Using React & Redux on a WordPress Site

The contents of this workbook represent a much less complex, but still very much applicable, version of a project I was asked to undertake years ago for a client.

There was an old WordPress site which many individuals in the business were used to using. I needed to add a bunch of additional functionality but I wasn't allowed to replace much of the WordPress functionality or change the way the site was deployed.

I found the only way to do this was to essentially build a React / Redux application on top of the WordPress site so that it looked like the original design. In reality, there's two or more applications working together.

This workbook demonstrates the key concepts of what I learned and how I decided to implement them in the final product, and I look forward to sharing them with you.

This workbook is split into four sections, where each section builds upon the result of the previous one.

The first and most important one is where you'll learn to **add multiple independent React components to a WordPress page**.

The goal of this is to demonstrate how React can be used to attach to the existing WordPress content and do additional actions with it that aren't managed by WordPress or plugins.

Following this, you'll **connect all React components on a WordPress page to a Redux application.**

By using Redux in this way, all of the independent React components will be connected together into a single Redux application. This is a nontraditional way to use React and Redux together since lots of the established patterns follow more of the single-page-app methodology.

Next, you'll learn how to **use the Redux application along with the WordPress REST API.**

Connecting to the WordPress REST API will provide the capability to perform additional actions on WordPress data without having to do things like refresh the page or integrate WordPress plugins.

Lastly, you'll add functionality to **use the Redux application to asynchronously load WordPress page templates.**

This is similar to the previous section, however instead of accessing the REST API, you're going to be reloading only parts of a page template, rather than entire pages. Think of this as merging both the Redux application and WordPress stacks together to essentially form one large single-page-application.

What you'll need

1. WordPress Environment

Since this workbook focuses on integrating React with WordPress, having access to a WordPress environment is required.

2. Have NodeJS installed

An environment that will be able to build and run Javascript is necessary for converting React code into something that can run in a web browser.

3. Text Editor

This can be anything. Vim, Atom, Sublime Text, heck even NotePad.

4. Basic WordPress / React Knowledge

Any familiarity with WordPress and React is encouraged to make following along with the examples easier, but it's not required.

Project Setup

Referencing the GitHub Examples

In the following sections, each code snippet or some particular paragraph might reference a file in the workbook Github repo.

When that happens, a file bar across will appear fixed to the top of the current workbook page.

This visual aid should help indicate the specific file referenced by the workbook material or by the complete version of the code snippet. This is done to provide a guide as progress is made through the workbook maintaining focus on the specific material and limiting confusion as much as possible.

Any code snippets relevant to a page in the workbook will display like this:

```
function announceToTheWorld() {
  alert('Hello World!')
}

announceToTheWorld()
```

WordPress Setup

The main thing WordPress needs to manage here is loading (at least one) Javascript files written later on kbook.

One way to think of it is, WordPress is being used as an application server to load a psuedo-app on top of the already established WordPress core page templates.

When all is said and done, WordPress will still function the same way, only with some additional functionality managed by some creative Javascript.

To accomplish all of this, all changes will be done in the `functions.php` file, preferably at the theme or plugin level.

All of the WordPress examples kbook will be done in a child theme based on the WordPress TwentyTwenty theme. There are many reasons to use a child theme, but the purposes of this workbook it will provide easy access to creating new hooks in the WordPress lifecycle.

Instead of regurgitating the steps for setting this up, the official WordPress process is outlined here.

Javascript Project Setup

For the Javascript portion of this workbook, npm will be used to manage package dependencies. npm is the most common package manager out there with a very mature and active community.

Please feel free to use other package managers if they are more familiar to you. The important part of this this stage of the workbook is to make sure you are able to install the necessary packages.

If you haven't already, make sure to initialize your project with `npm init`. Once that is done, the next step is to install all of the frameworks and utilities.

The most important of package needed is React and its peer, ReactDOM.

```
npm install --save react react-dom
```

Since nothing in the modern web development world is easy, in order to run your complex Javascript in a modern browser, it needs to be bundled into a single file able to be included by the web browser.

At the end of the Javascript section, all of the Javascript code will need to be combined into a single file called a bundle. Webpack is one tool created to handle this process and is what this workbook will use.

In order to use Webpack, we need to install the core libraries, as well as some code to be able to call it from the command line:

```
npm install --save webpack webpack-cli
```

Let's Add React to WordPress

OK, now to get into the good stuff. It's time to write some Javascript and PHP code so that we can display React components on our WordPress site.

First, the dynamic page components will be created using Javascript and JSX. This will be what looks at the HTML code created by WordPress and modifies it to add the dynamic experience of what a single-page-application is normally responsible for.

After that, the necessary PHP code will be added to the WordPress install. What needs to happen here is WordPress needs to be configured to serve the new Javascript code written above.

Our dynamic React component will change the font color when it gets clicked. Not super complex but the main goal here is to demonstrate the process that goes into a design such as this.

The React hook `useState` is used to store the color that will then be placed into the style prop of the component itself.

Calling the function `getRandomColor` will generate a random HEX color for us when the component initializes and when a click event occurs.

```
function getRandomColor() {
  return Math.floor(
    Math.random() * 16777215
  ).toString(16)
}
```

```
import React, { useState } from 'react'

const DynamicPostTitle = (p) => {
  const [color, setColor] = useState(
    getRandomColor()
  )
  return (
    <div
      onClick={() => setColor(getRandomColor())}
      style={{ color }}
    >
      {p.children}
    </div>
  )
)
```

Our dynamic React component should reuse the existing WordPress post title, so the target selector must first be found in order for React to properly "mount" to the DOM.

Using the TwentyTwenty theme, we locate the selector at `.entry-content h2` and use it our code as below.

(Illustrate target)

```
import ReactDOM from 'react-dom'

...

const component = document.querySelector('.entry-content h2')
if(component !== null) {
  const title = component.innerHTML
  ReactDOM.render(
    <DynamicPostTitle>
      {title}
    </DynamicPostTitle>,
    component
  )
}
```

With the dynamic React component written, it needs to be included in WordPress.

For this step, the Javascript code that was written needs to be bundled into a single file. Once this is exists, the Javascript file can be referenced in a `<script></script>` tag in a WordPress template.

Now that all the React code is written, it's time to create the Javascript bundle that will be loaded by WordPress to include it in the `<head>` assets.

With Webpack, I used a super basic `webpack.config.js` file that I copy-pasted from the web. Nothing super fancy and gets the job done of creating a Javascript file that will run on an HTML page.

```
{
    entry: 'index.js',
    output: {
        path: path.resolve(__dirname, 'js'),
        filename: 'bundle.js'
    }
}
```

In npm, a simple build script needs to be added before a command like this can be executed:.

```
npm run build
```

Once that script's added, the script just needs to be ran on the command line and the Javascript bundle should be saved to where WordPress can load it.

```
{
  ...
  "scripts": {
      ...
      "build": "webpack --mode production",
      ...
  },
  ...
}
```

In order for our React / Redux code to run, we need to tell WordPress to load it how to find our Javascript file so it can be included in the page DOM.

We'll accomplish through the usage of the WordPress actions and the wp_enqueue_script function.

All this code does is utilize WordPress to include our Javascript bundle and place a `<script>` tag on any WordPress page in the theme.

```
// Execute the `workbook_add_js` function on the `wp_head` WordPress `wp_he
add_action('wp_head', 'workbook_add_js');

//
function workbook_add_js() {
  wp_enqueue_script(
    'workbook_js_bundle',
    get_template_directory_uri() . '/js/bundle.js',
    array(),
    '1.0.0',
    true
  );
}
```

OK so let's recap:

1. Dynamic React component created
2. Javascript bundle file created
3. WordPress Javascript asset included in the page template

We should have all of the things necessary to show the new dynamic component in action in WordPress!

Let's load up the browser...

(image of browser)

Connect React Components with Redux

Let's take this a step further and add multiple React components that can communicate with each other. In order for this to be possible, we need to introduce Redux.

Since our WordPress site is served by Apache, the DOM isn't something that our React app can directly control. With Redux, we can attach a store and an application context to every React component. This is going to allow us to handle actions one component to affect what happens on a different one.

We're also going to expand outside just one page, and support multiple pages on our WordPress site.

The steps to take to create this application are the following:

1. Create a Redux Provider

With the knowledge of how Redux works, a provider will be created to link multiple React components together and allow them to communicate with each other.

2. Update Blog Index page

Once the Redux application is created, all of the summarized blog posts on the index page can now be controlled by React.

3. Update Blog Post page

With all of the work done in the previous steps, some small tweaks to the existing blog post page implementation will connect this React component to the Redux application.

An initial setup task is to update the npm package dependencies to include the Redux library. Using npm, it's installed into the project using the following command:

```
npm install --save redux@4.0.2
```

We'll also need to tweak out DynamicPostTitle component for it to be able to communicate.

First, some basic imports are added to bring in some Redux functionality, both as part of the Redux library itself and some custom that will be written later on.

Lastly, the export identifier is changed to connect the component to a Redux Container and attach it to a Redux reducer.

```
import { connect } from 'react-redux'
import * as Actions from 'redux-actions'
```

```
const mapStateToProps = (state) => {
  return state
}

export default connect(mapStateToProps)(DynamicPostTitle)
```

We'll also make the component much "dumber" and have it just take information from the reducer. We'll keep the `getRandomColor` function where it is but change how the click event handler works. The React lifecycle management code can go away since the reducer is managing this for us.

Once our changes are done, the guts of the `DynamicPostTitle` component look like this:

```
import Actions from '../redux/Actions'

const DynamicPostTitle = (p) => (
  <div
    onClick={() => p.dispatch(
      Actions.setColor(getRandomColor())
    )}
    style={{ color: p.color }}
  >
    {p.children}
  </div>
)
```

Only a single Redux action is used in this example, and this is the file that's referenced the updated React component.

This action will be used by the React component to trigger an event to be recorded in the Redux reducer which will update the color.

The Redux action type could be extracted to a constant, however for the purposes of this example it's not necessary.

```
export const setColor = (color) => ({
  color,
  type: 'SET_COLOR'
})
```

Now that the we've updated our component and created the Redux action, the only remaining thing to do is create the reducer to manage the state of the current font color.

We'll create a file which contains our store, and implements our reducer.

```
import { createStore } from 'redux'

export const reducer = (state, action) => {

  if(action.type === 'SET_COLOR') {
    return {
      color: action.color
      ...state,
    }
  }

  return {
    ...state
  }
}

const store = createStore(reducer, {
  color: '#000000'
})

export default store
```

Lastly, we're going to modify our entry point file and add our new Redux store along with some other basic Redux code to complete the application changes.

To do this we need to import some new packages and variables from our new Redux files:

Notice the `DynamicPostTitle` component is being wrapped in a Redux Provider. This essentially "provides" the Redux context to any children React components alongside any other props already being passed to the component.

```
import { Provider } from 'react-redux'
import store from 'redux-store'
```

```
const components = docuement.querySelectorAll()

for (let i = 0; i < components.length; i++) {
  const title = component.innerHTML
  ReactDOM.render(
    <Provider store={store}>
      <DynamicPostTitle>
        {title}
      </DynamicPostTitle>
    </Provider>,
    component
  )
}
```

With these changes, complete, it's time to rebuild everything and watch it work.

You should be able to click any post title and observe all of them change color at the same time.

(preview interaction)

Integrating WordPress REST API into Redux

COMING

SOON

Rendering partial WordPress templates with Redux

COMING

SOON