React Project- Would You Rather App?

Jean-François Vanreusel

Introduction

The Would-You-Rather App is a game where users answer questions given two different options. Users can also ask new questions. The users who answer and ask the most questions get more points and are on top of the leaderboard.

Views

This application will use 8 Views:

- **Login** View
- **Home** View (with a list of questions for the logged-in user)
- **QuestionAnswer** View
- **QuestionResult** View
- **New Question** View
- **Leaderboard** View
- **Logout** View
- **404** View

Below is more information, including screenshots about each view. The initial views were created on paper, so I use screenshots to show what the final result looks like.

Login View

The **Login** View looks as follows

It includes:

- **Nav** component
- **Login** component

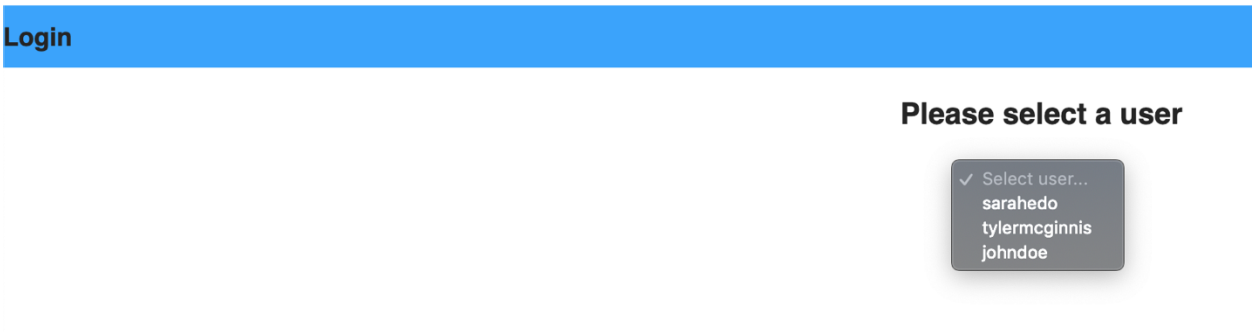

Figure 1: Login view

The Navigation bar includes only displays the Login menu when the user is not logged in. When the user is logged in, all the options are available including the name of the logged in user as shown below.

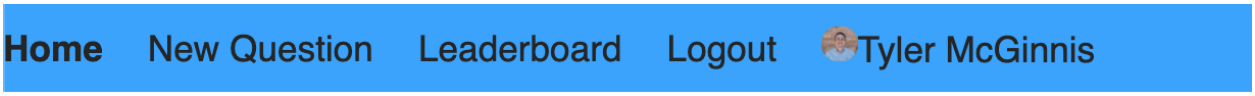

Figure 2: Navigation Bar for logged in user

Home View

When the user is logged in, the Home View looks as follows

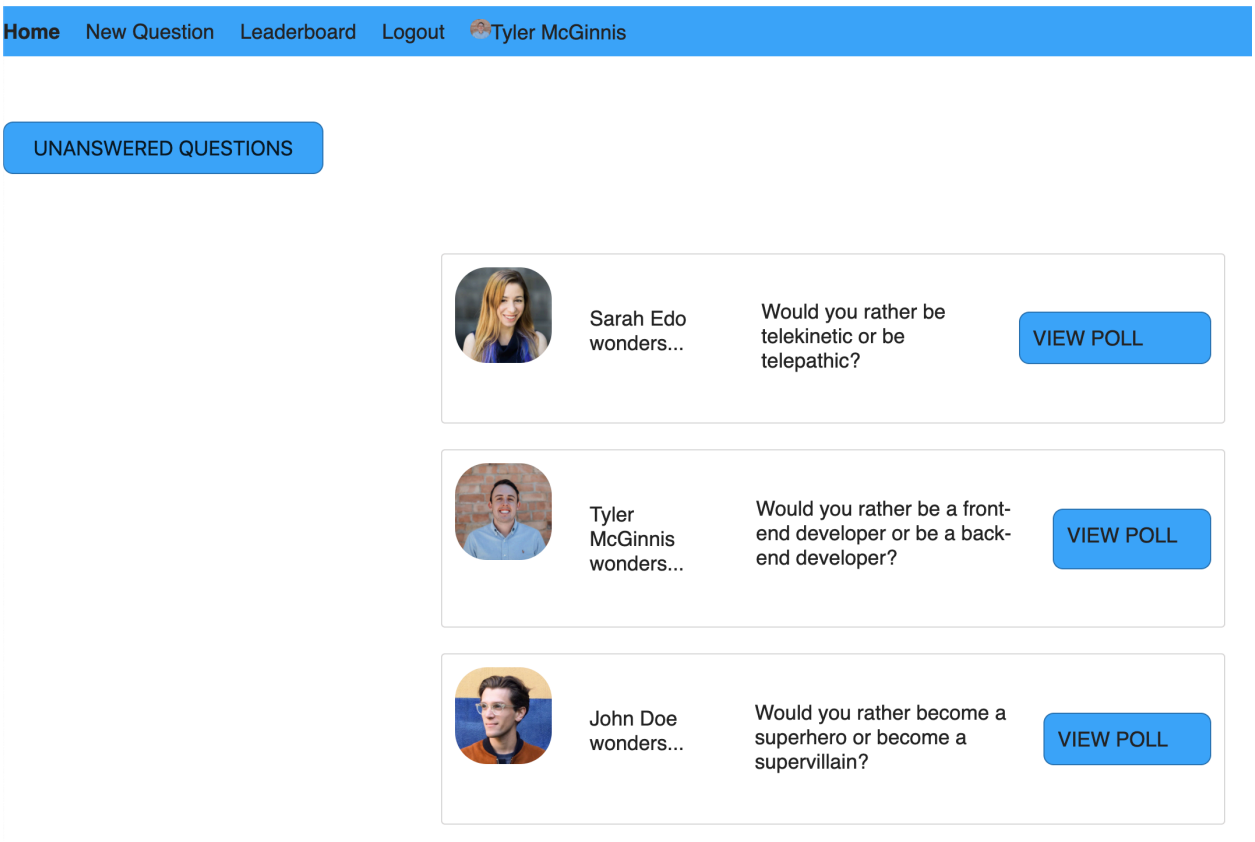

Figure 3: Home view for loggedIn user

This view includes:

- **Nav** component (navigation bar at the top of the screen)
- **QuestionList** component (the list of questions displayed under each-other
- **Question** component – the unit of a Question List component.
- A toggle button to manage the Question List content. The button toggles between unsanswered and answered questions. By default, it displays unanswered questions to motivate users to respond to more questions.

When the user clicks on the **"view poll"** button, it will display more details on the question. If the question is already answered by the current user, it will show how users answered this question via the **QuestionResults** component/view. If the user hasn't answered the question yet, he/she will be given the opportunity to do so with the **QuestionAnswer** component/view. We use the **QuestionPage** component to display the correct component.

QuestionAnswer View

The Question View looks as follows:

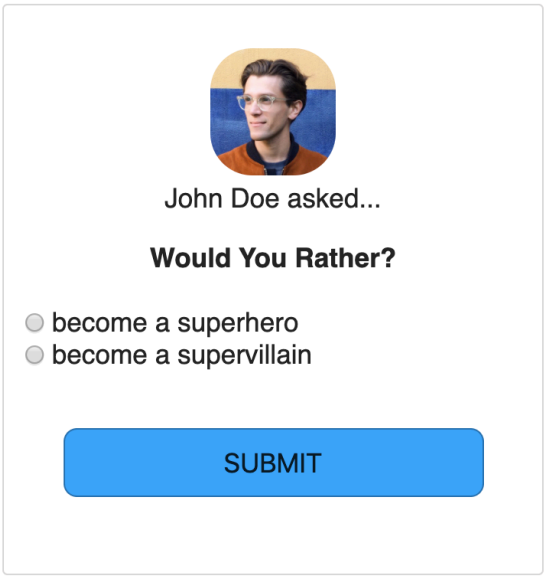

Figure 4: Question Details seeking user's answer

It includes:

- **Nav** component
- **QuestionAnswer** component

The **QuestionAnswer** component shows the person who asked that specific question, his/her name, and the 2 options to pick from.

QuestionResult View

When the user clicks on the "View Poll" button for questions that he/she has already addressed, the Question Results get displayed as follows.

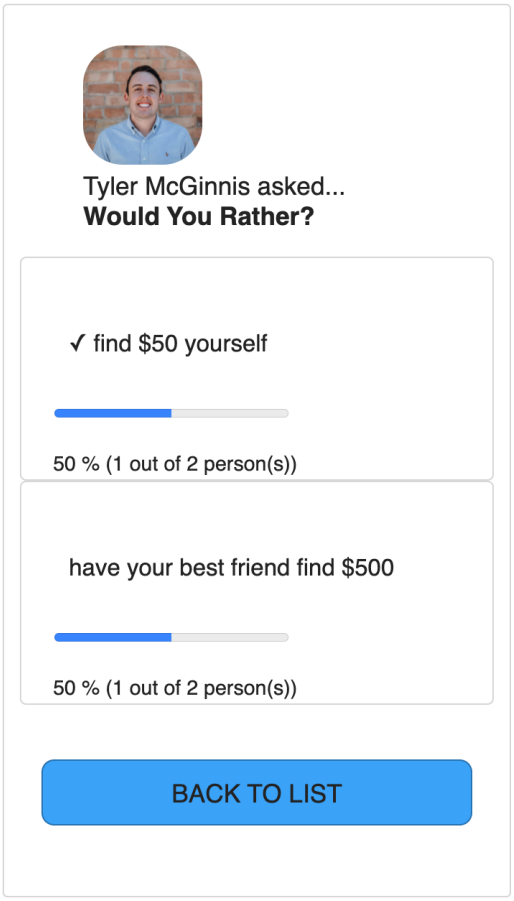

Figure 5: Question Results once answered by user

The view includes:

- **Nav** component
- **QuestionResult** component

The **QuestionResult** component shows the user who asked the question, his/her name. It also displays the two options (via **OptionResult** component) and how participants responded to those two choices. The choice picked by the active user is indicated by the checkmark.

New Question View

The New Question View enables users to create new questions. The View looks as follows:

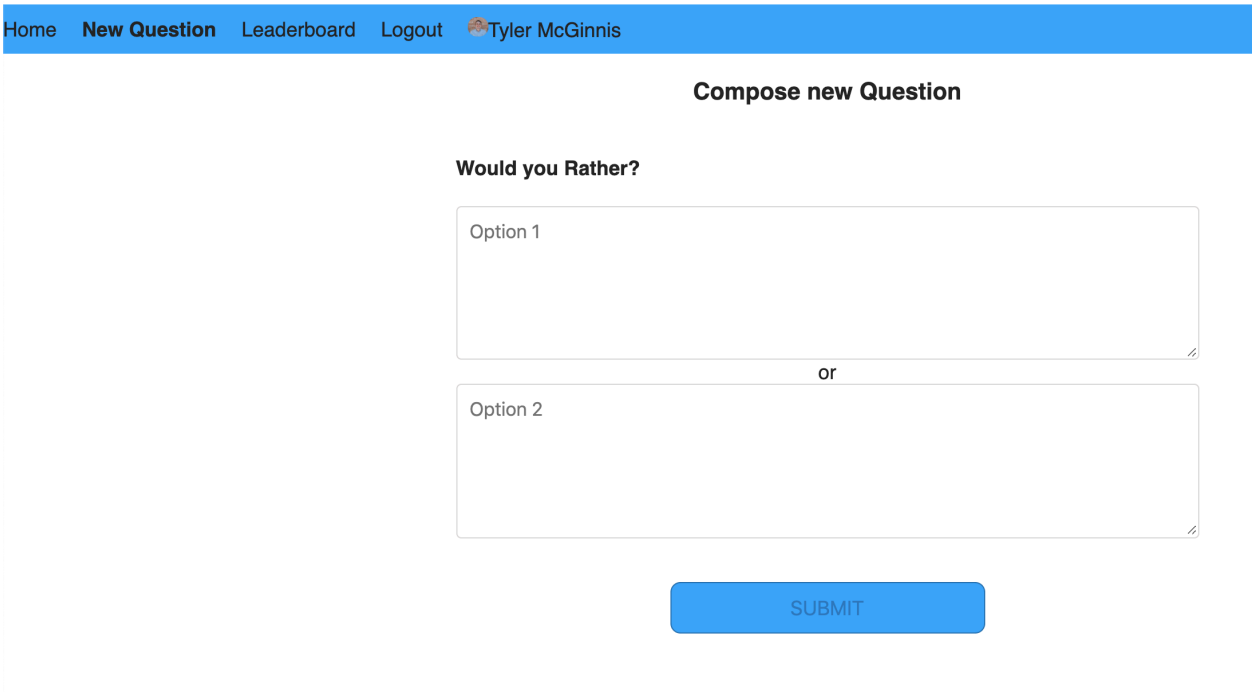

Figure 6: New Question View

The view includes:

- **Nav** Component
- **New Question** Component

The **New Question** Component displays two text fields where users can type in their 2 options. The Submit button is activated once text is present in both option fields.

Leaderboard View

The Leaderboard View summarizes the score achieved by the users. It looks as follows:

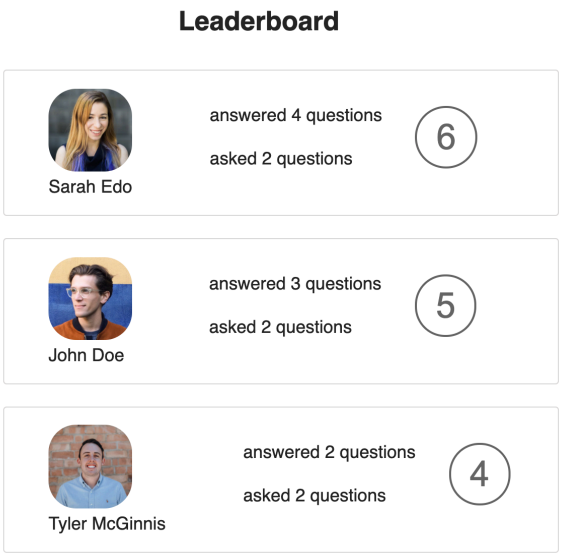

Figure 7: Leaderboar view

Each entry on the Leaderboard shows the user, the number of questions answered and asked as well as the total score achieved by this user.

It includes:

- **Nav** Component
- **Leaderboard** component
- **User** Component

Logout View

The Logout view allows users to logout of the application or switch to another user. It looks as follows.

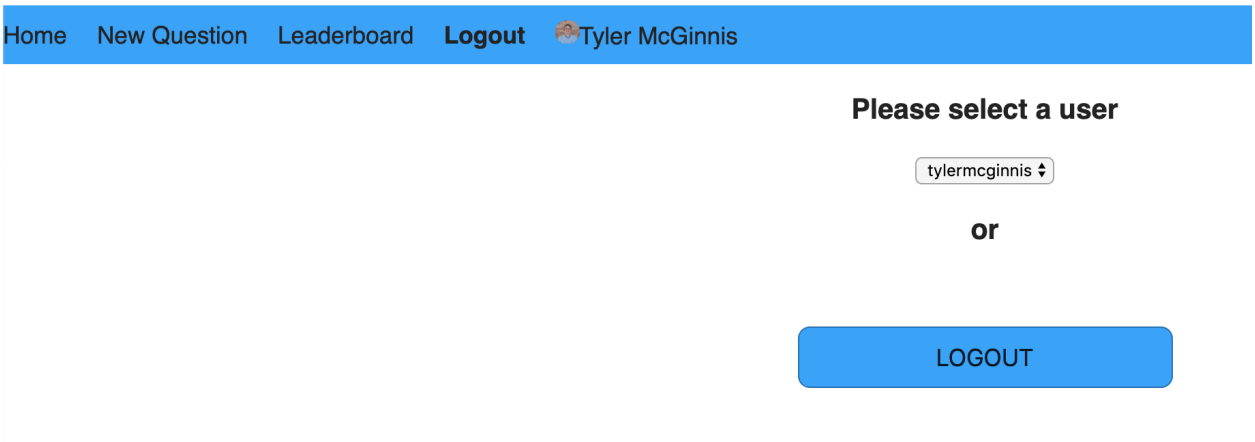

Figure 8: Logout view

The Logout view leverages the **Login** component.

404 View

When the user reaches a page which is not available, it will be displayed a “404 page” and redirect that user to the Home page.

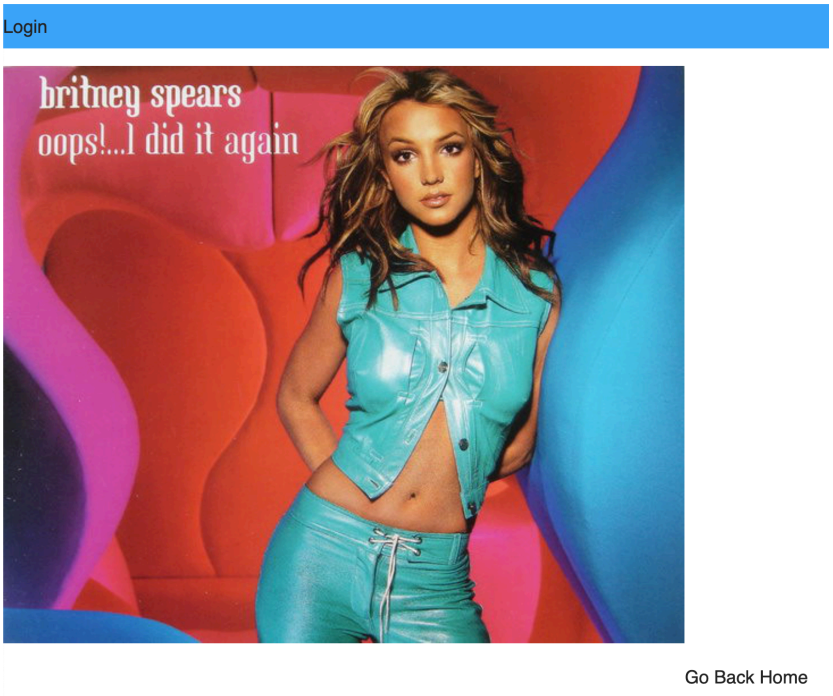

Figure 9: 404 Page -- Oops! I did it again!!

Private Route

One of the key requirement of this application is that users need to be logged in in order to access most functionalities. To control access to the authorized locations, we created a Private Route component which checks if users are authenticated before allowing them to reach a page.

Redux Store

Our application leverages Redux to manage its store (i.e. state and methods to access/change that state).

In the case of this application, we decided to attach the following data to our state store:

- **AuthedUser** (authenticated user)
- **Questions** (list of questions available in the app)
- **Users** (list of users participating in the app)

The **Questions** slice of the state in the store will be modified by *actions* that go through the *questions* reducer. The **Users** slice of the state in the store will be modified by actions that go through the *users* reducer. And, similarly, the **authedUser** portion of the state in the store will be modified by actions that go through the *authedUse*r reducer.

In our project, we first created our action creators. We created action creators which use the API to load the users and questions. Once those are received, those action creators return respective actions. Those actions can then be processed by the **reducers** via the dispatchers.

Actions

We create action creators for **Questions**, **Users** and **Authenticated User** (Authed User).

Questions

- RECEIVE_QUESTIONS – to load Questions into the store.
- ADD_QUESTION – to add a single question to our existing list of questions.
- ADD_ANSWER_TO_QUESTION – to add a user's answer to a specific question.

Users

- RECEIVE_USERS – to load Users into the store.
- ADD_QUESTION_ANSWER_TO_USER to record the user's answer to a question.

AuthedUser

- SET_AUTHED_USER – to set the user in the store

Reducers

The Reducers are methods that modify the state in the store. Their signature is basically as follows:

```
(previousState, action) => newState
```

Reducers take the existing state and an action and return a new state.

In our app, the questions reducer will impact the questions part of the state changes, the users reducer will impact the users part of the state changes and the autheduser reducer will impact the authedUser part of the state.

The **Redux** store provides the **Dispatch** function which enables us to send actions to the dispatcher which then sends them to the proper reducer. Note that the Dispatch function is provided to connected components (see below).

Connected Components (containers)

Connected Components (aka containers) are components that need access to the Store. Those components get access to the store via the **connect** function. As they access the store, they can transform (or simply pass through) the store data as a prop for the component. This work occurs in the function mapStateToProps. The first argument of this function represents the part of the store which needs to be accessed. We use data de-structuring to access the part that

we're interested in. The second object represents the props object passed to the Component as it gets created. Below is a typical example of a mapStateToProps function used in this project.

```
function mapStateToProps( { users } ){
 return {
   userIds: Object.keys(users)
       // sort users in score descending order
       .sort((a,b) => (Object.keys(users[b].answers).length + users[b].questions.length) -
                      (Object.keys(users[a].answers).length +
users[a].questions.length))
   }
}


//special export because it's a connected component
export default connect(mapStateToProps)(Leaderboard)
```

A connected component will also be exported with the connect function. This function uses the mapStateToProps function as argument and returns a function as result. This resulting function takes the component as argument.

Data Access API

The data is stored in a fake database (_DATA.js). More information on the data is available here.

Redux Middleware (Thunk)

The purpose of the Redux Middleware is to separate data fetching operations from the data display operations (in the components). This is especially important when data is stored in a database or gets accessed in async mode. We don't want the UI to freeze because the data is not yet fully loaded.

Redux middleware manages this data management complexity for us. As part of the middleware, we also added a **logger** middleware which allows us to catch information about about state as it gets modified by the actions and reducers.