# Programming Assignment 4

The goal of this assignment is for you to gain experience with back end web development. Specifically, you will be using the Node library Express to create the back end of a website. You will be connecting the front end you developed during Programming Assignment 3 with a backend that you will be writing. This means that you will not be graded on any of the front end components of this assignment since they were already evaluated during the previous assignment. Your grade will solely be based off of your backend endpoint implementation. Think of each of them as a function that need to be correct.

You and potentially one group member will be tasked with creating a basic website using Handlebars, JavaScript, and Express. You will earn points for this assignment by implementing specific endpoints. For each of the endpoints, I will give you a list of requirements that the endpoint must fulfill.

Each endpoint will be described as they would in industry. Usually every endpoint is specified as a task in whatever project management tool team uses. For every task there will be a user story, additional details/requirements, example input, and example output.

A few reminders about how groups work in CS316

- When you decide on a group you must inform me of your decision
- You may not switch groups throughout the semester

## Endpoint Descriptions

I will be breaking up the endpoints into two sections, GET and POST requests. The reason for this separation is that the GET requests will have much simpler implementations than their corresponding POST requests.

### GET - /user/login

As a user I want to be able to log in to the web application.

This page should not change depending on who is accessing it.

**Output**

```
html for the login page
```

---

### POST - /user/login

As a user I want to be able to log in to the web application.

Your accepted username and password combinations should not be hard coded in your JavaScript. This means you **must** use some form of long term storage solution to keep track of the username/password pairs. Normally, you would store this information in a database and take many different steps to encrypt that data. For this assignment you will not have to use a database or encryption. Instead I recommend that you use a storage library such as conf. This should allow you to store username/password pairs easily.

*It is very bad practice to store authentication information in this manner and you should never do this in industry!*

**Input**

```
{
    "username": "user",
    "password": "password"
}
```

This is just example input. The actual request will be application/x-www-form-urlencoded since it is being made by a HTML form.

**Output**

**on failure**

```
return the html for the login page. the login page should have an error
showing that the username/password pair is not valid.
```

**on success**

```
redirect to /user/:user_id for your own account
```

---

## GET - /user/new

As a user I want to be able to sign up for the web application.

This page should not change depending on who is accessing it.

**Output**

```
html for the sign up page
```

---

## POST - /user/new

As a user I want to be able to sign up for the web application.

**An important note for user creation:** Every user needs to have a unique identifier assigned to them. I recommend that you use a library that can generate a uuid/v1 such as uuid. While keeping track of a running increment may seem easy using uuid/v1 is probably easier and better a practice.

Your accepted username and password combinations should not be hard coded in your JavaScript. This means you **must** use some form of long term storage solution to keep track of the username/password pairs. Normally, you would store this information in a database and take many different steps to encrypt that data. For this assignment you will not have to use a database. Instead I recommend that you use a storage library such as conf. This should allow you to store username/password pairs easily.

*It is very bad practice to store authentication information in this manner and you should never do this in industry!*

**Input**

```
{
    "username": "user",
    "email": "user@example.com",
    "password": "password",
    "verified_password": "password",
    "phone": "1112223333"
}
```

**Output**

**if password and verified_password do not match**

```
keep them on the new user page and display an error saying that the
passwords do not match
```

**if the username already exists in the system**

```
keep them on the new user page and display an error saying that the
username is taken
```

**if the email already exists in the system**

```
keep them on the new user page and display an error saying that the
username is taken
```

**otherwise**

```
create a new user and redirect them to /user/login
```

---

## GET - /user/:user_id

As a user I want to be able to view and edit my account information.

The page should load and display the user's information that matches the given id in the path.

**Output**

**if the user does not exist**

```
you should return a status code 404 and inform the user that the page does
not exist
```

**if the user exists**

```
html for the sign up page filled in with the user's information
```

---

## POST - /user/:user_id

As a user I want to be able to edit my account information.

The response for this page should be html that resembles the behavior of corresponding GET request. The only difference is that the user may be updated through this request.

**Input**

```
{
    "username": "user",
    "email": "user@example.com",
    "phone": "1112223333"
}
```

Notice that you are given the username and email in the input. **You do not need to handle these two parameters if you do not plan on doing the extra credit.**

**Output**

**if the user does not exist**

```
you should return a status code 404 and inform the user that the page does
not exist
```

**otherwise**

```
update the user and return html for the user edit page but with a success
message at the top
```

---

## Rubric

One potential pain point in this assignment is going to be getting it to work with persistent storage like what is provided by the conf library. Because of this, you may earn partial credit by using hard coded values for some of the endpoints. However, 35 of the 100 points come from persistent storage requirements meaning that you should at least try to get one or two of the endpoints working with persistent storage.

- total 5 pts: GET - /user/login
    - 5 pts: sends file with correct status code
- total 20 pts: POST - /user/login
    - 5 pts: sends file with correct status code
    - 5 pts: logic works as described
    - 10 pts: logic works with persistent storage
- total 5 pts: GET - /user/new
    - 5 pts: sends file with correct status code
- total 20 pts: POST - /user/new
    - 5 pts: sends file with correct status code
    - 5 pts: logic works as described
    - 10 pts: logic works with persistent storage
- total 10 pts: GET - /user/:user_id
    - 5 pts: sends file with correct status code
    - 5 pts: logic works with persistent storage
- total 20 pts: POST - /user/:user_id
    - 5 pts: sends file with correct status code
    - 5 pts: logic works as described
    - 10 pts: logic works with persistent storage
- total 10 pts: HTML - Bootstrap/Handlebars Alert System
    - 5 pts: uses bootstrap's alert feature
    - 5 pts: one implementation in the base layout
- total 10 pts: HTML - 404 page
    - 5 pts: says that the resource was not found
    - 5 pts: **not** implemented as a redirect

## Extra Credit Opportunity

You may earn 10 pts extra credit by implementing a way for users to update both their username and email through the POST - /user/:user_id request. I'm not going to enumerate all of the edge cases this introduces, so think about everything that will have to be changed to implement this! I recommend that if you want to do this additional opportunity you plan it into your initial design, as adding it when complete may prove difficult.