# Continuous Assessment 1 – React Web App Development

Matthew Pantaleon

# User Stories

The main focus of the application is to know what each civilization and units have to offer as well as the ability to compare civilizations and units amongst each other.

* As a new player I want to know more about the civilizations and units to learn more about the game.
* As an experienced player I want to know more about the nuance stats of units and manage resources to improve my knowledge of the game.
* As a player overall I want to compare civilizations and units just out of curiosity to slightly improve on the game.

# Wireframes

My original design of the application looked something like so in *figure 1*:

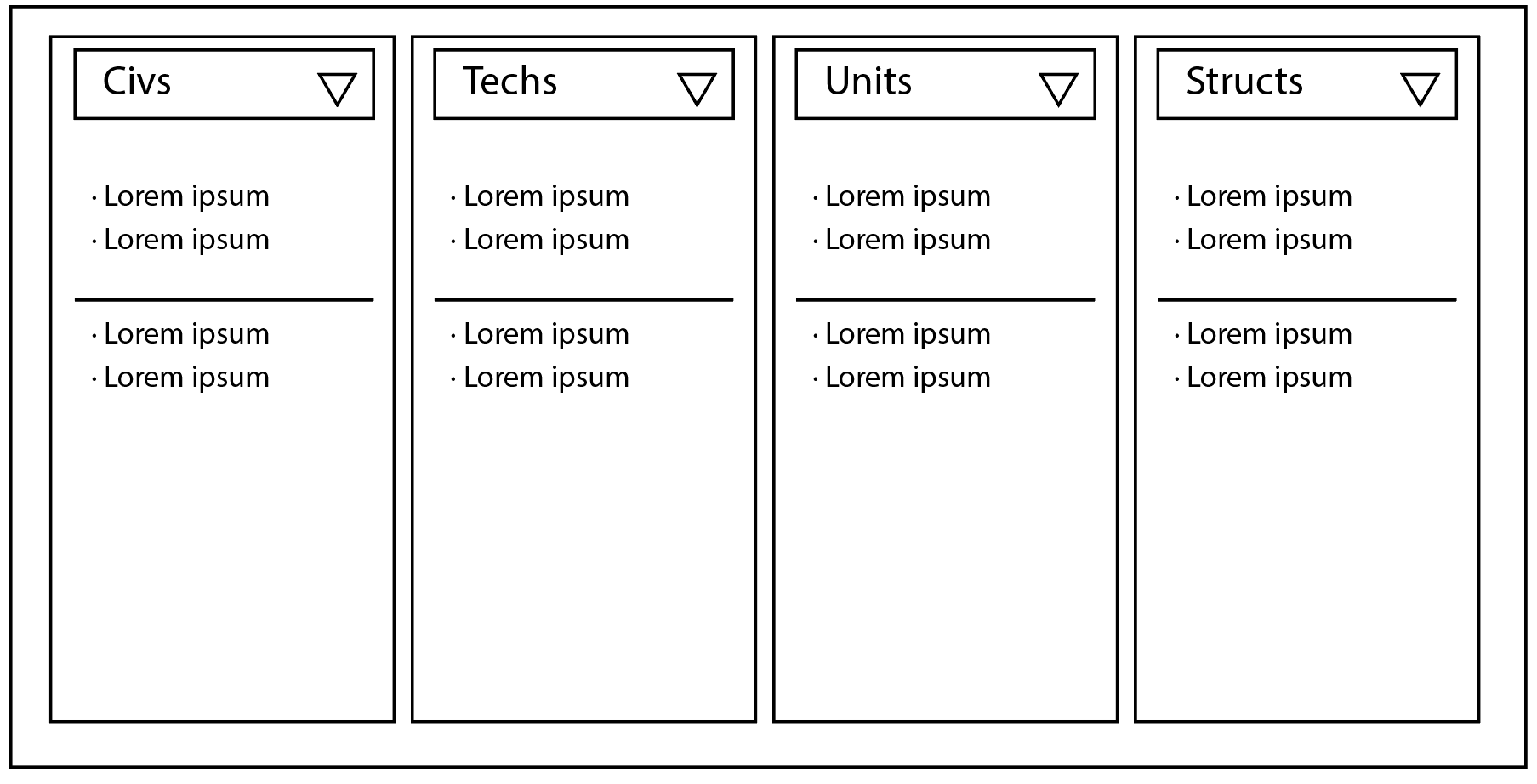


Figure 1 Initial Wireframe

After looking into the API it is too big to have a simple UI like this. Also the structure of the API is a bit disjointed in relation to the other endpoints, there are four endpoints for each category.

* Civilization objects only specify their respective bonuses and point to their unique units and techs. Does not point to any other techs or units available and unavailable to itself.
* Tech objects only refer to their effects and the units that are affected by it, the structure where it is researched and not to the civilizations that have access to it.
* Unit objects only refer to its stats and the structure it is made from and not the civilizations that have access to them.
* Structures refer to the structure description and their functionality and all age states. When directly accessing a structure with its name but not id, will return all the age states for that structure.

Due to the discontinuity of the API endpoints the design went from encompassing all of the API to just civilizations and a select handful of units as seen in *figures 2 and 3*.

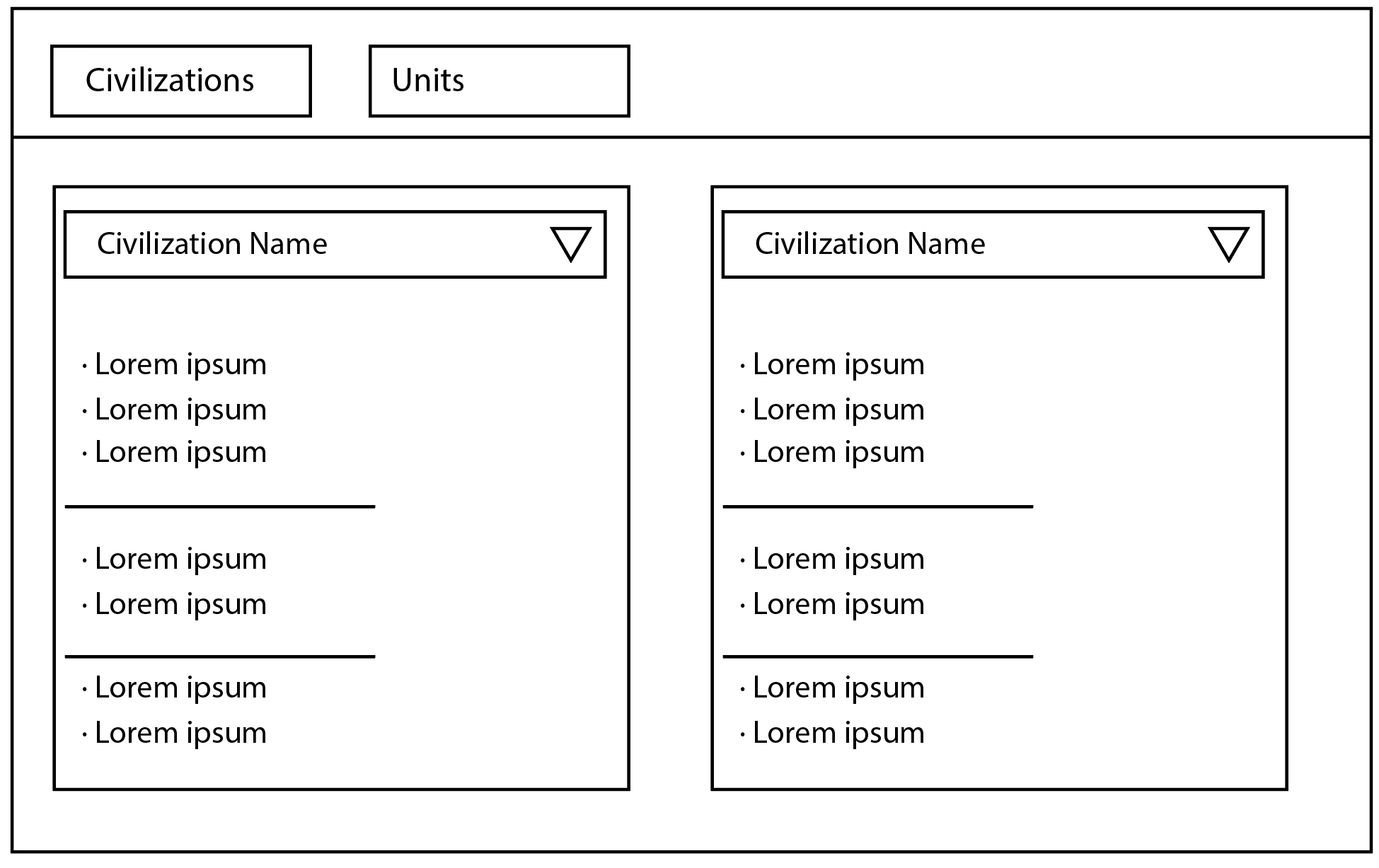


Figure 2 Civilization Tab

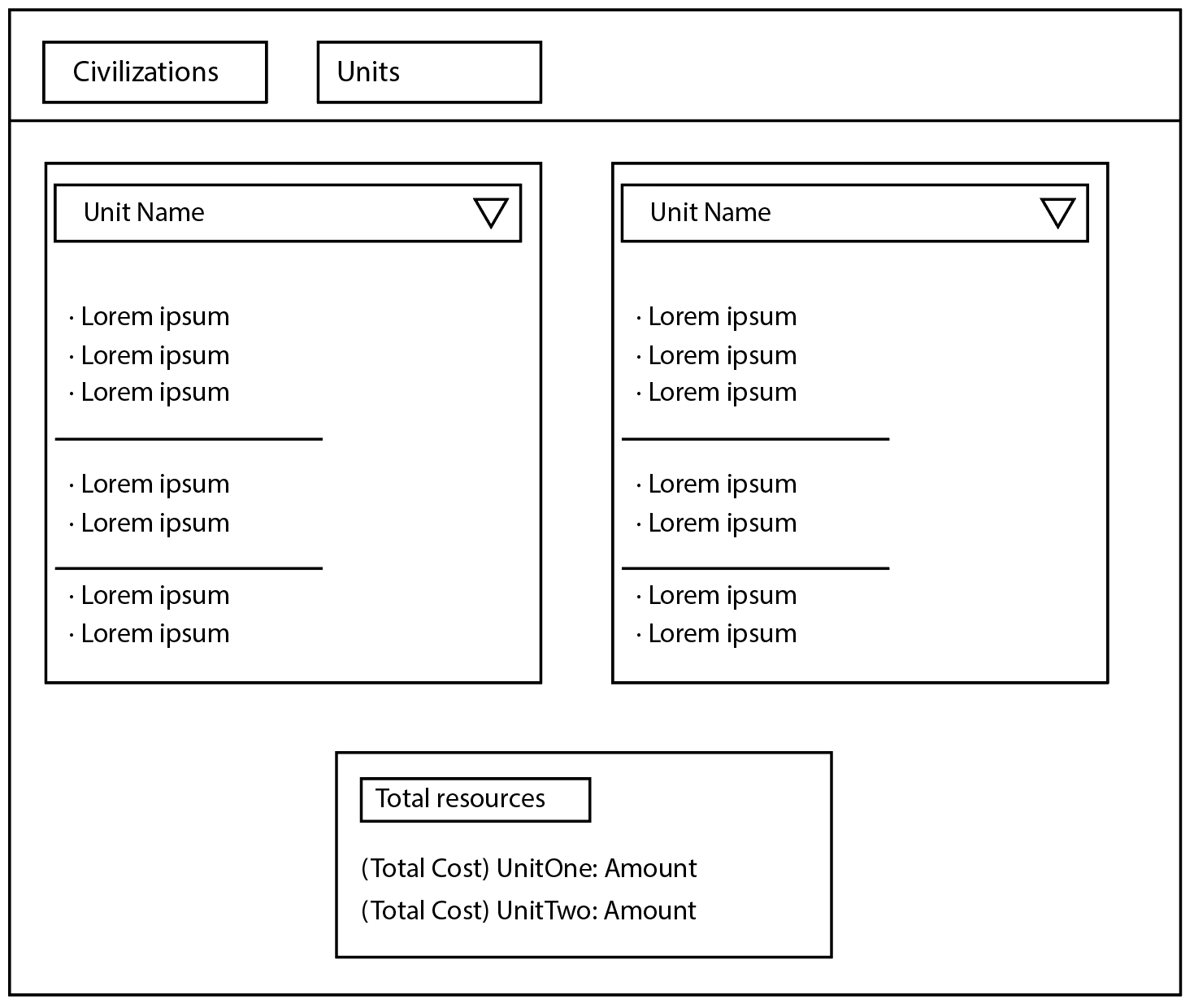


Figure 3 Unit Tab

# Implementation

This is the final Component Structure of the application in *figure 4.*

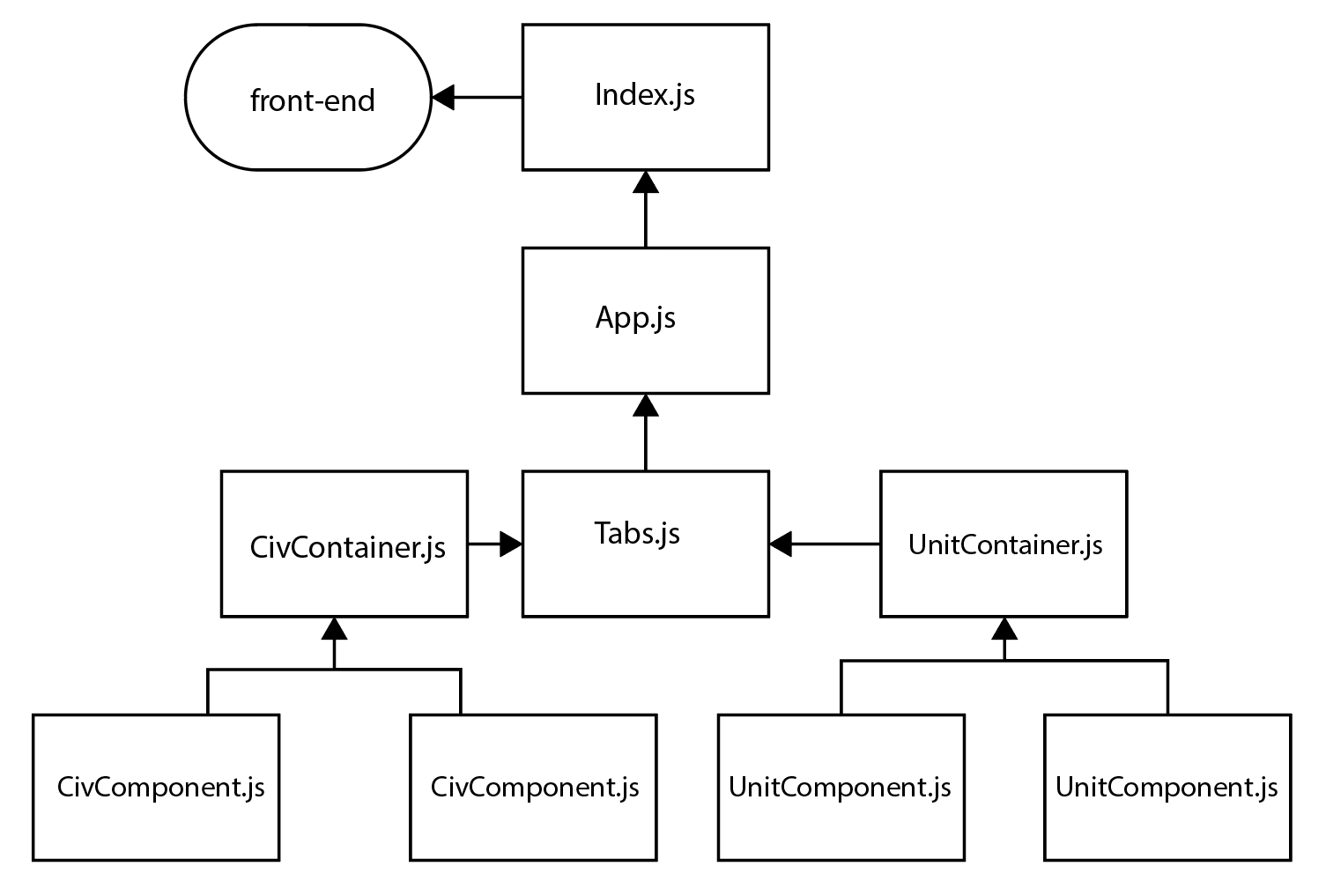


Figure 4 Component Structure

Index.js references a DOM element in which the App.js will be rendered onto and cascades to any other connected component.

App.js contains the component Tab from tabs.js, this renders a navigation for each link and route inputted. I streamlined the scalability of adding routes by using an array of objects, similar to how Vue-router does it where it contains the component, path and a front-end name.



Figure 5 Route Array

In the tabs component it validates the prop input using propTypes similar how Vue.js passes props from parent to child components.

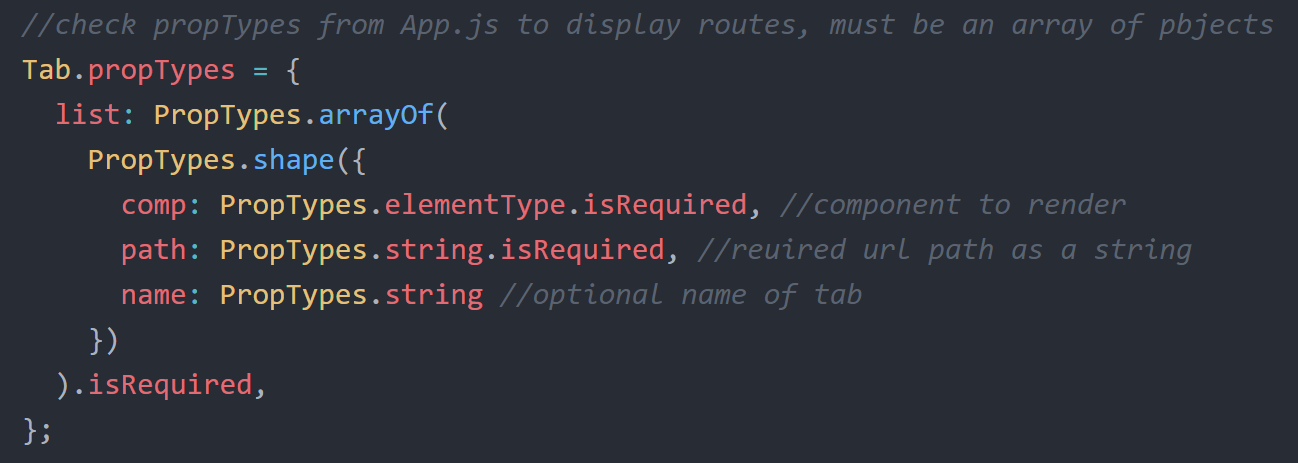


Figure 6 Prop Validation

In *figure 6* it validates the prop if it is an array and the shape of each value in that array. The name property can be optional.

Then the render loops through the array to display the links and routes. Using JQuery to dynamically style is tab option accordingly.

CivContainer.js just is the parent component of CivComponent.js, the container is needed for encapsulating the multiple CivComponents for navigational use. It can accept up a string and/or integer for its arguments or only one array containing API endpoints.

APILoader is where the application makes all API calls it is a function that returns and object with an array of an array of objects depending on the arguments passed. String or array for first argument and an optional id.

CivComponents and UnitComponents initially load the API and afterwards store the data to the localStorage and so as long the localStorage with the items exist each mounting of those components will directly use the localStorage instead of calling the API reducing server load. Using localStorage isn’t really recommended especially for bigger APIs but the *Age of Empires API* is small enough. Code example show in *figure 7.* Since the API returns an array of endpoints for unique techs and units the API has to be called for this every time.



Figure 7 Component LocalStorage

Upon mounting the CivComponent and UnitComponents it defaults to the first civilization or unit respectively.

Since the API contains no images, only a handful of units have been selected to be compared with. The component has all the units I needed to filter it using arrays with unit names as the values.

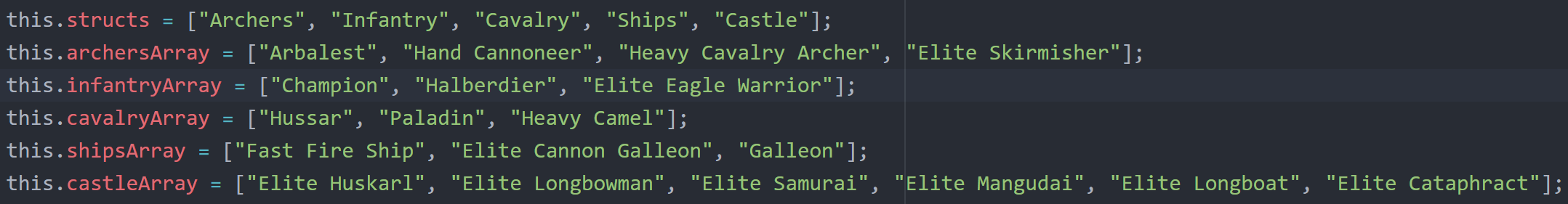


Figure 8 Unit Name Arrays

On mounting of the UnitComponent it filters the units array to get objects corresponding to the value in each name array and selects the default unit to the Arbalest. Then it sends back the some data back to the CompareContainer component.

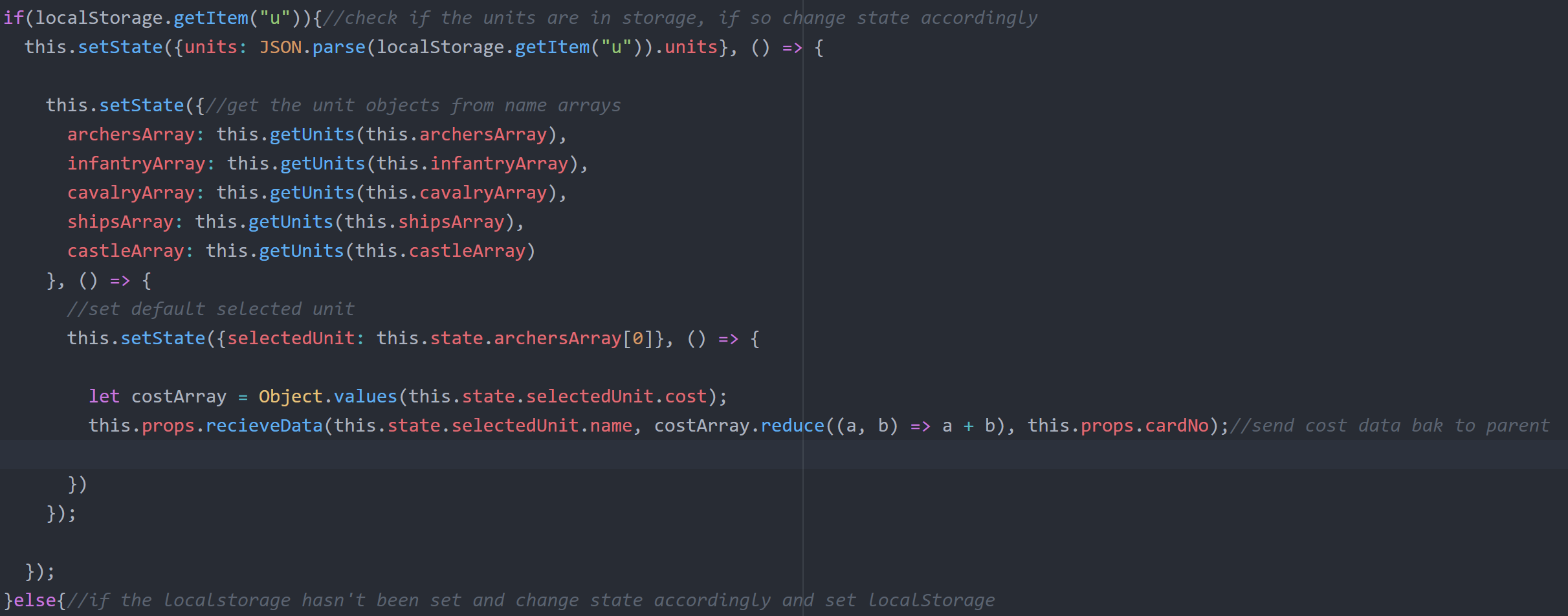


Figure 9 Unit Component Mount

If for some reason the localStorage for units is null, UnitComponent can call the API again and follow through normally. Each UnitComponent has props which contains a callback function to return the name and total cost of the unit as well as an identifying variable to determine which values to alter on the CompareContainer unit calculator.



Figure 10 UnitComponent props

The UnitComponent has two dropdowns showing the unit type and individual unit as shown here in *figure 11.*

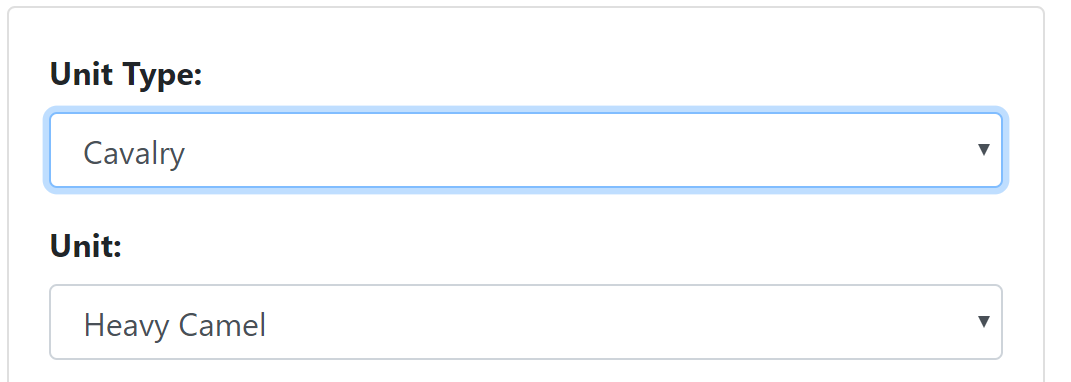


Figure 11 Two Dropdowns

This is to sort the units in a more UI friendly manner hence also why the unit names array was split to their respective unit type. The unit type dropdown affects the values from the unit dropdown.

When switching unit type the unit selected value will default to the first value of the new selected array of units.

Images for the civilizations and units are done through using the names of those objects and invokes a require method each time. *Figure 12* shows for the selected unit image.



Figure 12 Unit Image source

Each unit object’s cost is stored as an object but its total numerical value is what the unit calculator needs, so converting all the values into an array and reducing it, then passing it back to the parent component along with the unit name and identifying variable. As shown in *figure 13.*

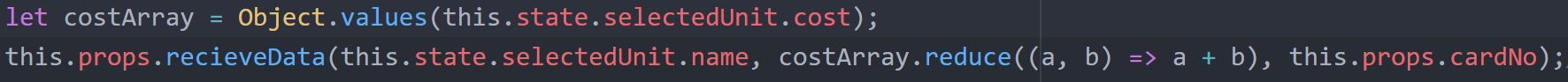


Figure 13 Using Callback Function

# Reflection

I did not get to use all of the API in particular the technologies and structures and that would require importing my own images which would be time consuming and the API structure is disjointed enough that more implementation of logic would have to be made to accommodate the disjointedness. If the API did come with images the application could have more functionality and compare all units.

I used JQuery’s AJAX with asynchronous set to false, this is bad practice as it can take a while to load the page all the while with no user feedback, I don’t fully understand how Promise objects work and I wanted to guarantee that there is data ready when the components load.

The API does not have an *Access-Control-Allow-Origin* header I couldn’t seem to get around this, both the development server and deployed server on firebase when using chrome will complain about this and the application just doesn’t load.

I had to completely rewrite a lot of code regarding setState functions as I was using too many and causing crashes as there were too many asynchronous processes happening.

I planned to include blacksmith upgrades in the unit comparison component but just did not have the time to properly implement it with the components current implementation.

Running the application in chrome, it takes a second to run but testing it on Firefox it can take several minutes to boot up, especially if the localStorage is empty. It could be because of firebase or the actual API requests.

I learned how to effectively use callbacks and how to manage API requests and Bootstrap interacts with React and Router Components.

GitHub: <https://github.com/MatthewPantaleon/React-CA1>

FireBase: [age-of-empires-2-comparator.web.app](https://age-of-empires-2-comparator.web.app)