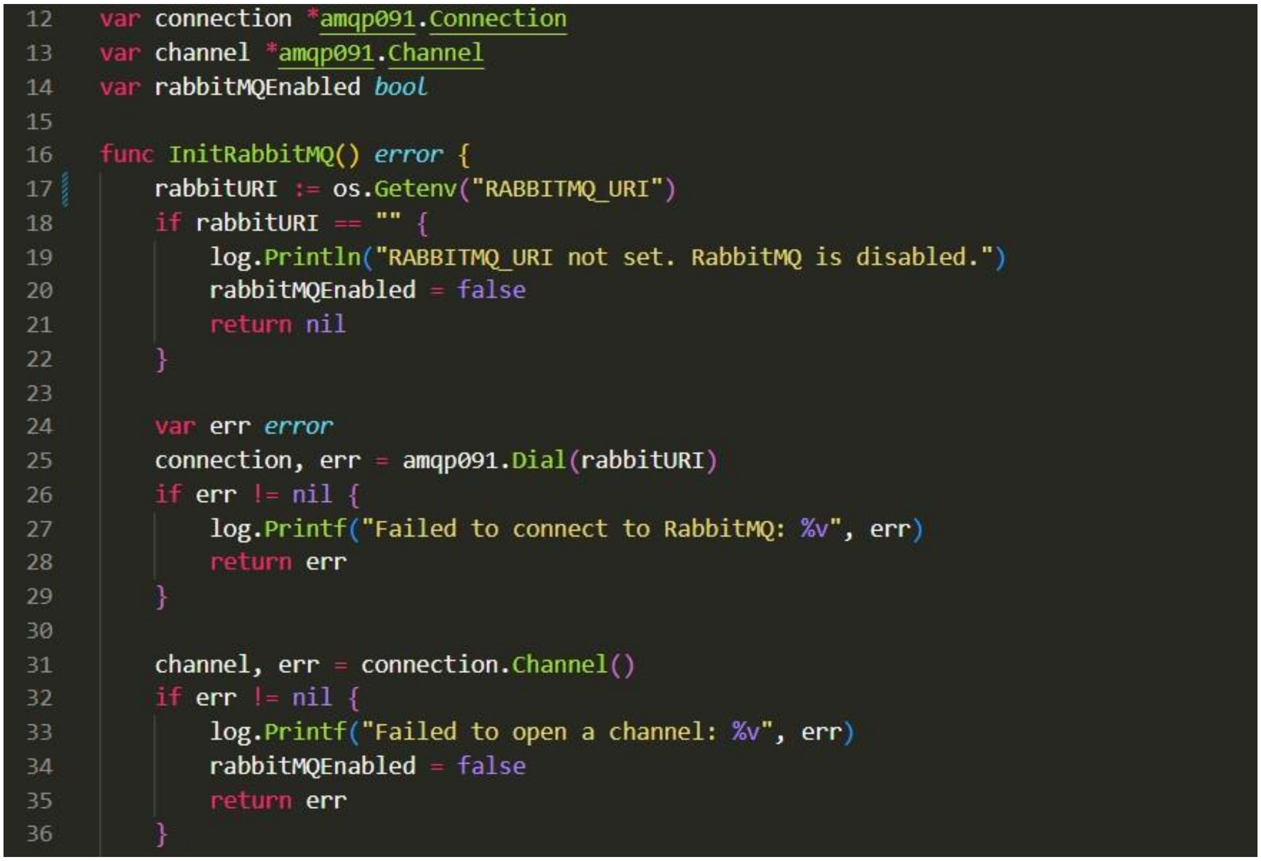
I use RabbitMQ as the message queue for communication between microservices. A message queue facilitates asynchronous communication, allowing services to remain loosely coupled and scale effectively. Currently, I have implemented RabbitMQ for two services: the Song Service and the Song Suggestion Service. Over time, I plan to integrate the message queue with other services as needed.

The Song Service sends song-related events (create, delete, and update) via RabbitMQ to the Song Suggestion Service. This communication ensures that the Song Suggestion Service has the necessary song data to function correctly. Although the Song Suggestion Service only requires a subset of the song data, it is crucial that this data remains consistent with the Song Service. For example, if a song is updated or deleted in the Song Service, the changes must reflect accurately in the Song Suggestion Service to maintain data integrity across both services.

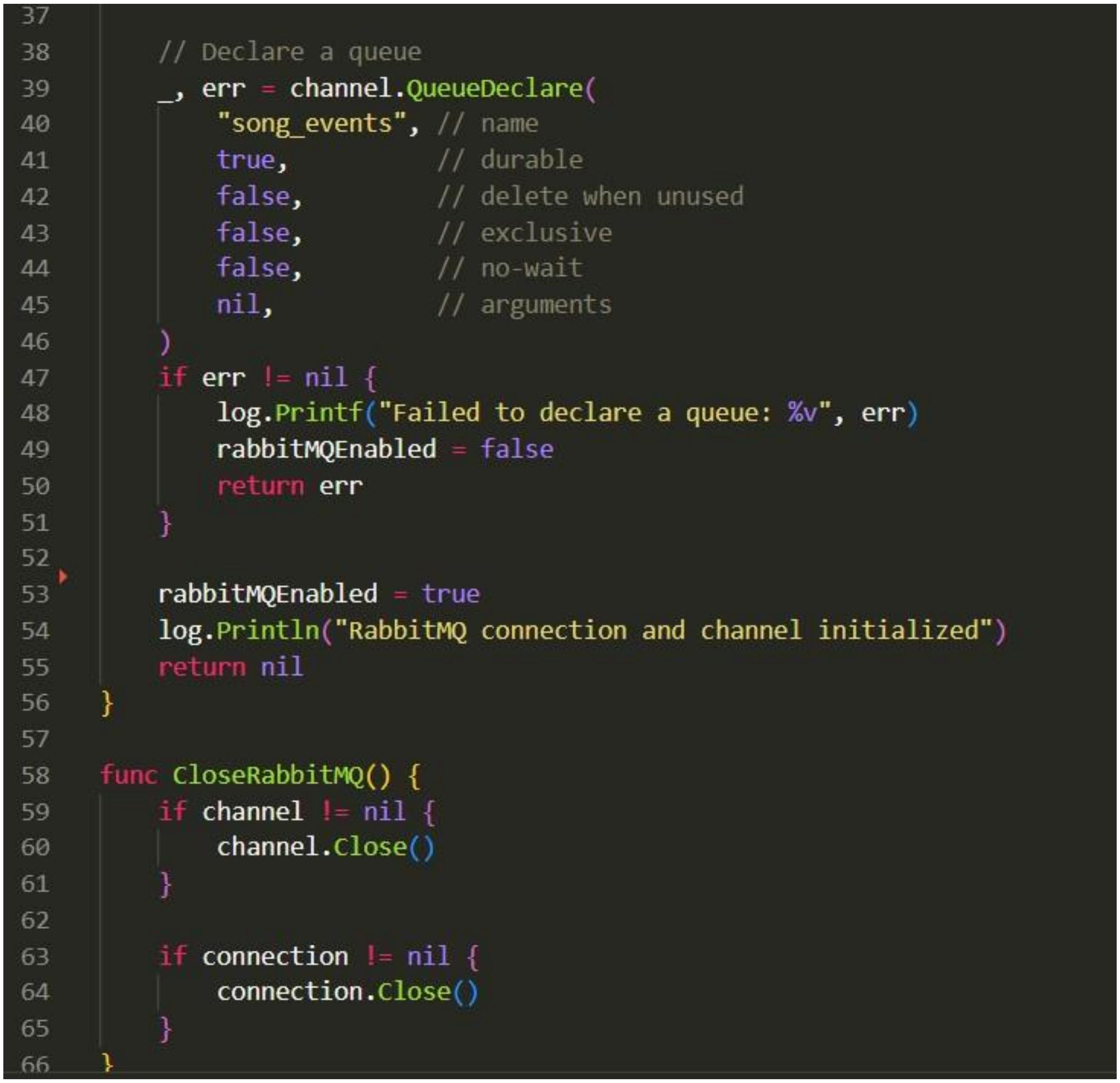
The setup

For the service who is sending the message, we call it producer, that is the song-service. The service that is receiving the message, we call it consumer, which is the song-suggestion service. I’m going to use RabbitMQ but on docker, so I run RabbitMQ container. The container needs to be running for the message to be send. Here is producer file that I setup.

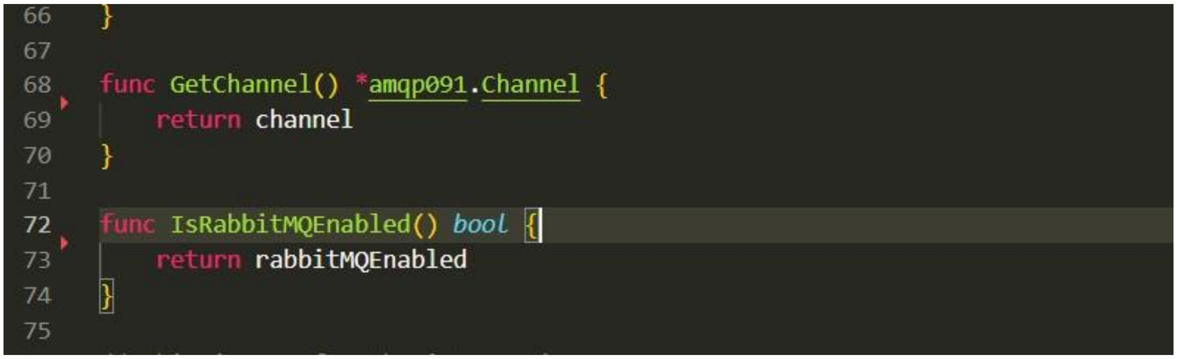
The service that sends messages is called the producer, which in this case is the Song Service. The service that receives messages is called the consumer, which is the Song Suggestion Service. I am using RabbitMQ for messaging, running it in a Docker container. For the messaging system to function, the RabbitMQ container must be running to send and receive messages and it done locally. Below is the producer setup file I have configured and also this is built on Golang.



```go
var connection *amqp091.Connection
var channel *amqp091.Channel
var rabbitMQEnabled bool

func InitRabbitMQ() error {
    rabbitURI := os.Getenv("RABBITMQ_URI")
    if rabbitURI == "" {
        log.Println("RABBITMQ_URI not set. RabbitMQ is disabled.")
        rabbitMQEnabled = false
        return nil
    }

    var err error
    connection, err = amqp091.Dial(rabbitURI)
    if err != nil {
        log.Printf("Failed to connect to RabbitMQ: %v", err)
        return err
    }

    channel, err = connection.Channel()
    if err != nil {
        log.Printf("Failed to open a channel: %v", err)
        rabbitMQEnabled = false
        return err
    }
```

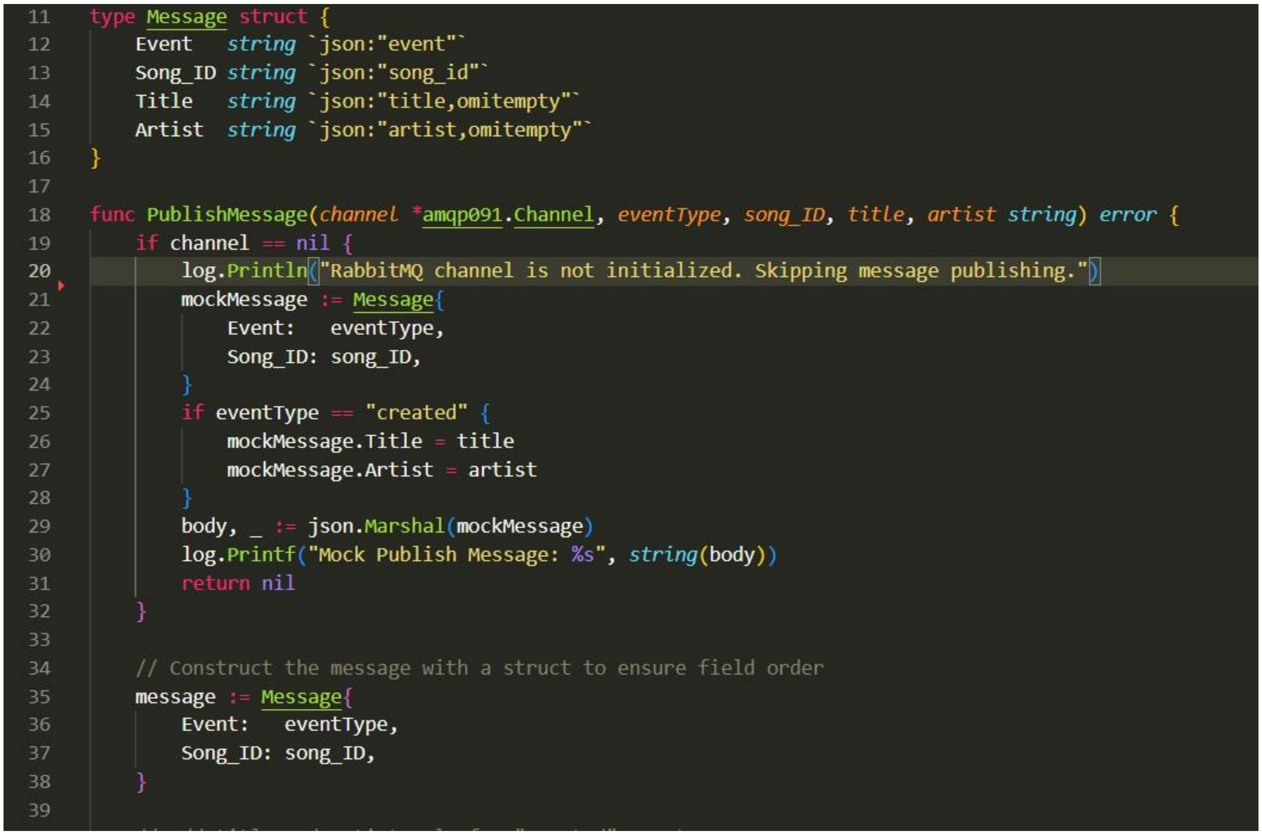


```go

    // Declare a queue
    _, err = channel.QueueDeclare(
        "song_events", // name
        true,          // durable
        false,         // delete when unused
        false,         // exclusive
        false,         // no-wait
        nil,           // arguments
    )
    if err != nil {
        log.Printf("Failed to declare a queue: %v", err)
        rabbitMQEnabled = false
        return err
    }

    rabbitMQEnabled = true
    log.Println("RabbitMQ connection and channel initialized")
    return nil
}

func CloseRabbitMQ() {
    if channel != nil {
        channel.Close()
    }

    if connection != nil {
        connection.Close()
    }
}
```

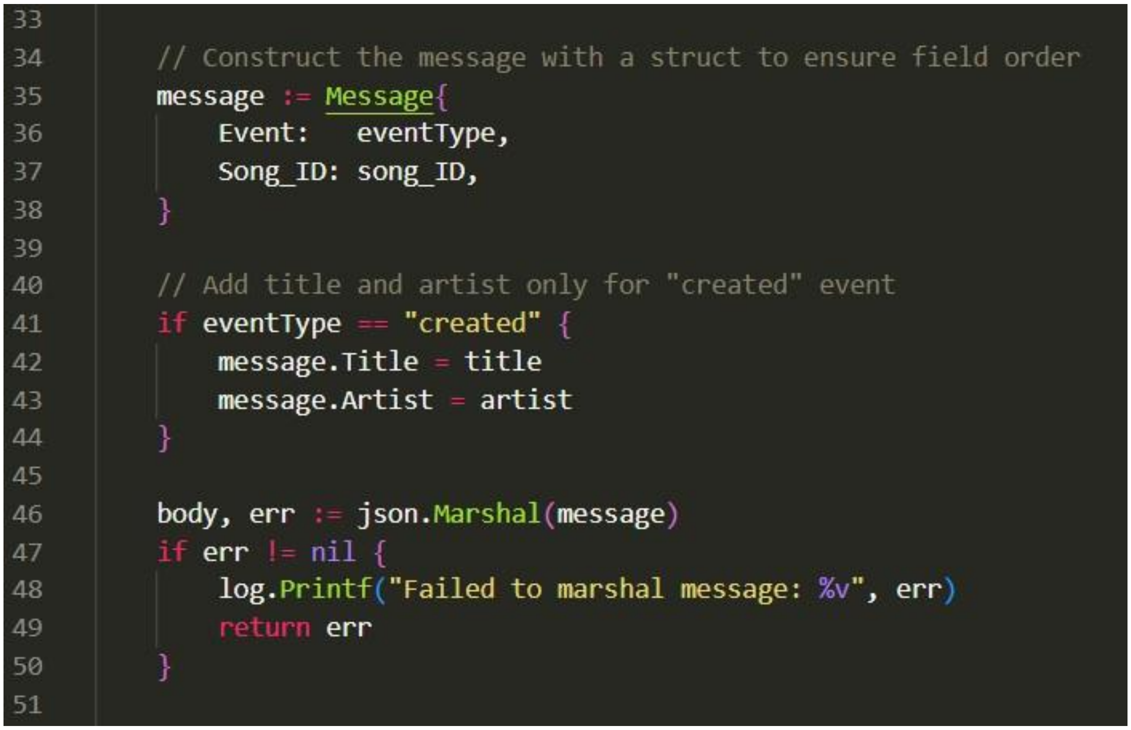


```go
}

func GetChannel() *amqp091.Channel {
    return channel
}

func IsRabbitMQEnabled() bool {
    return rabbitMQEnabled
}

```

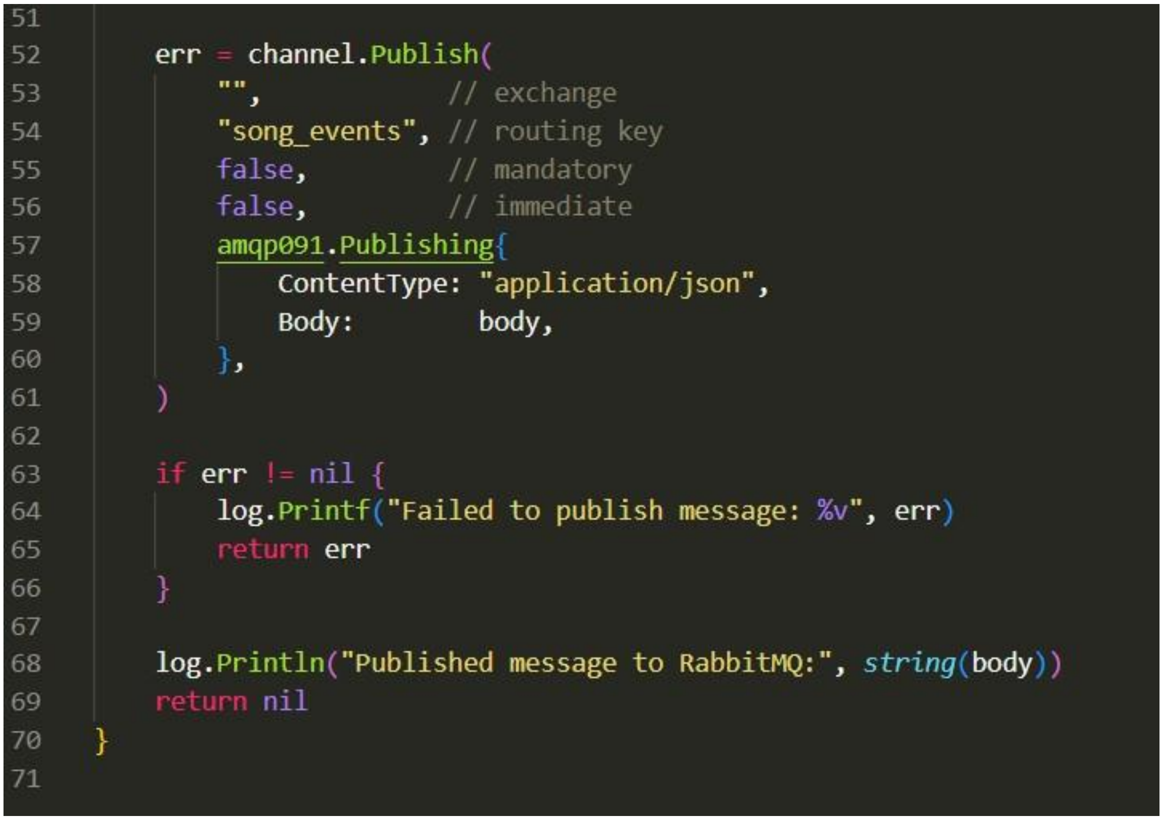
This code establishes a connection between the Song Service and RabbitMQ, using the RABBIT_MQ environment variable to retrieve the RabbitMQ API URL. Additionally, I have implemented a log to notify me if the RABBIT_MQ environment variable is missing. This setup is useful for running the service locally, allowing me to test Song Service requests without requiring RabbitMQ to be active.



```go
type Message struct {
    Event   string `json:"event"`
    Song_ID string `json:"song_id"`
    Title   string `json:"title,omitempty"`
    Artist  string `json:"artist,omitempty"`
}

func PublishMessage(channel *amqp091.Channel, eventType, song_ID, title, artist string) error {
    if channel == nil {
        log.Println("RabbitMQ channel is not initialized. Skipping message publishing.")
        mockMessage := Message{
            Event:   eventType,
            Song_ID: song_ID,
        }
        if eventType == "created" {
            mockMessage.Title = title
            mockMessage.Artist = artist
        }
        body, _ := json.Marshal(mockMessage)
        log.Printf("Mock Publish Message: %s", string(body))
        return nil
    }

    // Construct the message with a struct to ensure field order
    message := Message{
        Event:   eventType,
        Song_ID: song_ID,
    }

```



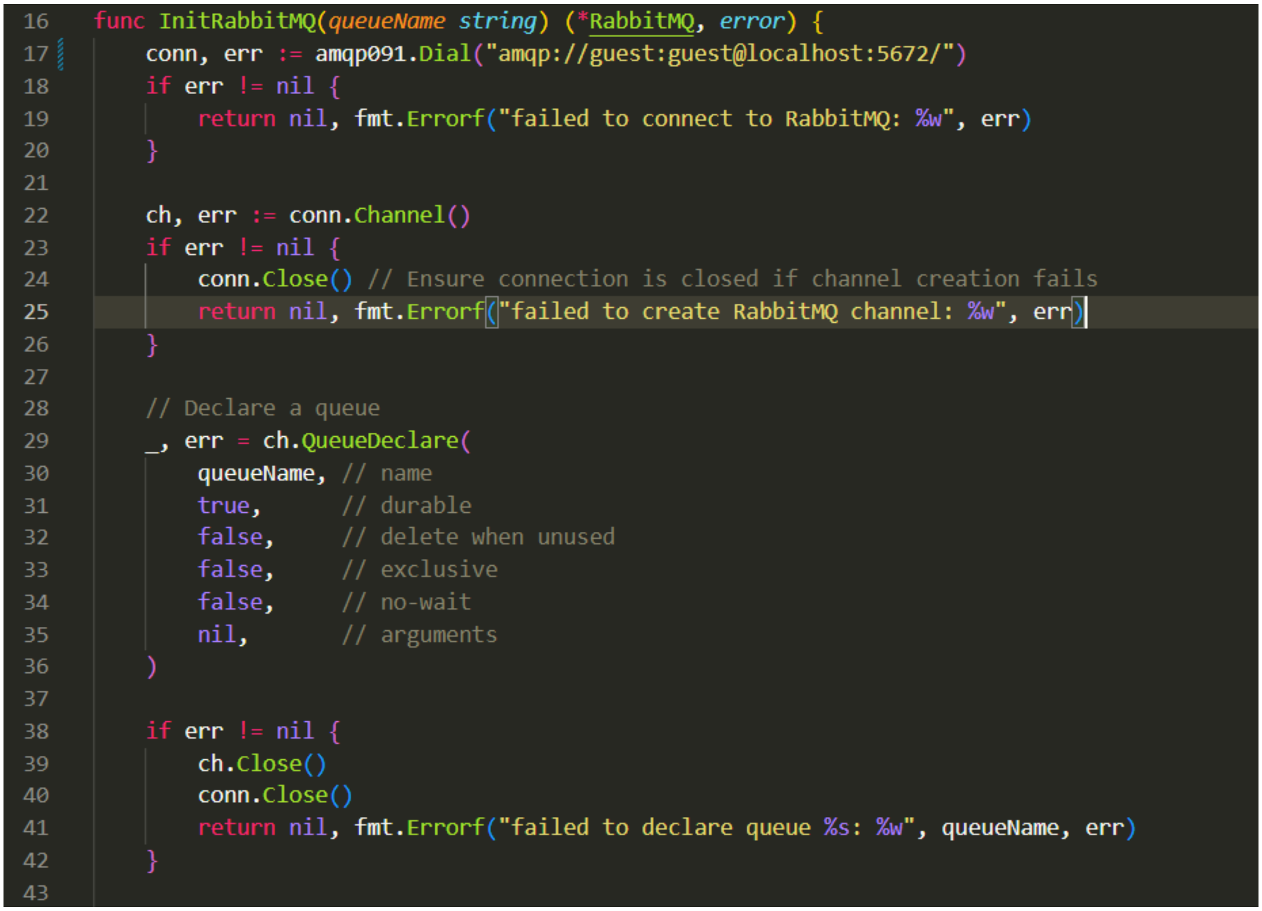
```go

    // Construct the message with a struct to ensure field order
    message := Message{
        Event:   eventType,
        Song_ID: song_ID,
    }

    // Add title and artist only for "created" event
    if eventType == "created" {
        message.Title = title
        message.Artist = artist
    }

    body, err := json.Marshal(message)
    if err != nil {
        log.Printf("Failed to marshal message: %v", err)
        return err
    }

```



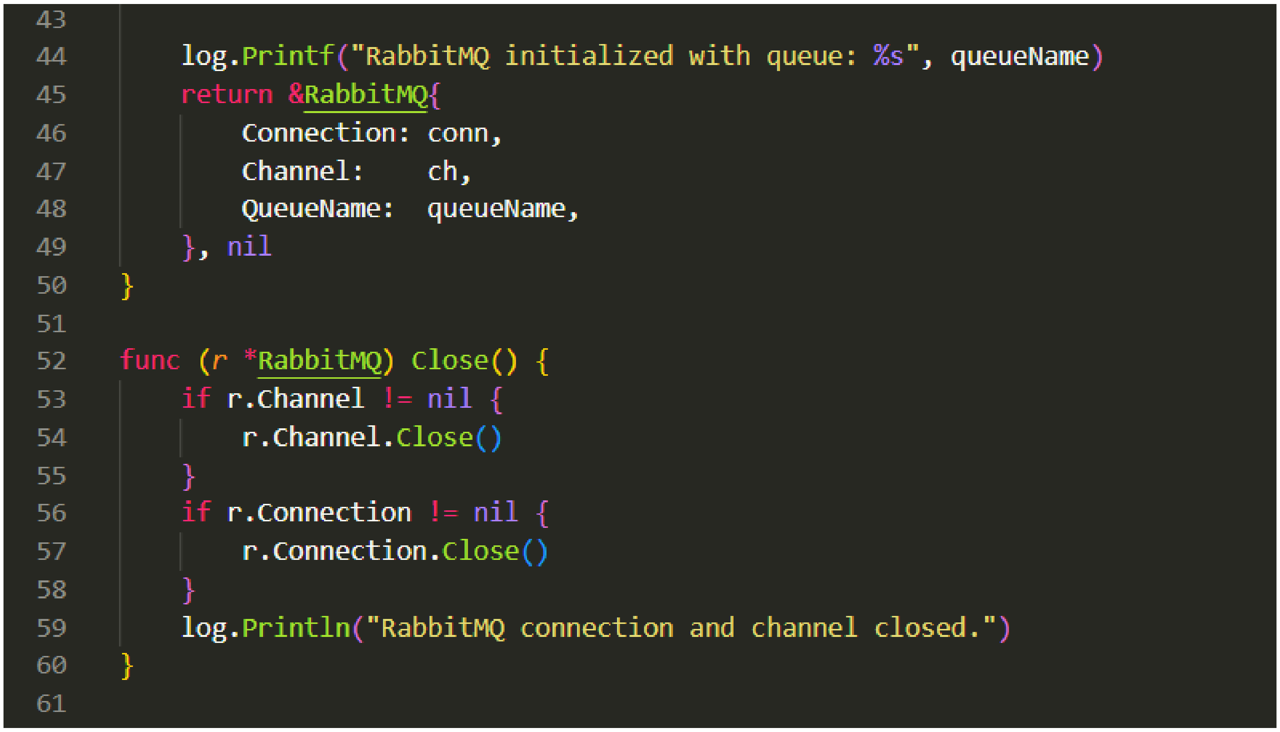
```go

    err = channel.Publish(
        "",            // exchange
        "song_events", // routing key
        false,         // mandatory
        false,         // immediate
        amqp091.Publishing{
            ContentType: "application/json",
            Body:        body,
        },
    )

    if err != nil {
        log.Printf("Failed to publish message: %v", err)
        return err
    }

    log.Println("Published message to RabbitMQ:", string(body))
    return nil
}

```

Here is how publishing works: you call the publish function, which sends a message to RabbitMQ and waits for the consumer to process it. In the publish function, it is configured to handle messages for creating and deleting songs. To specify the type of operation, I simply set the eventType to create to send a message for creating a song, or to delete to send a message for deleting a song. The message for creating, it will send the event, song_id, title and artist. For the delete message it will send the event and song_id.

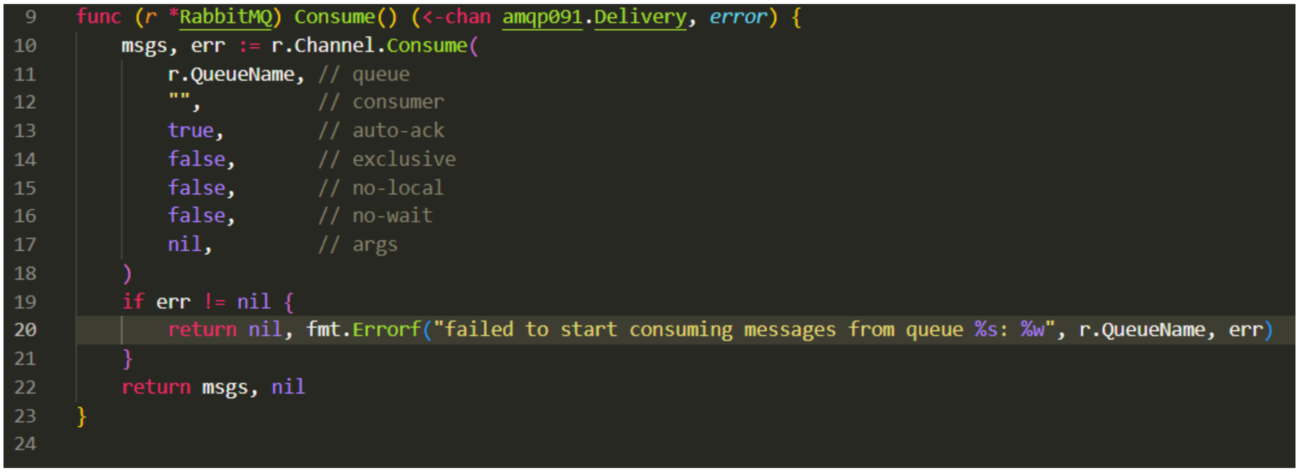
This publish function is called within the handler (known as a controller in Java). Below is the consumer implementation for the Song Suggestion Service.



```go
func InitRabbitMQ(queueName string) (*RabbitMQ, error) {
    conn, err := amqp091.Dial("amqp://guest:guest@localhost:5672/")
    if err != nil {
        return nil, fmt.Errorf("failed to connect to RabbitMQ: %w", err)
    }

    ch, err := conn.Channel()
    if err != nil {
        conn.Close() // Ensure connection is closed if channel creation fails
        return nil, fmt.Errorf("failed to create RabbitMQ channel: %w", err)
    }

    // Declare a queue
    _, err = ch.QueueDeclare(
        queueName, // name
        true,      // durable
        false,     // delete when unused
        false,     // exclusive
        false,     // no-wait
        nil,       // arguments
    )

    if err != nil {
        ch.Close()
        conn.Close()
        return nil, fmt.Errorf("failed to declare queue %s: %w", queueName, err)
    }

```

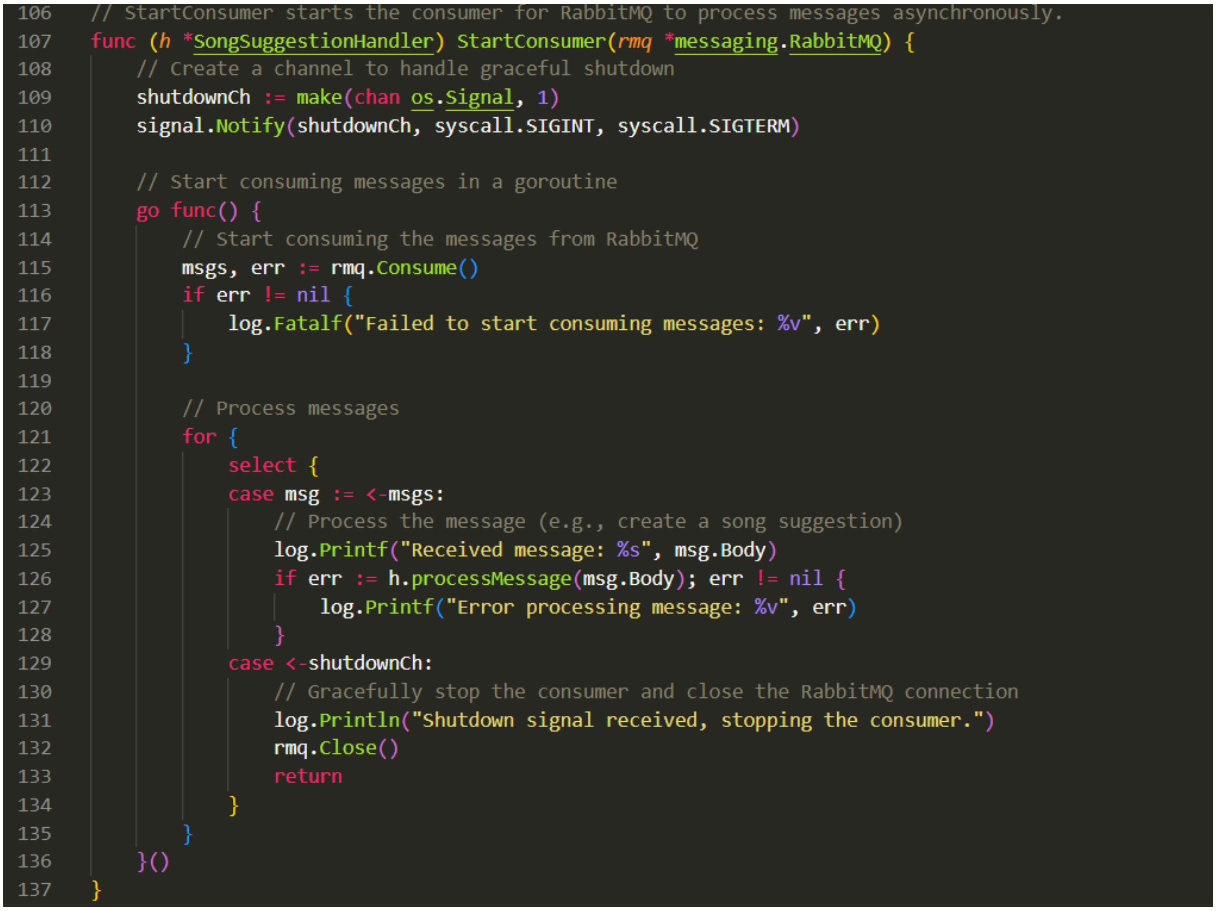


```go

    log.Printf("RabbitMQ initialized with queue: %s", queueName)
    return &RabbitMQ{
        Connection: conn,
        Channel:    ch,
        QueueName:  queueName,
    }, nil
}

func (r *RabbitMQ) Close() {
    if r.Channel != nil {
        r.Channel.Close()
    }
    if r.Connection != nil {
        r.Connection.Close()
    }
    log.Println("RabbitMQ connection and channel closed.")
}

```

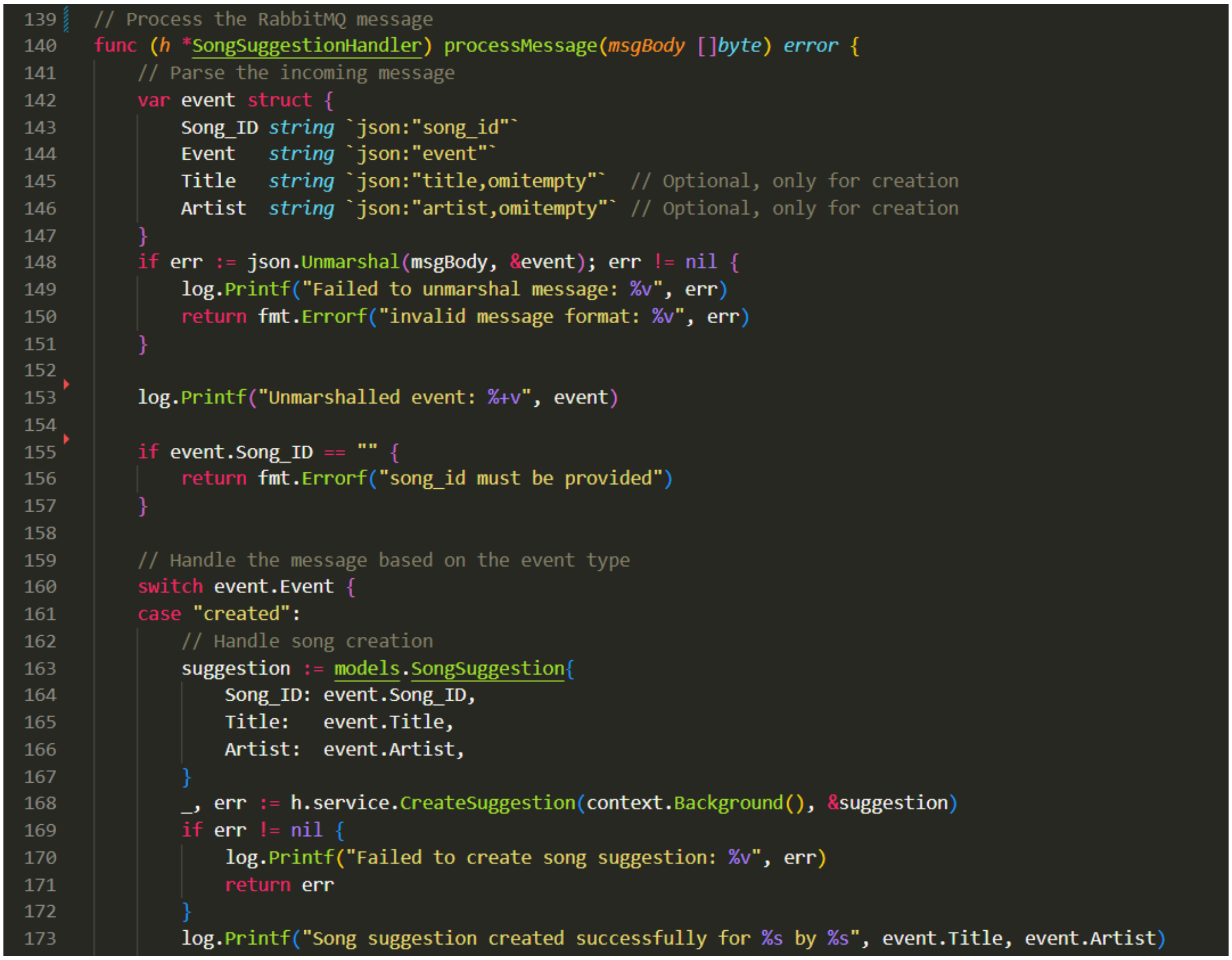


```go
func (r *RabbitMQ) Consume() (<-chan amqp091.Delivery, error) {
    msgs, err := r.Channel.Consume(
        r.QueueName, // queue
        "",          // consumer
        true,        // auto-ack
        false,       // exclusive
        false,       // no-local
        false,       // no-wait
        nil,         // args
    )
    if err != nil {
        return nil, fmt.Errorf("failed to start consuming messages from queue %s: %w", r.QueueName, err)
    }
    return msgs, nil
}

```

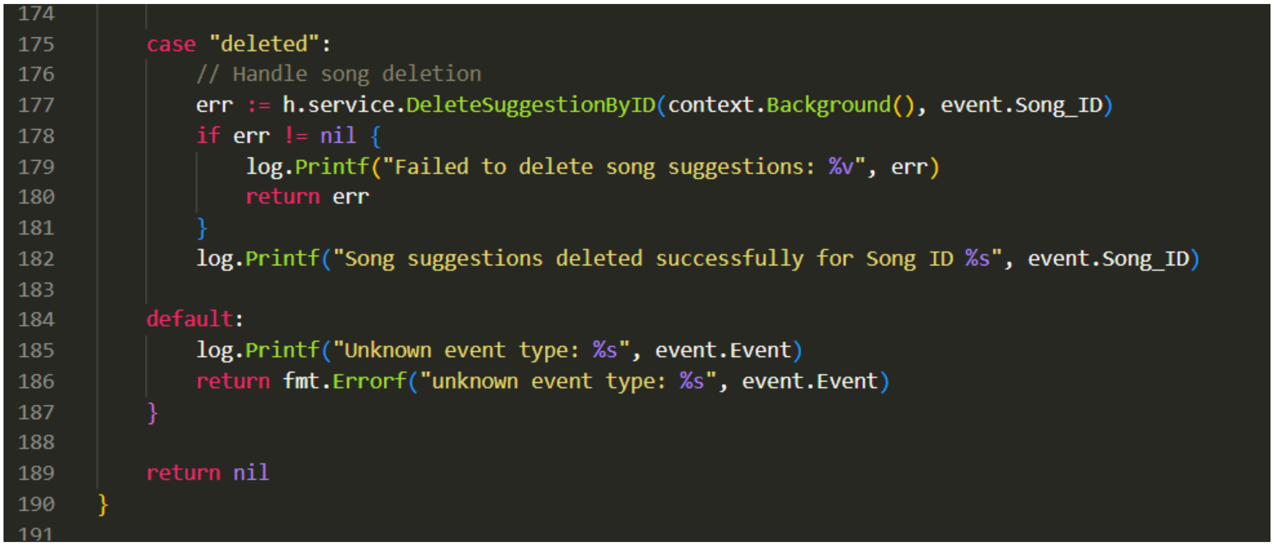
For the consumer, the connection to RabbitMQ is the same as for the producer; the same code is used to establish the connection. I have also created a consume function to start processing messages from the RabbitMQ queue.



```go
// StartConsumer starts the consumer for RabbitMQ to process messages asynchronously.
func (h *SongSuggestionHandler) StartConsumer(rmq *messaging.RabbitMQ) {
    // Create a channel to handle graceful shutdown
    shutdownCh := make(chan os.Signal, 1)
    signal.Notify(shutdownCh, syscall.SIGINT, syscall.SIGTERM)

    // Start consuming messages in a goroutine
    go func() {
        // Start consuming the messages from RabbitMQ
        msgs, err := rmq.Consume()
        if err != nil {
            log.Fatalf("Failed to start consuming messages: %v", err)
        }

        // Process messages
        for {
            select {
            case msg := <-msgs:
                // Process the message (e.g., create a song suggestion)
                log.Printf("Received message: %s", msg.Body)
                if err := h.processMessage(msg.Body); err != nil {
                    log.Printf("Error processing message: %v", err)
                }
            case <-shutdownCh:
                // Gracefully stop the consumer and close the RabbitMQ connection
                log.Println("Shutdown signal received, stopping the consumer.")
                rmq.Close()
                return
            }
        }
    }()
}
```



```go
// Process the RabbitMQ message
func (h *SongSuggestionHandler) processMessage(msgBody []byte) error {
    // Parse the incoming message
    var event struct {
        Song_ID string `json:"song_id"`
        Event   string `json:"event"`
        Title   string `json:"title,omitempty"`  // Optional, only for creation
        Artist  string `json:"artist,omitempty"` // Optional, only for creation
    }
    if err := json.Unmarshal(msgBody, &event); err != nil {
        log.Printf("Failed to unmarshal message: %v", err)
        return fmt.Errorf("invalid message format: %v", err)
    }

    log.Printf("Unmarshalled event: %+v", event)

    if event.Song_ID == "" {
        return fmt.Errorf("song_id must be provided")
    }

    // Handle the message based on the event type
    switch event.Event {
    case "created":
        // Handle song creation
        suggestion := models.SongSuggestion{
            Song_ID: event.Song_ID,
            Title:   event.Title,
            Artist:  event.Artist,
        }
        _, err := h.service.CreateSuggestion(context.Background(), &suggestion)
        if err != nil {
            log.Printf("Failed to create song suggestion: %v", err)
            return err
        }
        log.Printf("Song suggestion created successfully for %s by %s", event.Title, event.Artist)
```



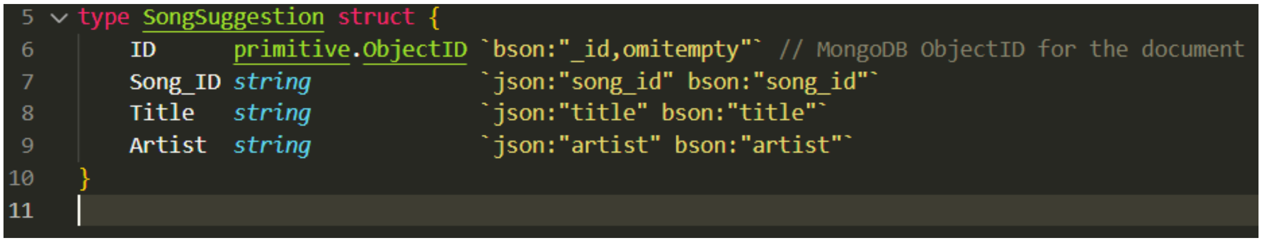
```go

    case "deleted":
        // Handle song deletion
        err := h.service.DeleteSuggestionByID(context.Background(), event.Song_ID)
        if err != nil {
            log.Printf("Failed to delete song suggestions: %v", err)
            return err
        }
        log.Printf("Song suggestions deleted successfully for Song ID %s", event.Song_ID)

    default:
        log.Printf("Unknown event type: %s", event.Event)
        return fmt.Errorf("unknown event type: %s", event.Event)
    }

    return nil
}

```

In the handler (or controller in Java), I created a function called startConsumer, to start consuming messages using the consume method. The messages are processed by a function called processMessage. The processMessage function determines the action based on the event string: if the event is created, it creates a new song; if the event is delete, it deletes the song.

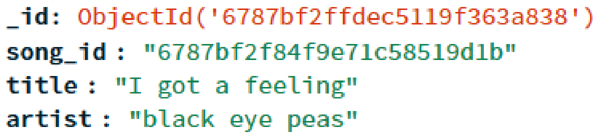
For a create event, the Song Suggestion Service consumes the following data: event, song_id, title, and artist. For a delete event, it consumes the event and song_id. The song_id corresponds to the ID from the Song Service.

The Song Suggestion Service has its own unique ID for songs. I designed it this way to avoid tight coupling between the services. Each service maintains its own ID, allowing the Song Suggestion Service to delete a song by referencing the song_id from the Song Service instead of relying on the primary key of the Song Suggestion Service.

Below, you can see the MongoDB model and data structure illustrating this approach.

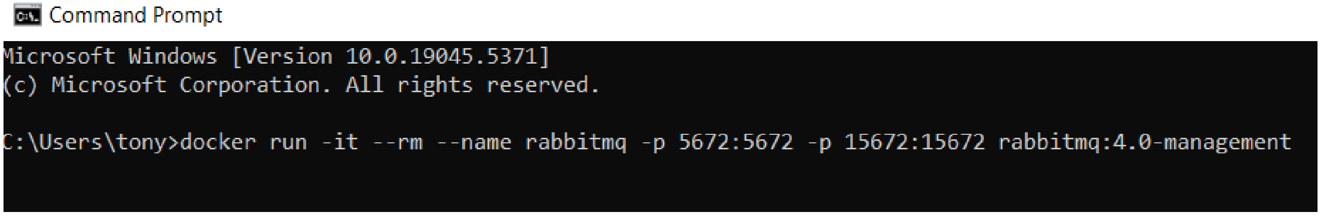


```go
type SongSuggestion struct {
    ID      primitive.ObjectID `bson:"_id,omitempty"` // MongoDB ObjectID for the document
    Song_ID string             `json:"song_id" bson:"song_id"`
    Title   string             `json:"title" bson:"title"`
    Artist  string             `json:"artist" bson:"artist"`
}
```

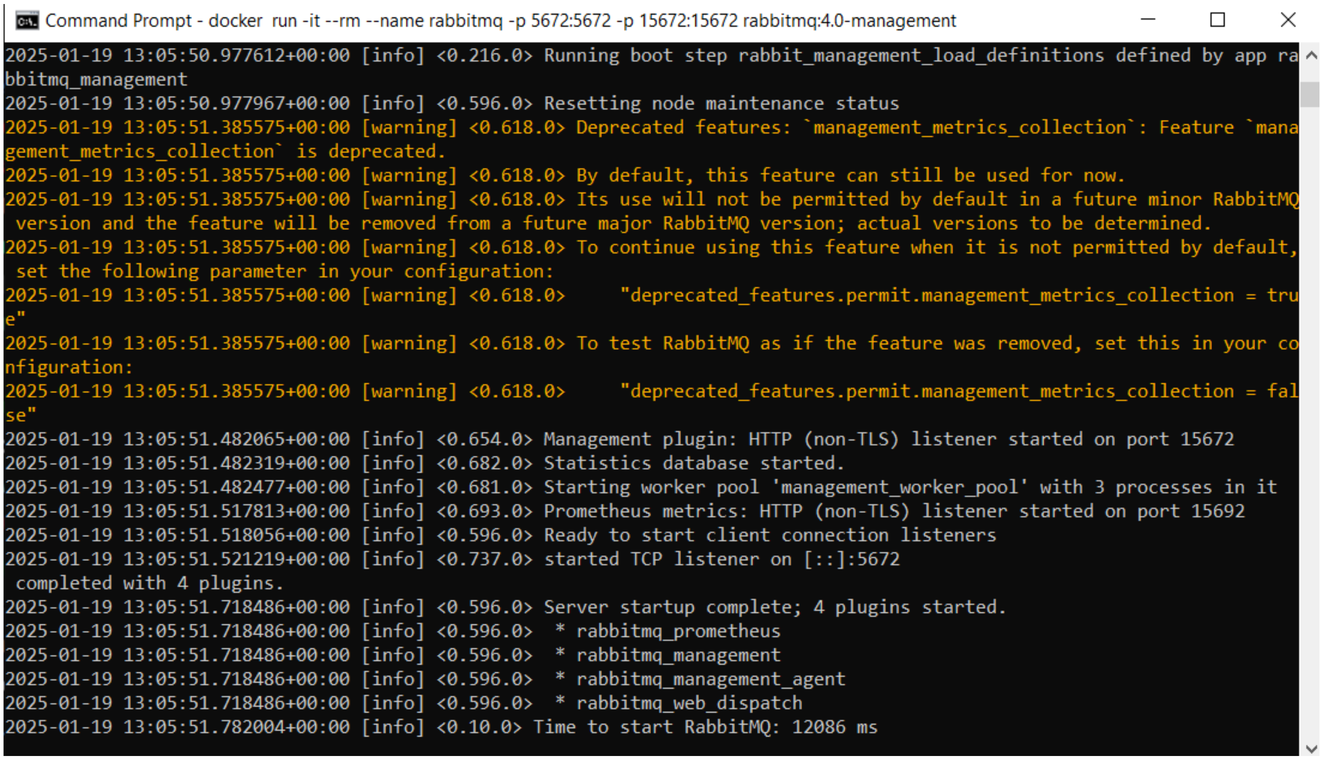


```
_id: ObjectId('6787bf2ffdec5119f363a838')
song_id : "6787bf2f84f9e71c58519d1b"
title : "I got a feeling"
artist : "black eye peas"
```

To enable the Song Service to publish messages and the Song Suggestion Service to consume them, I need to start the RabbitMQ container in Docker. This can be done locally by running the following command in the command prompt: docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4.0-management



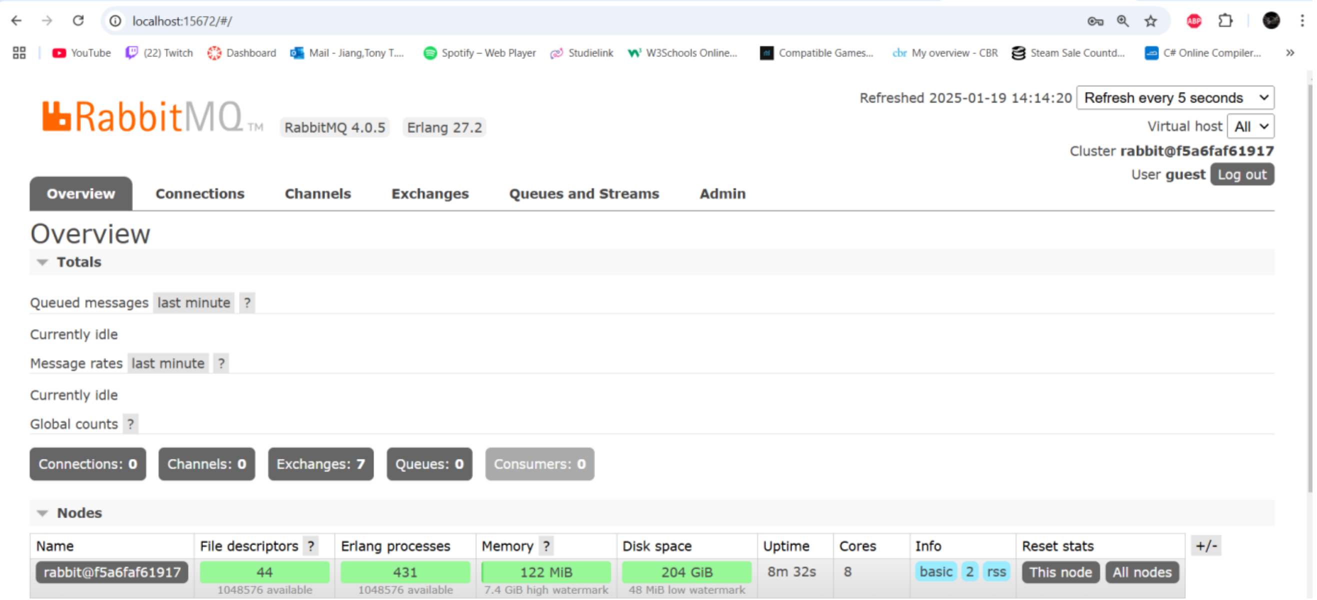
```
Microsoft Windows [Version 10.0.19045.5371]
(c) Microsoft Corporation. All rights reserved.

C:\Users\tony>docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4.0-management
```



Command Prompt - docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4.0-management

```
2025-01-19 13:05:50.977612+00:00 [info] <0.216.0> Running boot step rabbit_management_load_definitions defined by app ra
bbitmq_management
2025-01-19 13:05:50.977967+00:00 [info] <0.596.0> Resetting node maintenance status
2025-01-19 13:05:51.385575+00:00 [warning] <0.618.0> Deprecated features: `management_metrics_collection`: Feature `mana
gement_metrics_collection` is deprecated.
2025-01-19 13:05:51.385575+00:00 [warning] <0.618.0> By default, this feature can still be used for now.
2025-01-19 13:05:51.385575+00:00 [warning] <0.618.0> Its use will not be permitted by default in a future minor RabbitMQ
 version and the feature will be removed from a future major RabbitMQ version; actual versions to be determined.
2025-01-19 13:05:51.385575+00:00 [warning] <0.618.0> To continue using this feature when it is not permitted by default,
 set the following parameter in your configuration:
2025-01-19 13:05:51.385575+00:00 [warning] <0.618.0>     "deprecated_features.permit.management_metrics_collection = tru
e"
2025-01-19 13:05:51.385575+00:00 [warning] <0.618.0> To test RabbitMQ as if the feature was removed, set this in your co
nfiguration:
2025-01-19 13:05:51.385575+00:00 [warning] <0.618.0>     "deprecated_features.permit.management_metrics_collection = fal
se"
2025-01-19 13:05:51.482065+00:00 [info] <0.654.0> Management plugin: HTTP (non-TLS) listener started on port 15672
2025-01-19 13:05:51.482319+00:00 [info] <0.682.0> Statistics database started.
2025-01-19 13:05:51.482477+00:00 [info] <0.681.0> Starting worker pool 'management_worker_pool' with 3 processes in it
2025-01-19 13:05:51.517813+00:00 [info] <0.693.0> Prometheus metrics: HTTP (non-TLS) listener started on port 15692
2025-01-19 13:05:51.518056+00:00 [info] <0.596.0> Ready to start client connection listeners
2025-01-19 13:05:51.521219+00:00 [info] <0.737.0> started TCP listener on [::]:5672
 completed with 4 plugins.
2025-01-19 13:05:51.718486+00:00 [info] <0.596.0> Server startup complete; 4 plugins started.
2025-01-19 13:05:51.718486+00:00 [info] <0.596.0>  * rabbitmq_prometheus
2025-01-19 13:05:51.718486+00:00 [info] <0.596.0>  * rabbitmq_management
2025-01-19 13:05:51.718486+00:00 [info] <0.596.0>  * rabbitmq_management_agent
2025-01-19 13:05:51.718486+00:00 [info] <0.596.0>  * rabbitmq_web_dispatch
2025-01-19 13:05:51.782004+00:00 [info] <0.10.0> Time to start RabbitMQ: 12086 ms
```

Once RabbitMQ is running, you can access the RabbitMQ UI at http://localhost:15672/. The default credentials are guest for both the username and password. After starting both services, they will establish their connections to RabbitMQ. You can verify this in the RabbitMQ UI, where the connections from both services will be visible.



Refreshed 2025-01-19 14:14:20 Refresh every 5 seconds

Virtual host All

Cluster **rabbit@f5a6faf61917**

User **guest**

Overview

Totals

Queued messages last minute ?

Currently idle

Message rates last minute ?

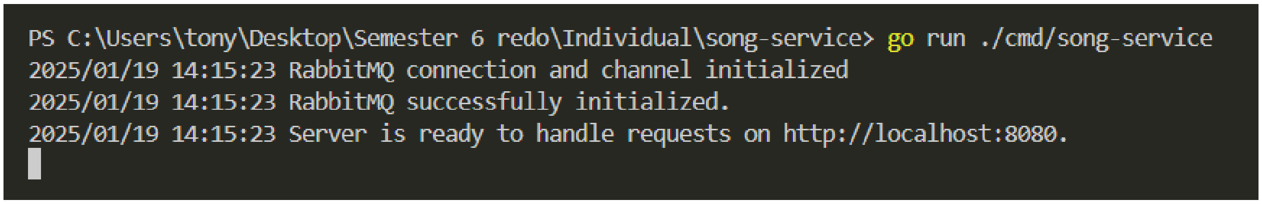
Currently idle

Global counts ?

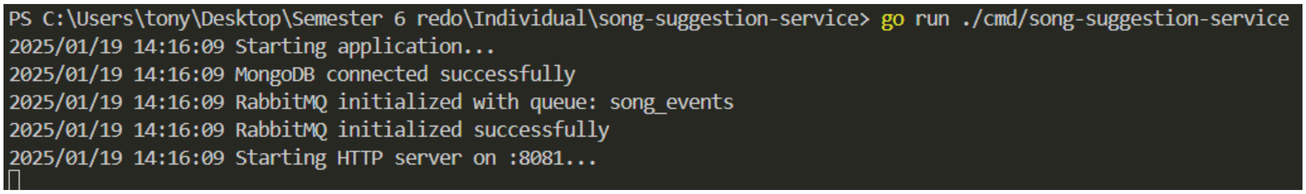
Nodes

Name	File descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Cores	Info	Reset stats
rabbit@f5a6faf61917	44 1048576 available	431 1048576 available	122 MiB 7.4 GiB high watermark	204 GiB 48 MiB low watermark	8m 32s	8	basic 2 rss	This node All nodes

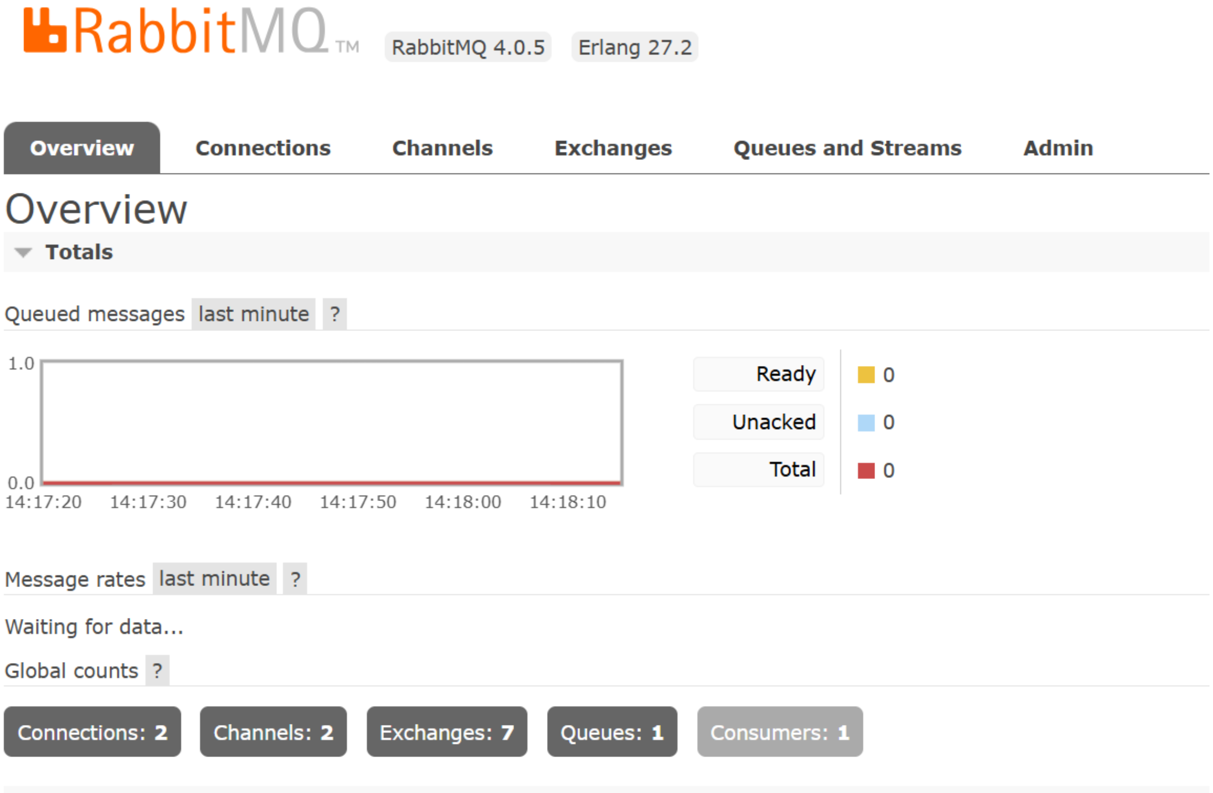
+/-



```
PS C:\Users\tony\Desktop\Semester 6 redo\Individual\song-service> go run ./cmd/song-service
2025/01/19 14:15:23 RabbitMQ connection and channel initialized
2025/01/19 14:15:23 RabbitMQ successfully initialized.
2025/01/19 14:15:23 Server is ready to handle requests on http://localhost:8080.
```



```
PS C:\Users\tony\Desktop\Semester 6 redo\Individual\song-suggestion-service> go run ./cmd/song-suggestion-service
2025/01/19 14:16:09 Starting application...
2025/01/19 14:16:09 MongoDB connected successfully
2025/01/19 14:16:09 RabbitMQ initialized with queue: song_events
2025/01/19 14:16:09 RabbitMQ initialized successfully
2025/01/19 14:16:09 Starting HTTP server on :8081...
```



Overview

Totals

Queued messages last minute ?

Ready 0
Unacked 0
Total 0

Message rates last minute ?

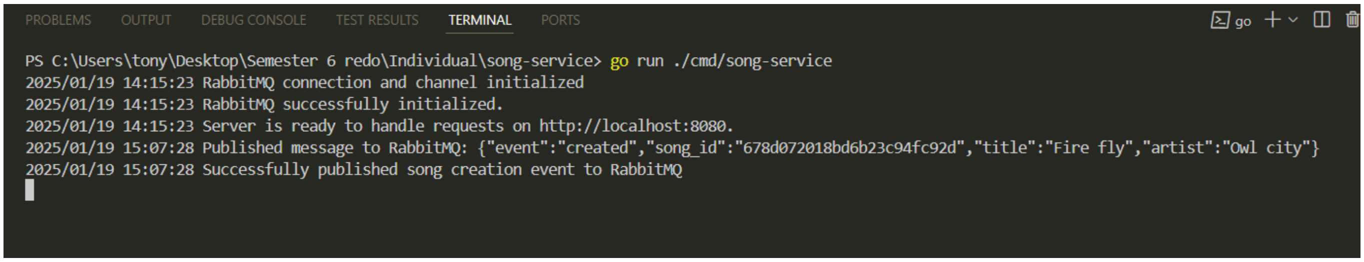
Waiting for data...

Global counts ?

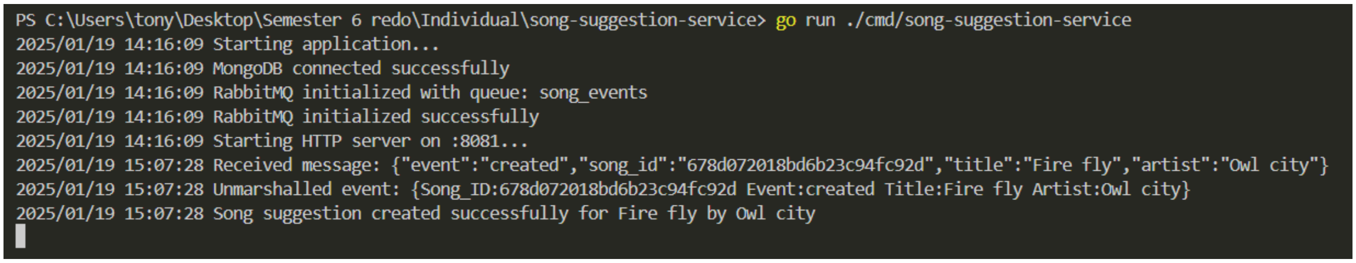
Connections: 2 Channels: 2 Exchanges: 7 Queues: 1 Consumers: 1

After that, I can create a song in the Song Service. The data will be sent as a message to the queue and wait for the Song Suggestion Service to consume it. This process can be verified in several ways:

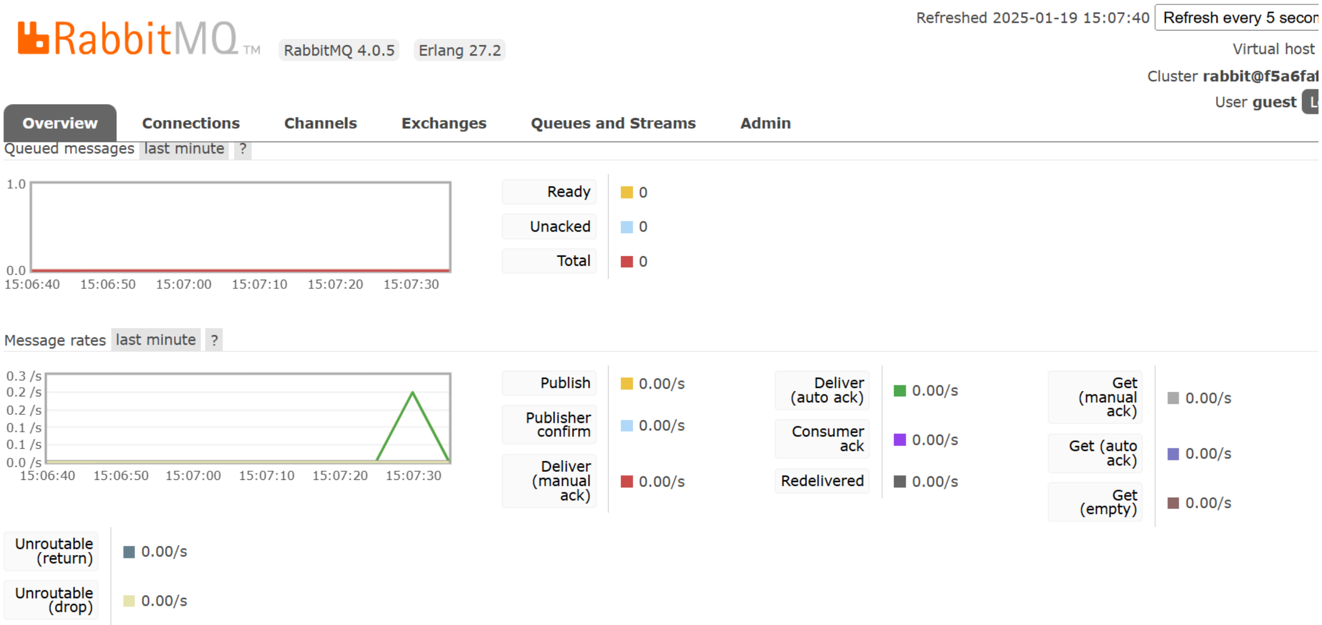
- The logs of the Song Service, which confirm that the message has been sent.
- The logs of the Song Suggestion Service, which confirm that the message has been consumed.
- The RabbitMQ UI, where the graph shows that the message was queued and consumed.
- The databases of both the Song Service and Song Suggestion Service, which will contain consistent data.



```
PS C:\Users\tony\Desktop\Semester 6 redo\Individual\song-service> go run ./cmd/song-service
2025/01/19 14:15:23 RabbitMQ connection and channel initialized
2025/01/19 14:15:23 RabbitMQ successfully initialized.
2025/01/19 14:15:23 Server is ready to handle requests on http://localhost:8080.
2025/01/19 15:07:28 Published message to RabbitMQ: {"event":"created","song_id":"678d072018bd6b23c94fc92d","title":"Fire fly","artist":"Owl city"}
2025/01/19 15:07:28 Successfully published song creation event to RabbitMQ
```



```
PS C:\Users\tony\Desktop\Semester 6 redo\Individual\song-suggestion-service> go run ./cmd/song-suggestion-service
2025/01/19 14:16:09 Starting application...
2025/01/19 14:16:09 MongoDB connected successfully
2025/01/19 14:16:09 RabbitMQ initialized with queue: song_events
2025/01/19 14:16:09 RabbitMQ initialized successfully
2025/01/19 14:16:09 Starting HTTP server on :8081...
2025/01/19 15:07:28 Received message: {"event":"created","song_id":"678d072018bd6b23c94fc92d","title":"Fire fly","artist":"Owl city"}
2025/01/19 15:07:28 Unmarshalled event: {Song_ID:678d072018bd6b23c94fc92d Event:created Title:Fire fly Artist:Owl city}
2025/01/19 15:07:28 Song suggestion created successfully for Fire fly by Owl city
```



Refreshed 2025-01-19 15:07:40 Refresh every 5 secon

Virtual host

Cluster **rabbit@f5a6fa**

User **guest**

Queued messages last minute ?

Ready 🟨 0

Unacked 🟦 0

Total 🟥 0

Message rates last minute ?

Publish 🟨 0.00/s

Publisher confirm 🟦 0.00/s

Deliver (manual ack) 🟥 0.00/s

Deliver (auto ack) 🟩 0.00/s

Consumer ack 🟪 0.00/s

Redelivered ■ 0.00/s

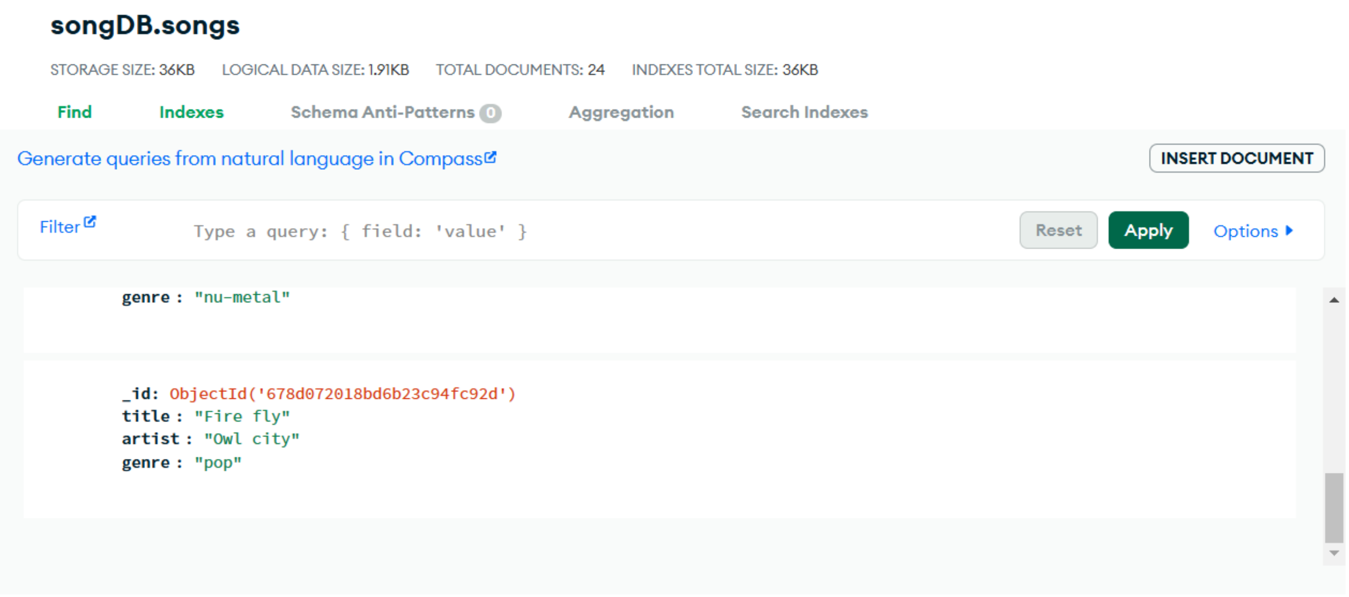
Get (manual ack) ■ 0.00/s

Get (auto ack) ■ 0.00/s

Get (empty) ■ 0.00/s

Unroutable (return) ■ 0.00/s

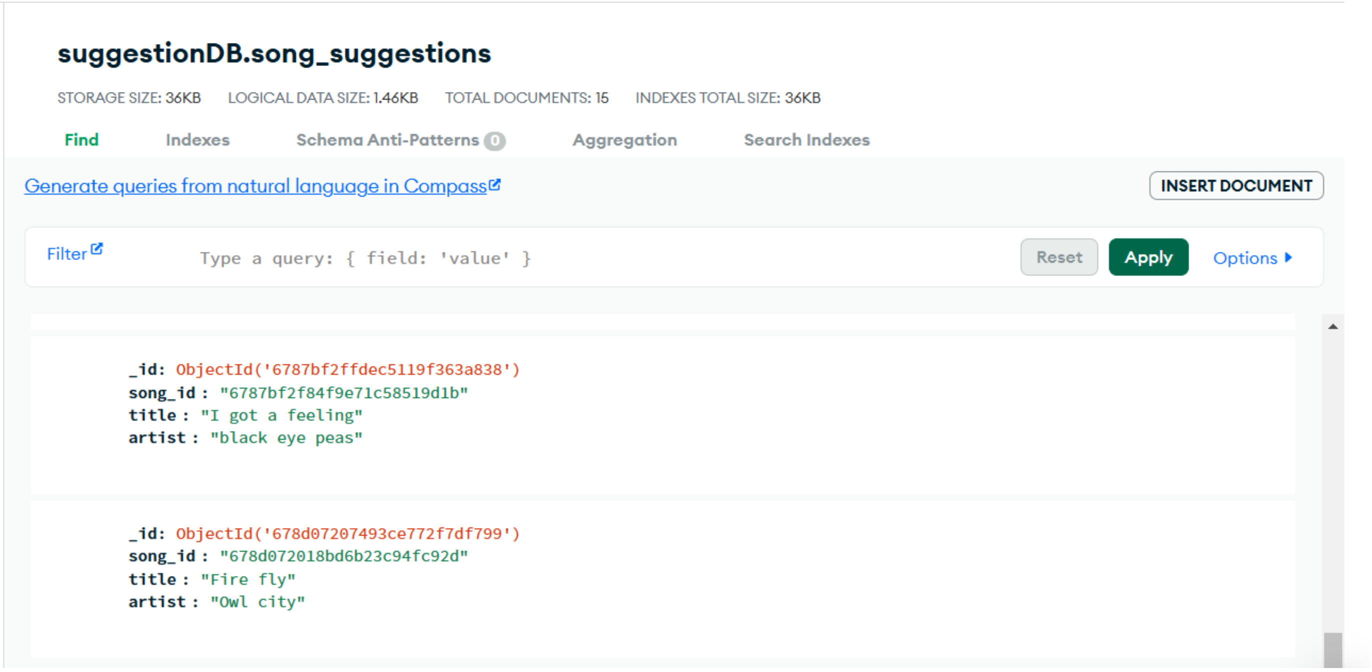
Unroutable (drop) ■ 0.00/s



songDB.songs

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 1.91KB TOTAL DOCUMENTS: 24 INDEXES TOTAL SIZE: 36KB

```
genre : "nu-metal"

_id: ObjectId('678d072018bd6b23c94fc92d')
title : "Fire fly"
artist : "Owl city"
genre : "pop"
```



suggestionDB.song_suggestions

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 1.46KB TOTAL DOCUMENTS: 15 INDEXES TOTAL SIZE: 36KB

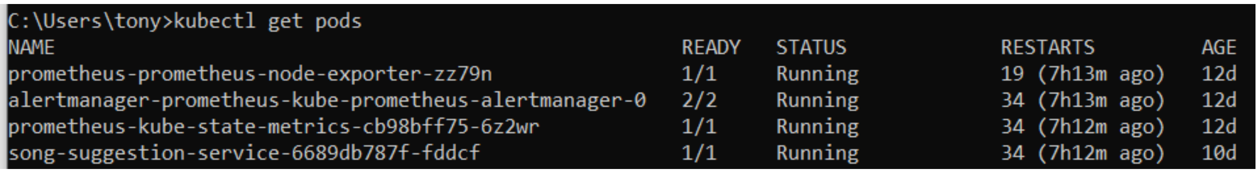
```
_id: ObjectId('6787bf2ffdec5119f363a838')
song_id : "6787bf2f84f9e71c58519d1b"
title : "I got a feeling"
artist : "black eye peas"

_id: ObjectId('678d07207493ce772f7df799')
song_id : "678d072018bd6b23c94fc92d"
title : "Fire fly"
artist : "Owl city"
```

Load testing tool (individual)

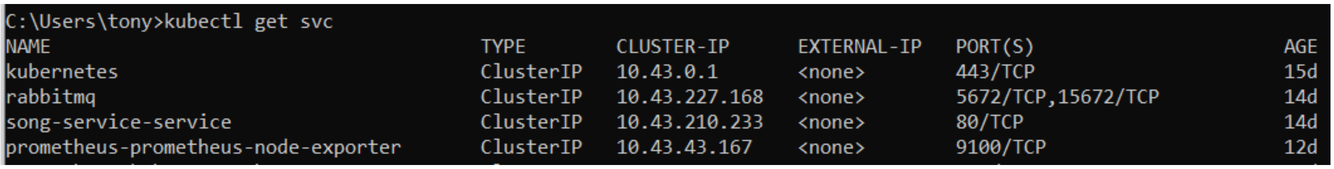
For load testing, I used k6 to test the song suggestion service deployed on k3d. Load testing simulates how many users can use your application before it crashes. It is typically done to measure how many requests your service can handle during peak hours, the time it takes to execute requests, and other performance metrics. This helps determine the number of user requests your service can manage before requiring additional resources. I performed the load test on a GET method to assess how many requests it could handle, essentially simulating multiple users calling the GET function.

How I did the load test

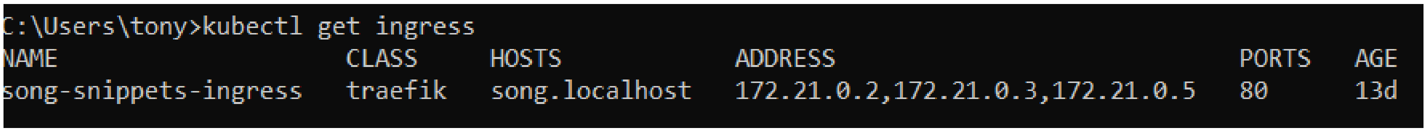
I deployed my song suggestion service to k3d and created a service and ingress to simulate a production environment. I also added a Horizontal Pod Autoscaler (HPA), which automatically adds or removes pods (instances of your application) based on workload. To set this up, I created an hpa.yaml file and pointed it to my deployment—in this case, the song suggestion service—to enable scaling.



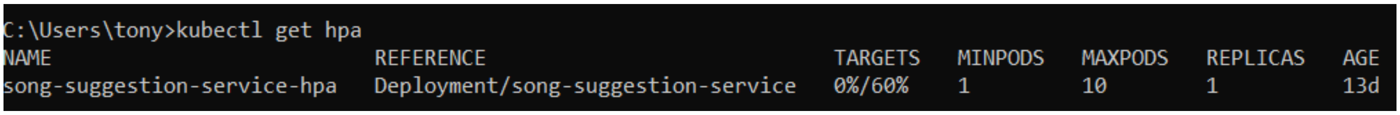
```
C:\Users\tony>kubectl get pods
NAME                                                   READY   STATUS      RESTARTS          AGE
prometheus-prometheus-node-exporter-zz79n              1/1     Running     19 (7h13m ago)    12d
alertmanager-prometheus-kube-prometheus-alertmanager-0 2/2     Running     34 (7h13m ago)    12d
prometheus-kube-state-metrics-cb98bff75-6z2wr          1/1     Running     34 (7h12m ago)    12d
song-suggestion-service-6689db787f-fddcf               1/1     Running     34 (7h12m ago)    10d
```



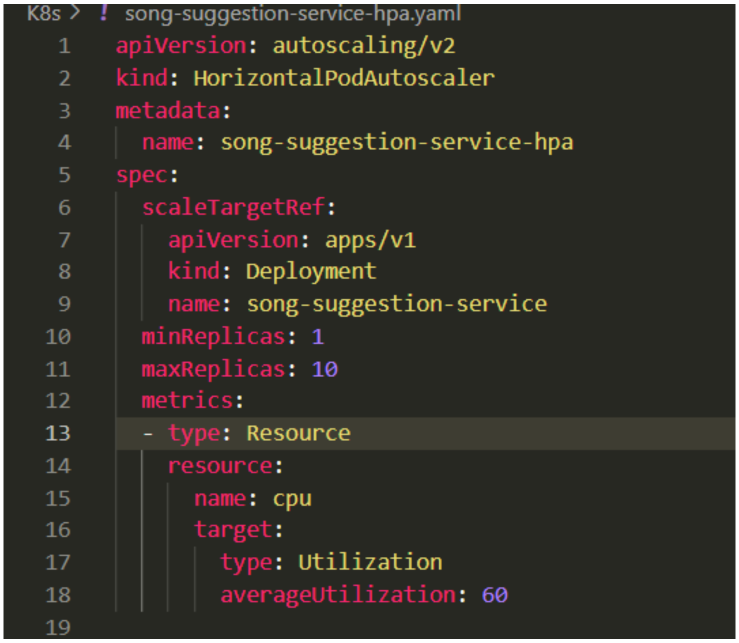
```
C:\Users\tony>kubectl get svc
NAME                                  TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
kubernetes                            ClusterIP   10.43.0.1       <none>        443/TCP                      15d
rabbitmq                              ClusterIP   10.43.227.168   <none>        5672/TCP,15672/TCP           14d
song-service-service                  ClusterIP   10.43.210.233   <none>        80/TCP                       14d
prometheus-prometheus-node-exporter   ClusterIP   10.43.43.167    <none>        9100/TCP                     12d
```



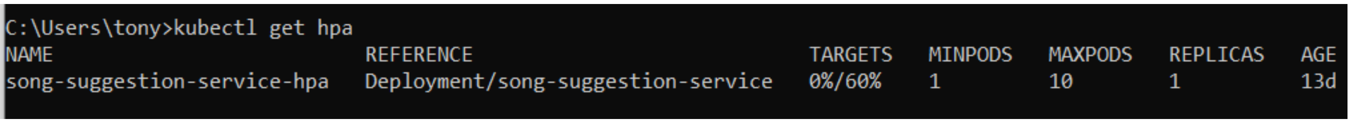
```
C:\Users\tony>kubectl get ingress
NAME                    CLASS     HOSTS            ADDRESS                              PORTS   AGE
song-snippets-ingress   traefik   song.localhost   172.21.0.2,172.21.0.3,172.21.0.5     80      13d
```



```
C:\Users\tony>kubectl get hpa
NAME                          REFERENCE                                 TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
song-suggestion-service-hpa   Deployment/song-suggestion-service        0%/60%    1         10        1          13d
```

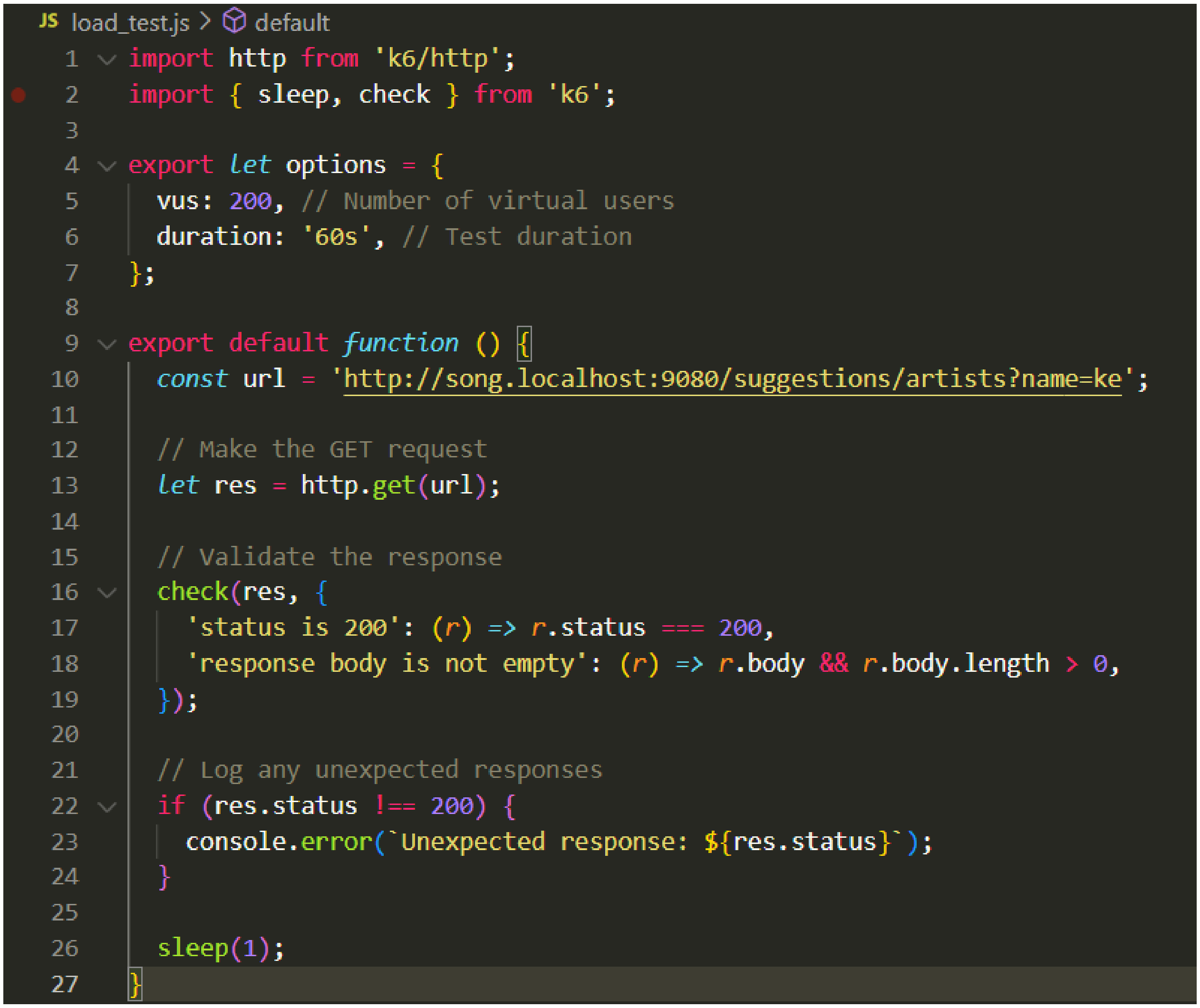


```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: song-suggestion-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: song-suggestion-service
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 60

```

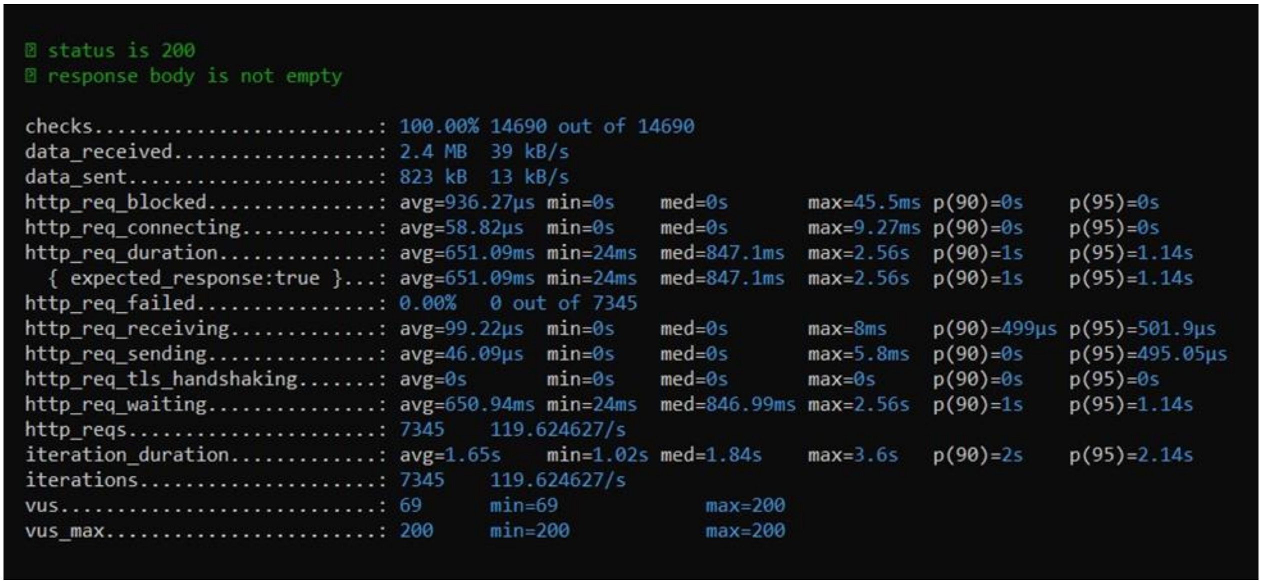


```
C:\Users\tony>kubectl get hpa
NAME                        REFERENCE                              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
song-suggestion-service-hpa   Deployment/song-suggestion-service   0%/60%    1         10        1          13d
```

After that, I created a k6 load test script in JavaScript and pointed it to my ingress endpoint for load testing. In the load test, I specified that it should call the endpoint: http://song.localhost:9080/suggestions/artists?name=ke using 200 virtual users accessing the endpoint to search for artists whose names start with 'ke' over a duration of 60 seconds. The test ensures that the response status is 200 and that the response body is not empty. It also logs any unexpected errors and simulates a 1-second delay to mimic user actions. To run the test, I used the command k6 run load_test.js.



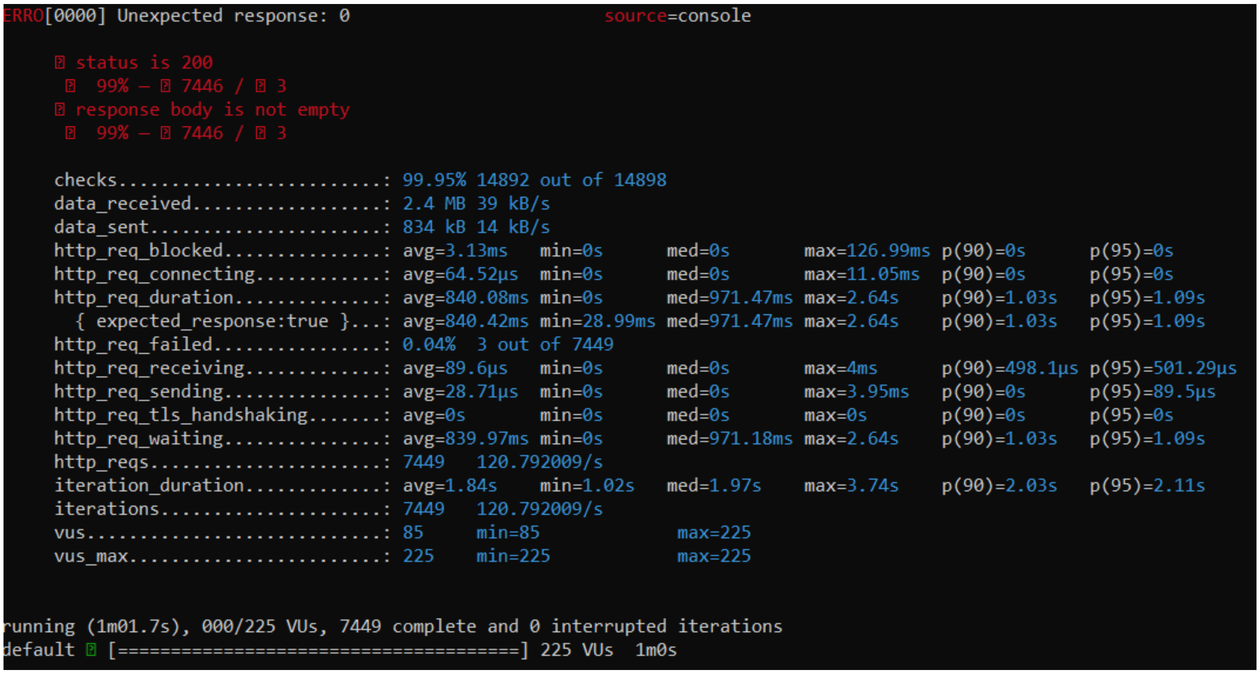
```javascript
import http from 'k6/http';
import { sleep, check } from 'k6';

export let options = {
  vus: 200, // Number of virtual users
  duration: '60s', // Test duration
};

export default function () {
  const url = 'http://song.localhost:9080/suggestions/artists?name=ke';

  // Make the GET request
  let res = http.get(url);

  // Validate the response
  check(res, {
    'status is 200': (r) => r.status === 200,
    'response body is not empty': (r) => r.body && r.body.length > 0,
  });

  // Log any unexpected responses
  if (res.status !== 200) {
    console.error(`Unexpected response: ${res.status}`);
  }

  sleep(1);
}
```



```
□ status is 200
□ response body is not empty

checks.........................: 100.00% 14690 out of 14690
data_received..................: 2.4 MB  39 kB/s
data_sent......................: 823 kB  13 kB/s
http_req_blocked...............: avg=936.27µs min=0s    med=0s       max=45.5ms p(90)=0s     p(95)=0s
http_req_connecting............: avg=58.82µs  min=0s    med=0s       max=9.27ms p(90)=0s     p(95)=0s
http_req_duration..............: avg=651.09ms min=24ms  med=847.1ms  max=2.56s  p(90)=1s     p(95)=1.14s
  { expected_response:true }...: avg=651.09ms min=24ms  med=847.1ms  max=2.56s  p(90)=1s     p(95)=1.14s
http_req_failed................: 0.00%   0 out of 7345
http_req_receiving.............: avg=99.22µs  min=0s    med=0s       max=8ms    p(90)=499µs p(95)=501.9µs
http_req_sending...............: avg=46.09µs  min=0s    med=0s       max=5.8ms  p(90)=0s     p(95)=495.05µs
http_req_tls_handshaking.......: avg=0s       min=0s    med=0s       max=0s     p(90)=0s     p(95)=0s
http_req_waiting...............: avg=650.94ms min=24ms  med=846.99ms max=2.56s  p(90)=1s     p(95)=1.14s
http_reqs......................: 7345    119.624627/s
iteration_duration.............: avg=1.65s    min=1.02s med=1.84s    max=3.6s   p(90)=2s     p(95)=2.14s
iterations.....................: 7345    119.624627/s
vus............................: 69      min=69            max=200
vus_max........................: 200     min=200           max=200
```

After the test is done running, it will output the results. From the results, you can see that it handled 7,345 requests, and the checks indicate that the test was 100% successful. The output also includes additional details that can help you analyze the performance of your service. While load testing helps simulate how many users your service can handle simultaneously, for more specific insights like resource utilization, it is better to add monitoring tools. Load testing is primarily focused on understanding the capacity of your service.

My maximum is 200 users simultaneously calling the GET method for 60 seconds. Beyond that, it won’t achieve a 100% success rate due to the limited resources of my laptop, particularly its CPU.



```
ERRO[0000] Unexpected response: 0                        source=console

    □ status is 200
     □  99% — □ 7446 / □ 3
    □ response body is not empty
     □  99% — □ 7446 / □ 3

    checks.........................: 99.95% 14892 out of 14898
    data_received..................: 2.4 MB 39 kB/s
    data_sent......................: 834 kB 14 kB/s
    http_req_blocked...............: avg=3.13ms   min=0s       med=0s       max=126.99ms p(90)=0s       p(95)=0s
    http_req_connecting............: avg=64.52µs  min=0s       med=0s       max=11.05ms  p(90)=0s       p(95)=0s
    http_req_duration..............: avg=840.08ms min=0s       med=971.47ms max=2.64s    p(90)=1.03s    p(95)=1.09s
      { expected_response:true }...: avg=840.42ms min=28.99ms med=971.47ms max=2.64s    p(90)=1.03s    p(95)=1.09s
    http_req_failed................: 0.04%  3 out of 7449
    http_req_receiving.............: avg=89.6µs   min=0s       med=0s       max=4ms      p(90)=498.1µs p(95)=501.29µs
    http_req_sending...............: avg=28.71µs  min=0s       med=0s       max=3.95ms   p(90)=0s       p(95)=89.5µs
    http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
    http_req_waiting...............: avg=839.97ms min=0s       med=971.18ms max=2.64s    p(90)=1.03s    p(95)=1.09s
    http_reqs......................: 7449   120.792009/s
    iteration_duration.............: avg=1.84s    min=1.02s    med=1.97s    max=3.74s    p(90)=2.03s    p(95)=2.11s
    iterations.....................: 7449   120.792009/s
    vus............................: 85     min=85             max=225
    vus_max........................: 225    min=225            max=225


running (1m01.7s), 000/225 VUs, 7449 complete and 0 interrupted iterations
default □ [======================================] 225 VUs  1m0s
```

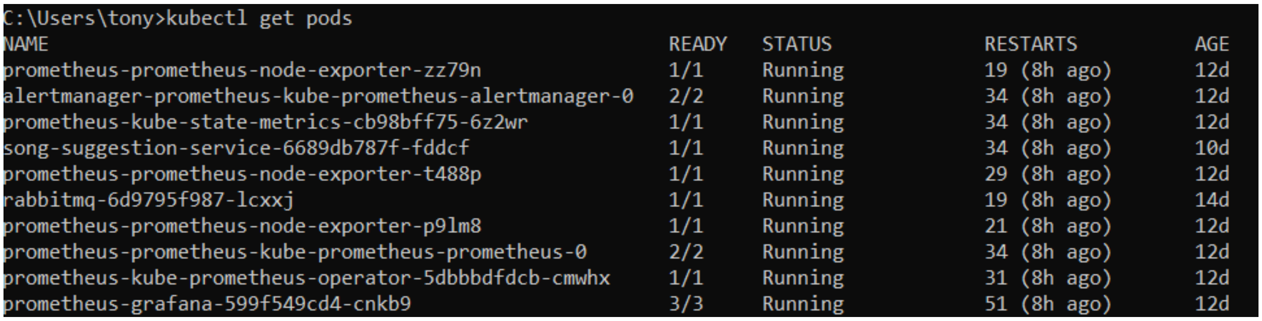
Monitoring tools (individual)

The monitoring tools I'm using are Prometheus and Grafana because they are open-source, widely used, and popular. Prometheus collects the metrics from your service, while Grafana visualizes these metrics in the form of graphs.

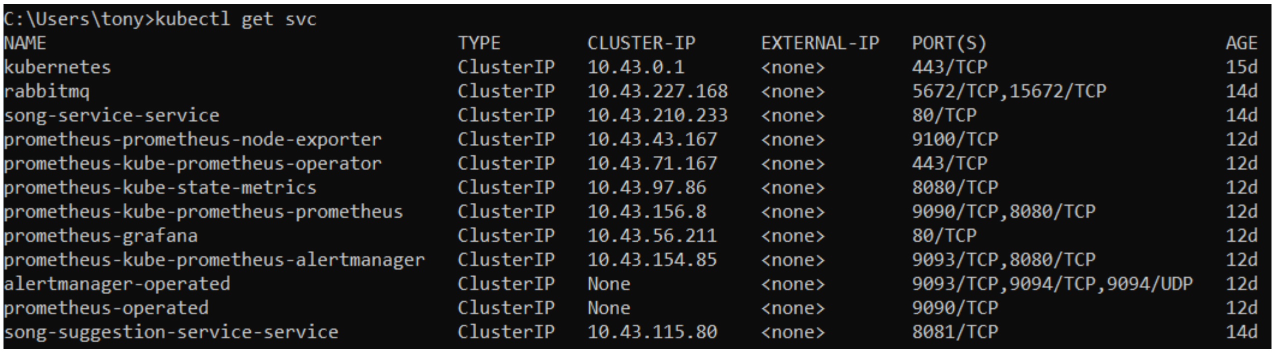
You can use these monitoring tools for various purposes. I'm using them to monitor the scalability of my application. Specifically, I'm using Prometheus and Grafana to monitor my song suggestion service, which is deployed on k3d Kubernetes.

Setup

I added Prometheus and Grafana to my k3s Kubernetes cluster, where my song suggestion service is deployed. I did this because I want to monitor the pod for my song suggestion service. To access the Prometheus UI, I had to port-forward the prometheus-kube-prometheus-prometheus service to port 9090 so I could access it on localhost:9090. This allows me to use the UI and see what Prometheus is scraping in Kubernetes, which is useful to verify if it's scraping data from my song suggestion service. The Grafana UI is accessible on port 3000 at localhost:3000. I didn't need to do any port-forwarding for this; it just works out of the box.

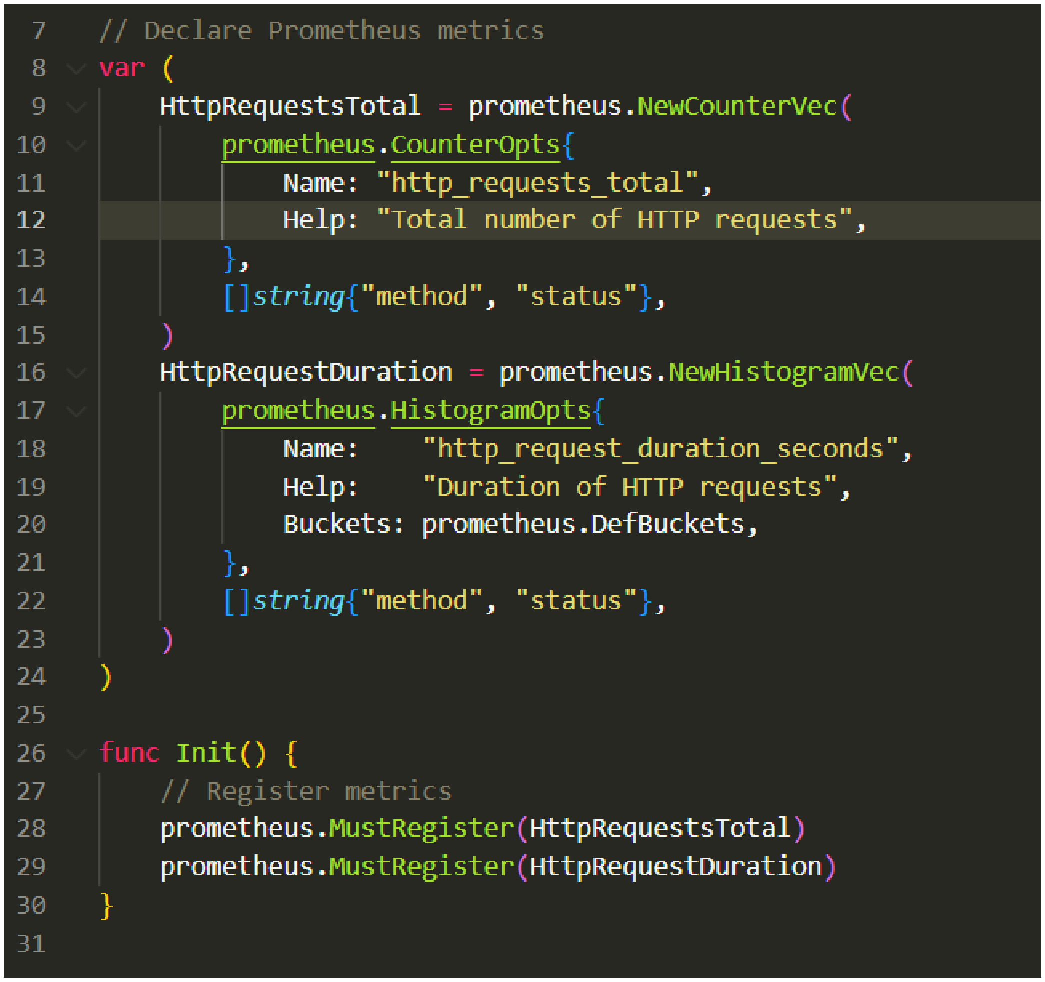


```
C:\Users\tony>kubectl get pods
NAME                                                 READY   STATUS             RESTARTS         AGE
prometheus-prometheus-node-exporter-zz79n            1/1     Running            19 (8h ago)      12d
alertmanager-prometheus-kube-prometheus-alertmanager-0   2/2     Running            34 (8h ago)      12d
prometheus-kube-state-metrics-cb98bff75-6z2wr        1/1     Running            34 (8h ago)      12d
song-suggestion-service-6689db787f-fddcf             1/1     Running            34 (8h ago)      10d
prometheus-prometheus-node-exporter-t488p            1/1     Running            29 (8h ago)      12d
rabbitmq-6d9795f987-lcxxj                            1/1     Running            19 (8h ago)      14d
prometheus-prometheus-node-exporter-p9lm8            1/1     Running            21 (8h ago)      12d
prometheus-prometheus-kube-prometheus-prometheus-0   2/2     Running            34 (8h ago)      12d
prometheus-kube-prometheus-operator-5dbbbdfdcb-cmwhx   1/1     Running            31 (8h ago)      12d
prometheus-grafana-599f549cd4-cnkb9                  3/3     Running            51 (8h ago)      12d
```



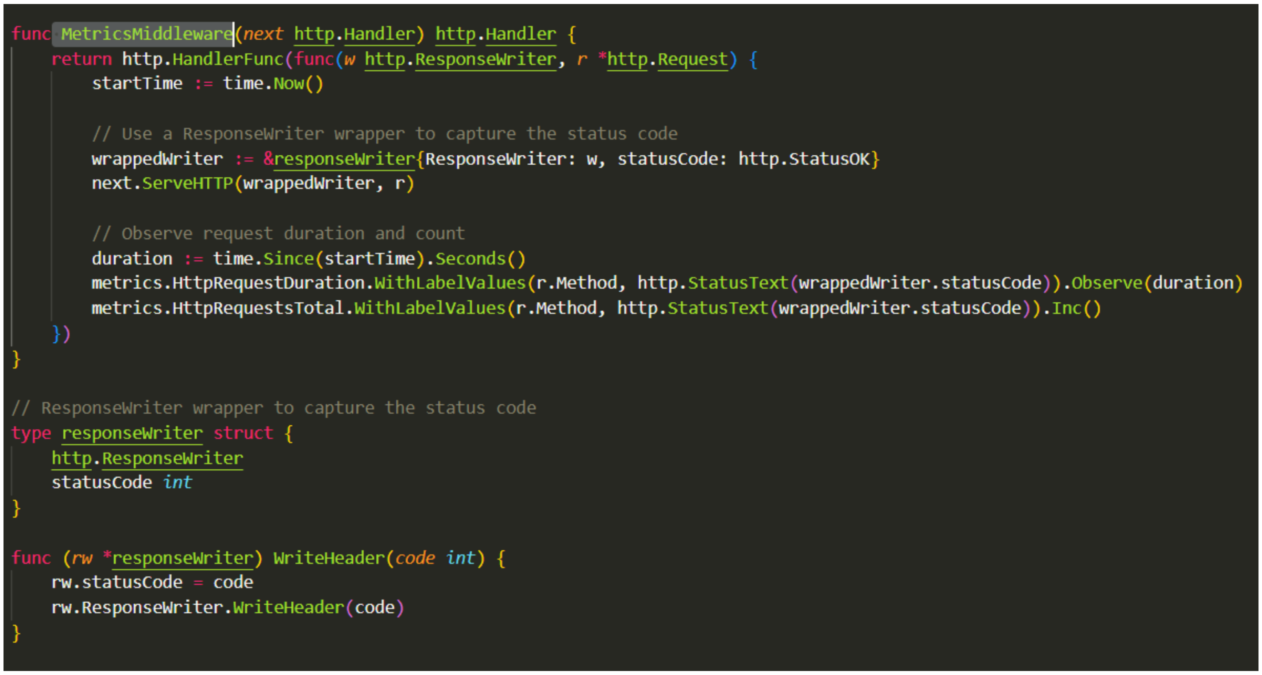
```
C:\Users\tony>kubectl get svc
NAME                                      TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
kubernetes                                ClusterIP   10.43.0.1       <none>        443/TCP                      15d
rabbitmq                                  ClusterIP   10.43.227.168   <none>        5672/TCP,15672/TCP           14d
song-service-service                      ClusterIP   10.43.210.233   <none>        80/TCP                       14d
prometheus-prometheus-node-exporter       ClusterIP   10.43.43.167    <none>        9100/TCP                     12d
prometheus-kube-prometheus-operator       ClusterIP   10.43.71.167    <none>        443/TCP                      12d
prometheus-kube-state-metrics             ClusterIP   10.43.97.86     <none>        8080/TCP                     12d
prometheus-kube-prometheus-prometheus     ClusterIP   10.43.156.8     <none>        9090/TCP,8080/TCP            12d
prometheus-grafana                        ClusterIP   10.43.56.211    <none>        80/TCP                       12d
prometheus-kube-prometheus-alertmanager   ClusterIP   10.43.154.85    <none>        9093/TCP,8080/TCP            12d
alertmanager-operated                     ClusterIP   None            <none>        9093/TCP,9094/TCP,9094/UDP   12d
prometheus-operated                       ClusterIP   None            <none>        9090/TCP                     12d
song-suggestion-service-service           ClusterIP   10.43.115.80    <none>        8081/TCP                     14d
```

For scraping the metrics from my song suggestion service, I first created the metrics I want to monitor. These metrics track the number of requests and the request duration in seconds, ensuring that they are available for Prometheus to collect.

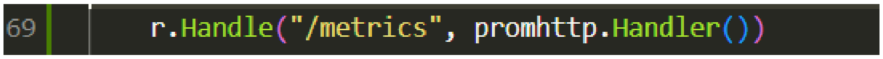


```go
// Declare Prometheus metrics
var (
    HttpRequestsTotal = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "http_requests_total",
            Help: "Total number of HTTP requests",
        },
        []string{"method", "status"},
    )
    HttpRequestDuration = prometheus.NewHistogramVec(
        prometheus.HistogramOpts{
            Name:    "http_request_duration_seconds",
            Help:    "Duration of HTTP requests",
            Buckets: prometheus.DefBuckets,
        },
        []string{"method", "status"},
    )
)

func Init() {
    // Register metrics
    prometheus.MustRegister(HttpRequestsTotal)
    prometheus.MustRegister(HttpRequestDuration)
}
```

I created the MetricsMiddleware handler. Middleware is a function or piece of code that sits between the incoming HTTP request and the final processing of that request. It tracks HTTP request metrics, such as request duration and the total number of requests. The middleware wraps around the handler, measures the time taken for each request, and updates the Prometheus metrics accordingly. It also captures the HTTP status code of each response to include in the metrics. Additionally, I added a /metrics endpoint to expose the metrics for Prometheus to scrape.

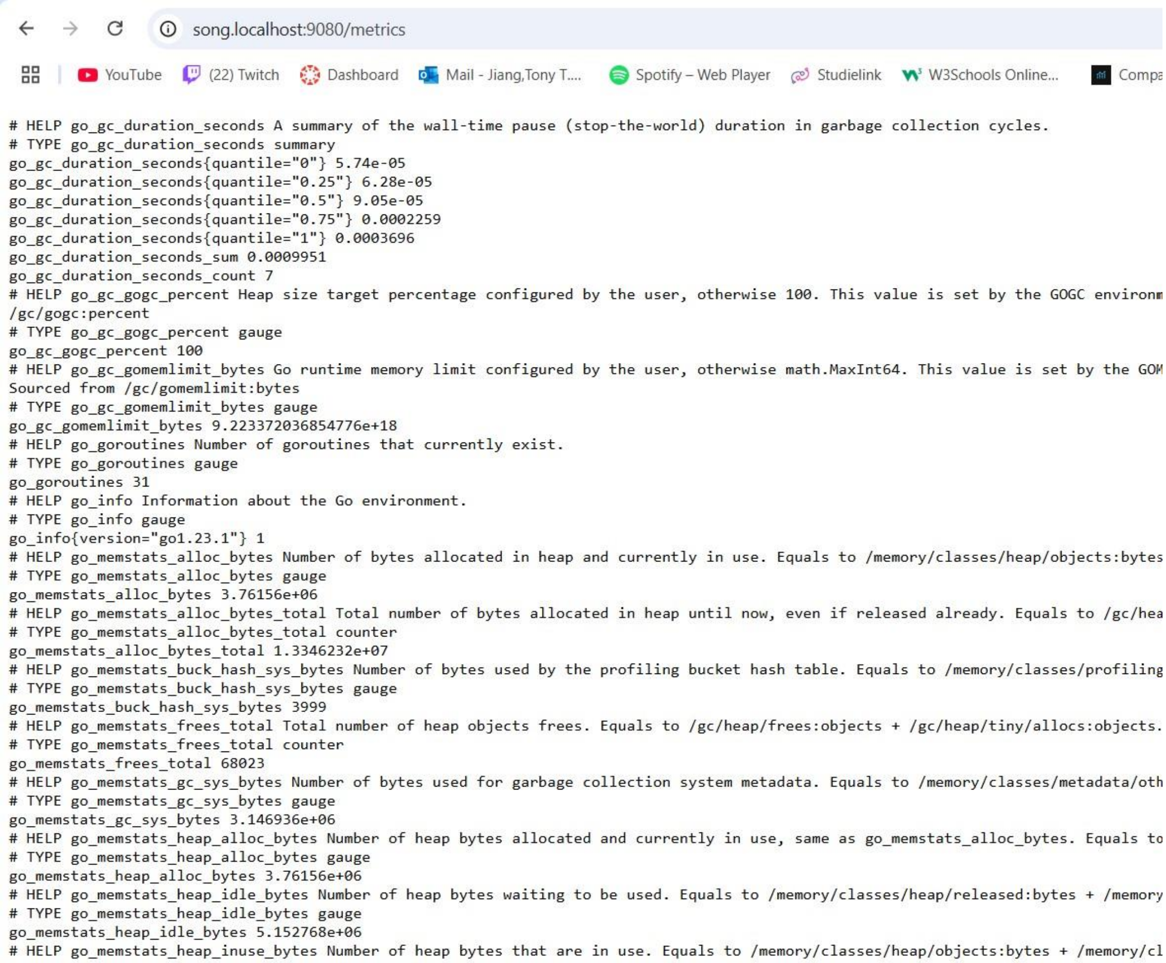


```
func MetricsMiddleware(next http.Handler) http.Handler {
	return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
		startTime := time.Now()

		// Use a ResponseWriter wrapper to capture the status code
		wrappedWriter := &responseWriter{ResponseWriter: w, statusCode: http.StatusOK}
		next.ServeHTTP(wrappedWriter, r)

		// Observe request duration and count
		duration := time.Since(startTime).Seconds()
		metrics.HttpRequestDuration.WithLabelValues(r.Method, http.StatusText(wrappedWriter.statusCode)).Observe(duration)
		metrics.HttpRequestsTotal.WithLabelValues(r.Method, http.StatusText(wrappedWriter.statusCode)).Inc()
	})
}

// ResponseWriter wrapper to capture the status code
type responseWriter struct {
	http.ResponseWriter
	statusCode int
}

func (rw *responseWriter) WriteHeader(code int) {
	rw.statusCode = code
	rw.ResponseWriter.WriteHeader(code)
}
```

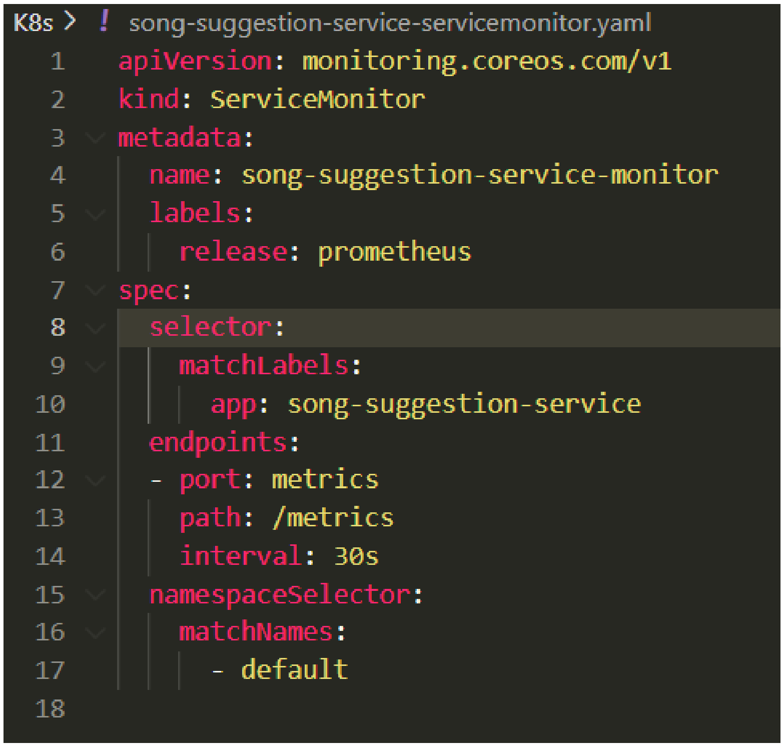


```
	r.Handle("/metrics", promhttp.Handler())
```

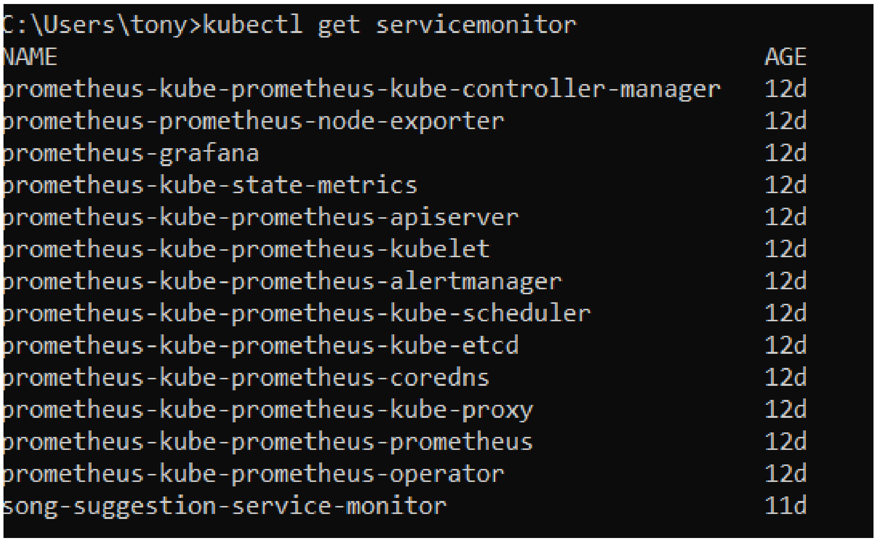
You can see the metrics endpoint below. Before Prometheus can scrape data from the song suggestion service in k3d Kubernetes, I need to create a servicemonitor.yaml file, which tells Prometheus which pods to monitor and at which endpoint. After applying the servicemonitor.yaml to Kubernetes, the song suggestion service should appear in the Prometheus UI, indicating that it is being scraped in Kubernetes.



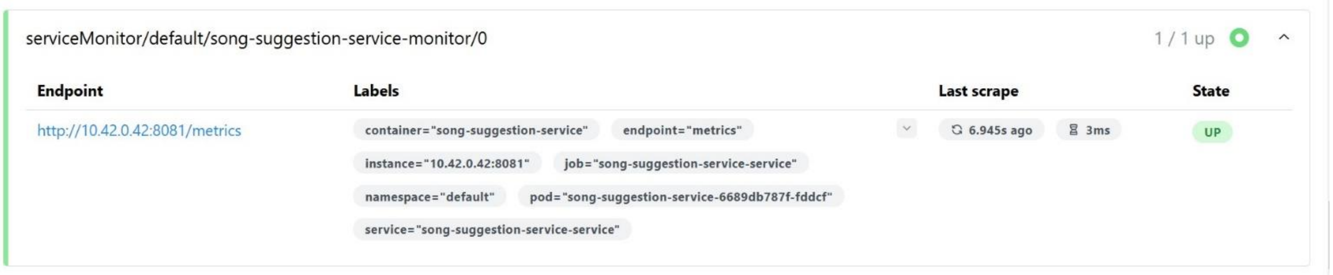
```
# HELP go_gc_duration_seconds A summary of the wall-time pause (stop-the-world) duration in garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 5.74e-05
go_gc_duration_seconds{quantile="0.25"} 6.28e-05
go_gc_duration_seconds{quantile="0.5"} 9.05e-05
go_gc_duration_seconds{quantile="0.75"} 0.0002259
go_gc_duration_seconds{quantile="1"} 0.0003696
go_gc_duration_seconds_sum 0.0009951
go_gc_duration_seconds_count 7
# HELP go_gc_gogc_percent Heap size target percentage configured by the user, otherwise 100. This value is set by the GOGC environm
/gc/gogc:percent
# TYPE go_gc_gogc_percent gauge
go_gc_gogc_percent 100
# HELP go_gc_gomemlimit_bytes Go runtime memory limit configured by the user, otherwise math.MaxInt64. This value is set by the GOM
Sourced from /gc/gomemlimit:bytes
# TYPE go_gc_gomemlimit_bytes gauge
go_gc_gomemlimit_bytes 9.223372036854776e+18
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 31
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.23.1"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated in heap and currently in use. Equals to /memory/classes/heap/objects:bytes
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 3.76156e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated in heap until now, even if released already. Equals to /gc/hea
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 1.3346232e+07
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table. Equals to /memory/classes/profiling
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 3999
# HELP go_memstats_frees_total Total number of heap objects frees. Equals to /gc/heap/frees:objects + /gc/heap/tiny/allocs:objects.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 68023
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata. Equals to /memory/classes/metadata/oth
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 3.146936e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and currently in use, same as go_memstats_alloc_bytes. Equals to
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 3.76156e+06
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used. Equals to /memory/classes/heap/released:bytes + /memory
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 5.152768e+06
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use. Equals to /memory/classes/heap/objects:bytes + /memory/cl
```



```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: song-suggestion-service-monitor
  labels:
    release: prometheus
spec:
  selector:
    matchLabels:
      app: song-suggestion-service
  endpoints:
  - port: metrics
    path: /metrics
    interval: 30s
  namespaceSelector:
    matchNames:
      - default

```



```
C:\Users\tony>kubectl get servicemonitor
NAME                                                AGE
prometheus-kube-prometheus-kube-controller-manager   12d
prometheus-prometheus-node-exporter                  12d
prometheus-grafana                                   12d
prometheus-kube-state-metrics                        12d
prometheus-kube-prometheus-apiserver                 12d
prometheus-kube-prometheus-kubelet                   12d
prometheus-kube-prometheus-alertmanager              12d
prometheus-kube-prometheus-kube-scheduler            12d
prometheus-kube-prometheus-kube-etcd                 12d
prometheus-kube-prometheus-coredns                   12d
prometheus-kube-prometheus-kube-proxy                12d
prometheus-kube-prometheus-prometheus                12d
prometheus-kube-prometheus-operator                  12d
song-suggestion-service-monitor                      11d
```

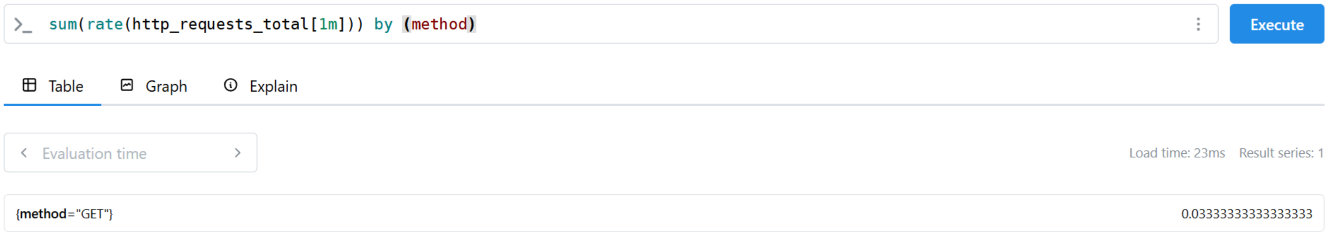


serviceMonitor/default/song-suggestion-service-monitor/0 — 1 / 1 up

Endpoint	Labels	Last scrape	State

http://10.42.0.42:8081/metrics	container="song-suggestion-service"
	endpoint="metrics"
	instance="10.42.0.42:8081"
	job="song-suggestion-service-service"
	namespace="default"
	pod="song-suggestion-service-6689db787f-fddcf"
	service="song-suggestion-service-service"
		6.945s ago
		3ms
			UP

With that I can use the query in Prometheus to get the total request over time my entering this expression in the query: sum(rate(http_requests_total[1m])) by (method), which would give me the result.



```
>_ sum(rate(http_requests_total[1m])) by (method)
```

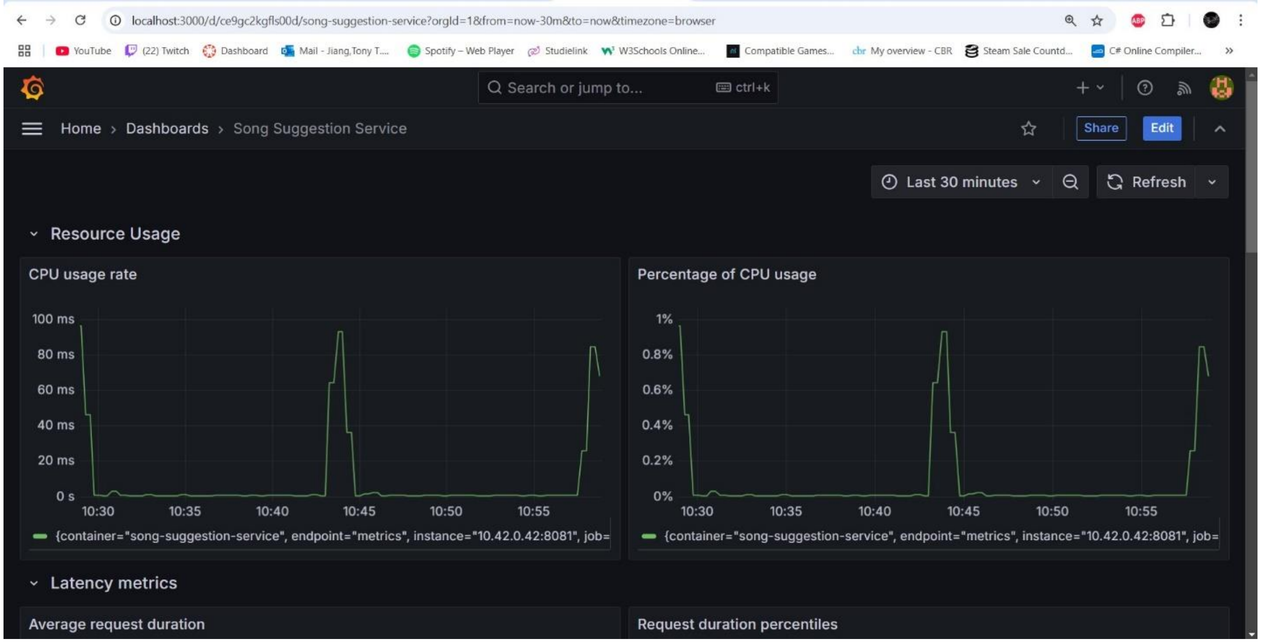
Load time: 23ms Result series: 1

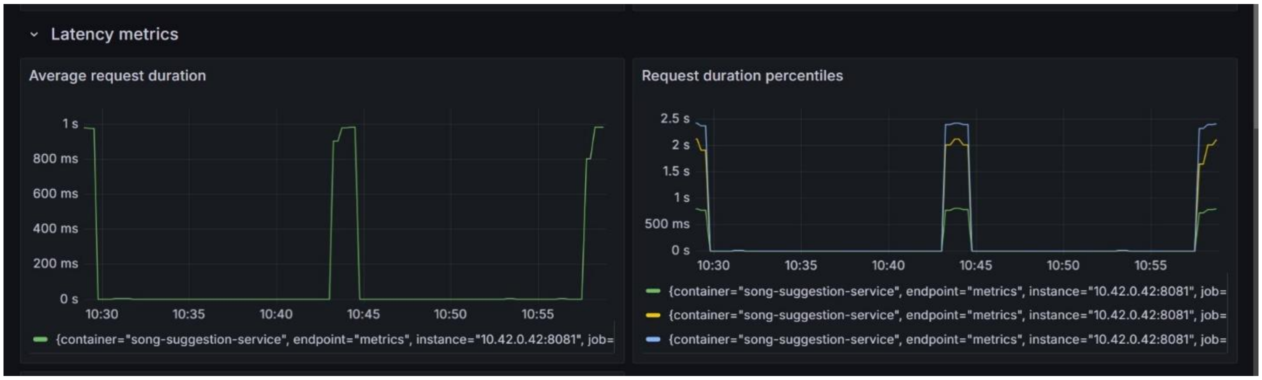
{method="GET"}	0.03333333333333333

The result is usually too plain, which is why I'm using Grafana to display a graph and provide a better understanding of what you're looking at. In Grafana, I created a custom dashboard for monitoring the song suggestion service. I'm monitoring how to scale the song suggestion service by tracking the following metrics:

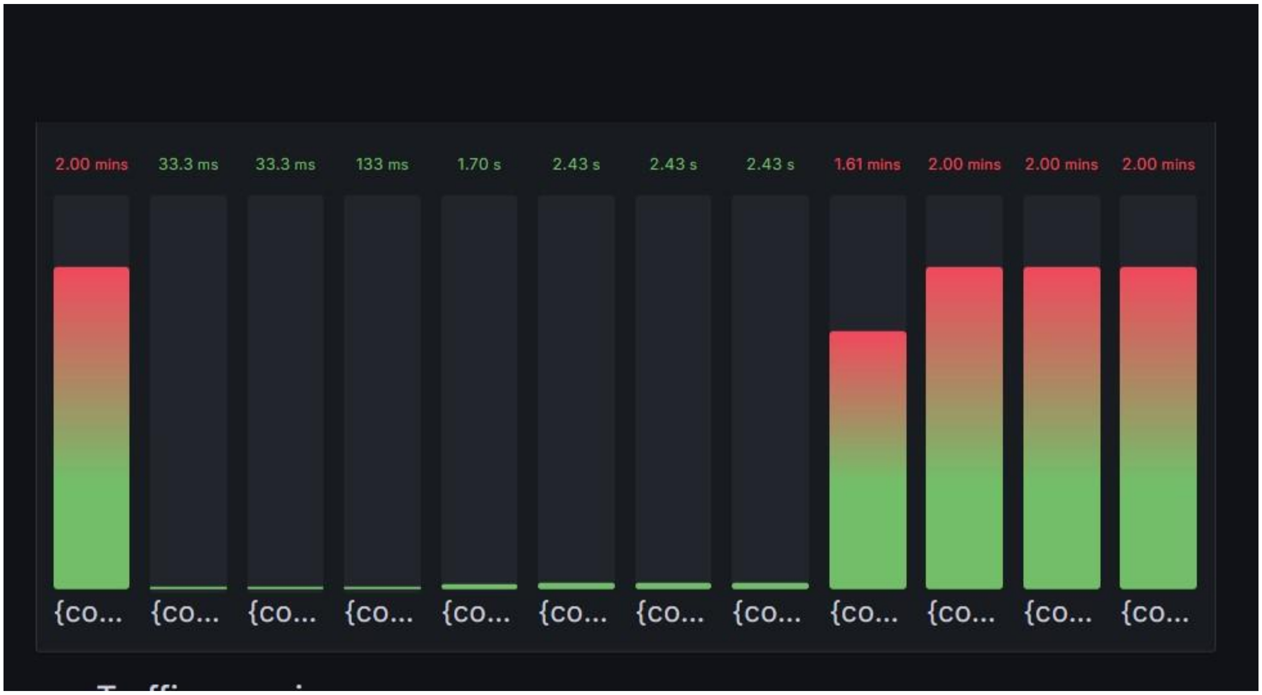
- CPU Usage Rate: Monitors CPU usage as seconds of CPU time per second.
- Percentage of CPU Usage: Displays CPU usage as a percentage of available capacity.
- Average Request Duration: Tracks the average request duration over time.
- Request Duration Percentiles: Displays the 50th, 95th, and 99th percentile response times to monitor latency.
- Request Duration Heatmap: Displays a heatmap of request durations to identify patterns.

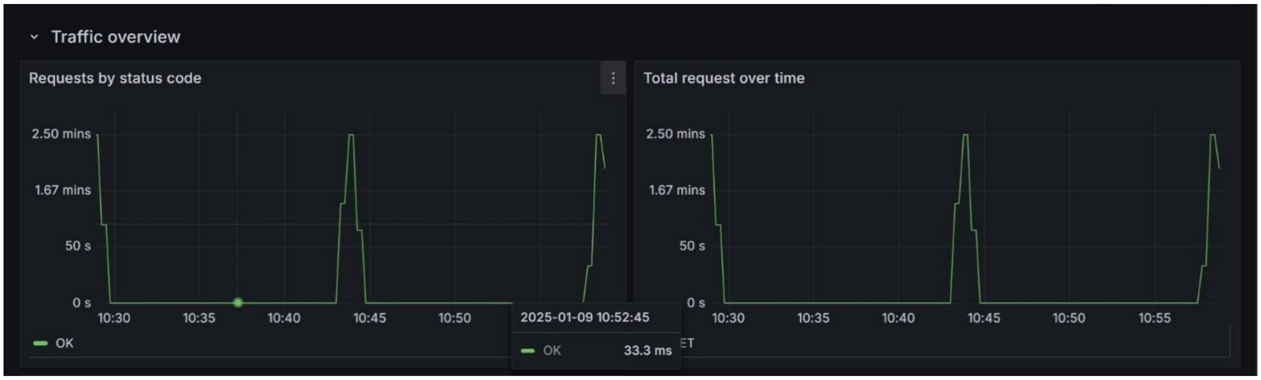
- Requests by Status Code: Monitors how many requests result in successful, client error, or server error status codes.
- Total Requests Over Time: Tracks the request rate per second, grouped by HTTP methods like GET, POST, DELETE, and PUT.





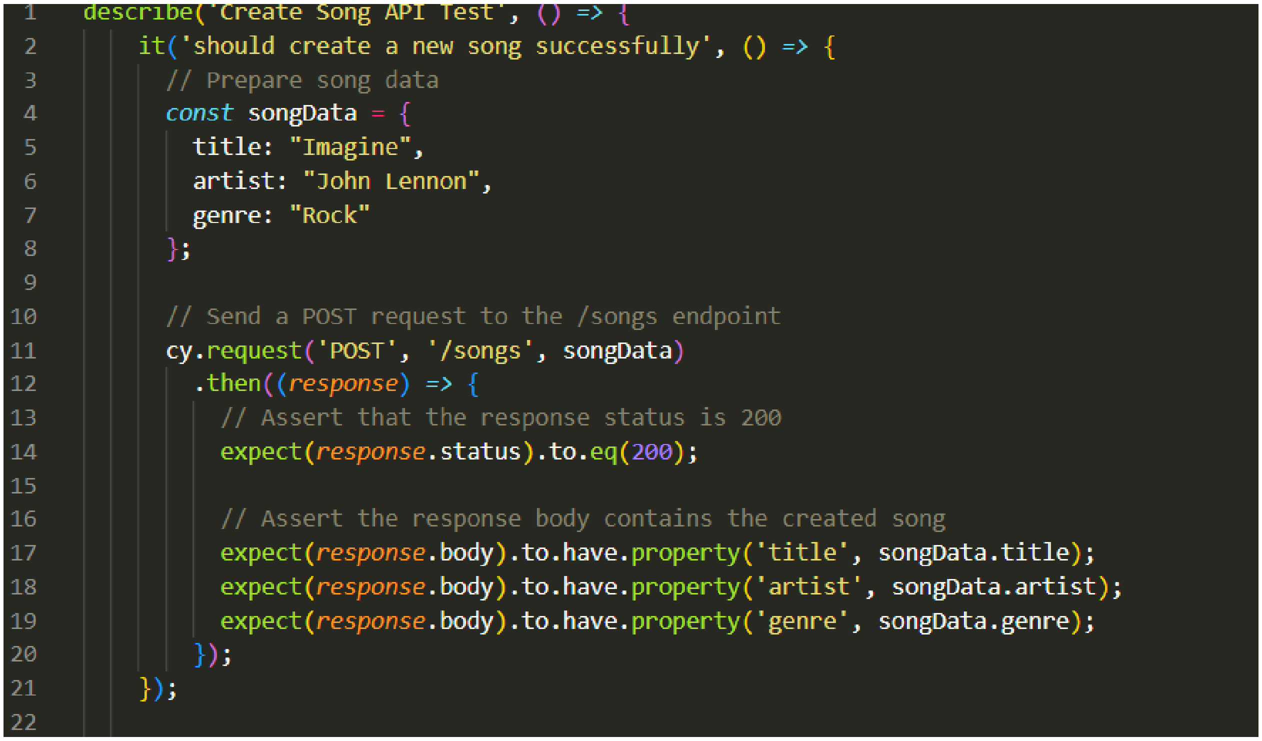
Latency metrics
Average request duration
1 s
800 ms
600 ms
400 ms
200 ms
0 s
10:30
10:35
10:40
10:45
10:50
10:55
{container="song-suggestion-service", endpoint="metrics", instance="10.42.0.42:8081", job=
Request duration percentiles
2.5 s
2 s
1.5 s
1 s
500 ms
0 s
10:30
10:35
10:40
10:45
10:50
10:55
{container="song-suggestion-service", endpoint="metrics", instance="10.42.0.42:8081", job=
{container="song-suggestion-service", endpoint="metrics", instance="10.42.0.42:8081", job=
{container="song-suggestion-service", endpoint="metrics", instance="10.42.0.42:8081", job=



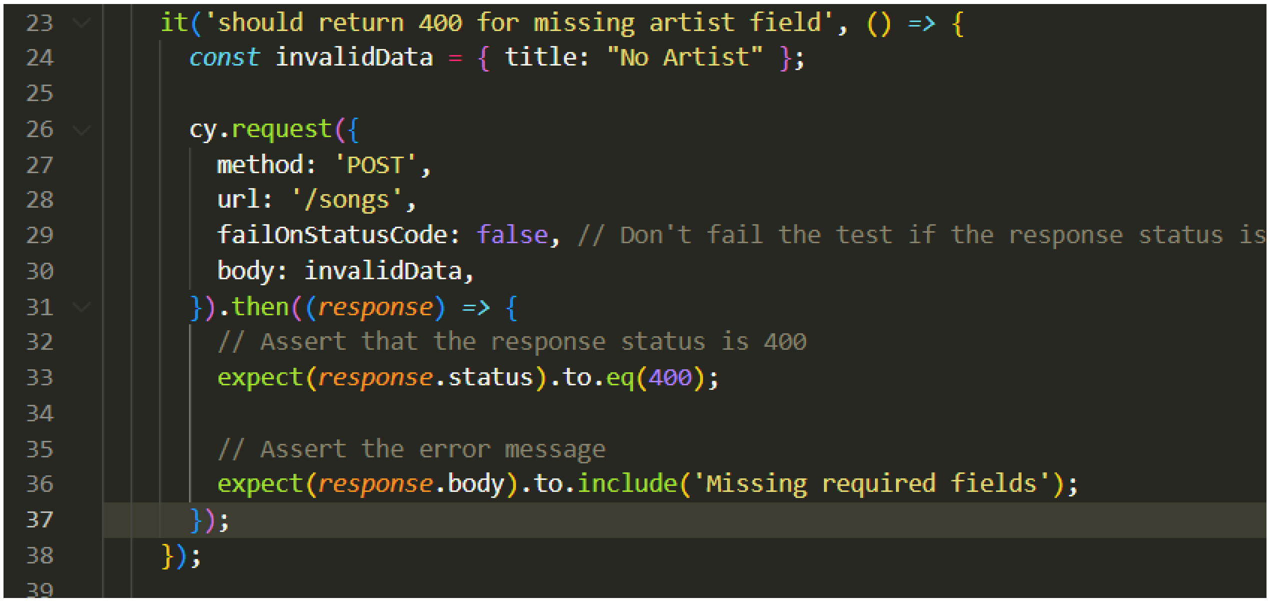


End to end testing tool (individual)

For the end-to-end tests, I use Cypress. I use it to test the happy path and verify the expected results from my endpoints. You don’t need to create a frontend to do this; you can test the endpoints directly. I tested the 'create song' endpoint in my song service, covering both the happy path and the error flow. These tests are written in a .cy.js file, as Cypress uses this type of file to execute tests. I wrote both the happy path and error scenarios in the test file, which I named createSong.cy.js.

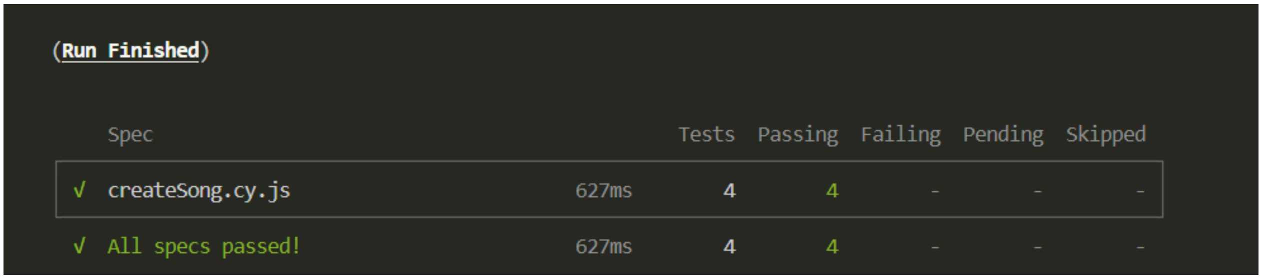


```
describe('Create Song API Test', () => {
  it('should create a new song successfully', () => {
   // Prepare song data
   const songData = {
    title: "Imagine",
    artist: "John Lennon",
    genre: "Rock"
   };

   // Send a POST request to the /songs endpoint
   cy.request('POST', '/songs', songData)
    .then((response) => {
     // Assert that the response status is 200
     expect(response.status).to.eq(200);

     // Assert the response body contains the created song
     expect(response.body).to.have.property('title', songData.title);
     expect(response.body).to.have.property('artist', songData.artist);
     expect(response.body).to.have.property('genre', songData.genre);
    });
  });

```



```
  it('should return 400 for missing artist field', () => {
   const invalidData = { title: "No Artist" };

   cy.request({
    method: 'POST',
    url: '/songs',
    failOnStatusCode: false, // Don't fail the test if the response status is
    body: invalidData,
   }).then((response) => {
    // Assert that the response status is 400
    expect(response.status).to.eq(400);

    // Assert the error message
    expect(response.body).to.include('Missing required fields');
   });
  });
```

I run the test by typing npx cypress run. After running the test, it will show the result, indicating whether it passed or failed. In my case, the test passed.



```
(Run Finished)


     Spec                                              Tests  Passing  Failing  Pending  Skipped
  ┌────────────────────────────────────────────────────────────────────────────────────────────┐
  │ √  createSong.cy.js                       627ms        4        4        -        -        - │
  └────────────────────────────────────────────────────────────────────────────────────────────┘
    √  All specs passed!                      627ms        4        4        -        -        -
```

With that I can added to my CI/CD pipeline.