Frameworks and Architectures of the Web - Week 15 Workshop

Building Web Applications in React: Part 2

In this week's workshop, we will continue and build upon the 'Piano' example we started last week. To allow us to learn more about React and dive into some new concepts, we've provided a new example to work from. This example is based upon the final result of the tutorial from Week 14, except with some modifications:

- Most notably, we now have full access to all 88 keys of the piano.
- The piano now has dynamic range: notes can be played at ***pp*** (pianissimo - very soft), ***mf*** (mezzo forte - with moderate force), and ***ff*** (fortissimo - very strong or loud). As we'll find out later, having dynamic range allows us to explore richer musical possibilities with this app.
- Due to the fact that the piano has dynamic range, our state property within App.js is now modified: each 'key' now has a `isPlayedAtDynamic` property, which can now represent four possible values:
    - `none` – the key is not pressed
    - `pp` – the key is pressed softly
    - `mf` – the key is pressed with moderate force
    - `ff` – the key is pressed with very strong force
- Changing the `isPlayedAtDynamic` property for each key either stops a note (if set to 'none') or plays a note at the appropriate dynamic (`'none'`, `'pp'`, `'mf'`, `'ff'`). As such, the Key component (Key.js) has been modified so that it can be 'played' at these dynamic ranges.
- Apart from these changes, the overall structure of our component-based app is the same:

```
App
 |
 +- KeyboardDisplay
 |
 +- Keyboard
    |
    +- Key
    |
    +- Key
    |
    +- ...
```

In this walkthrough, we'll modify our app so that it can play entire songs, very much like a Pianola. The user will have the capability to select a song and either play or stop that song. This will be done by:

- Loading data from an external API.
- Interpreting musical notational data from that API, and transforming that in a way so that the piano is 'played' by a) setting / unsetting the `isPlayedAtDynamic` property for appropriate keys on the piano; b) depressing and controlling the 'damper pedal' of the piano and c) controlling a) and b) through the sequencing and timing of a) and b) to produce the miracle of music.

Before continuing, I would like to thank the University of Iowa Electronic Music Studios for so kindly providing these high quality piano samples, and to Mace Ojala who has spent countless hours converting and preparing these samples for you to enjoy.

Okay, so let's begin.

Download the Files from the LearnIT Site, Setup the Project

First, download the `react-piano-tutorial-before` ZIP file. Unzip the file and open your Terminal. In your terminal, `cd` the directory to your project file and run:

```
npm install
```

then

```
npm start
```

Once you're up and running, you should be able to preview your project in your browser. Load the project into your editor of choice.

If you're having trouble at this point, ask a TA.

Playing Chords on the Piano

Before we start playing music, we'll need to abstract the actual actions that the instrument is capable of playing. Specifically, in order to play music, our piano should be able to:

a) Play a single note.
b) Play multiple notes at the same time (a chord).
c) Raise the damper by pressing the damper pedal, so that the notes are sustained.
d) Release the damper pedal, so that any sustained notes are stopped.

a) and b) can be accomplished by writing a single method called `setDynamicForNotes()` that allows the piano to play one or more multiple notes at a time. The method will copy the `state.keys` property, select the appropriate keys, and change their state so that they are played at the desired dynamic. We'll add this method to our App.js class:

```
setDynamicForNotes(notes, dynamic = 'mf') {
  let keys = [...this.state.keys];
  notes.forEach(note => {
    let key = this.state.keys.find((el) => note === el.note);
    key.isPlayedAtDynamic = dynamic;
  });
  this.setState({
    keys: keys
  });
}
```

The above method:

- Creates a deep copy of the state's key's array via the spread operator: `[...this.state.keys]`.
- The `notes.forEach` method iterates over each note as passed into the function, and then sets the dynamic of each key.
- Our now modified `keys` array within this function is now applied back to the component's state, so that our piano can 'respond' to the key changes and play the notes.

We'll call this method inside our `handleKeyDown()` method which is triggered from the `Keyboard` component when the user presses a key so that we can bind this logic to our user interface. Our `handleKeyDown()` method should look as follows:

```
handleKeyDown (pianoKey) {
  console.log('key is pressed');
  this.setDynamicForNotes([pianoKey.note]);
}
```

We'll also need to correctly bind the correct `this` context within our constructor by adding the following to our class's `constructor()` method.

```
this.setDynamicForNotes = this.setDynamicForNotes.bind(this);
```

Test and run the app and play some keys, what happens?

Understanding Component Lifecycle Methods

You may notice that when you test your app, every time you press a key, all of the other keys have been pressed 'play' again. For example, if you press `C4` and then `E4`, both `C4` and `E4` play again at the same time? Surely this is not how a piano should work!

The reason why you're getting this unusual behaviour comes down to the way we're managing state in the piano. When the `setDynamicForNotes()` method is called, the entire keyboard state is updated even if only a single key is pressed, so that keys which are already 'playing' start 'playing' again.

To solve this problem, we'll need to know modify our `Key` component so that it should only 'play' a note at a specific dynamic if it is not already played at that dynamic. We'll do this by tracking the current status, or 'dynamic' of the key. In the Key.js file, add the following to its `componentDidMount()` method:

```
this.dynamic = this.props.pianoKey.isPlayedAtDynamic;
```

This means that we set the 'dynamic' property of its state class to its initial value (which is most likely `none`) when the key is initialised. Next, we'll set this property again once a note is played. Add the following as the first line of the `componentDidUpdate()` method:

```
this.dynamic = this.props.pianoKey.isPlayedAtDynamic;
```

Finally, we'll need to add the `shouldComponentUpdate()` life cycle method. This method gets called by React when a component's properties change, and we use that to add logic to determine if a component should do anything in response to changes to its properties. Inside this method, we are comparing the note's current dynamic to its new value, and we return 'true' if the note's dynamic actually changes:

```
shouldComponentUpdate() {
  return this.dynamic !== this.props.pianoKey.isPlayedAtDynamic;
}
```

Test and run your app, and the piano should behave normally.

Stopping and Damping Notes

Next, we'll add the logic to stop notes. In the way that we've written our `setDynamicForNotes()` in our App class, we can just simply call that method and set the `dynamic` value of those notes to `none`. Add the following to the `handleKeyUp()` method:

```
this.setDynamicForNotes([pianoKey.note], ‘none');
```

Next, we'll add in the logic for raising and releasing the damper pedal. First of all, we'll add a state property to our App class so we can track whether the damper pedal is depressed or not. Add the following to the app's `state` object in its constructor:

```
damperPedalDown: false,
```

Next, we'll add the following methods to our App class.

```
damperPedalDown() {
  this.setState({
    damperPedalDown: true,
  });
}

damperPedalUp() {
  let keys = [...this.state.keys];
  keys.forEach((key) => key.isPlayedAtDynamic = 'none');
  this.setState({
    damperPedalDown: false,
    keys: keys
  });
}
```

As usual, don't forget to bind their `this` contexts within the App’s `constructor()` method.

```
this.damperPedalDown = this.damperPedalDown.bind(this);
this.damperPedalUp = this.damperPedalUp.bind(this);
```

The `damperPedalDown()` method simply sets the `state` property to indicate that the damper pedal is down, whereas the `damperPedalUp()` resets the state property and also sets all notes' dynamic values to ‘none' — the `damperPedalUp()` method effectively stops all currently playing notes.

Next, we should be able to control the damper pedal using the space bar. Add the following `componentDidMount()` method to the App class.

```
componentDidMount() {
  document.addEventListener('keydown', (e) => {
    if (e.keyCode === 32) {
      this.damperPedalDown();
    }
  });
  document.addEventListener('keyup', (e) => {
    if (e.keyCode === 32) {
      this.damperPedalUp();
    }
  });
}
```

If you have the ReactJS Chrome Browser extension installed, you can actually inspect and view the `'damperPedalDown'` state dynamically as you press and release the spacebar. Give it a try!

We should add some logic to our `handleKeyUp()` method so that notes are not stopped if the damper pedal is released.

```
handleKeyUp (pianoKey) {
  console.log('key is raised');
  if (!this.state.damperPedalDown) {
    this.setDynamicForNotes([pianoKey.note], 'none');
  }
}
```

Also, while we are here, let's modify our `handleKeyDown()` method so that the `noteDisplay` state is set to the current note:

```
handleKeyDown (pianoKey) {
  console.log('key is pressed');
  this.setDynamicForNotes([pianoKey.note]);
  this.setState({
    noteDisplay: pianoKey.note
  });
}
```

Okay, so now we have the mechanics of the piano down-pat. Let's play some music!

Downloading a Song List from an API

Our Piano should enable the user to select from an increasingly larger range of songs from Tim's Music API. First we will get our app to download a list of songs and display them to the user.

First, we'll add the user interface component to our App.js file, which is a simple select control that shows a list of songs. Add the following before the closing `</div>` tag within the JSX of your App's `render()` method:

```
<div className="ctrl-group">
  <select>
    <option>Select a song ...</option>
  </select>
</div>
```

In our app, we'll be storing 'songs' as part of its state, so add the following song data to the app's state property in its `constructor()`:

```
songs: [
  {id: 2, title: 'Riverside', artist: 'Agnes Obel'},
  {id: 4, title: 'Sky Dancing"', artist: 'Patrick Watson'}
],
```

In a few moments, we'll be replacing this hard-coded data with data fetched from a Web API. We'll modify our JSX so that we'll be displaying the App's 'songs' state property, so change this:

```
<select>
  <option>Select a song ...</option>
</select>
```

to this …

```
<select onChange={this.handleSongSelectOnChange}>
  <option defaultValue>Select Song ... </option>
  {this.state.songs.map((song) => <option key={song.id} value={song.id}>{song.title}</option>)}
</select>
```

You should now see the two songs from our state property as options within the `<select>` control.

Now ideally, rather than hardcode our song selections into the app, we'll be fetching this information from Tim's Music API. This is so that when Tim adds new songs to his library, your app will get the most up-to-date list of music. We'll do this by:

a) Fetching music from an API.
b) Setting the 'songs' state of our app to reflect data retrieved from the API.

We'll add the following to the end of our `componentDidMount()` method:

```
fetch('http://itu.dk/people/tiwr/song/list.php')
.then(response => response.json())
.then(result => {
  this.setState({
    songs: result.data
  });
})
.catch(err => console.log(err));
```

So what does this actually do? Let's break this down …

```
fetch('http://itu.dk/people/tiwr/song/list.php')
```

This calls a script on a remote server to give us a list of songs.

```
.then(response => response.json())
```

Once the list is retrieved, the results are then converted into JSON format -- a format that turns textual data from a server into easy to use JavaScript objects.

```
.then(result => {
  this.setState({
    songs: result.data
  });
})
```

This then sets the 'songs' state of your application to the 'data' property of the results as retrieved from the server.

```
.catch(err => console.log(err));
```

Finally, this catches and displays any nasty errors that might come along the way — for example, if your Internet connection drops our while the API request is being made.

Finally, we'll set our `songs` state to an empty array, so that the select dropdown is initially empty at first, and then it is populated with songs once the data is loaded from an API.

```
songs: [],
```

If everything is working fine, you should see a more up-to-date song list in the `<select>` control (and `setState()` magically updates our UI for us).

Preparing and Initialising a Song

Next, we would like the user to actually play a song. To do this, we'll need to:

a) Download the song data (we'll again use `fetch()` and the Music API).
b) Add some state to our application so that we can track the tempo, whether a song is playing or not, and determine what position we are within the song while it is playing.

First of all, lets add some state to our application. We'll add a single empty `currentSong` property to our state, and a `songIsPlaying` property to determine if a song is playing or not:

```
currentSong: {},
songIsPlaying: false,
```

We'll be populating this `currentSong` state property with data retrieved from our music API.

First, we'll bind an event handler to the `<select>` control's `onChange` property

```
<select onChange={this.handleSongSelectOnChange}>
  ...
</select>
```

Next, we'll add a method stub to our `App` class:

```
handleSongSelectOnChange(event) {
  console.log(event.target.value);
}
```

Finally, as usual, don't forget to bind the `this` context to the `handleSongSelectOnChange` event handler within the App's `constructor()` method:

```
this.handleSongSelectOnChange = this.handleSongSelectOnChange.bind(this);
```

Save your file, and test your app by selecting items within the `<select>` control. Do you notice the numbers that appear when you change the song selection? These are the unique IDs that are bound to each song. We'll use these IDs to let our Music API know which song we'll like.

We're going to call the Music API to download the appropriate song data, and set the `currentSong` property of our app's state:

```
handleSongSelectOnChange(event) {
  let id = event.target.value;
  fetch('http://itu.dk/people/tiwr/song/get.php?id=' + id)
  .then(response => response.json())
  .then(result => {
    this.setState({
      currentSong: result.data
    })
  })
  .catch(err => console.log(err));
}
```

If all is working well, you should be able to see `currentSong` update within the application's state when you select the song from the dropdown.

Next, we'll need to track the length of each beat and the current position within the song. Add the following state property to your application:

```
currentSongPosition: 0,
beatDuration: 0,
```

We'll compute the length of each beat with the following function, which we'll add to the App class:

```
getBeatDuration(timeSignature, bpm) {
  switch (timeSignature) {
    case '3/8':
    case '6/8':
    case '9/8':
      return 60000 / bpm / 2;
    case '2/4':
    case '4/4':
    default:
      return 60000 / bpm;
  }
}
```

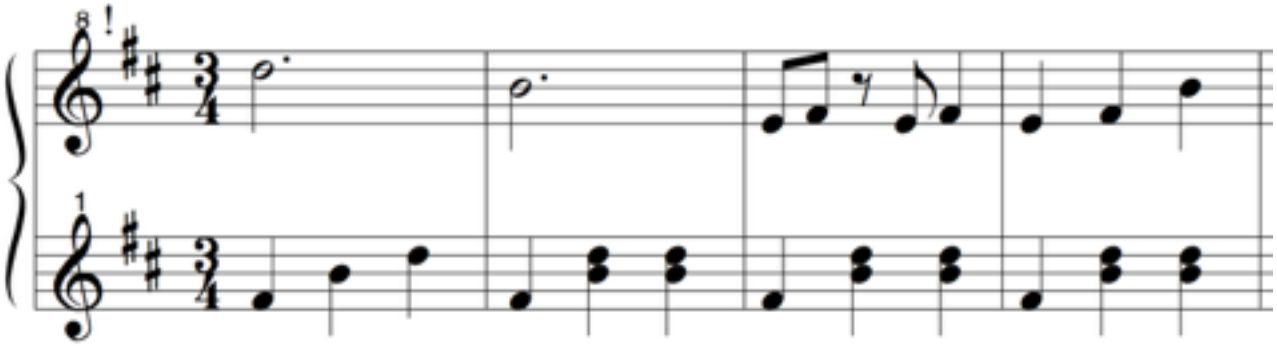
In our `handleSongSelectOnChange()` method, we'll set `beatDuration` by calling the getBeatDuration() method. Modify the `this.setState` method within `handleSongSelectOnChange()` to the following:

```
this.setState({
  currentSong: result.data,
  beatDuration: this.getBeatDuration(result.data.info.timeSignature, result.data.info.bpm)
});
```

Finally, once we're all setup, we are ready play.

Playing and Controlling Music Playback

Finally, we'll hear some music. In order to fully understand what we're going to do next, we'll need to have a look at the data structures and processes that are used to encode and play back music. Each song as fetched from the Music API has two parts: an `info` section that lists information such as the title of the song, artist and time signature, and an `actions` section which is an array of 'commands' that we will give to the piano. Here's a sample of 'actions' the first 4 bars from Patrick Watson's 'Sky Dancing' …




```
{"action": "pedal", "state": "down"},
{"action": "chord", "notes": ["Gb4", "D6"], "dynamic": "mf", "duration": 1},
{"action": "chord", "notes": ["B4"], "dynamic": "pp", "duration": 1},
{"action": "chord", "notes": ["D5"], "dynamic": "pp", "duration": 1},
{"action": "pedal", "state": "up"},
{"action": "pedal", "state": "down"},
{"action": "chord", "notes": ["Gb4", "B5"], "dynamic": "mf", "duration": 1},
{"action": "chord", "notes": ["B4", "D5"], "dynamic": "pp", "duration": 1},
{"action": "chord", "notes": ["B4", "D5"], "dynamic": "pp", "duration": 1},
{"action": "pedal", "state": "up"},
{"action": "pedal", "state": "down"},
{"action": "chord", "notes": ["Gb4", "E5"], "dynamic": "mf", "duration": 0.5},
{"action": "chord", "notes": ["Gb5"], "dynamic": "mf", "duration": 0.5},
{"action": "chord", "notes": ["B4", "D5"], "dynamic": "pp", "duration": 0.5},
{"action": "chord", "notes": ["E5"], "dynamic": "mf", "duration": 0.5},
{"action": "chord", "notes": ["B4", "D5", "Gb5"], "dynamic": "mf", "duration": 1},
{"action": "pedal", "state": "up"},
{"action": "pedal", "state": "down"},
{"action": "chord", "notes": ["Gb4", "E5"], "dynamic": "mf", "duration": 1},
{"action": "chord", "notes": ["B4", "D5", "Gb5"], "dynamic": "mf", "duration": 1},
{"action": "chord", "notes": ["B4", "D5", "B5"], "dynamic": "mf", "duration": 1},
{"action": "pedal", "state": "up"},
```


When we play music, we are performing each action in a controlled, timely manner. From looking at the above actions, it is evident that at the beginning of each bar, the pedal is depressed, and at the end of each bar, the pedal is raised. When the pedal is held down, notes and chords will be played -- only to be released again at the end of each bar. This is exactly what we'll be doing in our piano: we'll be issuing it commands, each with a small delay between each command so as to create the magic of music.

We'll need to add a couple of methods to our App class in order to control and track playback.

```
playCurrentSong() {
  console.log('song is playing');
}

performNextAction() {
  console.log('running next action');
}

stopCurrentSong() {
  console.log('song is stopped');
}
```

Don't forget to bind these in our App's `constructor()` method:

```
this.playCurrentSong = this.playCurrentSong.bind(this);
this.performNextAction = this.performNextAction.bind(this);
this.stopCurrentSong = this.stopCurrentSong.bind(this);
```

Next, we'll add the following JSX to our App's `render()` method:

```
<div className="ctrl-group">
  <button onClick={this.playCurrentSong} disabled={!this.state.currentSong.info}>Play Song</button>
  <button onClick={this.stopCurrentSong} disabled={!this.state.currentSong.info}>Stop Song</button>
</div>
```

When the user clicks on 'Play Song', the `playCurrentSong()` method will fire. This will initiate the `performNextAction()` method, and the `performNextAction()` will call itself (with a timer) until the song is stopped, or until the song finishes.

```
playCurrentSong() {
  if (!this.state.songIsPlaying) {
    this.setState({
      songIsPlaying: true,
      currentSongPosition: 0,
      noteDisplay: this.state.currentSong.info.title
    });
    this.performNextAction();
  }
}

performNextAction() {
  let currentAction = this.state.currentSong.actions[this.state.currentSongPosition];
  let delay = 0;
  if (currentAction.action === 'chord') {
    this.setDynamicForNotes(currentAction.notes, 'none');
    this.setDynamicForNotes(currentAction.notes, currentAction.dynamic);
    delay = currentAction.duration * this.state.beatDuration;
  }
  if (currentAction.action === 'pedal') {
    if (currentAction.state === 'down') {
      this.damperPedalDown();
    }
    if (currentAction.state === 'up') {
      this.damperPedalUp();
    }
  }
  if (this.state.currentSongPosition < (this.state.currentSong.actions.length - 1)) {
    setTimeout(() => {
      if (this.state.songIsPlaying) {
        this.setState({
          currentSongPosition: this.state.currentSongPosition + 1
        });
        this.performNextAction();
      }
    }, delay);
  } else {
    this.setState({
      songIsPlaying: false,
      currentSongPosition: 0,
    });
  }
}

stopCurrentSong() {
  this.setState({
    songIsPlaying: false,
    currentSongPosition: 0,
    noteDisplay: ''
  });
  this.damperPedalUp();
}
```

If you have gotten this far, well done! You have successfully created a fairly sophisticated React application that has the capability of playing recorded music. In the coming weeks, the Music API will be expanded so that existing songs will be completed and new songs will be added.

Also, if you have any song requests you would like added to the API (ideally with sheet music if you could find it), feel free to e-mail me. If I have time I could add it to the API.