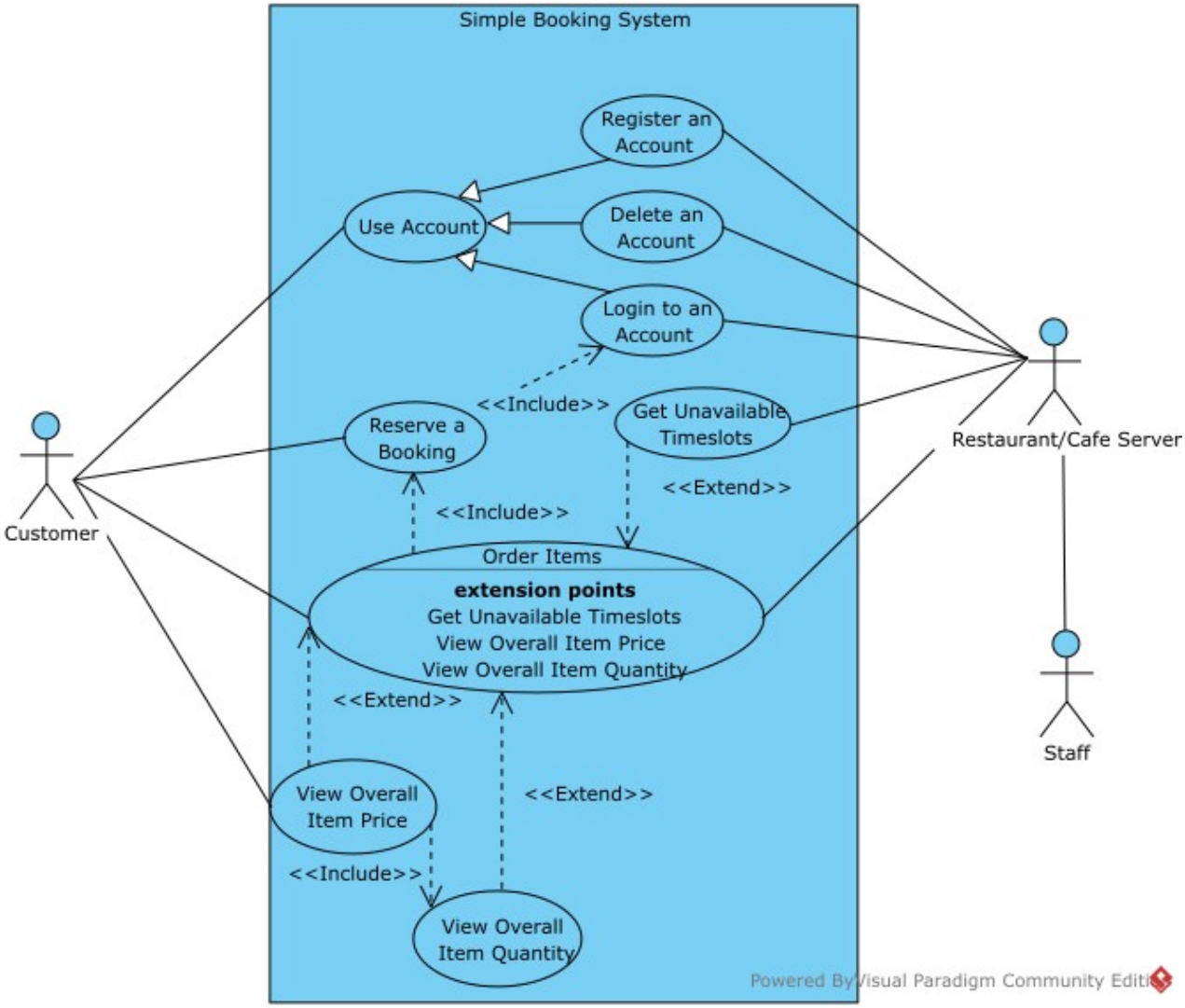
Simple Booking System – Design Documents

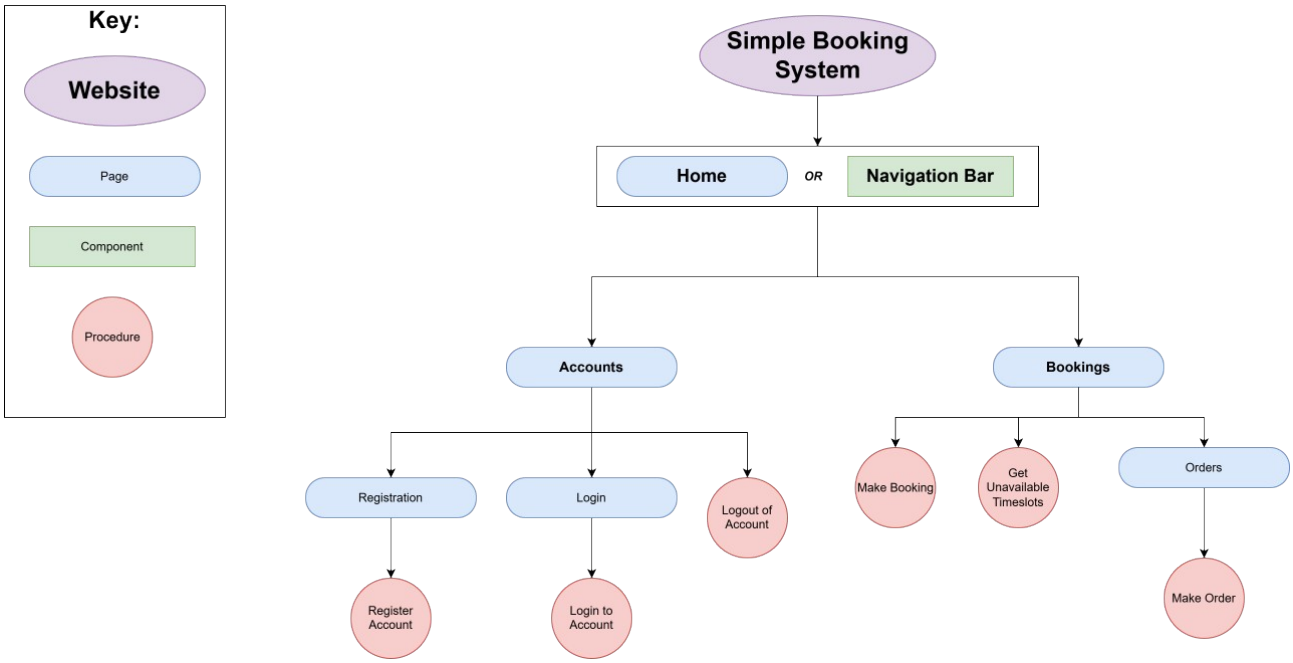
Front-end

Use Case Diagram



This is a use case diagram which showcases the various functional requirements of the system from the business proposal and brief in the form of features. The key feature is that it shows the relationship between features and breaks them down into chunks that can be modelled individually. These will form a basis for the rest of these design documents.

Sitemap



This is a sitemap that shows how the user will access the various functionality seen in the Use Case Diagram. It shows the journey the user will take to accomplish certain actions. In a development context, the page hierarchy shown here will be used with some kind of page routing system, such as those seen in web frameworks. In the PHP framework Laravel, this would be using the Route facade to respond to POST and GET requests. In this instance, the key represents the following:

- A “Page” represents a response to a GET request
- A “Procedure” represents a response to a POST request, such as a form, or an AJAX GET request
    - (However, the “Get Unavailable Timeslots” procedure will be an AJAX script that responds to a GET request)
- A “Component” is a reusable UI component that is available across every page

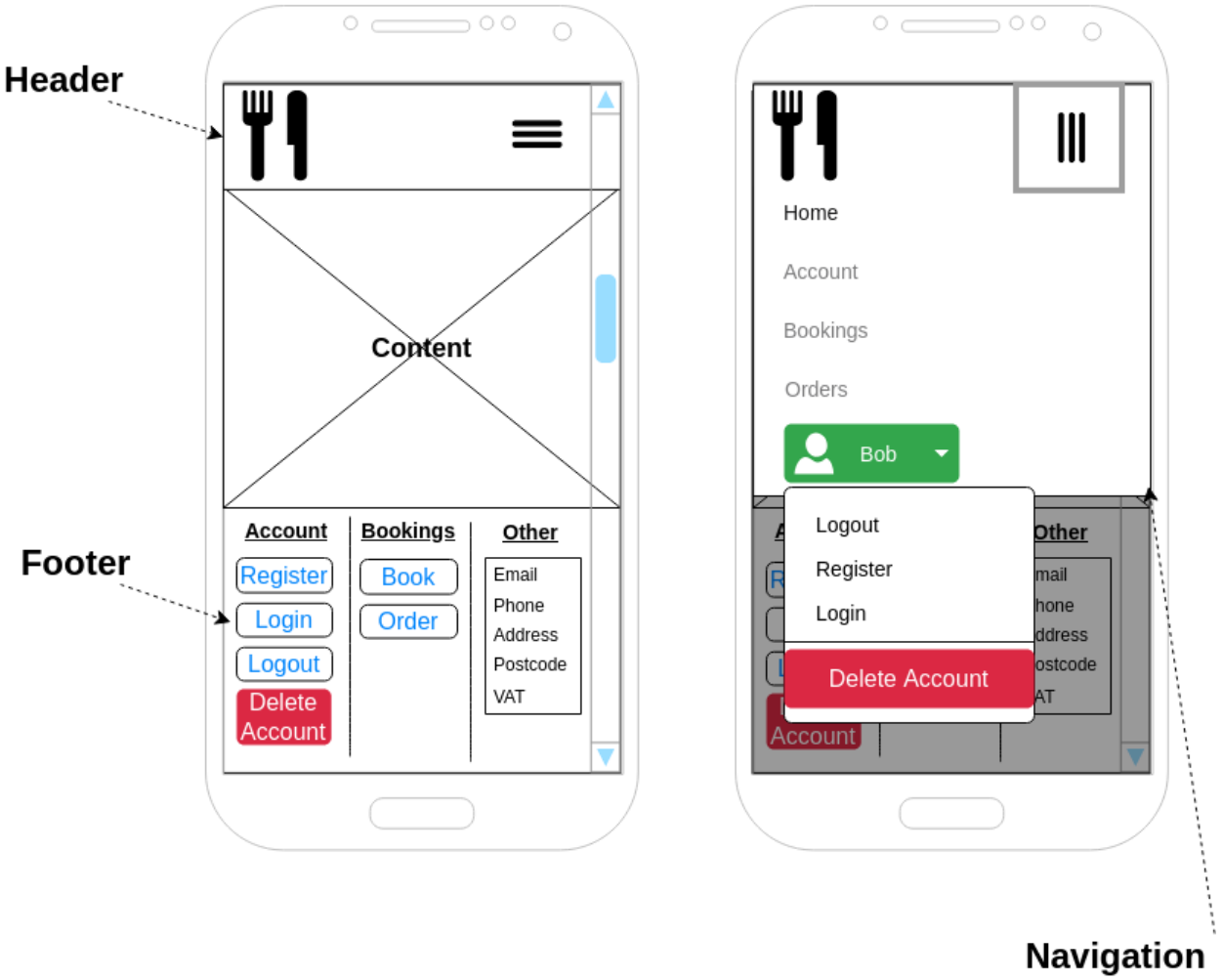
Wireframes

Overview

Wireframes will be used to sketch out what the user interface may look like for the system. In this case, placeholders will be marked with 2 diagonal lines intersecting to form a cross (these could be image assets or other components that are yet to be sketched out. Interactive elements like buttons and links will usually have an outline with rounded corners to signify interactivity. Bootstrap-style colours (e.g. success, danger, warning, info, primary, secondary) will be used to convey extra meaning, improving the user experience. This is because the implementation will likely use a front-end framework/toolkit like Bootstrap.

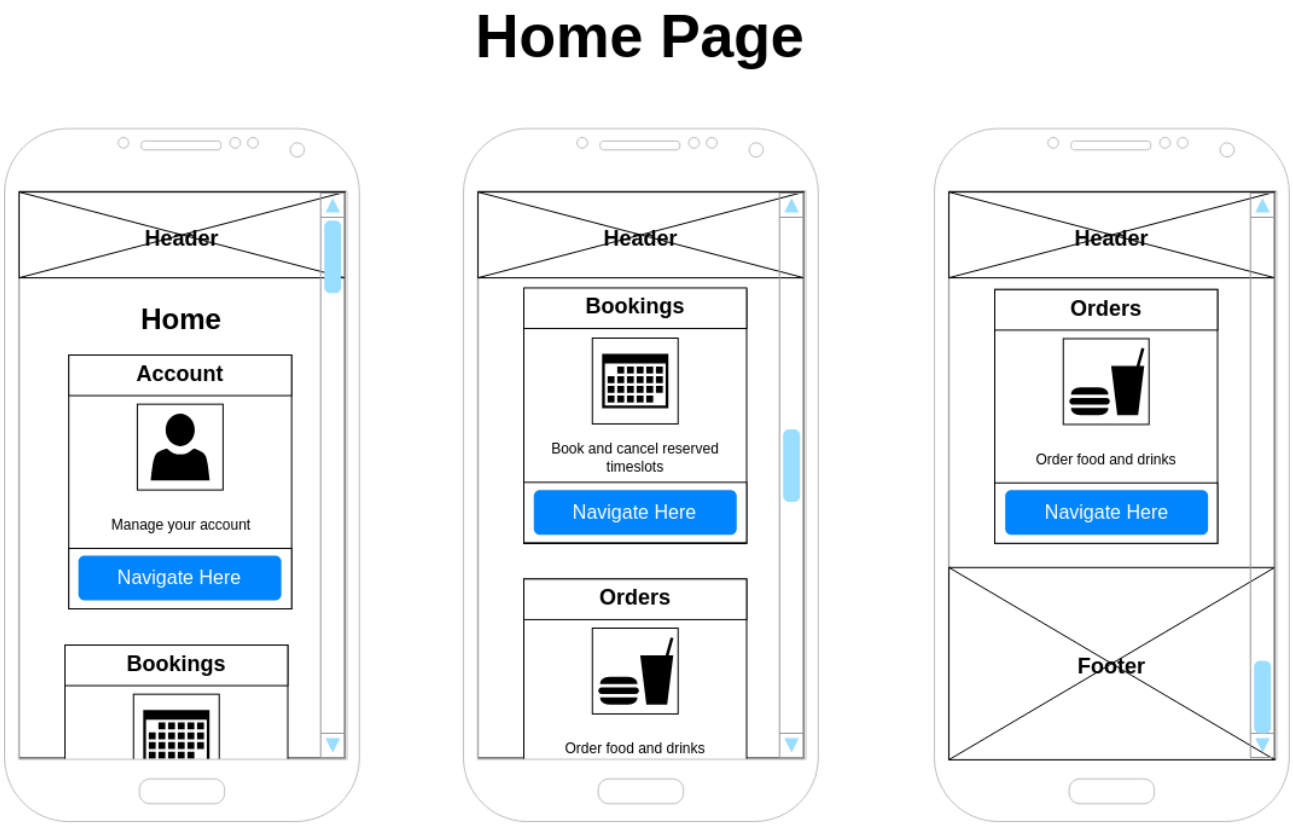
Components / Base

Components / Base



This is a wireframe that shows the persistent components that will be developed. A few things to note here is that the brand/logo button (denoted by a cutlery icon) will conventionally allow the user to navigate back to home easily. In addition to this, the various navigation items (Home, Account, etc.) will take the user to the various different pages (see sitemap from earlier) containing functionality to fulfill the different use cases (identified in the use case diagram). To denote that the user is on that page, the text will be a solid black, but will be grayed out if that page is inactive (this is a standard means of communicating such information), which is a feature that will have to be programmed in to the view, potentially using a templating engine such as Blade. Another feature is the ability to see the logged in account. In the actual implementation the button could be a grayed-out outline if the user is not logged in, or it could not be shown at all. The accounts navigation item should take the user to whatever page is relevant to them, dependending on if they are logged in or have created an account in that session. A standard hamburger menu icon is used, denoted by the three lines, which should appear if the user is on mobile, but on desktop they will see the navigation items strewn across the header without such a menu. The header will be sticky and persistent (as seen in the wireframes ahead) so that when the user scrolls down, they will still be able to navigate no matter what page they are on, improving the user experience.

Home



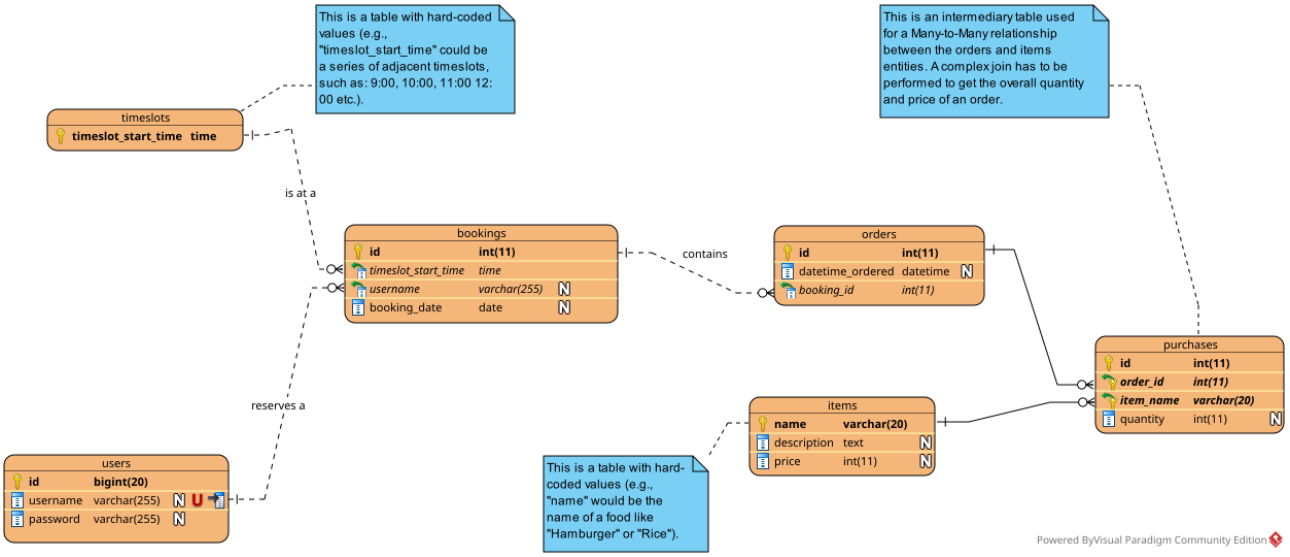
The home page contains responsive cards that the user can navigate to. Icons are used to convey extra meaning, making the solution more user-friendly, and a blue (primary) colour is used to communicate that the button is to navigate elsewhere. A description of the page is shown in the card container to help the user understand the purpose of the page and why they would want to navigate to it. These cards resemble the common front-end design pattern of using cards, and could be implemented using a front-end framework like Bootstrap. The cards stack on top of each other in the vertical direction, but on a larger landscape screen, some of the them could be placed on the same row to make better use of the screen space available.

Back-end

Overview

The solution uses the Model View Controller (MVC) architecture to organise functionality and logic across the application. Views will be based on the front-end pages identified by the sitemap and wireframes as these will be what is rendered on the client. Controllers will be based upon the views and forms in those views (identified as procedures in the sitemap). Models will be based on real-world business objects and concepts in the business domain. In order to take advantage of MVC, Object-Oriented Programming (OOP) will be the easiest methodology to implement the designs in.

Entity Relationship Diagram (ERD)

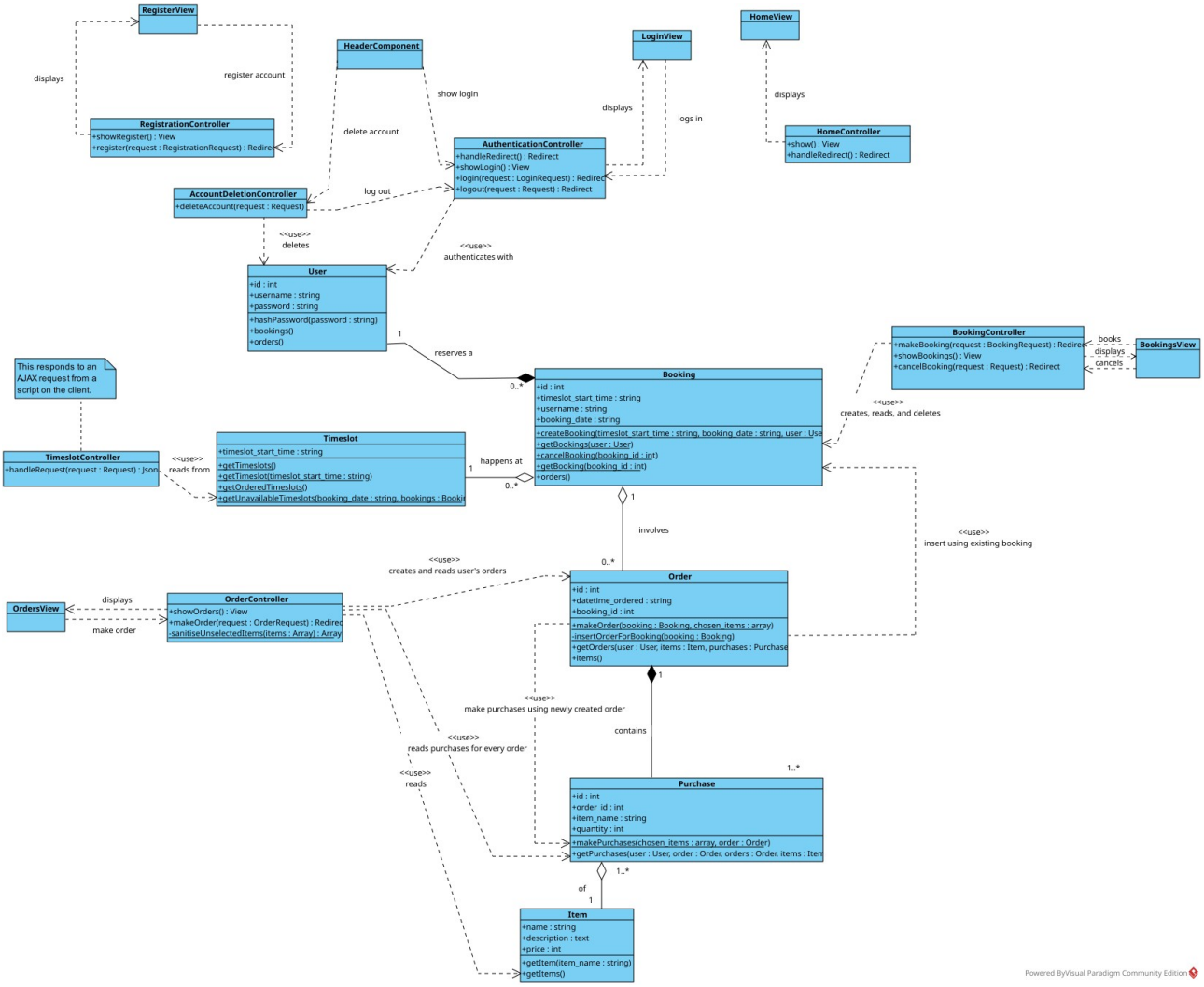


As identified in the use case diagram, certain functionality interacts with other functionality within the system. To demonstrate this, relationships in an ERD are used, as some features will Create, Read, Update, and Delete (CRUD) certain entities in the database, which will have an impact on the CRUD operations of other features. For example, the user has to reserve a booking, but they will use the booking they have reserved to order items. However, they can only order items when the current time is in the timeslot they have reserved. The overall price and quantity should be calculated and shown to the user. The only way to achieve this efficiently is with a data schema utilising relationships between these various entities. This allows easy querying of, for instance, the orders of a particular user, with bookings being the intermediary table.

The ERD represents a static view of the database schema used to implement the database. The relationships could be implemented using foreign key constraints in a data table engine like InnoDB.

See the notes attached to tables for more information about intermediary tables and hardcoded values.

Class Diagram



A class diagram has been used here to represent the format of entities as Models, and their interaction with Views and Controllers. It is worth noting that the “View” classes here will most likely not be classes when implemented, but more dynamic templates provided by a templating system like Blade in a server-side framework such as Laravel. The controllers receive requests and handle them. The relationships between entities, as well as model lifetimes are shown here.

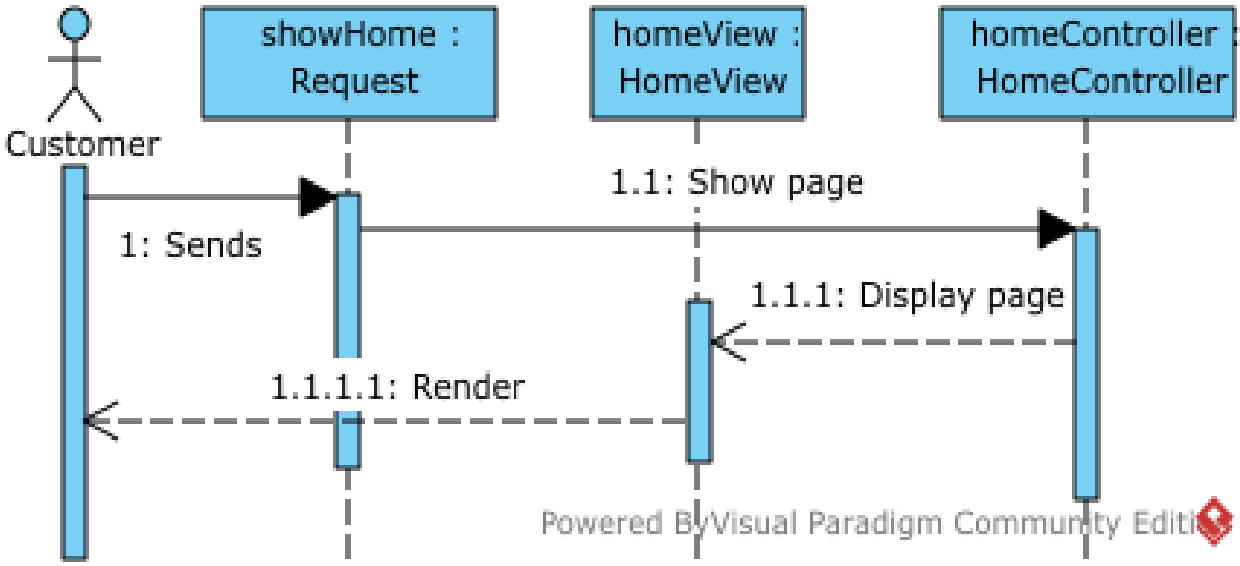
What is **not** shown here in detail is dependency injection. The dependencies are represented by dotted lines with arrow ends. Dependencies will be injected into the controller so they can be used to respond to a request. Dependencies include instantiated models, like the currently logged in user, or the bookings and orders in the database. Some models (like bookings and orders) will be filtered using static helper methods attached to the model. This is to abstract database-specific and row-level details about database and ORM operations from the controller using that helper method (the function user). This will also allow easy error handling with try, catch clauses on the helper method.

Dependency injection will also make it easier to unit test the code.

The makeOrder method is an interesting method as it performs two insert queries. This is shown in the order items sequence diagram later on in this document.

Sequence Diagrams

Home



This is a basic sequence diagram showing the process of displaying the home page to the user. The request is sent by the actor (the user of the customer-facing side of the system, or customer, in this instance). This request is received by the server and the server routes the request to the home controller. The home controller receives this and then renders the view, which is then displayed back the user as a response to the request.

Accounts

Overview

The accounts module can be split into roughly two modules handling CRUD and authentication actions:

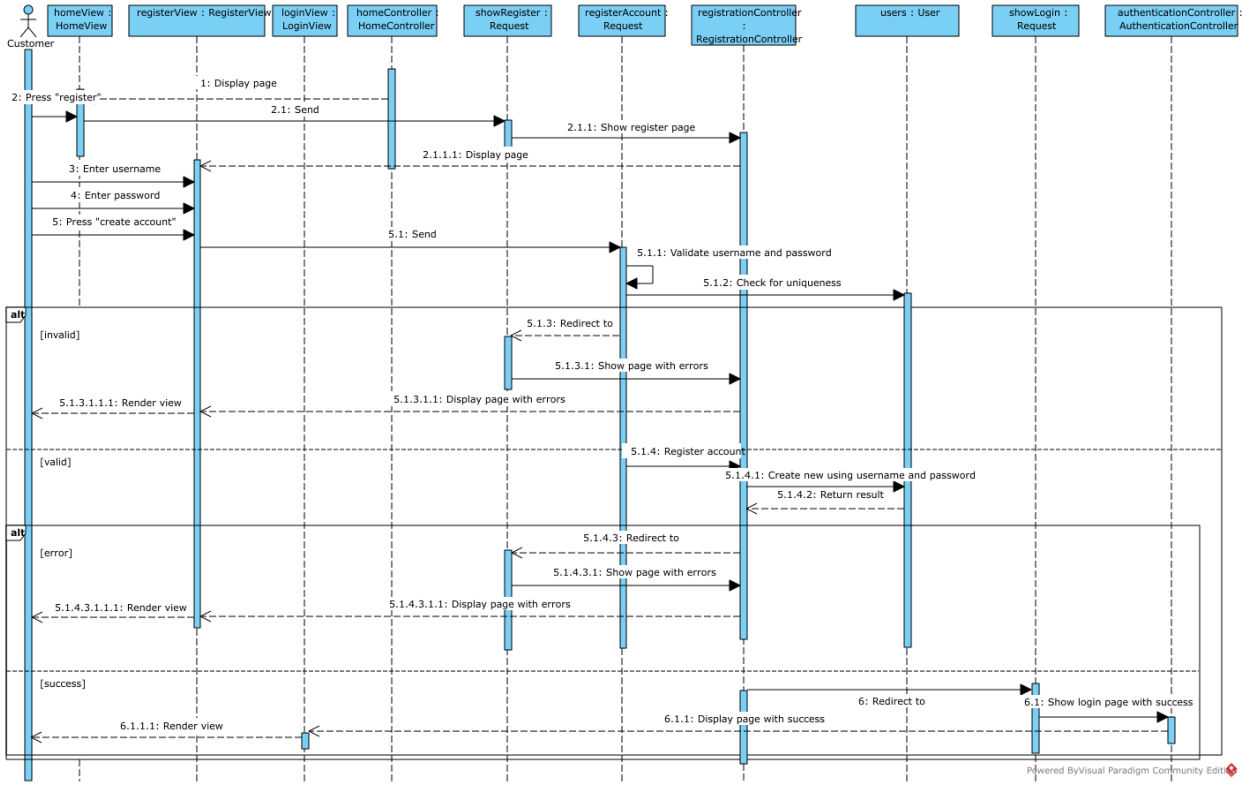
- The writing to the User model from the database, handled by the RegistrationController and AccountDeletionController
- The read operations such as logging in and logging out the user into/from their account, handled by the AuthenticationController

The AccountDeletionController also has a side-effect of logging out the user as their session is no longer valid. The read operations read from the session, but the write operations may also read for validation purposes (e.g., checking if the username of the user to be created is unique).

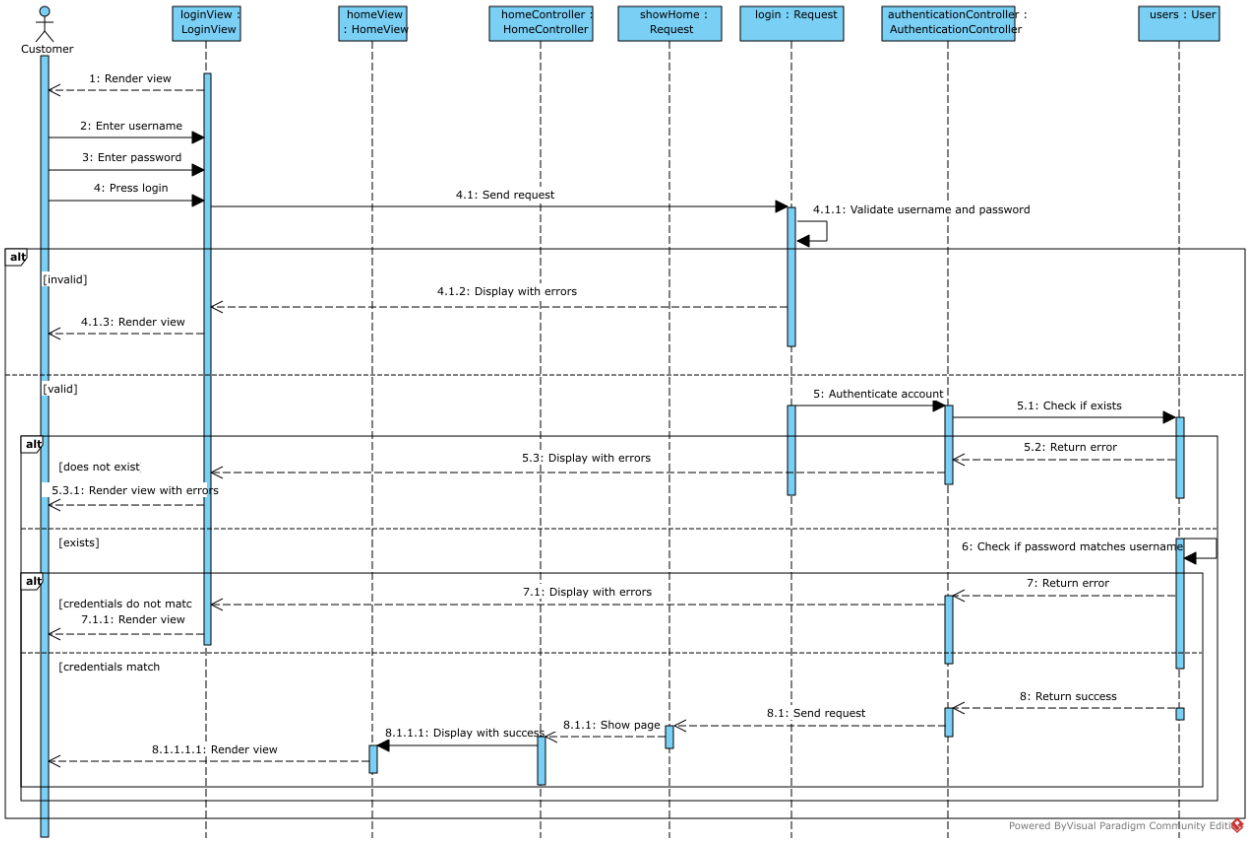
Validation details will be shown later on in the algorithm flowcharts.

As seen by the instantiation of User as users, the User model is injected as a dependency of the account system. The handling of the session will very much depend on the implementation language and framework. In vanilla PHP, this would be done through the $_SESSION superglobal, but in Laravel this would be handled through the Auth facade.

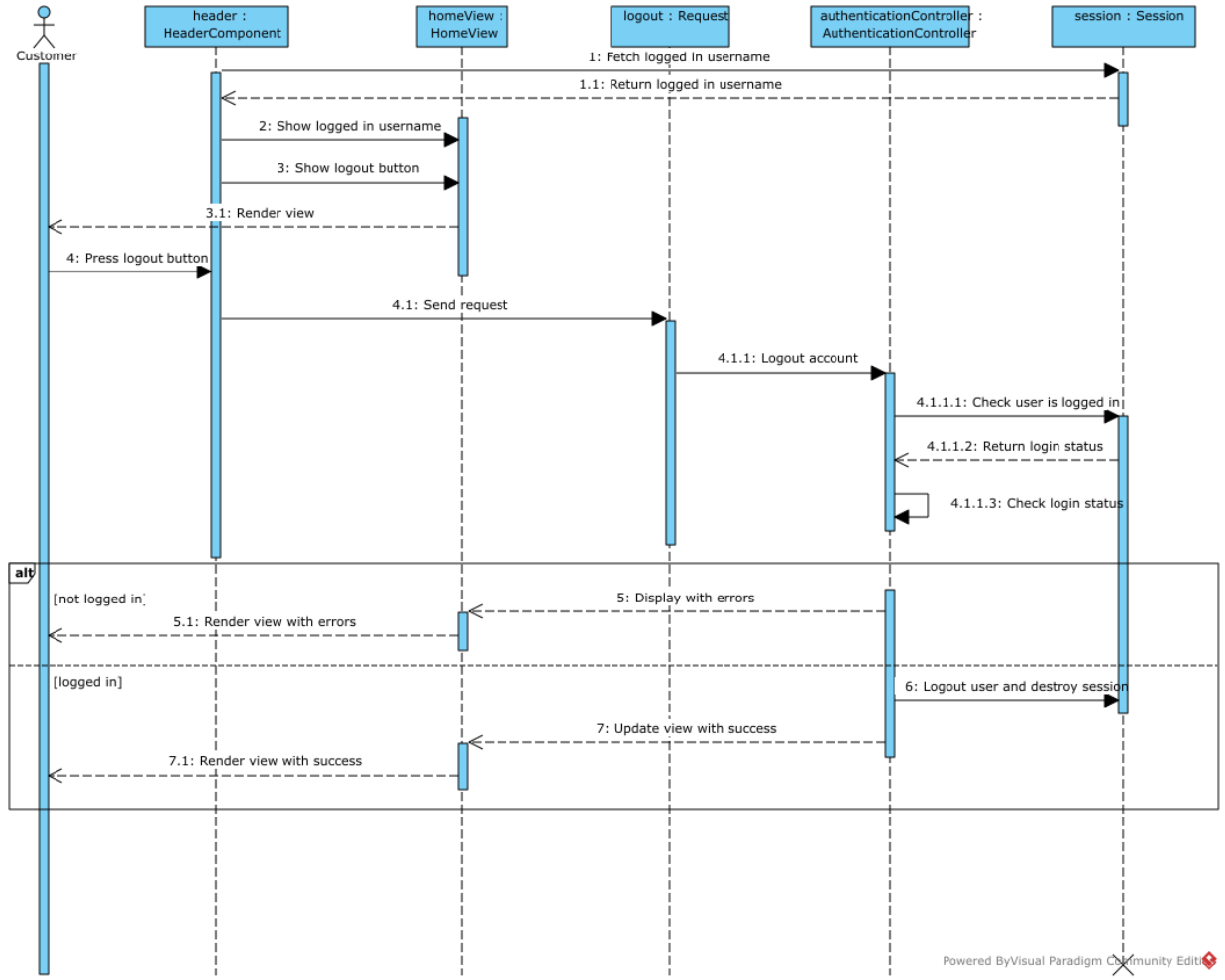
Register



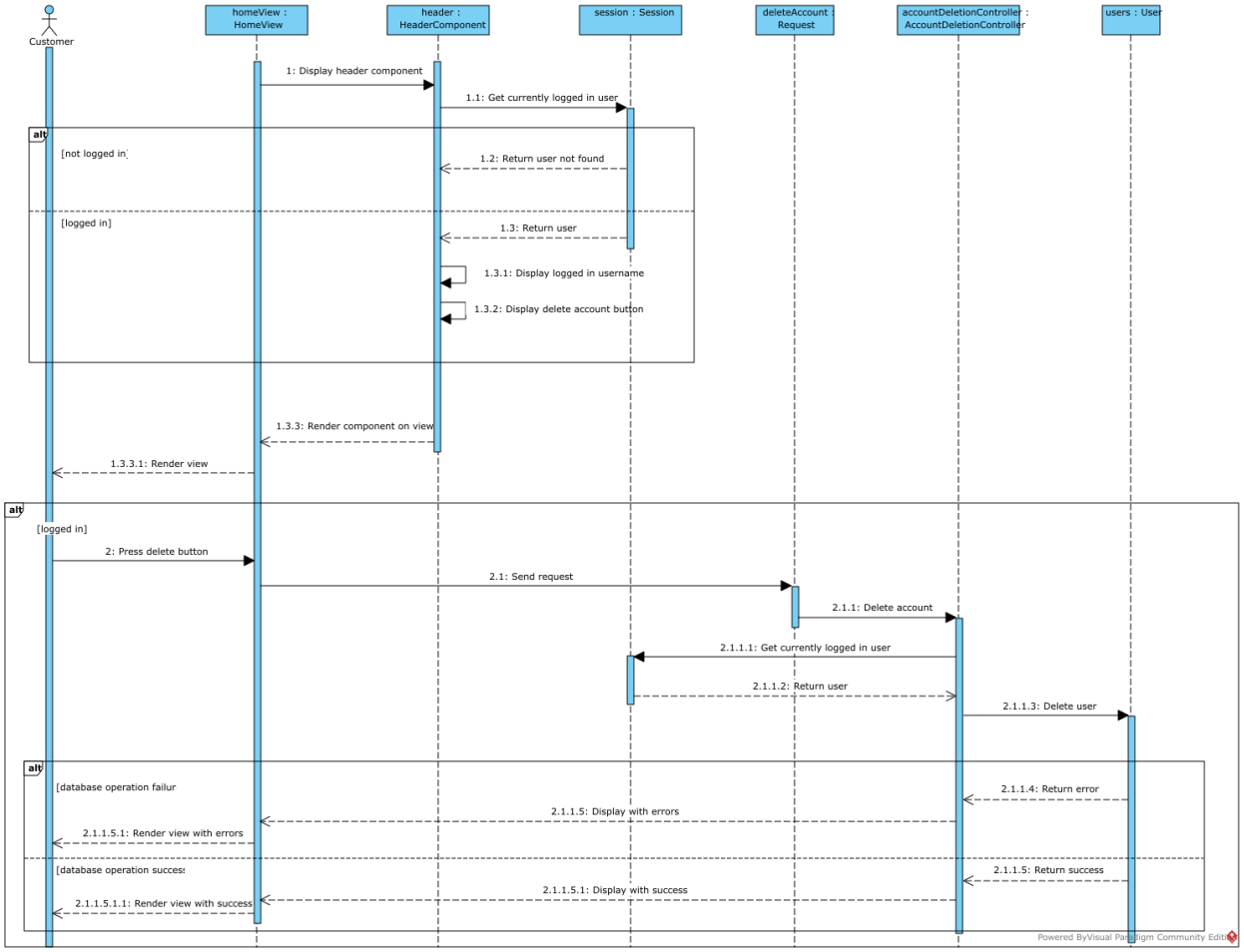
Login



Logout

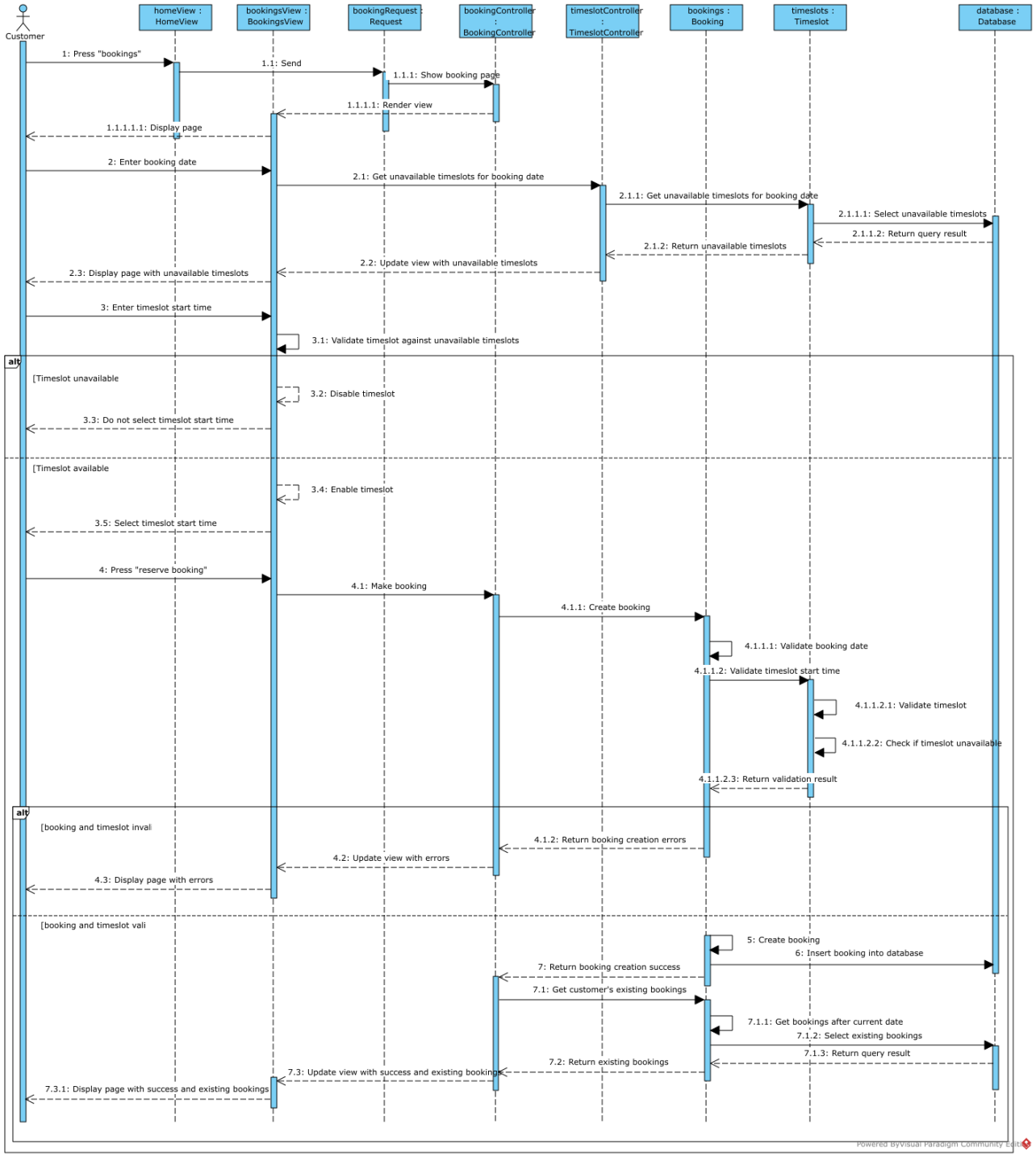

Customer
header : HeaderComponent
homeView : HomeView
logout : Request
authenticationController : AuthenticationController
session : Session
1: Fetch logged in username
1.1: Return logged in username
2: Show logged in username
3: Show logout button
3.1: Render view
4: Press logout button
4.1: Send request
4.1.1: Logout account
4.1.1.1: Check user is logged in
4.1.1.2: Return login status
4.1.1.3: Check login status
alt
[not logged in]
5: Display with errors
5.1: Render view with errors
[logged in]
6: Logout user and destroy session
7: Update view with success
7.1: Render view with success
Powered ByVisual Paradigm Community Editi

Account Deletion



Orders

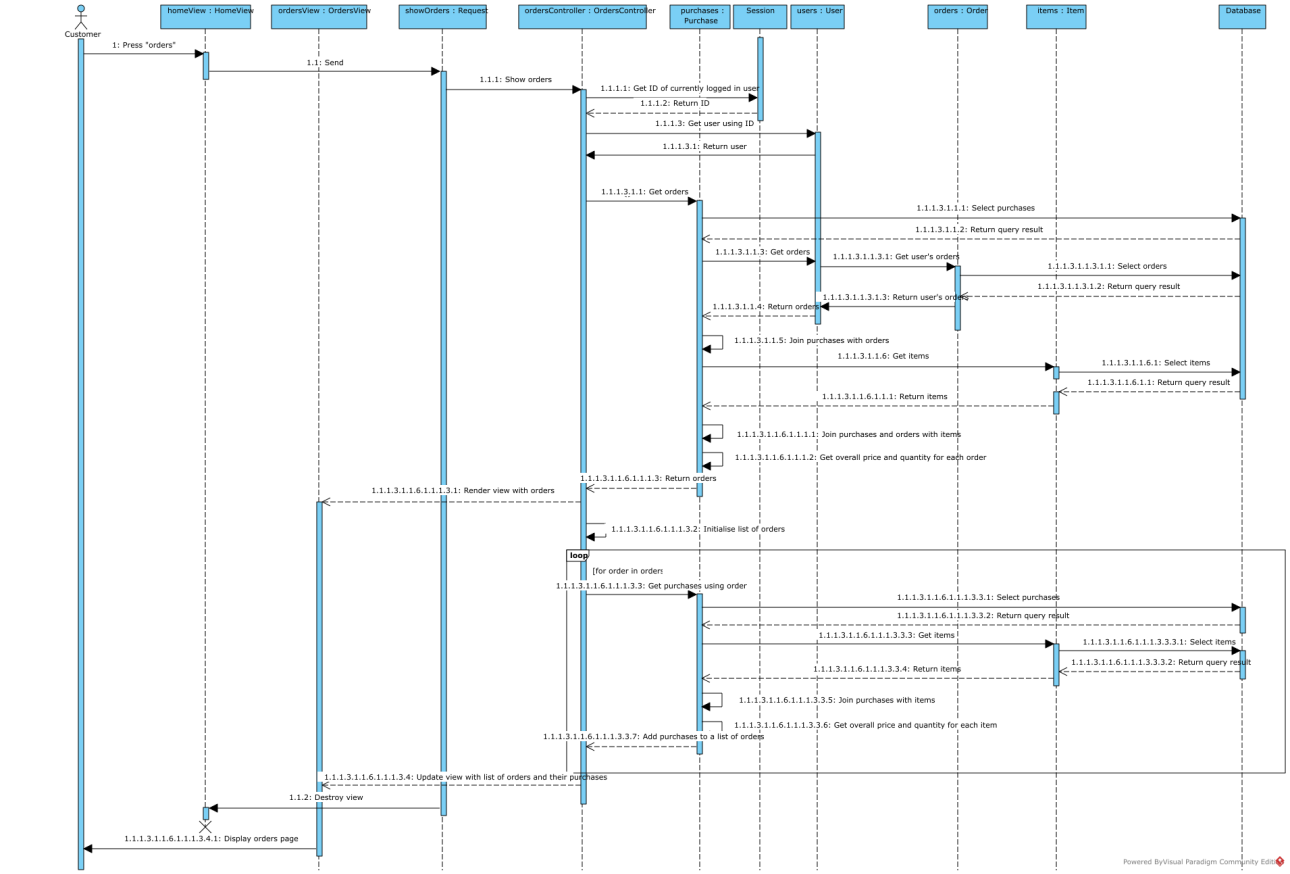
Book a Timeslot



Dynamic updating of the view through JavaScript's AJAX is not an easy concept to show in a sequence diagram, so I have shown it from the point of view of the user when they trigger the event that runs a script on the web server. This will be validated on the client-side (perhaps through HTML's "disabled" attribute) and on the server-side (through AJAX using a client-side script and corresponding server-side script that provides a response).

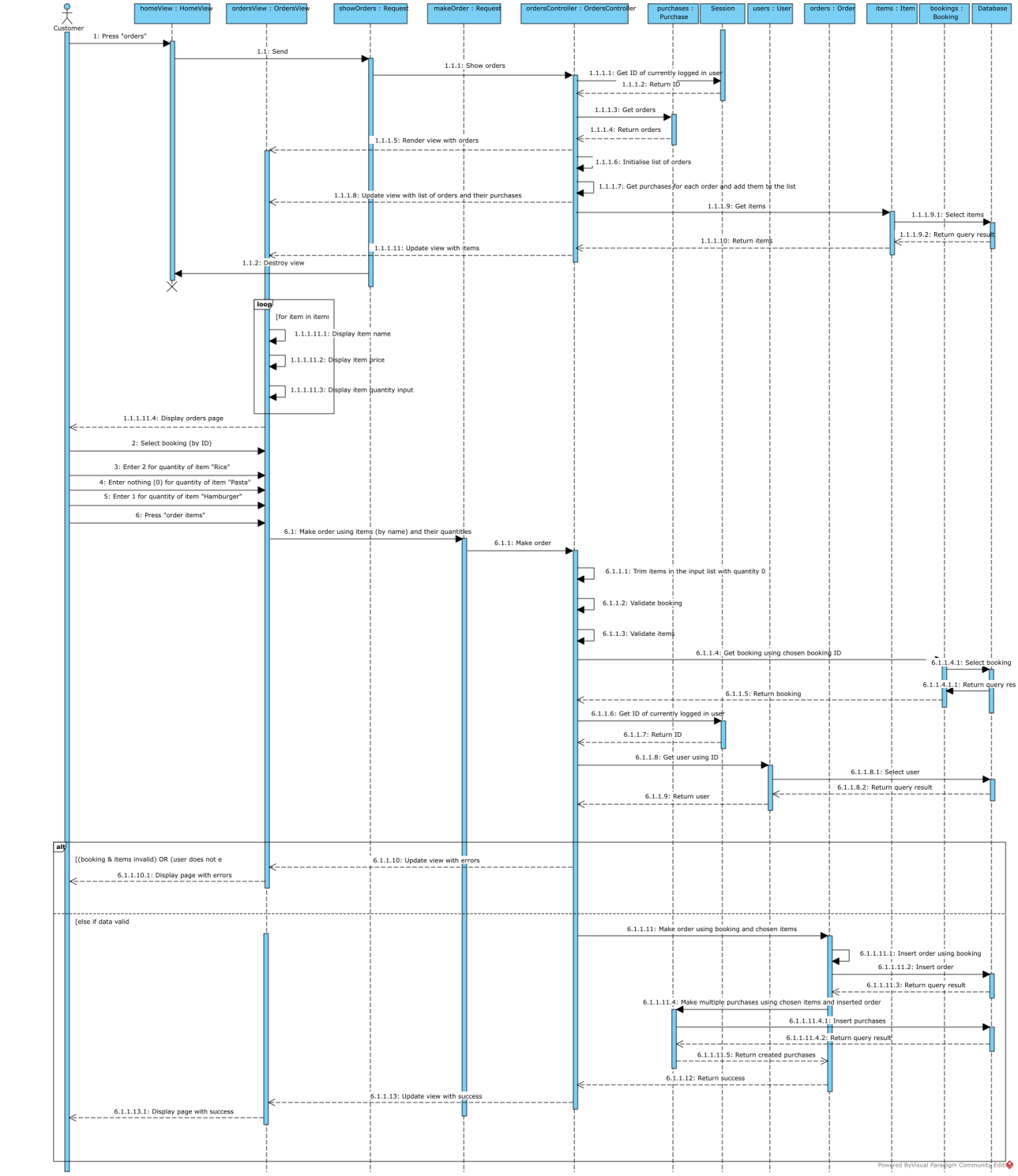
There will be many consistency checks (e.g., making past booking dates invalid) and look up checks, but these are deliberately not shown on the sequence diagram (it is supposed to be abstract) to keep it concise. These are shown abstractly through self-messages that start and end in the same object instance. They will be represented in detail in their corresponding algorithm flowcharts.

View Orders and Purchases



Various queries are performed here to infer information from the database that is not available as raw data. This includes getting the total price of a single order. This would be done by joining all purchases associated with an order to get the quantities of different items, and then joining those items by their names in order to get price of each item. Then an aggregate function like SUM is used to get the total price or quantity of an order. It is also possible to get the total price and quantity of a purchase, or the price, total price, and quanity of each item in a purchase. SQL Inner Joins may have to be used in order to obtain this information, and these could be implemented using PDO Prepared Statements or using Laravel's Query Builder to create SQL statements. In addition to this, Laravel's Object-Relational Mapper (ORM), Eloquent, could be used to obtain models that are related to each other in a single code statement.

Order Items



Ordering items involves a form with inputs capable of sending an array to the server (a feature of HTML forms using the name attribute). These are then indexed by a server-side script. A potential implementation could be an associative array indexed by the item names as keys, and the chosen item quantities as values. In this design, a private santiser method (sanitiseUnselectedItems) that trims the items from the array that have a quantity of 0 (as these will be sent to the server even though they are not actually selected by the user). This will be a private method in the controller because it is a feature of the controller that should be encapsulated privately. Controllers are supposed to be a pattern that handles user input, and the method here santisises the user's input.

The ordering of an item involves two insert queries. One of them inserts an order into the orders table. The other uses the ID of this order and items provided by the controller received by the user to make purchases in the purchases table by inserting multiple rows (a feature of SQL).

Algorithm Flowcharts

Test Strategy