System Design Document for Hyro

TDA367, Chalmers tekniska högskola

Eimer Ahlstedt, Emil Lindblad, Sebastian Kvaldén, Timothy Nilsson, Erik Larsson

24 oktober 2021

Table of contents

1 Introduction

The purpose of this document is to give insight into the technical side of the application Hyro. Descriptions of testing methodology, software architecture, system design and more will be given in the following chapters.

Hyro, the application in question, is a simple platform enabling its users to participate in a sharing economy, and thus save both time and money. Using Hyro, users can both list items they own as available for rent and browse the listings of other users. The platform is meant to offer a beginning to endsolution, including adding items for rent, browsing, booking and paying.

Through Hyro users will be able to save money by renting items at a lower price than from big companies, minimize their environmental impact by maximising the potential of already produced items and let their personal belongings generate passive income.

1.1 Definitions, acronyms and abbreviations

- **Jira** - A tool used for Agile software management.
- **JSON** - JavaScript Object Notation. A human readable data format usually used for data interchange with webservers. We use it in our application for storing and persisting data between sessions.
- **JavaFX** - A Java framework used for creating graphical users interfaces.
- **GitHub** - A tool used for storing code repositories.
- **FXML file** - A type of file used by JavaFX. Contains information on visual elements and what code is connected to each element.
- **MVC design pattern** - A design pattern of object oriented programming. The goal of the pattern is to divide a code base into three parts, one Model, one View and one Controller, each one with different responsibilities.

2 System Architecture

The application is designed around the MVC design pattern, which means that the structure can be broken down into three different components: Model, View and Controller. All these components, each with their own responsibility work together to

2.1 Model

The Model component is the central part of the application. It contains and manages all the data in the application. It receives instructions from the controller on what to do. The model is then split up in internal packages such as User, Listing and Booking packages.

2.2 View

Anything that the end user can see and interact with is displayed and created by the View component. The specific views are sub classes to an abstract class to ensure they all behave the same and are exchangeable throughout the program.

2.3 Controller

Connection between the View and the Model is handled by the Controller component. The controller takes the user input from the View and decides what to get from the model and updates the view with the new data.

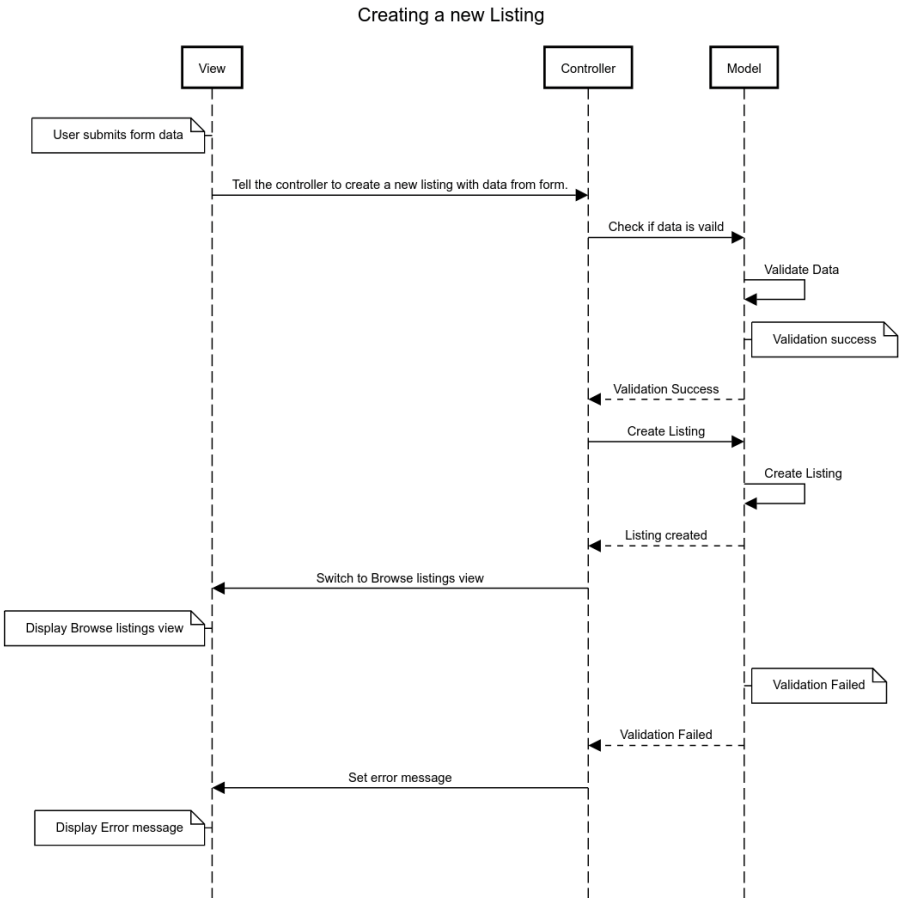
2.4 Databases

The application uses a basic form of databases to store data while the application is not in use. This data is then loaded into the model when the application is started and saved back to the files

when exited instead of continuously loading data from the database. The databases are connected to the data-handlers in the model via a *Read*/*Write*-classes.

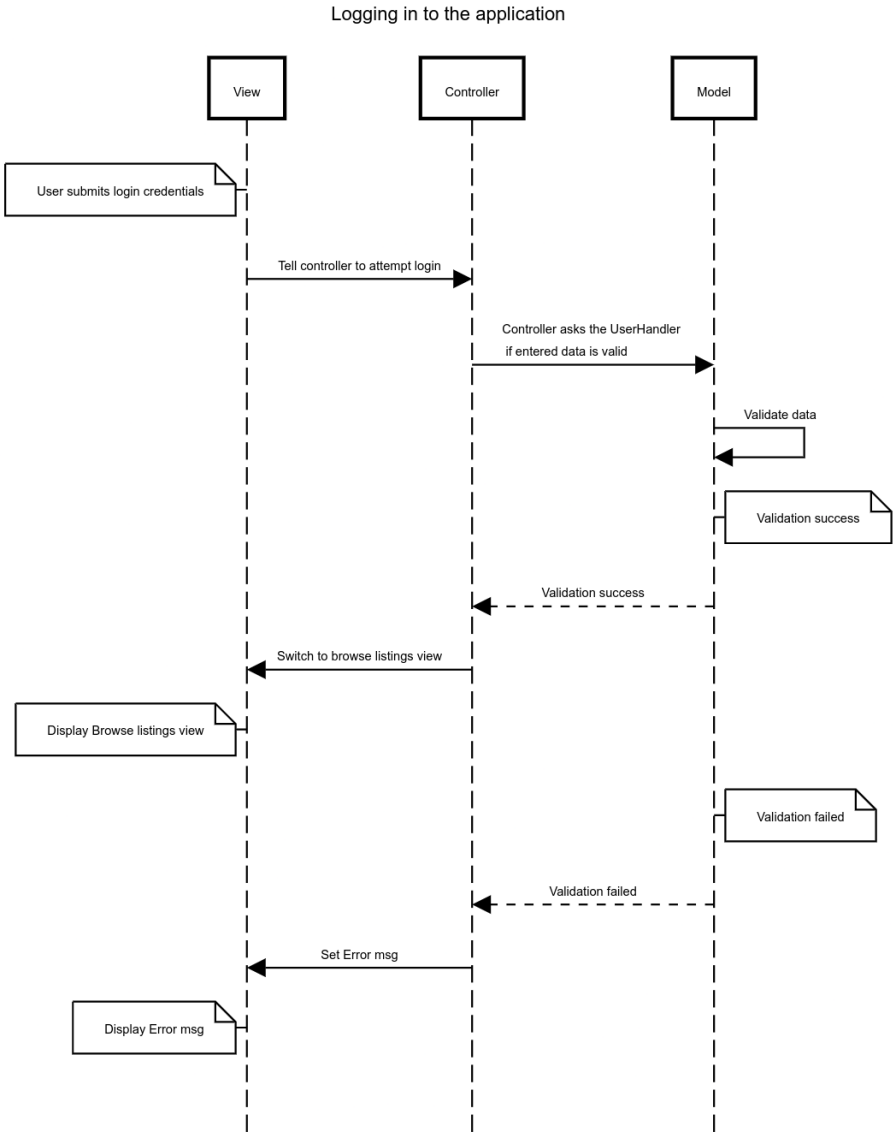
2.5 Application flow

A good way to describe how all these components work together is with a sequence diagram. In the example below (1), we will look at the flow though the application when creating a new listing.



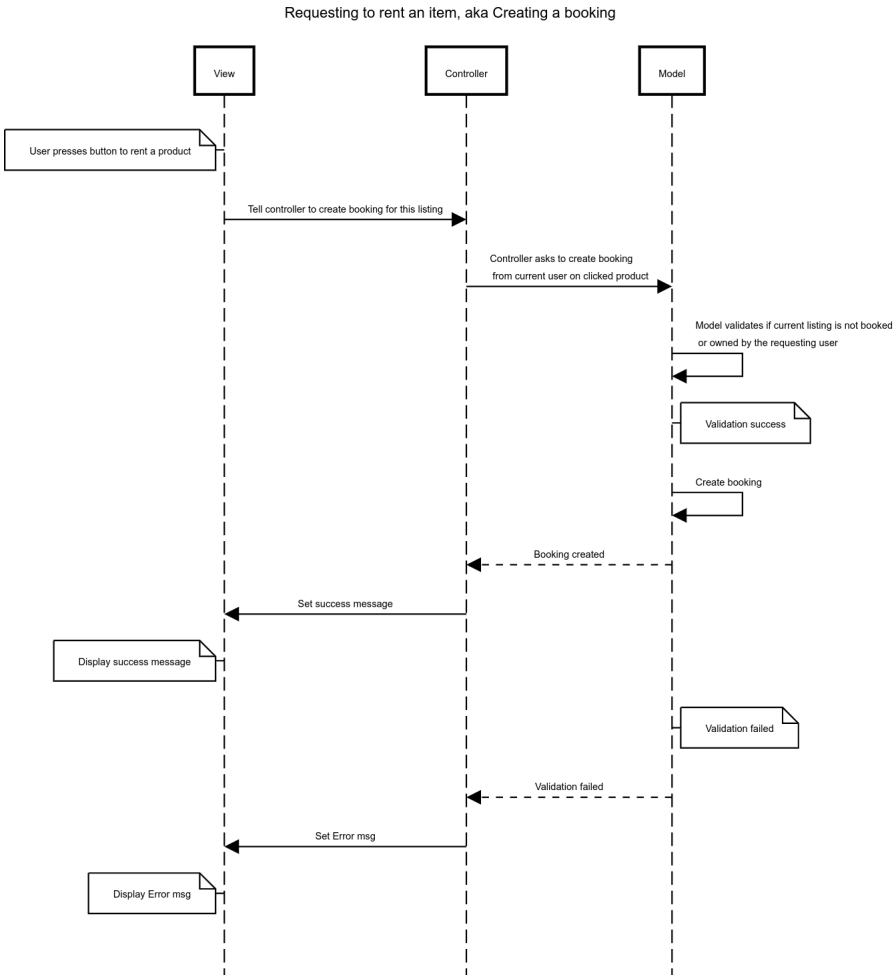
Figur 1: A sequence diagram for creating a new listing

A similar example of the flow though the application is when a users attempts to login:



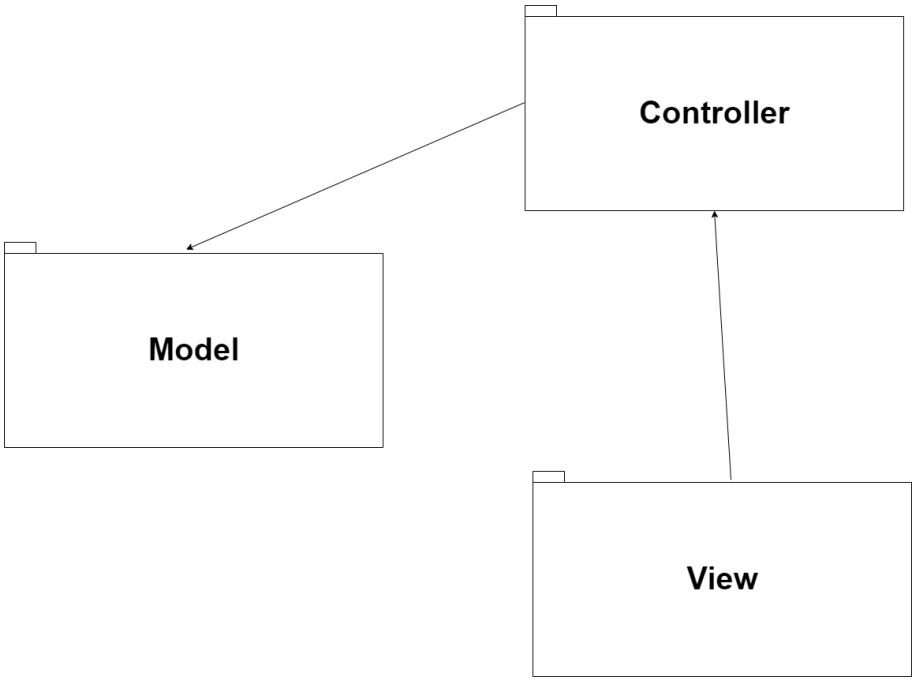
Figur 2: A sequence diagram for logging in to the application

This final example shows the flow for creating a booking of a listing



Figur 3: A sequence diagram for creating a booking

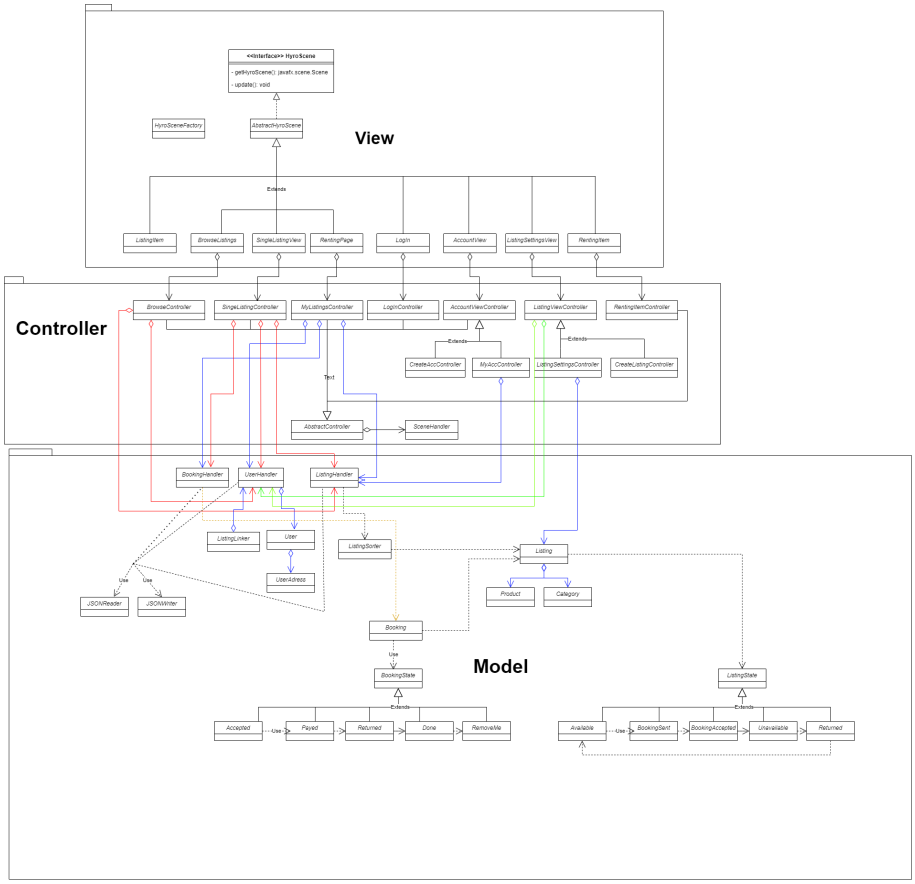
3 System Design



Figur 4: Package Diagram MVC

3.1 Model View Controller

MVC is implemented with the mindset that view should be ”dumb”, controller should be ”thin” and model should be ”smart”. The model is also completely independent from the others and controllers and view could be replaced without having to change the model.



Figur 5: UML Class Diagram of entire application

3.2 Relation between Design Model and Domain model

The back-end part of the domain model is strongly related to the design model. The domain model describes the application with the most crucial parts of the application and their dependencies, whilst the design model goes into detail about all the classes that are related to each parts. The dependencies between the big components are the same in the domain model and the design model. The only real deviation of this is the relation between user in the domain model and in the design model. The domain model describes the actual user using the application and what it comes in contact with. The design model describes the actual user class and its dependencies with the other classes.

3.3 Design Patterns

Other than MVC the code makes use of four Object-Oriented design patterns.

An Abstract Factory is used to create the different scenes of the application. This helps in instancing scenes and gathers creation logic in one place.

In two cases the Template Method pattern is implemented to reduce code reuse. When to scenes have the exact same buttons and fields, but different actions are needed when buttons are pressed, the Template Method allows one View class to be able to hold different types of controllers, changing the behaviour of the Scene depending on the controller that was given to the View class at start.

To change behaviour of Bookings and Listings a State pattern is used. Different states contain different information on what text should be displayed and how Bookings and Listings should interact. The State pattern makes the process of completing a booking very expandable, since there is no need to change any code in the Booking/Listing classes. Instead, one can simply create more states and connect them where needed. Note: due to issues with serializing interfaces in the GSON library, the State pattern currently revolves around a concrete class, which the states

extends. This is definitely not the best solution, and the concrete class should be changed to an interface if possible.

Singleton pattern is used in various parts of the application in classes where we only want one instance to be created. This is because the various handlers is handling database entries in JSON. If multiple instances of handlers where to be created it would be nearly impossible to display all entries that should exist in run-time.

4 Persistent Data Management

4.1 User Data

The application uses a database to store information about users and their products. This functionality is the responsibility of the JSON package which consists of the classes JSONWriter and JSONReader. They both use Gson [1], a serialization/deserialization library, to convert Java Objects to and from the JSON file format. The .json files are stored in the resource folder JSONFiles and are written and read by the java.io Writer and Reader respectively.

4.2 Images

All images are stored in a folder within the application where they can be directly accessed and loaded into JavaFX. All images are handled using a utility class called ImageHandler.

ImageHandler is a singleton, that is in charge of loading and fetching images. Whenever a part of the application needs to display an image, the ImageHandler first checks if the image has previously been loaded. If it has it then simply fetches the image from the ”cache” and displays it with JavaFX. If the images has not previously been loaded it loads the images by matching the name of the file with correspondent one in the Images folder. After it has been loaded it also places the image in the HashMap to avoid loading the same images multiple times during runtime.

5 Quality

5.1 Tests and Issues

We are using JUnit to write unit tests for our model classes. The tests are located in /src/test/java.

We use Travis CI for continuous integration. `https://app.travis-ci.com/github/emillindblad/17E-OOP-Project`

5.2 Analytics

5.2.1 Project Dependencies

Compile			
GroupId	ArtifactId	Version	Type
com.google.code.gson	gson	2.8.8	jar
org.openjfx	javafx-controls	16	jar
org.openjfx	javafx-fxml	16	jar
Test			
junit	junit	4.11	jar

5.2.2 Code Dependencies

Code - JDepend		
Class/Package	Used by	Uses
Model.UserPackage	Model.Booking, Model.ListingLinker, Controllers, View	None
Model.Listing	Controllers, Model.Booking, View	ListingLinker
Model.Booking	Controllers	Model.Listing, Model.UserPackage
Model.InputChecker	Controllers	Model.Listing
Model.ListingLinker	Model.Booking, View	Model.UserPackage

5.2.3 Quality - PMD

No major discrepancies between our code and best practices were found. Minor design flaws were flagged that were either insignificant or simply incorrect assessments by PMD. Documentation could be improved in Controllers and View.

Many JUnit assertions could be improved by including a message and simplified by changing assertion type.

5.3 Access Control and Security

To ensure that no one except the logged-in user can access their information everything passes through a *Handler*-class in the *User Package* where no information is accessible unless a successful login has been made. This is implemented so that the program only shows the logged in users information and all other information that are connected to users are handled internally in the user package except for user ID-markers that enables us to connect a listing to a user by only showing the client code a string and not the entire User object.

Furthermore the *handler* is structured in a way that there are no public methods to access the complete list of users and the only information that will be visible from the client side is that of the logged-in user and internal identification markers the client will use when sending calls to the *handler* when an interaction with another user is made.

The *getLoggedInUser*-method that enables access to the users data are in the current implementation only called by the client code when the program is in a state where a user has left the log-in screen and are therefor already logged-in. If the usage of this method would be used somewhere else where it is not intended to (when no user is logged in) it would create a ”NullPointer”-exception and close the application. In the current development state of the program this is acceptable because if the client code would call this method without a logged-in user that signals that something has gone wrong and the application is in a state where we have not intended it to be.

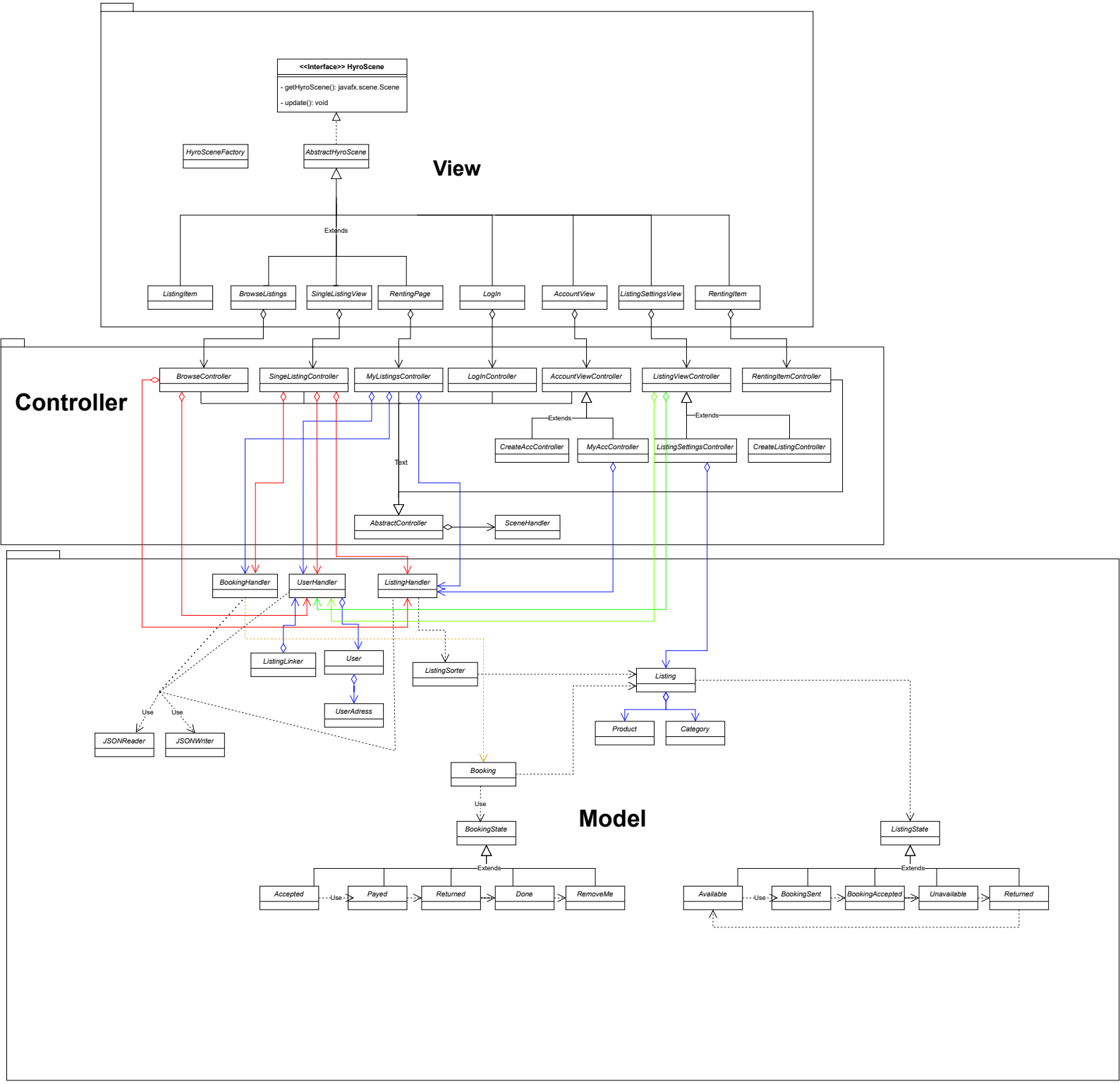
The *Listing Package* has a similar structure as the *User package* centered around a *handler* that holds the listing data. This is to control what listings to be show at what time and to who. Both the User and Listing handler are constructed as Singletons to ensure that all data objects end up in the same handler. This could be considered a problem because these classes drift towards becoming *god-classes* but with no real database this problem could be solved differently to avoid this.

Finally, a known security issue is that user’s passwords are stored as unencrypted plain text in the users.json file

References

[1] URL: https://github.com/google/gson.

A Appendix


<<Interface>> HyroScene
- getHyroScene(): javafx.scene.Scene
- update(): void
HyroSceneFactory
AbstractHyroScene
View
Extends
ListingItem
BrowseListings
SingleListingView
RentingPage
LogIn
AccountView
ListingSettingsView
RentingItem
Controller
BrowseController
SingeListingController
MyListingsController
LogInController
AccountViewController
ListingViewController
RentingItemController
Extends
Extends
CreateAccController
MyAccController
ListingSettingsController
CreateListingController
Text
AbstractController
SceneHandler
BookingHandler
UserHandler
ListingHandler
ListingLinker
User
ListingSorter
Listing
UserAdress
Use
Use
JSONReader
JSONWriter
Product
Category
Booking
Use
Model
BookingState
ListingState
Extends
Extends
Accepted
Use
Payed
Returned
Done
RemoveMe
Available
Use
BookingSent
BookingAccepted
Unavailable
Returned