Assignment Report

# Summary

The Store API is an ecommerce platform of some sort that has two types of users, a customer and a store administrator. A customer mostly desires to browse and review products and where applicable buy them. An administrator on the other hand seeks to perform administrative tasks like creation of other administrators, system roles, manufacturers etc.

With that said the Store API is split into a RESTful API and 1..N front-ends so that if the need arises, different clients can be written to consume the API. For example, if we had users that would like to use the platform through an iOS application then all that would be needed will be to create the iOS client. This kind architecture is robust and more responsive to change.

# Database Design

The design of the database is driven by the following entities:

1. User
2. Customer
3. Product
4. Review (required for the customer to review products they have purchased)
5. Category (required so that we can place a product to the category it belongs)
6. Manufacturer (necessary so that a product should have a manufacturer)
7. Discount (required so that discount can be applied accordingly)
8. PaymentMethod (required for a customer to select a mode of paying)
9. Role (required for the necessity of user classes)

**Relationships:**

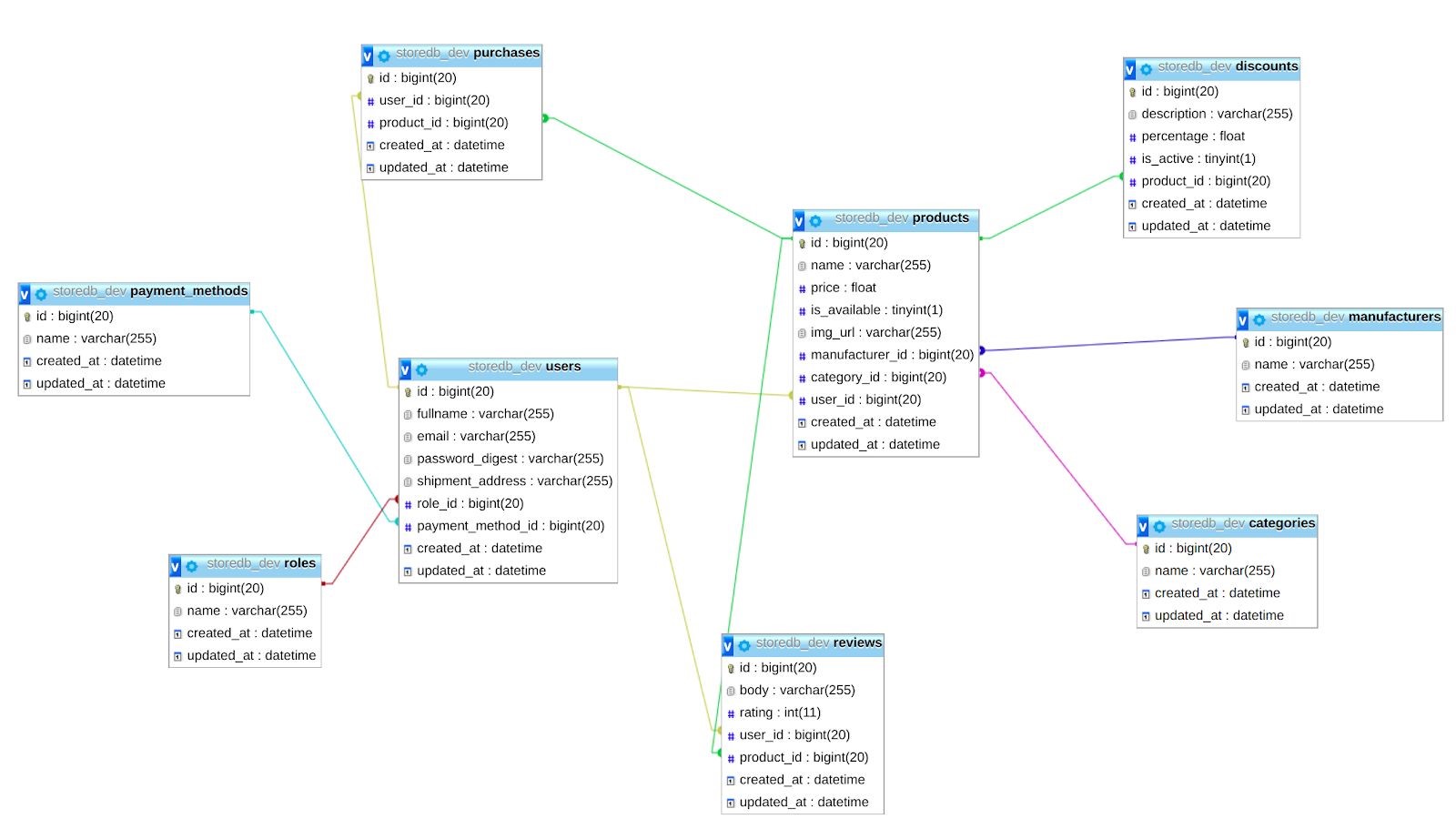
* A Product belong-to Manufacturer
* A Product belong-to Category
* A Product belongs-to User (was created by the concerned User)
* A Discount belong-to Product
* A Review belongs-to Product
* A Review belongs-to User
* A User has-a Role
* A User has-a PaymentMethod

**Denormalizations:**

* User and Customer will be stored in the same table. Yes, there will be Users with null fields but the number of Customers will greatly outnumber the number of Users so the effect of that denormalization is negligible.

**Schema Image:**

Figure 1:



# User Requirements

**Functional:**

* Customers should be able to register
* Customers should be able to search for products using various filters like category and manufacturer
* Customers should be able to purchase products
* Customers should be able to review items
* Administrators should be able to add products
* Administrators should also be able to search like customers
* Administrators should be able to view customer details and purchase histories
* Administrators should be able to update stock

**Non-functional:**

* The platform should only allow authenticated and authorized users
* The platform should allow concurrent database connections

# Architectural Design

**Logical Design:**

Figure 2:

Interface-compliant client

RESTful API

Interface-compliant client

Interface-compliant client

Interface-compliant client

**Process Design:**

To illustrate a singular process design we will consider the scenario in which a user wishes to create a Role.

Figure 3:

REST Client

A

U

T

H

Roles Controller

Role Model

DB

1. First the request is made and it hits the Authentication layer/controller which determines if the request is authentic and authorised. If so, then let it hit its destination which in this case is the RolesController. If not, then return HTTP status 401.
2. Upon hitting the RolesController, it establishes that all necessary parameters for creating a role are met. If not, then return HTTP status 422 (unprocessable entity). If yes, then proceed to ask the model to create the Role.
3. The Role model creates a Role instance and tries to commit it to the database, if any error occurs then the platform responds with HTTP status 500 (internal server error). If not then it bubbles back up to the RolesController.
4. On success the RolesController then responds with HTTP status 204 (created).

**Rationale:**

* A RESTful API allows for the same resources to be reusable across multiple clients which reduces the amount of work and code required to support multiple platforms or clients.
* REST allows us to define concrete interfaces that clients can use to interface with our system. Concrete interfaces often equal to less chances of misinterpretation.
* With REST, as long as the API is well documented then almost everyone can build a client for it.
* REST liberates us to use multiple stacks since almost every language can speak HTTP.

# Use Cases

|  |  |  |  |
| --- | --- | --- | --- |
| Actor | Action | Description | Condition |
| Customer | register | Be able to use the front end of the App | App Must be Active |
| Customer | search product catalog | Be able to view stock and get access to a purchasing option | Must be a registered Customer |
| Customer | purchase products | Gain the desire to browse through products | Product must be available and in stock |
| Customer | review products | Be able to share views on products to help the store in the building of some valuable hierarchy | The product must be available to post a review |
| Administrator | add products | Be able to update stock | Must be registered as Admin |
| Administrator | search products | Be able to view all stock | Product must be available in the list |
| Administrator | view customer details and purchases | Be able to view customer activity and purchase processes | Must have full access of the system |

# Implementational Design Patterns

* Decorator design pattern is an architecture used to ​**add​** functionality or names to **classes and/or modules​**. This is a fairly more adaptable technique of assigning functionality than inheritance.

The pattern has been used both in the React client and REST API. In the client it has been used to ​support**​** the reusable components with access to the application’s global state and state methods. This allows components to have access to the global application state instead of using other uglier methods.

In the REST API it has been used to support controllers with custom error handling capabilities and authentication methods and names.

* Observer design pattern is a behavioural design pattern that allows entities to subscribe to changes to state of another entity. The concerned entity keeps track of all other who want to be notified of changes, ​**subscribers​**, and when a change occurs, cycles through whilst notifying each of them of the change.

This in contrast to other methods allows us to write less code to implement similar behaviours. For example, in the front-end ​**Redux​** internally implements the observer pattern so that whenever state changes, any entities using the concerned names are notified.

* Singleton design pattern is a pattern that is used when there is only ever going to be one instance of an entity. All other subsequent requests for an instance are satisfied by sending a reference to the already created instance.

This pattern has mostly been used in the REST API to implement the ExceptionHandler, Messages class, JWT code and auth code. You’ll notice that you’ll ever only need one instance of these. More instances would not make sense and just pollute application state.

* Component-based view design. Not a standard-term but what it means in this context is to organize our HTML+JS+CSS under one reusable namespace to avoid code duplication. For example, (in the React client), instead of writing multiple NavBar code snippets in each view we ​**abstracted​** the Navbar away so that any view that uses it need only import it and refer to it through <NavBar />.

* Module bundling. Not a pattern perse, but an approach to front-end development that involves building a single file (bundle) of all the code the front-end application uses. This helps in dropping unused names in dependencies and reduces overhead when it comes to dependency resolution in production. In this case the React client uses ​**webpack ​**(a module bundler that uses the tree-shaking algorithm).

* Flux data architecture, is a way of architecting applications. Flux is mostly concerned with how the application handles state. It aims for methods of handling state that are functional and deterministic. This in turn assists developers in developing less error-prone and easy-to-digest applications.

With that said, I am actually a beginner on this architecture and I got more insight and understanding by using ​[this link here](https://facebook.github.io/flux/)​ which helped me to realise better the benefits and values of using this design.

However, on the other hand, one hallmark of good software engineering is to not just do lots of objects (designs) because you can easily do it but about being able to process the required procedures. At this stage the platform is all but CRUD so throwing in a lot of patterns in there would not make sense.

For example, one could say put the domain logic in service objects and turn the controllers into request-response objects and the models into data-definition objects but of what utility would that be when all you have under (POST /reviews) is one statement? At this stage none, so it is therefore appropriate to leave it that way.

Technology Stack

* **React​** is a JavaScript library for building user interfaces.
* **Redux​** is an open-source JavaScript library for managing application state. It is most commonly used with libraries such as React or Angular for building user interfaces.
* **ESLint​**. ESLint is an open source project originally created by Nicholas C. Zakas in June 2013. Its goal is to provide a pluggable linting utility for JavaScript.
* **Ruby​** is a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.
* **Ruby on Rails​** is a web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern.
* **Webpack​**. At its core, webpack is a static module bundler for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph which maps every module your project needs and generates one or more bundles.
* **MySQL​**. MySQL is an open-source relational database that will enable a well-organized management of the store. It is reliable, stable, secure, compatible and powerful key to structured query language