Assignment #1: Robocat User Service 1.0

Goal

Over the course of the semester, we will be adding features to the Robocat multiplayer game from our textbook, ***Multiplayer Game Programming***. 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 connection state; everything required to satisfy a request is contained in the request itself.

app.post

Technology

You will implement your service using Node.js and Express, along with other NPM packages you may find useful. You should use the version of Node that we are using in class, which is the one installed on the class VM appliance unless told otherwise. Your code must listen on **localhost (127.0.0.1), port 3100.**

In this assignment, your user and session objects will be stored in local memory, in data structures in your JavaScript code. That means that the data will not be persisted outside of the process. We will add persistence in a future assignment.

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 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.

You should use other NPM packages as appropriate. In particular, you should look into base64url encoding, and random token generation. You probably do not need the body-parser package, as the version of Express we're using provides express.json() and other related middleware.

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.

data structure

user

session

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 VM. We will use AWS in a future assignment.

We will run Node simply, by directly calling **node** with the primary JavaScript file, which must be called **assignment1.js**. You can and should use other JavaScript files to isolate elements of the implementation.

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

Testing

We will use a web API testing tool called Postman for this project. It is already installed on the class VM appliance. There is a unit-test file accompanying this document, and a correct implementation will pass all tests. It is already configured to run against localhost:3100.

Your submission will be graded in the same VM environment provided for class. You could develop and test in any environment that supports Node, but I strongly suggest you test in the class VM appliance provided in Euclid.

Submission

For this assignment, you will zip up:

- assignment1.js, the primary Javascript file
- Your other Javascript files (modules)
- package.json
- package-lock.json

… in a single file, without any subdirectories (including node_modules). You will name the file YourAlias-CS261-1.zip, where YourAlias is the username you use to log into your email and lab PC.

Grading

75 points: Functionality

Starting with 75, you will lose 1 point for each failed Postman unit test. There are 75 tests in 31 requests in the provided Postman 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 JSDoc (http://usejsdoc.org/) 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.
- *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. 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. I suggest base64url encoding the username, as you cannot use base64 itself as it is used within URLs))

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 being 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) — key! not ID.
- **Token** (another random identifier generated by the server) tmp p/w

The ID is **not** returned with the resource.

You should find a library to generate a random string for session and token. Sessions should be stored by their 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** and the **token** must also be in the client request. If they're missing or invalid, the operation should fail.

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)
- token

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.

collection

create collection.

import json

localhost