Phase IV Design Report

Github Repository: https://github.com/dylanmccoy/ee-461l-idb
Website: https://je66yliu.github.io/songtrackr/

Team Information

Team Members	Email	Github Username
Kevin Chen	kchen3568@gmail.com	kxc3568
Matthew Golla	matthew.golla8@gmail.com	matthewgolla
Jerry Liu	je66y.liu@gmail.com	je66yliu
Dylan McCoy	dylan.mccoy@utexas.edu	dylanmccoy
Diana Shao	dianashao01@gmail.com	dianashao
Ryuichi Yanagi	ryanagi@utexas.edu	ryuichiyanagi

Canvas Group: Morning-6
Project Name: SongTrackr

Information Hiding

It is difficult to imagine that any of the core functionality between instances might change anytime soon, since Billboard has published a weekly list of the 100 most popular songs and their respective artists since 1957. However, it is possible that in the future we would like to add more instances to the website or that our API sources might change. Because our Python scrapers are separate from the web application - that is, there is no coupling between the two - we can modify the scrapers to add more instances to our database or adjust to any API changes. We also could implement functionality such that if data that wasn't found in the database is queried, we access the API to add the data. Another possible change would be the user interface of the application, such as adding accompanying new features or a stylistic update to remain modern-looking. Since each component has a separate file, the design

elements can be created and modified without changing the structure of the main web app or interfering with its code.

The modularization of our codebase makes it easier to add new changes to our application. Components can be added by creating a new file for each component, which can be locally tested before exporting it to the main app file to be deployed. Furthermore, each component is self-contained, so it can be changed easily without affecting other components. For example, in Phase III we needed to implement sorting into our model pages. With our current modularization approach, we were able to implement this by only modifying the component of each model, not needing to make changes to any of our other components like the Home page or any model instances. By abstracting the specific design choices of each component away, we were able to efficiently implement new functionality to our website.

```
Class App extends Component {
// App only knows props and which components to render
      render() {
            <Component Y {props => x, y, z}/>
            <Component X>
            <SongGrid {props => {...props} /
            <Component Z>
        }
}
```

Figure 1. App.jsx code snippet

```
Class SongGrid extends Component {
      menuItems = [
            {List of all attributes to sort by},
            {Func: () this.handleSort(this.sortBy('attribute'))},
      ]
      render() {
            <div> render all the things </div>
            <Dropdown  sortButton  menuItems= {this.menuItems>/>
      }
}
```

Figure 2. SongGrid.jsx code snippet

While the modularity of the code makes attributes extensible, it reduces readability and performance in some cases. For example, some functions take many unclear parameters, which can be difficult to figure out without documentation. Furthermore, each function that takes other functions as parameters has additional overhead that reduces performance. To combat the readability issue, we use function and variable names that describe their purpose without needing to read all of the surrounding code. To mitigate the performance loss, we took the bottleneck data retrieval functions and only called them when that specific data was requested, reducing the number of function calls.

Design Patterns

We incorporated two design patterns into the backbone of our design: Singleton and Observer.

Singleton Design Pattern

The singleton design pattern was used for state management, where a single instance of the state tree is created for the lifetime of the app. Having multiple instances of the state may result in unexpected behaviors; for example, different components may read from different state instances leading to inconsistencies in displayed data. This also made our code much easier to manage as there is a single point of access to the global state no matter where the current component is located in the component hierarchy.

```
class SongGrid extends Component {
    constructor(props) {
        super(props);
        this.state = {
            songList: [],
            currentPage: 1
        };
    }
```

```
class SongInstance extends Component {
    initialState = {
        trackName: '',
        artistNames: '',
        albumName: '',
        albumArt: '',
        trackUrl: ''
    };
```

Figure 3. Multiple Local States Before Refactor

```
const INITIAL_STATE = {};

const modelReducer = (state = INITIAL_STATE, action) => {
    switch(action.type) {
        case ModelActionTypes.UPDATE_SONG_LIST:
            return {
                ...state,
                songList: action.payload
            };
        case ModelActionTypes.UPDATE_ARTIST_LIST:
            return {
                ...state,
const rootReducer = combineReducers({
    model: modelReducer
});
```

```
export const store = createStore(rootReducer, applyMiddleware(...middlewares));

ReactDOM.render(
    <Provider store={store}>
        <HashRouter>
            <PersistGate persistor={persistor}>
                <App />
            </PersistGate>
        </HashRouter>
    </Provider>,
    document.getElementById('root')
);
```

Figure 4. Single Instance of a Global State After Refactoring

The library we used to achieve this design pattern was Redux. Although the singleton design is not immediately obvious in our code, the underlying redux store is always instantiated with a single instance and passed to the app, as achieved by the createStore method and the Provider component shown above. The reducer functions can be called from any component in the app and will always update the same instance of the state.

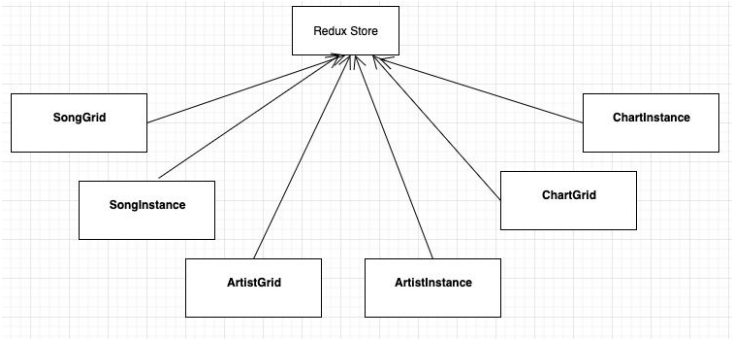


Figure 5. Our Singleton Design Pattern

Pros:

- Reduces code clutter - Each component does not need to implement its own local state.
- Increases predictability -Having the singleton means only one single source of truth, so the data will always be consistent.

Cons:

- No encapsulation - the store is accessible from any component, and every component has the same level of access as any other component.
- Multiple instances of components - A component may need to have multiple instances with different states, while the store is only a singleton. This means that multiple copies

of certain fields may need to be stored in the state, so components are not fully reusable or encapsulated.

Observer Design Pattern

With the previously mentioned singleton store, each component needs to observe the store in order to make its updates appear on the display. In this case, the observers are the components, and the global store is the observable. We achieve this through the connect method of redux. Each component is wrapped with the connect method to create a higher order component that can be interfaced with the global store. Before implementing the observer pattern, we had our own updater functions in the main app component to send information to other components and to tell them to update, as seen in Figure 6 below. We also had to pass those functions down to all components that would use them, leading to a lot of code repetition and clutter.

```
class App extends Component {
    constructor(props) {
        super(props);
        this.state = {
            songList: [],
            artistList: [],
            currentSongView: null,
            currentArtistView: null
        };
    }

    handleUpdateSongList = songList => {
        this.setState({ songList: songList });
    }

    handleUpdateSong = song => {
        this.setState({ currentSongView: song });
    };

    handleUpdateArtist = artist => {
        this.setState({ currentArtistView: artist });
    };
```

```
<Route path={'/songs/table'}>
    <SongTable songList={songList} handleUpdateSongList={this.handleUpdateSongList} handleUpdateSong={this.handleUpdateSong}
</Route>

<Route path={'/songs/page/:pageNum'} render={props => <SongGrid {...props} songList={songList} handleUpdateSongList={this.han
```

Figure 6. Updater Functions Before Implementing the Observer Design Pattern

After implementing the observer pattern using redux, the main App.jsx file is a lot cleaner and only consists of a render method. The second image in Figure 7 below is an example of a component that plays an observer role. Note that the connect method and the mapStateToProps allow it to observe global state through the props.

```
class App extends Component {
    render() {
        return (
            <div className='App'>
                <Header />
                <Switch>
                    {/* HOME & ABOUT */}
                    <Route exact path='/'>
                        <HomeCarousel />
                    </Route>
```

```
const mapStateToProps = createStructuredSelector({
    songList: selectSongList
});

const mapDispatchToProps = dispatch => ({
    updateCurrentSong: song => dispatch(updateCurrentSong(song)),
    updateCurrentArtist: artist => dispatch(updateCurrentArtist(artist)),
    updateSongList: songList => dispatch(updateSongList(songList))
});

export default withRouter(connect(mapStateToProps, mapDispatchToProps)(withStyles(styles)(SongGrid)));
```

Figure 7. After Implementing Observer Pattern

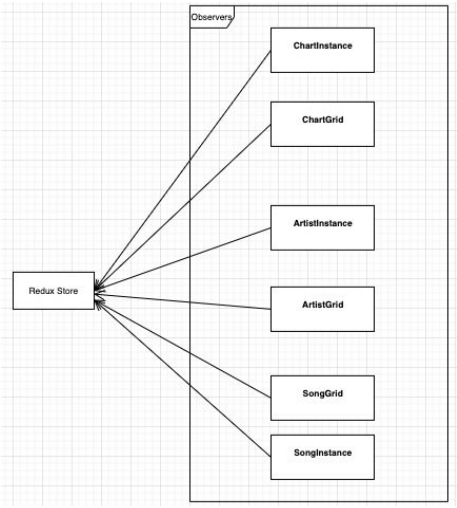


Figure 8. Our Observer Design Pattern

Pros:

- Reduce code clutter - Just like the singleton pattern with the global store, making each component an observer eliminates the need to pass every single state update down the component hierarchy.
- Code modularity - Components will not depend on the state of other components, only the global store. Whenever you want to create a new component, you do not need to modify the data flow for other components to account for the new component you are adding.

Cons:

- Boilerplate code - To use redux observer pattern, there is a lot of boilerplate code to set up, including the reducers, selectors, and actions. In addition, you have to create the mapStateToProps and mapDispatchToProps objects for every component you want to

observe and dispatch actions to the global store. Though this can be seen as repeated code, it's a fair tradeoff for having the convenience of a global state management.

Refactoring

Three of our refactors included template method pattern with sorting functionality, template method pattern with the backend, and container pattern with the frontend display.

```
handleSortName = () => {
    const { songList, updateSongList } = this.props;
    songList.sort(this.byName);
    updateSongList(cloneDeep(songList));
};

handleSortArtist = () => {
    const { songList, updateSongList } = this.props;
    songList.sort(this.byArtist);
    updateSongList(cloneDeep(songList));
};
```

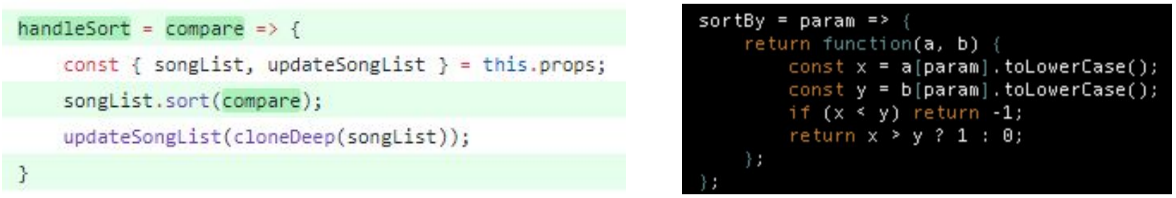
```
byName = (a, b) => {
    const x = a.name.tolowercase();
    const y = b.name.tolowercase();
    if (x < y) return -1;
    return x > y ? 1 : 0;
};

byArtist = (a, b) => {
    const x = a.artist.toLowerCase();
    const y = b.artist.toLowerCase();
    if (x < y) return -1;
    return x > y ? 1 : 0;
};

byAlbum = (a, b) => {
    const x = a.album_name.toLowerCase();
    const y = b.album_name.toLowerCase();
    if (x < y) return -1;
    return x > y ? 1 : 0;
};
```

Figure 9. Sorting Functionality Before Refactor (Individual Sort Handlers and Comparators)

A code smell we originally had was duplicated code. As you can see in Figure 9, we had a different handler and comparator for each type of String based sorting, where the difference between each method was just the parameter used to sort. To fix this, we used a template method pattern refactor in order to create a more generalized sorting process.



```
handleSort = compare => {
    const { songList, updateSongList } = this.props;
    songList.sort(compare);
    updateSongList(cloneDeep(songList));
}
```

```
sortBy = param => {
    return function(a, b) {
        const x = a[param].toLowerCase();
        const y = b[param].toLowerCase();
        if (x < y) return -1;
        return x > y ? 1 : 0;
    };
};
```

Figure 10. Sorting Functionality After Refactor (Generalized Sort Handler and Comparator)

As seen in Figure 10, we extracted the general functionality into a template sort handler and comparator. Instead of having an individual handler and comparator associated with each implemented sort, we use the handler to pass the desired parameter to the comparator. The comparator then uses the parameter to know which data to access and compare with. By implementing our sorting functionality this way, we can now add a different kind of String type

sort by passing a different parameter to the handler. With our earlier implementation, we would have had to add another handler and comparator function in order to add a new String type sort.

Figure 11. Backend Functionality Before Refactor

Figure 11 shows another example of duplicated code segments in our old codebase. Each model type required its own separate get and post, with similar code. To fix this, again we used a template method pattern refactor to generalize loading models and instances.

Figure 12. Backend Functionality After Refactor

After the refactor, we have generic getModel() and getInstance() methods, as seen in Figure 12, which are called by each model type. If we want to add a new model type, we can just call the template model and instance methods and it will load the model type from the database. In our old implementation, we would need to add another app.get() call to the database and also associated app.post() calls to the database depending on the model type.

Another code smell we had was overly long render functions contained in a single component. One such example is in the SongInstance component, where we originally had all of our code that rendered the display written out in that one render function. This created a really long file that was hard to read, as partly shown in Figure 13.

```
<article>
    <div className='cf pa2'>
        <div className='pa2'>
            <a href={trackUrl} className='dib' target='_blank' rel='noopener noreferrer'>
                {
                    albumArt ?
                        <img src={albumArt} alt='album art' className='mw5 center db outline black-10 link dim'/>
                    :
                        <i style={{ color: '#b3b3b3' }}>Cannot display image</i>
                }
            </a>
            <dl className='mt2 f6 lh-copy'>
                <dt className='clip'>Title</dt>
                <dd className='fw6 ml0 white truncate w-100'>{trackName}</dd>
                <dt className='clip'>Artist</dt>
                <dd
                    className='ml0 gray truncate w-100 artist-link'
                    onClick={async () => {
                        updateCurrentArtist({});
                        history.push(`/artist/${artistNames}`);
                        const artistData = await fetch(`${process.env.REACT_APP_BACKEND_URL}/searchartistid`, {
                            method: 'post',
                            headers: {'Content-Type': 'application/json'},
                            body: JSON.stringify({ id: artistId })
                        });
                        const artist = await artistData.json();
                        updateCurrentArtist({
                            artistName: artist.name,
                            popularity: artist.avg_pop,
                            timeOnCharts: artist.time_on_charts,
                            albums: artist.albums,
                            songs: artist.songs,
                            profileArt: artist.images.length ? artist.images[0].url : '',
                            chartsIn: artist.charts_in
                        });
                    }}
                >
                    {artistNames}
                </dd>
                <dt className='clip'>Album</dt>
                <dd className='fw3 ml0 white truncate w-100'>{albumName}</dd>
            </dl>
```

Figure 13. SongDisplay Functionality Before Refactor

```
return (
    currentSongView && currentSongView.trackName ?
        <SongDisplay />
```

Figure 14. SongDisplay Functionality After Refactor

We fixed this issue by creating a separate stateless SongDisplay component that contained the formatting functionality for displaying individual songs. This way, we could move all the code responsible for formatting to a separate component, resulting in a much cleaner version of SongInstance. In the SongInstance component, we simply referenced the stateless SongDisplay component instead of having to write out all the code. This can be seen in Figure 14. This is an example of the container design pattern, since we are separating the presentation and functionality into multiple container components.