Unit Testing

In this assignment, you'll be writing tests for your endpoints using the Jest and Supertest libraries. Supertest facilitates making HTTP requests in a testing environment, while assertions will be handled with the Jest library.

Initial Setup

- First, make sure to have Jest installed.
- Next install supertest.
- Create a `test` directory where you put your test files
- Optionally, create a script in your package.json file to run jest.

1. Test, Dev, and Prod Environment

Testing is a crucial step in the software development lifecycle as it ensures the correctness of functionalities. Therefore, it's essential to create a dedicated environment for testing, which mainly involves setting up and populating the database.

The development environment (dev env) is where developers write their code with specific configurations for the system and the database. On the other hand, the production environment (prod env) is where the application is served and receives real-world client requests. So it has different values for the configurations.

Each environment has its own set of configurations, which can be expressed as environment variables. To achieve this, we'll dynamically create a separate environment file based on another environment variable that describes the type of environment (test, dev, or prod).

> Note: We're going to use the same server instance of the database but different databases for test and dev environments.

- Create a `.env.test` file that contains variables for the testing environment. For example, the `DATABASE_URL` would be set to `mongodb://127.0.0.1:27017/test`.

- Create a `.env.dev` file that contains variables for the testing environment. For example, the `DATABASE_URL` would be set to `mongodb://127.0.0.1:27017/dev`.
- Create a utility function that dynamically loads the env file based on the env the application is running on

Your function would look like this

JavaScript

```
function envLoader(env = "dev") {
  switch (env) {
    case "test":
      // Load test env file
      return dotenv.config({
        path: ".env.test",
      });
    case "dev":
      // TODO: Load dev env file
      break;
    case "prod":
      // TODO: Load prod env file
      break;
    default:
      // TODO: Load dev env file
```

```
      break;
  }
}
```

- Now, replace calls for any other to `dotenv.config()` with this `envLoader` function and pass the type of the environment the server is running on, for example, `process.env['APP_ENV']` which has one of these values (test, dev, and prod)
- To dynamically set this value and run the application with it, we're going to use a cross-env library. By pre-prepending it to the server script, it would have the following behavior.

Unset

```
cross-env APP_ENV=test npm start ## Will start the server in a test env

cross-env APP_ENV=dev npm start ## Will start the server in a dev env

cross-env APP_ENV=prod npm start ## Will start the server in a prod env
```

2. API Tests

In this part of the assignment, you'll write some API tests for the authentication and challenges endpoints.

- Create a test suite file.

- You should **before all** the tests, insert some dummy data into the test database.

- Insert one coder and one manager.
- Insert two challenges created by that manager.
- Create two submissions, one for each challenge, the first one passes, and the second one fails.
- Create a test case that `should return an unauthorized error when the user is not logged in`.
- Create a test case that `should return an unauthorized error when the user passes an invalid token`.
- Create a test case that `should return a valid token when the correct credentials are passed to the login endpoint`.
- Create a test case that `should return all the challenges for the coder after login`.
- You must ensure that there's one `Completed` challenge and one `Attempted` challenge.
- You should **remove all** the test data that was previously inserted and close the Mongoose connection **after all** tests are done.