Boundless - Design Document

System Interaction 2
CRC Cards 3
A Concrete Example of Updating State 4
Software Architecture Diagram 7
System Decomposition 8

System Interaction

Boundless will be using the following technologies:

- Server and Database: Firebase
- Front-end: React.js, React Native (for mobile experience), Redux (state management), JavaScript, CSS

The initial version of Boundless will be a web application, but we are planning on building a mobile experience as well. We expect most of our users to be using modern web browsers (Firefox, Chrome, Microsoft Edge), so there will be no issues with making requests to our Firebase server.

Requests will be made through user interaction with the GUI and go through Redux, a library for state management.

Below are some definitions to help in understanding the role of Redux and its functionality.

Redux Store:

- Holds application state
- Accessible from all components within the application
- i.e. a global state container

Actions:

- Payloads of information that send data from the application to the Redux Store
- In the context of our application:
    - Actions will be defined to interact with the Firebase DB, obtaining the requested data that will be fed to the reducer, which in turn triggers a state change within our component

Reducers:

- Specifies how application state changes in response to actions sent to the Redux Store

Dispatcher:

- Dispatches an action, the only way to trigger a state change within our component
- A part of the Redux Store

CRC Cards

Reducer	
Responsibilities	Collaborates
- To hold a set of sub-states of the global states. - Also responsible handling actions, and updating state accordingly.	- Actions - View
Reducer == Model	
In a redux framework, the store (ie: Global State) is connected to a set of reducers. Where each reducer holds a state for a specific view. Each reducer is also responsible for handling specific actions executed by from the view.	

Actions	
Responsibilities	Collaborates
- To return a payload to the reducers upon execution.	- Reducers - Views/Components
Actions == Controller	
When we want to something to happen in a app, we do this through actions which are basically async functions that are handled directly by the reducers. Reducers in the redux framework are directly connected to the global state of the app. Upon handling the action, the global state changes.	

Dispatcher	
Responsibilities	Collaborates
- Dispatches an action, the only way to trigger a state change within our component	- Actions - Reducers

	- Component

A Concrete Example of Updating State

User interaction with a “View Profile” Component:

- There could be a “View Profile” button within the component
- On pressing this button, a “Fetch profile” action will be dispatched
- Code will be written to specify the behaviour associated with dispatching this action
- i.e. We want to pull information about this user from the FirebaseDB, and return this data in a format we specify (JSON)
    - An example JSON:

    user = {

    “user_id”: 1,

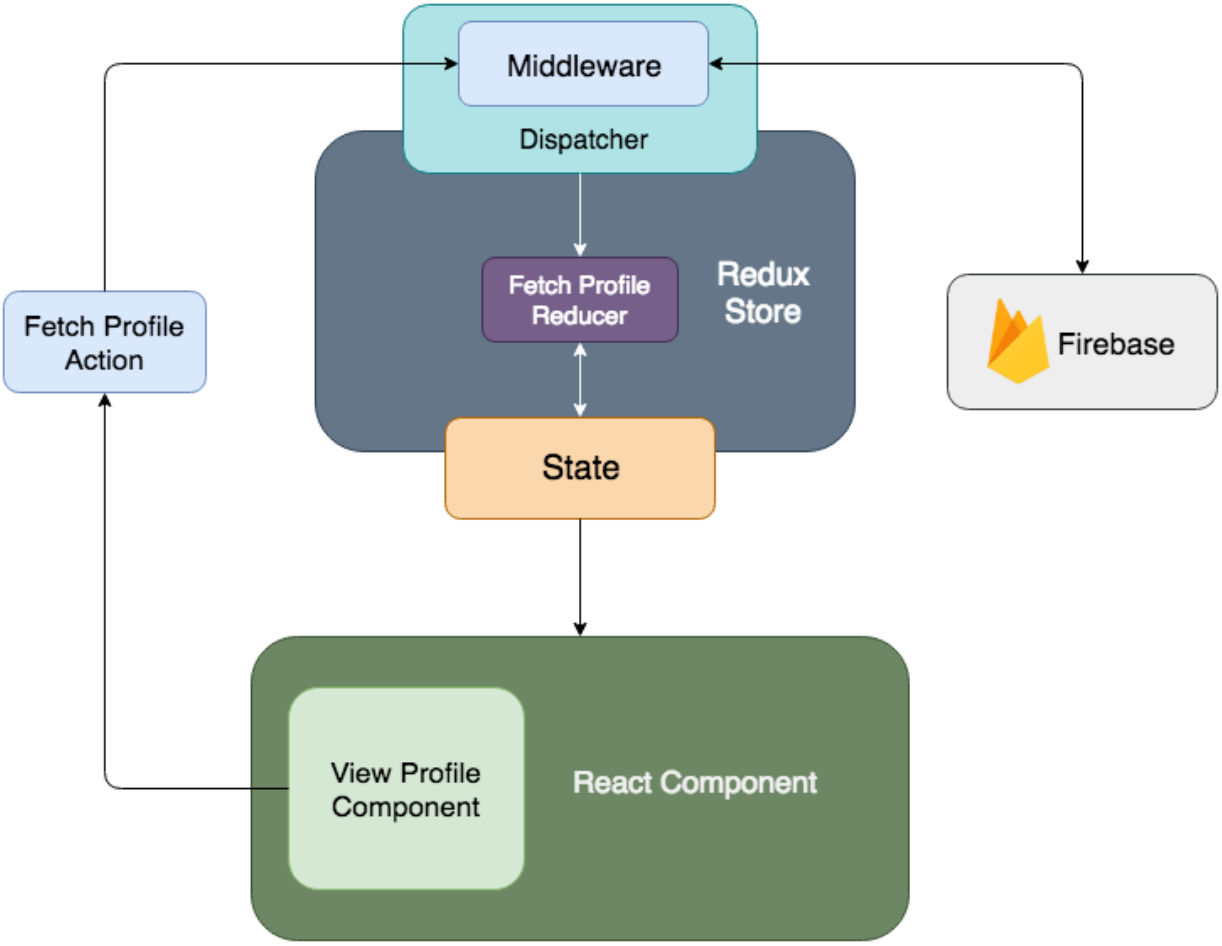
    “name”: “Test User”,

    “email”: “test@test.com",

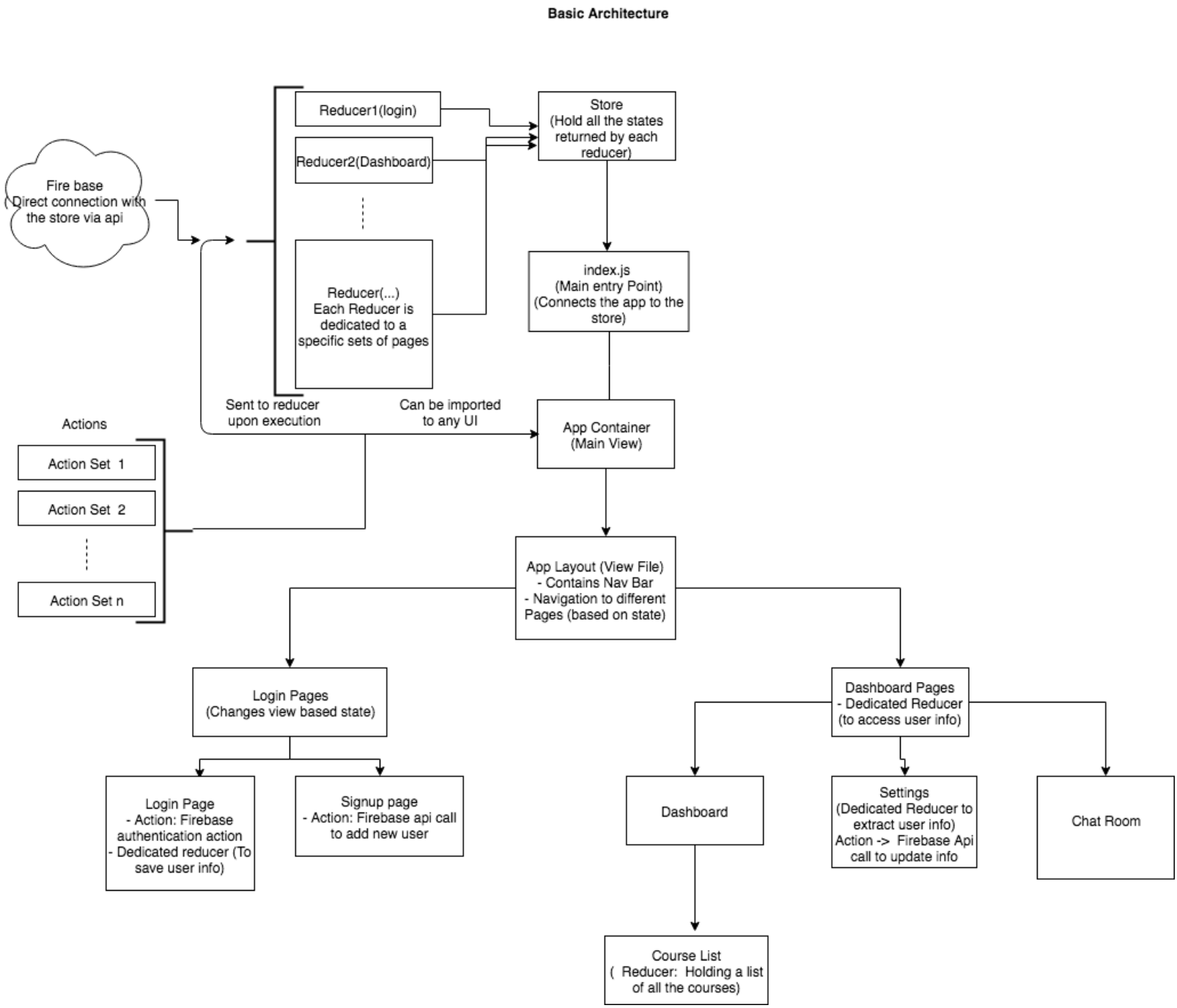
    “major”: “Computer Science”

    }

- The reducer will take this data and update the Redux Store in a manner we specify:
    - e.g. Update the field the Redux Store called “user_requested” by replacing its current value with the “user” JSON.
- The component from which the action was dispatched will have code to respond to changes in the Redux Store
    - e.g. The View Profile Component can display attributes within the “user_requested” entry in the Redux Store



The diagram on the next page gives a visual representation of these interactions:



Software Architecture Diagram

System Decomposition

To handle any sort of error through-out the program we will be using promises, which will allow us to execute different sets of code based on whether the promise returned successfully or with an error.

Each view will also hold authentication protocol that will be used to check the validation of the users input.

As the boundless platform will be on a web platform, supporting older browsers and lesser known ones is another issue. In this case, we would suggest the users to use a compatible browser while the boundless team creates a plan to provide additional support to these web browsers.

For IDE and other errors relating from that, our approach is having a bug reporting system where users can fill out a form that specifies the bug and methods in replicating the bug. The developers at Boundless will handle these reports in a timely manner while updating users in each update through patch notes.

Should users encounter server connectivity issues, for example, connecting to the server or being unable to login, the boundless team will examine the issue and determine the approach strategy to undertake. For example, if the user is unable to log into the platform and the server are down, the boundless team will attempt to send feedback and error status of the server to the user. Before this, we will prompt the user with a message to refresh or restart the application to see if it's client-side rather than server side.

For other unexpected problems, we rely on validation mechanisms that will output the respective error message to both the team at boundless but also to the users. If the issue is not resolved in a timely manner, users can fill out reports of these issues on the bug reporting form where it's sent directly to the developers at boundless.