Milestone 2

SENG350 Software Architecture - Group 3-6

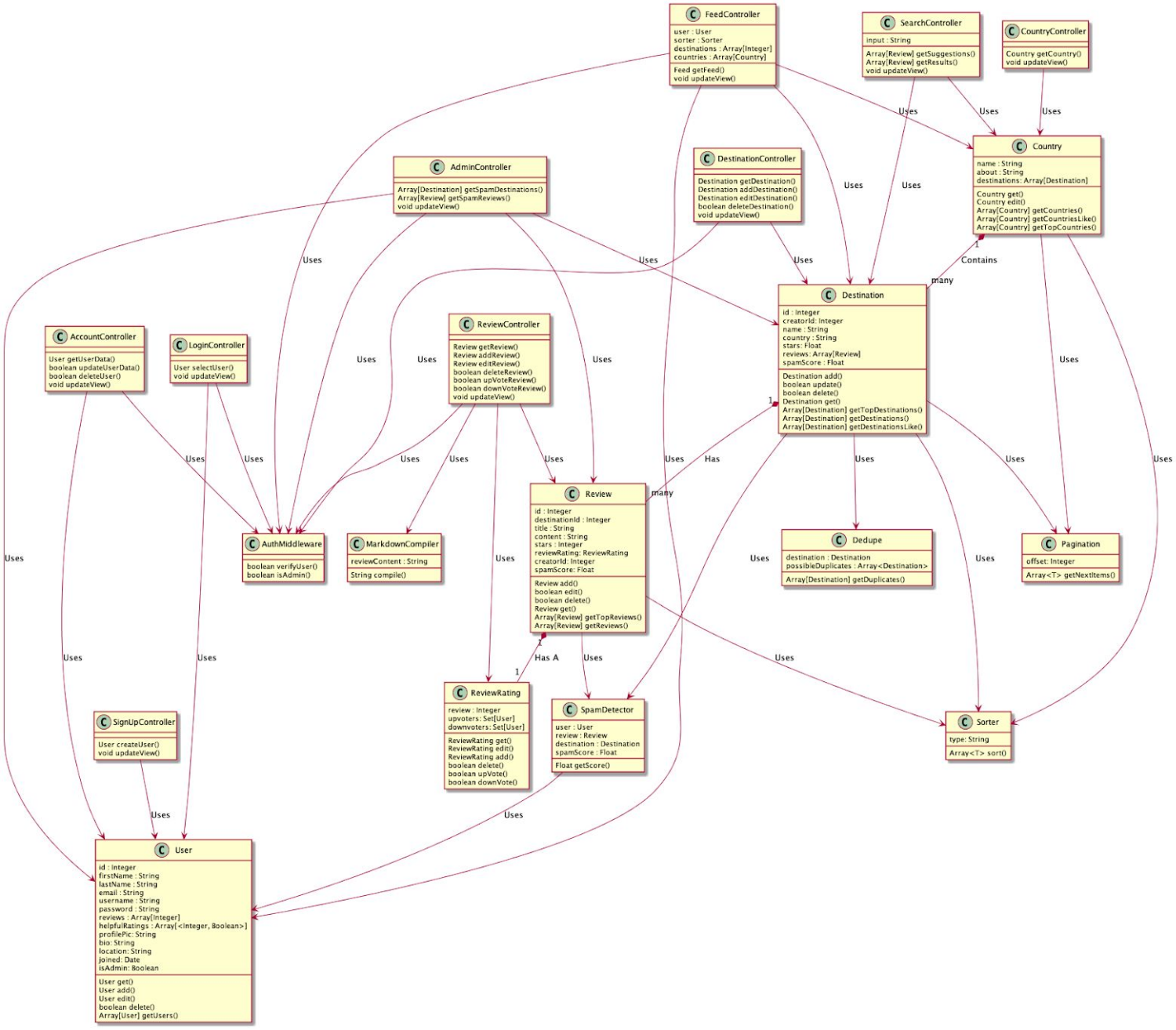
Product Description

The project our team has chosen to build is a web application that offers users that would like to travel to a certain country a list of destinations to browse that have been rated and reviewed from other users. The reviews of these destinations such as cities, national parks, etc. provide users with authentic descriptions of experiences from travellers that can include photographs and perhaps shared secrets that would not necessarily be found with a quick online search.

Destinations will also be able to be rated with the classic thumbs up, thumbs down approach. As the user base expands, the collected data will improve the user experience in two ways. The first being that once a user selects a country that they wish to visit, the highest rated destinations will be shown first (this will be based on the 5 star system). The second being that as more reviews are published, a general consensus will be able to form and will increase credibility of not only the website, but users as well.

The long term plan (past the scope of this course) would be to create a community of travellers that can share their experiences abroad, keep a timeline of their travels through reviews. At a later stage, a feed might be introduced, creating plenty of opportunities for the application and users, including the ability to follow each other, more personal investment, etc. and a by-product of this is more opportunity to monetize the product and scale it.

Class Diagram

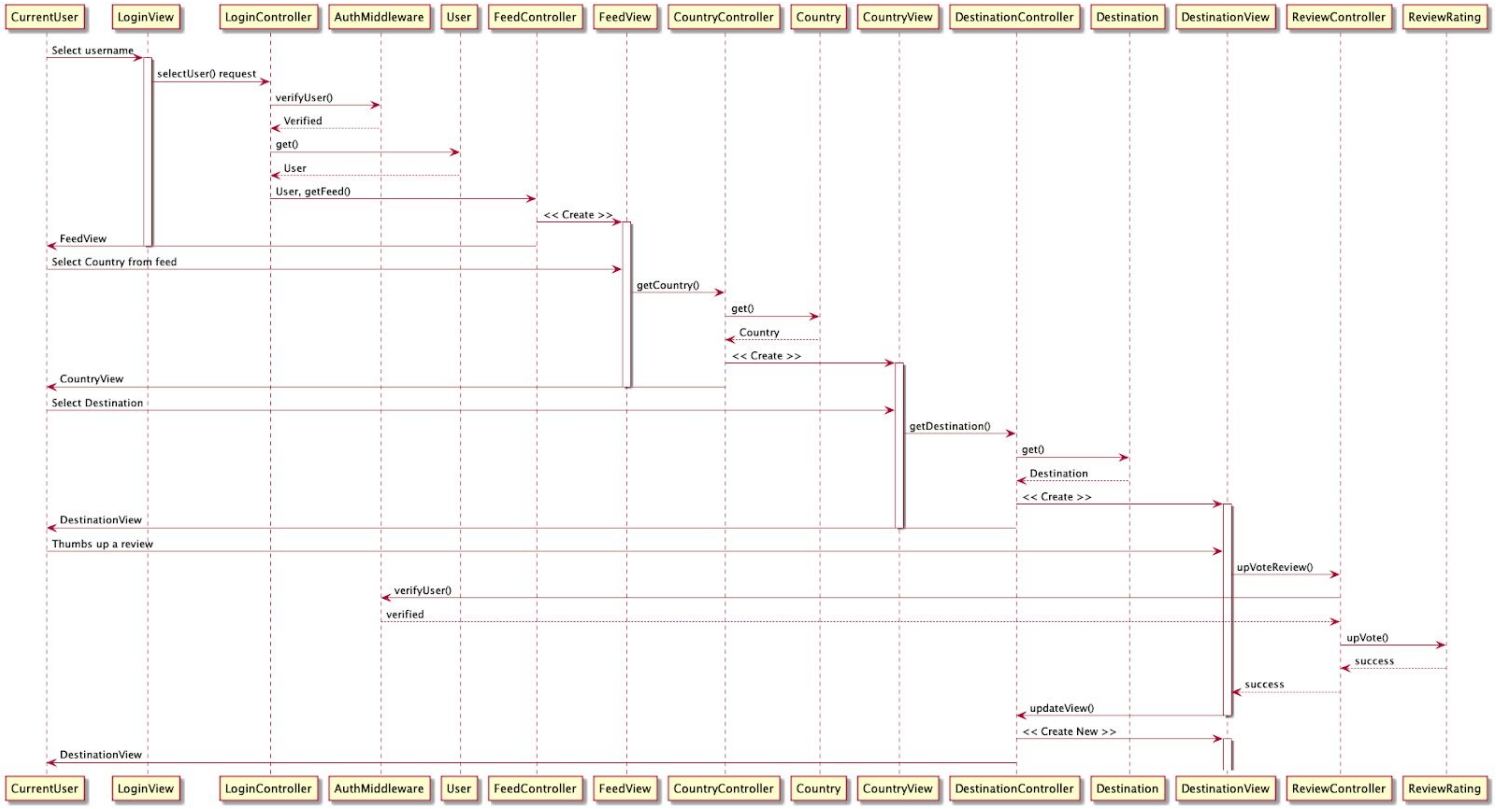


Context: Each controller has an associated view which the controller can render. Each view is able to call methods on controllers based on what is presented on a view. (ie. On the Country

view, there are links to destinations which in turn are hitting the API and calling functions on the DestinationController). A view is also able to call a function on the controller to update the view. These are not present in the diagram as they all serve the same purpose and would significantly clutter the diagram. The view code is run in the browser and is included in the html page rendered by the controller.

Additionally, the system components are coupled in a hierarchical way such that higher level classes call methods on lower level classes but not vice versa. This is done in a way such that classes can be tested independently through data mocking and minimal function mocking (basic functions that return data structures).

Sequence Diagram



Context: In the above diagram, User, Country, and Destination are all model components that interact with the database. CurrentUser is the user interacting with the system. The lifeline of views is denoted with the wide bar on the timeline. Views are destroyed upon a new view being displayed to the user.

Architecture Decision Records

Updating the view

- *Status:* Proposed
- *Context:* The model needs to be updated when data in the system changes.
- *Decision:* The controller will create and update the view. When the view makes a request to the controller to perform some operation, the controller will update the model and upon success of updating the model, it will recreate the view. This allows for easier testing and light coupling of the components. Only the controller is dependent on other classes.
- *Consequences:* The controller will be responsible for a significant portion of the logic, but in turn, the model will simply be responsible for updating data in the database and returning the newly updated data back to the controller for rendering the page. The current state of the page will not be maintained, so entire pages will need to be re-rendered, which can be a bit slow.

Rejected: The traditional approach of having the model interact with the view directly and keeping track of what is currently displayed. We feel the stateless approach will be easier to manage.

Signing in and performing all system operations

- *Status*: accepted
- *Context*: To verify users, we need to consider how to do the simple auth for operations that are attached to a specific user.
- *Decision*: The user will select their account from a list of current users and the login controller will verify the user exists and set a query string to use for auth in the future. All operations that need auth will have auth middleware attached, verify the auth query string, and assert to the controller the user that performed the operation.

- *Consequences*: Auth is stateless from the server side, so every time the user needs to be verified. This can be slower than other approaches such as stateful but are easier to manage.

Rejected: Using a stateful authentication approach

Component Coupling for Testability

- *Status: Proposed*
- *Context:* Components need to interact with each other, but we want them to be testable independent of one another.
- *Decision:* Controller components are the only components that will be tightly coupled with their models. Model components will not be coupled with their controllers and will have few couplings with simple components (sorters). This allows for the controller components to be tested using mock data and simple function mocks, and allows other components to be tested directly or again with basic function mocking.
- *Consequences:* Since the components are loosely coupled, some component needs to do the orchestration of all of the functionality. The components handling this will be the controller components, so they will end up being more complex, but still testable.

Rejected: Tight coupling of components

Server Side Rendering

- *Status:* Proposed
- *Context:* In order to render some of the views, a few different pieces of data can be required, which may need a few database accesses.
- *Decision:* In order to provide the fastest view rendering experience, the web pages will be server side rendered. When many different pieces of data are required dynamically, having the view rendered on the server side can be faster as lest network requests are made in order to render the page. One network request will be made from the browser to render a new view. The server will handle the rendering of the page, as it has less overhead getting the necessary data for a webpage.
- *Consequences:* Some simple updating will be slower as it will take a full re render of the page on the server side, as opposed to just fetching a single piece of data and updating the view client side.

Rejected: Client side rendering

Data Modelling

- *Status:* Proposed
- *Context:* The data needs to be stored in the database in a way that optimizes speed for CRUD operations
- *Decision:* The data will be stored in the Mongo database in embedded documents, using a denormalized data model. This model will be optimal and have strong performance for read operations, and also be able to execute request and retrieval commands in a single operation on a per document basis. Additionally, this model allows for data updates in a single atomic write operation.
- *Consequences:* The duplication of data in the embedded documents would increase data storage and in extreme cases could hinder read operation performance. In the rare case that the application is scaled and complex many-to-many relationships are introduced into the database, the denormalized model could increase the complexity of development.

Rejected: Normalized data models are not as common as the proposed design choice, yet it was still considered to choose the optimal solution for the database and related operations.