**School of Computing**

**ST0503 Back End Web Development**

**Practical 7**

**Authentication and Authorization**

**Objectives:**

After completing this lab, you should be able to:

* Implement JWT for authentication and authorization
* Secure the REST API endpoints

In this practical, we will implement authentication, so that only logged in users can access certain endpoints .

Open MySQLWorkbench and execute the following SQL to add the password column:

use friendbook;  
  
ALTER TABLE user ADD COLUMN password VARCHAR(255) NOT NULL;

If you inspect the password column for the user table, you will notice that all our user’s have a password of '' (empty string).

# 1. Login endpoint

Let’s implement the endpoint (POST /login/) for logging in. We will send back a JWT token if the login details are correct. JWT tokens allow us to identify who is calling our endpoints.

## 1.1. JWT secret key

JWT token needs to be signed with a secret so that we can verify that a token sent to us is untampered. Let’s setup environment variables for storing our JWT secret key.

Create a new config.js file in the main project folder

var secretKey='dfkhfkda6812683216jcxzm876875@!#@$dsd';

module.exports=secretKey;

## 1.2. User.verify()

Install the jsonwebtoken package:

npm install --save jsonwebtoken

Add the verify method to the User model:

verify: function (username, password, callback) {

var dbConn = db.getConnection();

dbConn.connect(function (err) {

if (err) {//database connection gt issue!

console.log(err);

return callback(err, null);

} else {

const query = "SELECT \* FROM user WHERE username=? and password=?";

dbConn.query(query, [username, password], (error, results) => {

if (error) {

callback(error, null);

return;

}

if (results.length === 0) {

return callback(null, null);

} else {

const user = results[0];

return callback(null, user);

}

});

}

});

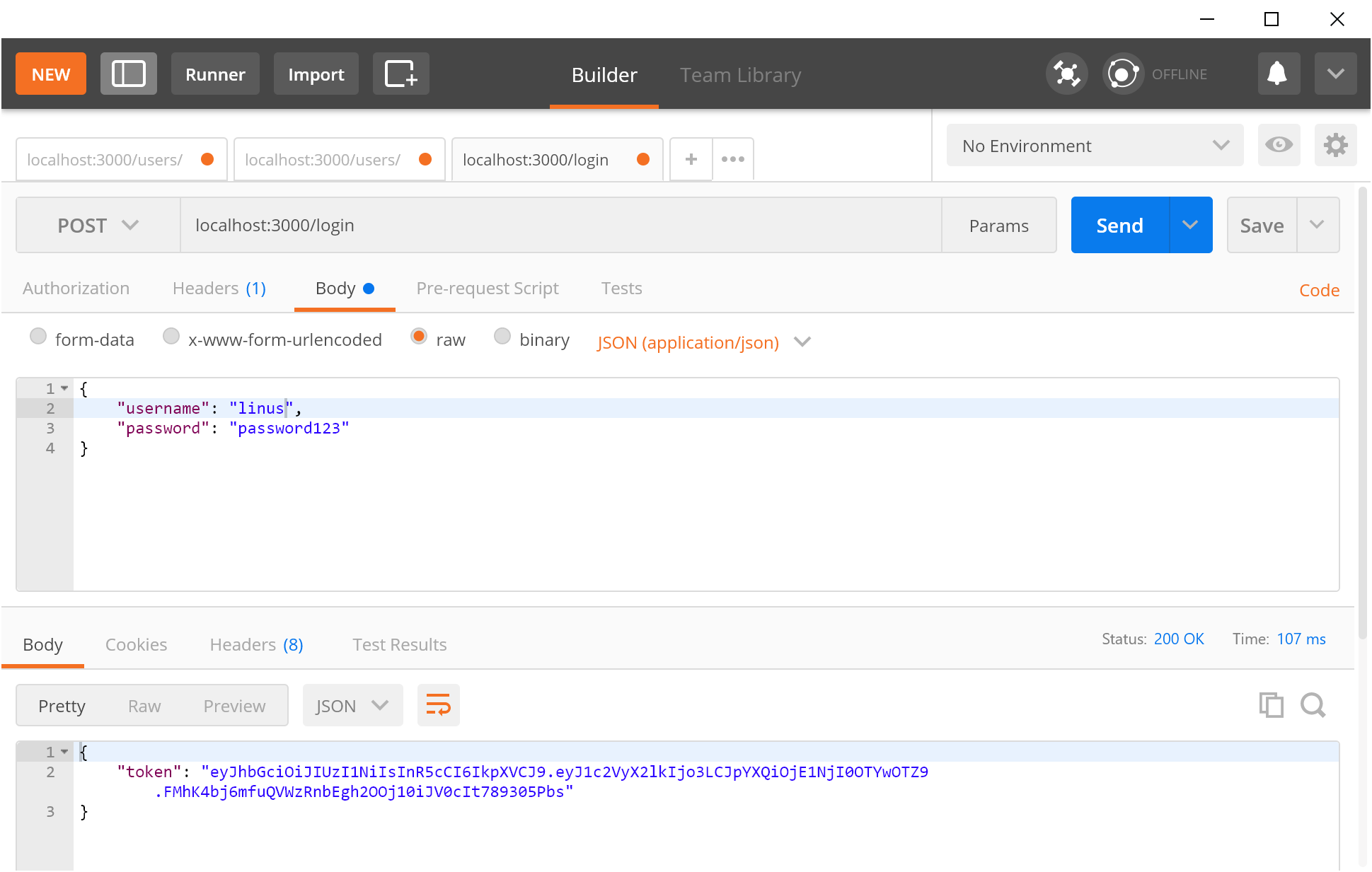
}

Add the POST /login/ handler to app.js. Make sure to import JWT\_SECRET and jsonwebtoken:

...  
...  
...  
const jwt = require("jsonwebtoken");  
const JWT\_SECRET = require("../config.js");   
  
app.post("/login/", (req, res) => {  
 User.verify(  
 req.body.username,  
 req.body.password,  
 (error, user) => {  
 if (error) {  
 res.status(500).send();  
 return;  
 }  
 if (user === null) {  
 res.status(401).send();  
 return;  
 }  
 const payload = { user\_id: user.id };  
 jwt.sign(payload, JWT\_SECRET, { algorithm: "HS256" }, (error, token) => {  
 if (error) {  
 console.log(error);  
 res.status(401).send();  
 return;  
 }   
 res.status(200).send({  
 token: token,  
 user\_id: user.id  
 });  
 })  
 });  
});  
...  
...  
...  
  
module.exports = app;

We send a 401 Unauthorized when the login fails. Otherwise, we send the JWT token and the user ID. Note that the JWT token’s payload contains the user ID, which is important because when the user sends the JWT token in a HTTP request, we can decode it to check which user is accessing our endpoint. We send the user ID even though we could decode the token on the client side to retrieve the user ID for simplicity’s sake.

Test out the endpoint on Postman:



Provide a wrong password. You should get a 401 Unauthorized. Repeat with a non-existent username. You should also see a 401 Unauthorized.

## 3.3. Securing our endpoints

We will now require users to provide the authentication header to access certain endpoints (POST and PUT ones). When we make a request to an authenticated endpoint, the JWT token should be included in the Authentication header with a “Bearer” prefix.

Create a file named isLoggedInMiddleware.js in a new **auth** directory with the following content:

const jwt = require("jsonwebtoken");  
const JWT\_SECRET = require("../config.js");  
  
var check = (req, res, next) => {  
 const authHeader = req.headers.authorization;  
 if (authHeader === null || authHeader === undefined || !authHeader.startsWith("Bearer ")) {  
 res.status(401).send();  
 return;  
 }  
 const token = authHeader.replace("Bearer ", "");  
 jwt.verify(token, JWT\_SECRET, { algorithms: ["HS256"] }, (error, decodedToken) => {  
 if (error) {  
 res.status(401).send();  
 return;  
 }  
 req.decodedToken = decodedToken;  
 next();  
 });  
};

module.exports=check;

As you see in the jwt.verify() callback, if the verification is successful, we store the decoded token in req.decodedToken and invoke the next() method to pass the req and res object to the next middleware (authenticated endpoint handlers). Otherwise, we send a 401 Unauthorized.

Mount the middleware before the following handlers in controller.js:

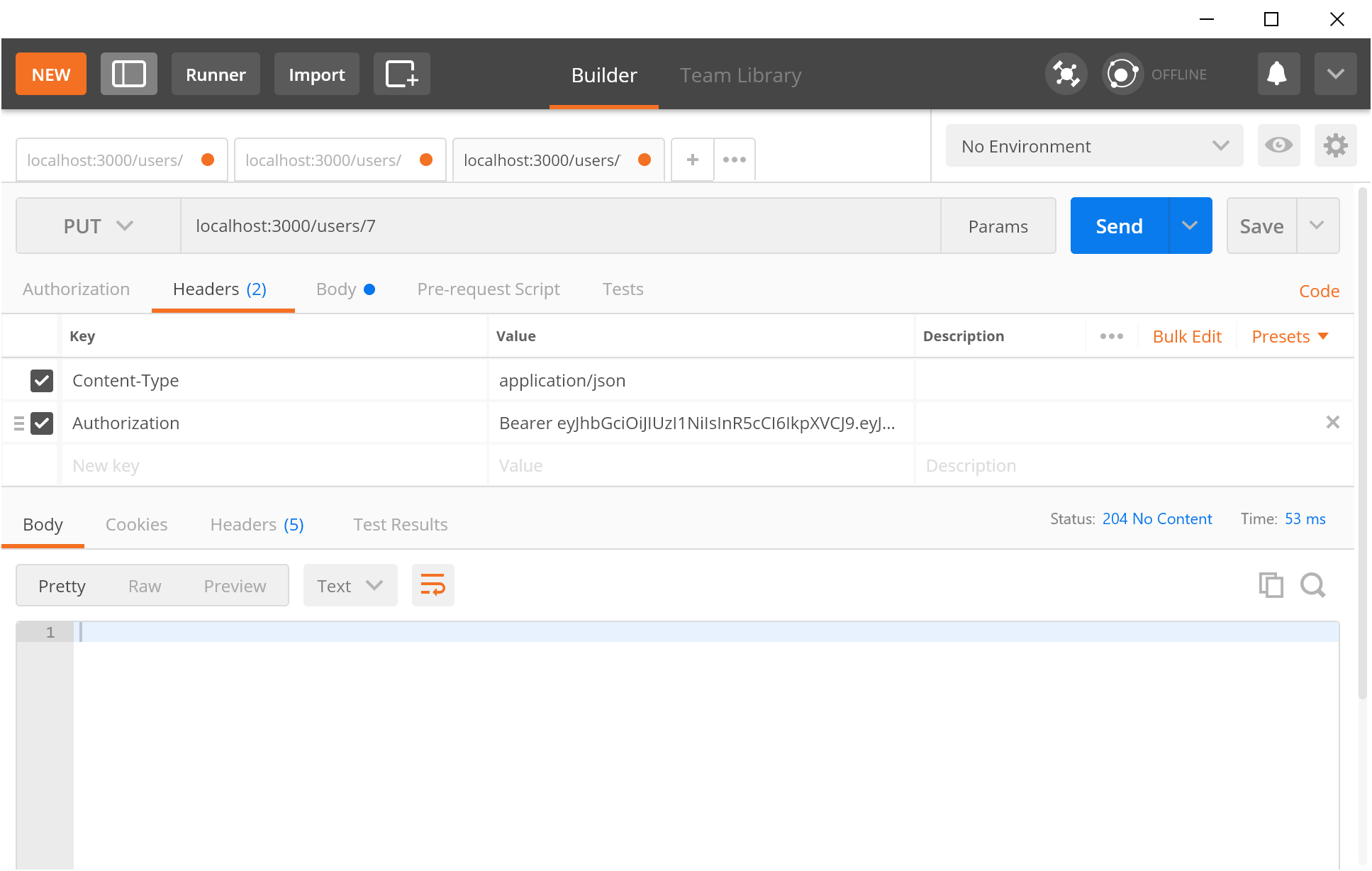
const isLoggedInMiddleware = require("../auth/isLoggedInMiddleware");  
...  
app.put("/users/:userID/", isLoggedInMiddleware, (req, res, next) => {  
...  
app.post("/users/:userID/friends/:friendID", isLoggedInMiddleware, (req, res) => {  
...  
app.delete("/users/:userID/friends/:friendID/", isLoggedInMiddleware, (req, res) => {

Make a request to PUT /users/:userID. Put in Linus’ user ID. You should get a 401.

Now, make the same request, but with the Authorization header. The value should be the JWT token from Section 3.2. with “Bearer” as a prefix, and the body should be the following:

{  
 "full\_name": "Linus Torvalds",  
 "username": "linus",  
 "bio": "Works on Linux and Git!",  
 "date\_of\_birth": "1970-10-19"  
}

You should be get a 201 No Content:



Repeat the same request with no Authorization header. You should get a 401 Unauthorized.

While the three endpoints now require authentication, they do not have any authorization implemented. Any one that is logged in can freely update or delete users. We only want the logged in user to be able to update or delete their own accounts, and add or remove friends to their own account.

Add a check after the userID has been parsed. If the user ID in the decoded token does not match the one in the request parameters, send a 403.

app.put("/users/:userID/", isLoggedInMiddleware, (req, res, next) => {  
 const userID = parseInt(req.params.userID);  
 if (isNaN(userID)) {  
 res.status(400).send();  
 return;  
 }  
  
 // user ID in the request params should be the same as the logged in user ID  
 if (userID !== req.decodedToken.user\_id) {  
 res.status(403).send();  
 return;  
 }  
 ...

app.post("/users/:userID/friends/:friendID", isLoggedInMiddleware, (req, res) => {  
 const userID = parseInt(req.params.userID);  
 const friendID = parseInt(req.params.friendID);  
 if (isNaN(userID) || isNaN(friendID)) {  
 res.status(400).send();  
 return;  
 }  
   
 if (userID === friendID) {  
 res.status(400).send();  
 return;  
 }  
  
 // user ID in the request params should be the same as the logged in user ID  
 if (userID !== req.decodedToken.user\_id) {  
 res.status(403).send();  
 return;  
 }  
 ...

app.delete("/users/:userID/friends/:friendID/", isLoggedInMiddleware, (req, res) => {  
 const userID = parseInt(req.params.userID);  
 const friendID = parseInt(req.params.friendID);  
  
 if (isNaN(userID) || isNaN(friendID)) {  
 res.status(400).send();  
 return;  
 }  
  
 // user ID in the request params should be the same as the logged in user ID  
 if (userID !== req.decodedToken.user\_id) {  
 res.status(403).send();  
 return;  
 }  
 ...

Test out the endpoints with Postman to ensure that only the logged in user is be able to update or delete their own accounts, and add or remove friends to their own account.

### 3.4. Hiding the password

Perform a GET /users/1/ request. You should see that the password property is present. We should not expose it through our REST endpoints.

Modify the SELECT SQL queries in both models to include all user columns except for password.

**New User.findByID() query**

SELECT id, full\_name, username, bio, date\_of\_birth, created\_at FROM user WHERE id = ?;

**New User.findAll() query**

SELECT id, full\_name, username, bio, date\_of\_birth, created\_at FROM user;

**New User.showFriends() query**

SELECT user.id, user.full\_name, user.username, user.bio, user.date\_of\_birth, user.created\_at  
FROM user  
INNER JOIN friendship  
ON user.id = friendship.fk\_friend\_one\_id  
WHERE friendship.fk\_friend\_two\_id = ?;

OR

SELECT user.id, user.full\_name, user.username, user.bio, user.date\_of\_birth, user.created\_at  
FROM user, friendship  
where user.id = friendship.fk\_friend\_one\_id  
and friendship.fk\_friend\_two\_id = ?;

**New Post.findLikers() query**

SELECT user.id, user.full\_name, user.username, user.bio, user.date\_of\_birth, user.created\_at FROM user  
INNER JOIN likes ON likes.fk\_user\_id = user.id  
WHERE likes.fk\_post\_id = ?

OR

SELECT user.id, user.full\_name, user.username, user.bio, user.date\_of\_birth, user.created\_at FROM user, likes where likes.fk\_user\_id = user.id  
and likes.fk\_post\_id = ?

**New Post.findLikersByPostIDs query**:

SELECT user.id, user.full\_name, user.username, user.bio, user.date\_of\_birth, user.created\_at, likes.fk\_post\_id FROM user  
INNER JOIN likes ON likes.fk\_user\_id = user.id  
WHERE likes.fk\_post\_id IN (?);

OR

SELECT user.id, user.full\_name, user.username, user.bio, user.date\_of\_birth, user.created\_at, likes.fk\_post\_id FROM user,likes where likes.fk\_user\_id = user.id and likes.fk\_post\_id IN (?);

The user.password column should no longer be exposed through our API.

## 3.5. Exercise: Securing the Post endpoints

Modify the handlers for:

1. POST /posts/
2. PUT /posts/:postID
3. DELETE /posts/:postID/

such that the logged in user can only create/update/delete the Post resource if it belongs to them. You may need to use the Post.findByID() method to get the fk\_poster\_id.

For the following endpoints:

1. POST /posts/:postID/likers/:likerID/
2. DELETE /posts/:postID/likers/:likerID/

Modify the handlers such that 403 Forbidden is sent if likerID !== req.decodedToken.