Summary

The front-end application was developed using React and Vite. Additional packages such as Redux, Tailwind, Lodash, and React Router were used to 'mimic' API calls and caching, CSS styling, and routing.

Folder Structures

The application follows a feature-based pattern. The `Components` folder contains feature-specific components and reusable `elements`. The `API` folder contains configuration of the Redux Store and his reducers. The `Hooks` folder contains the the `useLoadStations` which will update the store stations data if necessary with the JSON data and would also include hooks that would deal with different types of HTTP requests. The `Pages` folder contains main components for each route, and would be restructured into entity-specific folders as the project grows. The `Router` folder contains the `routes.jsx` file for defining routes and components. The `Utils` folder contains helper functions and if necessary constants.

Tailwind Config

Colors from Figma were added to the Tailwind config, named according to context (e.g., `red` is `error`, `green` is `normal`).

TSConfig

The TSConfig was used to set up absolute imports, which are important for simplifying import statements and improving code maintainability.

Implementations

On mount renders all available stations

This is achieved by the `useLoadStations` hook, which aside from also dealing with the persistance (which I will talk about it later on), it also created the array of objects defined by the `Station` Type.

The data is fetched from a JSON, that is located locally in a `data` folder, this had to be this way, since unfortunately I was not able to implement the backend in Kotlin, but I believe the hook behavior would be almost identically being the only difference the way the data is fetched, since we still would read it from a JSON, which would be the format that would come from the API Request.

Search Functionality

```
const filteredStations: Station[] = useMemo(() => {

const filteredBySocketTypes = stations.filter((station) =>
(station.socketType.includes('Normal') && socketTypes.Normal) ||
(station.socketType.includes('Rápido') && socketTypes.Rapido));


if (searchTerm) {
        return searchStations(filteredBySocketTypes, searchTerm,
searchFilters);
}

return filteredBySocketTypes

}, [stations, searchTerm, socketTypes])
```

This search functionality is achieved by two different components `SearchInput` which deals with the updating of the `searchTerm` which when is updated, an `useMemo` callback (Which is defined in the `Stations.tsx` page component) that when is triggered it will run an util method I developed `searchStations`, which will take 3 arguments, the current stations data filtered already by the socket types, the search term, and the search filters state and look for field values that include that term but only care about those fields if the search filter for it is turned on.

This `Search Filters` states are stored in redux, which is also what deals with their update.

Socket Type Filter Functionality

The `useMemo` callback that is previously shown aside from filtering by the current searchTerm, it also filters by the Socket Types that are enabled, the control of those states are also done by storing it in redux, and handling their sets and gets in the same way we do with the Search Filters

This `useMemo` only triggers when the stations or searchTerm or the socketTypes are updated to avoid not necessary re-renders.

App Data Management

The overall data management is achieved by the integration with redux and the `useMemo` that I have been mentioning

Stations Favorite Persistance

This is achieved by using the package `redux-persist`, which basically will listen when the data that is defined to be persisted, in this case, only the `stations` has been rehydrated.

This can be seen in action in the `useLoadStations` where I use the `subscribe` method from the `redux-persist` to listen for the rehydration of that data set.

I believe this could bring performance issues, but if I had more time, I could have it made only persist the necessary fields, such as the favorite state and station id, to reduce the amount of data that needs to be stored and retrieved.