File Structure and Mock APIs

In **UI Layout and Project Structure**, we built up a hardcoded initial UI layout for ***Project Mini-Mek*** using Semantic-UI-React, added a tab bar component controlled by Redux, and started using a "feature-first"-style folder structure for our code. In **Using Redux-ORM**, we saw how to use Redux-ORM to manage relational data in a Redux store. Now we'll put those pieces into practice. In this section, **we'll prepare the app for feature development, define data models with Redux-ORM, use those models to load data into the store and display it, and add the ability to track which items are selected**.

Making File Structure Consistent

As described previously, we're using a "feature-first" folder structure. However, not all the code qualifies as a "feature". There's really three main divisions: common code that's generic and reused throughout the application, code that's specific to a feature, and the application-level code that ties those features together.

We'll do some more file shuffling to reflect those concepts:

> **Commit d5a218f: Move core project files to /app for consistency**

After the move, the project file structure now looks like this:

```
- src
  - app
    - layout
      - App.js, App.css
    - reducers
      - rootReducer.js
    - store
      - configureStore.js
  - common
```

```
    - components
    - utils
  - features
    - mechs
    - pilots
    - tabs
    - unitInfo
    - unitOrganization
  - index.js
```

Extracting Components in Features

The original UI layout we made in Part 4 has a single component for each of the tab panels. It also has fake data hardcoded right into the UI layout itself. That's not going to work well when we start dealing with actual data. So, before we can work with data, we should extract some components from those panels.

Looking at our `<Pilots>` component, it really consists of two main parts: the list of pilots, and the form showing the details of a single pilot. It makes sense to extract separate components for each of those, so we'll split out a `<PilotsList>` component and a `<PilotDetails>` component.

In addition, the pilots list has a couple different parts. The header section is static and won't change. We could leave that as part of, the render method in `<PilotsList>`, but we might as well split it out as a `PilotsListHeader>` while we're at it. At the same time, we definitely need to be able to render an individual row for each pilot entry, so we'll create a `<PilotsListRow>` component that we can use while rendering the list.

Since we're transitioning from hardcoded text in the UI, we should start making use of some kind of sample data. For the moment, the simplest approach is to treat `<Pilots>` as a "container" component, store an array with one item in its state, and pass that down to the list and details components.

> **Commit 16fa599: Extract components from Pilots panel, and add initial test data**

The main part of our `<Pilots>` component now looks like this (skipping library imports):

library imports).

features/pilots/Pilots.jsx

```
import PilotsList from "./PilotsList";
import PilotDetails from "./PilotDetails";

const pilots = [
    {
        name : "Natasha Kerensky",
        rank : "Captain",
        age : 52,
        gunnery : 2,
        piloting : 3,
        mechType : "WHM-6R",
    }
];

class Pilots extends Component {
    state = {
        pilots : pilots,
    }

    render() {
        const {pilots} = this.state;

        // Use the first pilot as the "current" one for display, if availa
ble.
        const currentPilot = pilots[0] || {};

        return (
            <Segment>
                <Grid>
                    <Grid.Column width={10}>
                        <Header as="h3">Pilot List</Header>
                        <PilotsList pilots={pilots} />
                    </Grid.Column>
                    <Grid.Column width={6}>
                        <Header as="h3">Pilot Details</Header>
                        <Segment >
                            <PilotDetails pilot={currentPilot} />
                        </Segment>
                    </Grid.Column>
                </Grid>
            </Segment>
        );
    }
}
```

Since our Mechs list is effectively identical so far, we'll do the same set of transformations on that as well.

> **Commit a13f21c: Extract components from Mechs panel, and add initial test data**

There's one new and useful tidbit to show out of this commit. In Battletech, a Battlemech can weigh anywhere from 20 to 100 tons, in 5-ton increments. Mechs are frequently described and grouped based on their "weight class". Mechs from 20-35 tons are "Lights", 40-55 tons are "Mediums", 60-75 tons are "Heavies", and a Mech weighing 80-100 tons is an "Assault". This is a description that can be derived based on the known weight of a Mech type. So, we can write a small selector function that takes in a weight value, and returns a description of the weight class:

features/mechs/mechSelectors.js

```
const WEIGHT_CLASSES = [
    {name : "Light", weights : [20, 25, 30, 35]},
    {name : "Medium", weights : [40, 45, 50, 55]},
    {name : "Heavy", weights : [60, 65, 70, 75]},
    {name : "Assault", weights : [80, 85, 90, 95, 100]},
];

export function getWeightClass(weight) {
    const weightClass = WEIGHT_CLASSES.find(wc => wc.weights.includes(weig
ht)) || {name : "Unknown"};
    return weightClass.name;
}
```

Nothing too fancy here. We define an array with one entry per weight class, and use the `Array.find()` method to return the first entry that matches a filter function. Our filter function uses the `Array.includes()` method to see if the given weight value is in the list of weights for that weight class. If we don't find a valid weight class, we provide a default result so that we can always `return weightClass.name`. This selector can then be used in `<MechsListRow>` and `<MechDetails>` to provide a display value for the given Mech instance.

Adding a Mock API for Sample Data

We're at a point where we need to start working with some data. In a bigger application or more realistic example, that would probably mean standing up a separate backend server to send back responses, or at least using some kind of "mock API" tool.

In order to keep this tutorial application as focused as possible, we're going to follow the classic advice of "Do The Simplest Thing That Could Possibly Work" / "You Ain't Gonna Need It". In this case, that means **writing a file to contain our sample data, directly importing it into the source code, and writing a mock API query function that just returns that data in a promise.**

We'll need a bit of UI to actually let us call that mock API function. Since we've already got our tabs bar set up, the simplest way to add some UI is to just create a new panel with a button that fetches the data, and add that as another tab. We'll label it the "Tools" panel for now. It might also make a good place to add "Import" and "Export" buttons down the road.

> **Commit 5bc2c28: Add a Tools panel and mock API for loading sample data**

The only really interesting bits of out of here are the initial sample data, the mock API function, and our thunk action creator:

data/sampleData.js

```
const sampleData = {
    unit : {
        name : "Black Widow Company",
        affiliation : "wd",
    },
};

export default sampleData;
```

data/mockAPI.js

```
import sampleData from "./sampleData";
```

```
export function fetchData() {

    return Promise.resolve(sampleData);
}
```

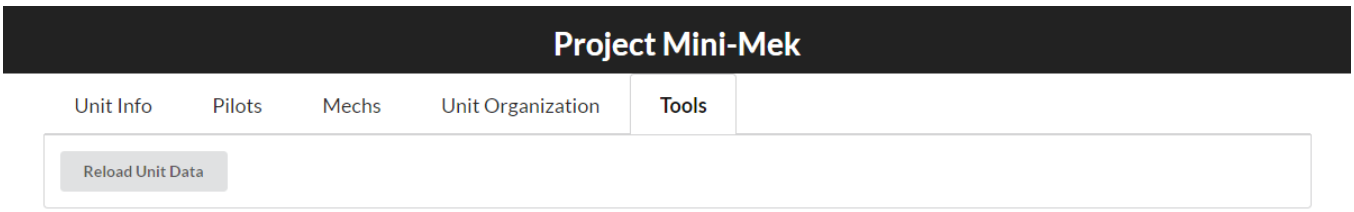
features/tools/toolActions.js

```
import {fetchData} from "data/mockAPI";

import {DATA_LOADED} from "./toolConstants";

export function loadUnitData() {
    return (dispatch, getState) => {
        fetchData()
            .then(data => {
                dispatch({
                    type : DATA_LOADED,
                    payload : data
                })
            });
    }
}
```

After adding the panel, the UI looks like this:



Feel the excitement! :)