Online Bookstore JSON API

Bradley Marques

February 6, 2022

1 Introduction

A proof-of-concept bookstore JSON API was developed using Ruby on Rails in a docker container. The API allows a user to authenticate by getting a JSON Web Token (JWT). The API also allows a user to index, show, update, create, and delete books. Best practice of Test-Driven-Development (TDD) was followed, as well as the Ruby on Rails "convention over configuration" paradigm. The problem was decomposed into constituent models of a User and a Book (Section 2.1). These two models are expanded upon by considering their database representation in a relational database (Section 2.2). The required API endpoints and responses are designed (Section 2.3), and assumptions about the domain are highlighted (Section 2.4). Ruby on Rails was chosen because of familiarity and therefore speed of development, as well as relevant software packages to handle testing, data seeding, JSON API serialization, etc. (Section 3). Not all work was completed within the deadline (Section 4). Technical trade-offs are briefly discussed (Section 5) as well as work required for production readiness (Section 6).

2 Problem Analysis

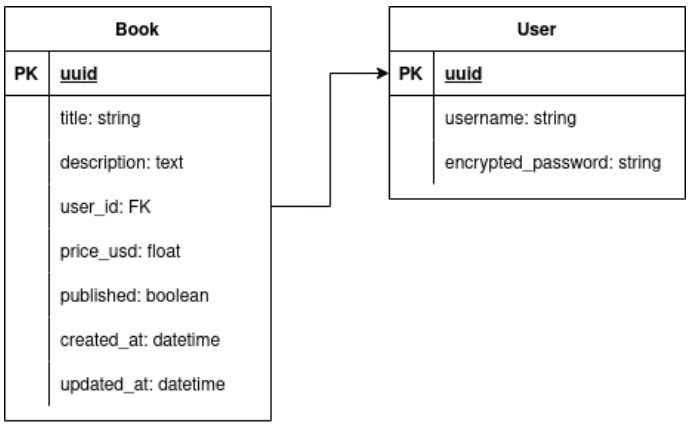
2.1 Domain

The domain contains the following models:

- **User** - (a.k.a. Author) the user of the system. Has API credentials in the form of username and password and has zero to many books.
- **Book** - a Book that an Author has written and wishes to self-publish.

2.2 Models and Database design

The entity relation diagram below was created before implementation:



Some notes:

- To prevent "database walking" and therefore increase application security, universally unique identifiers (uuids) are used instead of sequential database ids.

- The `published` and `encrypted_password` fields were planned but not completed within the four-hour deadline (see Section 4)

2.3 API Endpoints

The application requires REST endpoints to perform CRUD actions on Book resources. It is also prudent to version APIs to allow for maintainability when new versions are developed.

Therefore, the following API endpoints were implemented:

1. `GET base_url/api/v1/books` to index/list all Books.
2. `GET base_url/api/v1/book/:id` to show a single Book.
3. `POST base_url/api/v1/books` to create a new Book.
4. `PATCH base_url/api/v1/book/:id` to update a Book.

The following API endpoints were planned but not completed within the deadline:

1. `POST base_url/api/v1/token` to get a new JWT.
2. `DELETE base_url/api/v1/books/:id` to delete/un-publish a Book.

2.4 Assumptions Made

The following assumptions are made for the purposes of the proof-of-concept:

- This application will be deployed once per bookstore, and there is therefore no need to model a "Store" in the domain.
- There is a one-to-many relationship between a User and Books (i.e. Books cannot have multiple Authors, and a User can have zero to many Books).
- Book prices are in United States Dollars (USD).
- There are no requirements at the moment for user management.
- Pagination is not required on the Book list endpoint (i.e. small number of resources).
- When listing/indexing books, they should be sorted by title in alphanumeric order.

3 Choice of Technology

The list below summarizes the choice of technology used and gives a brief justification for each choice:

- **Containerization**: Docker
- **Programming Language**: Ruby 3.1.0 - the latest stable version at time of writing.
- **Framework**: Ruby on Rails 7.0.1: chosen for familiarity and therefore speed of development. Latest stable version at time of writing.
- **API standard/specification**: JSON API: gold-standard for JSON API responses.
- **Database**: PostgreSQL: powerful and feature-rich open-source relational database

Additionally, the following Rails-specific gems were used:

- **Authentication**: JWT as specified by brief.
- **Authorization**: Pundit - gem for simple OOP authorization.
- **Testing**: Minitest
- **Database Seeding**: Factory Bot
- **Seed data generation**: Faker

4 Work Completed and Not Completed Within Deadline

A four-hour deadline was imposed on the proof of concept, and the entire brief was not fulfilled. Work successfully completed includes:

- Containerization using Docker.
- Authentication using JWT (mocked in tests).
- Authorization:
  - A Book can only be updated by its Author.
  - A Book can only be deleted by its Author.
- JSON REST API to:
  - Index all Books as an unauthenticated or authenticated User.
  - Show a Book as an unauthenticated or authenticated User.
  - Create a Book as an authenticated User only.
  - Update a Book as an authenticated User only.
- Model tests.
- Controller tests.

Work not completed in the timeframe includes:

- Credential obfuscation.
- API endpoints to authenticate using JWT (as mentioned, this was mocked in tests).
- Books have cover images.
- Books are searchable by title, author, or description.
- Pagination of Book resources on the index endpoint.

5 Technical Trade-offs

The following trade-offs were made:

- **Monolithic Application** - as it stands, the application is a monolith and is not decomposed into microservices.
- **Multi-author books** - books cannot currently be co-authored.
- **User roles** - at the moment, a User is by default an Author. However, as the system grew, it would be prudent to add the idea of a "role" such that users could be admins to manage the system, bookstore owners to manage stock, etc.
- **Pagination** - currently there is no pagination on the books index action. If there were millions of books, this endpoint would be unusable.
- **Search** - currently, a user is unable to search through books.
- **Cover Images** - this was deemed out-of-scope. Cover images would not sit in the relational database, but would rather sit on something like Amazon S3. This is therefore almost a separate system.

6 Work Required for Production Readiness

To achieve production-readiness, the following is required:

1. Complete the brief by completing work currently unfinished (Section 4)
2. Decide on hosting for the web application (example: Heroku)
3. Decide on hosting for the database (example: Heroku PostgreSQL)
4. Decide on hosting for cover images (example: Amazon S3)
5. Develop continuous integration and continuous development (CI & CD) pipeline (example: GitHub actions)
6. Implement logging (example: Papertrail)
7. Implement API documentation for developers to consume
8. Implement load-balancing if not already provided
9. Implement API throttling to prevent DDOS attacks