ionic Whitepaper

Guide to Building Design Systems With Web Components

Design systems have been a hot topic in the design world for a few years now, but many in the development space are just getting acquainted with them.

In fact, when we first set out to build Stencil—an open source Web Component compiler used to build custom component libraries—to support Ionic Framework (our other open source project), design systems were pretty far from our minds.

Since then, we've witnessed a large and growing community of developers using Stencil to build custom component libraries to support global design systems for companies like Upwork, Volkswagen, Mastercard, and Porsche. We eventually built a version of Stencil, along with additional software, services, support, and training, specifically designed to address the needs of design systems.

Through the many large-scale design projects our team has worked on, we've started to recognize the ways that developers can best support their team's design systems goals, and identify where and how things can go wrong along the way.

We've also received lots of great questions about how Web Components can help teams build design systems, and why you would use a component compiler like Stencil instead of relying on a pre-existing component library like Bootstrap or Ionic.

The goal of this guide is to share our experiences and best practices with developers who are embarking on a design systems journey and have similar questions.

We'll start with some of the basics of design systems for anyone who's new to the topic, and then we'll look at why they're valuable, why design systems projects fail, and the benefits of Web Components and Web Component compilers, before investigating Stencil more closely.

We hope you find this guide valuable. Get in touch if you have questions, feedback, or just want to chat more about your design systems goals and challenges.

Yours,

The Stencil Team

CONTENTS

If you're building a handful of apps in a startup or small business, delivering a consistent user experience across teams and projects is relatively easy. But enforcing a consistent set of design standards is much more challenging if you're part of a larger organization—especially one with many distributed teams and concurrent projects, or hundreds of developers and designers. The problem is amplified when you consider the diversity of technologies and frameworks in use in most enterprises today, which makes it challenging to find a single approach that works for everyone on the team.

This is a problem that we hear about frequently. One way to solve it is by implementing a design system using custom, framework-agnostic Web Components.

In this whitepaper, we'll take a detailed look at design systems implementation with Web Components, and step through:

1. What design systems are
2. Why design system projects fail
3. Ensuring design system success using Web Components (WCs)
4. An introduction to Stencil, our Web Component compiler
5. A look at the Web Component ecosystem and how they pair with JS frameworks like Angular, React, and Vue
6. Examples of companies using Web Components today

By the end of this paper, you will have a solid idea of what you need to get started with Web Components and design systems, and some helpful tactics to ensure your project's success.

Introduction to design systems

A design system addresses how all of the properties around your apps and websites look, feel, sound, and act. Implementing them involves developing a centralized library of UI components that can be shared across teams and projects to simplify design and development and avoid duplication of effort, while ensuring consistent brand experiences, performance, and accessibility, all at scale.

When done right, implementing a design system means that developers no longer need to spend time thinking about how to build components or how to tackle repeat problems that are common to every app, such as application search, calendar widgets, or data table grids. Instead, a rich library of custom UI components shared across all projects allows developers to focus on what matters most: solving customer problems and delivering value.

"That's the beauty of building a design system. By deciding on a detail once, you free up your entire product development team to focus on solving actual customer problems."

- The Hubspot team on why design systems are valuable

However, many design systems fail to realize their full potential. Too many projects begin with great passion and enthusiasm, yet fizzle out due to lack of enterprise-wide adoption — often after tremendous investments of time and money have already been sunk into the endeavor.

Why design system projects fail

The complete list of causes is unique to every business, but a few common practices often contribute to most failed design system projects.

1. Stopping at design

The first bad practice is stopping at the design stage. Without a working component library available to development teams, design consistency is never fully realized because development teams either aren't aware of the design requirements, aren't good at implementing design standards, or simply ignore them.

Stopping at the design stage also fails to address the deep problems around developer re-work and parallel development efforts across projects—where one team is building a component that 20 other teams have already built.

2. Building a HTML and CSS library

The second bad practice is building a design system using pure HTML and CSS components that developers then copy and paste into their respective projects. There are a few challenges with this approach.

For one thing, pure HTML and CSS components lack any dynamic functionality. You can create and share a date-picker, for example, but each team that uses it will still need to program the logic required to make it function.

On top of that, it is nearly impossible to track and maintain proper version control of copy & paste code. And if you want to use your components with any JS frameworks like Angular, React, or Vue, you'll need to add—and maintain—separate libraries and wrappers for each framework that you want to use them in.

Thus, this approach often results in a brittle implementation that negates a lot of the promised benefits around avoiding rework while increasing time and cost savings.

3. Limited component-building expertise

For designers and developers that go beyond HTML and CSS to build real-code components from scratch, they often come up against what UI library makers have known for years: this stuff is *hard*. A single feature might take months to perfect into something that looks and feels great on any platform or device, and performs like your users expect under a wide variety of circumstances.

Lacking in-house component building expertise can slow down design systems development and limit adoption if the custom UI components don't deliver on the required functionality, accessibility, platform compatibility, and performance.

3. Betting on a single technology

The final misstep is building a design system on top of a single technology. A common scenario is when one team builds a component library based on their framework of choice, and then meets friction when they attempt to roll it out company-wide. While the original team might have built their library in React, a third of the organization is using Angular, and another third is building with Vue.

This problem is particularly acute in large, globally distributed organizations, where it's nearly impossible to get all teams to standardize on a single framework—never mind a specific version of a single framework—or development technology.

"[N]ascent design systems teams fall into a trap of coupling a UI with a specific tech stack. Creating a technology dependency in order to achieve a specific UI style inherently limits where that UI can go. That might not be an issue if your design system only serves one or two applications that share the same technology stack, but this becomes a big issue for organizations that manage tons of applications built on a smorgasbord of technology. "

- Brad Frost, Managing Technology-Agnostic Design Systems

While these scenarios address some of the major reasons design systems projects fail, there are still many more that you may encounter along the way. From creating a library but failing to get other teams in the organization to adopt it, to lack of executive sponsorship or issues related to company culture, there are almost as many reasons for projects to fail as there are projects.

The question then becomes, how do you ensure that: 1) your design systems project succeeds, and 2) you gain a full understanding of all the benefits they have to offer?

Formula for success

To ensure successful implementation and adoption of your design system, we believe that the right solution must include:

1. **Real-code components.** Dynamic, working components that can be shared throughout an organization with proper tracking and version control.

2. **Access to component-building expertise.** Access to expertise that can help you build completely custom components from scratch, or an existing component library that you can borrow from and customize to meet your unique design specifications and brand standards.

3. **Technology-agnostic.** Your component library will work with any framework or technology, and can be deployed anywhere.

Why Web Components?

In our experience, the best way to ensure success when implementing your design system is to build a library of custom Web Components—using a Web Component compiler like Stencil—that can be shared and consumed across your organization, in any project, in any tech stack.

Web Components use a set of standardized APIs that are natively supported in all modern browsers, and supported in older browsers using polyfills. WCs are uniquely suited to the needs of a universal design system for a number of reasons.

1. They're framework agnostic

One of the most appealing benefits of Web Components is the fact that they give your development teams the flexibility to choose the underlying tools and frameworks—and versions of those frameworks—that they prefer. As we pointed out earlier, one of the great challenges of implementing a universal design system is getting all of your development teams to standardize on just one set of technologies. Using Web Components, each team can use what works best for them, giving them complete freedom to use the tools they love—today and tomorrow.

This liberates teams from the highly volatile landscape of JavaScript frameworks and tooling. By using a consistent set of web standards, Web Components don't depend on a specific framework like Angular, React, or Vue. You can use Web Components with any of these frameworks—and we encourage you to do so to take advantage of the many benefits they provide—but the great thing is that you won't depend on that framework for your components to work.

After all, as much as we love the hot frameworks of today, who knows what tomorrow will bring? By choosing Web Components, you insulate yourself from the threat of tech churn, so you'll no longer have to worry whether you're picking the right horse.

2. They're highly customizable

By definition, a design system implies a customized collection of UI components that match your specific brand standards and style guidelines. While there are no limits to what your designers can conceptualize, implementing those customizations in some development environments and frameworks can be tricky.

Web Components can be styled and customized to match any design pattern you want to achieve. With the simple use of HTML, JavaScript, and CSS, you can build a library of UI components that match whatever your designers dream up.

3. Deploy them across mobile, desktop, web

Another great advantage is that your component library will work across all projects, not just desktop web apps.

For example, using a hybrid mobile framework like Ionic, you can deploy Web Components across just about any platform or device, from native iOS and Android apps, to Electron and desktop web apps, and even Progressive Web Apps.

You can also open up the design of your WCs to change based on the platform that the component is running on. If you'd like a more native style when running on Android or iOS, you can slightly change the design (or structure) your component

Ionic Framework is a UI component library used by over 5 million developers worldwide. Throughout the past 6+ years, we've spent a lot of time perfecting our components so they're extremely performant, accessible, and customizable.

However, Ionic's earlier versions were built on top of Angular. With the evolution of JavaScript frameworks, and the rise of React and Vue, we realized that continuing to build on top of Angular severely limited the amount of developers we could support.

On top of that, Ionic needs to ensure that we can deliver updates to components without breaking apps, and that our components feel like they're completely native to the JS framework that each developer is using.

Web Components began to help us solve the JS and maintenance problems, and we built Stencil to take the developer experience across the finish line.

Even though we built Stencil to solve our own problems building a component library used at scale, we decided to make it available to everyone by making it a free, MIT Open Source project. Stencil gained a large amount of traction in the Design Systems community at large enterprises (who face the exact same problems we do), and the rest is history!

- Adam Bradley, Director of Open Source Engineering at Ionic

Limitations of Web Components

Now that we know the value that Web Components have to offer, we should also note that there is a catch: Web Components, on their own, are not enough. They currently run on a fairly primitive set of standards, so you may need a little extra help to get them to meet your objectives. Some of the limitations include:

→ When you try to use pure vanilla Web Components in an application, functionality like server-side rendering and progressive enhancement is not supported by default

→ Some out-of-date clients don't support the Web Components standard

→ WCs technically work with any framework, but there are some limitations like lack of type support and input bindings, and challenges passing properties to components

The good news is that, with help from open source tools like Stencil, you can overcome all of these challenges.

Let's dive deeper into how it works.

Introduction to Stencil

Stencil is a compiler that generates Web Components (more specifically, Custom Elements), and combines the best concepts of the most popular frameworks into a simple build-time tool.

Since Stencil generates standards-compliant Web Components, they can work with many popular frameworks right out of the box, and can be used without a framework. Stencil also enables key capabilities on top of Web Components, particularly pre-rendering, and objects-as-properties (instead of just strings).

Compared to using Custom Elements directly, Stencil provides extra APIs that make it simpler to write fast components. APIs like Virtual DOM, JSX, data-binding, and async rendering (inspired by React Fiber) make fast, powerful components easy to create, while still maintaining 100% compatibility with Web Components.

Stencil was created by the Ionic Framework team to help build faster, more capable components that worked across all major frameworks. Web Components offered a solution to both problems, pushing more work to the browser for better performance while targeting a standards-based component model that all frameworks could use.

However, Web Components by themselves weren't enough. Building fast web apps required innovations that were previously locked up inside of traditional web frameworks. Stencil was built to pull these features out of traditional frameworks and bring them to the fast emerging Web Component standard.

Web Components vs. Frameworks

The Web Component ecosystem has a diverse set of players, each with a different long-term vision for what Web Components can and should do.

Some think Web Components should replace third-party app frameworks, while others think that Web Components are more suited for leaf/style/design nodes and shouldn't get in the business of your app's component system. There are also many app framework developers that either don't see the point of Web Components or consider them to be an affront to frontend innovation.

At Ionic, our vision is somewhere in the middle. In the long term, we see app development teams continuing to use their framework of choice. We envision these frameworks continuing to get better, smaller, and more efficient, with increasingly good support for targeting and consuming Web Components—and as companies continue to embrace them for shared design systems, big teams will be consuming an increasing amount of Web Components.

At the same time, we believe that an indispensible feature for Web Components is solving those component distribution and design system problems. We also believe, however, that 90% of the market doesn't have those problems to begin with, calling into question the productivity of the current debate about the merits of Web Components.

Pairing WCs with JS frameworks

Stencil is a compiler that generates Web Components (more specifically, Custom Elements), and combines the best concepts of the most popular frameworks into a simple build-time tool.

Angular

At a glance: Angular provides decent out-of-the-box support for Web Components, but there are some notable limitations that you'll run into if you're trying to build dynamic applications with Web Components and Angular. That includes a lack of two-way data binding, limited type support, and an inability to access Angular-specific properties in your components.

Here's an example of a slider Web Component in Angular without bindings:

```
<range-slider
  min=”0”
  max=”1000”
  [value]=”rangeValue”
  (change)=”selectValue = $event.details.value”>
</range-slider>
```

The good news is that, with Stencil bindings, you can use Web Components in your Angular project as if they were actual Angular components, complete with two-way data binding, type support, and access to Angular properties.

Here's the full list of what you get with Stencil bindings:

- → Web Components become available as Angular components
- → Get types for your components
- → Can use Angular-specific properties on components
- → Two-way data binding with ngModel
- → Developers import actual Angular Library
- → Feels like interacting with Angular components

In the example below, we're using the same slider Web Component in Angular, but this time with the Stencil bindings:

```
<range-slider
  min="0"
  max="1000"
  [(ngModel)]="selectValue">
</range-slider>
```

You can see that it includes two-way data binding with ngModel.

React

At a glance: React has very limited out-of-the-box support for Web Components. While their documentation shows a few very simple examples of working with WCs in React, you will quickly grow past those basic use cases if you're building any sort of dynamic application.

For example, React can only pass strings and numbers to Web Components and it cannot listen to custom events.

Here's a code sample of a Web Component in React without bindings:

```
const userDetailRef = useRef(null);

React.useLayoutEffect(() => {
  const { current } = userDetailRef;
  current.user = {
    userId: 1234,
    userName: ‘jthoms1’,
    age: 38
  };
  current.addEventListener(‘onUserSelect’, (customEvent) =>
    selectUser(customEvent);
  );
}, [userDetailRef]);

return (
  <user-details ref={userDetail}></user-details>
);
```

The Stencil bindings allow you to use Web Components in your React project with the full range of functionality that you would expect. That includes:

- → Web Components become available as React components
- → Get types for your components
- → Developers import actual React Library
- → Feels like interacting with React components
- → Support for complex property types
- → Support for events

Here's the sample Web Component in React, this time with Stencil bindings:

```
return (
  <UserDetails
    user={{ userId: 1234, userName: 'jthoms1', age: 38 }}
    onUserSelect={selectUser}
  ></UserDetails>
);
```

As you can see, the components appear as though they are React components and all properties get passed correctly including functions, objects, and arrays. The bindings also account for custom events by creating a prop called "on<EventName>". These allow React developers to interact with the Web Components as though they are React components.

Vue

At a glance: Vue has done a nice job of integrating with Web Components out-of-the-box, but there are a few issues with the developer experience.

For example, you will not be able to use v-model on inputs, as shown in the code sample below.

```
<range-slider
  min=”0”
  max=”1000”
  v-bind:value=”rangeValue”
  v-on:onChange=”selectValue = $event.details.value”>
</range-slider>
```

Adding Stencil bindings will enable you to use Web Components in your project as though they were actual Vue components.

- → Web Components become available as Vue components
- → Get types for your components
- → Can use v-model on inputs
- → Developers import actual Vue Library
- → Feels like interacting with Vue components

Using a Web Component with Stencil bindings in Vue:

```
<range-slider
  min=”0”
  max=”1000”
  v-model=”selectValue”>
</range-slider>
```

Who’s using WCs today

If you’re considering using Web Components, one thing you want to see is production examples. Here are a few that demonstrate the value that Web Components bring to the table.

Ionic Framework (versions 4 and up) has been a very successful Web Component-based design system/UI framework. Web Components are now in thousands of app store apps, and nearly 4 million new Ionic Framework projects are being created every year. And one great thing about it is that many Ionic Framework developers probably don’t even know they are using Web Components—they integrate seamlessly in Angular as well as our upcoming React and Vue support.

There are a mix of startups and large enterprises building with Web Components today.

On the startup side, popular workout app Sworkit recently rolled out their new PWA and Native app using Ionic Framework 4, using Web Components both on the web and in the app stores.

On the enterprise side, Salesforce recently moved their Lightning components to Web Components and has been a pioneer in Web Component-based design systems.

Upwork, a publicly held company that helps match freelancers to jobs, recently shipped Web Components on their homepage that were built with Stencil. There are many other enterprise examples, from Amazon and Adidas to Panera, Porsche, and more.

"At Porsche, we have a heterogenous ecosystem of products built with Angular, React or without any framework. As a design system team with a small number of developers, to give us the flexibility we needed and keep pace with our development roadmap, we wanted to standardize on one set of UI components that would work across any product. Building a custom design system based on Web Components has enabled us to do that."

- Marcel Bertram, Design Systems Lead at Porsche

Public sector organizations are also getting on board. The State of Michigan recently rolled out a design system based on Web Components and Stencil.

Stencil is driving an increasingly large portion of our business at Ionic. Enterprise customers continue to get on board, and the major driver is that Stencil and Web Components help solve their component distribution and creation problems, across their many properties and teams, which often don't have a prescribed set of frontend technologies in use.

Conclusion

In this guide, we set out to show how design systems help companies deliver a consistent user experience by enforcing design standards across their apps. By implementing design systems to solve these problems, teams save time, save money, and can focus on what matters most: solving customer problems.

But too many design system projects fail at the outset or never realize their full potential.

There are a few common reasons for that:

1. They leave implementation to each individual product team
2. They build HTML & CSS components without JS functionality
3. They lack the in-house expertise to build fully functional components
4. They bet on a single technology like Vue, React, or Angular

The goal of design systems is to have one set of actual code-based components that work everywhere, across all technologies. Web Components are the way to achieve this.

1. They work with any framework
2. They work on mobile, desktop, and web (PWA)
3. They are super fast and light
4. They are open and standards-based

Unfortunately, Web Components can't get us all the way to our goals on their own. Sometimes Web Components need a little help to realize their full potential.

So what's the answer? How do we complete successful design systems projects using Web Components for every situation? Tools like Stencil help complete the last mile that you need to make Web Components perfect for design systems.

Now that we have the high-level details down, you can ensure your design system project is a success by:

1. Building your design system on Web Components
2. Using a Web Component compiler like Stencil
3. Generating framework bindings to work with your JS framework(s) of choice

How to get started

Stencil's out-the-box features will help you build your own library of universal UI components that will work across platforms, devices, and front-end frameworks. Check out the documentation to learn more.

Additionally, the Ionic team (makers of Stencil) offers Stencil Enterprise, an enterprise-ready version of Stencil that includes comprehensive assistance and tooling for enterprises embarking on their design systems journey. If this is your first time building a design system, or if you're new to Stencil, get in touch with one of our Solutions Engineers for a free consultation on how to meet your goals and get the most out of the platform.

Learn more at https://stenciljs.com/design-systems