Working with JWT Server-Side Authentication

Initial Setup

This worksheet builds on your knowledge from the week 8 prac and the JWT Client-Side Authentication worksheet. It is expected that you have already setup MySQL with the world database, and connected your application to the database using knex (see the week 8 prac for details of how to use knex). To begin with, we are going to add some additional dependencies to your application. We will explore what they do later, though the names may give the game away:

```
npm install jsonwebtoken bcrypt
```

The next step is to modify your world database by adding a new table which will store the users of your website. In this worksheet we will show the operations from the GUI-based MySQL Workbench app, but you can easily do everything you need to do from the vanilla MySQL command line. Choose the approach that works for you.

Initially your world database should have the following tables: city, country, and countrylanguage.

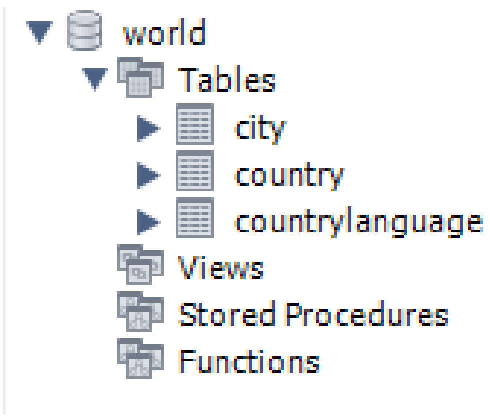


Figure 1 world database

Begin creating a new table by right clicking on the Tables link, and selecting "Create table…". We will then fill in the table with the following details:

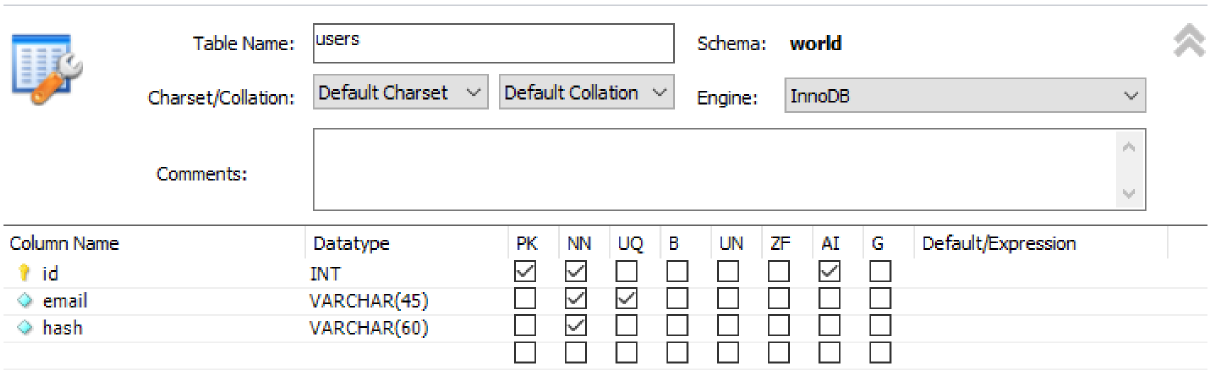


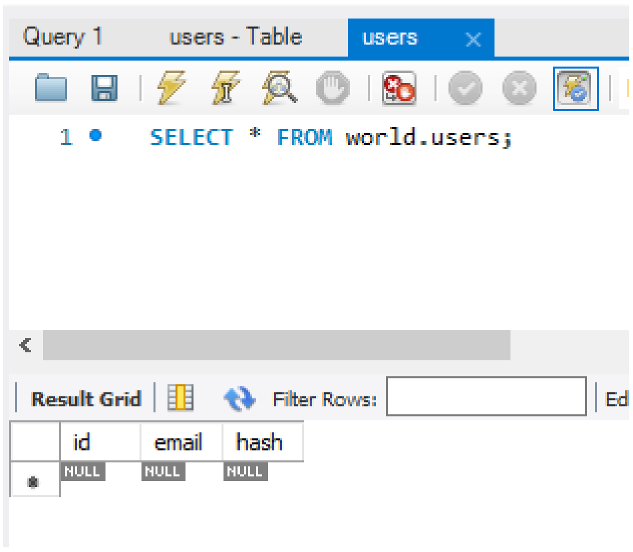
Table Name: users
Schema: world
Charset/Collation: Default Charset / Default Collation
Engine: InnoDB
Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	✓	✓					✓		
email	VARCHAR(45)		✓	✓						
hash	VARCHAR(60)		✓							

Figure 2 Users table

Note that the options shown above may not match the specification for assignment 2 when it is released. I will leave it as an exercise for the reader to explore what each of the settings does, and how changing them affects your table, but as a guide: PK = Primary Key, NN = Non Null, UQ = Unique and AI = Auto Increment. Most of these are pretty clear here. For a quick explanation, see: https://stackoverflow.com/questions/3663952/what-do-column-flags-mean-in-mysql-workbench

Once you are happy with your settings, hit the Apply button and follow the prompts to save the table to your database. If you are successful, you should be able to run the following query in the database:



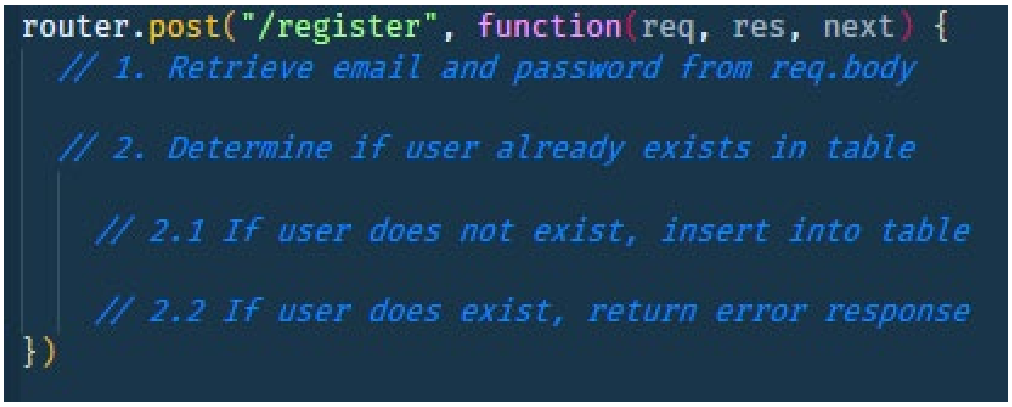
```
SELECT * FROM world.users;
```

	id	email	hash
*	NULL	NULL	NULL

Figure 3 Users table

Register Route

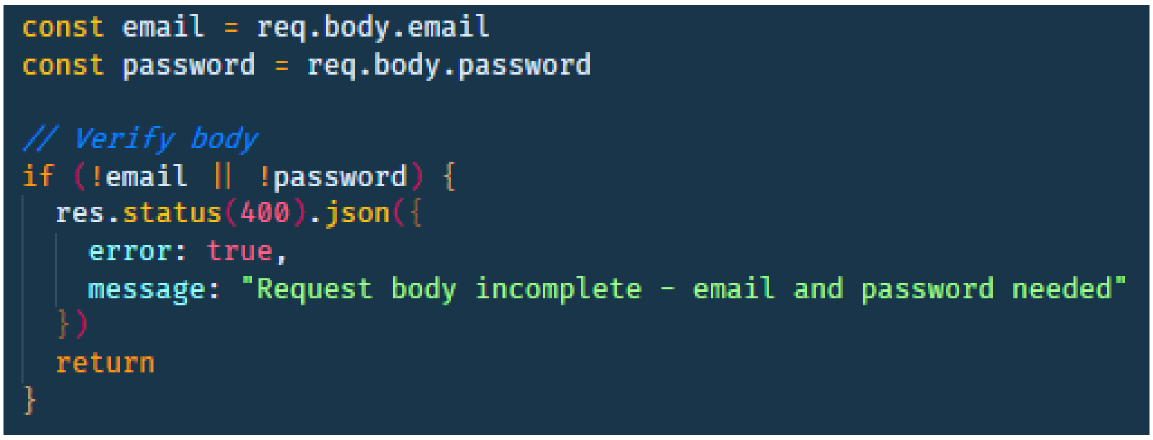
Moving back to the express app, the first new route we must create will be `/users/register`. This will allow us to play with hashing passwords and inserting data into the database. Open your `users.js` file (in the routes subdirectory) and add the following boilerplate:



```
router.post("/register", function(req, res, next) {
  // 1. Retrieve email and password from req.body

  // 2. Determine if user already exists in table

    // 2.1 If user does not exist, insert into table

    // 2.2 If user does exist, return error response
})
```

Figure 4 Register route boilerplate

The first, and simplest step, to registering an account is retrieving the user's credentials from the body of the request. For this route, we expect the route will receive an `email` and `password` input, thus we can simply extract the values from the request body and perform a simple check that the inputs exist. For more robust applications, we may also wish to check other aspects of the inputs such as if they are the correct type, or if the email is a valid email.

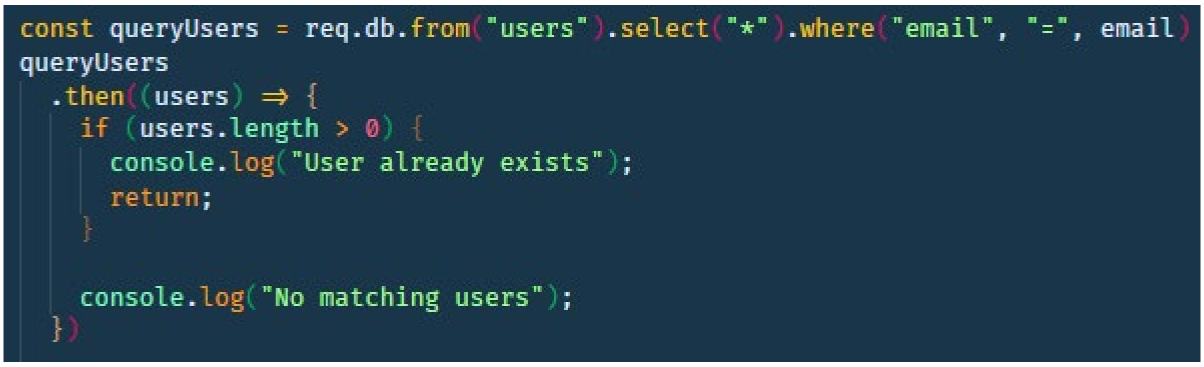


```
const email = req.body.email
const password = req.body.password

// Verify body
if (!email || !password) {
  res.status(400).json({
    error: true,
    message: "Request body incomplete - email and password needed"
  })
  return
}
```

Figure 5 Retrieve email and password from request body

Note in the example above that the response specifies a status value and then returns. You will find a complete list of HTTP status codes here https://httpstatuses.com/ or at Mozilla's MDN site, and a quick check shows that 400 indicates a Bad Request. The Bad Request code will indicate to users of the API that their request to the server was incorrectly formatted and that they should check the documentation again. After the server provides a response, it is important to `return` to prevent any further code from running.

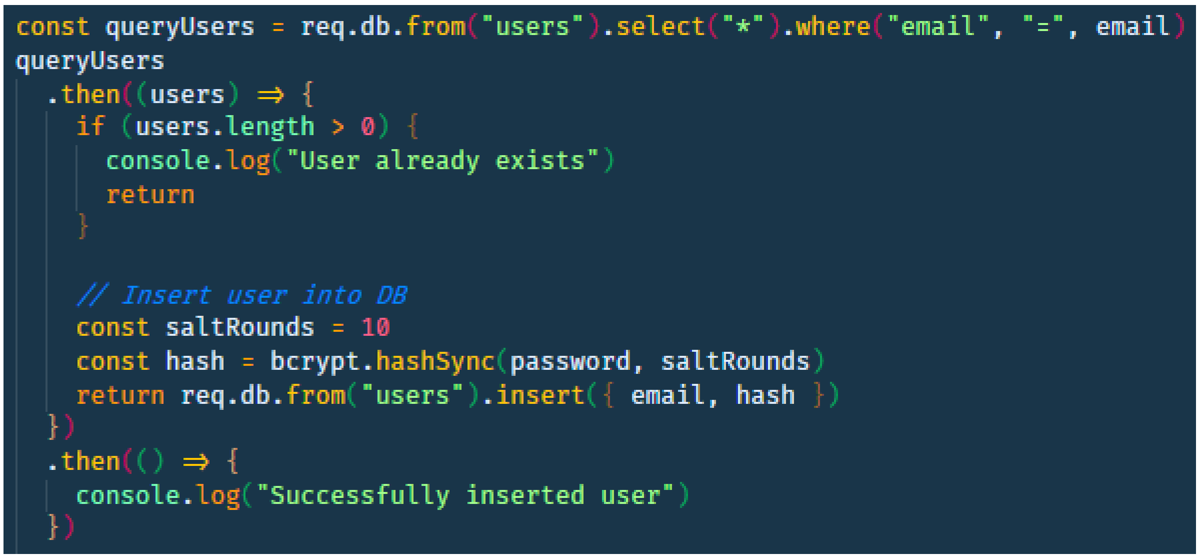
The next step in registration is to determine if the user already exists in the database. This prevents forgetful users from re-registering an account, or malicious users from attempting to gain access to other accounts. To do this we first need to make a request to the users table to check for any users who have the same email address. In our application the email address is akin to a username, and thus there can only ever be one user registered under the address. You may recall that when we set up the table, we set the UQ flag to ensure that entries in the email column are unique and a matching email can never be inserted. To check for users, we need to request accounts in the users table where the email matches the email provided by the request.



```
const queryUsers = req.db.from("users").select("*").where("email", "=", email)
queryUsers
  .then((users) => {
    if (users.length > 0) {
      console.log("User already exists");
      return;
    }

    console.log("No matching users");
  })
```

Figure 6 Request any matching accounts in the users table

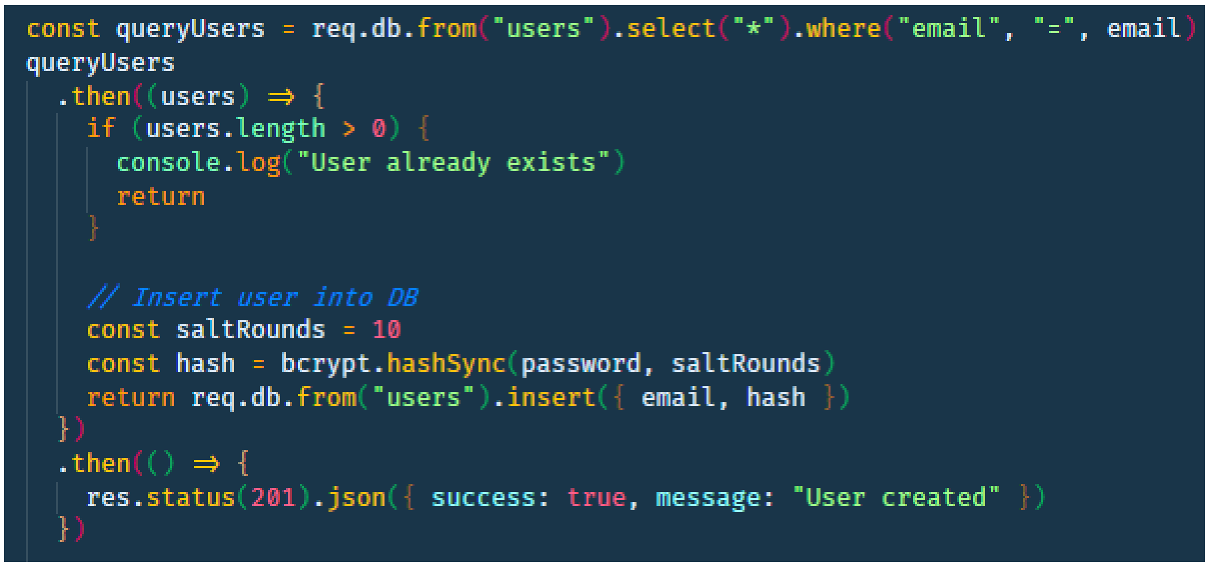
The response from the server will return a list of matching results, thus if the length of the array is greater than 0 a user must exist in the table. Once we have established that a user does not exist in the table, we can work on inserting their details. A best practice when it comes to storing user credentials is to never save the password. This post here https://bit.ly/2zyaHVU by Lucas Kauffman tells us a lot more about hashing, and why it is important to use it on passwords. It is a few years out-of-date now, however much of what he discusses is still relevant today. You may remember that we installed a dependency called `bcrypt` in the initial setup of the project. `bcrypt` is a library which handles the process of hashing and comparing hashed passwords. We will use it to insert accounts in the users table, and verify login attempts. You can read about the library here: https://github.com/kelektiv/node.bcrypt.js.



```
const queryUsers = req.db.from("users").select("*").where("email", "=", email)
queryUsers
  .then((users) => {
    if (users.length > 0) {
      console.log("User already exists")
      return
    }

    // Insert user into DB
    const saltRounds = 10
    const hash = bcrypt.hashSync(password, saltRounds)
    return req.db.from("users").insert({ email, hash })
  })
  .then(() => {
    console.log("Successfully inserted user")
  })
```

Figure 7 Inserting user into database

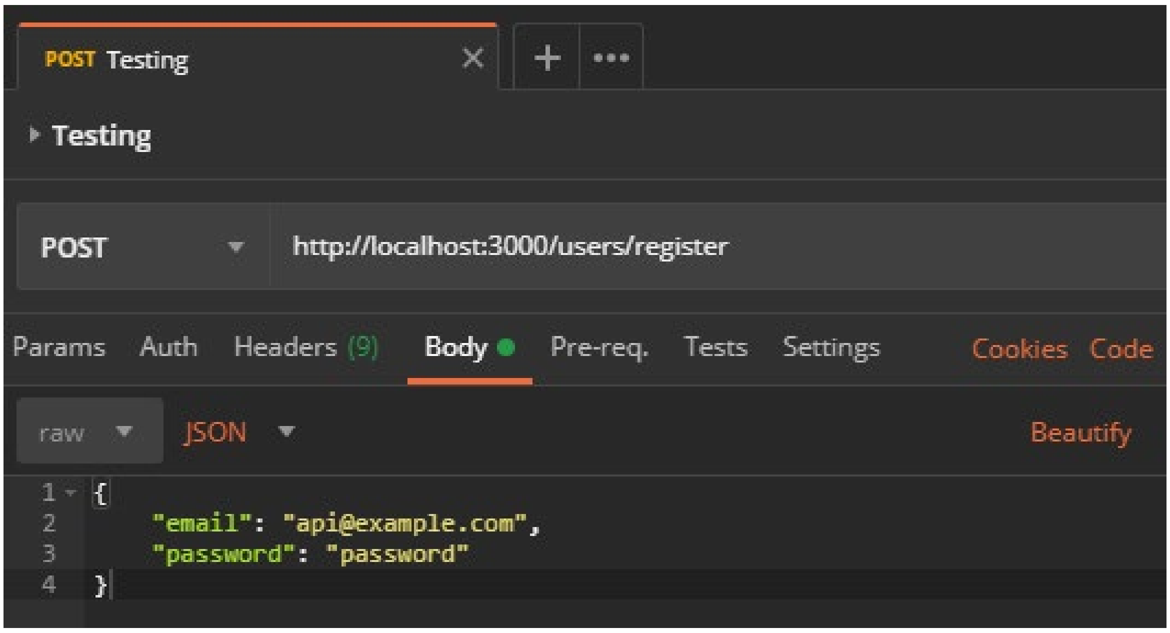
A few things are happening in the couple of lines of code added. We set a variable called `saltRounds`, this will generate a salt value that has been processed for 10 rounds. A salt value is added to a password as a way of preventing users with the same passwords creating the same hash value. The second interesting piece of code is where we hash the password using `bcrypt.hashSync(password, saltRounds)`. This is taking the password given by the API request, combining it with the salt value, and hashing the result. The hashed output should hopefully be impossible for an attacker to decrypt in any reasonable amount of time, thereby ensuring the security of our application. We finally take this hashed password, and the provided email, and insert them into the users table of the world database. If the request is valid, the express application will log that the user was successfully inserted. Handling any errors from the database will be left up to the reader, however the final result should output a 201 response with a "User created" message. The completed route can be seen below:



```
const queryUsers = req.db.from("users").select("*").where("email", "=", email)
queryUsers
  .then((users) => {
    if (users.length > 0) {
      console.log("User already exists")
      return
    }

    // Insert user into DB
    const saltRounds = 10
    const hash = bcrypt.hashSync(password, saltRounds)
    return req.db.from("users").insert({ email, hash })
  })
  .then(() => {
    res.status(201).json({ success: true, message: "User created" })
  })
```

Figure 8 Return 201 status when user successfully created

With the route now operational, we can view the results produced in the database. There are a number of programs you may use to mock requests to your API. A recommended software is Postman: https://www.postman.com/downloads/, however you may use any approach you feel comfortable with. The following settings should hit your express application with minimal changes (you may need to adjust the port number to 3001):



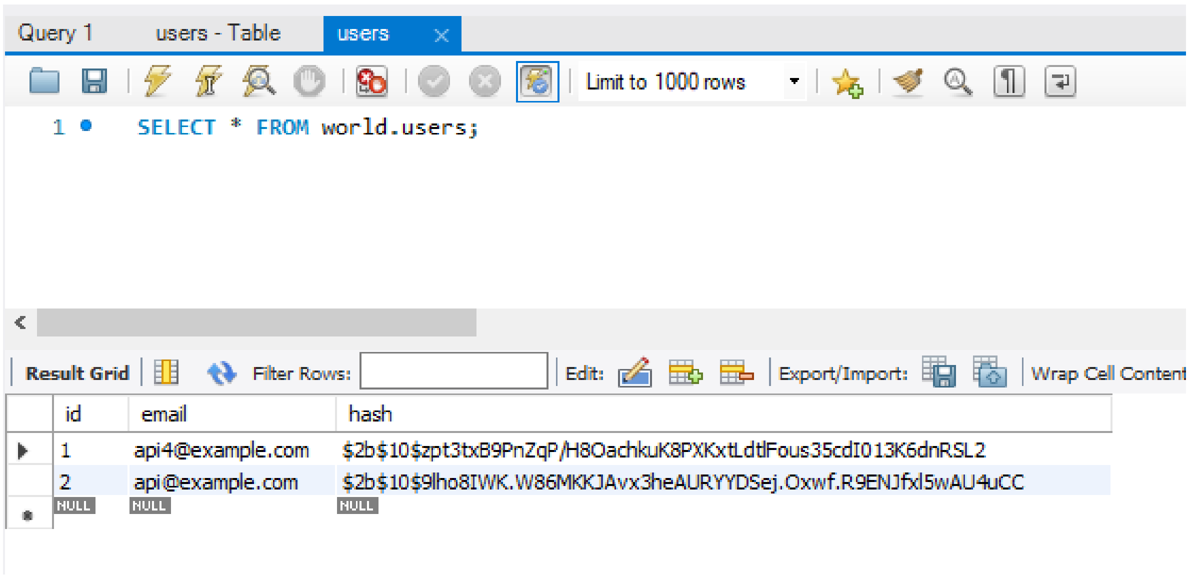
POST http://localhost:3000/users/register

raw JSON

```
{
    "email": "api@example.com",
    "password": "password"
}
```

Figure 9 Mock API request

If you successfully hit the API, it should return a 201 response with the message "User created". Viewing the users table in MySQL workbench should produce results similar to those shown below:



```
SELECT * FROM world.users;
```

id	email	hash
1	api4@example.com	$2b$10$zpt3txB9PnZqP/H8OachkuK8PXKxtLdtlFous35cdI013K6dnRSL2
2	api@example.com	$2b$10$9lho8IWK.W86MKKJAvx3heAURYYDSej.Oxwf.R9ENJfxl5wAU4uCC
NULL	NULL	NULL

Figure 10 Registered users

Login Route

Now that registered accounts exist in the users table, we can allow them to log in. This will be done through the /users/login route which we will create inside the `users.js` file alongside the register route. Copy the following boilerplate into your file:

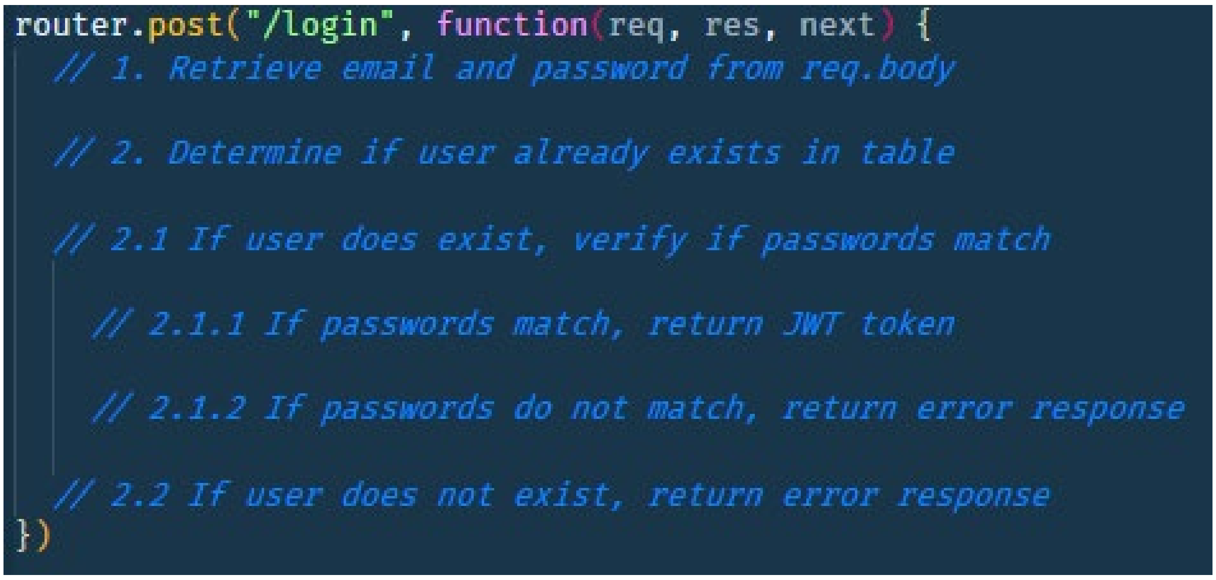

```
router.post("/login", function(req, res, next) {
  // 1. Retrieve email and password from req.body

  // 2. Determine if user already exists in table

  // 2.1 If user does exist, verify if passwords match

    // 2.1.1 If passwords match, return JWT token

    // 2.1.2 If passwords do not match, return error response

  // 2.2 If user does not exist, return error response
})
```

Figure 11 Login route boilerplate

As with the register route, we will verify that the email and password fields have been provided:

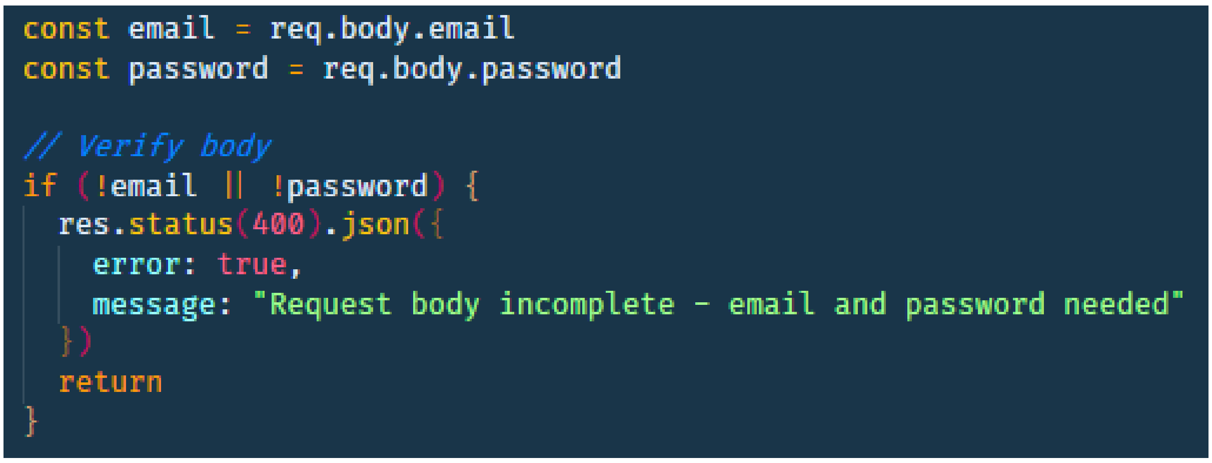

```
const email = req.body.email
const password = req.body.password

// Verify body
if (!email || !password) {
  res.status(400).json({
    error: true,
    message: "Request body incomplete - email and password needed"
  })
  return
}
```

Figure 12 Retrieve email and password from request body

We once again check if the user exists in the table, however this time we are checking to verify if the account exists. If it does not, we should return early.

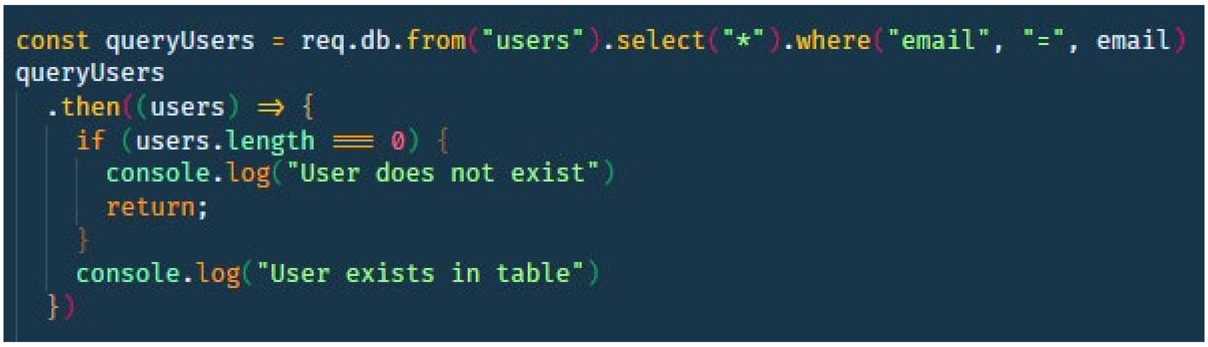
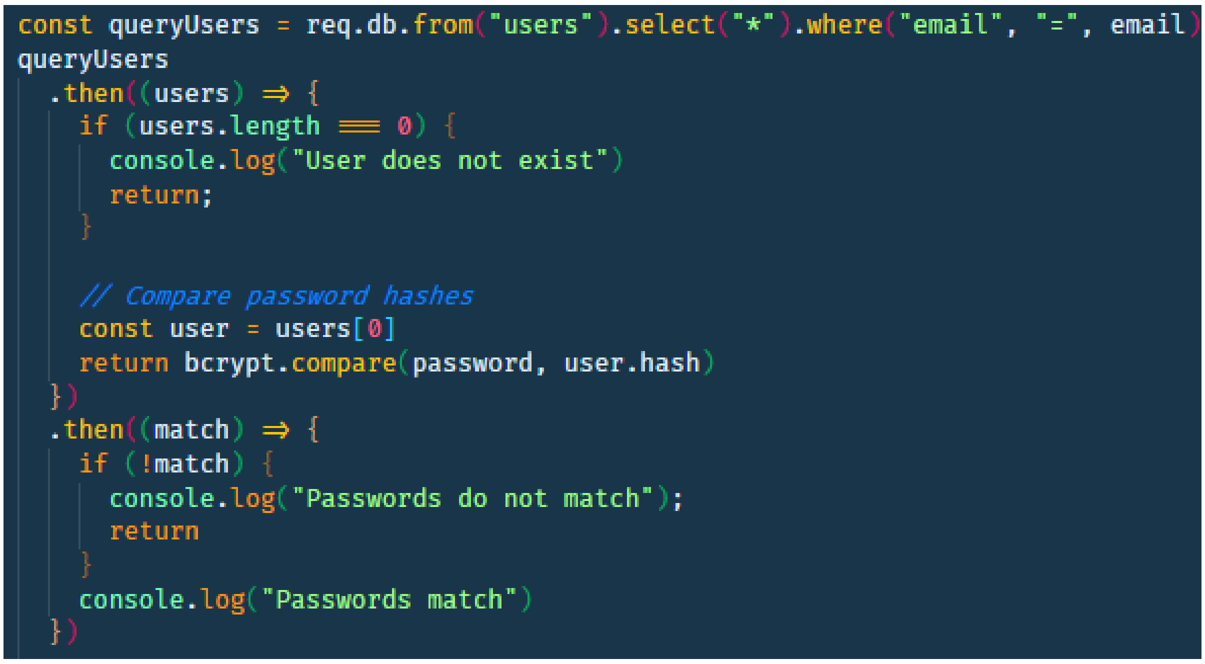

```
const queryUsers = req.db.from("users").select("*").where("email", "=", email)
queryUsers
  .then((users) => {
    if (users.length === 0) {
      console.log("User does not exist")
      return;
    }
    console.log("User exists in table")
  })
```

Figure 13 Request any matching accounts in the user table

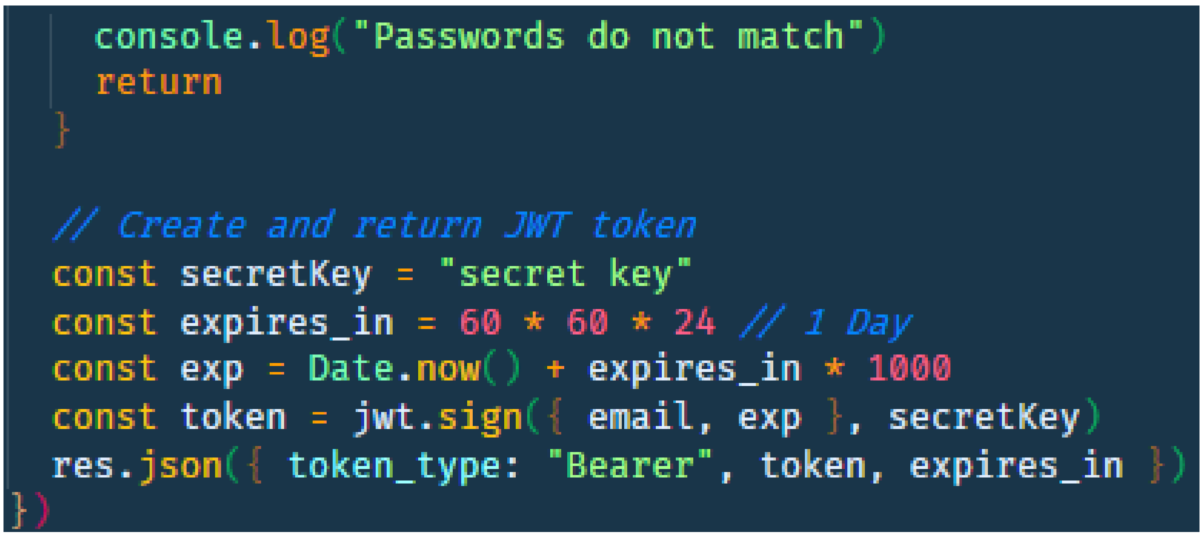
At this point we must compare the password provided by the request with the value stored in the database. We will perform the comparison of passwords using `bcrypt` as it will handle taking the plaintext password provided by the request, and hashing it using the same algorithm and salt as the stored password. This is possible because the hashing algorithm and salt are stored with the hash output given previously when the hashed password is saved into the database.



```
const queryUsers = req.db.from("users").select("*").where("email", "=", email)
queryUsers
  .then((users) => {
    if (users.length === 0) {
      console.log("User does not exist")
      return;
    }

    // Compare password hashes
    const user = users[0]
    return bcrypt.compare(password, user.hash)
  })
  .then((match) => {
    if (!match) {
      console.log("Passwords do not match");
      return
    }
    console.log("Passwords match")
  })
```

Figure 14 Compare passwords

Once the comparison has occurred, we can finally produce the JWT token which will allow the user to perform authenticated requests to the server. There are a couple of options required when attempting to sign a token which have been highlighted in the code sample below:



```
      console.log("Passwords do not match")
      return
    }

    // Create and return JWT token
    const secretKey = "secret key"
    const expires_in = 60 * 60 * 24 // 1 Day
    const exp = Date.now() + expires_in * 1000
    const token = jwt.sign({ email, exp }, secretKey)
    res.json({ token_type: "Bearer", token, expires_in })
})
```

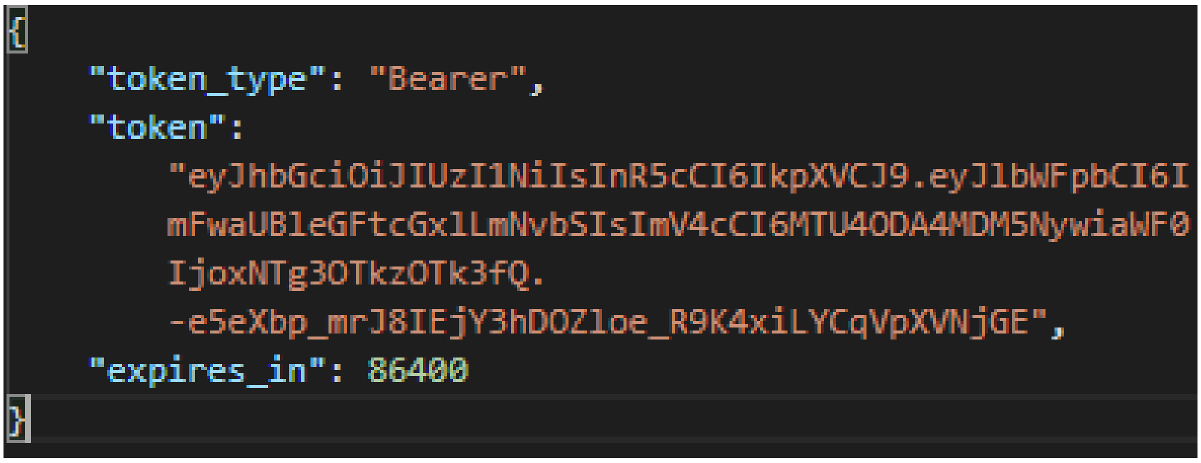
Figure 15 Signing JWT token

The `secretKey` is a value used to produce the signature in the third segment of the JWT token. The signature is an important part of the token as it is used to verify if the user is who they say they are (see Lecture 7 and the Client Side JWT worksheet for details). This should be kept private - as an attacker with the key will be able to modify any token and access unauthorized parts of the system.

The other important values are found in the `expires_in` and `exp` variables. These define how long the token will remain valid from the time it was created, and the exact date and time at which it will expire. Depending on the sensitivity of content on your website, this timeframe may be made longer or shorter. Usually a token is valid for a single day, however in applications such as banking this may be reduced to as little as 15 minutes, for reasons that should be obvious.

The final component is the content stored inside the JWT token. In the above code example, we are storing the email and expiration date so that both the client and the server will know the user's identifying details (email) and when the token will expire. Only the server can modify these details while ensuring that the signature is consistent with the content. If a malicious user attempts to modify their email for example, they would be unable to produce another signature for the token. This would be easily detected, and access to the resource would be denied.

Much like with the register router, handling exceptions has been left to the reader, with only the successful response being given. If you were to make a mock request using a valid email and password, you should now receive the following output:

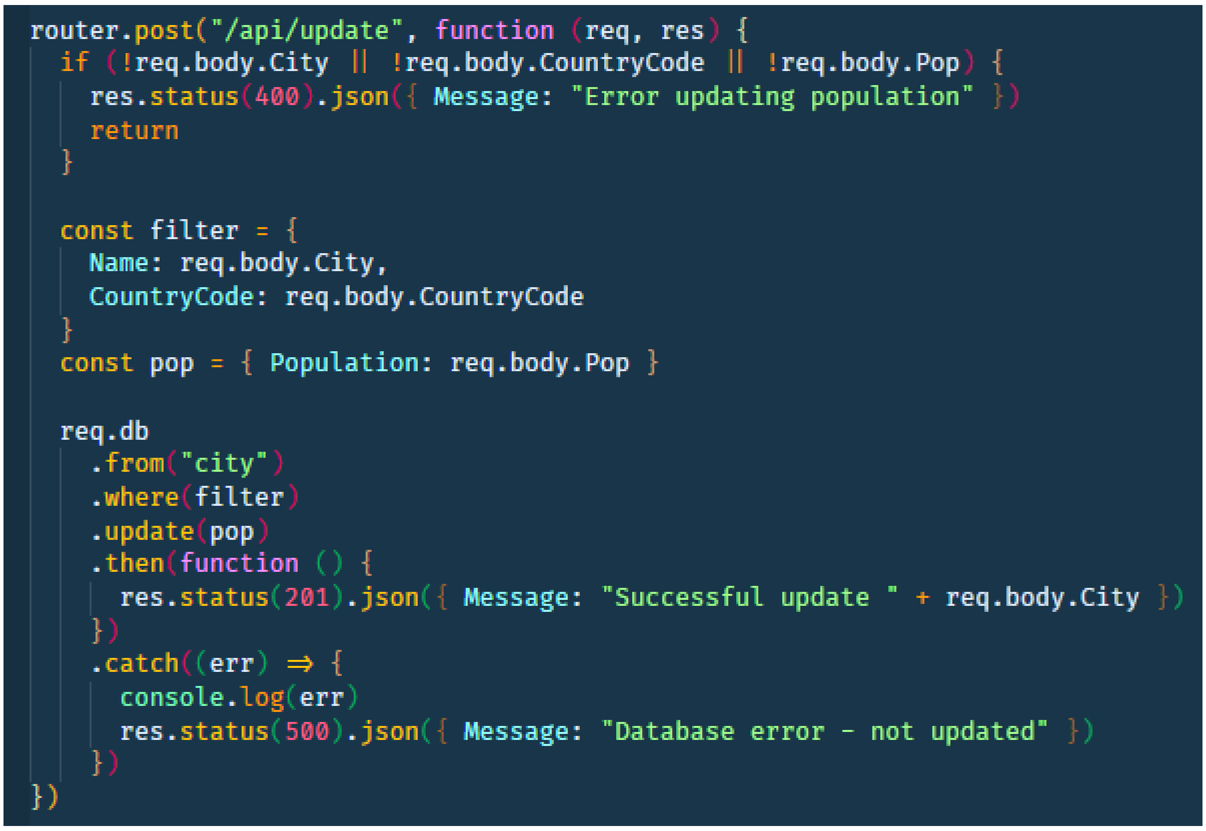


```
{
    "token_type": "Bearer",
    "token":
        "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6I
        mFwaUBleGFtcGxlLmNvbSIsImV4cCI6MTU4ODA4MDM5NywiaWF0
        IjoxNTg3OTkzOTk3fQ.
        -e5eXbp_mrJ8IEjY3hDOZloe_R9K4xiLYCqVpXVNjGE",
    "expires_in": 86400
}
```

Figure 16 Example response

Authenticated Routes

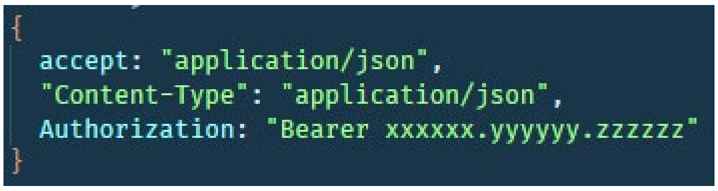
The final step in implementing token-based authentication for your server is to add authenticated routes – routes which verify a provided token and enable access only if the verification is successful. In the following example, we will modify the update route from the week 8 practical. The relevant code is found inside the `index.js` file:



```
router.post("/api/update", function (req, res) {
  if (!req.body.City || !req.body.CountryCode || !req.body.Pop) {
    res.status(400).json({ Message: "Error updating population" })
    return
  }

  const filter = {
    Name: req.body.City,
    CountryCode: req.body.CountryCode
  }
  const pop = { Population: req.body.Pop }

  req.db
    .from("city")
    .where(filter)
    .update(pop)
    .then(function () {
      res.status(201).json({ Message: "Successful update " + req.body.City })
    })
    .catch((err) => {
      console.log(err)
      res.status(500).json({ Message: "Database error - not updated" })
    })
})
```

Figure 17 Update route from index.js

To turn this route into an authenticated route, we need to add the ability to retrieve the JWT token, and verify it is valid. We will follow the specification for Assignment 1 and assume that the token is provided in the request headers under the Authorization flag. An example of the headers which should be submitted to the route can be seen below:



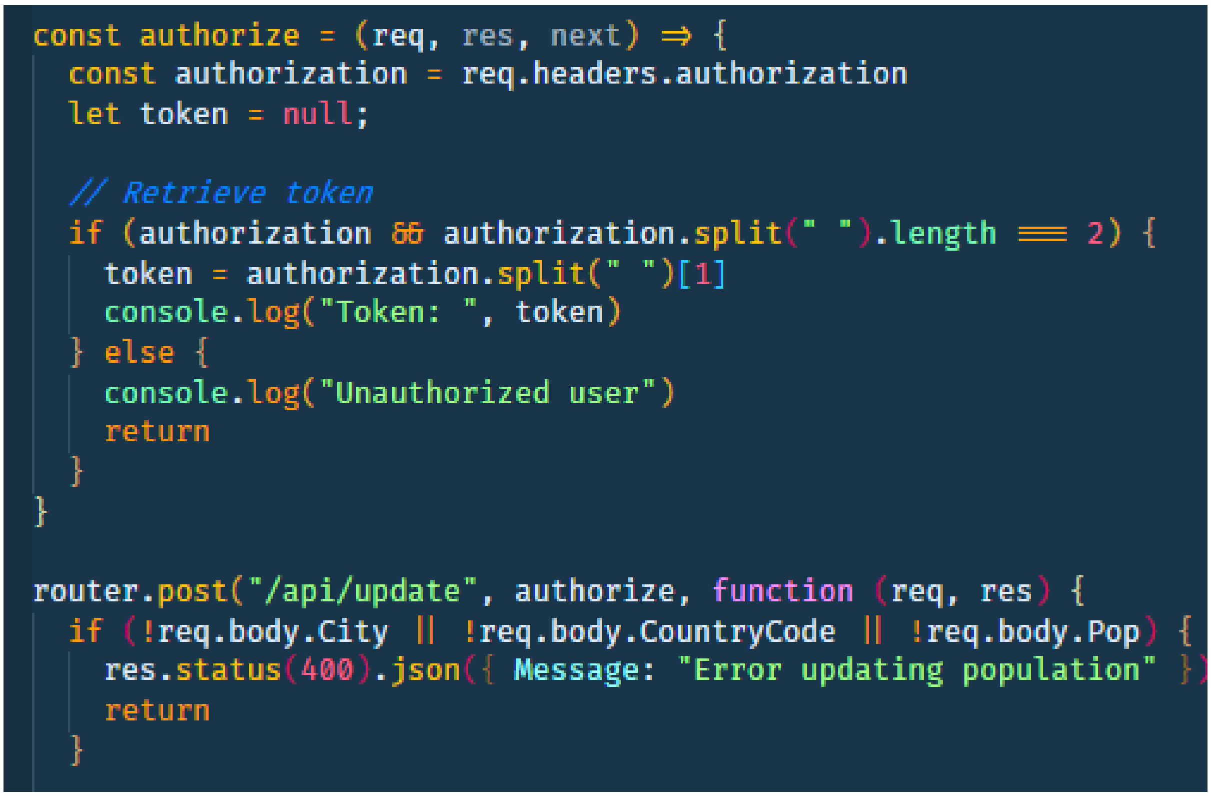
```
{
  accept: "application/json",
  "Content-Type": "application/json",
  Authorization: "Bearer xxxxxx.yyyyyy.zzzzzz"
}
```

Figure 18 Example request headers

Thus, we need to find the authorization header, and extract the token from the Bearer string. Depending on your REST API design, you may wish to perform checking of the request body before or after the authorization request. For the purposes of this worksheet, we will perform authorization checks *before* checking the request body. We will begin by creating a custom middleware called `authorize` which will perform the authorization checks for the route. Initially, these checks will be very simple: if we can find a Bearer token on the Authorization header then we will go ahead. We will show a more sensible approach later on.

To keep the example code manageable, we will begin by using `console.log` statements instead of proper responses – see the comment right at the end of the worksheet about the need to fix these. This will remain your responsibility – you have seen enough examples of these by now – and if you don't fix these responses, the requests will just hang…

For now, consider the code below:



```
const authorize = (req, res, next) => {
  const authorization = req.headers.authorization
  let token = null;

  // Retrieve token
  if (authorization && authorization.split(" ").length === 2) {
    token = authorization.split(" ")[1]
    console.log("Token: ", token)
  } else {
    console.log("Unauthorized user")
    return
  }
}

router.post("/api/update", authorize, function (req, res) {
  if (!req.body.City || !req.body.CountryCode || !req.body.Pop) {
    res.status(400).json({ Message: "Error updating population" })
    return
  }
```

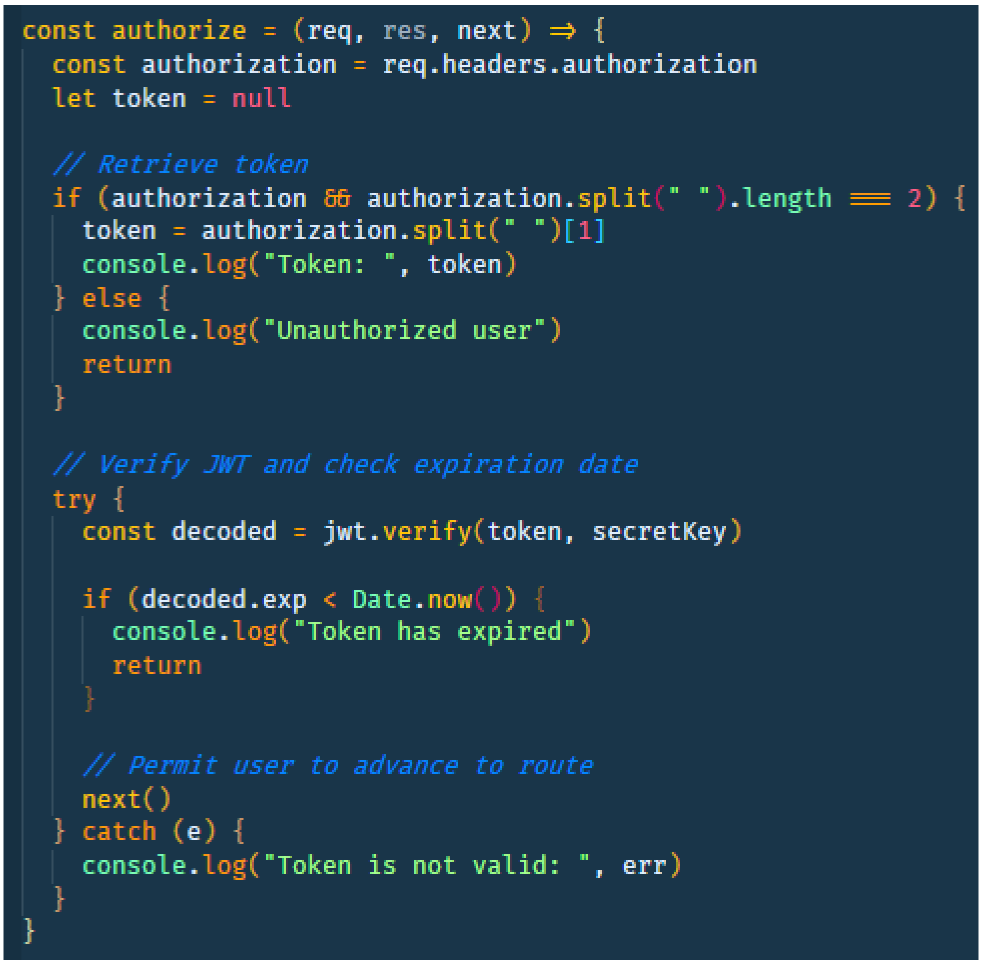
Figure 19 Extracting token from request headers

Here `authorize` parses the header and checks its format, accepting any string in the right place as a valid token. Perhaps more importantly we have added it to the list of items on the POST route.

As we saw in lecture 7, we can associate a number of middleware services with an Express route. Here, `authorize` will be called before the function which handles the update request. For this second function to run (the third input into `router.post`), the preceding function (`authorize`) must call `next()`, otherwise it will be skipped. If we were to access this route right now, it would detect if a token exists, and if so, print to the server's console the state of the token. To properly implement authorization, it will need to undertake appropriate verification of the token, and then call `next()` only when these criteria are satisfied. We will show this later.

We should also look more closely at how the token is extracted from the authorization header. First the authorization header is extracted from the request body. Using the `split` function (https://www.w3schools.com/jsref/jsref_split.asp), the authorization value is split into two segments containing the token type and its value. If both exist (this conditional is mostly a sanity check), we grab the token and assign it to the token variable.

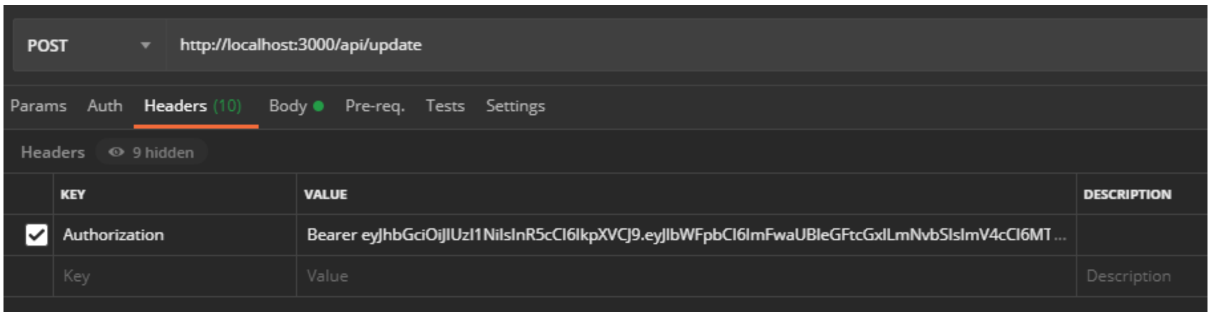
The next step we must take to finish the implementation of the `authorize` middleware is to use the `jsonwebtoken` library (https://github.com/auth0/node-jsonwebtoken) to verify the token properly. This library provides both asynchronous and synchronous functions to do this. For the purposes of this example, we will use the synchronous function as it is simpler to understand.



```
const authorize = (req, res, next) => {
  const authorization = req.headers.authorization
  let token = null

  // Retrieve token
  if (authorization && authorization.split(" ").length === 2) {
    token = authorization.split(" ")[1]
    console.log("Token: ", token)
  } else {
    console.log("Unauthorized user")
    return
  }

  // Verify JWT and check expiration date
  try {
    const decoded = jwt.verify(token, secretKey)

    if (decoded.exp < Date.now()) {
      console.log("Token has expired")
      return
    }

    // Permit user to advance to route
    next()
  } catch (e) {
    console.log("Token is not valid: ", err)
  }
}
```

Figure 20 Token verification

Here the `jsonwebtoken verify` call relies on the secret key we created for the login route (recall the token decoding tool from Lecture 7). If verification fails, the function will throw an error which we catch and handle at the bottom of the function. If the token was decoded successfully, we do a final check to determine that it remains valid. If it has expired, we return and stop processing. If it has not expired, we can run the `next()` function, permitting the `/api/update` handler to run. You can test this using Postman (grabbing your authorization token from the /users/login route):



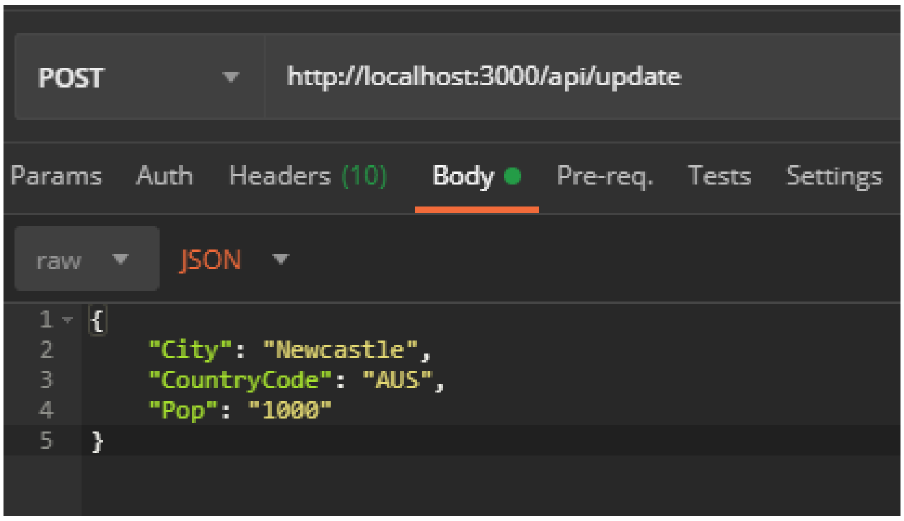
POST http://localhost:3000/api/update

Params Auth Headers (10) Body ● Pre-req. Tests Settings

Headers 9 hidden

	KEY	VALUE	DESCRIPTION
✓	Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFwaUBleGFtcGxlLmNvbSIsImV4cCI6MT ...	
	Key	Value	Description

Figure 21 Postman authenticated headers



```
http://localhost:3000/api/update
```

```
{
    "City": "Newcastle",
    "CountryCode": "AUS",
    "Pop": "1000"
}
```

Figure 22 Postman update body

Finally, as noted above, we must go through the code and replace all the console outputs with error responses of the form:

`res.status(401).json({error: true, message: "Unauthorized"}).`

If you don't do this, the request will just hang and the user will never receive a response. This is left as an exercise for the reader.

Final Notes

After this worksheet you should now have a general idea of how to implement authenticated routes. Be aware that directly implementing the code we have shown above will not give you a passing mark in this section of Assignment 2. You will still need to implement error messages, and extract repeating code. This tutorial is only a stepping stone, and you should not blindly copy the code without understanding the underlying material.