ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEMPR006

APPLICATION DEVELOPMENT IN THE FACE OF EVOLVING WEB TECHNOLOGIES AT THE NATIONAL IGNITION FACILITY

E. Pernice, C. Albiston, R. Beeler, E. Chou, C. Fry, M. Shor, J. Spears,
D. Speck, A. Thakur, S. West
Lawrence Livermore National Laboratory, Livermore, USA

Abstract

The past decade has seen great advances in web technology, making the browser the de-facto platform for many user applications. Advances in JavaScript, and innovations such as TypeScript, have enabled developers to build large scale applications for the web without sacrificing code maintainability. However, this rapid growth has also been accompanied by turbulence. AngularJS arrived and saw widespread adoption only to be supplanted by Angular 2+ a few years later; meanwhile other JavaScript-based languages and developer tools have proliferated. At the National Ignition Facility (NIF), the Shot Setup Tool (SST) is a large web-based tool for configuring experiments on the NIF that is being developed to replace a legacy Java Swing application. We will present our experience in building SST during this turbulent time, including how we have leveraged TypeScript to greatly enhance code readability and maintainability in a multi-developer team, and our current effort to incrementally migrate from AngularJS to React.

INTRODUCTION

The Campaign Management Tool (CMT) is an application that is used to configure experiments at the NIF. Originally developed as a commissioning tool for the NIF laser, CMT was subsequently put to work as the production experiment editor for NIF experimental operations. It remained under constant development for 15 years supporting the ever-expanding stable of NIF target diagnostics and the ongoing refinement of the NIF laser. However, CMT's architecture was not optimal for the development focus of the program, and it also carried a very steep learning curve for software developers. In 2014 CMT was identified as a bottleneck for shot operations as the NIF sought a dramatic increase in its experiment shot rate. A project was undertaken to address CMT usability and maintainability concerns. These concerns included an outdated technology stack as well as several architectural features that inhibited maintainability. As a result of this effort, we decided a new application was needed to meet programmatic needs [1].

It was critical to select a technology stack with long term sustainability and which followed current computing trends. Since the initial development of CMT, a highly interactive desktop application, web browser-based applications had increased significantly in capability and ubiquity. Model-View-Controller (MVC) frameworks, such as AngularJS and Backbone, allowed much better client-side rendering and supplanted server-side rendering, such as JavaServer Pages (JSP), as the standard for interactive web applications. In order to maintain parity with CMT functionality we decided to build SST as a client rendered web application, on a RESTful back end. In 2015 we began development on SST.

INITIAL TECHNOLOGY STACK

For our initial release of SST, we adopted a handful of front-end technologies that would help us achieve our early goals but that would also give us flexibility to adapt to changes in the ecosystem or evolving requirements.

TypeScript

Typescript [2] was a new and somewhat unproven technology at the time we adopted it. TypeScript overcame some of our largest concerns about moving from a large, Java-heavy application to a large JavaScript-heavy one because it is a superset of JavaScript that adds many features to improve code maintainability and scalability. TypeScript code is transpiled to JavaScript as a build step, which is then deployed with the application.

Compile-time error checking In a vanilla JavaScript application, many types of errors that would have been compile time errors in Java are run-time errors and will only be raised when the offending code is executed. This was a concern for us because a typo in a variable or method name could cause a bug that might not be found until after deployment. Even troubleshooting in a development environment would be a headache. TypeScript addressed this problem directly and effectively by failing during our build cycle when validity errors such as these were detected.

Self-documenting code A type-safe language like Java intrinsically provides certain self-documenting features. Consider a method that takes a ShoppingCart as a parameter and returns a list of Items. In JavaScript, clarity must be achieved by apt variable names or comments, which are less explicit and less reliably correct. TypeScript allows us to provide type annotations, which are checked for correctness during transpilation.

Advanced language features TypeScript provides backward compatible use of features from future JavaScript releases such as classes, decorators, and lambdas. This allowed us to support many browsers. It also provides features exclusive to TypeScript, like interfaces and private members. As a team with mainly Java expertise, the addition of familiar object-oriented constructs was welcome.

IDE support Compared to Java, IDE support is very limited for JavaScript. TypeScript enables excellent support for IDE functions like auto-completion, refactoring automation, and searching for method or variable references.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEMPR006

Jasmine/Karma/PhantomJS

A major factor in our inability to meet escalating programmatic needs with CMT was its lack of unit tests. This made changes to CMT risky because small changes often had unintended consequences. We knew that it would be critical to make unit testing an early focus in SST in order to help protect against the risks and costs associated with buggy code and enable long-term maintainability.

To achieve our unit testing goals on the front-end we used Jasmine [3], a popular assertion library. In order to run our Jasmine tests, we used Karma [4] as a test runner, and PhantomJS [5] as a headless browser, which enabled testing in a continuous integration environment.

SystemJS

SystemJS [6] is a module loader that supports asynchronously fetching modules from the server. It worked with TypeScript out of the box, was simple to configure, and it was recommended by Google for use in Angular 2.

JQuery

As the demand for more interactive browser applications has increased over the years, JQuery [7] has served as a useful bridge between server and client rendered applications. JQuery provides a convenient way for a server rendered application to do DOM manipulation on the client while leaving the heavy lifting up to the server. Although it is not well suited for large client rendered applications, it was a powerful tool for our occasional needs to directly update the DOM.

AngularJS

AngularJS [8] is a client-side MVC framework that was widely adopted by web developers in the early development of SST. It enables client-side rendering of templates and single page application routing. It had many useful features to aid in development of SST.

Highly interactive With two-way bindings AngularJS offered an easy way to build an interactive, dynamically updating UI. Promise-based digest updates allowed us to interact with our REST API with confidence knowing that any model changes would be rendered.

Structured architecture As an MVC framework, AngularJS provided an architectural skeleton that was helpful for building a large client-side application. Out of the box, it provided most of what we needed, such as client-side routing, dependency injection, and a promise-based HTTP service.

Long-term stability AngularJS was backed by Google, and widely used by the web development community. Angular 2.0 was on the horizon but hadn't been released yet. Though React had emerged as a potential up-and-comer, its long-term stability was not yet established.

LIBRARY MIGRATIONS

Over the past several years, the entire JavaScript ecosystem has been changing rapidly. New modules are being added to NPM (the de-facto repository for JavaScript packages) at an unprecedented rate. Fig. 1 illustrates this, showing the number of packages in the NPM repository over time compared to other popular repositories, like Maven Central. This development focus has led to a steady increase in sophistication of JavaScript-based frameworks and technologies.

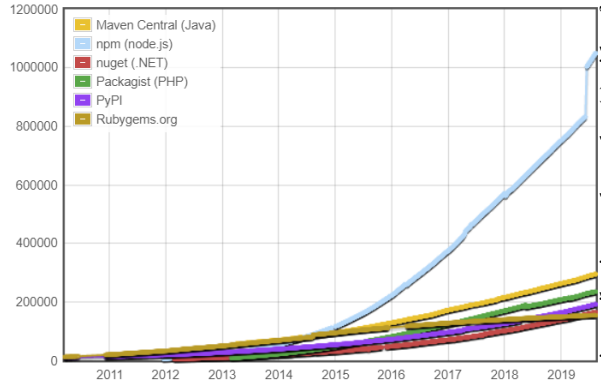

Figure 1: Number of available modules in repositories for various languages, as of 8/12/2019 [9].

These changes have come at a cost. The continuing evolution of front-end technologies necessitates that applications adapt quickly or become legacy. The landscape of front-end development looks very different today than when SST development began, and we have been forced to develop strategies to migrate to different technologies. This is not to say that these changes have been negative. Front-end technologies have not been simply changing for change's sake. Instead, they have been markedly improving. Changes that we have been "forced" to make have been quite positive, improving the overall maintainability, readability, and reliability of our code.

Jasmine/Karma/PhantomJS to Jest

Jest [10] is a fully featured JavaScript unit testing framework developed by Facebook. It has been steadily growing in popularity since 2016 while our previous unit testing stack has been declining, as seen in Fig. 2.

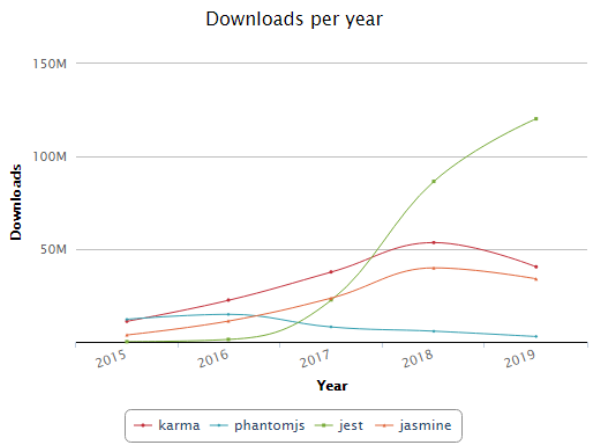

Figure 2: Karma, PhantomJS, Jasmine, and Jest downloads per year via NPM, between January 1, 2015 and July 31, 2019 [11].

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEMPR006

Although our unit testing technology stack was workable, migrating to Jest solved several important problems for us:

- After upgrading other client libraries, we began to experience issues with Karma silently skipping some tests and reporting a successful run.
- Jest is self-contained, and it does not require an additional test-runner or browser. Installing (now discontinued) PhantomJS as a headless browser on our build servers had been a source of frustration.
- Debugging in our IDEs with mapped TypeScript files 'just worked' in Jest. Karma was unreliable in providing similar functionality.
- Jest was far simpler and required less configuration
- Jest added new features such as snapshot testing.

Furthermore, migrating to Jest was painless. Jest uses an assertion syntax that is almost completely compatible with Jasmine. Because our tests were written with Jasmine, Jest worked with minimal configuration, and allowed us to drop Jasmine, Karma, and PhantomJS dependencies.

SystemJS to Webpack

As SST grew in complexity, SystemJS began to strain under its weight. Circular dependencies were fixable but were difficult to troubleshoot in SystemJS. Since SystemJS does not provide any support for bundling or minification out of the box, our application had to asynchronously load hundreds of code files in the browser at application start. Each of these files were fetched in its own HTTP request. This dependency loading delay significantly impacted our performance.

Meanwhile, Webpack [12], along with libraries like ts-loader that augment it with TypeScript support, grew in adoption and maturity, as seen in Fig. 3.

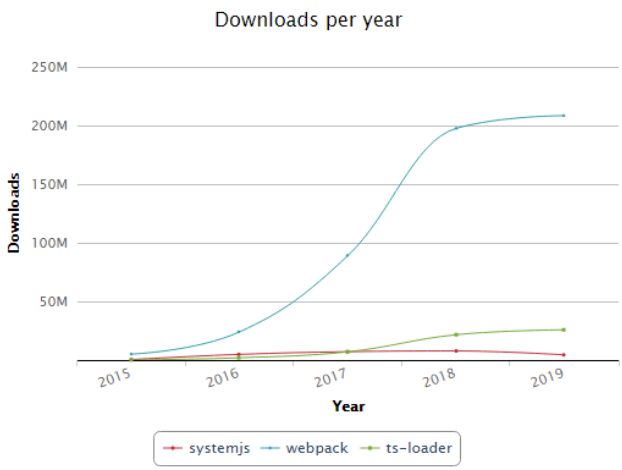

Figure 3: SystemJS, Webpack, and ts-loader downloads per year via NPM, between January 1, 2015 and July 31, 2019 [13].

Webpack enabled us to bundle and minify our client code, significantly cutting down on the number and size of HTTP requests at our application start. This benefit outweighed the drawback of the complexity in configuring it.

MOVING AWAY FROM THE ANGULARJS FRAMEWORK

On July 1, 2018 AngularJS entered a three-year period of Long Term Support [14]. AngularJS had been replaced by Angular [15]. The new version of the technology was not backward compatible with the old. When we began development on SST we had planned to upgrade from AngularJS to Angular once it was released, however, we did not initially appreciate the amount of effort that this would entail. The Angular team had documented some guidelines and practices to help smooth the transition, but no straightforward migration path existed. After an analysis, we determined that the effort required to upgrade to Angular would be comparable to the cost of a framework change. We began to consider the possibility of replacing AngularJS, rather than upgrading it, and began to evaluate React.

React

React initially caught our attention because of its popularity. Although Angular has enjoyed stable growth, React's adoption has been outpacing Angular's consistently in annual NPM downloads, as seen in Fig. 4.

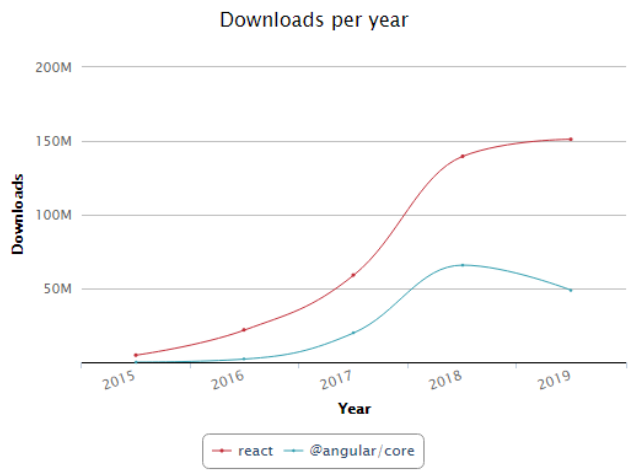

Figure 4: Angular and React downloads per year via NPM, between January 1, 2015 and July 31, 2019 [16].

A more interesting metric for us was React's level of developer satisfaction. Fig. 5 shows the results over the past few years of an annual survey of JavaScript developers with 20,268 developers surveyed in 2018 [17]. While adoption of both Angular and React grew rapidly, the developer community commonly expressed dissatisfaction with Angular.

When we considered the unavoidably major effort of moving from AngularJS, these market indicators made React an interesting candidate for a replacement. After further investigation, several technical features of React drove us to commit to it.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEMPR006

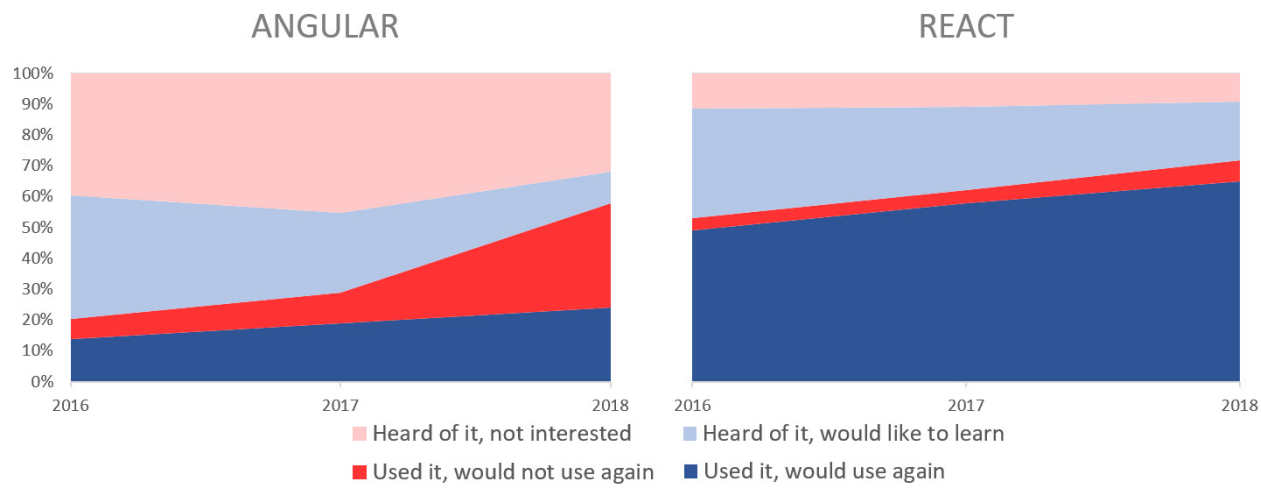

Figure 5: Historic developer satisfaction survey results from State of JS [17].

Just JavaScript Unlike Angular/AngularJS, which have their own templating languages, React view files (JSX/TSX files) are essentially JavaScript. Rendering an array of elements boils down to using vanilla JavaScript constructs like *map()*, rather than using special directives like *ngFor* or *ngRepeat*. Our team has found React easier to reason about and the learning curve has been much lower than it was with AngularJS.

Type checked views One problem that we had with AngularJS was the lack of any type checking in templates. We used TypeScript on the client to ensure safe refactoring, but this only worked to a point. For example, if we renamed a variable that was referenced in an AngularJS template but forgot to update it in the template, there was no transpilation step to catch the error. Angular had the same limitation. In React, however, we were able to simply code our views in TSX files (the TypeScript equivalent of a JSX file). TypeScript was then able to handle type checking for our view code just as well as it did for everything else.

Snapshot testing Snapshot testing is a feature supported by Jest that takes a "snapshot" of a rendered React component. That snapshot is committed to source control and compared on subsequent runs of the test. This approach provides a quick and simple way of testing that the rendered version of a component does not change unexpectedly.

State Management with Mobx

As we explored React further, we quickly encountered a major limitation. State management is a common hurdle for developers working with React. React's state management is clean and simple, which works well a la carte for small applications. As applications grow in complexity and wish to share state between nested components, the need for more sophisticated state management often arises. Redux [18] is perhaps the most widely adopted solution for this, but it is also somewhat heavy handed. Integrating Redux into an already large application like SST would have required significant amounts of re-architecting.

Mobx [19] is a reactive programming library that is also often used to tackle the problem of state management. Mobx allows decorating certain properties as observables and uses generated getters and setters to track their references and mutations. When an observing piece of code, like a React component, references an observable, Mobx tracks this reference as a dependency and updates the observer whenever the observable mutates. This mechanism enables a React component to observe anything in the application and automatically re-render when it mutates. This was an incredibly useful, intuitive and lightweight way for us to manage the state of our React components.

Migration Strategy

A key concern in our evaluation of React was whether we could migrate gradually. Attempting to migrate our entire application in one massive push would have been too large an undertaking. We needed to be able to move away from AngularJS while continuing to release new SST functionality for ongoing programmatic requirements. These demands led us to seek a strategy by which we could migrate AngularJS code at our own pace, and perhaps only when a component needed to be updated.

Fortunately, this was made possible by react2angular [20]. React2Angular is a library that enables using React components from within AngularJS components.

AngularJS components We needed to migrate our 53 AngularJS components to React components. Much of our AngularJS template code was reusable as TSX, and much of our AngularJS controller code was reusable directly in the React component. This compatibility made migrations significantly less costly than full rewrites.

Because react2angular allowed us to use React components from within AngularJS code, these migrations needed to occur from the bottom up. That is, AngularJS components that did not reference any other AngularJS

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEMPR006

components were migrated first. Frequently, our AngularJS components depended on injected AngularJS services. In these situations, we manually injected the dependencies into the React component using the AngularJS *$injector*.

AngularJS services In parallel with the migration of our AngularJS components, we needed to migrate our 20 AngularJS services to wean ourselves off of the AngularJS dependency injection mechanism. We decided to use Inversify [21] for dependency injection. These migrations involved some refactoring, but no code rewriting.

AngularJS core services Our final step, after migrating all of our components and services away from AngularJS, will be to remove our dependencies from core AngularJS services. We have many dependencies on core AngularJS services, such as $timeout, $q, and $http, which we will need to find replacements for. Because these services interact deeply with AngularJS's digest updates, this step cannot not be undertaken until we have no AngularJS components, and thus no reliance on digest updates.

Current Progress

To date we have migrated 36 out of 53 AngularJS components to React, and we have migrated 13 out of 20 AngularJS services to Inversify. We have taken a stance of migrating a piece of AngularJS code either (1) when we need to update the code anyway, or (2) when we cannot parallelize other programmatic work and have developer resources to commit. This has enabled us to work toward this migration at a pace that does not inhibit us from meeting other programmatic needs.

CONCLUSION

The rapid pace of front-end technology evolution over the past several years has been extraordinarily challenging for application developers. It is impossible to predict when or by how much the ecosystem will stabilize, but through this turbulence, we have learned to keep several mitigating factors in mind:

- The JavaScript ecosystem isn't just changing, it is improving, and our adaptation to evolving web technologies has been overwhelmingly positive. Changes we made (or were forced to make, in the case of AngularJS) greatly improved our code reliability and maintainability.
- Many other applications are in the same situation. The web is evolving for everyone, and as widely used technologies exit and new ones enter, the community has stepped up to ease the pain. For example, migrating from AngularJS to React is enabled by community tooling, such as react2angular. And migrating to Jest was incredibly simple for us, due to the syntax compatibility with Jasmine.
- Gradual evolution is helpful. Technical debt grows over time, but by staying abreast of new technologies and updating our dependencies periodically, we can keep our technical debt down gradually rather than having to make larger, more jarring changes.
- Unit testing is critical. Thorough unit testing allows us to evolve with the ecosystem, making changes and refactoring as needed with confidence.

Although change can be costly, this cost can be partially mitigated with diligence and planning, and it is not without a benefit.

ACKNOWLEDGEMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

REFERENCES

[1] A. D. Casey *et al.*, "Strategies for Migrating to a New Experiment Setup Tool at the National Ignition Facility", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, pp. 311-314. doi:10.18429/JACoW-ICALEPCS2017-TUMPL05

[2] TypeScript, https://www.typescriptlang.org

[3] Jasmine, https://jasmine.github.io

[4] Karma, https://karma-runner.github.io/latest/index.html

[5] PhantomJS, https://phantomjs.org

[6] SystemJS, https://github.com/systemjs/systemjs

[7] JQuery, https://jquery.com

[8] AngularJS, https://angularjs.org

[9] Module Counts, http://www.modulecounts.com

[10] Jest, https://jestjs.io

[11] npm-stat: karma, phantomjs, jasmine, jest, https://npm-stat.com/charts.html?package=karma&package=phantomjs&package=jasmine&package=jest&from=2015-01-01&to=2019-07-31

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

[13] npm-stat: systemjs, webpack, ts-loader, https://npm-stat.com/charts.html?package=systemjs&package=webpack&package=ts-loader&from=2015-01-01&to=2019-07-31

[14] Stable AngularJS and Long Term Support, https://blog.angular.io/stable-angularjs-and-long-term-support-7e0776

[15] Angular, https://angular.io

[16] npm-stat: react, @angular/core, https://npm-stat.com/charts.html?package=react&package=%40angular%2Fcore&from=2015-01-01&to=2019-07-31

[17] The State of JavaScript 2018, https://2018.stateofjs.com

[18] Redux, https://redux.js.org

[19] Mobx, https://mobx.js.org

[20] react2angular - npm, https://www.npmjs.com/package/react2angular

[21] Inversify, http://inversify.io