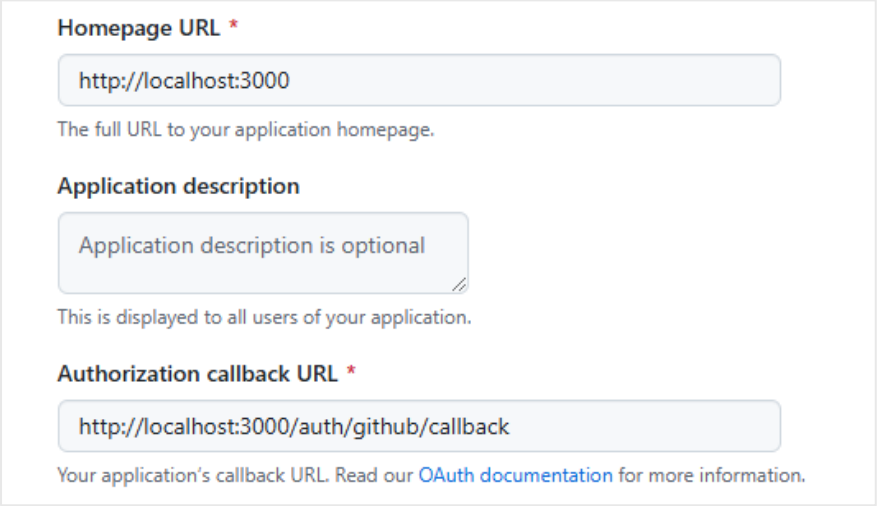
LAB 6: OAuth2, Json Web Token and Message Queue

1. OAuth2 with Github

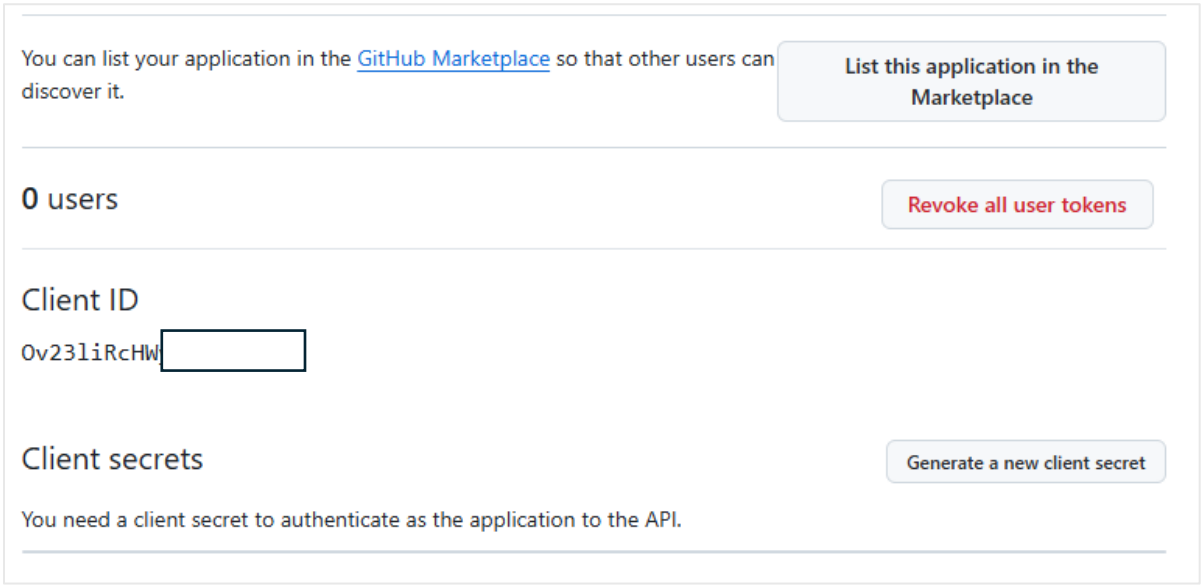
In this exercise, we will implement the login method using OAuth2 with GitHub

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.

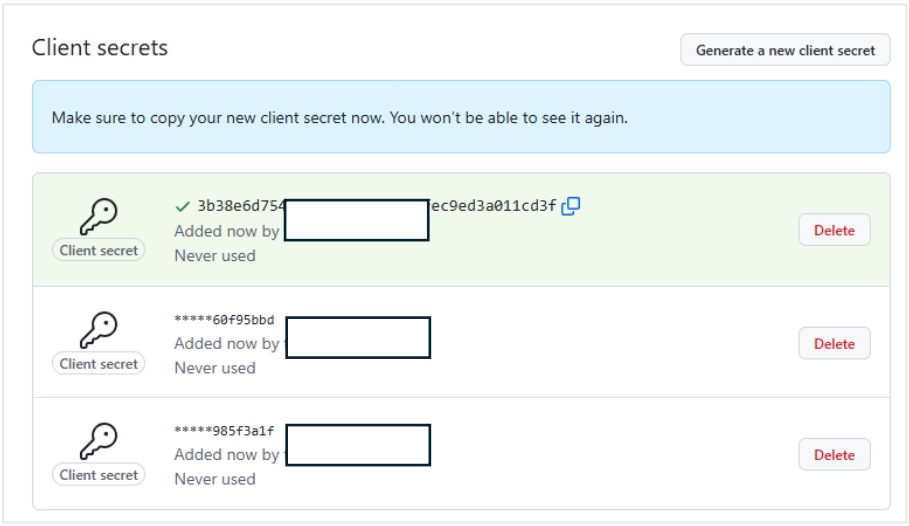
https://github.com/settings/developers

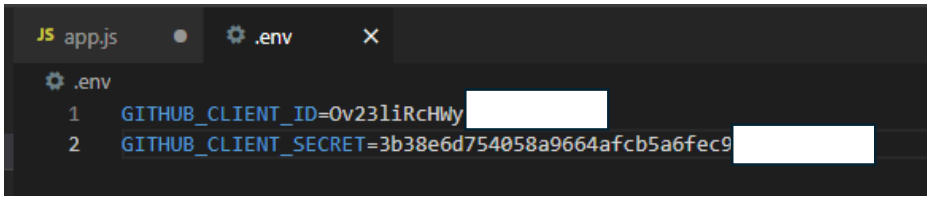


After successfully creating the project, we will have a ClientID. To create Client secrets, select 'Generate a new client secret.'



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





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');
  });
```

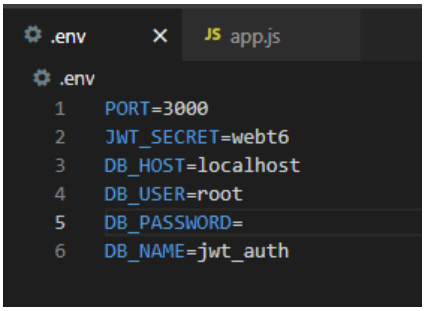
Once everything is complete, go to **localhost:3000** to log in through **GitHub**. Some information will be returned when a user logs in through **GitHub**.

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.



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,

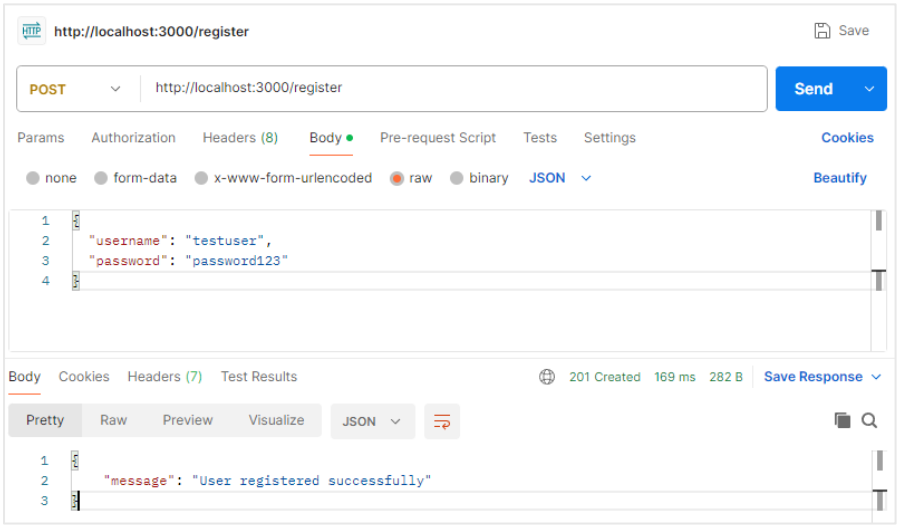
token VARCHAR(500) NOT NULL,

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:



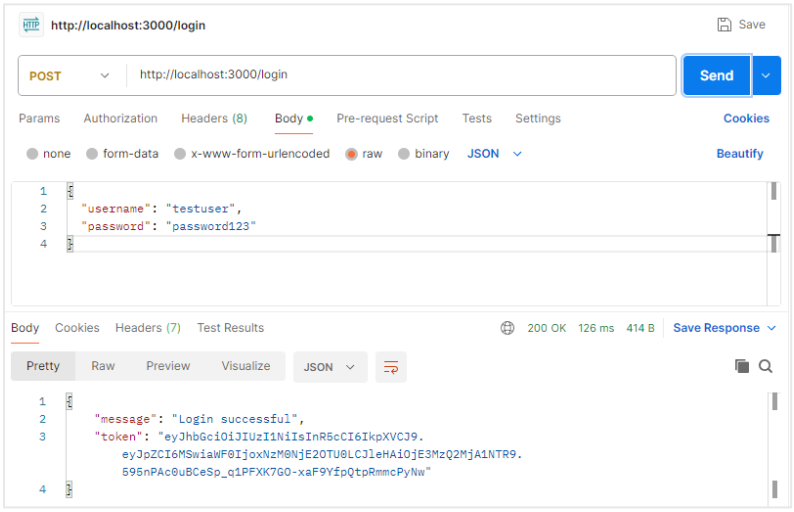
Database:

id	username	password
1	testuser	$2a$10$fMHZGNt9JVXlEDU7WN0uEejMBDFPkRTy/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:



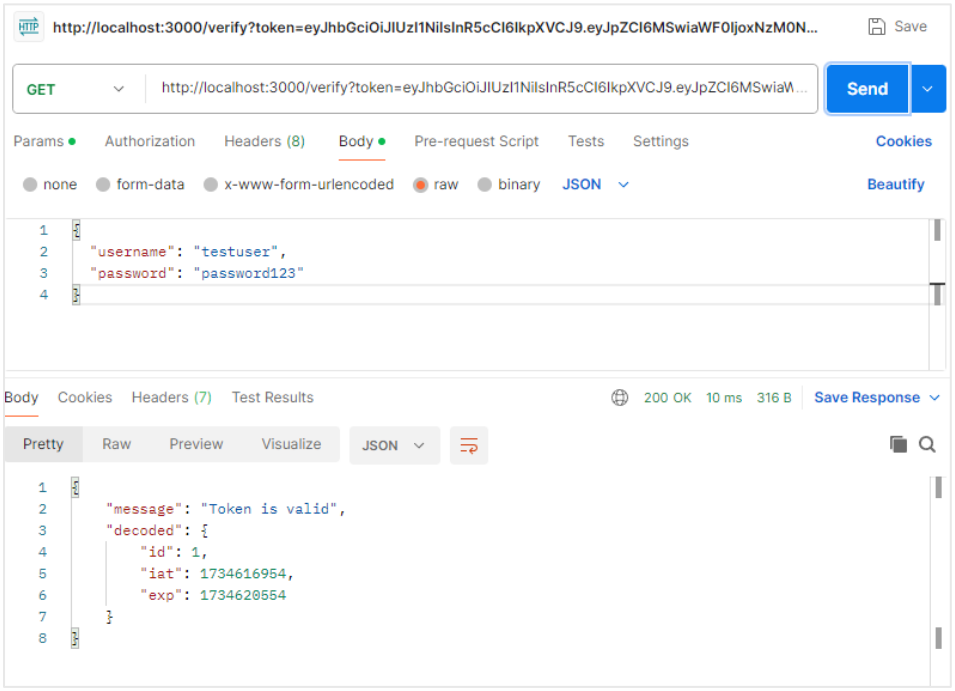
Database:

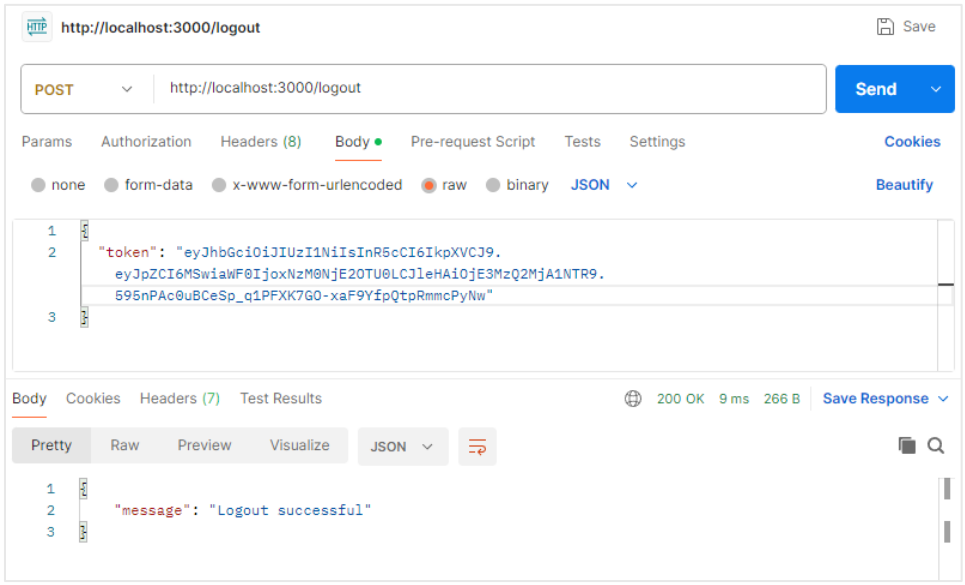
id	user_id	token
1	1	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwia...

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}`);
});
```

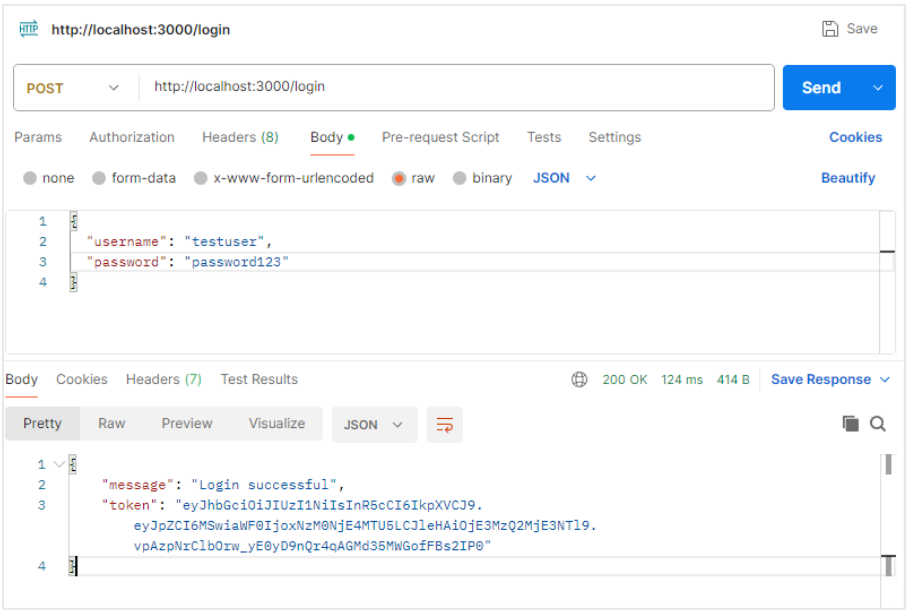
Endpoint:



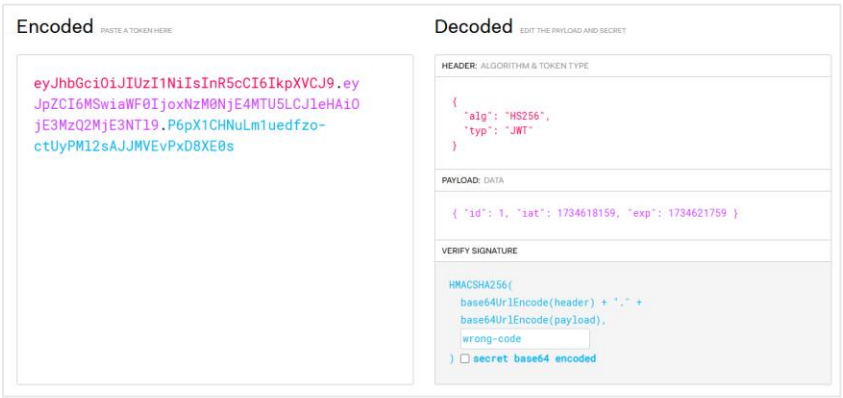


You can verify the token on the website **jwt.io**. If the **JWT_SECRET** is incorrect, the token will not match the one created in the database earlier.

True **JWT_SECRET**



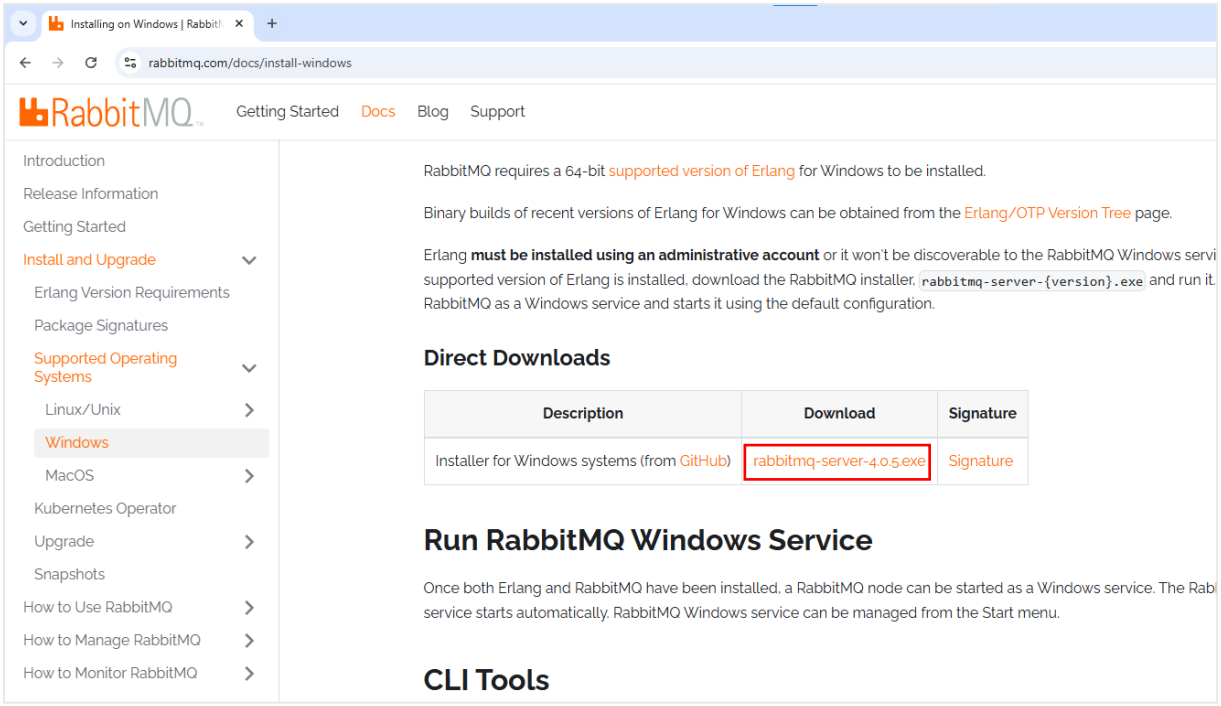
False **JWT_SECRET:**



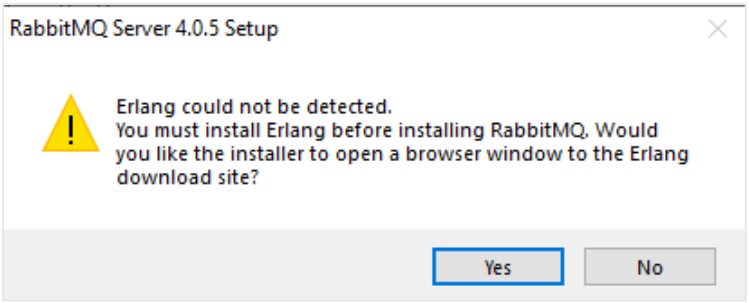
3. Message Queue:

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.

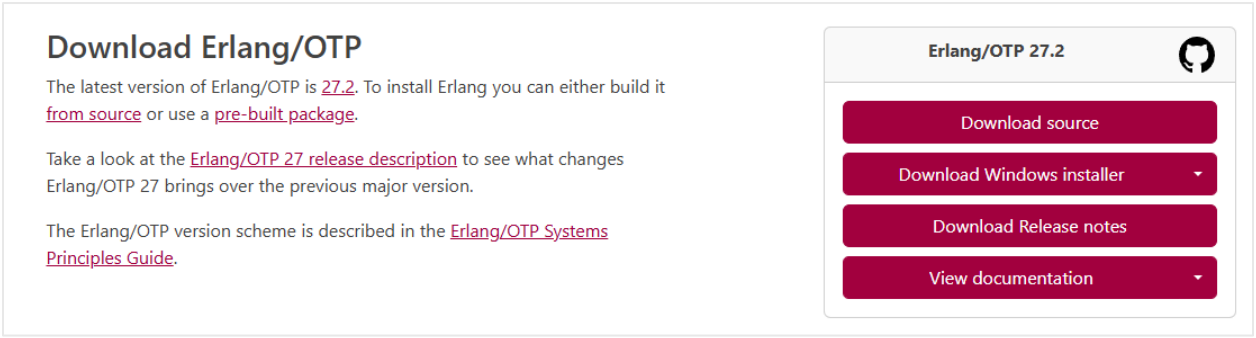
To download RabbitMQ, follow this link: *https://www.rabbitmq.com/docs/install-windows*. If you are using a Mac, you can view the installation guide here: *https://www.youtube.com/watch?v=6DL3lrI1xE0*



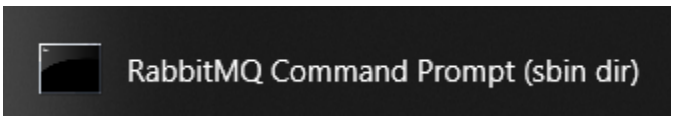
RabbitMQ requires you to install Erlang.



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

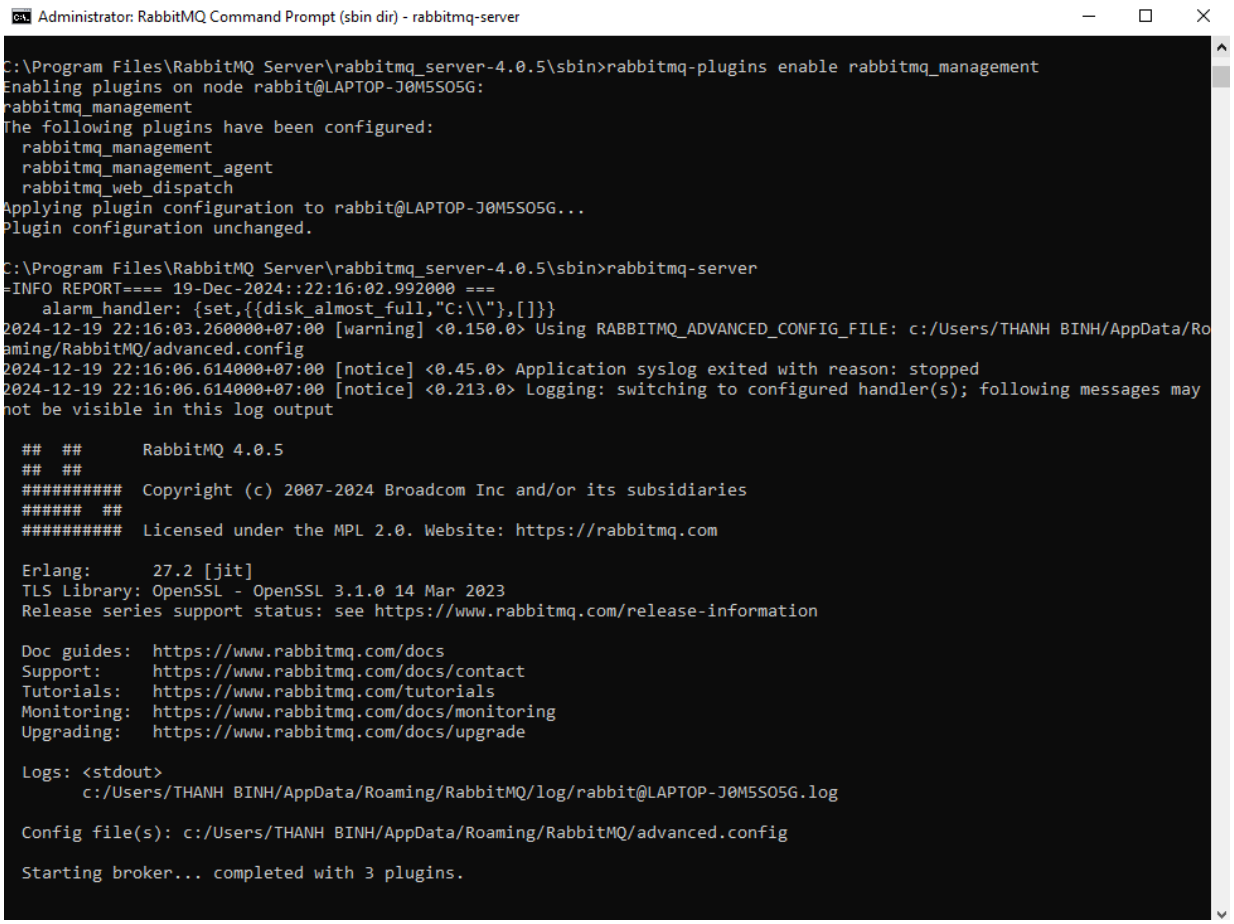


After installation, find the RabbitMQ Command Prompt and run the following commands:

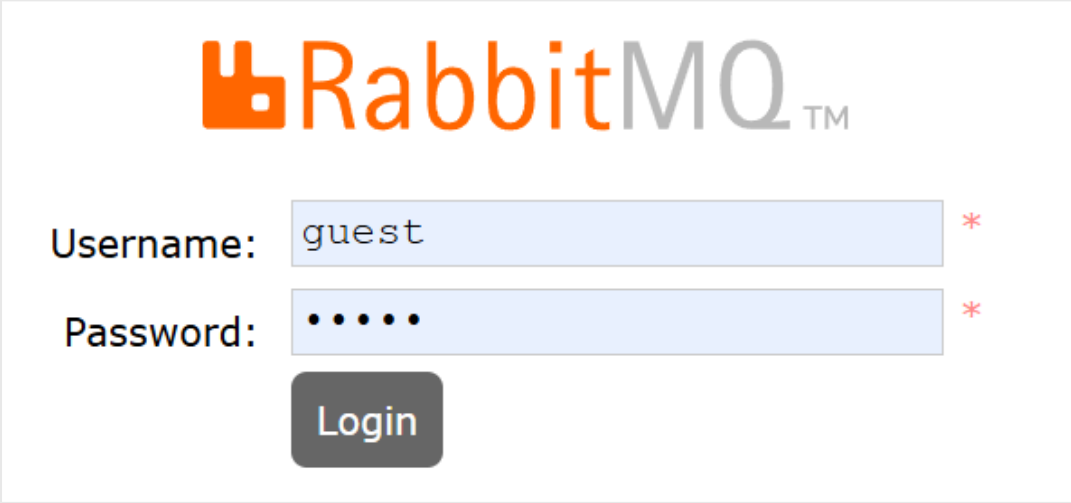


rabbitmq-plugins enable rabbitmq_management

rabbitmq-server



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:

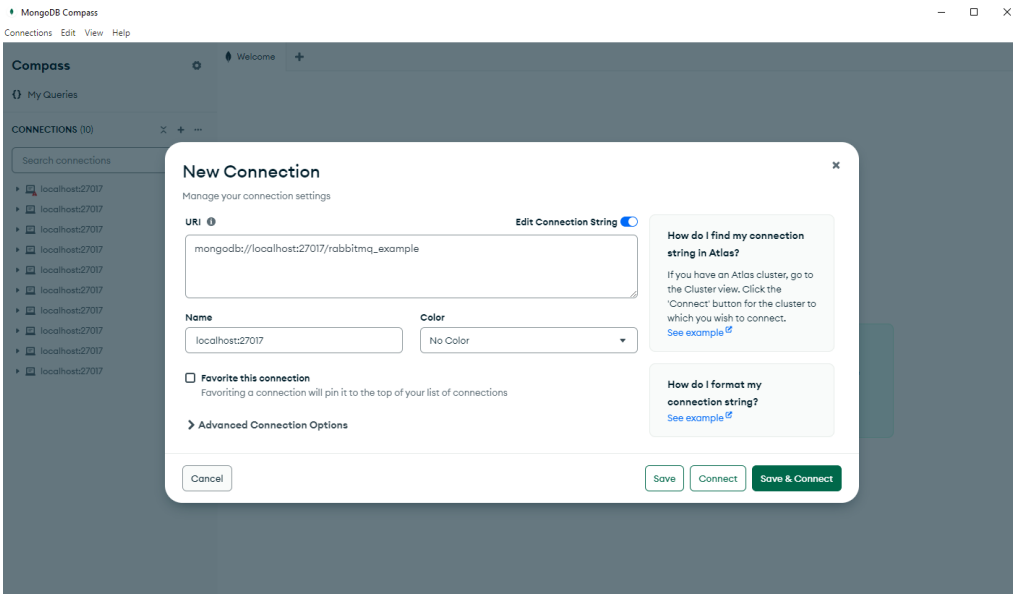
The libraries used, configuration parameters, and data storage structure.

```
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"
Message sent: {"id":"27875832-5c11-4317-8959-742adb346752","name":"Marianne Jacobson","email
Message sent: {"id":"5cbdc1df-0c2e-46b0-b2ba-08e23d695544","name":"Allison Lakin","email":"
Message sent: {"id":"416ffc66-4c8a-4f97-9a40-0e9cc98e87d1","name":"Bobbie Huels-Zieme","emai
Message sent: {"id":"708ea374-7923-432f-97b8-b51e6e4db422","name":"Ms. Sara McGlynn IV","ema
Message sent: {"id":"2faedbff-e616-481d-838f-148e6640c24b","name":"Cristina Hauck","email":"
```

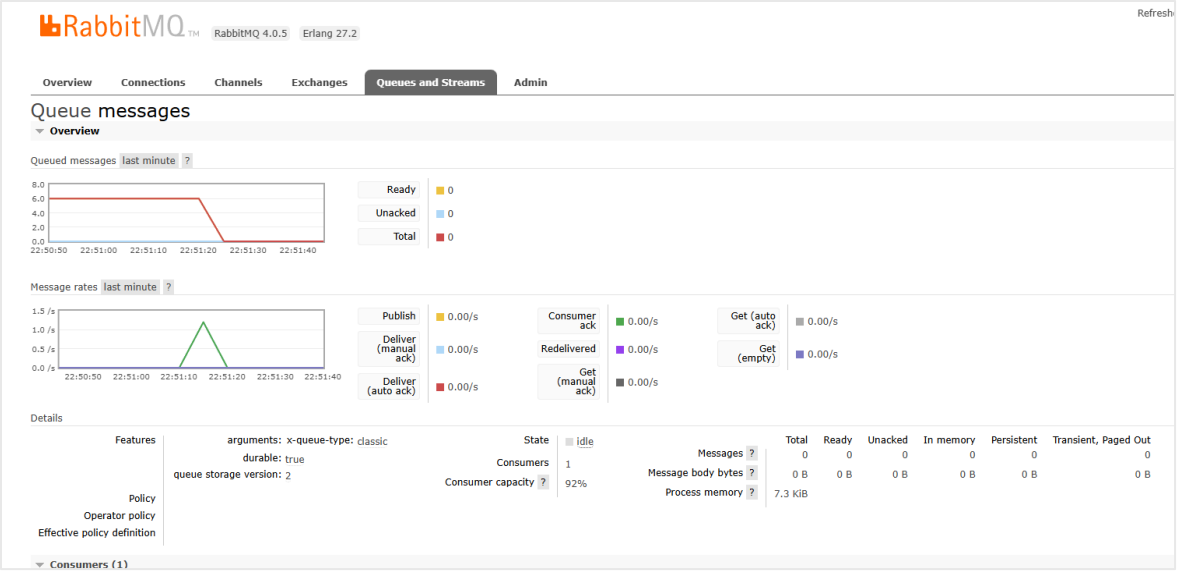
Consumer:

```
Message received: {"id":"d68ebae3-b0ec-4020-b317-4cc67f3fe3da","name":"Dominick Fay-DuBuque","email":"Stella_B
Message saved to MongoDB: {
  message: '{"id":"d68ebae3-b0ec-4020-b317-4cc67f3fe3da","name":"Dominick Fay-DuBuque","email":"Stella_Bernier
  _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
19T15:46:51.987Z"}
Message saved to MongoDB: {
  message: '{"id":"2cd2c02b-e4c9-41a6-94c5-200c7284427a","name":"Wesley Farrell","email":"Telly_Schmeler33@gma
46:51.987Z"}',
  _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.B
Message saved to MongoDB: {
  message: '{"id":"d0cfb00b-5527-4845-bd3b-73464d6d65ee","name":"Miss Latoya Osinski PhD","email":"Ray.Bayer34
  _id: new ObjectId('67643ff1a4cff3a57cca3e54'),
  timestamp: 2024-12-19T15:46:57.005Z,
  __v: 0
```

MongoDB:

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.