

PROJECT Design Documentation

*The following template provides the headings for your Design Documentation. As you edit each section make sure you remove these commentary 'blockquotes'; the lines that start with a > character and appear in the generated PDF in italics but do so only **after** all team members agree that the requirements for that section and current Sprint have been met. **Do not** delete future Sprint expectations.*

Team Information

- Team name: Party Rockers
- Team members
 - Ryan Lembo-Ehms
 - Ethan Battaglia
 - Victor Rabinovich
 - Samuel Roberts
 - Alex Carron

Executive Summary

Party Rockers is an ecommerce website that customers can buy rocks to fulfill all of their geological desires. On this website customers can sign in with their own username and secure password, at which point they will have access to the many rock on the website. Customers will be able to look at the details of their rocks and even customize them. Once they found the items of their choice they'll be able to add them to their cart at which point they will be able to checkout and get the rocks of their dreams. Admins will be able to set inventory and add/edit/remove items from the store.

Remember it's not just a boulder it's a rock.

Purpose

We are developipng this website as a term project for a class in preparation of our peers using our website at the end of the semester.

This project involves building an e-store platform for selling rocks. They're are two main user groups.

E-Store Owners

Small business owners specializing in selling rocks who require a customized e-store for their business

They're user goals are to...

- Manage rock inventory.
- Have authentication for accessing the platform.
- Add, remove, and edit rock products.
- Ensure persistent data storage to reflect changes made by users.

Customers

Users interested in purchasing rocks from the e-store

They're user goals are to...

- View a list of available rocks.
- Search for specific types of rocks.
- Add or remove rocks from the shopping cart.
- Complete purchases securely through checkout.

Glossary and Acronyms

[Sprint 2 & 4] *Provide a table of terms and acronyms.*

Term	Definition
SPA	Single Page
OO	Object Oriented
MVVM	Model View ViewModel
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
DAO	Data Access Object
API	Application Programming Interface

Requirements

This section describes the features of the application.

Definition of MVP

[Sprint 2 & 4] *Provide a simple description of the Minimum Viable Product.*

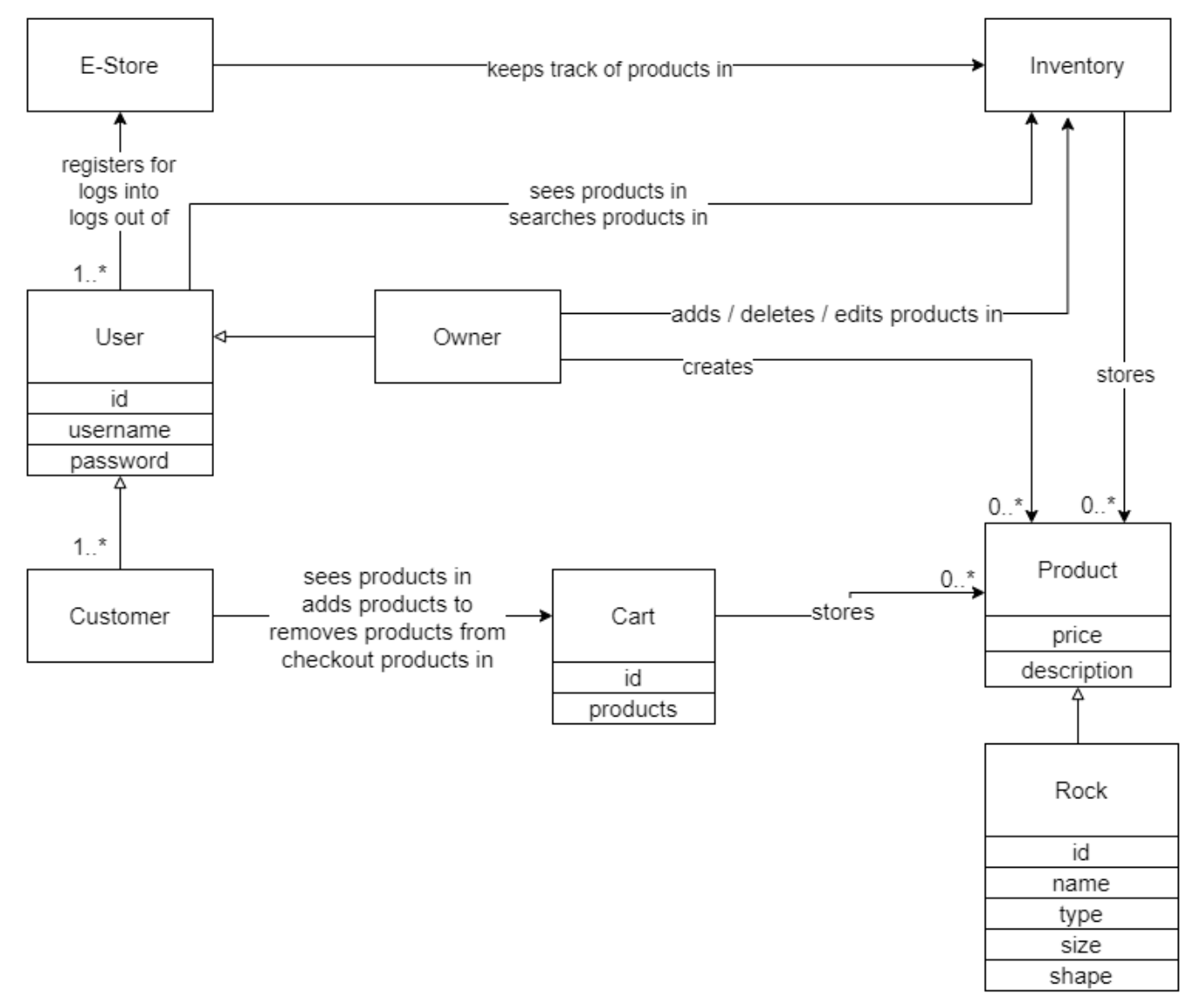
Enable users to register and log in with basic authentication. Customers can browse a list of rocks, search by name, add/remove items from their cart, and proceed to checkout. Admins can manage inventory by adding/removing rocks, updating details, and setting quantities. Data persistence ensures continuity across user sessions. Additional features include rock customization for customers and password authentication for enhanced security.

MVP Features

- Creating and signing into an account
- Shopping carts tied to a specific user
- Inventory management

Application Domain

This section describes the application domain.



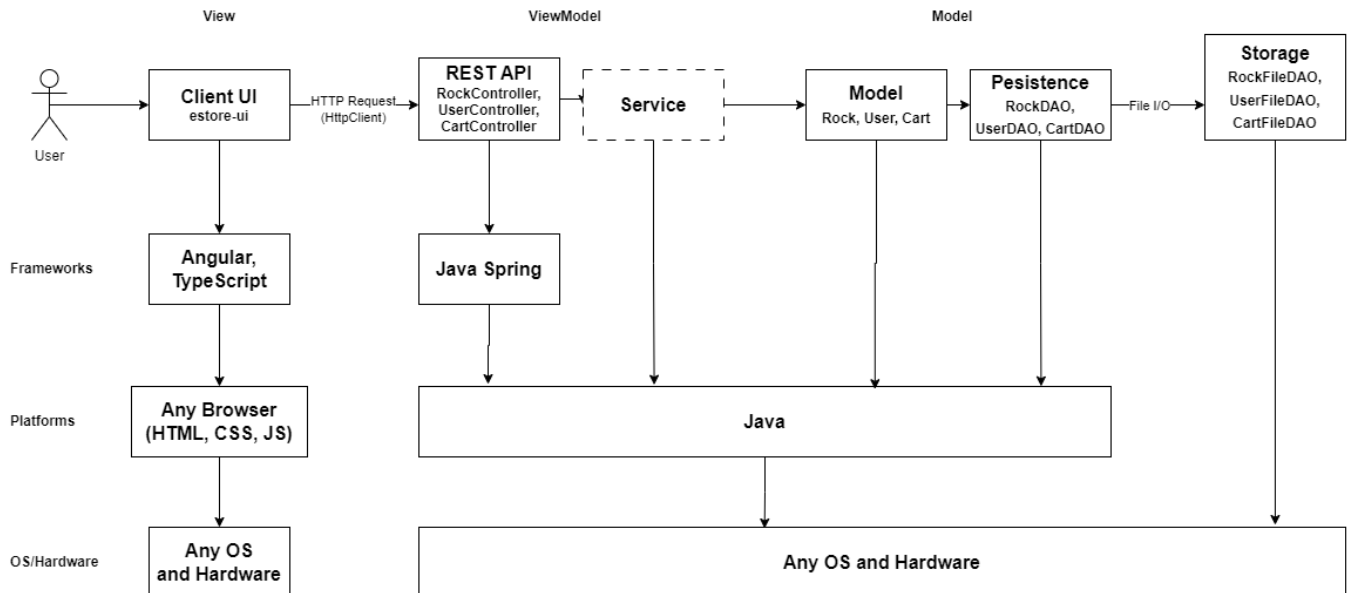
[Sprint 2 & 4] Provide a high-level overview of the domain for this application. You can discuss the more important domain entities and their relationship to each other.

The domain of this application is an e-store system where users interact with products in an inventory. Users, who can be either an owner or customers, register, log in, and log out of the e-store. Users browse products in the inventory, search for specific items. Only customers add them to their cart and proceed to checkout. Only owners can manage products in the inventory.

Architecture and Design

Summary

The following Tiers/Layers model shows a high-level view of the webapp's architecture. **NOTE:** detailed diagrams are required in later sections of this document.



The web application, is built using the Model–View–ViewModel (MVVM) architecture pattern.

The Model stores the application data objects including any functionality to provide persistence.

The View is the client-side SPA built with Angular utilizing HTML, CSS and TypeScript. The ViewModel provides RESTful APIs to the client (View) as well as any logic required to manipulate the data objects from the Model.

Both the ViewModel and Model are built using Java and Spring Framework. Details of the components within these tiers are supplied below.

Overview of User Interface

This section describes the web interface flow; this is how the user views and interacts with the web application.

Provide a summary of the application's user interface. Describe, from the user's perspective, the flow of the pages in the web application.

ViewModel Tier

- **RockController Class** - This class creates the REST API calls such that the Angular view can send a http request to the ViewModel and a corresponding action from the rock model takes place. This class creates the urls for the calls, and defines what functions it will call, status codes to return and error handles.
- **UserController Class** - This class creates the REST API calls such that the Angular view can send a http request to the ViewModel and a corresponding action from the User model takes place. This allows for calls to create new users, updating users, retrieving users, and gathering the information necessary to sign in users.
- **CartController Class** - This class create the REST API call such that the Angular view can sent a http request to the ViewModel and a corresponding action from the user model takes place. This allows for calls to add, delete, and view the shopping cart.

[Sprint 4] *Provide a summary of this tier of your architecture. This section will follow the same instructions that are given for the View Tier above.*

*At appropriate places as part of this narrative provide **one** or more updated and **properly labeled** static models (UML class diagrams) with some details such as critical attributes and methods.*

 Replace with your ViewModel Tier class diagram 1, etc.

Model Tier

- Rock - This class defines what a rock object should look like, and the properties it holds.
- RockDao - This class creates an interface in order to access or manipulate the information related to the rock object which can then be implemented by some storage system.
- User - This class defines what a user object should look like, and the properties it holds.
- UserDao - This class creates an interface in order to access or manipulate the information related to a user which can then be implemented by some storage system.
- Cart - This class defines what a shopping cart object should look like, and the properties it holds.
- CartDao - This class creates an interface in order to access or manipulate the information related to a shopping cart which can then be implemented by some storage system.
- Password - This class handles the methods for checking if a password meets our security requirements, hashing passwords, and generating strong passwords.

***[Sprint 2, 3 & 4]** Provide a summary of this tier of your architecture. This section will follow the same instructions that are given for the View Tier above.*

The model tier is the core of the application. It defines essential classes and interfaces for managing rocks, users, and shopping carts. These components manipulate the data of and interaction with the application to make sure the business logic and information is handled consistently

*At appropriate places as part of this narrative provide **one** or more updated and **properly labeled** static models (UML class diagrams) with some details such as critical attributes and methods.*

 Replace with your Model Tier class diagram 1, etc.

OO Design Principles

We have considered the following OO principles for our project:

- Single Responsibility - Each class is responsible for one task and should be very good at that one task
- Dependency Inversion - High level classes should not rely on lower level classes instead they should rely on abstractions
- Information Expert - The class that has the information needed to complete a task should be the one to implement it
- Open/Closed - A class should be extended by another class not modified
- Law of Demeter - Keep coupling low. Meaning that a class should only use the methods of the classes directly linked to it.
- Pure Fabrication - Creating helper classes to maintain single responsibility

***[Sprint 2, 3 & 4]** Will eventually address up to **4 key OO Principles** in your final design. Follow guidance in augmenting those completed in previous Sprints as indicated to you by instructor. Be sure to include any diagrams (or clearly refer to ones elsewhere in your Tier sections above) to support your claims.*

- **Single Responsibility:** Single Responsibility is used through our controller classes when we use them to handle all our api calls and doesn't need to care about how the data is stored. Similarly, the DAO does not worry about how data is being asked for as it just interacts with the controller. If you look at the tiers and layers diagram within the architecture and design summary it can be seen how the classes in the model have different responsibilities. Some perform the actual model, others are persistence for API call handling, and then storage. And within these different tiers there are classes to handle certain information whether it be related to users, carts, or rocks.
- **Dependency Inversion:** Dependency Inversion tells us that high level modules should not rely lower level modules instead they should each rely on abstractions. We implemented this in our model via the RockDao, UserDao, and CartDao classes. These classes are abstract classes that define method headers that can be used by a lower level class to define the behavior of each method, and used by a higher level module to call these methods so it can retrieve information. If the way we access or store our objects changes we can create a new implementation of the class without affect our higher level http calls. As seen with our model class diagrams we have the DAO classes that are abstract classes that are interfaces for the FileDao classes to implement. Also seen in the model class diagram is how the controllers use the interface rather than directly connecting to the FileDao implementations. This is further supported with the following code snippet which displays the constructor for the Rock Controller and demonstrates how a dependency can be injected as long as it implements the RockDao interface.

```
public RockController(RockDAO rockDao) {  
    this.rockDao = rockDao;  
}
```

- **Pure Fabrication:** One of the classes that is a pure fabrication in our project is the Password class. Password does not represent an entity

[Sprint 3 & 4] OO Design Principles should span across **all tiers**.

- **Controllers** are used when needed to make api calls to any of our objects instead of having the UI directly interact with our data.

Testing

This section will provide information about the testing performed and the results of the testing.

Acceptance Testing

[Sprint 2 & 4] Report on the number of user stories that have passed all their acceptance criteria tests, the number that have some acceptance criteria tests failing, and the number of user stories that have not had any testing yet. Highlight the issues found during acceptance testing and if there are any concerns. For Sprint 1 there were 7 user stories, all of which had all of their acceptance criteria pass.

For Sprint 2 we planned to complete an additional 11 user stories, at the end of which

- 8 had passed all of their acceptance criteria
- 3 that had mostly but not fully passed their acceptance criteria. The acceptance criteria that failed were
 - Given that I am on the products page when there are products in the inventory then I see each product and short description.

- ■ This had failed because we had yet to include all fields of a rock item in the details page. The following acceptance criteria failed due to the decision of moving away from the admin using product ids and rather interacting with a GUI, as a result the acceptance criteria will be modified or removed for sprint 3.
- Given a user is logged in as an admin when that user enters a valid product id and attempts to remove it then the product is removed from our inventory
- Given a user is logged in as an admin when that user enters an invalid product id and attempts to remove it then the product is not removed from our inventory
- Given a user is logged in as an admin when that user enters a valid product id and valid product information and attempts to update it then the product is updated in our inventory
- Given a user is logged in as an admin when that user enters an invalid product id or invalid product information and attempts to update it then the product is not updated in our inventory

As of Sprint 3 all failed acceptance criteria from Sprint 2 have passed.

Unit Testing and Code Coverage

[Sprint 2 & 4] Include images of your code coverage report. If there are any anomalies, discuss those.

Overall Code Coverage

In general, we had excellent code coverage. In fact, nearly all files were completely covered except a couple.























estore-api

estore-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.estore.api.estoreapi.persistence	<div><div></div></div>	84%	<div><div></div></div>	85%	10	64	28	184	4	37	0	3
com.estore.api.estoreapi	<div><div></div></div>	88%		n/a	1	4	2	7	1	4	0	2
com.estore.api.estoreapi.controller	<div><div></div></div>	100%	<div><div></div></div>	100%	0	34	0	135	0	22	0	3
com.estore.api.estoreapi.model	<div><div></div></div>	100%	<div><div></div></div>	100%	0	30	0	52	0	26	0	3
Total	139 of 1,750	92%	8 of 86	90%	11	132	30	378	5	89	0	11

Anomalies

We only have one anomaly, and it is the CartFileDAO.java class. This class was difficult to write tests for and eventually we weren't able to complete them all. More specifically, the challenges came from ensuring that file writing is done correctly.

estore-api > com.estore.api.estoreapi.persistence > CartFileDAO											
CartFileDAO											
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	
getRocksFromCart(Cart)		0%		0%	3	3	11	11	1	1	
addCart(int)		41%		50%	1	2	4	7	0	1	
addItem(int, int)		0%		n/a	1	1	5	5	1	1	
deleteItem(int, int)		0%		n/a	1	1	5	5	1	1	
save()		0%		n/a	1	1	3	3	1	1	
load()		100%		100%	0	2	0	5	0	1	
getCartsArray()		100%		100%	0	2	0	8	0	1	
getCart(int)		100%		100%	0	2	0	4	0	1	
CartFileDAO(String, ObjectMapper)		100%		n/a	0	1	0	5	0	1	
getCarts()		100%		n/a	0	1	0	2	0	1	
static {...}		100%		n/a	0	1	0	1	0	1	
Total	130 of 276	52%	5 of 12	58%	7	17	28	56	4	11	

Ongoing Rationale

*[Sprint 1, 2, 3 & 4] Throughout the project, provide a time stamp (yyyy/mm/dd): **Sprint # and description** of any **major** team decisions or design milestones/changes and corresponding justification.*

(2024/03/17): Sprint #2 Decided to have the admin only update and delete through the UI instead of using id's to interact with specific projects.