



We may be on the cusp of a new revolution in Web development.

BY TAYLOR SAVAGE

Componentizing the Web

THERE IS NO task in software engineering today quite as herculean as Web development.

A typical specification for a Web application might read: The app must work across a wide variety of browsers. It must run animations at 60fps. It must be immediately responsive to touch. It must conform to

a specific set of design principles and specs. It must work on just about every screen size imaginable, from TVs and 30-inch monitors to mobile phones and watch faces. It must be well engineered and maintainable in the long term.

With the latest Web technologies, this list grows: the Web application must work offline. It must be able to send push notifications. It must sync in the background.

Of course, not all new Web projects must meet this complete set of requirements—some may be more full-fledged single-page apps, others more publishing- or e-commerce-focused—but this diversity of applications makes the Web developer's job even more difficult.

The real challenge arises when the expectations of the Web application meet the realities of the Web platform. The raw materials that are available to build Web applications have not remotely kept up. The building blocks of HTML—such as “<div>,” “<p>,” “<h1>,” and “”—are great for doc-

ument markup, but they are not sufficient to build today's complex, app-like interfaces and websites.

As a result, the Web development community has evolved a vast array of frameworks to tackle the problem of building sane interfaces out of the basic elements the platform provides. The scope and diversity of available Web frameworks is immense—a thriving ecosystem that only a platform as powerful and flexible as the Web could support. As of this writing, more than 60 frameworks are listed on TodoMVC (Figure 1),¹⁰ a showcase for framework usage. This flourishing framework ecosystem is truly amazing.

Or is it?

Though many Web frameworks are incredible feats of engineering, are extraordinarily popular, and have large ecosystems in their own right, there are a few key problems with the multi-framework model that limit Web developer productivity.

Frameworks are wide-reaching, hard dependencies. The inherently

Figure 1. Partial list of frameworks showcased in TodoMVC.

...
 agularjs
 ariatemplates
 atmajs
 backbone
 backbone_marionette
 batman
 canjs
 chaplin-brunch
 closure
 componentjs
 cujo
 derby
 dijon
 dojo
 duel
 durandal
 emberjs
 enyo_backbone
 epitome
 exoskeleton
 extjs_deftjs
 firebase-
 angular
 flight
 foam
 gwt
 jquery
 kendo
 knockback
 knockoutjs
 lavaca
 maria
 meteor
 mithril
 montage
 mozart
 ...

global nature of HTML, CSS, and JavaScript is one of many factors that have driven frameworks to be all-encompassing factories, rather than pay-as-you-go toolkits. Choosing a Web framework is typically the first major technological decision in a new Web project. Because frameworks tend to be overarching, almost every line of code or item of markup must be written with the framework in mind. Angular, for example, provides complex view management and routing, dependency injection, internationalization and accessibility features, low-level animation support, and more. Meteor provides entire frontend and backend stacks, from a UI library down to database drivers. These are incredibly full-featured application platforms in and of themselves, but they lead to lock-in from day one and are extremely challenging to migrate off of. If you choose to switch frameworks, you will probably be starting from scratch.

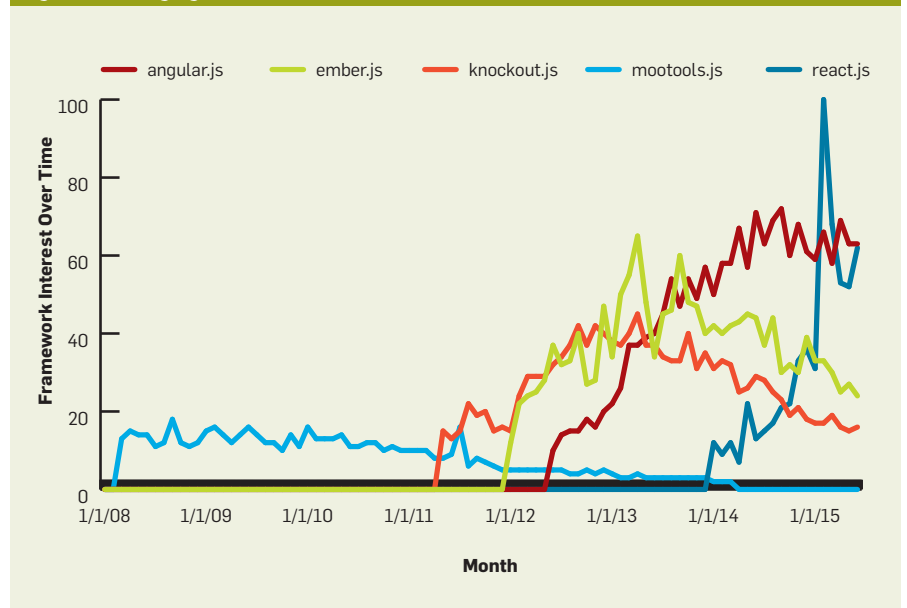
Fashionability. Web frameworks come and go like seasons. Figure 2 shows the rise and fall in popularity of five JavaScript libraries, according to Google Trends. The Web development community is constantly craving the next new thing, and rightly so—Web application requirements and device and browser capabilities are evolving so rapidly the tools must evolve rapidly to keep up. Unfortunately this means yesterday's hot

new framework might be today's old news, and the massive investment a company makes in a technology can quickly become out of date. To stay relevant in the job market, Web developers must keep up with the latest technologies, running on an ever-accelerating treadmill of Hacker News framework announcements, tutorials, and starter kits.

Lack of interoperability. A sane component model is critical to scalable interface development. Frameworks typically define their own component models for organizing and rendering interfaces. Angular has directives, and React and Ember each have their own notion of "Component." Yet a component built in one framework's model has no meaning outside that framework—you cannot use an Angular component in an Ember application, short of including multiple overlapping and redundant dependencies. Coupled with the lock-in that frameworks often require, the lack of interoperability makes writing universally reusable, encapsulated components nearly impossible on the Web.

The holy grail, the "Goldilocks" solution, would be if Web developers could pick the application architecture best suited for the problem they were trying to solve, and could reuse interface components across projects. The Web development community could have thriving ecosystems for each framework and organizational

Figure 2. Changing interest in various frameworks over time.



philosophy, as well as a shared, universal ecosystem of components that could be used in any Web application regardless of framework. Large organizations could share sets of globally maintained components that conformed to a consistent style, but that all teams could use regardless of their stacks.

This Web development utopia seems impossible to achieve, or at least technically infeasible, because it would require broad agreement among different frameworks on a consistent treatment for components. The Web, however, has precedent for these universal components—HTML elements themselves.

Consider the `<Select>` Element

The `<select>` element provides a simple dropdown menu. All frameworks understand and can leverage `<select>`—it is baked right into the platform. It works across all browsers with a generally predictable interface. It does one job, and does it well.

Moreover, `<select>` has an API surface area that makes it particularly easy to work with and valuable to use, as shown in Figure 3a–f.

It is *composable* (Figure 3a). `<select>`, composed with `<option>` for its items, generates a fully formed dropdown menu.

It is completely *declarative* (3b). A wide variety of different features can be applied using attributes in the markup.

It is *flexible* (3c). Depending on its children and attributes, it can provide different interfaces and functionality.

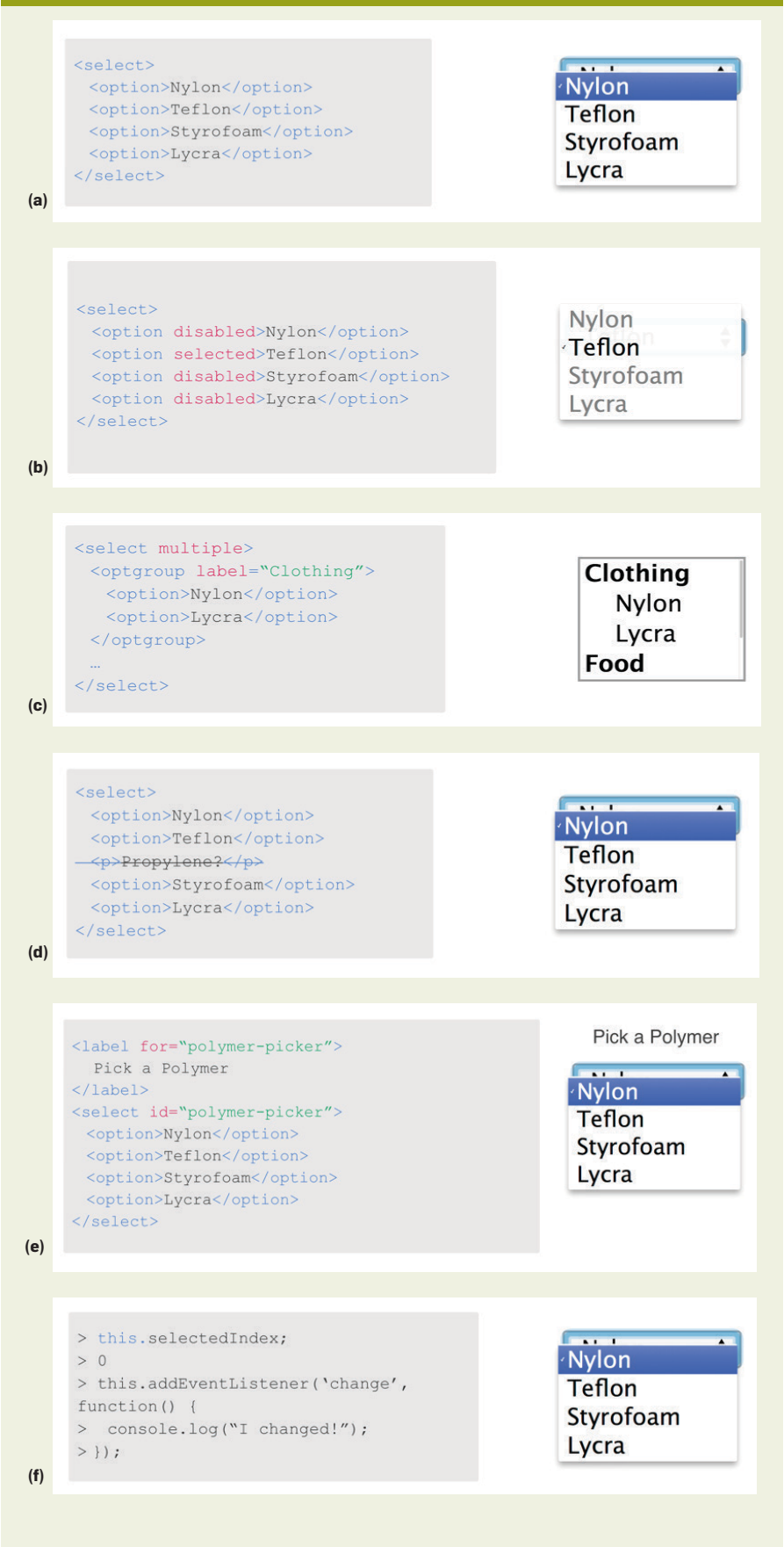
It is *forgiving* (3d). An incorrect child does not crash the application, but is simply ignored.

It is internally *accessible* (3e). Once focused, it provides all the necessary handles in order to be accessible.

Beyond what it provides declaratively, `<select>` can be *scripted* (3f). It emits events that can be listened to and acted on, and it has an imperative API that can be leveraged.

Finally, `<select>` is extremely simple to use, and can be used in just about any context. With minimal declarative markup, a developer gets this extremely powerful behavior. Regardless of the framework, this simple markup and DOM API are well understood and usable.

Figure 3. (a) A composable select element. (b) Declarative attributes. (c) Flexible interface. (d) Forgiving syntax. (e) Accessibility to styles. (f) Scriptable events.



The “<select>” element is certainly useful, but the platform only provides a limited set of such elements, and that set is woefully out of date. There are two potential ways to expand this power: increasing the number of platform-defined elements, or providing the primitives for developers to create their own elements, with all the power of native elements.

With the Extensible Web Manifesto,¹⁴ browser vendors decisively landed on the latter approach, favoring providing the primitives needed to expand Web platform features over providing higher-level abstractions directly in the platform.

What would it take to build an element like this? What sort of primitives and features of the platform might a developer need in order to build as eminently reusable an element as <select>?

A few key pieces of <select> make it useful. It has a declarative API in the form of attributes, an imperative API off its DOM node, a composition model in terms of the child elements it supports, and a standard visual interface.

Thus, to create a similar custom element, a developer would need to:

Define its API: Give the element a name, and give it imperative methods as well as declarative attributes that can be used to affect its behavior.

Define its template: Provide the element with some basic visual layout if it requires any local UI.

Encapsulate it from the document: The element’s internals should be invisible from the document. That is, adding an element to a document should not have unintended side effects.

Define its composition model: Specify what kind of children the element can accept and how it manages its children.

Manage its dependencies: A custom element should be able to use other elements in its local UI, and so should be able to specify and load the definitions of any elements it depends on.

Of course, you could build such encapsulation and template features at the framework level, and many frameworks do. But to realize the dream of broadly interoperable components, these features need to be provided at the platform level, so components built with them can be reused, just like <select>.

Enter Web Components

Web Components is the umbrella term for a handful of new W3C specs that give developers the primitives needed to build such interoperable, platform-level features. The individual specs that make up Web Components map almost directly to the specific features needed to create a truly interoperable element.

Define Its API: The Custom Elements spec² describes how an element might be given a name, define an API surface area, and respond to different events in its life cycle. The registration of a custom element boils down to a simple call as illustrated in Figure 4.

This allows the developer to specify a tag for the element and pass in a prototype for all instances of the element. At this point, all instances of “<my-element>” in the document are upgraded from HTMLUnknownElement to the prototype of the element that was passed in. Note the specific syntax may evolve slightly from the time of writing—for the latest spec, see <http://w3c.github.io/webcomponents/spec/custom/>.

Life-cycle callbacks defined in the custom elements spec give the element author more granular control over the element at particular stages of its life cycle. These callbacks include “createdCallback,” called when the element is created and has been registered; “attachedCallback,” called when the element is inserted into the document; “detachedCallback,” called when the element is removed from a document; and “attrib-

uteChangedCallback,” called when an attribute of the element is set, changed, or removed.

Between the second document.registerElement argument and the attributeChangedCallback, element authors can specify an element’s imperative and declarative APIs.

Define its template: Many elements have some included UI, such as buttons, inputs, selects, headers, and lists. To build a truly platform-level element, element authors should be able to specify the UI for their own element. Rather than define a new DSL or expose C++ hooks, Web developers should be able to use the language of the Web itself—HTML and CSS—to define an element’s template.

True templates have a few key properties. The template’s mere existence should not have side effects for the document, the template should have to be explicitly selected in order to be used and so should be encapsulated away from the main document, and it should be inert until it is actually cloned and used.

The problem of templating on the Web is one all UI frameworks face. A number of workarounds have evolved to provide this behavior, but none meet all the criteria for a true template. Some attempts at templating use a block of markup in the main document with “display: none;” to hide it until it is cloned and used, but this can have layout and performance side effects. Sticking HTML inside a <script> tag is another common approach for templating, but this can lead to security issues with “.innerHTML” and is clunky to manipulate prior to being actually initialized as DOM.

The HTML5 <template> element⁹ provides the complete set of features one would expect for a true template. It is parsed but not rendered, and inert until used, and its contents are encapsulated from the main document in the form of a document fragment. This allows element authors to use a true template to define the look and feel of a custom element, to be cloned and used each time a custom element is created and inserted into a document.

Encapsulate it from the document: Templates give element authors a

Figure 4. A simple call for a custom element spec.

```
var MyElement = document.registerElement('my-element', {
  prototype: Object.create(HTMLElement.prototype)
});
```


way to associate markup with an element, but once the element and its associated markup are inserted into a page, there should be a way to keep them isolated from the document itself. For example, an HTML5 <video> tag has a play button associated with it, but the user of the tag should not have to worry about accidentally styling or selecting the play button when using CSS or “document.querySelector” within their main document.


The Shadow DOM spec⁷ provides the mechanism for this crucial encapsulation. It introduces the notion of a “Shadow Root”—a separate, scoped tree that lives in the DOM but is protected from accidental interference by CSS selectors or DOM manipulation methods. Shadow DOM is the encapsulation primitive that lets elements be used without fear of side effects—either the element accidentally leaking style to its host document, or the host document accidentally leaking effects or style to the element.

Shadow DOM is a subtle and complex but incredibly important primitive. Fortunately, as of this writing, all major browser vendors either have shipped or are working on implementing Shadow DOM.

Define its composition model: To work like a native HTML element, a custom element must be able to accept and manipulate children. The Shadow DOM specification introduces the concept of “Distribution”⁸—the ability to specify insertion points within a shadow root where specific children can be “distributed” into the main document.

This allows a custom element author to define what kinds of children an element accepts and how it interacts with them. Authors can use this to specify how a “select” element might look for and project its “option” children. Distribution essentially provides another API surface area for an element, in the form of children it accepts.

Manage its dependencies: With the ability to define its own internal markup via a template and shadow root, one could imagine a custom element relying on other custom elements for its internal UI. Since multiple elements may depend on the same custom element, there



The scope and diversity of available Web frameworks is immense—a thriving ecosystem that only a platform as powerful and flexible as the Web could support.



must be some way for elements to declare their dependencies and for the browser to load and de-duplicate such shared dependencies.

The HTML Imports specification³ provides this mechanism—a way for an element author or Web developer to load HTML-based dependencies in HTML. As of this writing, the spec authors are working to reconcile this HTML loading and de-duping mechanism with the forthcoming ES6 module loading and de-duping mechanism.

With these four crucial new features—Custom Elements, Templates, Shadow DOM, and HTML Imports—Web developers finally have the platform-level primitives needed to create truly reusable custom elements, with all the power of native HTML elements.

How to Leverage Web Components

The question remains: How can individuals and organizations benefit from the capabilities provided by Web Components?

The most immediate use case for Web Components is in building user interfaces. General best practice in software engineering dictates that systems be isolated and componentized. This guideline can now be directly applied to building Web UI at the platform level, with custom elements as the components.

The first step in building a new Web-based product that uses Web components as interface elements would naturally be to build the set of custom elements the whole product would share. Because custom elements can encapsulate their own look and feel, this element-creation step might include building a visually consistent set of buttons, data tables, menus, layout templates, and other UI components as elements to be used across the application or suite of applications.

Considering the unit of work for a frontend engineering team to be a component, rather than defining the unit of work as a screen or flow, starts to unlock the organizational power of Web Components. By freeing up each individual to focus on a single custom element at a time, teams can minimize both visual inconsistency and duplicate work. All appearances of an

element within an interface—all the buttons used in an application, for example—are reuses of a single custom element, and are consistent in terms of look and feel.

Individual engineers also may benefit from minimizing the cost of switching context between building application logic and UI. By spending the time up front to ensure perfect element design independent of how the application works, pixel-perfection goes from being last-minute polish to being a necessary stage in application development. The app creation stage that follows this component creation stage is also much more streamlined. From the first line of app logic code written, the application looks and feels complete. It becomes much easier to get a sense of the final application throughout the development process, helping catch user experience flaws and begin useful QA processes early in the development cycle.

The efficacy of the custom-element-based interface model is felt even more strongly on a team's second project. Since they have already spent the time to construct a pixel-perfect set of interface elements, the costs of building the second interface are dramatically lowered. This frees the team to focus on new features and overall performance, rather than reinventing UI. Any improvements to the elements that come out of the second project can be seamlessly incorporated into the first project. Each product becomes a capital investment that pays dividends over the entire lifetime of the custom element set.

The engineering practice of reusing elements is also innately different than copying and pasting front-end layout from an old project into a new one. Custom elements, designed in isolation from the beginning, are specifically meant to be flexibly reused. Raw layout code or markup in old projects was rarely designed to be used in a different context, and it can be full of bugs and quirks. Overarching stylesheets lack encapsulation, and can quickly build up cruft and become difficult to maintain. If an element needs to be updated to fit a new use case that becomes a capital



A single consistent set of UI elements helps ensure synergy and efficiency between design and engineering teams. Reused elements ensure brand consistency, guaranteeing the exact same look and feel everywhere.



investment in the custom element that benefits all future users. If raw layout code needs to be updated, it often becomes one additional hack on top of a patchwork of non-systematic edits that rapidly decays into spaghetti code.

A single consistent set of UI elements also helps ensure synergy and efficiency between design and engineering teams. Reused elements ensure brand consistency, guaranteeing the exact same look and feel everywhere. A visual brand will thus be enforced not only at the design level but also at the implementation level. A set of elements provides a living style guide, and makes it much easier to align engineering with design: design no longer occurs in a vacuum, as each visual piece and tweak can be quickly incorporated into a component and tested in situ. Visual design overhauls are also much easier to achieve. Custom elements only have to be redesigned once and the new style can be quickly implemented everywhere, often with a straightforward upgrade to the element definition.

Organizations also benefit from separating custom element creation from element usage. Some engineers will simply be better suited to achieving pixel-perfect designs for elements, because they have an eye for animation, knowledge of the platform quirks, and a passion for visual detail. By having these engineers focus on creating custom elements to be used in many applications, their skill can be leveraged companywide.

Good UI design and performance is often as much art as science, and can take an artist's touch to get just right. Custom UI elements allow this artistic achievement to be broadly shared and utilized. Custom elements help make easy things easier, and difficult things easily repeatable. The broad and long-term advantage to having a single aligned set of UI elements makes it an obvious early investment for an engineering organization.

Perhaps most important, because custom elements based on the Web Component specs are built with the platform rather than with a specific framework, they can be reused regard-

less of what structural framework the next project is built with. The capital investment in a set of visually consistent elements persists far longer than a set of components built for one specific framework technology.

Of course, with great power comes great responsibility. Native HTML elements, built by browser vendors, have accessibility features baked in. The responsibility to make a custom element accessible falls on the element author. Just as a native HTML element would be incomplete if it was not naturally accessible, a custom element must bake in accessibility features as much as possible. To a certain extent, ecosystem dynamics should reward elements that are naturally accessible, but the element author community has a responsibility to explicitly prioritize accessibility from day one. Custom elements are not a magic accessibility wand—high-quality elements will be internally accessible, but application authors must also get accessibility right at the application level.

The Web Component Ecosystem

Beyond the benefits for a single team or organization, one can imagine the network effects that might be possible with an ecosystem of fully interoperable Web components.

Suites of custom elements could be created to make building full-fledged applications on the Web easier: different “UIKits” for different types of Web applications. Custom elements could be built for specific use cases, such as elements that make blogs easier to create, or e-commerce sites more effective and easier to use, or data visualization easier to achieve. Custom elements could breathe new life into the movement for a truly semantic Web, by making it possible to intertwine form and function into a single element. Catalogs could help organize a burgeoning ecosystem of elements, crowdsourcing element ratings, and reviews.

Such an ecosystem, predicated on platform-level interoperability, would require broad adoption of platform-level APIs that give Web developers the ability to create custom elements. The Web Components specs are a substantial undertaking. Since

their introduction in 2011 they have generated lively discussion and have evolved based on feedback. Though there is general agreement on the value of Web components, there are two important contentious pieces of the specs being worked out by implementers—reconciling HTML Imports with the forthcoming ES6 module system, and ironing out specifics of Shadow DOM behavior. You can follow and join the conversation on the public-webapps mailing list.⁶

Currently, the Template element is part of the living HTML spec⁹ and is broadly supported by modern browsers. HTML Imports, Shadow DOM, and Custom Elements have been seeing growing cross-browser enthusiasm, especially after recent meetings to address the more contentious pieces. They have been shipped in their entirety beginning with Chrome 36. Microsoft Edge recently announced¹² it is starting development on the HTML Template element, and stated positive views on the latest evolutions of the remaining specs. Firefox is shipping implementations under a flag, and recently published an in-depth article¹³ on the history of Web components, with the hopeful conclusion we are nearing broad cross-browser support.

Fortunately, developers can start building with the complete Web Components APIs today, using the comprehensive set of Web components polyfills.⁴ These JavaScript implementations support Custom Elements, HTML Imports, and Shadow DOM across the last two versions of major browsers starting with IE10, Safari 7, and the evergreen browsers Chrome and Firefox. Web Components-based libraries such as X-Tag,¹¹ Polymer,⁵ and Bosonic¹ rely on some of the polyfills for broad browser support, and include optimizations around the heavier parts of the polyfills to achieve production-ready performance.

Web developers today have it tough. But with the consistency of a sane, platform-level component model coupled with the wild, expansive power of the Web ecosystem, we might be on the cusp of a revolution in Web development. Happy componentizing!

Related articles on queue.acm.org

Web Services: Promises and Compromises

Ali Arsanjani, Brent Hailpern, Joanne Martin, and Peri Tarr

<http://queue.acm.org/detail.cfm?id=639315>

From COM to Common

Greg Olsen

<http://queue.acm.org/detail.cfm?id=1142043>

A Conversation with Roger Sessions and Terry Coatta

<http://queue.acm.org/detail.cfm?id=1095416>

References

1. <http://bosonic.github.io/>.
2. <http://w3c.github.io/webcomponents/spec/custom/>.
3. <http://w3c.github.io/webcomponents/spec/imports/>.
4. <https://github.com/webcomponents/webcomponentsjs>.
5. <https://www.polymer-project.org/1.0/>.
6. <https://lists.w3.org/Archives/Public/public-webapps/>.
7. <https://w3c.github.io/webcomponents/spec/shadow/>.
8. <https://w3c.github.io/webcomponents/spec/shadow/#distributions>.
9. <http://www.w3.org/TR/html5/scripting-1.html#the-template-element>.
10. <https://github.com/tastejs/todomvc>.
11. <http://x-tags.org/>.
12. Leithead, T. and Eicholz, A. Microsoft Edge and Web Components; <https://blogs.windows.com/msedgedev/2015/07/15/microsoft-edge-and-web-components/>.
13. Page, W. The state of Web Components; <https://hacks.mozilla.org/2015/06/the-state-of-web-components/>.
14. W3C Extensible Web Community Group. Extensible Web Manifesto, 2013; <https://extensiblewebmanifesto.org/>.

Taylor Savage is a product manager on the open Web platform team and lead PM on the Polymer project at Google. Prior to Polymer, he worked as project manager on new features for Google search.