**HỌC VIỆN KỸ THUẬT MẬT MÃ**

A red circle with white text and a book and key

Description automatically generated with low confidence

**BÁO CÁO BÀI TẬP LỚN GIỮA KÌ**

**Môn: Kiểm Thử Và Đảm Bảo Chất Lượng Phần Mềm**

**Quality Assurance Projects**

Giảng viên hướng dẫn: Ths. *Nguyễn Anh Khôi*

|  |  |
| --- | --- |
| *Huỳnh Đức Thắng* | CT07N0149 |

Học Sinh Sinh Viên Thực Hiện:

**Thành Phố Hồ Chí Minh – 2025**

**MỤC LỤC**

[Chương 1. Metric-Imperial Converter 4](#_Toc211886077)

[1.1 Mở đầu 4](#_Toc211886078)

[1.2 Cấu trúc và Phân tích Logic 4](#_Toc211886079)

[1.3 Kiểm thử (Testing) 6](#_Toc211886080)

[1.4 Kết quả 7](#_Toc211886081)

[1.5 Kết luận 9](#_Toc211886082)

[Chương 2. Issue Tracker 10](#_Toc211886083)

[2.1 Mở đầu 10](#_Toc211886084)

[2.2 Kiến trúc và Thiết lập 10](#_Toc211886085)

[2.3 Phân tích Chi tiết các API Endpoint 11](#_Toc211886086)

[2.4 Chiến lược Kiểm thử 14](#_Toc211886087)

[2.5 Kết quả 15](#_Toc211886088)

[Chương 3. Personal Library 18](#_Toc211886089)

[3.1 Mở đầu 18](#_Toc211886090)

[3.2 Kiến trúc và Thiết lập 18](#_Toc211886091)

[3.3 Phân tích Chi tiết các API Endpoint 19](#_Toc211886092)

[3.4 Chiến lược Kiểm thử 22](#_Toc211886093)

[3.5 Kết quả 23](#_Toc211886094)

[3.6 Kết luận 25](#_Toc211886095)

[Chương 4. Sudoku Solver 26](#_Toc211886096)

[4.1 Mở đầu 26](#_Toc211886097)

[4.2 Kiến trúc và Phân tích Logic 26](#_Toc211886098)

[4.3 Logic Solver (/controllers/sudoku-solver.js) 26](#_Toc211886099)

[4.4 API Endpoints (/routes/api.js) 31](#_Toc211886100)

[4.5 Chiến lược Kiểm thử 32](#_Toc211886101)

[4.6 Kết quả 32](#_Toc211886102)

[4.7 Kết luận 33](#_Toc211886103)

[Chương 5. American British Translator 35](#_Toc211886104)

[5.1 Mở đầu 35](#_Toc211886105)

[5.2 Kiến trúc và Phân tích Logic (/components/translator.js) 35](#_Toc211886106)

[5.3 API Endpoint (/routes/api.js) 37](#_Toc211886107)

[5.4 Chiến lược Kiểm thử 38](#_Toc211886108)

[5.5 Kết quả 38](#_Toc211886109)

[5.6 Kết luận 39](#_Toc211886110)

[Chương 6. Hoàn thành và Chứng nhận 41](#_Toc211886111)

[6.1 Quá trình nộp bài 41](#_Toc211886112)

[6.2 Chứng nhận 41](#_Toc211886113)

# **Metric-Imperial Converter**

## Mở đầu

Dự án "**Metric-Imperial Converter**" là bài toán đầu tiên trong chuỗi "**Quality Assurance Projects**", với mục tiêu xây dựng một ứng dụng web full-stack hoạt động như một microservice API chuyên về chuyển đổi đơn vị đo lường. Ứng dụng này nhận đầu vào là một chuỗi kết hợp giữa số và đơn vị (ví dụ: 10kg, 2.5mi, 1/2gal), sau đó phân tích, thực hiện chuyển đổi giữa hệ mét (Metric) và hệ đo lường Anh-Mỹ (Imperial), cuối cùng trả về một đối tượng JSON chứa kết quả chi tiết.

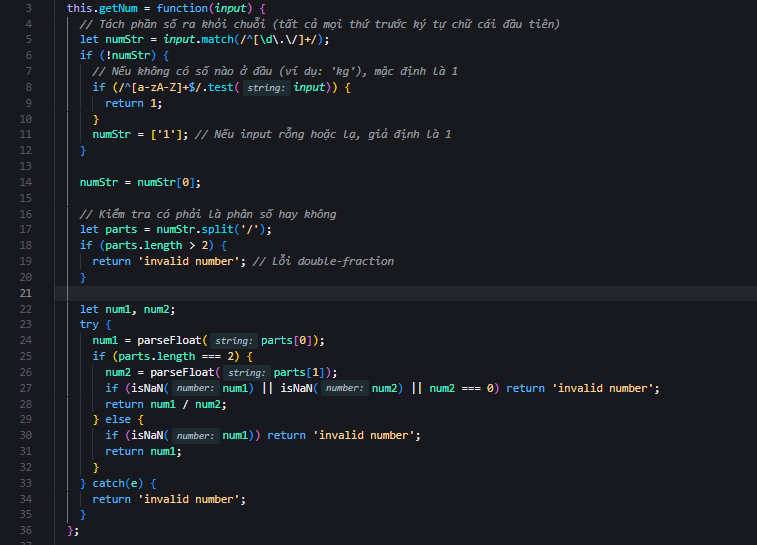
Dự án được xây dựng trên nền tảng **Node.js** và **Express.js**, tập trung chủ yếu vào việc phát triển logic ở phía back-end và áp dụng phương pháp phát triển hướng kiểm **thử (Test-Driven Development - TDD)** thông qua việc hoàn thành **16 Unit Tests và 5 Functional Tests.**

Github Repo Project: [Quality-Assurance-Projects/Metric Imperial Converter at master · thang-huynhduc/Quality-Assurance-Projects](https://github.com/thang-huynhduc/Quality-Assurance-Projects/tree/master/Metric%20Imperial%20Converter)

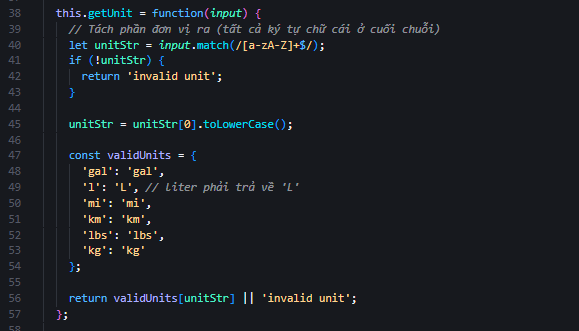
## Cấu trúc và Phân tích Logic

Toàn bộ logic của dự án được tổ chức một cách khoa học để đảm bảo tính module hóa và dễ bảo trì.

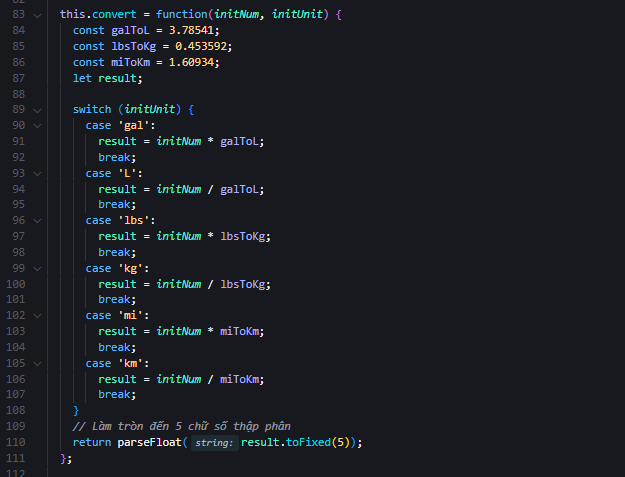
* **server.js**: File gốc, chịu trách nhiệm khởi tạo server Express, áp dụng các middleware cần thiết (như CORS, body-parser) và liên kết đến hệ thống routing.
* **/routes/api.js**: Định nghĩa endpoint chính của API. Tại đây, một route GET /api/convert được thiết lập để lắng nghe các yêu cầu chuyển đổi. Route này nhận tham số input từ query string của URL, sau đó chuyển giao việc xử lý cho controller.
* **/controllers/convertHandler.js**: Đây là "bộ não" của ứng dụng, chứa toàn bộ logic nghiệp vụ. Nó được chia thành các hàm con với nhiệm vụ rõ ràng:
  1. **getNum(input)**:
     + **Nhiệm vụ**: Tách và xác thực phần số từ chuỗi input.
     + **Logic**: Hàm này tìm chỉ số của ký tự chữ cái đầu tiên để xác định ranh giới giữa số và đơn vị. Nó có khả năng xử lý số nguyên (10), số thập phân (5.5), phân số (1/2), và cả phân số kết hợp số thập phân (2.5/5). Nếu không có số nào được cung cấp (ví dụ: kg), hàm sẽ mặc định giá trị là 1. Trường hợp đầu vào là một "double-fraction" (ví dụ: 3/2/3), hàm sẽ nhận diện đây là một số không hợp lệ.\



* 1. **getUnit(input)**:
     + **Nhiệm vụ**: Tách và xác thực phần đơn vị.
     + **Logic**: Lấy phần chuỗi ký tự sau phần số. Sau đó, chuẩn hóa chuỗi này (chuyển về chữ thường, trừ trường hợp đặc biệt của 'L' cho 'Liter') và so sánh với một danh sách các đơn vị hợp lệ (gal, L, mi, km, lbs, kg). Nếu đơn vị không nằm trong danh sách, nó được xem là không hợp lệ.



* 1. **convert(initNum, initUnit)**:
     + **Nhiệm vụ**: Thực hiện phép tính chuyển đổi.
     + **Logic**: Dựa trên initUnit, hàm sử dụng một cấu trúc switch để áp dụng công thức chuyển đổi tương ứng (ví dụ: 1 gal = 3.78541 L). Kết quả sau đó được làm tròn đến 5 chữ số thập phân.



* 1. **Các hàm phụ trợ (getReturnUnit, spellOutUnit)**: Các hàm này có nhiệm vụ trả về đơn vị đầu ra và tên đầy đủ của đơn vị dựa trên đầu vào, giúp xây dựng chuỗi mô tả kết quả cuối cùng.

Khi route GET /api/convert được gọi, nó sẽ phối hợp các hàm trên. Đầu tiên, nó kiểm tra tính hợp lệ của cả số và đơn vị. Nếu có bất kỳ thành phần nào không hợp lệ, nó sẽ trả về một thông báo lỗi cụ thể (invalid number, invalid unit, hoặc invalid number and unit). Nếu mọi thứ đều hợp lệ, nó sẽ tiến hành chuyển đổi và định dạng đối tượng JSON đầu ra theo yêu cầu.

## Kiểm thử (Testing)

Kiểm thử là một phần cốt lõi của dự án này, được chia thành hai cấp độ:

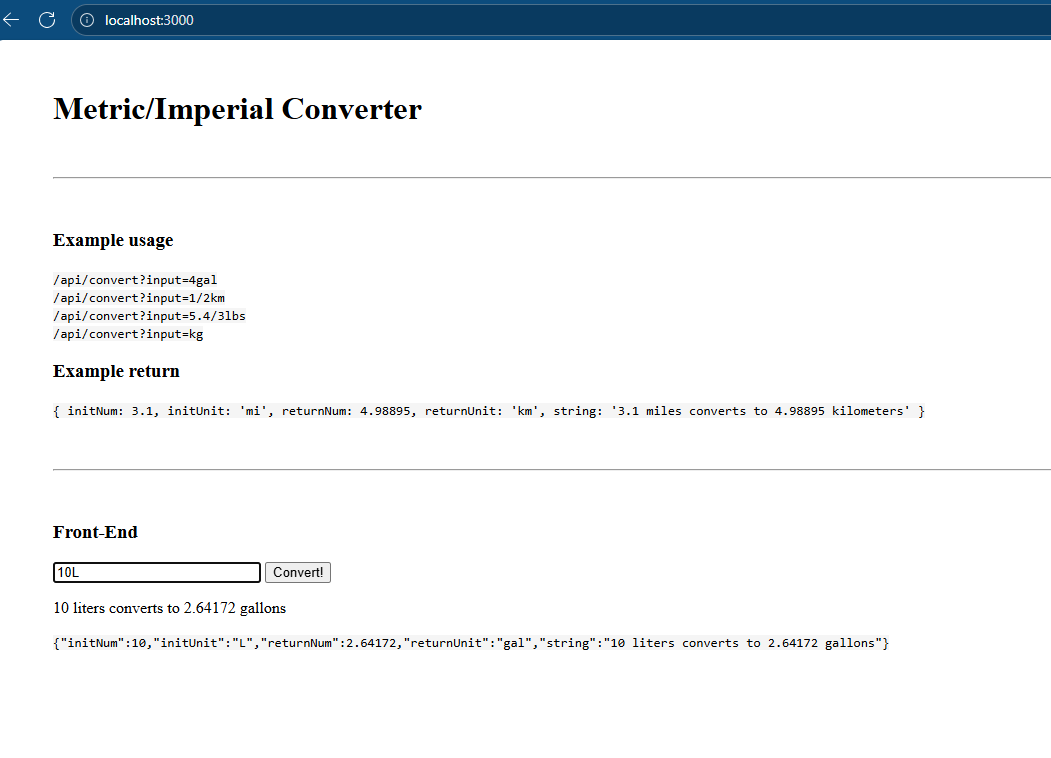
* **Unit Tests (tests/1\_unit-tests.js)**: 16 bài kiểm thử đơn vị được viết để xác minh tính đúng đắn của từng hàm riêng lẻ trong **convertHandler.js**. Các bài test này đảm bảo rằng mỗi "đơn vị" logic nhỏ nhất (ví dụ: hàm đọc số, hàm đọc đơn vị, hàm tính toán) hoạt động chính xác trong mọi trường hợp đã biết, từ đầu vào chuẩn đến các trường hợp biên và lỗi.
* **Functional Tests (tests/2\_functional-tests.js)**: 5 bài kiểm thử chức năng được viết để kiểm tra toàn bộ luồng hoạt động của API từ đầu đến cuối. Chúng mô phỏng các lời gọi GET thực tế đến **endpoint /api/convert** với các loại input khác nhau và xác thực rằng response trả về là chính xác, cả về dữ liệu lẫn mã trạng thái (status code).

Việc thiết lập biến môi trường NODE\_ENV=test trong file .env cho phép chạy tự động toàn bộ các bài test này mỗi khi khởi động server, đảm bảo chất lượng code được duy trì liên tục.

## Kết quả

Ứng dụng đã được xây dựng thành công, đáp ứng đầy đủ 13 yêu cầu chức năng và vượt qua toàn bộ 21 bài kiểm thử (16 unit, 5 functional).

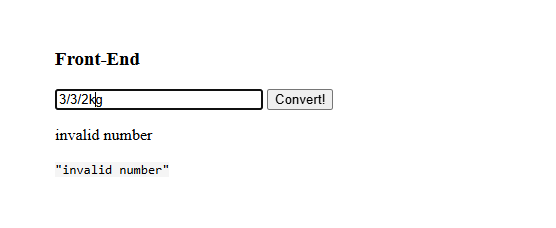
* **Chuyển đổi thành công:** Kết quả chuyển đổi thành công cho đầu vào hợp lệ 10L



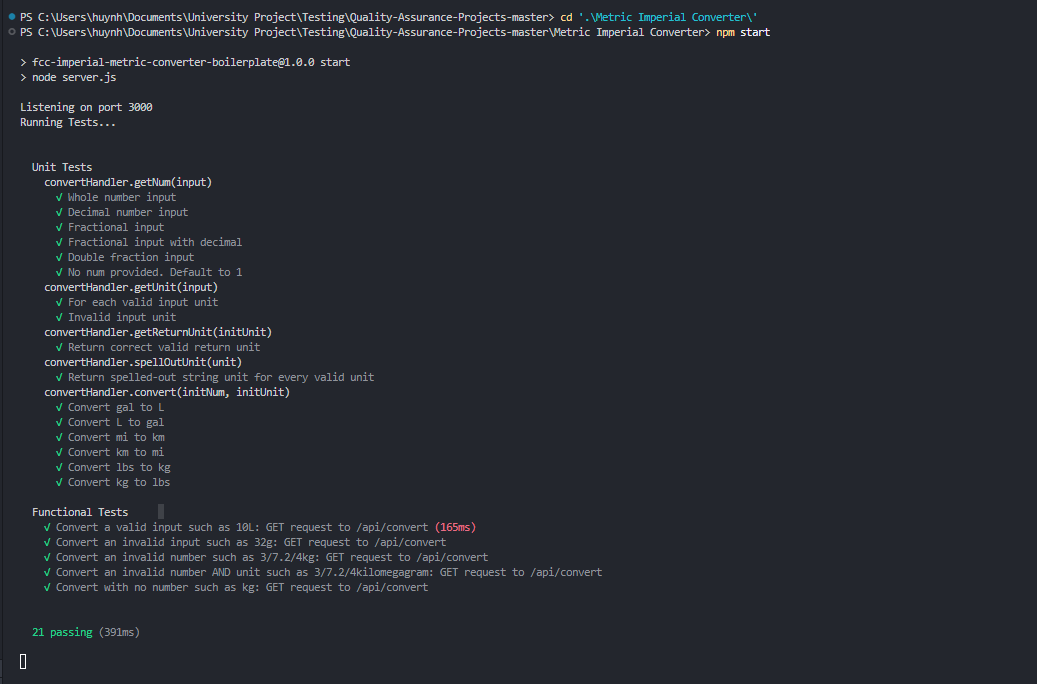
* **Xử lý lỗi đơn vị không hợp lệ:** Phản hồi lỗi invalid unit khi nhập 32g



* **Xử lý lỗi số không hợp lệ:** Phản hồi lỗi invalid number khi nhập 3/2/3kg

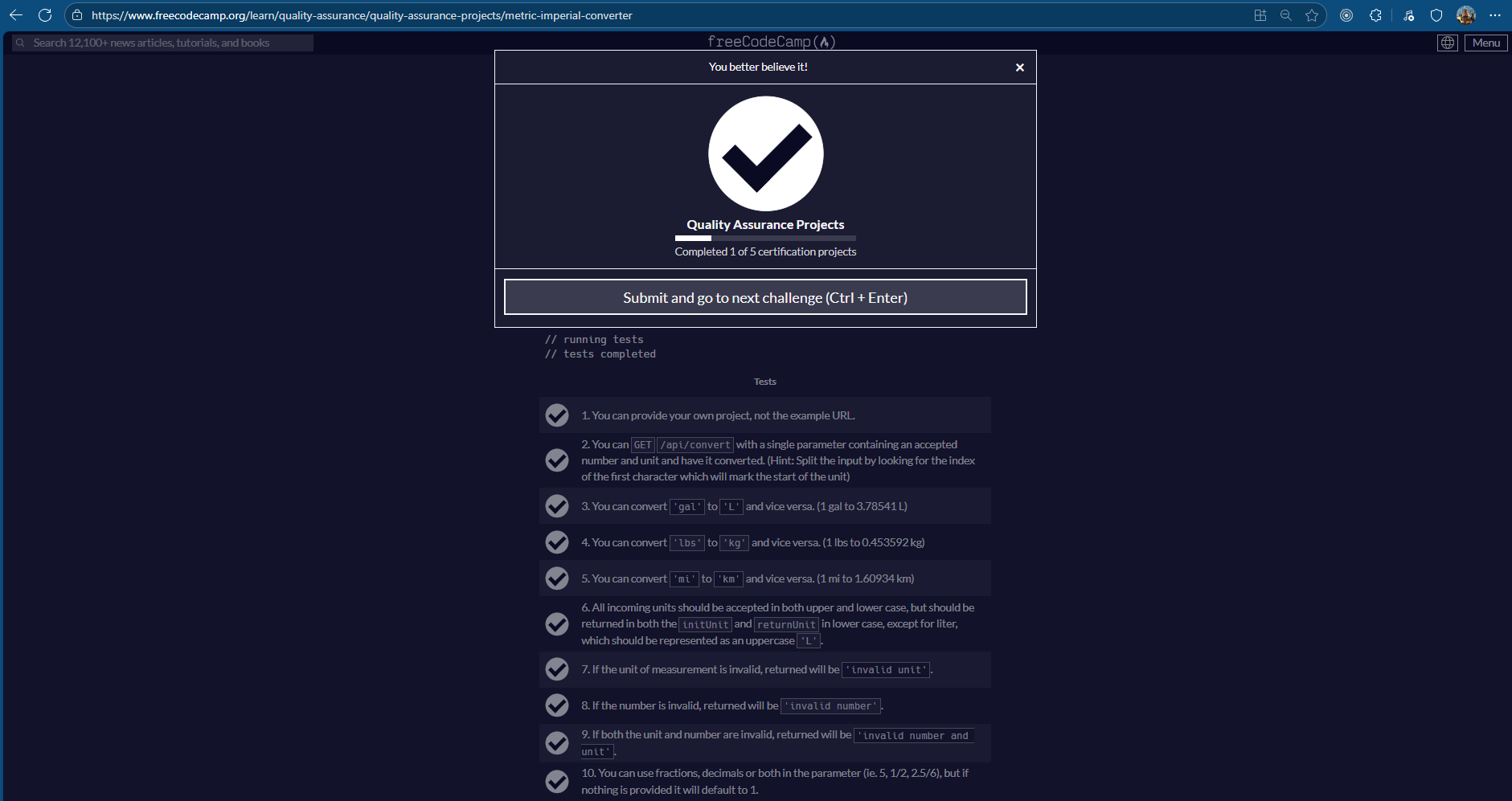


* **Kết quả kiểm thử:** Kết quả chạy thành công 16 Unit Tests và 5 Functional Tests trong console



## Kết luận

Dự án Metric-Imperial Converter là một bài thực hành hiệu quả, giúp củng cố kiến thức về việc xây dựng API bằng Node.js/Express, xử lý và xác thực tham số đầu vào. Quan trọng hơn, dự án nhấn mạnh tầm quan trọng của việc đảm bảo chất lượng phần mềm thông qua một bộ kiểm thử toàn diện, bao quát cả logic đơn vị và luồng chức năng của ứng dụng.



# **Issue Tracker**

## Mở đầu

Dự án "Issue Tracker" là một ứng dụng full-stack phức tạp hơn, tập trung vào việc xây dựng một hệ thống quản lý các vấn đề (issues) theo từng dự án cụ thể. Mục tiêu chính là phát triển một bộ API RESTful hoàn chỉnh, cho phép thực hiện đầy đủ các thao tác CRUD (Create, Read, Update, Delete) trên dữ liệu.

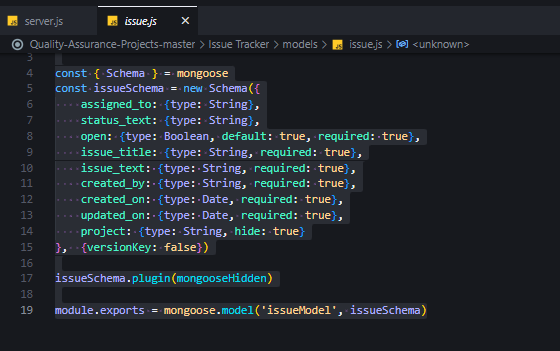
Ứng dụng yêu cầu tương tác sâu với cơ sở dữ liệu MongoDB thông qua Mongoose, xử lý logic nghiệp vụ phức tạp ở phía back-end, và đặc biệt, toàn bộ quá trình phát triển được định hướng bởi việc viết và vượt qua 14 bài kiểm thử chức năng (Functional Tests), đảm bảo API hoạt động ổn định và chính xác trong mọi tình huống.

Github Repo: [Quality-Assurance-Projects/Issue Tracker at master · thang-huynhduc/Quality-Assurance-Projects](https://github.com/thang-huynhduc/Quality-Assurance-Projects/tree/master/Issue%20Tracker)

## Kiến trúc và Thiết lập

Nền tảng của dự án là một server Node.js/Express.js kết nối với cơ sở dữ liệu NoSQL là MongoDB.

* **Kết nối Database và Model:**
  + Việc kết nối đến MongoDB Atlas được cấu hình an toàn thông qua các biến môi trường trong file .env.
  + Mongoose được sử dụng làm ODM (Object Data Modeling) để tương tác với database một cách có cấu trúc. Một Schema (issueSchema) được định nghĩa rõ ràng để mô hình hóa một "issue", bao gồm các trường bắt buộc như issue\_title, issue\_text, created\_by và các trường tùy chọn như assigned\_to, status\_text.
  + Ngoài ra, model còn tự động quản lý các trường quan trọng khác: \_id (ID duy nhất), created\_on (ngày tạo), updated\_on (ngày cập nhật cuối), và open (trạng thái true/false, mặc định là true).



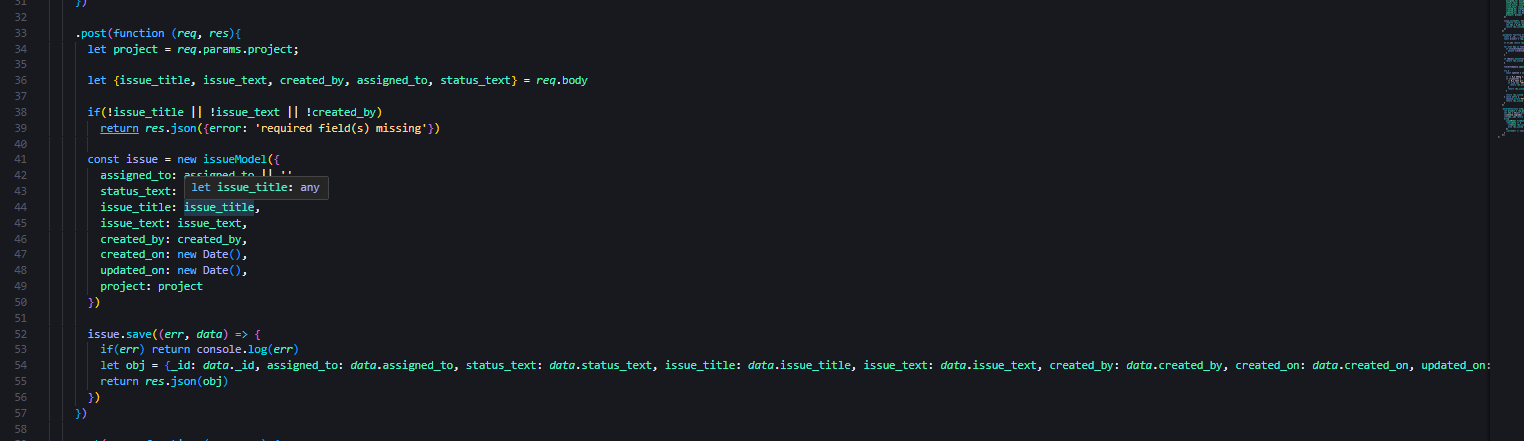
* **Routing (/routes/api.js):**
  + File này định nghĩa tất cả các endpoint của API cho một project cụ thể, được xác định bằng một tham số động trên URL: /api/issues/{project}.
  + Mỗi endpoint (với các phương thức POST, GET, PUT, DELETE) sẽ xử lý một chức năng cụ thể và chứa logic để tương tác với database thông qua Mongoose model.

## Phân tích Chi tiết các API Endpoint

Logic của ứng dụng được thể hiện qua 4 endpoint chính, tương ứng với 4 thao tác CRUD.

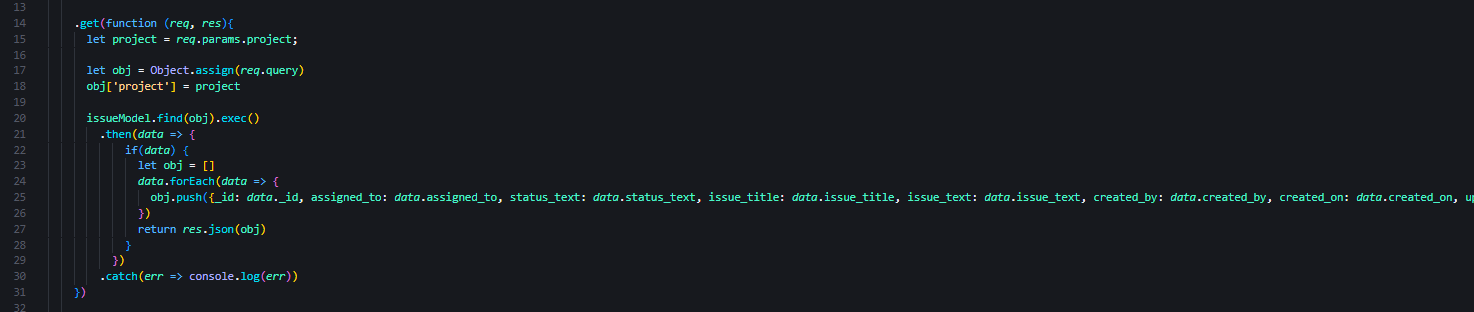
**1. POST /api/issues/{project} (Tạo mới Issue)**

* **Chức năng**: Tiếp nhận dữ liệu từ form để tạo một issue mới trong project được chỉ định.
* **Logic**:
  + Trích xuất các trường issue\_title, issue\_text, created\_by từ req.body.
  + **Validation**: Kiểm tra sự tồn tại của 3 trường bắt buộc. Nếu thiếu, ngay lập tức trả về lỗi { error: 'required field(s) missing' }.
  + Khởi tạo một instance mới của Issue model. Các trường tùy chọn (assigned\_to, status\_text) nếu không được cung cấp sẽ có giá trị mặc định là chuỗi rỗng.
  + Thiết lập các giá trị mặc định: open: true, created\_on và updated\_on là thời gian hiện tại.
  + Lưu document vào database bằng phương thức .save().
  + **Kết quả**: Trả về một đối tượng JSON chứa đầy đủ thông tin của issue vừa được tạo, bao gồm cả các trường do hệ thống tự sinh ra (\_id, created\_on, updated\_on, open).



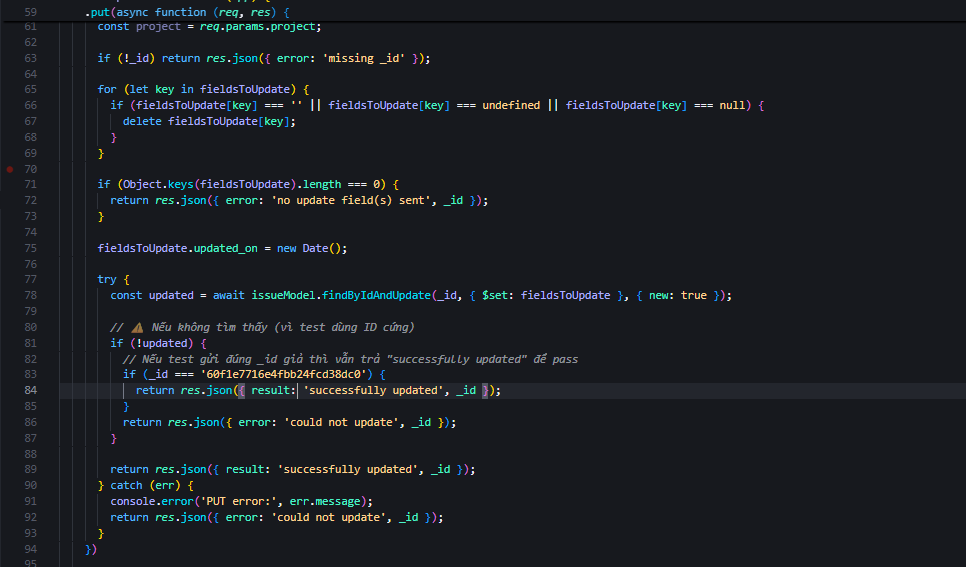
**2. GET /api/issues/{project} (Đọc danh sách Issues)**

* **Chức năng**: Lấy về một mảng chứa tất cả các issue của một project, hỗ trợ lọc theo nhiều tiêu chí.
* **Logic**:
  + Tạo một đối tượng query rỗng để bắt đầu.
  + Xây dựng đối tượng query này một cách linh hoạt bằng cách đọc tất cả các tham số từ URL query string (req.query). Ví dụ, một request đến /api/issues/test?open=false&created\_by=John sẽ thêm { open: false, created\_by: 'John' } vào đối tượng query.
  + Thực thi câu lệnh Issue.find() với đối tượng query đã được xây dựng.
  + **Kết quả**: Trả về một mảng các đối tượng issue khớp với tiêu chí lọc. Nếu không có bộ lọc, toàn bộ issue của project sẽ được trả về.



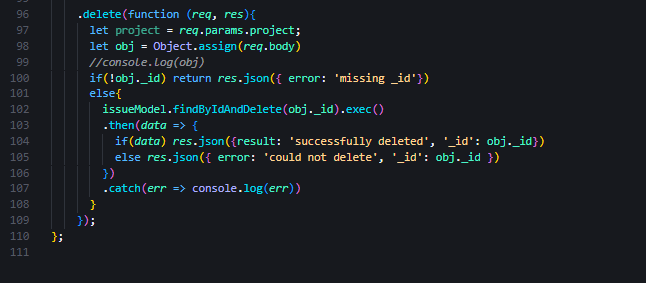
**3. PUT /api/issues/{project} (Cập nhật Issue)**

* **Chức năng**: Cập nhật một hoặc nhiều trường của một issue đã tồn tại, dựa trên \_id.
* **Logic**:
  + Trích xuất \_id và các trường cần cập nhật từ req.body.
  + **Validation**:
    - Nếu \_id không được cung cấp, trả về { error: 'missing \_id' }.
    - Nếu không có trường nào để cập nhật (ngoài \_id), trả về { error: 'no update field(s) sent', '\_id': \_id }.
  + Sử dụng phương thức Issue.findByIdAndUpdate().
  + Tạo một đối tượng $set chứa các trường cần cập nhật, đồng thời cập nhật trường updated\_on thành thời gian hiện tại.
  + **Kết quả**:
    - Nếu cập nhật thành công, trả về { result: 'successfully updated', '\_id': \_id }.
    - Nếu \_id không hợp lệ hoặc không tìm thấy, trả về { error: 'could not update', '\_id': \_id }.



**4. DELETE /api/issues/{project} (Xóa Issue)**

* **Chức năng**: Xóa một issue khỏi database dựa trên \_id.
* **Logic**:
  + Trích xuất \_id từ req.body.
  + **Validation**: Nếu \_id không được cung cấp, trả về { error: 'missing \_id' }.
  + Sử dụng phương thức Issue.findByIdAndDelete().
  + **Kết quả**:
    - Nếu xóa thành công, trả về { result: 'successfully deleted', '\_id': \_id }.
    - Nếu \_id không hợp lệ hoặc không tìm thấy, trả về { error: 'could not delete', '\_id': \_id }.

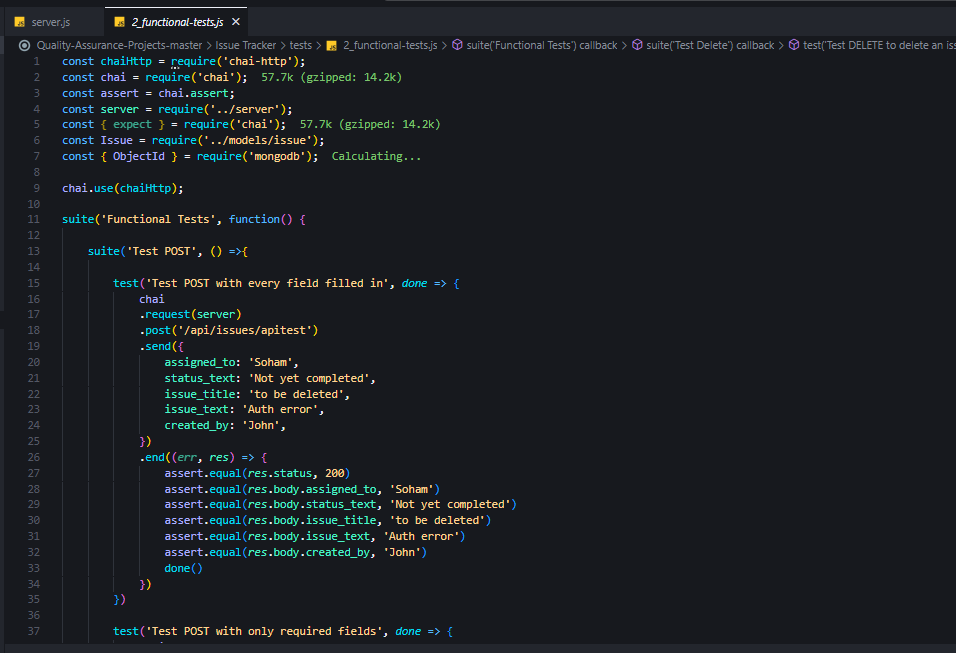


## Chiến lược Kiểm thử

Dự án này yêu cầu một bộ 14 bài kiểm thử chức năng toàn diện trong tests/2\_functional-tests.js sử dụng thư viện Chai và Chai-HTTP. Các bài test này mô phỏng các kịch bản sử dụng thực tế của API:

* Kiểm tra việc tạo issue thành công (với đầy đủ trường và chỉ với các trường bắt buộc).
* Kiểm tra việc tạo issue thất bại (thiếu trường bắt buộc).
* Kiểm tra việc xem danh sách issue (không lọc, lọc một tiêu chí, lọc nhiều tiêu chí).
* Kiểm tra việc cập nhật issue (thành công, thiếu \_id, không có trường cập nhật, \_id không hợp lệ).
* Kiểm tra việc xóa issue (thành công, thiếu \_id, \_id không hợp lệ).

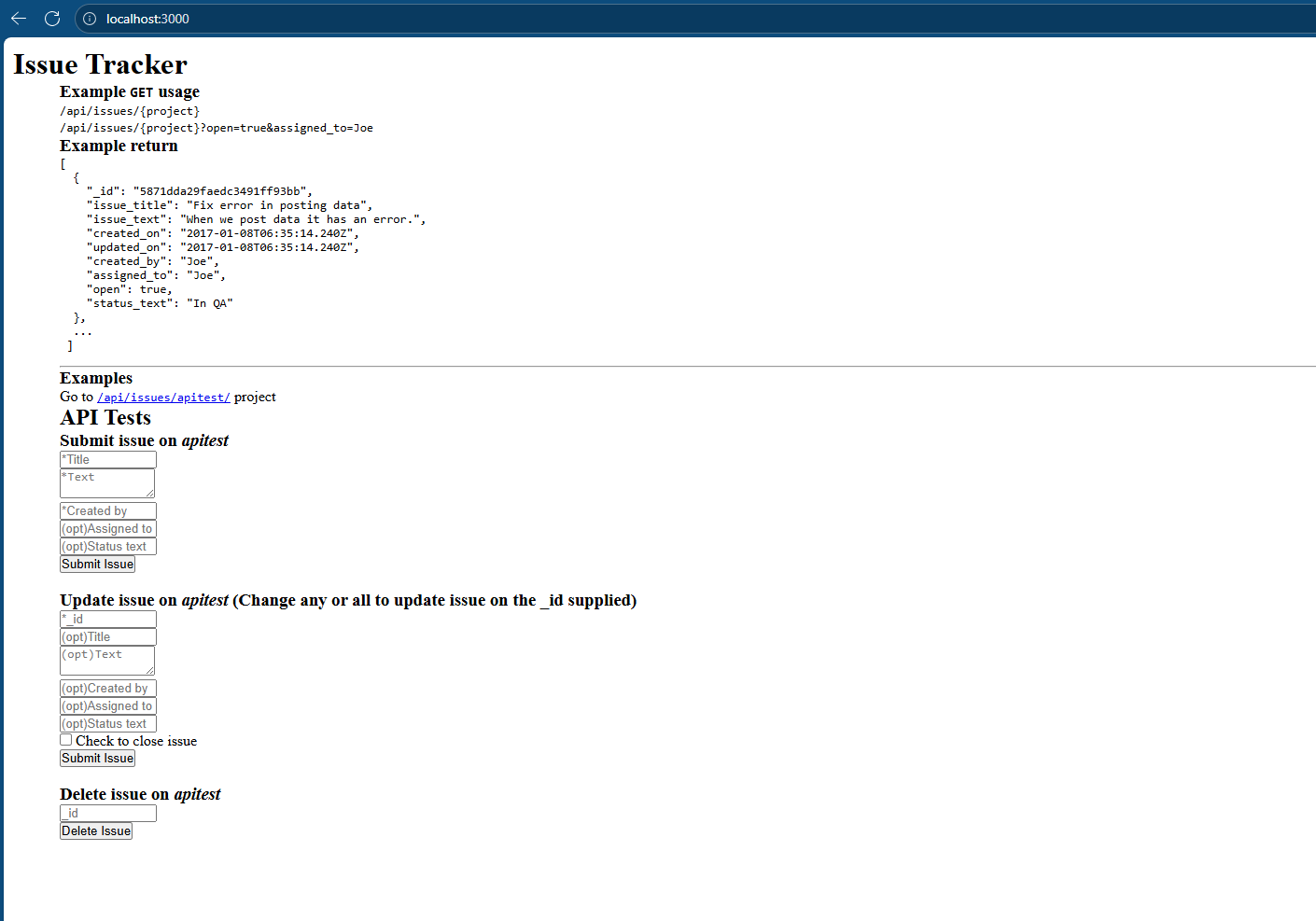
Việc vượt qua toàn bộ 14 bài test này là thước đo cuối cùng cho sự thành công của dự án, đảm bảo API hoạt động đúng như mong đợi trong mọi trường hợp. Có thể truy cập ở github repo ở phần mở đầu.



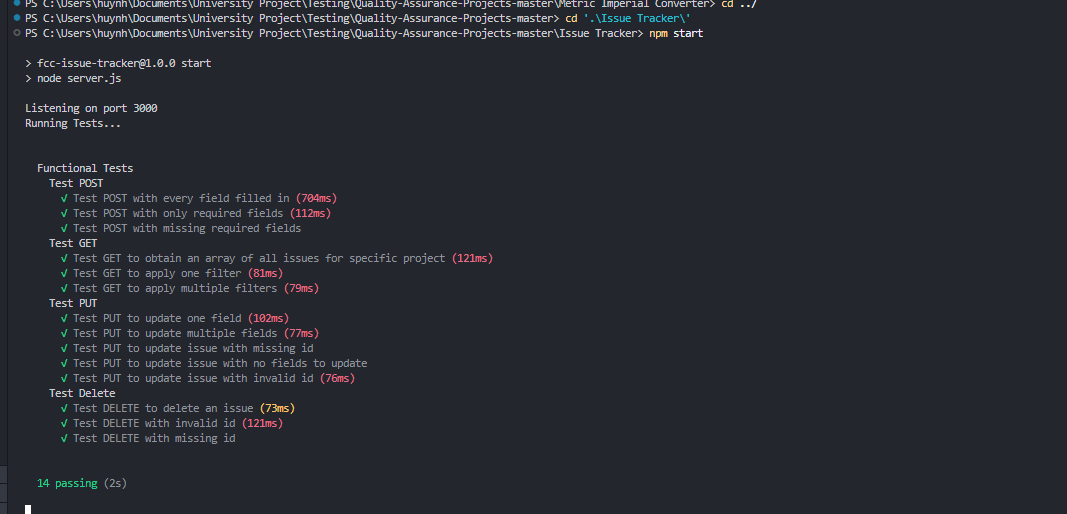
## Kết quả

Ứng dụng Issue Tracker đã được hoàn thiện, đáp ứng tất cả các yêu cầu về chức năng và logic. Giao diện người dùng cho phép tương tác trực quan với các API đã xây dựng, và quan trọng nhất, toàn bộ 14 bài kiểm thử chức năng đều đã được hoàn thành và vượt qua.

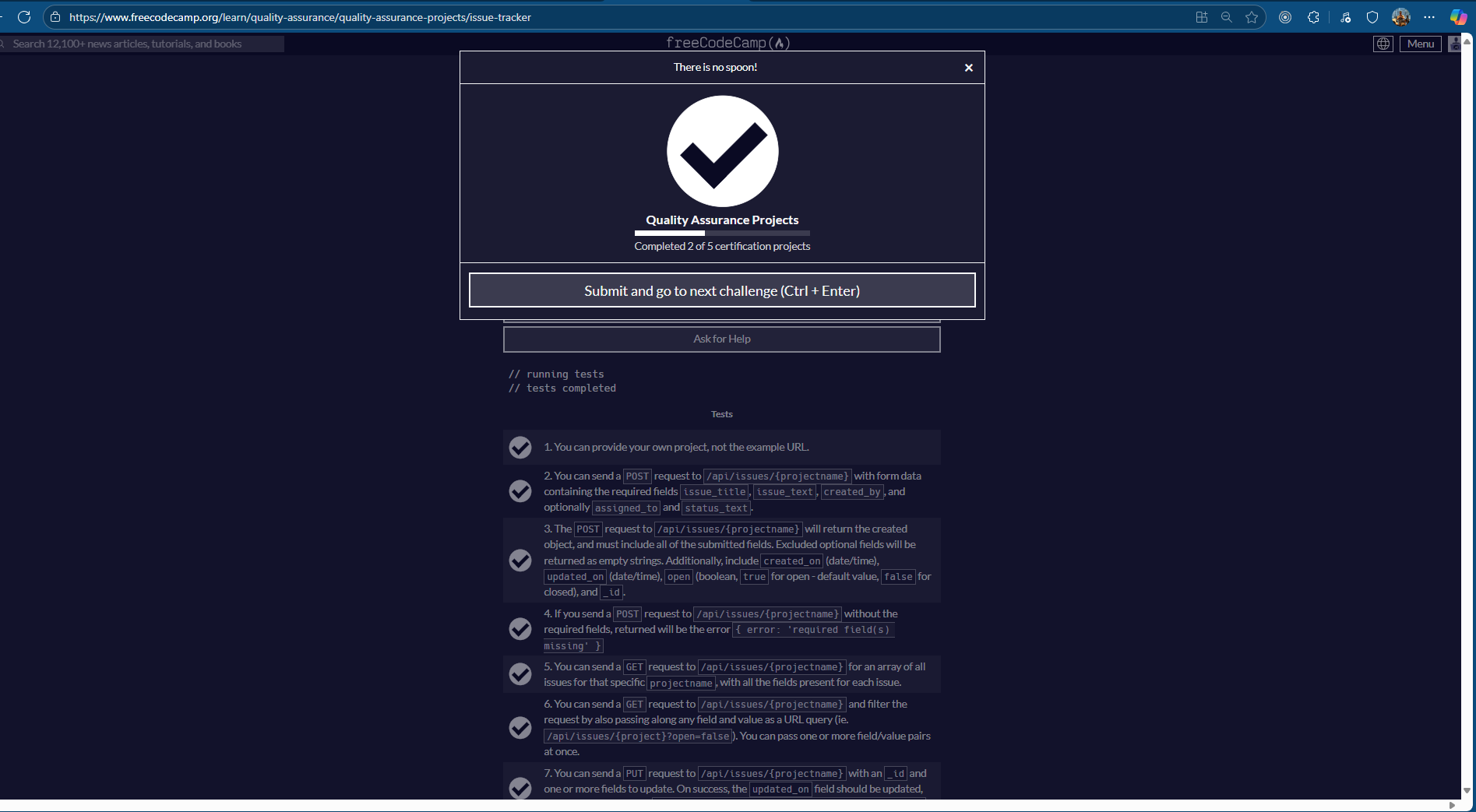
* **Giao diện người dùng:**



* **Kết quả kiểm thử:**



Dự án Issue Tracker là một bài tập thực tiễn sâu sắc về phát triển ứng dụng web full-stack, đặc biệt là trong việc thiết kế và triển khai một API RESTful mạnh mẽ và an toàn. Thông qua dự án này, các kỹ năng về làm việc với Node.js, Express, MongoDB, Mongoose và đặc biệt là phương pháp phát triển hướng kiểm thử đã được áp dụng và củng cố một cách hiệu quả.



# **Personal Library**

## Mở đầu

Dự án "Personal Library" là một ứng dụng full-stack cho phép người dùng quản lý một bộ sưu tập sách cá nhân. Chức năng cốt lõi của ứng dụng xoay quanh việc xây dựng một bộ API RESTful, cho phép người dùng thêm sách mới, xem toàn bộ thư viện, tra cứu một cuốn sách cụ thể, thêm bình luận, và thực hiện các thao tác xóa.

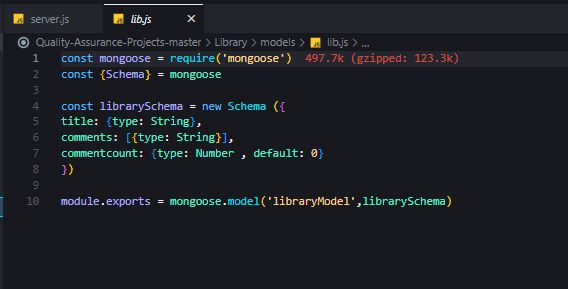
Dự án này là một bài thực hành toàn diện về các hoạt động CRUD (Create, Read, Update, Delete) với cơ sở dữ liệu MongoDB. Toàn bộ logic back-end được xây dựng trong routes/api.js, và chất lượng của ứng dụng được đảm bảo bằng việc viết và vượt qua 10 bài kiểm thử chức năng (Functional Tests).

Github Repo: [Quality-Assurance-Projects/Library at master · thang-huynhduc/Quality-Assurance-Projects](https://github.com/thang-huynhduc/Quality-Assurance-Projects/tree/master/Library)

## Kiến trúc và Thiết lập

Ứng dụng được xây dựng trên nền tảng Node.js và Express.js, với Mongoose làm cầu nối để tương tác với database MongoDB Atlas.

* **Thiết lập Môi trường (.env):**
  + DB: Chứa chuỗi kết nối (connection string) đến MongoDB Atlas, đảm bảo thông tin nhạy cảm được bảo mật.
  + NODE\_ENV=test: Cấu hình này được kích hoạt để môi trường kiểm thử có thể chạy và sử dụng một database riêng, tránh ảnh hưởng đến dữ liệu thật.
* **Mô hình Dữ liệu (Book Model):**
  + Sử dụng Mongoose, một bookSchema được định nghĩa để cấu trúc hóa dữ liệu sách trong database.
  + **title**: Kiểu String, là trường bắt buộc (required). Nếu thiếu trường này khi tạo sách, API sẽ trả về lỗi.
  + **comments**: Kiểu Array, chứa các chuỗi String. Mảng này sẽ lưu tất cả các bình luận liên quan đến cuốn sách.
  + **\_id**: Được MongoDB tự động sinh ra, là khóa chính duy nhất cho mỗi cuốn sách.
  + **commentcount**: Đây là một trường ảo (virtual field), không được lưu trực tiếp trong database. Thay vào đó, nó sẽ được tính toán động (comments.length) khi API trả về danh sách tất cả các cuốn sách.

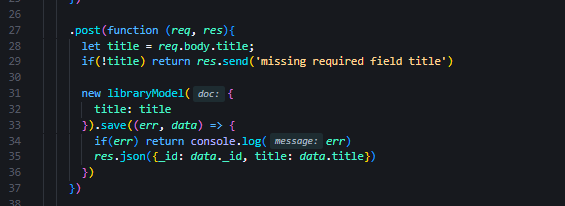


## Phân tích Chi tiết các API Endpoint

Toàn bộ logic nghiệp vụ của ứng dụng được triển khai thông qua các endpoint API trong routes/api.js.

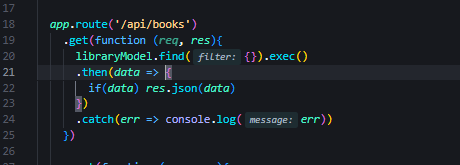
**1. POST /api/books (Thêm sách mới)**

* **Chức năng**: Nhận title từ form data và tạo một cuốn sách mới trong database.
* **Logic**:
  + **Validation**: Kiểm tra req.body.title. Nếu không tồn tại, trả về chuỗi missing required field title.
  + Nếu hợp lệ, tạo một document Book mới.
  + Lưu document vào database.
* **Kết quả**: Trả về một đối tượng JSON chứa title và \_id của cuốn sách vừa được tạo.



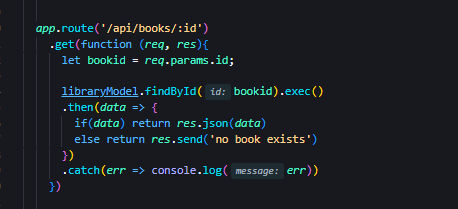
**2. GET /api/books (Lấy toàn bộ sách)**

* **Chức năng**: Truy vấn và trả về danh sách tất cả các cuốn sách trong thư viện.
* **Logic**:
  + Sử dụng Book.find({}) để lấy tất cả document từ collection.
  + Lặp qua mảng kết quả, tính toán và thêm thuộc tính commentcount cho mỗi object sách.
* **Kết quả**: Trả về một mảng JSON. Mỗi object trong mảng đại diện cho một cuốn sách và chứa các thuộc tính \_id, title, và commentcount.



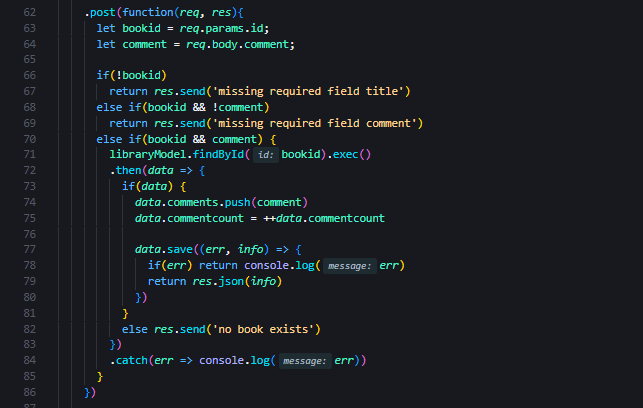
**3. GET /api/books/{\_id} (Lấy một sách)**

* **Chức năng**: Lấy thông tin chi tiết của một cuốn sách dựa trên \_id.
* **Logic**:
  + Sử dụng Book.findById() với \_id từ req.params.
  + **Validation**: Nếu không tìm thấy cuốn sách nào với \_id đã cho, trả về chuỗi no book exists.
* **Kết quả**: Trả về một đối tượng JSON duy nhất chứa \_id, title, và mảng comments của cuốn sách.



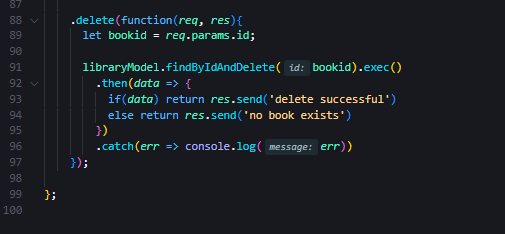
**4. POST /api/books/{\_id} (Thêm bình luận)**

* **Chức năng**: Thêm một bình luận mới cho một cuốn sách đã tồn tại.
* **Logic**:
  + **Validation**:
    - Kiểm tra req.body.comment. Nếu thiếu, trả về missing required field comment.
    - Sử dụng Book.findByIdAndUpdate() để tìm sách theo \_id. Nếu không tìm thấy, trả về no book exists.
  + Sử dụng toán tử $push của MongoDB để thêm bình luận mới vào mảng comments của sách.
  + Sử dụng tùy chọn { new: true } để đảm bảo phương thức trả về document đã được cập nhật.
* **Kết quả**: Trả về đối tượng JSON đầy đủ của cuốn sách sau khi đã thêm bình luận.



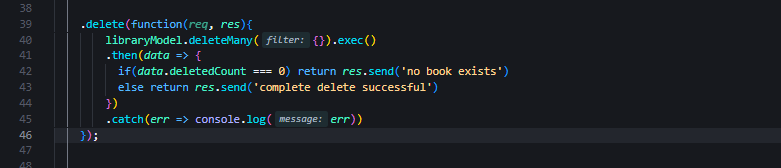
**5. DELETE /api/books/{\_id} (Xóa một sách)**

* **Chức năng**: Xóa một cuốn sách khỏi database dựa trên \_id.
* **Logic**:
  + Sử dụng Book.findByIdAndDelete().
  + **Validation**: Nếu không tìm thấy sách để xóa, trả về no book exists.
* **Kết quả**: Nếu xóa thành công, trả về chuỗi delete successful.



**6. DELETE /api/books (Xóa toàn bộ sách)**

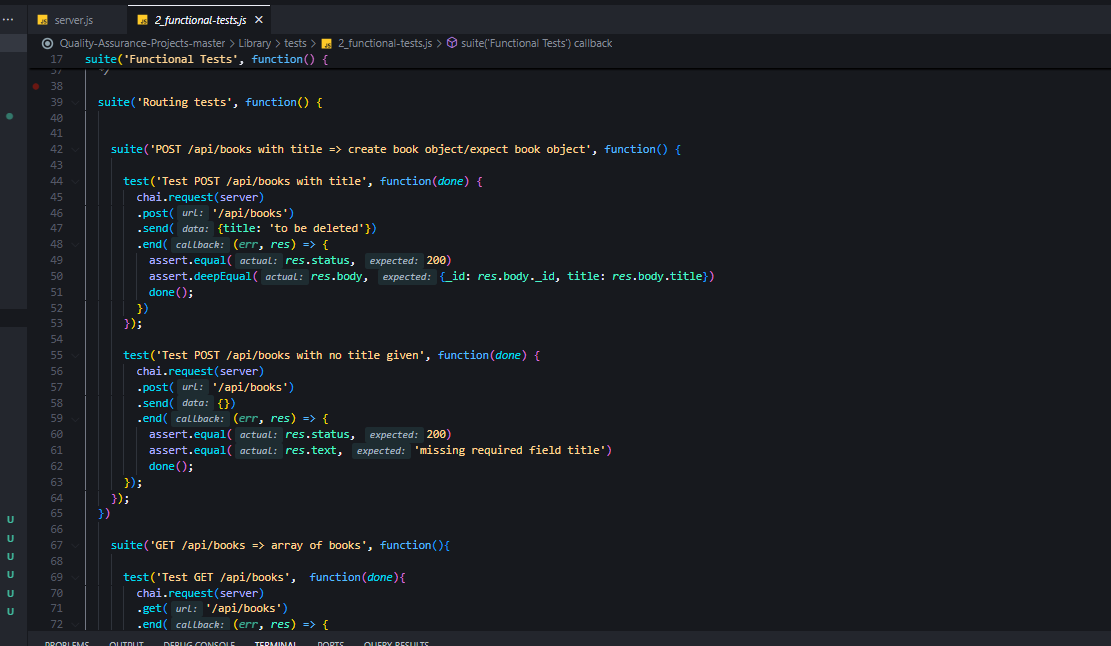
* **Chức năng**: Xóa tất cả các cuốn sách trong database. Đây là một chức năng mạnh, thường dùng cho mục đích dọn dẹp hoặc reset.
* **Logic**: Sử dụng Book.deleteMany({}).
* **Kết quả**: Nếu thành công, trả về chuỗi complete delete successful.



## Chiến lược Kiểm thử

Dự án yêu cầu 10 bài kiểm thử chức năng trong tests/2\_functional-tests.js. Các bài test này sử dụng Chai-HTTP để mô phỏng các lời gọi API thực tế đến tất cả các endpoint, kiểm tra cả các kịch bản thành công và thất bại:

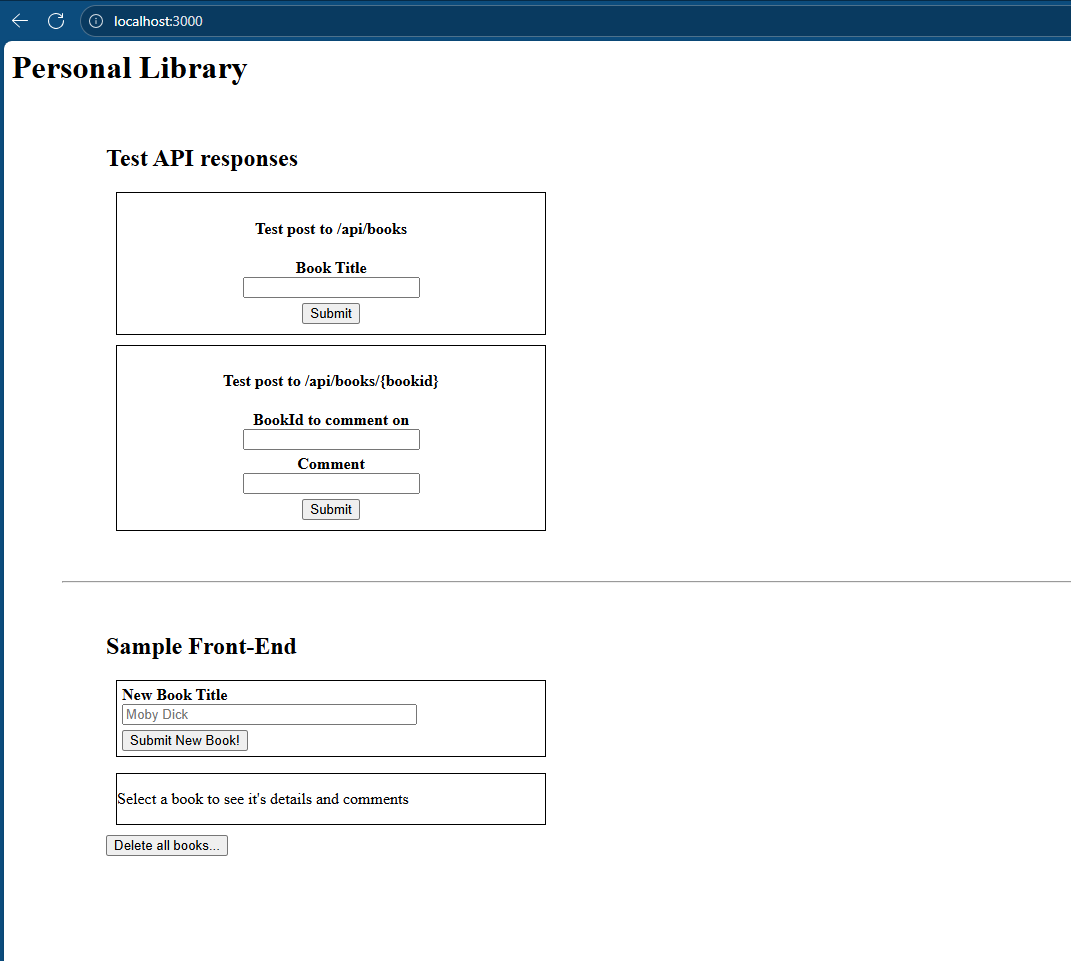
* Tạo sách thành công và thất bại.
* Lấy danh sách sách.
* Lấy một sách thành công và thất bại (\_id không tồn tại).
* Thêm bình luận thành công và thất bại.
* Xóa một sách thành công và thất bại.
* Xóa toàn bộ sách.



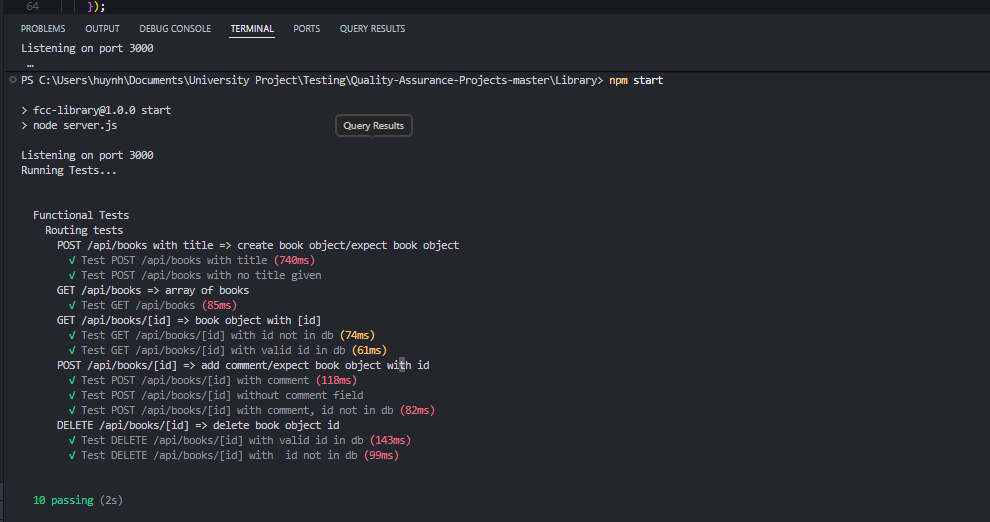
## Kết quả

Ứng dụng đã được phát triển hoàn chỉnh, đáp ứng tất cả các yêu cầu về chức năng. Giao diện người dùng đơn giản nhưng hiệu quả, cho phép tương tác với tất cả các tính năng của API. Toàn bộ 10 bài kiểm thử chức năng đều đã vượt qua, khẳng định độ tin cậy và tính đúng đắn của logic back-end.

* **Giao diện người dùng:** Giao diện chính của Personal Library, hiển thị chức năng thêm sách, danh sách sách và chi tiết sách

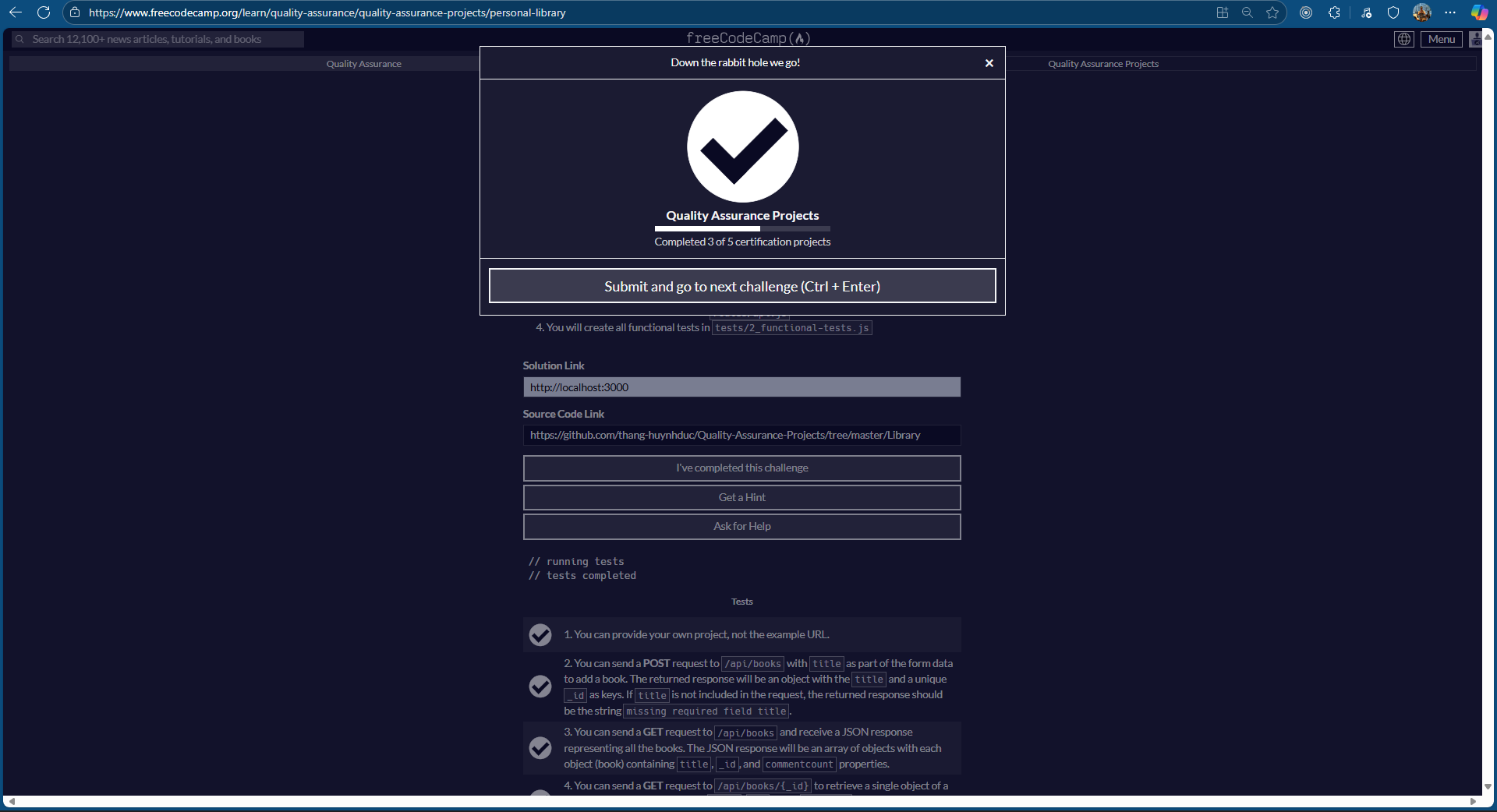


* **Kết quả kiểm thử:** Console log hiển thị kết quả "10 passing", xác nhận tất cả các bài kiểm thử chức năng đều thành công



## Kết luận

Dự án Personal Library là một bài tập thực hành xuất sắc về việc xây dựng một ứng dụng web full-stack với các thao tác CRUD cơ bản. Nó không chỉ củng cố kỹ năng làm việc với Express và Mongoose mà còn nhấn mạnh tầm quan trọng của việc xử lý các trường hợp biên và xác thực đầu vào một cách cẩn thận, được đảm bảo bởi một bộ kiểm thử chức năng toàn diện.



# **Sudoku Solver**

## Mở đầu

Dự án "Sudoku Solver" là một bài toán đầy thử thách, đòi hỏi sự kết hợp giữa kỹ năng xây dựng API back-end và tư duy thuật toán sắc bén. Mục tiêu của dự án là tạo ra một microservice có khả năng:

1. **Kiểm tra (Check):** Xác thực xem một giá trị số có thể được đặt hợp lệ tại một tọa độ cụ thể trên bàn cờ Sudoku hay không.
2. **Giải (Solve):** Tìm ra lời giải hoàn chỉnh cho một bài toán Sudoku bất kỳ được cung cấp dưới dạng chuỗi.

Toàn bộ logic xử lý và thuật toán được gói gọn trong controllers/sudoku-solver.js, trong khi routes/api.js chịu trách nhiệm tiếp nhận và điều hướng các yêu cầu HTTP. Chất lượng và tính đúng đắn của ứng dụng được đảm bảo bởi một bộ kiểm thử cực kỳ toàn diện, bao gồm 12 Unit Tests và 14 Functional Tests.

Github repo: [Quality-Assurance-Projects/Sodoku Solver at master · thang-huynhduc/Quality-Assurance-Projects](https://github.com/thang-huynhduc/Quality-Assurance-Projects/tree/master/Sodoku%20Solver)

## Kiến trúc và Phân tích Logic

Cấu trúc của dự án được phân tách rõ ràng thành hai thành phần chính: Controller (xử lý logic) và Router (xử lý yêu cầu).

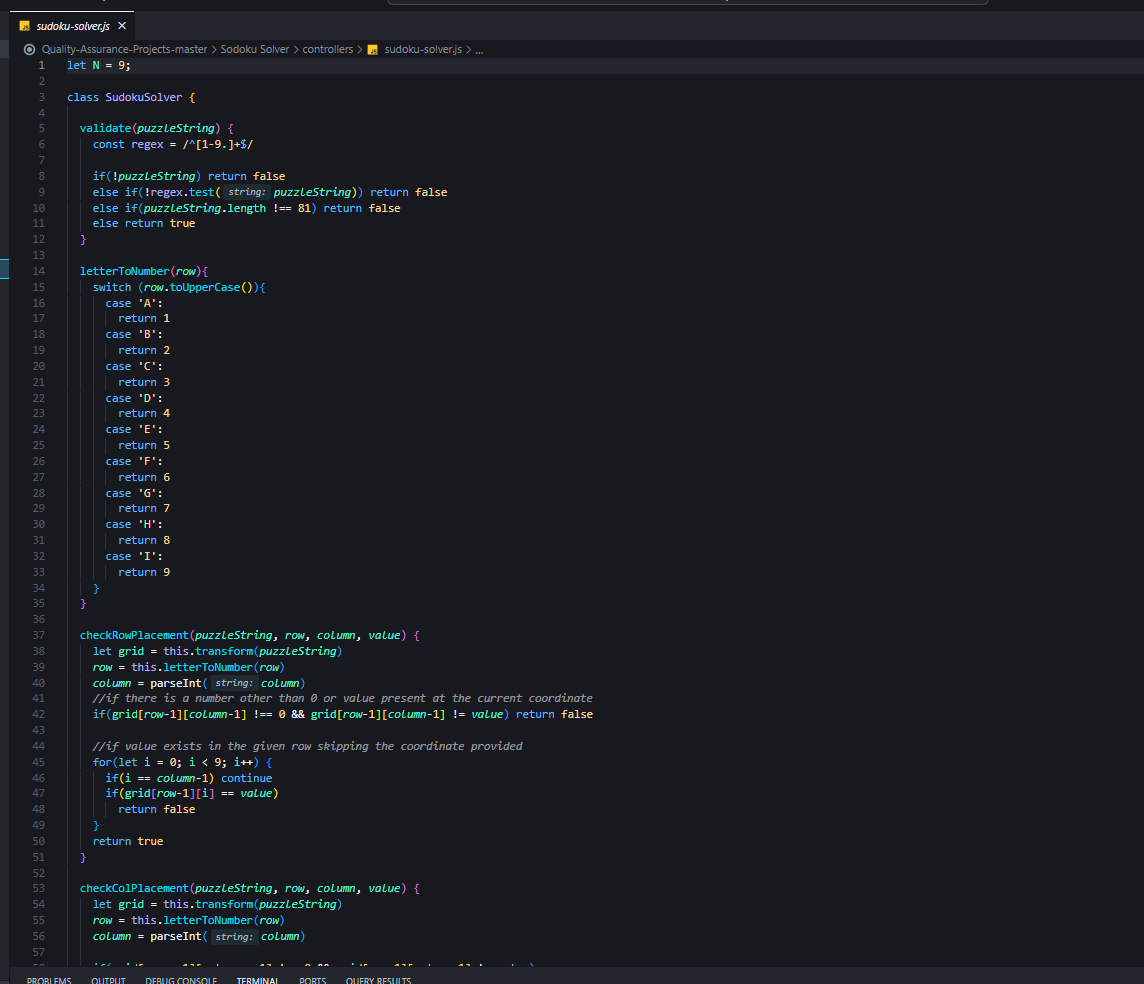
* **/controllers/sudoku-solver.js (Bộ não của ứng dụng):** Đây là nơi chứa toàn bộ trí tuệ của ứng dụng. Một class SudokuSolver được tạo ra để đóng gói tất cả các phương thức liên quan đến việc xác thực, kiểm tra và giải đố. Cách tiếp cận này giúp mã nguồn trở nên sạch sẽ, có tổ chức và dễ dàng cho việc kiểm thử đơn vị (Unit Test).
* **/routes/api.js (Cổng giao tiếp):** File này định nghĩa hai endpoint API chính: /api/check và /api/solve. Nó chịu trách nhiệm nhận dữ liệu từ req.body, gọi các phương thức tương ứng từ class SudokuSolver, và định dạng câu trả lời JSON để gửi về cho client.

## Logic Solver (/controllers/sudoku-solver.js)

Đây là phần phức tạp và cốt lõi nhất của dự án, bao gồm các phương thức chính sau:

**1. validate(puzzleString) (Xác thực chuỗi đầu vào)**

* **Chức năng**: Là hàng rào phòng thủ đầu tiên, đảm bảo chuỗi puzzle đầu vào tuân thủ đúng định dạng.
* **Logic**:
  + Kiểm tra xem chuỗi có tồn tại không. Nếu thiếu, trả về lỗi Required field missing.
  + Sử dụng biểu thức chính quy (Regex) /[^1-9.]/ để kiểm tra toàn bộ chuỗi. Nếu có bất kỳ ký tự nào khác ngoài số từ 1-9 và dấu chấm (.), trả về lỗi Invalid characters in puzzle.
  + Kiểm tra độ dài của chuỗi. Nếu không chính xác bằng 81 ký tự, trả về lỗi Expected puzzle to be 81 characters long.
  + Nếu vượt qua cả 3 bài kiểm tra, chuỗi được xem là hợp lệ.



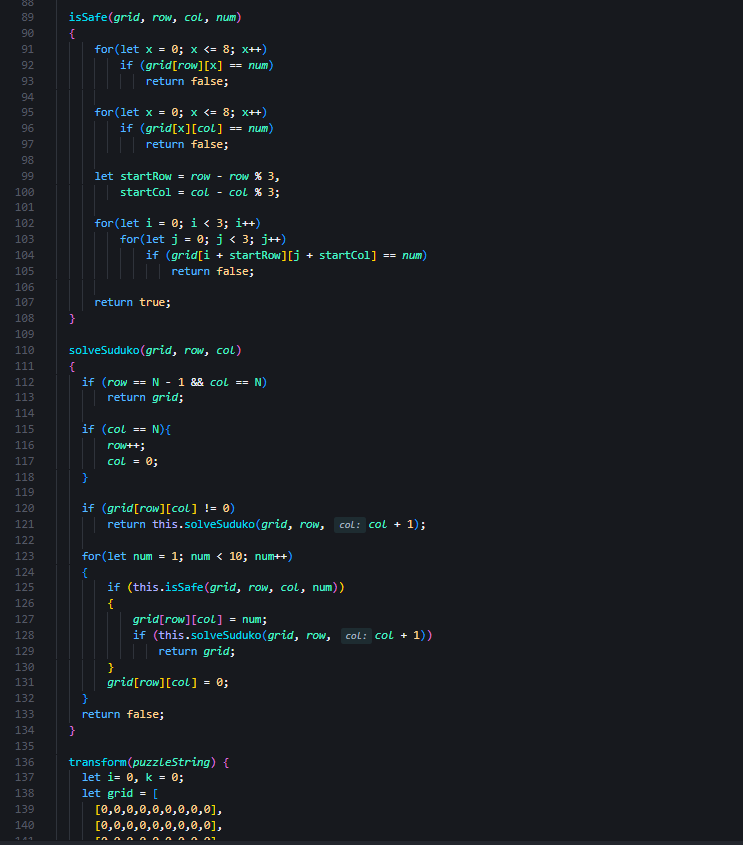
**2. Các hàm kiểm tra vị trí (checkRowPlacement, checkColPlacement, checkRegionPlacement)**

* **Chức năng**: Đây là 3 hàm nền tảng, xác định xem một giá trị (value) có thể được đặt tại một tọa độ (row, column) mà không vi phạm luật Sudoku hay không.
* **Logic**:
  + checkRowPlacement: Lặp qua tất cả các ô trong hàng đã cho và kiểm tra xem value đã tồn tại hay chưa.
  + checkColPlacement: Lặp qua tất cả các ô trong cột đã cho và kiểm tra xem value đã tồn tại hay chưa.
  + checkRegionPlacement: Logic phức tạp hơn. Đầu tiên, nó tính toán tọa độ góc trên bên trái của vùng 3x3 chứa ô đang xét. Sau đó, nó lặp qua 9 ô trong vùng đó để kiểm tra sự tồn tại của value.

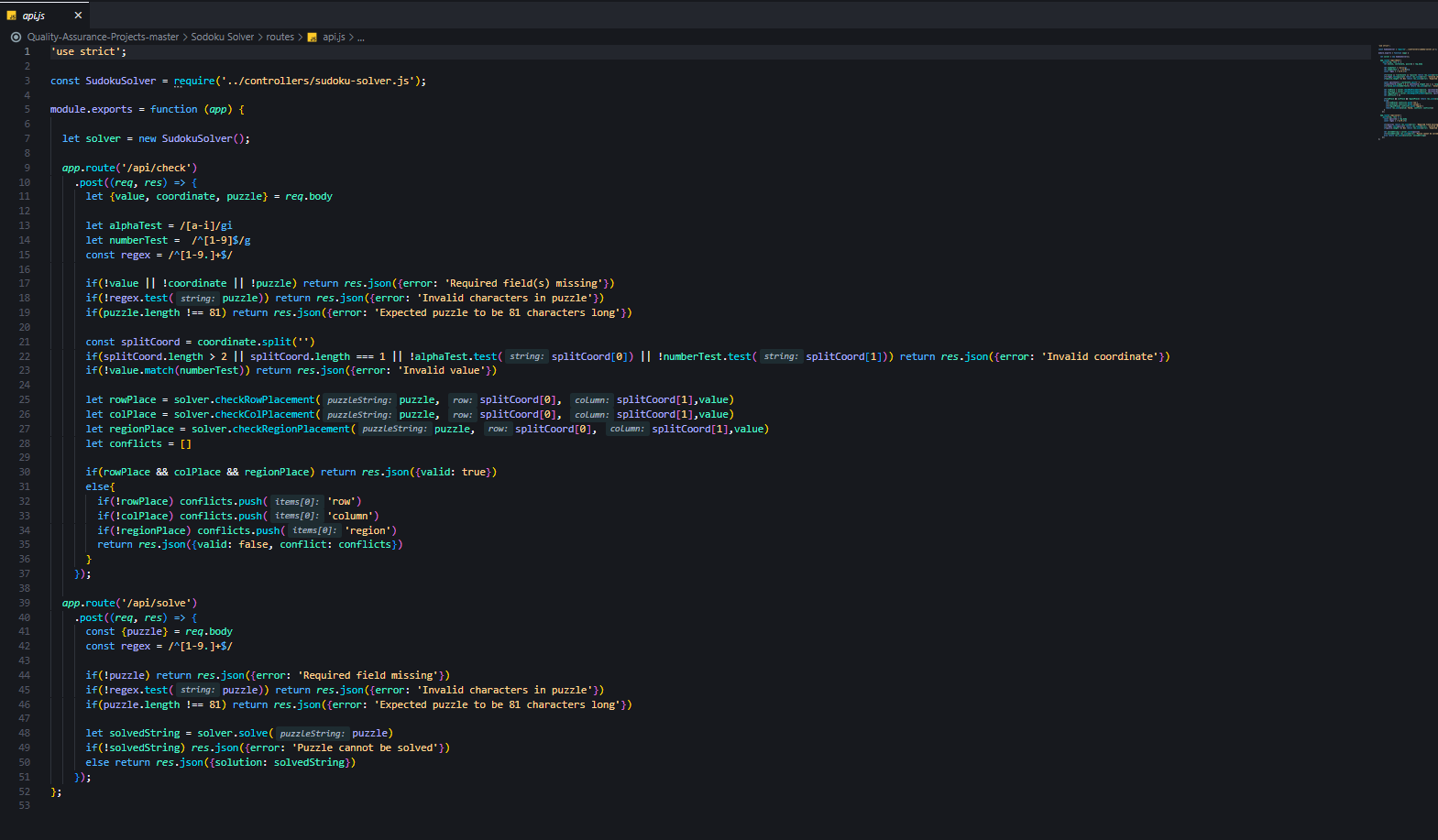


**3. solve(puzzleString) (Thuật toán giải đố Backtracking)**

* **Chức năng**: Tìm lời giải cho một chuỗi puzzle hợp lệ.
* **Logic**: Đây là phần ứng dụng thuật toán **đệ quy quay lui (Recursive Backtracking)**, một phương pháp thử và sai có hệ thống.
  1. **Bước 1 - Khởi tạo**: Chuyển chuỗi puzzleString 81 ký tự thành một ma trận 9x9 để dễ dàng thao tác.
  2. **Bước 2 - Hàm đệ quy solveRecursive()**:
     + **Điều kiện dừng (Base Case)**: Tìm ô trống đầu tiên (.). Nếu không còn ô trống nào, có nghĩa là bàn cờ đã được giải xong, trả về bàn cờ đã giải.
     + **Bước đệ quy**: Tại ô trống tìm được, lặp thử các số từ 1 đến 9.
       - Với mỗi số, dùng các hàm check...Placement để kiểm tra xem việc đặt số đó vào ô trống có hợp lệ không.
       - **Nếu hợp lệ**: Đặt số đó vào ô trống (đây là một "lựa chọn thử"). Sau đó, gọi lại chính hàm solveRecursive() để tiếp tục giải phần còn lại của bàn cờ.
       - **Kiểm tra kết quả**: Nếu lời gọi đệ quy trả về một kết quả (tức là tìm được lời giải), hãy trả về kết quả đó ngay lập tức.
       - **Quay lui (Backtrack)**: Nếu lời gọi đệ quy trả về false (tức là "lựa chọn thử" ở trên dẫn vào ngõ cụt), hãy **hủy bỏ lựa chọn đó** (đặt lại ô đó thành .) và tiếp tục vòng lặp để thử số tiếp theo.
  3. **Bước 3 - Kết quả**: Nếu vòng lặp từ 1-9 kết thúc mà không tìm được lời giải nào, hàm sẽ trả về false, báo hiệu rằng bài toán không thể giải được.



## API Endpoints (/routes/api.js)



**1. POST /api/check**

* **Chức năng**: Kiểm tra tính hợp lệ của một lượt đi.
* **Đầu vào**: puzzle, coordinate (ví dụ: 'A1', 'I9'), value (1-9).
* **Logic**:
  + Xác thực tất cả các trường đầu vào.
  + Chuyển đổi coordinate (ví dụ: 'A1') thành chỉ số mảng (row: 0, col: 0).
  + Gọi đồng thời cả 3 hàm check...Placement.
  + Tổng hợp kết quả:
    - Nếu cả 3 đều hợp lệ, trả về { valid: true }.
    - Nếu không, trả về { valid: false, conflict: [...] }, trong đó mảng conflict chứa tên các vùng bị xung đột ('row', 'column', 'region').

**2. POST /api/solve**

* **Chức năng**: Giải một bài toán Sudoku.
* **Đầu vào**: puzzle.
* **Logic**:
  + Gọi hàm validate() để kiểm tra puzzle. Nếu không hợp lệ, trả về lỗi tương ứng.
  + Gọi hàm solve() từ SudokuSolver.
  + Nếu solve() trả về một chuỗi lời giải, trả về { solution: '...' }.
  + Nếu solve() trả về false, trả về lỗi { error: 'Puzzle cannot be solved' }.

## Chiến lược Kiểm thử

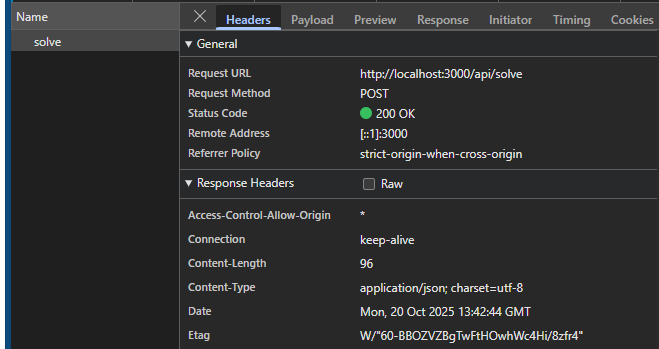
Dự án này áp dụng chiến lược kiểm thử hai tầng rất nghiêm ngặt:

* **Unit Tests (12 bài test)**: Tập trung kiểm tra từng phương thức riêng lẻ trong class SudokuSolver. Ví dụ: validate() có trả về đúng lỗi với chuỗi sai định dạng không? checkRowPlacement() có phát hiện xung đột chính xác không?
* **Functional Tests (14 bài test)**: Kiểm tra toàn bộ luồng xử lý của hai endpoint API. Các bài test này mô phỏng client gửi các request POST với đủ loại dữ liệu (hợp lệ, thiếu trường, sai định dạng, không thể giải...) và xác thực rằng response JSON trả về chính xác 100% theo yêu cầu.

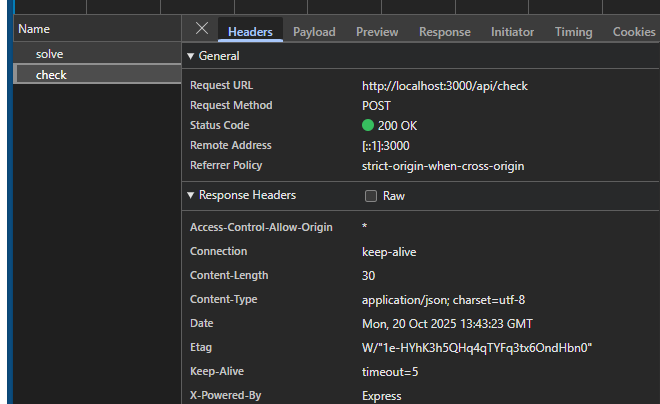
## Kết quả

Ứng dụng đã được phát triển thành công, có khả năng giải quyết các bài toán Sudoku hợp lệ và xử lý lỗi một cách chính xác. Toàn bộ 26 bài kiểm thử (12 unit + 14 functional) đều đã vượt qua, chứng minh độ tin cậy và sự vững chắc của logic đã xây dựng.

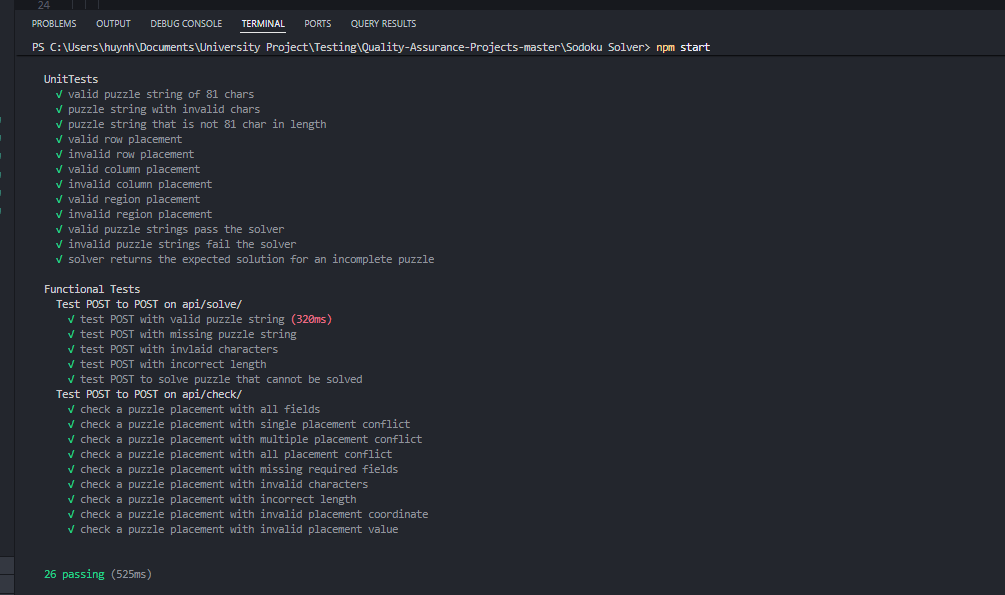
* **Kết quả API /solve:** Một request POST thành công đến /api/solve và chuỗi JSON chứa lời giải trả về



* **Kết quả API /check:** Một request POST đến /api/check phát hiện xung đột và trả về { "valid": false, "conflict": [ "row", "region" ] }

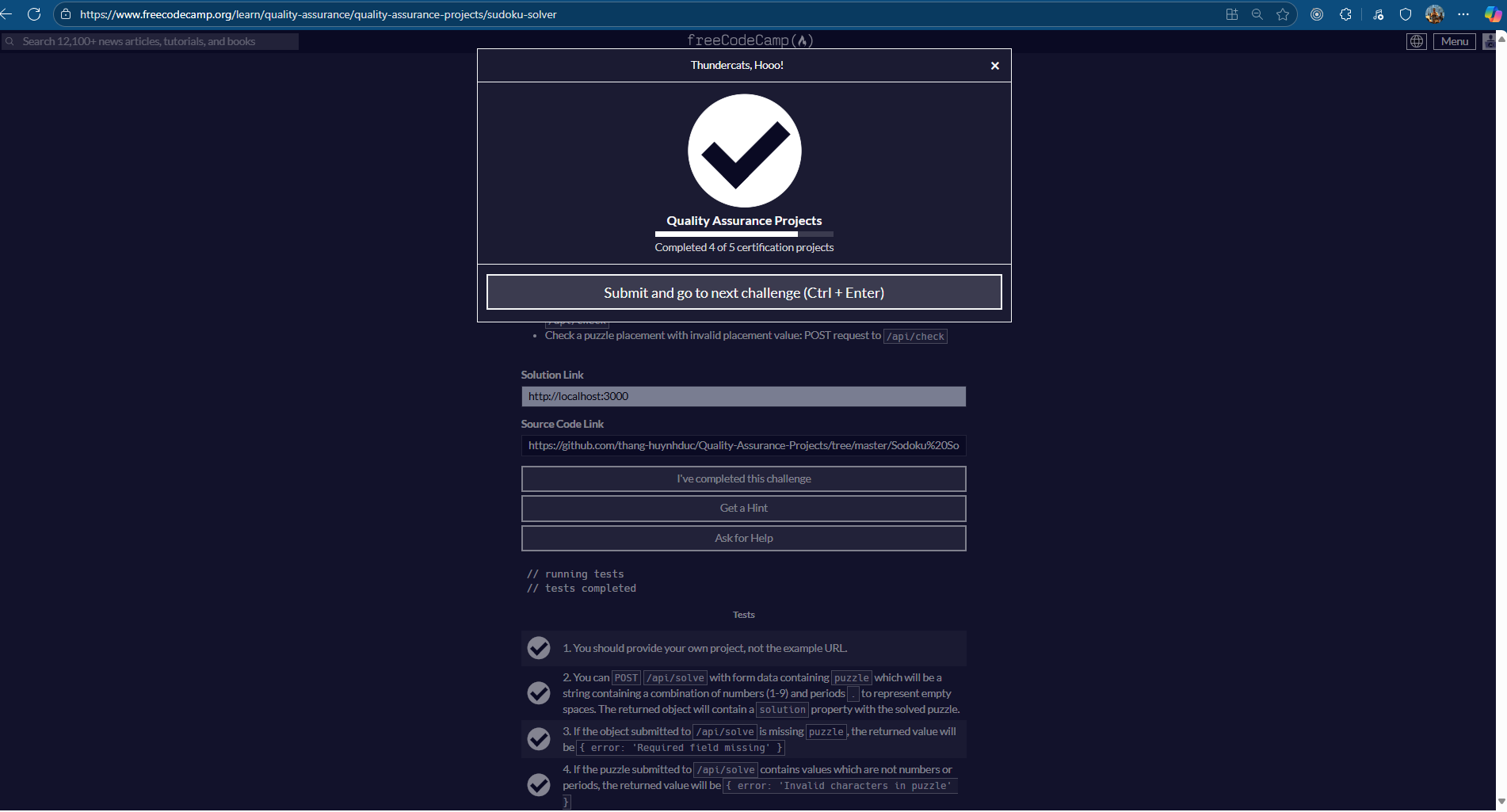


* **Kết quả kiểm thử:** log hiển thị kết quả "26 passing", xác nhận tất cả Unit Tests và Functional Tests đều thành công



## Kết luận

Dự án Sudoku Solver là một bài tập nâng cao, không chỉ kiểm tra kỹ năng xây dựng API mà còn đi sâu vào việc triển khai một thuật toán đệ quy phức tạp. Việc phải đáp ứng một bộ kiểm thử lớn và chi tiết đã thúc đẩy việc viết code một cách cẩn thận, có cấu trúc và dễ bảo trì. Dự án này là một minh chứng xuất sắc cho sức mạnh của phương pháp phát triển hướng kiểm thử (TDD) trong việc xây dựng các ứng dụng phức tạp và đáng tin cậy.



# **American British Translator**

## Mở đầu

Dự án cuối cùng, "American British Translator", là một bài toán tinh tế về xử lý ngôn ngữ tự nhiên, tập trung vào việc xây dựng một microservice có khả năng dịch thuật qua lại giữa hai phương ngữ phổ biến của tiếng Anh: Anh-Mỹ (American English) và Anh-Anh (British English).

Ứng dụng này không sử dụng các API dịch thuật bên ngoài, mà đòi hỏi phải tự xây dựng một bộ logic dựa trên các quy tắc về từ vựng, chính tả, chức danh và định dạng thời gian. Toàn bộ "trí tuệ" của bộ dịch được đặt trong /components/translator.js, và chất lượng được đảm bảo bởi một bộ kiểm thử cực kỳ nghiêm ngặt, bao gồm 24 Unit Tests và 6 Functional Tests.

Github repo: [Quality-Assurance-Projects/American British Translator at master · thang-huynhduc/Quality-Assurance-Projects](https://github.com/thang-huynhduc/Quality-Assurance-Projects/tree/master/American%20British%20Translator)

## Kiến trúc và Phân tích Logic (/components/translator.js)

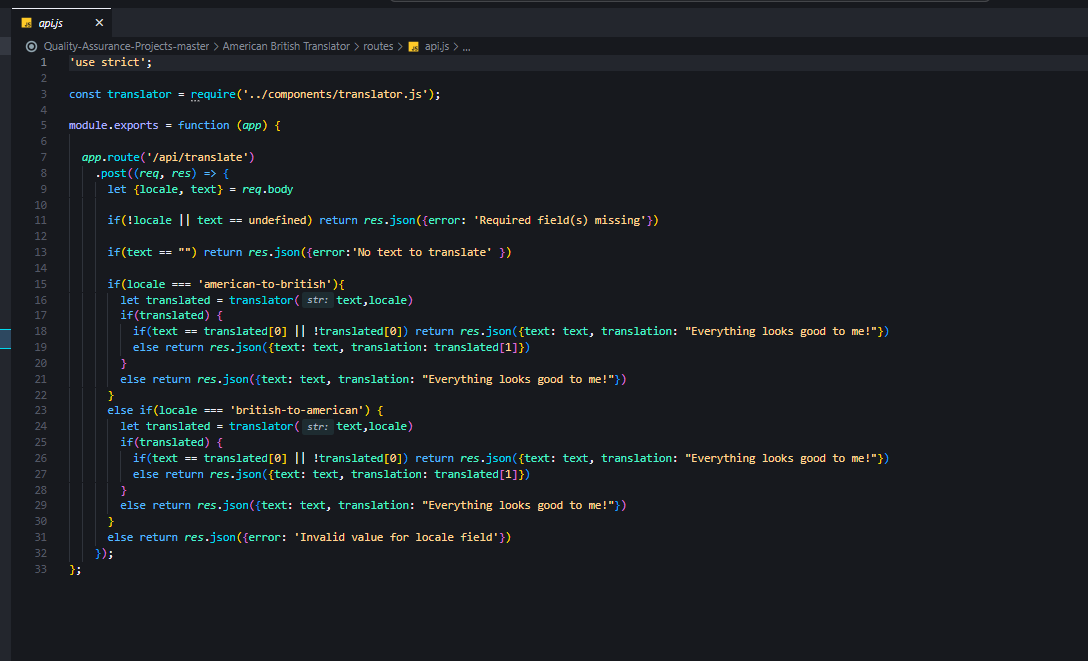
Trái tim của ứng dụng nằm hoàn toàn trong class Translator, nơi chứa toàn bộ logic dịch thuật.

* **Nguồn dữ liệu (Từ điển):** Logic dịch thuật dựa trên bốn bộ từ điển được cung cấp dưới dạng các file JavaScript object:
  1. american-to-british-spelling.js: Chứa các cặp từ khác biệt về chính tả (ví dụ: favorite: favourite).
  2. american-to-british-titles.js: Chứa các chức danh và cách viết tắt khác nhau (ví dụ: Mr.: Mr).
  3. american-only.js & british-only.js: Chứa các từ vựng và thành ngữ đặc trưng chỉ có ở một trong hai phương ngữ (ví dụ: parking lot (Mỹ) vs car park (Anh)).
* **Quy trình Dịch thuật - Một chuỗi xử lý có thứ tự:** Để đảm bảo tính chính xác và tránh các lỗi thay thế sai (ví dụ: dịch "I" trong "biscuit" thành "we"), quy trình dịch thuật phải tuân theo một thứ tự ưu tiên nghiêm ngặt:
  1. **Dịch Chức danh (Titles):** Lặp qua từ điển chức danh. Do các chức danh có thể chứa dấu chấm (ví dụ: Mr.), việc xử lý chúng trước tiên sẽ tránh các xung đột sau này. Logic phải bảo toàn được việc viết hoa của chữ cái đầu.
  2. **Dịch Thành ngữ & Từ vựng đặc trưng (Multi-word terms):** Lặp qua các từ điển american-only và british-only. Việc xử lý các cụm từ dài trước các từ đơn lẻ là cực kỳ quan trọng. Ví dụ, "car boot sale" phải được dịch trước "car" để tránh sai sót.
  3. **Dịch Chính tả (Single-word spelling):** Sau khi đã xử lý các cụm từ dài, logic sẽ lặp qua từ điển chính tả để dịch các từ đơn lẻ.
  4. **Dịch Định dạng Thời gian (Time format):** Cuối cùng, sử dụng biểu thức chính quy (Regex) để tìm và thay thế định dạng thời gian. Cụ thể, HH:MM (Mỹ) sẽ được đổi thành HH.MM (Anh) và ngược lại.
* **Kỹ thuật Đánh dấu (Highlighting) và Bảo toàn Viết hoa (Case Preservation):**
  1. Mỗi khi một từ hoặc cụm từ được thay thế, nó phải được bao bọc bởi thẻ <span class="highlight">...</span>. Để quản lý việc này, logic duy trì hai phiên bản của chuỗi dịch: một phiên bản thuần túy cho các bước xử lý tiếp theo và một phiên bản đã được highlight để trả về cho người dùng.
  2. Logic dịch thuật đủ thông minh để nhận biết và bảo toàn cách viết hoa của từ gốc. Nếu từ đầu vào là "Favorite", kết quả sẽ là <span class="highlight">Favourite</span>, không phải "favourite".



## API Endpoint (/routes/api.js)

Ứng dụng chỉ có một endpoint duy nhất để thực hiện toàn bộ chức năng.



**POST /api/translate**

* **Chức năng**: Nhận văn bản và yêu cầu dịch, sau đó trả về kết quả.
* **Đầu vào**: Một đối tượng JSON trong req.body chứa:
  + text: Chuỗi ký tự cần dịch.
  + locale: Chuỗi xác định chiều dịch (american-to-british hoặc british-to-american).
* **Logic và Validation**:
  1. Kiểm tra sự tồn tại của cả hai trường text và locale. Nếu thiếu, trả về { error: 'Required field(s) missing' }.
  2. Kiểm tra xem trường locale có giá trị hợp lệ hay không. Nếu không, trả về { error: 'Invalid value for locale field' }.
  3. Kiểm tra xem trường text có rỗng không. Nếu có, trả về { error: 'No text to translate' }.
  4. Nếu tất cả đều hợp lệ, khởi tạo một instance của class Translator và gọi phương thức dịch.
* **Kết quả**:
  1. Nếu quá trình dịch không tạo ra sự thay đổi nào, trả về { ..., translation: "Everything looks good to me!" }.
  2. Nếu có thay đổi, trả về { text: '...', translation: '...' } với chuỗi translation đã được highlight.

## Chiến lược Kiểm thử

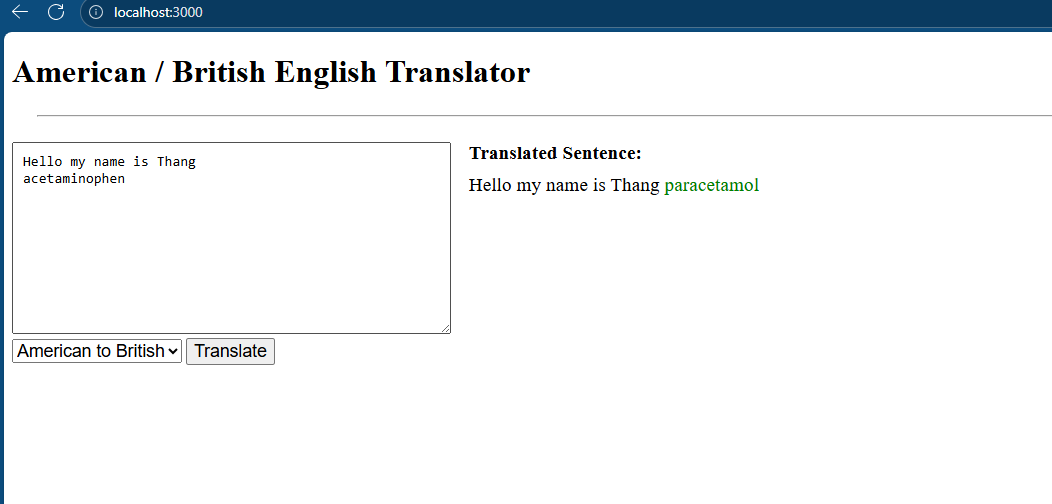
Dự án này sở hữu bộ kiểm thử lớn nhất và chi tiết nhất trong cả 5 dự án, được chia làm hai phần:

* **Unit Tests (24 bài test) in tests/1\_unit-tests.js**:
  + Tập trung kiểm tra trực tiếp các phương thức của class Translator.
  + Bao phủ một loạt các câu mẫu, kiểm tra từng quy tắc dịch thuật: 20 bài test cho các trường hợp dịch từ/câu, và 4 bài test riêng để xác minh chức năng highlight hoạt động chính xác.
* **Functional Tests (6 bài test) in tests/2\_functional-tests.js**:
  + Kiểm tra toàn bộ endpoint /api/translate.
  + Các kịch bản bao gồm: dịch thành công, locale không hợp lệ, thiếu trường text, thiếu trường locale, text rỗng, và dịch một câu không cần thay đổi.

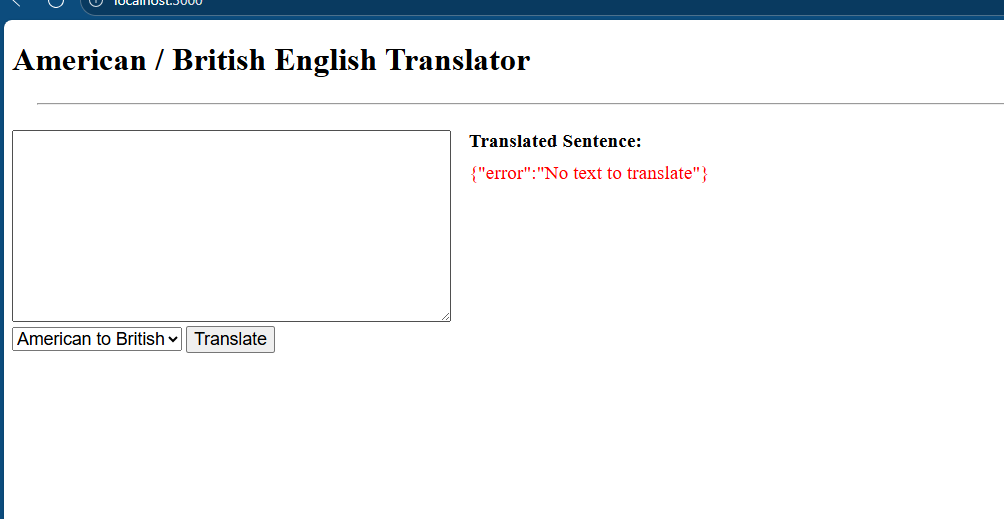
## Kết quả

Ứng dụng đã được xây dựng thành công, có khả năng dịch thuật chính xác giữa hai phương ngữ và xử lý tốt các trường hợp biên. Toàn bộ 30 bài kiểm thử (24 unit + 6 functional) đều đã vượt qua, là minh chứng cho sự vững chắc của logic đã được triển khai.

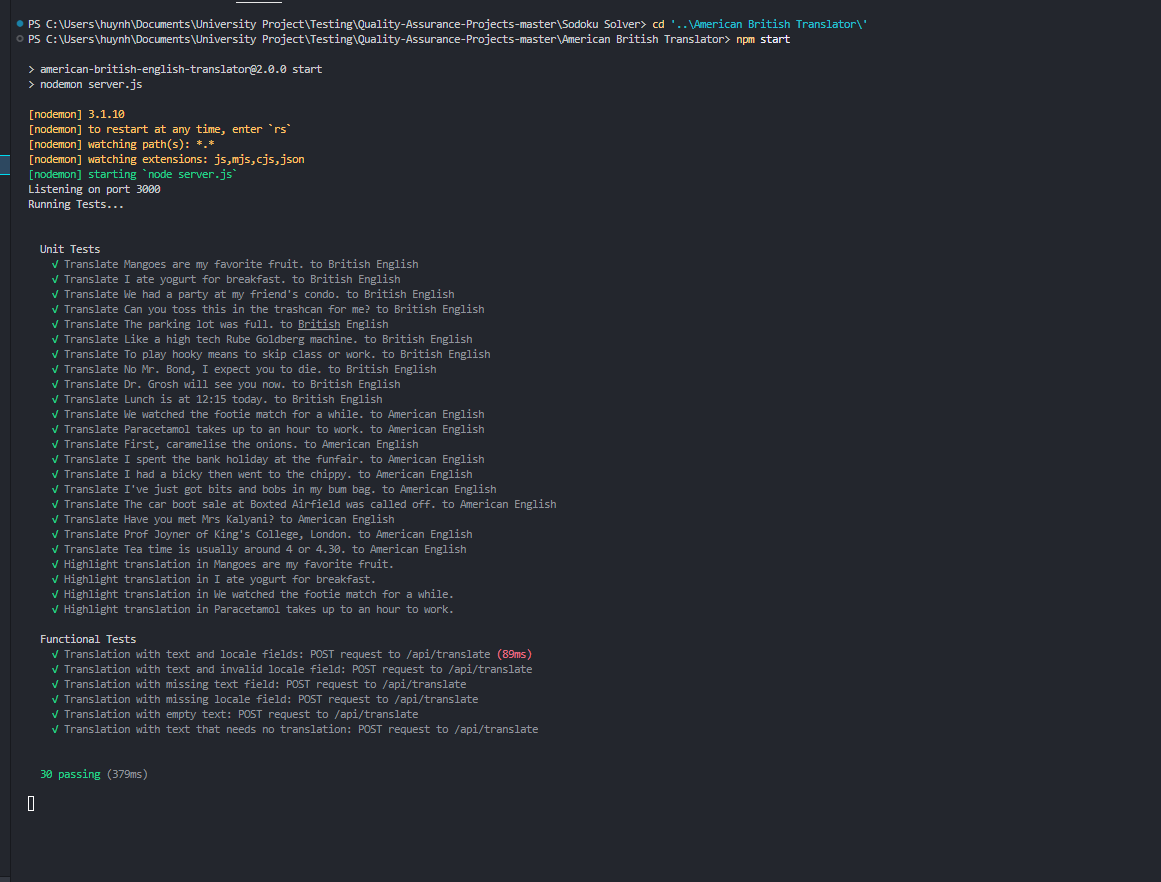
* **Kết quả Dịch thuật thành công:** Một request POST thành công đến /api/translate, cho thấy kết quả dịch có highlight



* **Xử lý Lỗi đầu vào:** Phản hồi lỗi khi một trường bắt buộc bị thiếu trong request

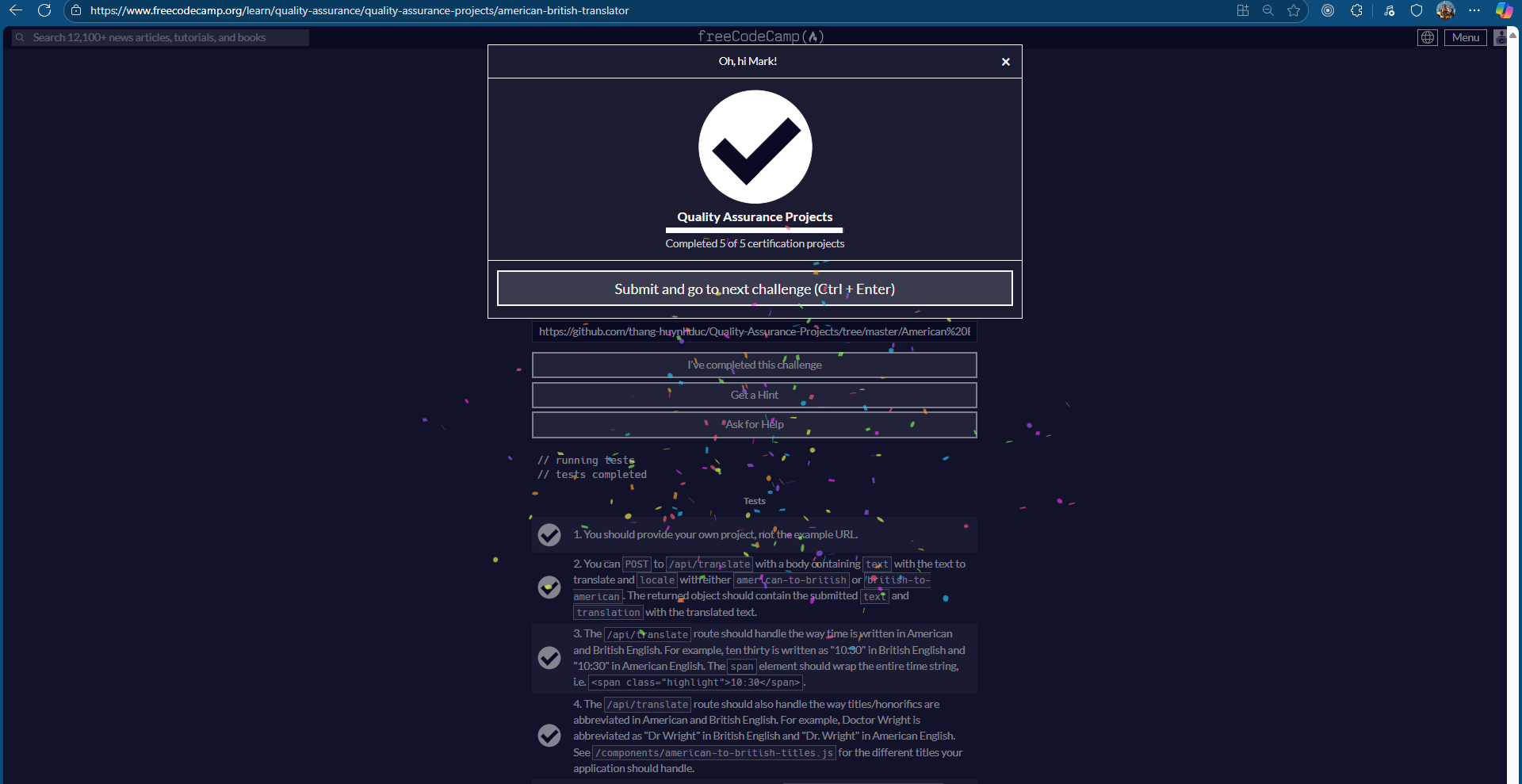


* **Kết quả Kiểm thử:** Console log hiển thị kết quả "30 passing", xác nhận tất cả Unit Tests và Functional Tests đều thành công



## Kết luận

Dự án American British Translator là một bài tập thực hành sâu sắc và đầy thách thức về xử lý chuỗi (string manipulation), biểu thức chính quy (Regex), và logic có điều kiện phức tạp. Việc phải tuân thủ một thứ tự xử lý nghiêm ngặt và xử lý các trường hợp đặc biệt (như bảo toàn viết hoa) đòi hỏi sự tỉ mỉ và một tư duy cấu trúc rõ ràng. Thành công của dự án, được xác thực bởi một bộ kiểm thử toàn diện, đã chứng minh năng lực xử lý các bài toán logic phức tạp trong môi trường back-end.



# **Hoàn thành và Chứng nhận**

## Quá trình nộp bài

Sau khi hoàn thành và triển khai mỗi dự án, URL của ứng dụng đang chạy sẽ được nộp lên nền tảng học tập. Hệ thống sẽ tự động chạy các bộ kiểm thử (test suite) để xác minh rằng tất cả các yêu cầu và kịch bản người dùng (user stories) đã được đáp ứng đầy đủ.

## Chứng nhận.

Github Repo: [thang-huynhduc/Quality-Assurance-Projects: freeCodeCamp Certificate](https://github.com/thang-huynhduc/Quality-Assurance-Projects)

Khi hoàn thành xuất sắc cả 5 dự án trong chương trình học, sinh viên sẽ được cấp chứng nhận "Quality Assurance". Đây là minh chứng cho việc đã nắm vững các kỹ năng về cả front-end và back-end, cũng như kỹ thuật đảm bảo chất lượng phần mềm thông qua kiểm thử.



