

LAB 6: OAuth2, Json Web Token and Message Queue

1. OAuth2 with Github

In this exercise, we will implement the login method using OAuth2 with GitHub

First, we access the following URL to create a new project and configure parameters such as Homepage URL and Authorization callback URL as shown below.

https://github.com/settings/developers

The screenshot shows the GitHub developer settings page for creating a new application. It includes fields for Homepage URL (http://localhost:3000), Application description (optional), Authorization callback URL (http://localhost:3000/auth/github/callback), and a note about listing the application in the GitHub Marketplace.

Homepage URL *

http://localhost:3000

The full URL to your application homepage.

Application description

Application description is optional

This is displayed to all users of your application.

Authorization callback URL *

http://localhost:3000/auth/github/callback

Your application's callback URL. Read our OAuth documentation for more information.

After successfully creating the project, we will have a ClientID. To create Client secrets, select 'Generate a new client secret.'

The screenshot shows the GitHub developer settings page after creating the application. It displays the Client ID (0v23liRchW) and a note about generating a new client secret.

You can list your application in the [GitHub Marketplace](#) so that other users can discover it.

List this application in the Marketplace

0 users

Revoke all user tokens

Client ID

0v23liRchW

Client secrets

Generate a new client secret

You need a client secret to authenticate as the application to the API.

Using the ClientID and Client secret, we will store these two values in a .env file.

The image shows two screenshots. The top screenshot is from the GitHub 'Client secrets' page, which displays three client secrets. Each entry includes a key icon, the secret ID (prefixed with a checkmark or asterisks), the date added ('Added now by [redacted]'), and the status ('Never used'). The bottom screenshot is a terminal window showing a .env file with two entries: GITHUB_CLIENT_ID and GITHUB_CLIENT_SECRET, both with their values redacted.

```
JS app.js .env
.env
1 GITHUB_CLIENT_ID=0v23liRcHwY [REDACTED]
2 GITHUB_CLIENT_SECRET=3b38e6d754058a9664afcb5a6fec9 [REDACTED]
```

Install the necessary libraries: **npm install express passport passport-github2 express-session**

Use the following code in **app.js**:

The libraries and configuration parameters:

```
require('dotenv').config(); 6.3k (gzipped: 2.8k)
const express = require('express');
const session = require('express-session'); 21.8k (gzipped: 7.5k)
const passport = require('passport'); 9.5k (gzipped: 2.9k)
const GitHubStrategy = require('passport-github2').Strategy; 26.9k (gzipped: 8.4k)

const app = express();

const GITHUB_CLIENT_ID = process.env.GITHUB_CLIENT_ID;
const GITHUB_CLIENT_SECRET = process.env.GITHUB_CLIENT_SECRET;

passport.use(
  new GitHubStrategy(
    {
      clientID: GITHUB_CLIENT_ID,
      clientSecret: GITHUB_CLIENT_SECRET,
      callbackURL: 'http://localhost:3000/auth/github/callback',
    },
    (accessToken, refreshToken, profile, done) => [
      return done(null, profile);
    ]
  )
);
```

```

    passport.serializeUser((user, done) => {
      done(null, user);
    });
    passport.deserializeUser((user, done) => {
      done(null, user);
    });

    app.use(
      session({
        secret: 'secret',
        resave: false,
        saveUninitialized: true,
      })
    );
    app.use(passport.initialize());
    app.use(passport.session());

```

The endpoints for logging in and displaying user information after login:

```

app.get('/', (req, res) => {
  res.send(`^<h1>Welcome</h1><a href="/auth/github">Login with GitHub</a>`^
  );
});

app.get(
  '/auth/github',
  passport.authenticate('github', { scope: ['user:email'] })
);

app.get(
  '/auth/github/callback',
  passport.authenticate('github', { failureRedirect: '/' }),
  (req, res) => {
    res.redirect('/profile');
  }
);

app.get('/profile', (req, res) => {
  if (!req.isAuthenticated()) {
    return res.redirect('/');
  }
  res.send(`^<h1>Profile</h1><pre>${JSON.stringify(req.user, null, 2)}</pre>`^
  );
});

```

```

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});

```

Once everything is complete, go to **localhost:3000** to log in through **GitHub**. Some information will be returned when a user logs in through **GitHub**.



A screenshot of a web browser window titled "localhost:3000/profile". The page displays a JSON object under the heading "Profile". The JSON structure is as follows:

```
{  
  "id": "",  
  "nodeId": "",  
  "display_name": "",  
  "username": "",  
  "profile": "",  
  "photos": [  
    {  
      "value": "https://avatars.githubusercontent.com/u/192225896?v=4"  
    }  
  ],  
  "provider": "github",  
  "_raw": "({\"login\":\"192225896\", \"node_id\":\"U_kgDOG3UiA\", \"avatar_url\": \"https://avatars.githubusercontent.com/u/192225896?v=4\", \"gravatar_id\": \"\", \"url\": \"https://api.github.com/users/192225896\", \"html_url\": \"https://github.com/192225896\", \"followers_url\": \"https://api.github.com/users/192225896/followers\", \"following_url\": \"https://api.github.com/users/192225896/following{/target}\", \"gists_url\": \"https://api.github.com/users/192225896/gists{/gist_id}\", \"starred_url\": \"https://api.github.com/users/192225896/starred{/owner}\", \"subscriptions_url\": \"https://api.github.com/users/192225896/subscriptions\", \"organizations_url\": \"https://api.github.com/users/192225896/orgs\", \"repos_url\": \"https://api.github.com/users/192225896/repos\", \"events_url\": \"https://api.github.com/users/192225896/events{/privacy}\", \"received_events_url\": \"https://api.github.com/users/192225896/received_events\", \"type\": \"User\", \"user_view_type\": \"public\", \"site_admin\": false, \"name\": null, \"company\": null, \"blog\": \"\", \"location\": null, \"email\": null, \"hireable\": null, \"bio\": null, \"twitter_username\": null, \"notification_email\": null, \"public_repos\": 0, \"public_gists\": 0, \"followers\": 0, \"following\": 0, \"created_at\": \"2024-12-19T13:47:01Z\", \"updated_at\": \"2024-12-19T13:47:08Z\"}  
},  
}
```

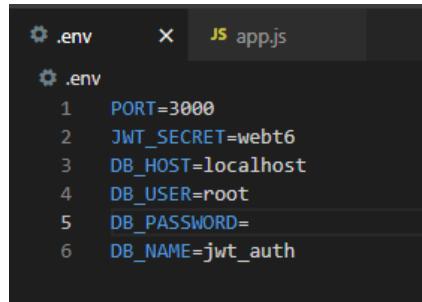
The JSON object contains several fields, including "id", "nodeId", "display_name", "username", "profile", "photos", "provider", and various URLs and statistics. The "photos" field contains a single URL for a GitHub user's profile picture. The "provider" field is set to "github". The "raw" JSON string is also displayed at the bottom of the object.

2. Json Web Token (JWT)

Next, we will learn about the mechanism of JWT.

First, install the necessary libraries.

Create a `.env` file to store environment variables.



The image shows a terminal window with two tabs: '.env' and 'app.js'. The '.env' tab contains the following environment variables:

```
PORT=3000
JWT_SECRET=webt6
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=
DB_NAME=jwt_auth
```

Use the following code in `app.js`:

The libraries needed to import and the configuration to connect to MySQL.

```
require('dotenv').config(); 6.3k (gzipped: 2.8k)
const express = require('express');
const bodyParser = require('body-parser'); 486.8k (gzipped: 212.1k)
const bcrypt = require('bcryptjs'); 21.6k (gzipped: 9.8k)
const jwt = require('jsonwebtoken'); 53.2k (gzipped: 15.8k)
const mysql = require('mysql2'); 781.9k (gzipped: 344.3k)

const app = express();
app.use(bodyParser.json());

const PORT = process.env.PORT || 3000;
const JWT_SECRET = process.env.JWT_SECRET || 'your_jwt_secret';
const DB_CONFIG = {
  host: process.env.DB_HOST || 'localhost',
  user: process.env.DB_USER || 'root',
  password: process.env.DB_PASSWORD || '',
  database: process.env.DB_NAME || 'jwt_auth',
};

const db = mysql.createConnection(DB_CONFIG);

db.connect((err) => {
  if (err) {
    console.error('Database connection failed:', err.stack);
    return;
  }
  console.log('Connected to MySQL database.');
});

const query = (sql, values) =>
  new Promise((resolve, reject) => {
    db.query(sql, values, (err, results) => {
      if (err) return reject(err);
      resolve(results);
    });
  });
}
```

SQL:

```
CREATE DATABASE jwt_auth;  
USE jwt_auth;  
CREATE TABLE  
users  
( id INT AUTO_INCREMENT PRIMARY KEY,  
username VARCHAR(50) UNIQUE NOT NULL,  
password VARCHAR(255) NOT NULL );
```

```
CREATE TABLE tokens  
( id INT AUTO_INCREMENT PRIMARY KEY,  
user_id INT NOT NULL,  
token VARCHAR(500) NOT NULL,  
FOREIGN KEY (user_id) REFERENCES users(id) );
```

Create an endpoint for user registration, receiving a username and password, then saving them to the database.

```
app.post('/register', async (req, res) => {  
  const { username, password } = req.body;  
  
  if (!username || !password) {  
    return res.status(400).json({ message: 'Username and password are required' });  
  }  
  
  try {  
    const hashedPassword = await bcrypt.hash(password, 10);  
    await query('INSERT INTO users (username, password) VALUES (?, ?)', [  
      username,  
      hashedPassword,  
    ]);  
    res.status(201).json({ message: 'User registered successfully' });  
  } catch (err) {  
    console.error(err);  
    res.status(500).json({ message: 'Error registering user' });  
  }  
});
```

Endpoint:

The screenshot shows the Postman interface. The URL is set to `http://localhost:3000/register`. The method is `POST`. The `Body` tab is selected, showing the following JSON payload:

```
1 "username": "testuser",
2 "password": "password123"
```

The response status is `201 Created`, and the message is `"User registered successfully"`.

Database:

id	username	password
1	testuser	\$2a\$10\$fmHZGNt9JYXIEDU7WN0uEejMBDFPkRTy/rkf8UotyWs...

Create an endpoint for the login process. If the user provides correct information, generate a JWT for the user and save it in the tokens table in the database.

```
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  if (!username || !password) {
    return res.status(400).json({ message: 'Username and password are required' });
  }

  try {
    const users = await query('SELECT * FROM users WHERE username = ?', [username]);
    const user = users[0];

    if (!user) {
      return res.status(401).json({ message: 'Invalid credentials' });
    }

    const isPasswordValid = await bcrypt.compare(password, user.password);

    if (!isPasswordValid) {
      return res.status(401).json({ message: 'Invalid credentials' });
    }

    const token = jwt.sign({ id: user.id }, JWT_SECRET, { expiresIn: '1h' });
    await query('INSERT INTO tokens (user_id, token) VALUES (?, ?)', [user.id, token]);

    res.status(200).json({ message: 'Login successful', token });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: 'Error logging in' });
  }
});
```

Endpoint:

The screenshot shows a Postman interface with the following details:

- Method: POST
- URL: http://localhost:3000/login
- Body (JSON):

```
1 {  
2   "username": "testuser",  
3   "password": "password123"  
4 }
```
- Response status: 200 OK (126 ms, 414 B)
- Response body (Pretty):

```
1 {  
2   "message": "Login successful",  
3   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
4         eyJpZCI6MSwiZWQiOjE2OTU8LC1leHAiOjE3MzQ2MjA1NTR9.  
5         595nPAc0uBCeSp_q1PFXX7G0-xaF9YfpQtRpRmmcPyNw"  
6 }
```

Database:

	id	user_id	token
1	1		eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwi...

Create an endpoint for user authentication and logout.

```
app.get('/verify', async (req, res) => {
  const { token } = req.query;

  if (!token) {
    return res.status(400).json({ message: 'Token is required' });
  }

  try {
    const decoded = jwt.verify(token, JWT_SECRET);
    res.status(200).json({ message: 'Token is valid', decoded });
  } catch (err) {
    res.status(401).json({ message: 'Invalid or expired token' });
  }
});

app.post('/logout', async (req, res) => {
  const { token } = req.body;

  if (!token) {
    return res.status(400).json({ message: 'Token is required' });
  }

  try {
    await query('DELETE FROM tokens WHERE token = ?', [token]);
    res.status(200).json({ message: 'Logout successful' });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: 'Error logging out' });
  }
};

// Start the server
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

Endpoint:

The screenshot shows two API requests in Postman:

GET Request to Verify Token:

- URL: `http://localhost:3000/verify?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiWF0joxNzMON...`
- Method: GET
- Body (raw JSON):

```
1 {  
2   "username": "testuser",  
3   "password": "password123"  
4 }
```
- Response (Pretty):

```
1 {  
2   "message": "Token is valid",  
3   "decoded": {  
4     "id": 1,  
5     "iat": 1734616954,  
6     "exp": 1734620554  
7   }  
8 }
```
- Status: 200 OK, 10 ms, 316 B

POST Request to Logout:

- URL: `http://localhost:3000/logout`
- Method: POST
- Body (raw JSON):

```
1 {  
2   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
3   eyJpZCI6MSwiWF0joxNzMONjE2OTU0LCJleHAiOjE3MzQ2MjA1NTk9.  
4   595nPAc0uBCeSp_q1PFXK7G0-xaF9YfpQtpRmmcPyNw"  
5 }
```
- Response (Pretty):

```
1 {  
2   "message": "Logout successful"  
3 }
```
- Status: 200 OK, 9 ms, 266 B

You can verify the token on the website jwt.io. If the **JWT_SECRET** is incorrect, the token will not match the one created in the database earlier.

True JWT_SECRET

The screenshot shows the jwt.io interface. On the left, under 'Encoded', is a long string of characters: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJpZCI6MSwiaWF0IjoxNzM0NjE4MTU5LC1leHA10jE3MzQ2MjE3NT19.vpAzpNrClbOrw_yE0yD9nQr4qAGMd35MWGofFBs2IP0. On the right, under 'Decoded', is the JSON payload: { "id": 1, "iat": 1734618159, "exp": 1734621759 }. Below the payload, there is a 'VERIFY SIGNATURE' section with code for HMACSHA256 and a 'secret' input field containing 'webt6'.

The screenshot shows a Postman request to 'http://localhost:3000/login'. The 'Body' tab is selected, showing a JSON payload: { "username": "testuser", "password": "password123" }. The response status is 200 OK, and the response body is: { "message": "Login successful", "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJpZCI6MSwiaWF0IjoxNzM0NjE4MTU5LC1leHA10jE3MzQ2MjE3NT19.vpAzpNrClbOrw_yE0yD9nQr4qAGMd35MWGofFBs2IP0" }.

False JWT_SECRET:

The screenshot shows the jwt.io interface. The token is identical to the one above: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJpZCI6MSwiaWF0IjoxNzM0NjE4MTU5LC1leHA10jE3MzQ2MjE3NT19.vpAzpNrClbOrw_yE0yD9nQr4qAGMd35MWGofFBs2IP0. However, in the 'VERIFY SIGNATURE' section, the 'secret' input field contains 'wrong-code'. The rest of the interface is identical to the first screenshot.

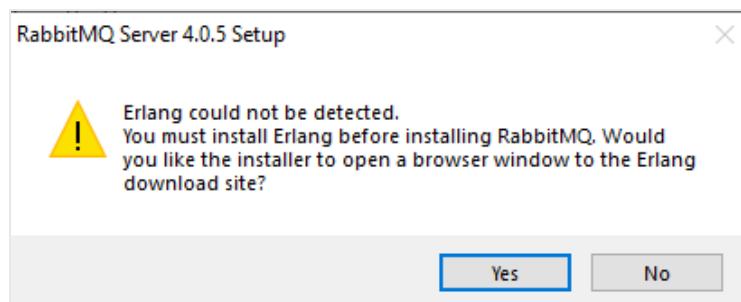
3. Message Queue:

Next, we will experiment with building a system consisting of two services and a message queue, where the message queue used here will be RabbitMQ.

To download RabbitMQ, follow this link: <https://www.rabbitmq.com/docs/install-windows>. If you are using a Mac, you can view the installation guide here: <https://www.youtube.com/watch?v=6DL3lrIlxE0>

The screenshot shows the official RabbitMQ documentation website at <https://www.rabbitmq.com/docs/install-windows>. The left sidebar has a navigation menu with sections like Introduction, Release Information, Getting Started, Install and Upgrade (which is expanded to show Erlang Version Requirements, Package Signatures, and Supported Operating Systems), and others. Under 'Supported Operating Systems', 'Windows' is selected. The main content area discusses Erlang requirements and provides a 'Direct Downloads' table. The table has three columns: Description, Download, and Signature. It lists an 'Installer for Windows systems (from GitHub)' with the file name 'rabbitmq-server-4.0.5.exe' highlighted with a red box. A 'Run RabbitMQ Windows Service' section follows, and a 'CLI Tools' section is partially visible at the bottom.

RabbitMQ requires you to install Erlang.



To download Erlang, visit this link: <https://www.erlang.org/downloads>

Download Erlang/OTP

The latest version of Erlang/OTP is [27.2](#). To install Erlang you can either build it [from source](#) or use a [pre-built package](#).

Take a look at the [Erlang/OTP 27 release description](#) to see what changes Erlang/OTP 27 brings over the previous major version.

The Erlang/OTP version scheme is described in the [Erlang/OTP Systems Principles Guide](#).

Erlang/OTP 27.2



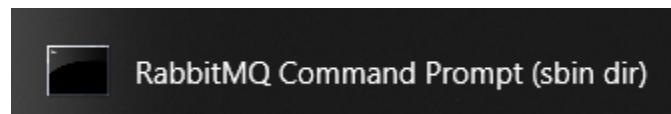
Download source

Download Windows installer ▾

Download Release notes

View documentation ▾

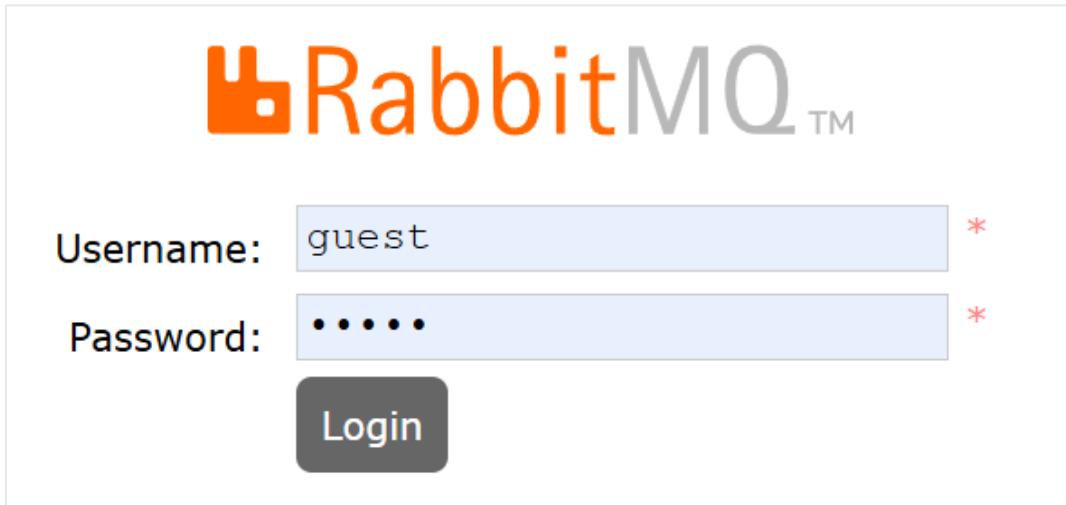
After installation, find the RabbitMQ Command Prompt and run the following commands:



```
rabbitmq-plugins enable rabbitmq_management  
rabbitmq-server
```

```
Administrator: RabbitMQ Command Prompt (sbin dir) - rabbitmq-server  
  
C:\Program Files\RabbitMQ Server\rabbitmq_server-4.0.5\sbin>rabbitmq-plugins enable rabbitmq_management  
Enabling plugins on node rabbit@LAPTOP-J0M5S05G:  
rabbitmq_management  
The following plugins have been configured:  
  rabbitmq_management  
  rabbitmq_management_agent  
  rabbitmq_web_dispatch  
Applying plugin configuration to rabbit@LAPTOP-J0M5S05G...  
Plugin configuration unchanged.  
  
C:\Program Files\RabbitMQ Server\rabbitmq_server-4.0.5\sbin>rabbitmq-server  
=INFO REPORT== 19-Dec-2024::22:16:02.992000 ===  
  alarm_handler: [set,{disk_almost_full,"C:\\\"},[]}  
2024-12-19 22:16:03.260000+07:00 [warning] <0.150.0> Using RABBITMQ_ADVANCED_CONFIG_FILE: c:/Users/THANH BINH/AppData/Roaming/RabbitMQ/advanced.config  
2024-12-19 22:16:06.614000+07:00 [notice] <0.45.0> Application syslog exited with reason: stopped  
2024-12-19 22:16:06.614000+07:00 [notice] <0.213.0> Logging: switching to configured handler(s); following messages may not be visible in this log output  
## ##      RabbitMQ 4.0.5  
## ##  
##### Copyright (c) 2007-2024 Broadcom Inc and/or its subsidiaries  
##### ##  
##### ## Licensed under the MPL 2.0. Website: https://rabbitmq.com  
  
Erlang:    27.2 [jit]  
TLS Library: OpenSSL - OpenSSL 3.1.0 14 Mar 2023  
Release series support status: see https://www.rabbitmq.com/release-information  
  
Doc guides: https://www.rabbitmq.com/docs  
Support: https://www.rabbitmq.com/docs/contact  
Tutorials: https://www.rabbitmq.com/tutorials  
Monitoring: https://www.rabbitmq.com/docs/monitoring  
Upgrading: https://www.rabbitmq.com/docs/upgrade  
  
Logs: <stdout>  
      c:/Users/THANH BINH/AppData/Roaming/RabbitMQ/log/rabbit@LAPTOP-J0M5S05G.log  
  
Config file(s): c:/Users/THANH BINH/AppData/Roaming/RabbitMQ/advanced.config  
  
Starting broker... completed with 3 plugins.
```

Once successful, go to **localhost:15672** and log in with the username and password, both set to '**guest**'.



We will create two separate services: one to send messages, called Producer, and one to receive and process messages, called Consumer.

For the Producer, run the following command to install the necessary libraries:

The libraries used and connection initialization.

```
const amqp = require('amqplib'); 144.4k (gzipped: 30.1k)
const express = require('express');
const bodyParser = require('body-parser'); 486.8k (gzipped: 212.1k)
const { faker } = require('@faker-js/faker'); 2.7M (gzipped: 875.1k)

const app = express();
app.use(bodyParser.json());

const RABBITMQ_URL = 'amqp://localhost';
const QUEUE = 'messages';

let connection;
let channel;
```

Generate fake data and initialize the connection to RabbitMQ

```
function generateFakeMessage() {
  return {
    id: faker.string.uuid(),
    name: faker.person.fullName(),
    email: faker.internet.email(),
    content: faker.lorem.sentence(),
    timestamp: new Date().toISOString(),
  };
}
```

```
async function initRabbitMQ() {
  try {
    console.log('Connecting to RabbitMQ...');
    connection = await amqp.connect(RABBITMQ_URL);
    channel = await connection.createChannel();
    await channel.assertQueue(QUEUE);
    console.log('RabbitMQ connected and queue asserted.');
  } catch (error) {
    console.error('Error initializing RabbitMQ:', error);
    process.exit(1);
  }
}
```

A function to send messages based on prepared parameters and automatically send messages every 5 seconds.

```
async function sendMessage(message) {
  try {
    if (!channel) {
      throw new Error('Channel is not initialized');
    }
    channel.sendToQueue(QUEUE, Buffer.from(JSON.stringify(message)));
    console.log(`Message sent: ${JSON.stringify(message)}`);
  } catch (error) {
    console.error('Error sending message:', error);
  }
}

function startAutoProducer() {
  console.log('Starting auto-producer...');
  setInterval(async () => {
    const fakeMessage = generateFakeMessage();
    await sendMessage(fakeMessage);
  }, 5000);
}
```

A function to close the connection and configure the running port.

```

async function closeRabbitMQ() {
  try {
    console.log('Closing RabbitMQ connection...');
    if (channel) await channel.close();
    if (connection) await connection.close();
    console.log('RabbitMQ connection closed.');
  } catch (error) {
    console.error('Error closing RabbitMQ connection:', error);
  }
}

process.on('SIGINT', async () => {
  await closeRabbitMQ();
  process.exit(0);
});

const PORT = 3000;
app.listen(PORT, async () => {
  console.log(`Producer running on http://localhost:${PORT}`);
  await initRabbitMQ();
  startAutoProducer();
});

```

For the Consumer, which receives and stores messages, the database used this time will be MongoDB. Run the following command to install the necessary libraries:

The libraries used, configuration parameters, and data storage structure.

```

const amqp = require('amqplib'); 144.4k (gzipped: 30.1k)
const mongoose = require('mongoose'); 886k (gzipped: 237k)

const RABBITMQ_URL = 'amqp://localhost';
const QUEUE = 'messages';
const MONGO_URI = 'mongodb://localhost:27017/rabbitmq_example';

mongoose
  .connect(MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('Connected to MongoDB'))
  .catch((err) => console.error('MongoDB connection error:', err));

// Define a Message Schema
const messageSchema = new mongoose.Schema({
  id: { type: String, required: true },
  name: { type: String, required: true },
  email: { type: String, required: true },
  content: { type: String, required: true },
  metadata: { type: Object, required: false },
  timestamp: { type: Date, required: true },
});

const Message = mongoose.model('Message', messageSchema);

```

A function to handle received messages.

```
async function consumeMessages() {
  const connection = await amqp.connect(RABBITMQ_URL);
  const channel = await connection.createChannel();
  await channel.assertQueue(QUEUE);

  console.log(`Waiting for messages in ${QUEUE}...`);

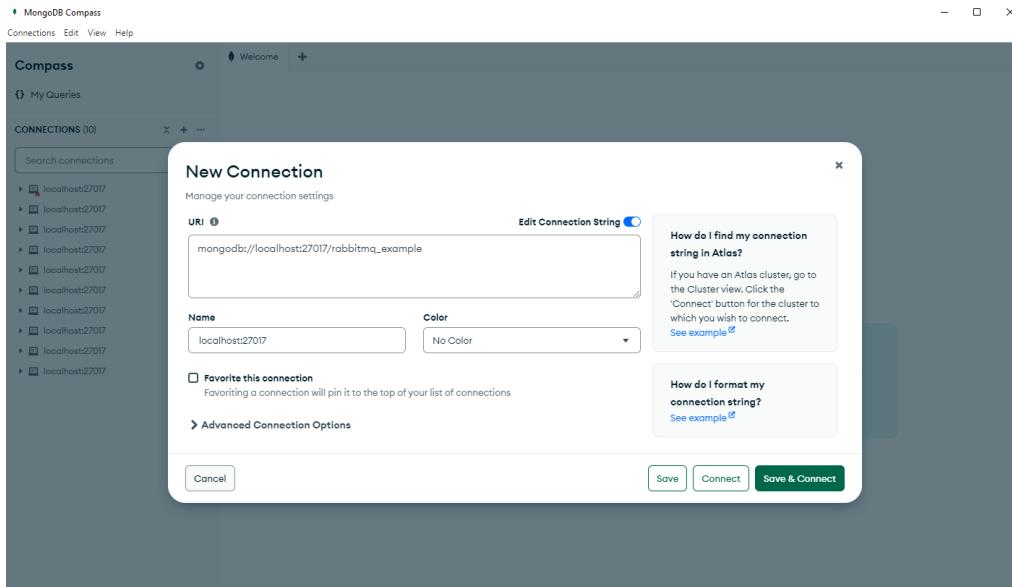
  channel.consume(QUEUE, async (msg) => {
    if (msg !== null) {
      const messageContent = JSON.parse(msg.content.toString());
      console.log('Message received:', messageContent);

      try {
        const savedMessage = await Message.create({
          ...messageContent,
          metadata: { source: 'RabbitMQ', priority: 'High' },
        });
        console.log('Message saved to MongoDB:', savedMessage);
      } catch (err) {
        console.error('Error saving message to MongoDB:', err);
      }
    }

    channel.ack(msg);
  });
}

consumeMessages().catch(console.error);
```

For the Consumer, which receives and stores messages, the database used this time will be MongoDB.



Once everything is complete, start the services in order:

Producer:

```

Producer running on http://localhost:3000
Connecting to RabbitMQ...
RabbitMQ connected and queue asserted.
Starting auto-producer...
Message sent: {"id":"72ae241e-a622-47d6-87d6-ffbdd3af1453","name":"Lester Dare-Gutkowski V",
Message sent: {"id":"5742fb83-4fff-45fc-ab0c-815e67bceeb1","name":"Claude Morar Jr.", "email": "claudemorarjr@example.com",
Message sent: {"id":"27875832-5c11-4317-8959-742adb346752", "name": "Marianne Jacobson", "email": "marianne.jacobson@example.com",
Message sent: {"id": "5cbdc1df-02e-46b0-b2ba-08e23d695544", "name": "Allison Lakin", "email": "allison.lakin@example.com",
Message sent: {"id": "416ffc66-4c8a-4f97-9a40-0e9cc98e87d1", "name": "Bobbie Huels-Zieme", "email": "bobbie.huels-zieme@example.com",
Message sent: {"id": "708ea374-7923-432f-97b8-b51e6e4db422", "name": "Ms. Sara McGlynn IV", "email": "sara.mcglynn.iv@example.com",
Message sent: {"id": "2faedbff-e616-481d-838f-148e6640c24b", "name": "Cristina Hauck", "email": "cristina.hauck@example.com"

```

Consumer:

```

Message received: {"id": "d68ebae3-b0ec-4020-b317-4cc67f3fe3da", "name": "Dominick Fay-DuBuque", "email": "Stella_Bernier@bayer34.com", "content": "I am a producer and I am sending you a message.", "timestamp": "2024-12-19T15:46:46.986Z", "v": 0}
Message saved to MongoDB: {
  message: {"id": "d68ebae3-b0ec-4020-b317-4cc67f3fe3da", "name": "Dominick Fay-DuBuque", "email": "Stella_Bernier@bayer34.com", "content": "I am a producer and I am sending you a message.", "timestamp": "2024-12-19T15:46:46.986Z", "v": 0},
  _id: new ObjectId('67643fe6a4cff3a57cca3e50'),
  timestamp: 2024-12-19T15:46:46.986Z,
  __v: 0
}
Message received: {"id": "2cd2c02b-e4c9-41a6-94c5-200c7284427a", "name": "Wesley Farrell", "email": "Telly_Schmeler@bayer34.com", "content": "I am a consumer and I am receiving your message.", "timestamp": "2024-12-19T15:46:51.987Z", "v": 0}
Message saved to MongoDB: {
  message: {"id": "2cd2c02b-e4c9-41a6-94c5-200c7284427a", "name": "Wesley Farrell", "email": "Telly_Schmeler@bayer34.com", "content": "I am a consumer and I am receiving your message.", "timestamp": "2024-12-19T15:46:51.987Z", "v": 0},
  _id: new ObjectId('67643feba4cff3a57cca3e52'),
  timestamp: 2024-12-19T15:46:51.990Z,
  __v: 0
}
Message received: {"id": "d0cfb00b-5527-4845-bd3b-73464d6d65ee", "name": "Miss Latoya Osinski PhD", "email": "Ray.Bayer34@gmail.com", "content": "I am a consumer and I am receiving your message.", "timestamp": "2024-12-19T15:46:57.005Z", "v": 0}
Message saved to MongoDB: {
  message: {"id": "d0cfb00b-5527-4845-bd3b-73464d6d65ee", "name": "Miss Latoya Osinski PhD", "email": "Ray.Bayer34@gmail.com", "content": "I am a consumer and I am receiving your message.", "timestamp": "2024-12-19T15:46:57.005Z", "v": 0},
  _id: new ObjectId('67643ff1a4cff3a57cca3e54'),
  timestamp: 2024-12-19T15:46:57.005Z,
  __v: 0
}

```

MongoDB:

```

{
  "_id": ObjectId('676440f2950127aaddfdc243'),
  "id": "a637c9f8-8fc3-4a78-91a9-329b18f9d3a5",
  "name": "Eric Maggio Jr.",
  "email": "Lera.Borer33@gmail.com",
  "content": "Communis cenaculum cohors tot deprecator tubineus amissio voluptate.",
  "metadata": Object,
  "timestamp": 2024-12-19T15:49:47.355+00:00,
  "__v": 0
}

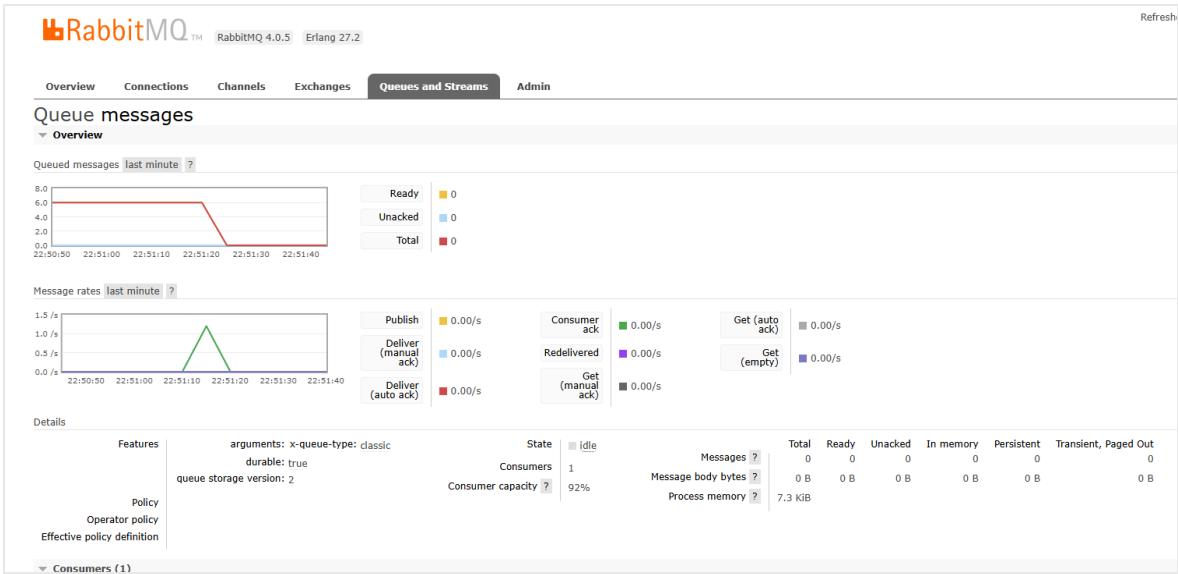
{
  "_id": ObjectId('676440f2950127aaddfdc244'),
  "id": "f4f0989e-1b3a-4c73-9923-cddfbb005f53",
  "name": "Raymond Halvorson",
  "email": "Justine_Weissnat@yahoo.com",
  "content": "Cimentarius magni occaecati surgo.",
  "metadata": Object,
  "timestamp": 2024-12-19T15:49:52.360+00:00,
  "__v": 0
}

{
  "_id": ObjectId('676440f2950127aaddfdc245'),
  "id": "71c9ab28-e60b-4897-8846-29dcad5f653",
  "name": "Donna Lebsack",
  "email": "Allison4@gmail.com",
  "content": "Distinctio crustulum sequi.",
  "metadata": Object,
  "timestamp": 2024-12-19T15:49:57.374+00:00,
  "__v": 0
}

```

At this point, data will be sent from the Producer to RabbitMQ and then to the Consumer. The Consumer will receive the data, process it, and store it in MongoDB.

Some information related to the Message Queue can be tracked in the RabbitMQ interface.



Exercise:

1. Reuse Task 2 to store additional user information in JWT and the database, including login time and login address.
2. Reuse Task 2 to create user roles, including admin and user. Add role information to JWT and the database, create an /admin endpoint, and check user roles to block users from accessing the /admin endpoint.
3. Reuse Task 3 to create a system with 2 producers and 1 consumer. The 2 producers send messages through RabbitMQ, and 1 consumer receives messages from RabbitMQ.