Assignment 4 (12.5%)
Endpoint testing and security

T-213-VEFF, Web programming I, 2024-1

Reykjavik University - Department of Computer Science, Menntavegi 1, 101 Reykjavík

Deadline: Friday, April 5th 2024, 23:59

This is the fourth assignment in Web Programming I, with the topic of writing endpoint tests with JEST. **This assignment has to be done individually, no group work is permitted.**

1 Overview

In this assignment, you will write endpoint tests that test a modified solution to Assignment 3 (library app with books and genres resources). This will include the use of request signing for security.

Hint: Especially for more complicated endpoints, you are encouraged to try out the endpoints using Postman before writing any tests.

2 Setup

In the supplementary material, you find a modified solution to Assignment 3 in the starter pack folder. There are two endpoints drastically different from Assignment 3. They are:

- GET /api/v1/reset
- POST /api/v1/genres

The former resets the server state back to the initial (seed) data. This can be incredibly helpful when testing.

The latter has the same functionality as in Assignment 3 (to create a new genre). However, it uses HMAC request signing with SHA-256 as a hashing algorithm. The used secret (also known as salt) is "bookSecret", and the string that is hashed contains the (lowercase) method and the (lowercase) path, separated by a space.

In the test sub-folder, you will find a file called **index.test.js**, already set up so that you can start writing your tests. The current setup makes sure that the server state is reset before each test. This means you always have four genres and three books, and you know the exact value of all their properties (see the reset endpoint for details). You should use this knowledge when writing assertions in your tests (e.g., you know the title, author and genreIds of all books, so you know what a "get all books" request should return). Currently, the file includes a single test. Some npm scripts have been included so that you can easily run both the server and the tests:

- **npm install** fetches the required packages
- **npm start** starts the server
- **npm test** runs all tests in test/index.test.js (when you use npm test, you should **not** have your server running)

3 Task

Your task in this assignment is to write nine tests described in this section.

3.1 Basic Tests (3 tests)

For the following endpoints

- **GET /api/v1/books**
- **GET /api/v1/genres/:genreId/books/:bookId**
- **PATCH /api/v1/genres/:genreId/books/:bookId**

Write a test, for each of them, that captures the success case (the request succeeds, resulting in a 2xx response code). These tests should not only test the status but also validate the response as described below.

For endpoints that return arrays, assert the following:

- The status code should be as expected (e.g., 200, 201)
- The response body is present when it should be
- The return type is an array
- The array contains the right amount of elements

For endpoints that return individual objects, assert the following:

- The status code should be as expected (e.g., 200, 201)
- The response body is present
- The response body is as expected
  - Only the right attributes are in the body
  - All attributes have the expected values

3.2 Failure Tests (5 tests)

In addition to the 3 success cases described in 3.1, write 5 tests for the following failure cases:

- **PATCH /api/v1/genres/:genreId/books/:bookId** should fail when an existing book is addressed using an incorrect, but existing genreId
- **PATCH /api/v1/genres/:genreId/books/:bookId** should fail when a request is made with a non-empty request body that does not contain any valid property for a book (title, author, genreId)
- **GET /api/v1/genres/:genreId/books/:bookId** should fail when the book with the provided id does not exist
- **POST /api/v1/genres/:genreId/books/** should fail when the request body does not contain the author property
- **POST /api/v1/genres** should fail when missing the correct authorization

For each of the failure cases, assert the following:

- The status code should be correct
- The response body is present
- The error message is as expected

3.3 POST Genre Tests (1 test)

Assume that you have intercepted the request depicted in Figure 1. Write a test that demonstrates that you can run a "replay attack" with a different request body using the intercepted information. In your test case, it is sufficient to assert that a 201 response code has been returned. The intercepted request is also found in the file **intercept.txt** in the supplement material.

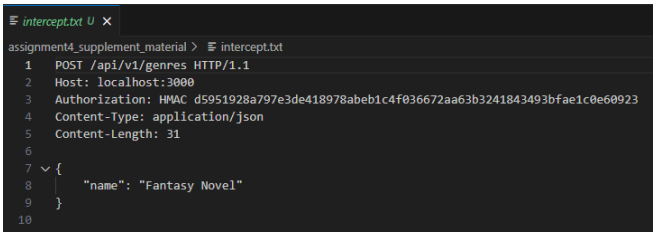

Figure 1: Intercepted POST request for genres

4 Requirements

The following requirements/best practices should be following

- The backend code should remain unchanged
- No extra files should be added. All test code should be added in test/index.test.js.
- The tests should be written using Jest and Supertest
- There are no restrictions on the ECMAScript (Javascript) version

5 Submission

The assignment is submitted via Gradescope. Submission should contain **only** the following:

- **index.test.js** containing all of your test code

Do **not** include the *node_ modules* folder, *package.json*, *package-lock.json* or any other file. No late hand-ins are accepted.

6 Grading and point deductions

Below you can see the criteria for grading, this list is not exhaustive but gives you an idea of how grading will be done for the project.

Critera	**Point deduction**
Basic Tests: 3 points	1 point per test. Point deductions when tests are incomplete (e.g., forgotten relevant assertions), do not work as intended (e.g., test always passes, leads to crashes), or do not have descriptive names/descriptions. No less than 0 points per test through deductions
Failure Tests: 5 points	1 point per test. Point deductions when tests are incomplete (e.g., forgotten relevant assertions), do not work as intended (e.g., test always passes, leads to crashes), or do not have descriptive names/descriptions. No less than 0 points per test through deductions
POST Genre Tests (replay attack): 2 points	1 point per test. Point deductions when tests are incomplete (e.g., forgotten relevant assertions), do not work as intended (e.g., test always passes, leads to crashes), or do not have descriptive names/descriptions. No less than 0 points per test through deductions
Other issues (using "var", regular for-loops, not using arrow functions, using the "promise" syntax instead of async/await, etc.)	Point deduction depending on severity.