

WEB COMPONENTS

Data Sharing & Framework Integration: Perks & Flaws Series - Part 4

Manuel Rauber

Jun 17, 2020

[HOME](#) > [ARTICLES](#) > [WEB COMPONENTS](#)

In this four part **article series**, we are exploring the **perks**, **flaws**, and current standards of forming Web Components. This last article will teach you how Web Components can share data and

ABOUT THE AUTHOR




Manuel Rauber

Manuel Rauber is Consultant at Thinktecture and focuses on Web Components, Angular, .NET Core & Node.js.

COMPONENTS & MORE

Article Series

1. The
Motivation
for using
Web
Components,
an
Introduction
 2. Perks of Web
Components
 3. Flaws of
Web
Components
 4. Data Sharing
and
Frameworks
- 

Subscribe to our free monthly newsletter for our experts' latest technical articles about Angular, .NET, Blazor, Azure, and Kubernetes.

Sharing Data and Services

If you are used to frameworks like Angular, React, or Vue, you know how to share data and services between your components, right? In Angular, you can utilize its built-in dependency injection, in React, you can use Redux and so on. You

put it into DI, inject a configuration object with the necessary URLs and you are good to go to use that class in every component. Those approaches are perfectly fine because your whole application is in the hands of the framework.

If you develop *enterprise-like* applications, you have a lot of data to share with your components. For example, API URLs, translations, number formats, date formats, color themes or global settings. Maybe there is also the possibility for changing some of these at run time.

When it comes to Web Components, you usually live in the world of your component. Is that really true? We may need to divide Web Components into two categories.

Self-Contained Web Components

For me, self-contained Web Components can work standalone. They do not have a dependency on any other Web

either via HTML attributes or JavaScript properties. By that, almost everything in your Web Components needs to be customizable, for instance, translatable texts, data formats, API URLs and more. It may have events if something in your component has happened, so the parent component can react to it. If you need services or depend on external data, you have to implement it right into your Web Component. Depending on your use case, it is also possible to raise an event for requesting data. For example, if you build a Web Component for virtual scrolling in lists, you can raise an event if you run out of data for your virtual elements. The parent then has to satisfy your request for more data.

Component-Library Web Components

Another approach comes to my mind if you are building a Web Component library. Within the library, I would expect

every Web Component expects to exist. This approach can also be application-wide, so it may feel like one of the current framework's components. It simplifies handling common and shared services and data tremendously. For example, if the user selects another language, and you have to re-translate your components, you can raise an event for that. Every component will follow that event and get the new translations from a service. With that approach, you can easily build base classes to handle such common features.

Conclusion

When developing Web Components, you need to start thinking much more about your data flows. Using SPA frameworks, they already have a ready-to-use solution for you. In the end, it may be a mix of both Web Component categories in each application, especially if you use third-party Web Components.

Creating Web Components with Frameworks

Working with Web Components without any framework, sometimes feels like going back (to the future) in time. Why? Currently, we are used to have frameworks like Angular, React, or Vue that provide certain features like a template engine, dependency injection, or a virtual DOM. For native Web Components, we do not have that, so we will find ourselves adding and removing event handlers and updating our DOM manually. Like we did many years ago!

However, using a framework often comes at the cost of file size. It needs more time to download, to parse and to execute, before the application can be presented to the user. To

load module or create a critical CSS path.

Now imagine a world, in which Web Components are built with big frameworks, like Angular or React. Speaking of Angular, even in its current form (with Ivy), a small Web Component will be around 1 MB big. Yes, there are tweets that your application is only 6 kb in size with Ivy, which is a "Hello World" application. If you include a form, your file size will increase significantly. As you may know, your application will not be built by a single component, but maybe 10, 30 or even 50. If each components size is 1 MB, you would have to download 50 MB, which is way too much for an application.

A Solution

A possible solution is not to use a framework at all. That may work for smaller Web Components or smaller applications. However, for bigger applications, that is simply not suitable.

relying on others. If you plan to create a component library for your own applications, that may be a valid option. For sure, it comes at costs of creating the core foundation of the framework, creating your own build pipeline, and so on. You have to consider many things from the [third article about the flaws of Web Components](#).

Another way is changing how you bundle a Web Component. Either you stuff the whole (tree-shaked) framework into each Web Component, or only the host application is responsible for loading the framework. Each Web Component just assumes that the framework has been loaded correctly and uses it. That could work because JavaScript is still globally scoped. But, have in mind that you can only load the whole framework this way. Tree-shaking is hardly possible because you do not know, which components may be loaded. If you do know, you can still use all techniques to get a smaller bundle size.

Components have to use the same version.

Upgrading the framework can be a long process, because you have to ensure, all Web Components are working correctly.

If you want to use one of the current frameworks, it might be helpful to have a look at the following little overview of their capabilities.

Please bear with me, that this overview contains only the frameworks, I have used for Palaver. The following section only describes the creation of Web Components with the frameworks, not the usage. Using Web Components in each framework worked like a charm, and there is already enough documentation available to get started.

Stencil.js is a framework designed for development of Web Components. It features a build pipeline, creation of documentation, data handling, and event raising with JSX, you have a comfortable way of creating templates. Also, Stencil.js helps you update your DOM, whenever the state of your component changed.

Stencil.js, for me, is a good foundation to use for your own Web Components. As their website states, it is a *toolchain for building reusable, scalable Design Systems*. It is not an entire application framework. If you need Form Validation or Dependency Injection, you have to build that yourself on top of Stencil.js.

LitElement

LitElement is from Google's Polymer project and helps to define Web Components. It is currently more syntactic sugar and solving problems within Web Components, for example attributes and properties

something similar and have to rely on existing bundlers like Webpack or ParcelJS.

Depending on your need, LitElement offers you everything to create very lightweight Web Components.

Vue.js

Vue.js has it's own wrapper [@vue/web-component-wrapper](#) which helps to wrap Vue.js components into Web Components. You have the whole feature set of Vue.js available. The wrapper also takes care of building your components and tells you how to write CSS for encapsulations. Additionally, it has some nice building flags to include or not include Vue.js in the resulting components. That helps with decreasing the bundle size of your Web Components.

Angular Elements

Angular Elements wraps your normal Angular components into Web Components. It integrates nicely with the Angular CLI, so basically, you have

size starts to decrease. If you rely on the `OnPush` change detection strategy, you can even remove `zone.js` to decrease the bundle size even more!

In my view, both Stencil.js and Angular Elements have the best toolchain and support for creating Web Components. In the end, when using Angular Elements, there is not much to care about, it works pretty well out of the box. However, as mentioned in the [third article about the flaws of Web Components](#), they lack type definitions when it comes to the usage of Web Components, even when built with Angular.

A Little Angular Note

Since I am an Angular developer, I want to make a little side note. If you want to consume Web Components, you need to add the

`CUSTOM_ELEMENTS_SCHEMA`.

With that enabled, a lot of compiler checks for unknown properties on elements are disabled. That could lead to misbehavior of your

only wraps your Web Components and use `CUSTOM_ELEMENTS_SCHEMA` in there, so the other parts of your application are not affected by it. [Like I have done with Palaver.](#)

React

React lovers, I am sorry to tell you, but React itself does not offer anything to create Web Components. There are community projects trying to solve the problem. However, if you compare them to the other frameworks mentioned, they need more time to mature. In the meantime, I would suggest taking a look at [Stencil.js](#).

Conclusion

While it is possible to create Web Components without any framework, in a real-world scenario, it is very unlikely to do so. Frameworks help to have a good foundation for your development. If you are not satisfied with one of the existing frameworks, it is very easy to build your own. With the help of bundlers like webpack or ParcelJS its a

However, if you hop on the train, you also have to address the current flaws of Web Components. Also, do not underestimate a good toolchain, which crunches your referenced assets.

For me, Web Components have one big advantage: you have to craft your component's API carefully. Not only in terms of attributes and properties but also your CSS variables, CSS Shadow Parts, content projection and events. That strengthens your API, and hopefully leads to smaller, very specific components.

And now? Don't forget to take a look at the demo application and play around with it. If you want to get notified about more articles about Web Components and other cool development stuff, don not miss to [subscribe to our monthly newsletter!](#)

Happy coding!

The Web Components (and possible solutions): Perks & Flaws Series - Part 3

MAY 13, 2020 | MANUEL RAUBER

The first article of this series introduced into the motivation for using Web Components. After looking at the perks in the second part, we are going to learn about the flaws of Web Components in this article. Please note that with the on-going development of the standards, some...

[READ ARTICLE](#)

The for Using Web Components, an Introduction: Perks & Flaws Series - Part 1

APR 17, 2020 | MANUEL RAUBER

Web Components - a term you most probably hear regularly in recent times as a web developer. With the standards of Shadow DOM, HTML Custom Elements, HTML Templates, and, in former times, HTML Imports, there finally is a native component model within the browser. In this four-part...

[READ ARTICLE](#)

The Perks of Web Components: Perks & Flaws Series - Part 2

APR 17, 2020 | MANUEL RAUBER

In the introduction article to this series, I wrote about the motivation, why to use Web Components. Now we are going to take a closer look at the perks of using them - from a technical and business point of view. If you are interested in all technical features of Web Components...

[READ ARTICLE](#)

SUBSCRIBE

Subscribe to our free monthly newsletter for our experts' latest technical articles about Angular, .NET, Blazor, Azure, and Kubernetes.

Enter email address