ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEDPR04

THE WEB AS THE PRIMARY CONTROL SYSTEM USER INTERFACE*

R. Neswold†, B. Harrison‡, Fermilab, Batavia, USA

Abstract

Fermilab's Control System uses a proprietary application framework written decades ago. Considered state-of-the-art at one time, the control system now lacks many features we expect from a modern interface and needs to be updated. Our investigation of Web browsers and JavaScript revealed a powerful, rich, and state-of-the-art development environment. We discuss JavaScript frameworks, JavaScript language features, and packaging tools. We also discuss issues we need to resolve before we are confident this can become our primary application platform.

INTRODUCTION

We set out to reimagine accelerator control applications. Exploring modern development tools and current best practices help us move away from existing, aging dependencies while improving the users' experience.

Fermilab's parameter page application is broadly considered to be the workhorse of the control room. The parameter page allows operators to freely request live data from any device in the control system. Key features include the ability to manipulate devices, query the control system for meta-information about a given device, plot the data over time, and provide textual context for the set of devices. The parameter page application also allows users to save their set of queries and notes for easy retrieval and, therefore, has become a simple way for machine experts to create basic applications. Many other applications are structured views of data that allow the user to read and manipulate data in a predefined, restricted, way.

We found these features in common with modern dashboard applications. Dashboards offer a series of configurable panels that can be added to a view and saved for later. Views are for a specific task or related data. Dashboards in the browser allow for easy shared access to saved views. Browsers also provide many accessibility features that would require lots of effort to implement in traditional applications. We can consider JavaScript applications so readily because we have an existing client library that allows for streaming accelerator data from the control system to JavaScript via WebSockets.

Identifying the dashboard's component-like structure led us to investigate web application frameworks that support self-contained and reusable code. We aim to provide operators with a blank canvas and the ability to intuitively add new panels with standard components. They then save this view and get a unique endpoint that they can return to in the future. We hope that this component-based design encourages developers to reuse and modify code rather than reimplement very similar features over many applications. In our investigation, we looked at framework maturity, adoption, and documentation. While others like Angular, VueJS, and Java fulfilled the requirements, ReactJS's ubiquity, quality documentation, and ease of use made it stand shoulders above.

* This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.
† neswold@fnal.gov
‡ beau@fnal.gov

INVESTIGATING REACT

React[1] is a JavaScript library used to create "components" along with an engine that efficiently renders changes in the DOM. A component is a JavaScript module that renders HTML elements and manages the state associated with them. Each component is self-contained; the outside state is provided when the component is created, but from that point on, the component updates its own, internal state. Since components are insulated from external effects, they can easily be combined to make more complicated components. React-based applications are nothing more than a series of nested components.

The React team provides a command-line tool to help set up a new React project called `create-react-app`. This tool creates a directory tree containing initial, sample JavaScript source along with the necessary configuration files to build your application. It also sets up an area used for creating unit tests for your project. As your application grows, tests should be added to make sure previously working features still work. The build environment includes another powerful feature where, after successfully building the project, a web server is launched to run your application. The server listens on the localhost address and opens a tab in the default browser on your desktop displaying your application.

React projects typically use an extension to JavaScript called JSX[2] which makes the rendering code much easier to understand. JSX allows you to use HTML-style tags in your JavaScript code, rather than the explicit function calls it takes to create the elements. Files using JSX notation have the file extension `.jsx`. When building the application, files with this extension are processed by converting any JSX notation into calls to `createElement()` rendering a `.js` file. The resulting file has the normal `.js` extension and can be loaded by the browser.

Aside from the component hierarchy, applications also require some logic to manage global state, interface to 3rd party libraries, and even to pass state between components. This logic can get complicated, and due to JavaScript's dynamic typing, simple mistakes aren't necessarily caught until the code path is executed, resulting in run-time errors. Fortunately, we found that we could use Microsoft's TypeScript language with our React projects, which eliminated a whole class of bugs in our code and helped speed up our development.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEDPR04


```
 1 import React, { useState } from 'react'
 2 import './ReactiveInput.css'
 3
 4 interface ReactiveInputProps {
 5     label: string,
 6     maxLength?: number
 7 }
 8
 9 const ReactiveInput: React.FunctionComponent<ReactiveInputProps> =
10     ({ label, maxLength = Infinity }) => {
11     const [currInput, setCurrInput] = useState('');
12
13     return (
14         <div className='reactiveInput'>
15             <label htmlFor='reactiveInput'>{label}</label>
16             <input
17                 type='text'
18                 name='reactiveInput'
19                 value={currInput}
20                 onChange={(event) => {setCurrInput(event.target.value)}}
21             />
22             <p className={currInput.length > maxLength ? 'invalid' : ''}>
23                 {currInput}
24             </p>
25         </div>
26     );
27 }
28
29 export default ReactiveInput;
```

Figure 1: Simple example of a react component.

TypeScript[3] is a free, open-source transpiler created by Microsoft which adds static type-checking to JavaScript. It follows most of JavaScript's syntax, except that function parameters and variables, are annotated with type specifications. TypeScript also introduces useful data types (like tuples), which JavaScript doesn't have, allowing you to write more expressive code.

Before the browser can run the code, the TypeScript source (`.ts`) is transpiled to JavaScript where type annotations are removed, and non-standard data types are converted to JavaScript counterparts. Although the end product is dynamically-typed JavaScript, the code has gone through extensive type analysis resulting in much fewer run-time errors. We were also happy to see that many 3rd-party libraries include a TypeScript declaration file so that you benefit from strict type-checking while using their API. We found that using TypeScript improved our productivity, even with simple examples, because we spent less time in the debugger analyzing exceptions caused by typing errors.

TypeScript can transpile to older versions of JavaScript, too, so, if you need to use an older browser, it can generate JavaScript that still implements modern behavior but uses older JavaScript features to implement them (of course, the resulting code is larger.) TypeScript also supports JSX syntax, so React source benefits from JSX's notation and TypeScript's code analysis.

Figure 1 shows an example of a simple React component. This component renders as a label, a text input field, and a text area to display processed output. Lines 4-7 is an example of TypeScript defining the layout of an anonymous JavaScript object. Line 9 shows how to attach type annotation to a variable. In this case, we're using a class name that uses generics to specify what type is used when passing in the component's properties. This function returns an HTML element that gets rendered on the web page. Lines 14-25 use JSX notation to specify the top-level element and its nested child elements. This notation is much easier to understand than the JavaScript functions calls required to create the elements. Note also, in the JSX syntax, that we inject the value of variables into the expanded output by putting the variable name in curly braces. Line 11 shows how to allocate state which is used each time the function is called. The function `useState()` returns the current state and a function to call to update the state. React uses the update function to track when a component's state changes so it can determine which subset of the page needs re-rendering. In this example, as the text is added, it is copied to the paragraph element. When the length of the string reaches `maxLength` characters, it gets displayed in red, rather than black.

The default HTML page for a React app is mostly empty. The body element typically contains a single div element with the ID of "root." To get everything started, the JavaScript starting function would, in the example, contain the following call:

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEDPR04

```
ReactDOM.render(
    <ReactiveInput
        label='Hello ICALEPCS 2019'
        maxLength={10} />,
    document.getElementById('root')
);
```

Figure 2 shows the resulting web page, with sample text, already entered.

Building React Projects

When working with JavaScript, the tool that builds your project manages dependencies, runs tests, and deploys your application is npm[4]. npm was developed for NodeJS as their package manager and has evolved to include tools useful for client-side development. You use npm to install global tools like TypeScript (`npm install -g typescript`) or the project creation script for React (`npm install -g create-react-app`). npm also installs 3rd-party libraries used by your project, along with their dependencies.

The chain of events that build and prep your application for deployment begins with "`npm run build`". This command runs a series of tools that are unique to a browser's model of loading code. To run your application in the browser's JavaScript interpreter, it needs to see all the JavaScript source used in an application; hence, there's no compile/link cycle. Instead, npm runs the project through a series of tools that try to produce the smallest sized JavaScript possible to shorten the load-time of your application. It also runs tools that rewrite portions of your code to maximize compatibility with your set of targeted browsers.

In our React applications, the first tool used is the TypeScript transpiler. As mentioned earlier, this step does extensive code validation based on the added type annotation. The validated code is emitted as JavaScript and is passed to the next tool, Babel.

Babel[5] is a transpiler, like TypeScript, but its mission is to rewrite -- if necessary -- the JavaScript so that it can run on the set browsers you specify. If your site needs to run on older browsers, you would add that requirement to the Babel configuration, and Babel would make sure that, no matter what advanced features of JavaScript you use, it'll run on that browser. Of course, if you target more recent versions of browsers, Babel passes more of your code through, untouched.

Babel supports plugins, and many are available in case you need other translations than compatibility. In our React projects, Babel is used to translate the JSX syntax into JavaScript.

Once your project's code has been prepped, the last tool invoked is WebPack[6]. WebPack's primary purpose is to combine all your source, and the source of 3rd party libraries used by your project, into one, large source file. As can be imagined, this could end up being quite large and, therefore, WebPack's secondary purpose is to make the source code as small as possible. It does this using several strategies. First, it makes sure that, if several modules import a library, it only gets included once. Next, it makes a pass through the source and removes code that isn't used (i.e., dead-code elimination.) These first steps can significantly reduce the final size of the project, but WebPack performs one more translation, called "minify." In the "minification" step, all extraneous whitespace (and comments) are removed, and identifiers (like variable and function names) are shortened. Once the build is complete, the project's source isn't human-readable, but it's easily read and run by a browser and is much smaller than the original body of source.

Figure 2: Sample output from react example.

Developer Tools

At the time of this conference, three major browser cores have emerged: Mozilla's "Gecko" (used in Firefox), Apple's WebKit (used in Safari), and Google's ChromeKit (used in Chrome and Microsoft's Edge browser.) Each core closely follows and implements the latest Web Standards. Each core also has an advanced JavaScript engine that uses a just-in-time (JIT) compiler to speed up hot-spots in your code. Also, most importantly, all these browsers come with a rich suite of developer tools. Usually tucked away in a menu, selecting the developer tools splits the current web page into two panes: one contains the rendered web page and the other displays the interface of the tools.

The debugger allows setting breakpoints, single-stepping through code, examining variables, and viewing the stack, as you would expect. It is more than a source code debugger, however. It also allows navigating the DOM tree shown in the web page; as you move the mouse cursor over DOM tree elements, regions of the web page are highlighted in real-time. Selecting elements in this view displays their CSS attributes in a side panel. Changes to these attributes are reflected immediately on the rendered document allowing you to try tweaks to the page without going through the build cycle.

The developer tools also include several profilers. Once the application has run through the code profiler, the source code view is annotated to show the hot spots. A "flame view" is also available to show the call stack and how much time is taken at each level. The code profiler also includes timing information for the GPU, which measures the rendering time of the page. There's a memory profiler to observe heap usage which can detect memory leaks or see if your code is invoking too many garbage collections. A network profiler shows how long it takes to load each resource in your application.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEDPR04

Aside from helping one to find and fix software bugs, these tools help streamline your code and minimize load times. We were delighted with the breadth of development covered by these tools.

Examples

Throughout our investigation, we found React easy to understand and use. React's component model isolates its logic from the rest of the application's so you feel confident in the results. Through our efforts, we developed a handful of example applications, each focusing on a perceived need for our department. In doing so, we became aware of several excellent, 3rd party libraries.

Material Design By recommending a framework, like React, we get consistency in the programming model. However, web programming also requires layout design and style guidelines, and so we need to address styling consistency. Specifically, we don't want each application to have a different look and feel.

Google has produced a set of guidelines, based on their research and experience, called Material Design[7]. Google uses them on the Android platform and web interfaces, and they closely describe other mobile platforms' interfaces. By following these guidelines, our applications will look attractive and consistent and will feel familiar and intuitive to our users. The npm repository has several React libraries that make it easy to follow these guides. We've only started using these libraries, but we feel it's a quick way to get our custom components to look professional.

Nivo Charts Plotting data is a staple in control's software, so we created React components that wrapped JavaScript chart libraries. One library, in particular, stood out for both its polished presentation of data as well as how many chart types it supports. The library is called Nivo Charts[8], and it lets you visualize data in many ways. It supports over 20 different types of charts, and each type has a set of variations. Their library can render most charts in two ways: using the HTML canvas element or generating SVG elements. Each has pros and cons, so the choice depends on the features you need and how much data you have to display. Charts rendered to SVG show additional information when the mouse cursor hovers over it. Charts using the canvas element don't have this feature. However, the canvas-based charts can handle lots of data points quickly, whereas SVG charts should restrict the quantity of data to keep it responsive.

WebGL Pushing our investigation further, we wondered how easy would it be to model our data using 3D graphics. We felt there might be situations where it would be useful to visualize what's going on in the machine, and we wanted to see what it would take to do this. We were pleased to find ThreeJS[9], a JavaScript library that wraps the WebGL API into an API that's easier to use. Initially, we created 3D objects programmatically but building more complicated models seemed daunting. Fortunately, ThreeJS can import model files in several formats. We used Blender3D to create a motion control station model and then exported the data. Our web application was able to read in the model, render it, and control it.

Progressive Web Apps Our last investigation was to create a "progressive web app" (PWA). "Progressive Web Apps provide an installable, app-like experience on desktop and mobile that are built and delivered directly via the web. They're web apps that are fast and reliable"[10].

Progressive Web Apps are attractive because they have the look and feel of a native application that runs on any platform, and we only wrote one application. The essential requirement for a progressive web app is to include a manifest file that tells the browser how to display the application and where to find the icon set. Google's PWA checklist provides details on what features are to be implemented to be considered a PWA. Chrome also has a built-in auditing tool, Lighthouse, to evaluate your adherence to the PWA standards.

The major hurdle for most web applications not feeling native is the fact that they don't display and have limited interactions when you are offline. PWAs cache useful resources allowing users to interact with the application even when they offline. We plan to conform to this PWA standard and imagine it could be useful to allow users to change configurations or refer to notes when offline.

Remaining Issues

Although our investigation showed great promise, there are still several details we need to resolve before committing to web applications. One important detail relates to deployment. How do we organize the apps on the webserver? How do new applications get added to this namespace? Is there a main page with links to each application? Do we create a categorization system to find an app easier?

Another concern is about version management. npm allows a project to track the latest versions of the libraries it uses. In the short time that we've been developing, we've seen quite a few patch-level updates. It may make sense to maintain the framework (React, in our case) as a separate resource to download and not require applications to include it in their web bundle. This would allow us to keep the framework up-to-date, and applications wouldn't have to be rebuilt each time a critical update is released.

CONCLUSION

Modern browsers provide a powerful and compelling environment for hosting acceleration applications. We are convinced that using web apps is the direction our department should take. There are comprehensive development tools already in browsers to handle all aspects of web development. Frameworks provide a professional, intuitive experience for users, and they hide browser differences from programmers to the point that they also work on mobile devices. Some frameworks also seamlessly support progressive web apps, so mobile users feel they're running native apps. Tools, like TypeScript and JSX, move many run-time issues to compile-time, making it easier to produce correct code. Most importantly, all these technologies are backed by huge companies (Google, Apple, Microsoft, Facebook) that have a stake in the success of the web.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEDPR04

REFERENCES

[1] React, a JavaScript library for building user interfaces, https://reactjs.org/

[2] SX. XML-like syntax extension to ECMAScript, https://facebook.github.io/jsx/

[3] TypeScript, JavaScript that scales, https://www.typescriptlang.org/

[4] npm - build amazing things, https://www.npmjs.com/

[5] Babel, The compiler for next generation JavaScript, https://babeljs.io/

[6] Webpack, https://webpack.js.org/

[7] Design - Material Design, https://material.io/design/

[8] Home - nivo, https://nivo.rocks/

[9] three.js - JavaScript 3D library, https://threejs.org/

[10] Progressive Web App Checklist, Google Developers, https://developers.google.com/web/progressive-web-apps/checklist