Creating Secured REST APIs

Hatnean Maria Cristina

Group 258-1

The Representational State Transfer Application Programming Interface (REST API) enables different systems to communicate and exchange data over the internet. REST APIs use a JavaScript Object Notation (JSON) data format with standard HTTP communication methods like: GET, POST, PUT, DELETE.

By using these common verbs and status codes, REST APIs provide a uniform interface across diverse applications, supporting all content types.

Since the REST API architecture enables stateless communication between the client, like a user, and the server, it doesn’t store information about current sessions or connections. The application handles: access, resource manipulation, session management.

Since REST APIs control how data travels between applications, attackers target them to gain unauthorized access to sensitive data.

One way of protecting your applications is by using access control. To protect your REST API, you must verify all clients that request access to the API by implementing:

* Authentication: verifying that the client is who it say it is
* Authorization: ensuring it can take the action that it wants to take

To do that, we can use JSON Web Tokens (JWTs). JWTs are digitally signed JSON objects that can be verified and trusted, enabling:

* Authentication: using public/private key pairs so that the signatures certified the party holding the private key is who it says it is
* Authorization: passing an access token in every request after the client successfully logs in

**Implementing secured REST APIs in an app:**

For this example, we will use an app with two services: an *authentication* service and an example service that we will call *service1*. In authentication we will generate a token for the user, and service1 will verify the token before answering the request.

In the authentication service, in the post function, we want to generate a jwt that will be passed as a json.

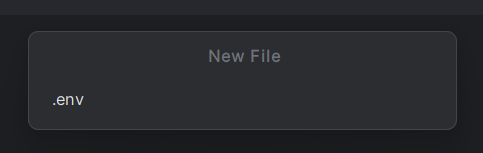
We will create a post method, and the url for accessing it is specified as a string '/auth/login'. For exemple purposes, we will verify what we get with a hardcoded user and password. If it is correct, we generate a token using the function *jwt.sign*.

If the credentials are wrong, we send a message. Also, if there is an error while creating the token, we send a server error message.

Our function should look like this:

app.post('/auth/login', async (req, res) => {  
 const { email, password } = req.body;  
  
 try {  
 if (email === 'admin' && password === 'admin') {  
 const token = *jwt*.sign({ id: 'user.\_id' }, *process*.env.JWT\_KEY, { expiresIn: '1h' });  
  
 res.json({ message: 'Login successful', token });  
 } else return res.status(401).json({ message: 'Invalid credentials' });  
 } catch (err) {  
 *console*.error(err);  
 res.status(500).json({ message: 'Server error' });  
 }  
});

The jwt.sign function needs a key, which should not be exposed in the code. For that, we created an .env file in which we added the environment variable for our key.



Content of .env file:

JWT\_KEY="your\_jwt\_secret"

Note that you will need to import some things in your app in order to use jwt:

const express = require('express')  
const *jwt* = require('jsonwebtoken');

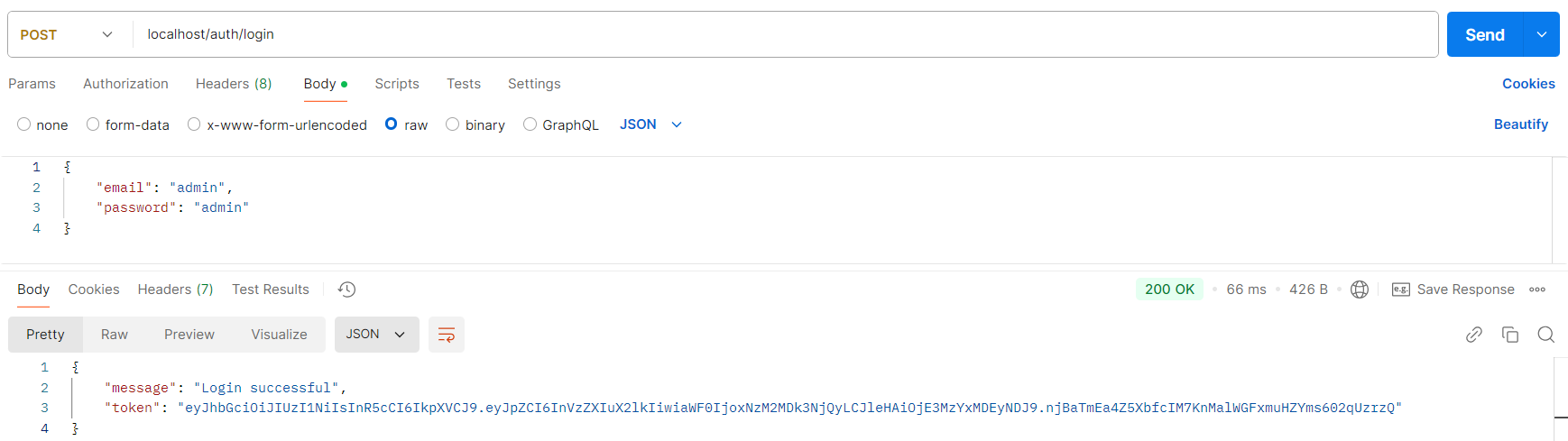
Now, to actually use the token, we will create another service named service1.

It will also have a post method accessed through the url ‘/service1’. In this method, we verify the web token using jwt.verify and the same key we defined earlier. If it is found, we will send the message that the user is authenticated. Otherwise, we send an error message.

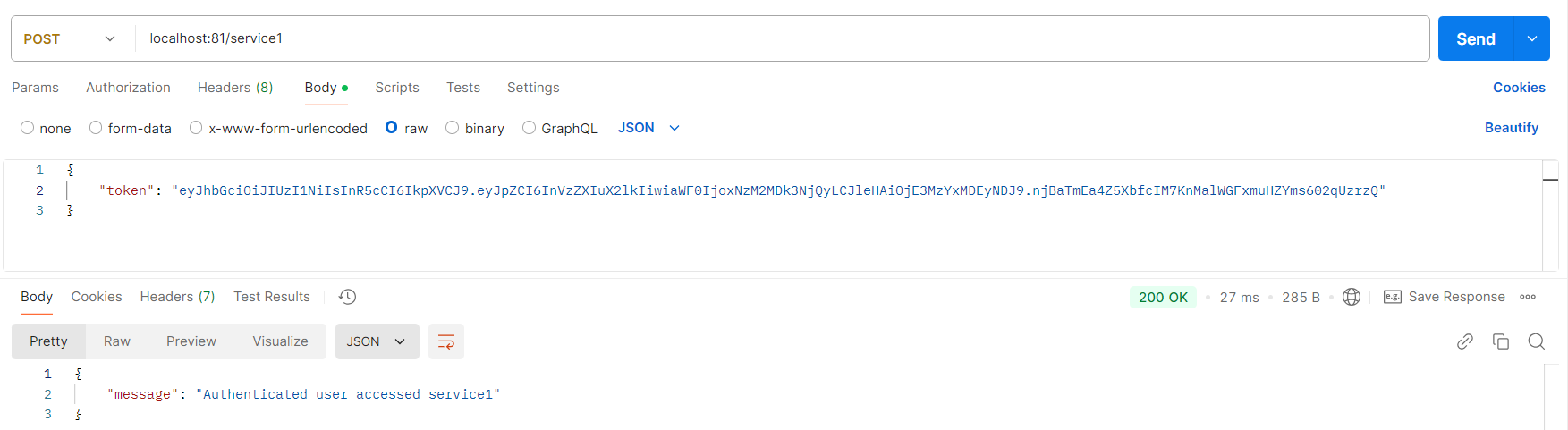
app.post('/service1', async (req, res) => {  
 const { token } = req.body;  
 try{  
 const verifToken = *jwt*.verify(token, *process*.env.JWT\_KEY)  
 if (verifToken) {  
 res.status(200).json({ message: 'Authenticated user accessed service1' });  
 }  
 } catch(err) {  
 *console*.error(err);  
 res.status(401).json({ message: 'User not authenticated' });  
 }  
  
});

Our example app uses docker for deploying the solution, so some of the code such as the environment variables import is found in the docker-compose.yml file.

To check the REST APIs, we will use Postman. Here are two example requests for checking these APIs:



Use the previously generated token in the body of the second request:



For the working example app, reference this git project: <https://github.com/Criss16/SOA-Tutorial/tree/main>