

[oyster.engineering /post/103479627773/half-a-million-books-in-your-browser-how-we-built](https://oyster.engineering/post/103479627773/half-a-million-books-in-your-browser-how-we-built)

Half a million books in your browser: How we built the Oyster Web Reader. Part 2: ES6 & Ember CLI

7-9 minutes


Half a million books in your browser: How we built the Oyster Web Reader. Part 2: ES6 & Ember CLI

This is part two in a series of posts about how we built the Oyster web reader. There are unique and fascinating challenges in bringing books to life in web browsers, and we'll outline some of those in this series: moving from our architecture, forward to our reader application, and then to the UI.

Starting a basic website in 2014: 1. Install Node 2. Install Bower 3. Pick CSS framework 4. Pick responsive approach ... 47. Write some HTML

— I Am Developer (@iamdeveloper)

October 2, 2014

One of the ways we have successfully managed the complexities of modern web applications at Oyster is by choosing technologies that lessen mental burden so that we can focus on the finer details and our specific use cases. Early on, we've been intrigued by ES6 as a way to write clearer, more modular JavaScript. In this post, I highlight the challenges of being one of the early adopters of ES6, and how certain features of ES6 have helped us make the web reader a success. image

ES6 Overview

ES6 contains significant changes to the core syntax and capabilities of the language. These changes include:

- Introduction of classes - While the current version supports function constructors and prototypical inheritance in lieu of classes, the syntax can be downright [confusing and complex](#). To hide this complexity, ES6 provides declarative ways to create classes and class-based inheritance that act as syntactic sugar around the existing pattern
- Arrow (\Rightarrow) syntax for anonymous functions that allow for less verbose lambda function creations. Unlike functions, arrows share the same lexical context (this) as their surrounding code
- Pattern matching for arrays and objects which brings the language closer to python and its functional counterparts
- Generators that allow for functions to suspend execution through the yield keyword and recede control to the callee. This allows for some extremely powerful abstractions for concurrent and asynchronous code beyond the next best alternative of Promises. As an example of the endless possibilities, if you are feeling adventurous, you can read how to implement [Communication Sequential Processes](#) (CSP), an interaction pattern used in concurrent languages, in JavaScript using generators
- Built in support for modules - The JavaScript community is currently split between two different standards of creating modules - the [AMD syntax](#) used commonly in the browser and the [CommonJS syntax](#) used in Node. To resolve the differences between the two standards, ES6 has built in support for modules for both the browser and the server. Modules allow us to organize our code into independent reusable blocks which makes maintenance, organization, and unit testing of components in isolation far more accessible

The aforementioned features had us sold on incorporating some parts of ES6 in the web reader. There was only one problem: no browser had support for ES6 at the time of development. We had to instead

approach the problem by writing code in ES6 and compiling it to ES5 before sending it to the browser.

ES6 Transpilers

A large part of the JavaScript ecosystem today, from languages such as CoffeeScript to ubiquitously used plugins such as jshint or uglifyJS, is based on static analysis of JavaScript. The static analysis is usually done through parsers that build Abstract Syntax Trees (AST), a tree representation of the structure of your code. The most popular parser for JavaScript is [Esprima](#). Such ASTs are widely used within compilers to transform your code from one language to another. The web community, in order to circumvent the lack of ES6 support, has similarly come up with [transpilers](#) that translate ES6 code to ES5 that modern browsers can understand.

We started with Google's Traceur transpiler. It was compiling our ES6 code in a build step but unfortunately we soon realized that some of the [outputted code](#) was hard to understand and debug. The ever-changing spec also created some inconsistencies at that time. Lastly, we had concerns around performance, as [outlined here](#). EmberJS is heavily optimized around compilers such as V8 and as such, we were advised by some of the Ember Core Team members to shy away from Traceur generated code, for both performance and maintainability reasons.

We would love to give Traceur a shot again but at that time we decided to limit our use of ES6 only to the organization of our code into ES6 modules. To that end, we started looking at ES6 transpilers that simply compile ES6 module syntax down to module formats the current generation of browsers can understand, while leaving the rest of the source code intact.

es6-module-transpiler

Esprima is a popular open source parser for JavaScript that has paved the way for transpilers that compile ES6 modules down to AMD as well as CommonJS formats. One such transpiler is the [es6-module-transpiler](#). There are plugins for the transpiler in all of the popular JS task systems such as grunt, gulp, and broccoli.

We were thrilled to incorporate the plugin into our grunt setup and start writing future-proof JS. But before we could start developing the web reader, we had to make some tweaks to the [Ember resolver](#); The current version of Ember resolves dependencies by looking them up on the global namespace so we had to extend Ember.DefaultResolver to look within our module system instead. [This](#) article from Ember Zone explains the role the resolver plays and ways to extend it.

Ember CLI

At this point, instead of re-inventing the wheel, [Stefan Penner](#) from the Ember Core team suggested that we use [Ember CLI](#), a command line build system that was tailor-made for writing Ember apps in ES6 modules.

While not without some stability issues, Ember CLI accelerated our development process by providing all of the following:

- A command line utility that automates the creation of common ember tasks
- An extendable build process using Node and broccoli. We were able to easily replace the default LESS setup and instead use the [broccoli-sass](#) plugin to write CSS in SCSS
- LiveReload to monitor changes in files and automatically refresh our pages as well as our test suite
- Built in support for tests using QUnit
- Boilerplate code generation for models, routes, controllers, and more
- Build outputs that are production-ready with features such as file minification and asset versioning

Conclusion

Upon incorporating Ember CLI into our development process and building an automated deployment step around it, we were able to significantly speed up our product development cycle. Writing modularized JavaScript and using the entire ecosystem of tools Ember CLI provides has also helped us keep our code manageable even as we continue to scale, enhance and refactor the Oyster web reader.

Finally, our approach has been validated by [Ember 2.0](#), the forthcoming version of Ember, which is slated to use ES6 modules and Ember CLI as the default standard for creating Ember apps.

Our ultimate goal is to help connect readers to book they'll love. Bringing half a million books to your browser is just one way we're pushing the boundaries of web development here at Oyster. If such possibilities excite you, email nitin@oysterbooks.com to see how you can help us get there together.