Assignment #2: User Service

Goal

Over the course of the semester, we will be building features for a simple replicated game. In this first assignment, you will build a user service, which we will improve and integrate in future assignments and labs.

Your web service will be RESTful, which means:

- *A resource-based, uniform interface, called via HTTP*. The API consists entirely of HTTP URLs, utilizing the expected semantics of HTTP verbs and errors per the HTTP RFCs.
- *Minimal state*. Ideally, a RESTful service maintains no state; everything required to satisfy a request is contained in the request itself.
    - This assignment will maintain the state of users and sessions between requests, but those will be externalized to databases in future assignments.

You will develop your code to satisfy the Postman test suite provided with this document. The API is generally documented below, but the full set of expectations is effectively documented by the unit tests. Your goal is to implement functionality until **all 75 tests pass**.

Technology

You will implement your service using Node.js and Express. In addition to installing the **express** package, you may choose to use the following additional NPM packages:

- **base64url** to generate the user ID deterministically from the username
- **uuid4** to generate random session IDs.

These packages would be installed locally, not globally, so they can be restored via **npm** install. You may *not* use other npm packages, including packages that are designed to provide user sessions.

You should use the version of Node that we are using in class, per the procedure defined in Assignment 1. That is the version that will be used during testing.

Your code must listen on **localhost (127.0.0.1), port 3100.** The service will respond to requests made in this pattern: http://localhost:3100/api/v1/, per the API specification below and the Postman test suite.

The unit tests will be sending JSON objects, which you will interact with in code. In order to use them in code, Express needs to parse the JSON into objects. You can tell Express to do this by telling it to use the **express.json()** middleware – ideally right after creating the object. So, the creation of the Express app would look like this:

```
const app = express();
app.use(express.json());
```

Once you've established that middleware, you can find the data in the **req.body** object. So, in the route for Create User, you could get the username that the test sent with **req.body.username** or **req.body["username"]**.

In this first assignment, you will store user and session objects in local memory, in data structures in your JavaScript code. That means that the data will not be persisted outside of the process; when the process ends, the user objects are not retained. We will add persistence in a future assignment. Don't overcomplicate the storage - you could use an array of users, or you could have a single "users" object that maps user IDs to user objects. Similarly, you could have an array of sessions, or you could have a single "sessions" object that maps session IDs to session objects.

The response's status codes will contain standard HTTP response codes. When the operation succeeds with a 200-range status code, the response body will contain a JSON object with the public-facing representation of the resource. The Postman suite does expect relevant HTTP error codes under specific conditions.

Your project should not output to the console, for this assignment.

Authentication and Security

The web service is designed to create and manage users, so that users can identify themselves by logging in. When a user logs in, they receive a **session** resource from the server.

In this assignment, the session resource will be returned from the login API within the response body as a JSON object. The client will provide the session resource as part of the JSON in the request. When we integrate persistence in a future assignment, we will update the transmission mechanism as well.

In many of the APIs below, the client will need to authenticate by providing the session resource. If the session that the user provided is valid and from that user, then the server will allow the operation to proceed. Otherwise, it will need to return an appropriate HTTP error code.

You will note that this API will be *very* insecure without a HTTPS (TLS) and a certificate. We will add that in a future assignment.

Deployment

For this assignment, your server will run within your Linux environment (WSL 1).

We will run Node simply, by directly calling **node** with the primary JavaScript file, which must be called **assignment2.js**. You can and should use other JavaScript files to isolate elements of the implementation, using CommonJS's **module.exports** and **require**.

When generating your package.json with **npm init**, you can accept the other defaults (description, etc.).

Testing

We will use a web API testing tool called **Postman** for this project (link goes to download). You should not "sign in" to use Postman – that is for paid functionality useful to teams (which won't be used in this class).

There is a Postman unit-test collection JSON file accompanying this document, and a correct implementation will pass all 75 tests. It is already configured to run against localhost:3100. The collection can be imported via the Import button or File->Import.

Your submission will be graded in the same environment described in Assignment 1.

Submission

For this assignment, you will zip up:

- Assignment2.js, the primary Javascript file
- Your other Javascript files (modules)
- package.json
- package-lock.json

… in a single archive, without any subdirectories (i.e., it should *not* include node_modules). You will name the file YourAlias-CS261-2.zip, where YourAlias is the username you use to log into your email and lab PC.

To test your submission: extract it to a new location, enter **npm install** to re-acquire your dependencies, and then run it (i.e., **node assignment2.js**), and run the unit tests in Postman.

NOTE: It is easy to forget your package.json and package-lock.json, and it's easy to accidentally include the node_modules directory!

Grading

75 points: Functionality

Starting with 75, you will lose 1 point for each failed Postman unit test. There are 75 tests in 30 requests in the provided Postman collection JSON file.

25 points: Code Quality

We will review and give feedback on your code based on five factors, each worth 0 to 5 points. Grading will be a bell curve pretty tightly centered on 3—getting 0 for a factor will be extremely unlikely if you put in any effort at all; getting a 5 will be likewise be unlikely.

- *Organization*. Your program should be broken into logic units, both functions and files, arranged sensibly in files, with filenames that make sense. Pull duplicated code out into shared functions.
- *Commenting*. Functions should have comments blocks at the start explaining how to call them. Inline comments in the body of your functions should be present wherever necessary to explain *why* and *how* the code works. Comments should not document *what* the code does—that should be clear from the names of functions and variables.
    - You may use JSDoc (http://usejsdoc.org/), as it is a good standard, but it is not required.
- *Naming*. Names of functions, variables, and constants should be meaningful, expressive, and consistent. You can use any naming scheme you wish, as long as you *have* a naming scheme.
- *Consistency*. Whatever your conventions are for braces, tabs vs. spaces, etc., stick to them!
- *No cruft*. Make your code look "finished". Do not leave debugging detritus or commented-out sections of old code lying around.

- In general - it's okay for code to have a "debug" or "verbose" mode, but that behavior should be controlled by a configuration setting in your deployment and should not make the code more confusing to read.

Submission Penalties

In addition to the grading rubric above, you can receive these penalties if your submission is not correct:

- -5 if your submission files are not named as above.
- -5 if your submission includes additional files.
- -10 if your server operates on an IP and port other than **localhost:3100**.

Resources

User

- **username** (provided by the user)
- **password** (provided by the user)
- **avatar** (provided by the user – simply a string for this assignment)
- **ID** (generated by the server, and deterministically determined from the username. I suggest base64url encoding the username, as the user ID is used within URLs, below))

Note that whenever a User resource is returned *to the user who created it*, the password is included in the response. *If the User resource is provided to a different user, the password should be omitted from the response.*

Session

- **ID** (the ID of the user to whom the session belongs)
- **session** (a random ID generated by the server)

The ID is **not** returned with the resource. Only the ID should be returned (named "session").

Session IDs should be randomly generated for each login; the uuid4 package provides a reasonable implementation. Sessions should be stored by their session ID.

API Reference

This API reference describes the general API, but **the true reference is the Postman unit tests**. There are HTTP errors expected in various scenarios that **are not described here**.

Note that when authentication is required, that means that the **session** will be in the client request. If the session missing or invalid, the operation should fail. Also, the password may or may not be returned for Get/Find User depending on whether the session's User ID matches the requested User ID.

Create User

Path: /api/v1/users/

Verb: POST

Authenticated: No

Request Body:

- username
- password
- avatar

Response Body:

- id (the user ID)
- username
- password
- avatar

Creates a new user, succeeding if that user did not already exist.

Update User

Path: /api/v1/users/:id

Verb: PUT

Authenticated: Yes

Request Body:

- username
- password
- avatar

Response Body:

- id (the user ID)
- username
- password
- avatar

Updates the specified user. Only the owner of the session may update itself; a user cannot update another user.

Get User

Path: /api/v1/users/:id

Verb: GET

Authenticated: Yes

Request Body: none

Response Body:

- id (the user ID)
- username
- password (only if the user is the same as the owner of the session)
- avatar

Retrieves the specified user by ID (the generated value). The password is only provided if the user requested is the same as the owner of the session. The other fields are accessible for any user by any user.

Find User by Name

Path: /api/v1/users?username=username

Verb: GET

Authenticated: Yes

Request Body: None

Response Body:

- id (the user ID)
- username
- password (only if the user is the same as the owner of the session)
- avatar

Retrieves the specified user, searching by username. The password is only provided if the user requested is the same as the owner of the session. The other fields are accessible for any user by any user.

Note that the username in question is passed on a query string, not in the request body.

Login

Path: /api/v1/login

Verb: POST

Authenticated: No

Request Body:

- username
- password

Response Body:

- session (the ID of the session)

Creates a session for the user provided, *if* the user exists and the password matches. Only one session should be active per user.