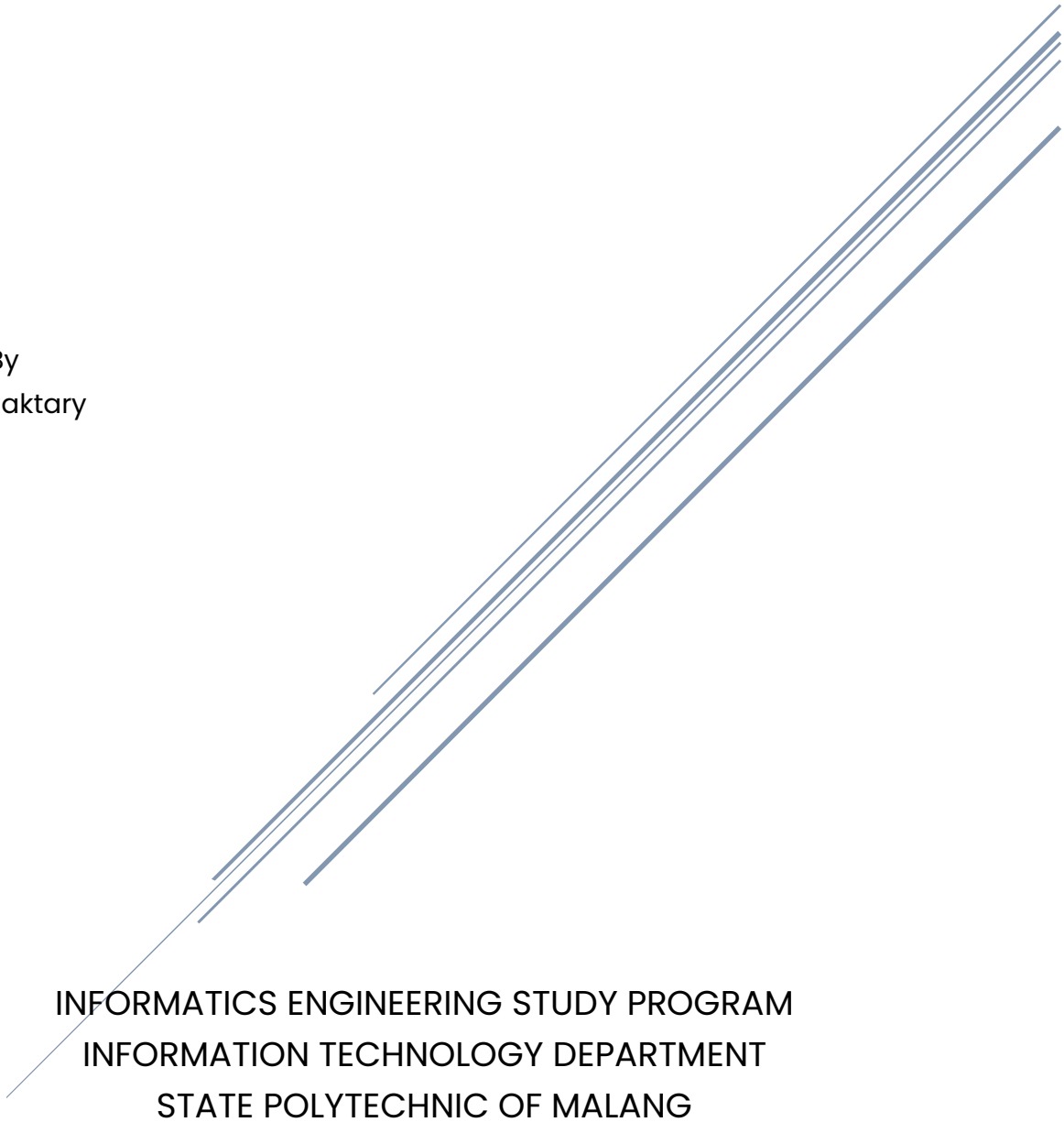
# GUIDE B04

## Create an Endpoint for Updating The User Profile

Arranged By
Omar Al-Maktary



INFORMATICS ENGINEERING STUDY PROGRAM
INFORMATION TECHNOLOGY DEPARTMENT
STATE POLYTECHNIC OF MALANG
2023

# Contents

# Create an Endpoint for Updating The User Profile

## Objectives

1. Students can create a PATCH endpoint for users of the application to update their profile data.
2. Students can create a web page to update users' profile data.

## Requirements

Having the correct hardware and software components is essential for ensuring the successful execution of the tasks outlined in this guide. The hardware configuration and software required for completing this guide tasks are as the following:

### Hardware Specifications

The minimum hardware specifications for running a NodeJS API application on the Windows operating system and using software such as Postman and Visual Studio Code are the following:

#### Minimum Requirements

- Processor: Intel Core i3 or equivalent.
- RAM: 4 GB.
- Storage: 500 GB HDD with at least 20 GB of available storage.
- Graphics: Integrated graphics card.
- Connectivity: Ethernet and Wi-Fi capabilities.

#### Recommended Requirements

- Processor: Intel Core i5 or equivalent.
- RAM: 8 GB or more.
- Storage: 256 GB SSD with at least 20 GB of available storage.
- Graphics: Integrated graphics card.
- Connectivity: Ethernet and Wi-Fi capabilities.

## Software required

It is important to have the correct software installed on your system to ensure that the application runs smoothly and meets performance expectations. The software required is as follows:

- Operating System: Windows 10 or later.
- NodeJS: Latest stable version installed.
- Visual Studio Code: Latest stable version installed.
- Postman: Latest stable version is installed.

### NPM Packages

- nodemon: Automatically restarts Node application on file changes.
- cross-env: Sets environment variables in a cross-platform way.
- jest: Creates and executes tests.
- jest-expect-message: Enhances Jest assertions with custom error messages.
- jest-image-snapshot: Adds image snapshot testing to Jest.
- puppeteer: Node library to control a headless Chrome or Chromium browser.
- supertest: Makes HTTP queries to the application and checks results.
- dotenv: Simplifies management of environment variables.
- express: NodeJS framework for creating apps with routing and middleware.
- ejs: Embedded JavaScript templating.
- express-ejs-layouts: Layout support for EJS in Express.
- mongoose: MongoDB object modeling library for NodeJS.
- bcryptjs: NodeJS library for hashing passwords with the bcrypt algorithm.
- cookie-parser: Parses cookies attached to the incoming HTTP requests.
- cors: Middleware for enabling Cross-Origin Resource Sharing (CORS) in Express.
- express-unless: Defining exceptions to other middleware functions in Express.
- jsonwebtoken: Manage authentication by creating and verifying tokens.

# Resource

- Documents: Guide B04
- Tests: api/testB04.test.js, web/testB04 .test.js

## Task Description

Students understand how to create two endpoints for updating a user's profile data and password. These endpoints will return the user data by using an access token. Students should also understand how to create a web interface that shows the edit data page.

## Start Coding

To create an endpoint for updating user data, students should understand the following tables which outline the structure and goals of the endpoint. Note that there is an additional header for the authorization which is the token obtained from the login or registration process. This token has the user id encoded which can be used to identify the user.

| **PATCH "/api/v1/profile" ENDPOINT STRUCTURE** | | | |
|---|---|---|---|
| ***API Endpoint Path*** | ***Request Method*** | ***Response Format*** | ***Description*** |
| "/api/v1/profile" | PATCH | JSON | Update the user's data. |
| **Additional Headers** | | | |
| ***Key*** | ***Value*** | | ***Description*** |
| "Authorization" | "Bearer accessToken" | | AccessToken is the value of the token obtained from the login process |
| **Request Parameters** | | | |
| ***Parameter*** | ***Type*** | | ***Description*** |
| "name" | String | | A String of the user's name. |
| "username" | String | | A String of the user's username. |
| "email" | String | | A String of the user's email. |
| **Response Parameters** | | | |
| ***Parameter*** | ***Type*** | | ***Description*** |
| "user" | Object | | An object for the user data. |
| "message" | String | | A message indicating the status of the response. |
| **Success Responses** | | | |
| ***HTTP Status Code*** | ***Response*** | | |
| 200 | { " user": { "name": "John Doe", "username": "johndoe", "email": "johndoe@gmail.com", "createdAt": "2023-02-20T07:32:14.786Z", "updatedAt": "2023-02-20T07:32:14.786Z", | | |

| | |
|---|---|
| | "id": "6424370fe2a9f3e77c1573ee"<br>},<br>"message": " Profile Updated Successfully"<br>} |
| **Error Responses** | |
| ***HTTP Status Code*** | ***Response*** |
| 401 | {message "Unauthorized"} |
| 400 | {message "Profile Update Failed"} |
| 500 | { error object } |

*Table 1 PATCH "/api/v1/profile" ENDPOINT STRUCTURE*

| **PATCH "/api/v1/profile/password" ENDPOINT STRUCTURE** | | | |
|---|---|---|---|
| ***API Endpoint Path*** | ***Request Method*** | ***Response Format*** | ***Description*** |
| "/api/v1/profile/password" | PATCH | JSON | Update the user's password. |
| **Additional Headers** | | | |
| ***Key*** | ***Value*** | | ***Description*** |
| "Authorization" | "Bearer accessToken" | | AccessToken is the value of the token obtained from the login process |
| **Request Parameters** | | | |
| ***Parameter*** | ***Type*** | | ***Description*** |
| "currentPassword" | String | | A String of the user's current password. |
| "newPassword" | String | | A String of the user's new password. |
| "confirmNewPassword" | String | | It should be the same as the new password. |
| **Response Parameters** | | | |
| ***Parameter*** | ***Type*** | | ***Description*** |
| "user" | Object | | An object for the user data. |
| "message" | String | | A message indicating the status of the response. |
| **Success Responses** | | | |
| ***HTTP Status Code*** | ***Response*** | | |
| 200 | {<br>" user": {<br>"name": "John Doe",<br>"username": "johndoe",<br>"email": "johndoe@gmail.com",<br>"createdAt": "2023-02-20T07:32:14.786Z",<br>"updatedAt": "2023-02-20T07:32:14.786Z",<br>"id": "6424370fe2a9f3e77c1573ee"<br>},<br>"message": " Password Updated Successfully" | | |

| | |
|---|---|
| | } |
| **Error Responses** | |
| ***HTTP Status Code*** | ***Response*** |
| 401 | {message "Unauthorized"} |
| 400 | {message "Password Update Failed"} |
| 400 | {message "New password do not match"} |
| 400 | {message "New password must be at least 8 characters"} |
| 400 | {message "Incorrect Password"} |
| 500 | { error object } |

*Table 2 PATCH "/api/v1/profile/password" ENDPOINT STRUCTURE*

Follow the steps below to complete the code for this guide document:

1. In the "auth.service.js" file, copy and complete the following code for checking the password.

```
async function checkPassword(id, password) {
  // Find user by id using FindById method
  const user = // Write your code here;
  // Check if the user exists and the password matches
  // Match password using bcrypt.compareSync
  if (// Write your code here) {
    // Return user
}
// If the user does not exist or the password does not match, throw an error
// The error message should be "Incorrect Password"
}
```

*Figure 1 Check The Password Function Code*

2. In the same file, copy and complete the following code for updating the user's data.

```
async function updateProfile(id, username, name, email) {
  // Find the user by id Using the findById() method
  const userData = // Write your code here
  // This is the user to check if the params are empty or not
  // If the params are empty, then use the data from the userData
  const update = {
    username: username == "" ? userData.username : username,
    name: name == "" ? userData.name : name,
    email: email == "" ? userData.email : email,
  };
  // Update the user using the update variable and the findByIdAndUpdate(id, update,
{new: true}) method
```

```
  // Set the new option to true
  const user = // Write your code here
  // Check if the user is not found
  // If the user is not found, throw an error
  // the error message should be "Profile Update Failed"
  if (!user) // Write your code here
  // Return the user.toJSON() and the message "Profile Updated Successfully"
}
```

*Figure 2 Updating Data Function Code*

3. In the same file, copy and complete the following code for updating the user's password.

```
async function updatePassword(id, password) {
  const update = {
    password: password,
  };
  // Find the user by id and update the password
  // Use the findOneAndUpdate(id, password, {new: true}) method
  const user = // Write your code here
  // If the user is not found, throw an error
  // The error message should be "Password Update Failed"
  if (!user) // Write your code here;
  // Return the user.toJSon and the message "Password Updated Successfully"
}
```

*Figure 3 Update Password Function Code*

4. Export the "checkPassword, updateProfile, updatePassword" functions at the end of the "auth.service.js" file.
5. In the "controllers/api/auth.controller.js" file, copy and complete the following code for handling update profile requests.

```
function updateProfile(req, res, next) {
  // Get user id from req.user.id
  // Create requiredFields array for validation
  // Required fields: username, name, email
  // Validate data
  validateData(req.body, res, requiredFields);
  // Call authServices.updateProfile
  // Send the id, username, name, and email
  authServices
    .updateProfile(id, username, name, email)
    .then((results) => // If successful, send the results with status 200)
```

```
    .catch((err) => // If error, call next(err));
}
```

*Figure 4 Update Profile Controller Function Code*

6. In the same file, copy and complete the following code for handling update password requests.

```
function updatePassword(req, res, next) {
  // Get user id from req.user.id
  // Required fields: currentPassword, newPassword, confirmNewPassword
  // Check if currentPassword is correct
  validateData(req.body, res, requiredFields);
  // Check if newPassword and confirmNewPassword match
  if (newPassword !== confirmNewPassword) {
    // If not, return 400 with message "New password do not match"
  }
  // Check if newPassword is at least 8 characters
  if (newPassword.length < 8) {
    // If not, return 400 with message "New password must be at least 8 characters"
  }
  // Call the checkPassword function from auth.service.js
  authServices
    .checkPassword(id, currentPassword)
    .then((results) => {
      // If checkPassword returns the user
      if (results) {
        // Hash the newPassword using bcrypt.hashSync with a salt of 10
        // Call the updatePassword function from auth.service.js
        authServices
          .updatePassword(id, password)
          .then((results) => {
            // if updatePassword returns the user, return 200 with the results
          })
          .catch((err) => // return the error in the next function
          // this if the updatePassword function returns an error);
      }
    })
    .catch((err) => // return the error in the next function
    // this if the checkPassword function returns an error);
}
```

*Figure 5 Update Password Controller Function Code*

7. Export the “updateProfile and updatePassword” functions at the end of the “controllers/api/auth.controller.js” file.
8. In the “routes/api/auth.routes.js” file, import the updateProfile and updatePassword functions from the API controller.
9. Finally, Create two new PATCH routes with the “/profile” and “/profile/password” paths.

## Running The API Application

For this guide and development purposes the command “npm run dev” is used to execute the command “nodemon server.js” which will run the “server.js” using the nodemon package. This package allows the server to reload if any changes occur in the code of the application.

Run the development command “npm run dev” in the terminal and notice the console message.

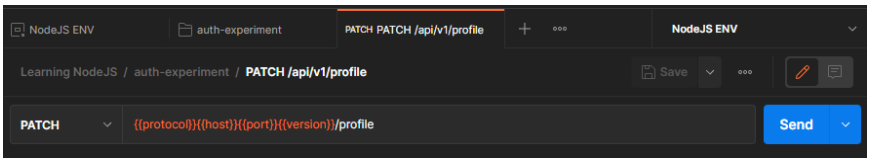
## Testing The API Application

In this section, several tests in different ways will be explored to verify the results of the student's work on this document.

### Using Postman

To test results from this guide on Postman, follow these steps:

1. In the “auth-experiment” collection, create two PATCH requests with the name “PATCH /api/v1/profile” and “PATCH /api/v1/profile/password”.
2. Make sure that the environment created is being used by selecting it from the top right option and then fill in the URL in the PATCH requests as the following: “{{protocol}}{{host}}{{port}}{{version}}/profile”




“{{protocol}}{{host}}{{port}}{{version}}/profile/password”

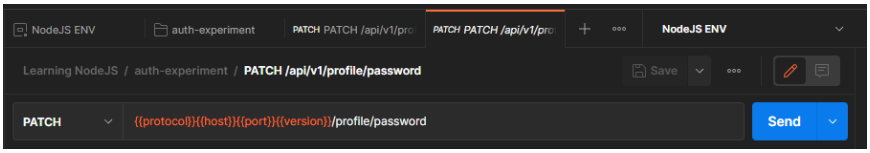
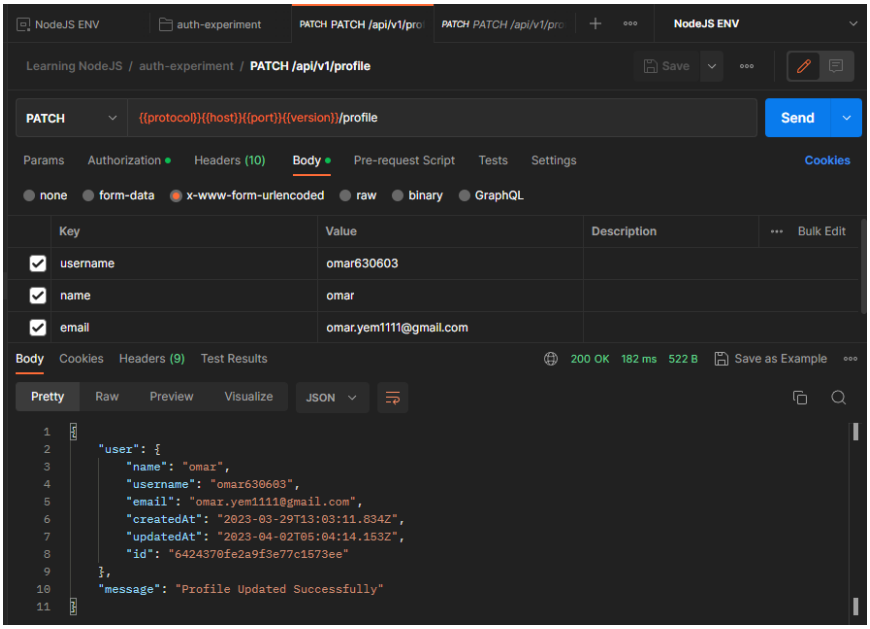



*Figure 6 Postman Request Configuration Bar*

3. If the token is still active then skip to step 7, and check the token by requesting the endpoint to fetch the user's profile. If the response returns the user's data then the token is still active.
4. In the Authorization tab, choose the "Bearer Token" as the type of authorization and fill the token field with "{{token}}" which should be created in the environment variables.
5. Request the login route created in the previous guide and save the token returned into the environment variable as the following:

*Figure 7 Postman Saving Token*

6. Make sure that the body field is filled correctly. The update profile endpoint requires the name, username, and email data. The update password endpoint requires the currentPassword. newPassword, confirmNewPassword data.
7. Go back to the PATCH requests and click "Send" then wait for the response. Postman should show results as the following if everything is working correctly:



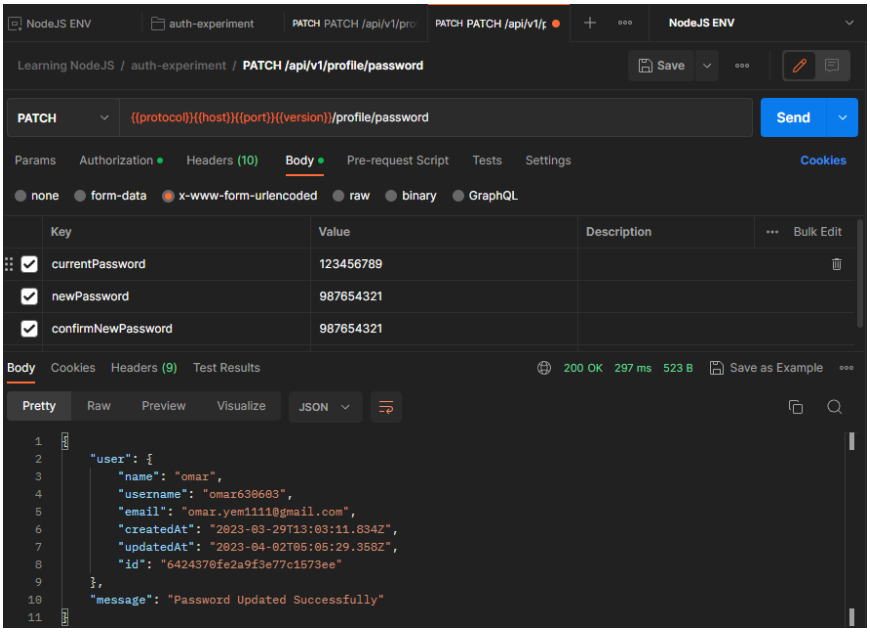



*Figure 8 Postman Request Results*

## Running The API Test File

Note: Sometimes the test will have an error of time limit, try to re-run the test or increase the testTimeout in scripts of the “package.json” file.

Verify results by following these steps:

1. Copy the file "testB04.test.js" from the "api" folder within the "tests" folder for this material to the "tests/api" folder of your project base directory.
1. Run the fill in the VSCode integrated terminal by running this command “npm run api-testB04” and then wait for results.
2. If everything is correct and working well the results in the terminal should look as the following:


*Figure 9 Successful Test Results*

3. If the test failed and it shows an error similar to the following figure, the error shows feedback for the cause of the error:


*Figure 10 Failed Test Results*

Try to find out why the test failed and fix it until the test result shows successful results.

## Creating The Web Interface

In this section, the web interface for the editing page will be created. The same basic endpoints explained in the previous sections will be implemented in a web page that can show the user's data.

To start working on the web interface, follow these steps:

1. In the "controllers/web/auth.controller.js" file, copy and complete the following code:

```
function updateProfile(req, res, next) {
  // Get the user id from the req.user.id
  if (req.method == "GET") {
  // If the method is GET, render the edit page in the auth/ folder
  // Data to be passed to the view:
  // title: "Auth-Experiment | Update Profile", user: req.user.data,
  // type: "profile", message: req.body.message,
 } else {
  // If the method is POST, update the user profile
  // Validate the required fields in the request body
  // the required fields are ["username", "name", "email"];
  const error = validateData(req.body, requiredFields);
```

```
    if (error !== "") {
      // If there is an error, set the method to GET and set the message
      // Return the updateProfile function
    }
    // Call the authServices.updateProfile function
    authServices
     .updateProfile(id, username, name, email)
      .then((results) =>
        // If the update is successful, render the edit page
        // Data to be passed to the view:
        // title: "Auth-Experiment | Update Profile", user: results.user,
        // type: "profile", message: results.message,
      )
    .catch((err) => {
      // If there is an error, set the method to GET and set the message
      // Return the updateProfile function
    });
  }
}
```

*Figure 11 "controllers/web/auth.controller.js" Update Profile Function Code*

Note that when rendering the edit page the data type is passed with a value of "profile". This data will be used to determine which inputs to show. In the "edit.ejs" file, the type data is checked and if it equals "profile" then it will show the inputs for the user's data. The data message passed to the view is used to show error messages.

2. In the same file, copy and complete the following code for updating the user's password.

```
function updatePassword(req, res, next) {
  // Get the user id from the req.user.id
  if (req.method == "GET") {
    // If the method is GET, render the edit page
    // Data to be passed to the view:
    // title: "Auth-Experiment | Update Password",  user: req.user.data,
    // type: "password", message: req.body.message,
  } else {
    // If the method is POST, update the user password
    // Validate the required fields in the request body
    // the required fields are ["currentPassword", "newPassword",
"confirmNewPassword"];
  const error = validateData(req.body, requiredFields);
```

```
  if (error !== "") {
    // If there is an error, set the method to GET and set the message
    // Return the updatePassword function
  }
  if (newPassword !== confirmNewPassword) {
    // If the new password and confirm new password do not match, set the method
to GET and set the message
    // Return the updatePassword function
  }
  if (newPassword.length < 8) {
    // If the new password is less than 8 characters, set the method to GET and set
the message
    // Return the updatePassword function
  }
  // Call the authServices.checkPassword function
  authServices
   .checkPassword(id, currentPassword)
    .then((results) => {
     // If the password is correct, hash the new password
    if (results) {
      // Call the authServices.updatePassword function
      authServices
      .updatePassword(id, password)
      .then((results) => {
        // If the update is successful, set the req.body.message to the
results.message
        // Call the getProfile function
      })
        .catch((err) => {
        // If there is an error with updating the password, set the method to GET and
set the message
        // Return the updatePassword function
      });
    }
   })
  .catch((err) => {
    // If there is an error with checking the password, set the method to GET and set
the message
    // Return the updatePassword function
   });
 }
}
```

*Figure 12 "controllers/web/auth.controller.js" Update Password Function Code*

Note that the data type is equal to the password which indicates that the view should show the password inputs.

3. Export the "updateProfile, updatePassword" functions at the end of the "controllers/web/auth.controller.js" file.
4. In the "routes/web/auth.routes.js" file, import the "updateProfile, updatePassword" functions from the web controller.
5. Add this new route as the following.

```
router.get("/profile/update", isLoggedIn, updateProfile);
router.post("/profile/update", isLoggedIn, updateProfile);
router.get("/profile/update/password", isLoggedIn, updatePassword);
router.post("/profile/update/password", isLoggedIn, updatePassword);
```

*Figure 13 "routes/web/auth.routes.js" New Routes*

6. In the "web/view/auth" folder create a new file named "edit.ejs".
7. In the "profile.ejs" file, copy the following code.

```
<% if (typeof message !== 'undefined') { %>
<div class="alert">
  <p class="message"><%= message %></p>
</div>
<% } %> <% if (typeof type !== 'undefined') { %> <% if (type == 'profile') { %>
<div class="container">
  <h1 class="title">Update your account's data</h1>
  <p class="description">Fill the form below to update your data</p>
</div>
<div class="container">
  <form action="/profile/update" method="POST">
    <div class="form-group">
      <label for="name">Name</label>
      <input
        type="text"
        name="name"
        id="name"
        class="form-control"
        value="<%= user.name %>"
      />
    </div>
    <div class="form-group">
      <label for="username">Username</label>
      <input
        type="text"
```

```
      name="username"
      id="username"
      class="form-control"
      value="<%= user.username %>"
     />
    </div>
    <div class="form-group">
     <label for="email">Email</label>
     <input
      type="email"
      name="email"
      id="email"
      class="form-control"
      value="<%= user.email %>"
     />
    </div>
    <button type="submit" class="btn btn-primary">Update</button>
  </form>
</div>
<% }else if(type == 'password'){ %>
<div class="container">
  <h1 class="title">Update your account's password</h1>
  <p class="description">Fill the form below to update your password</p>
</div>
<div class="container">
  <form action="/profile/update/password" method="POST">
    <div class="form-group">
     <label for="currentPassword">Current Password</label>
     <input
      type="password"
      name="currentPassword"
      id="currentPassword"
      class="form-control"
     />
    </div>
    <div class="form-group">
     <label for="newPassword">New Password</label>
     <input
      type="password"
      name="newPassword"
      id="newPassword"
      class="form-control"
     />
```

```
  </div>
  <div class="form-group">
   <label for="confirmNewPassword">Confirm Password</label>
   <input
    type="password"
    name="confirmNewPassword"
    id="confirmNewPassword"
    class="form-control"
   />
  </div>
  <button type="submit" class="btn btn-primary">Update</button>
 </form>
</div>
<% } %> <% } %>
```

*Figure 14 "profile.ejs" View Code*

Note that this code can show two different views even from the same file. This is done by using the "type" data that is passed from the controller.

## Running and Testing The Web Interface

If the application still running from the previous exercise then try to visit the following link http://localhost:8080/. If the application is not running in the terminal then use the command "npm run dev" to start the app.

If the user is not logged in then log in using the login page and it will automatically redirect to the profile page if the code is working correctly.

Upon visiting the URL, click the update data button or the update password button the web interface should look similar to the following:

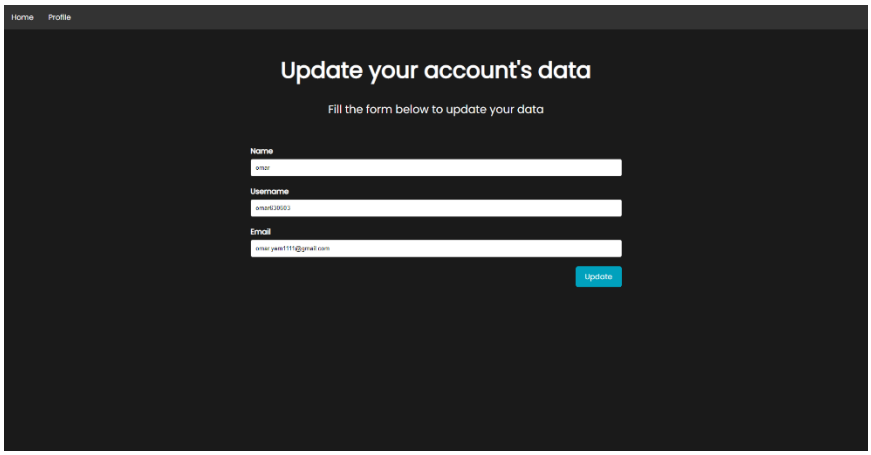



*Figure 15 Update Profile Page*

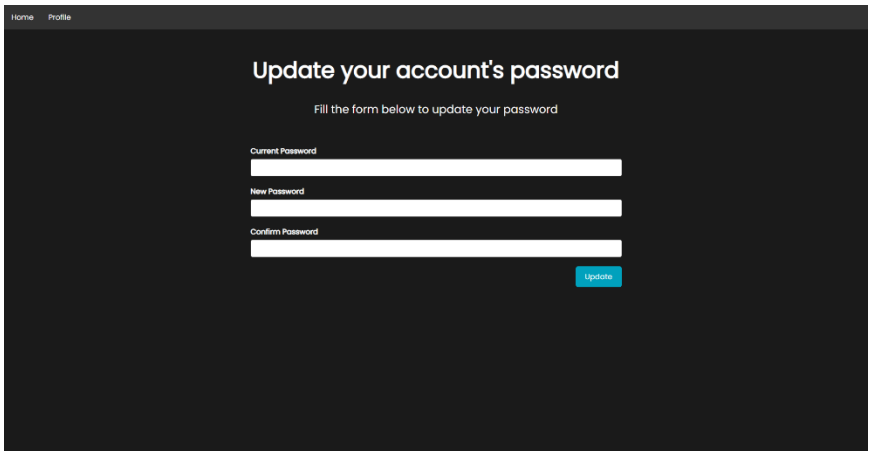



*Figure 16 Update Password Page*

Copy the file “testB04.test.js” and paste it to the folder “/tests/web” in your project directory. After that, run the command “npm run web-testB04” and notice the results.

If everything is correct and working well the results in the terminal should look as the following:

```
OMAR@LAPTOP-N1SUC5AB MINGW64 /d/Coding/Thesis/auth-experiment (main)
$ npm run web-testB04

> auth-experiment@1.0.0 web-testB04
> cross-env NODE_ENV=test jest -i tests/web/testB04.test.js --testTimeout=20000

 PASS  tests/web/testB04.test.js (7.967 s)
  Testing the edit page
    √ should login a user (511 ms)
    √ should have the name, username, email inputs in the edit page (299 ms)
    √ should not update the user's data if the inputs are empty (365 ms)
    √ should update the user's data (471 ms)
    √ should have three inputs for the password (244 ms)
    √ should update the user's password (886 ms)
    √ should not update the user's password if the current password is wrong (528 ms)
  Testing the login page image snapshots
    √ matches the expected styling for the edit page (535 ms)
    √ matches the expected styling for the edit password page (551 ms)

Test Suites: 1 passed, 1 total
Tests:       9 passed, 9 total
Snapshots:   2 passed, 2 total
Time:        8.089 s, estimated 11 s
Ran all test suites matching /tests\\web\\testB04.test.js/i.
```

*Figure 17 Successful Web Test Results*

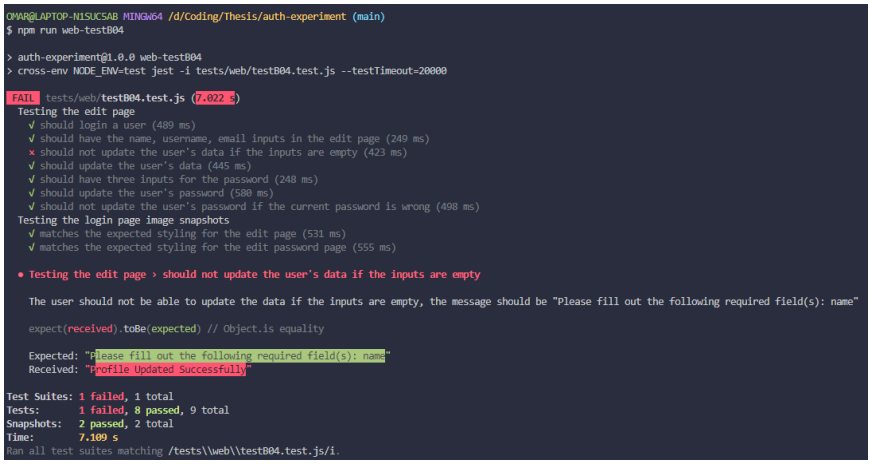



*Figure 18 Failed Web Test Results*

If you face a similar error try to figure out the reason for the problem until the test shows successful results.

## Results

This document outlines the intended outcomes of the fourth meeting for the second material on the topic of web programming using NodeJS. Students should learn how to create an API endpoint to update a user's data and password. The students should also learn how to create a new web interface to deal with the updating process.