Engineering Advanced Web Applications Coursework Documentation

Indrit Caca

Table of Contents

1. Introduction

1.1 Deployment Information

The web application is deployed on Render, a web application hosting cloud platform.

* The web application has been deployed using the free tier of the Render cloud platform, and as such is subject to inactivity shutdowns after 15 minutes. When receiving an incoming request, the service will come back up, but with a noticeable delay. As such it is recommended to allow the application some time (around 5 minutes at maximum) during the first request and after that, there should not be any delay with subsequent requests.

URL of deployed web application: https://saas-traffic-app-production.onrender.com

Hardware and software specifications are as follows:

Ruby Version: 3.3.0-p0

Rails Version: 7.1.2

Node Version: 20.10.0

Database: PostgreSQL 15

Configurations regarding deployment are shown in render-build.sh file, a shell file responsible for building assets and gems, migrating migrations and seeding the database with pre-specified records and render.yaml file, a YAML file responsible for connecting with Render PostgreSQL database as well as running initial server starting scripts.

1.2 How to Run

1.2.1 Prerequisites

Web application development machine specifications:

Ruby Version: 3.3.0-p0

Rails Version: 7.1.2

Node Version: 20.10.0

Database: PostgreSQL 15

1.2.2 Setting up the database

1. Download and install PostgreSQL

2. After setting up PostgreSQL, create a user in PostgreSQL

3. Go to database.yaml file and in development and test section replace username and password with username and password of the created user. Given that by default the PostgreSQL will be hosted on 127.0.0.1, the host doesn't need to be changed and the same thing goes to the database name which is related to the name of the web application.

1.2.3 Building the web application

1. Open a corresponding terminal in your machine with elevated privileges.

2. Execute command: bundle install. This will install all the gems specified in the project.

3. Execute command: rake db:create. This will create the necessary databases

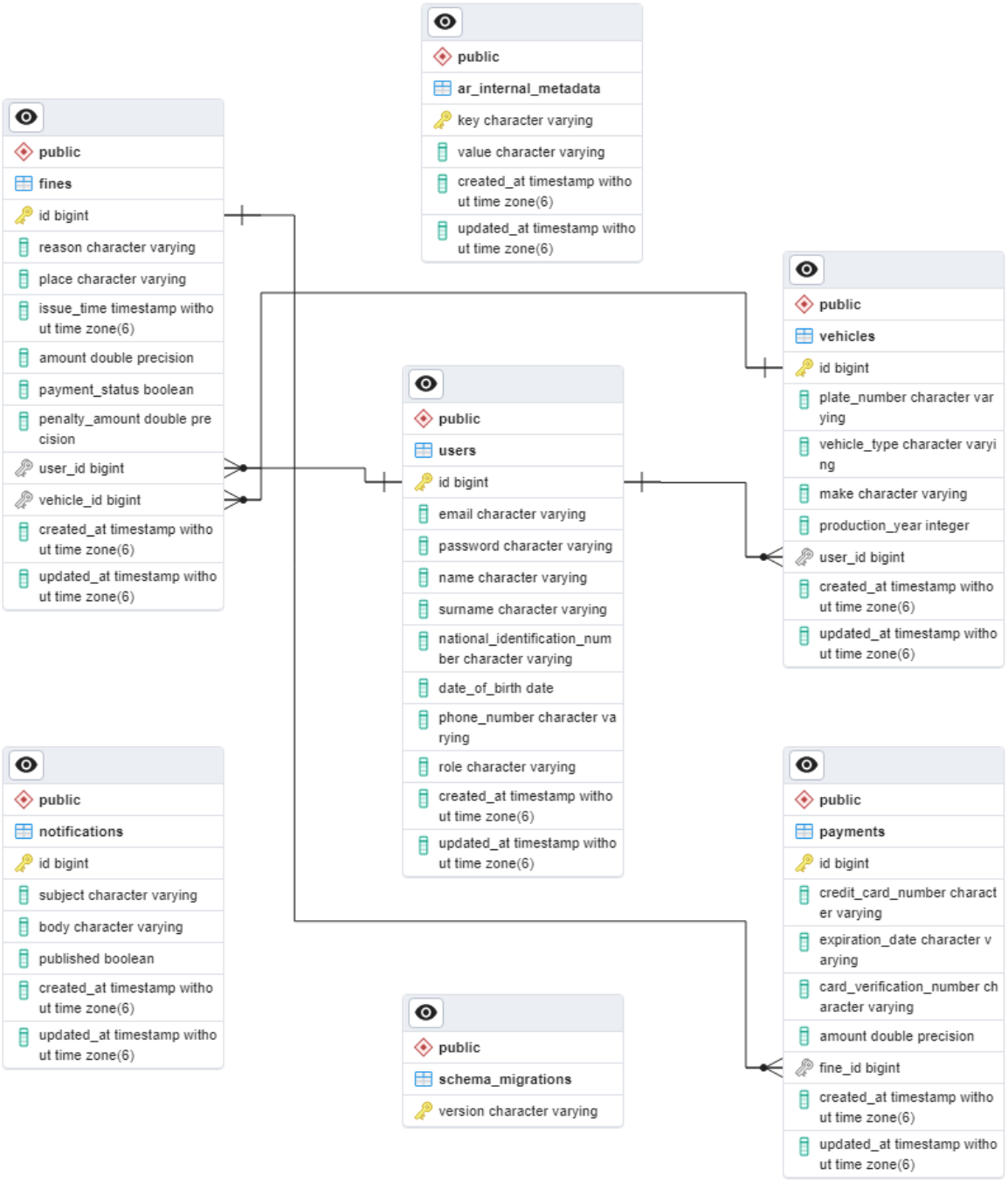
4. Execute command: rake db:migrate. This will use the migrations from the project to create corresponding database tables.

4. Execute command: rake db:seed. This will use pre-existing hashes to seed database tables

5. Execute command: rails server. This will start the web application. By default, Rails operates on port 3000, so the initial URL will be http://127.0.0.1:3000

After that, the program should be up and running.

2. ER Diagram



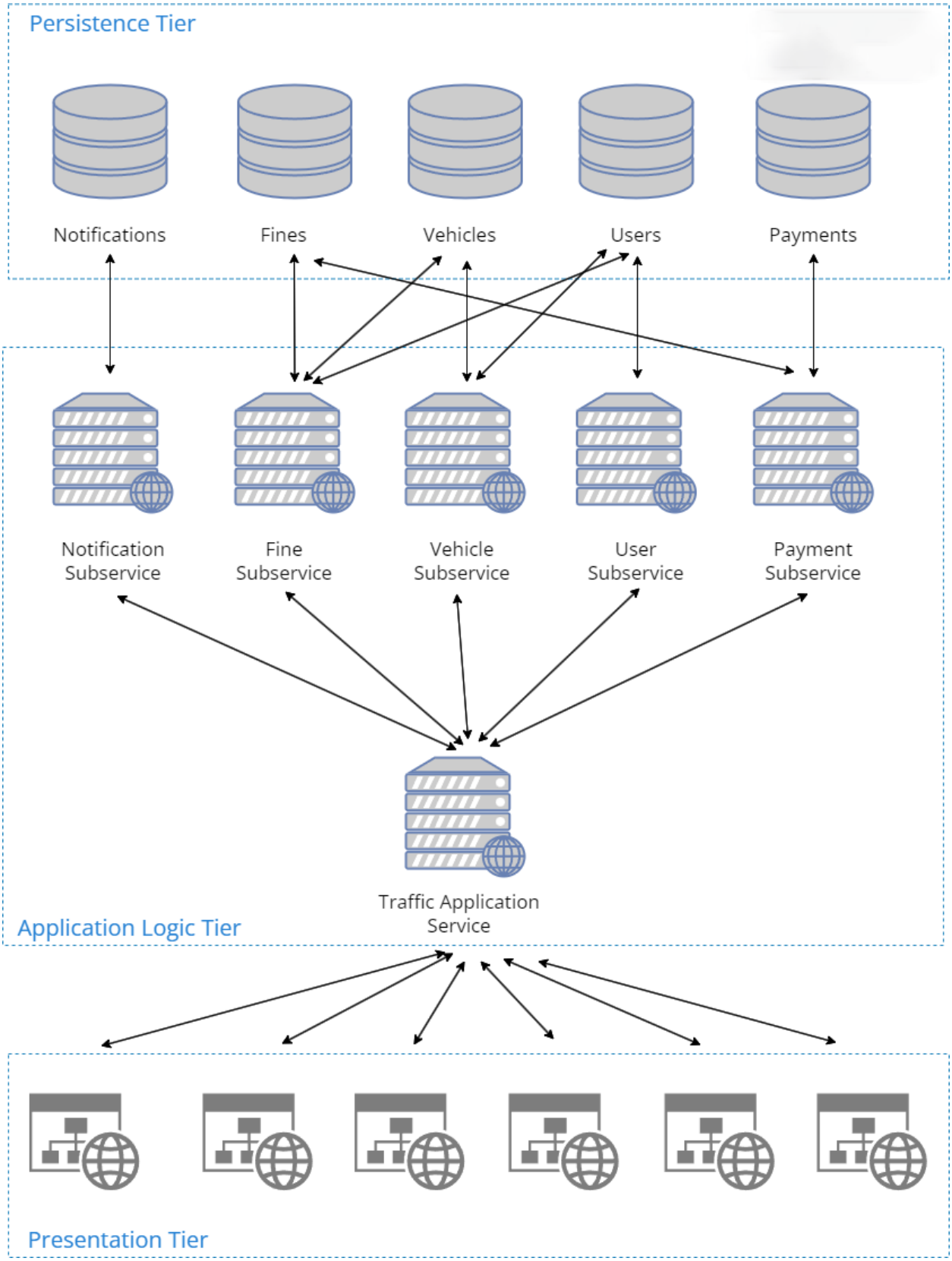
3. Application Architecture

3.1 Application architecture description

The application is SaaS (Software-as-a-Service) application, based on the monolithic design where all the resources have their own databases and their own subservices, all contained inside a single application service exposed to users. Based on this information it follows the multi-tiered client-server MVC model with the 3 distinct tiers:

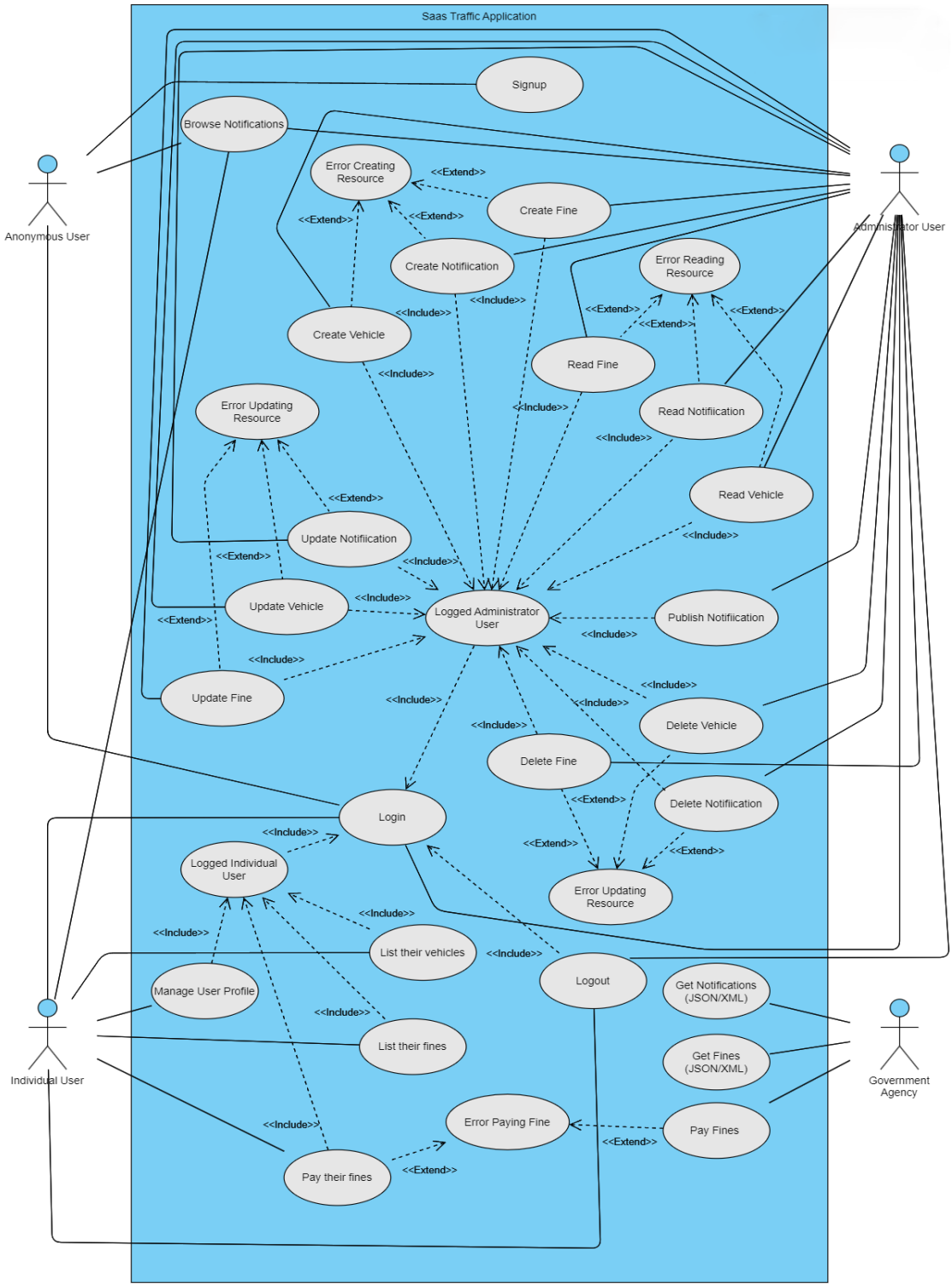
- Presentation tier with views which can be accessed by calling the specified API route made available by the system which ultimately corresponds to a specific resource subservice and as response they get either a HTML document which is accessible by the user clients through the usage of a web browser or XML/JSON which is returned for external actors like the governmental APIs;
- Application logic tier with controllers, which performs the application logic by operating on resources based on subservice called and the specific business logic required on that resources and by mediating as well as handling and checking the communication that is transmitted between corresponding views and the model of the given resource;
- Persistence or data tier with models which represents the entities (resources) in a relational format with attributes of the model corresponding to the columns in the database. This tier takes care of creating, reading, updating or deleting into the system, as well as validations necessary when this data is inserted.

3.2 Application architecture diagram


Persistence Tier
Notifications
Fines
Vehicles
Users
Payments
Notification Subservice
Fine Subservice
Vehicle Subservice
User Subservice
Payment Subservice
Traffic Application Service
Application Logic Tier
Presentation Tier

4. Use Cases

4.1 Use case diagram



4.2 List of use cases

In this application there are 4 types of actors: Anonymous users, Individual users, Administrator users and Governmental agencies. Below are use cases perform by these actors.

Anonymous users:

- Anonymous users can browse notifications.
- Anonymous users can log in.
- Anonymous users can sign up.

Individual users (can perform all the use cases of Anonymous users except signing up):

- Individual users can manage their profile.
- Individual users can list their vehicles.
- Individual users can list their fines.
- Individual users can pay their fines.
- Individual users can log out.

Administrator users (can perform all the use cases of Anonymous users except signing up):

- Administrator users can manage their profile.
- Administrator users can create notifications.
- Administrator users can update notifications.
- Administrator users can show notifications.
- Administrator users can publish notifications.
- Administrator users can delete notifications.
- Administrator users can create vehicles.
- Administrator users can update vehicles.
- Administrator users can show vehicles.

  Administrator users can delete vehicles.
- Administrator users can create fines.
- Administrator users can update fines.

- Administrator users can show fines.
- Administrator users can delete fines.

Governmental agencies:

- Governmental agencies can get notifications information.
- Governmental agencies can get fines information.
- Governmental agencies can pay fines.

5. Lessons Learned

1. DRY-ing the views improves readability.

   When initially working with the ERB files to create the views, I noticed that many controllers repeated the CRUD functionality buttons as a HTML part on many resources. The copy-paste technique seemed to work at first until there was an error I was not finding in one of the buttons. After that I decided to replace this part with a layout and simply adding the paths of the buttons. Immediately, the solution worked and the views were shorter and more readable.
2. RESTful routes reduce redundant methods.

   Initially when implementing the API methods for returning the notifications and fines as XML and JSON, I had implemented such functionality with a method for a specific format, meaning that to get the list of notifications there were 3 methods, 1 for HTML, 1 for JSON and 1 for XML. After going through the REST documentation, I found out that the REST provides a mechanism for making data available in multiple formats in a single method, allowing me to remove entirely the redundant methods.

6. Future Development

Regarding future development, it would worth mentioning the transition from the monolith architecture in service-oriented architecture where each resource (and as the result, the resource's database and the resource's service) on the system could be distributed, making it a microservice, and the operations on the resource could only be accessed by the microservice's interface exposed. Just like the APIs exposed for the governmental agencies work by exchanging JSON or XML, the same format could be applied for these microservices. While it is a more complex approach, it would enable the application to scale as the number of users, fines or vehicles increases. One such case can be for example going from using the traffic application in a city to using the traffic application in the whole country. Given such requirements, this proposed approach would suit the application better in case of a future development.