How to Choose the Best JavaScript Framework for Your Team:

A comparison of the top JavaScript frameworks available

Preface - About this E-book

Why JavaScript Frameworks 101?

Overall, this e-book has a singular, focused goal: to help you decide which JavaScript framework works best for you and your team by providing a technical, current, and informative summary of major JavaScript "MVC" frameworks available in 2017.

So why do you need to know about JavaScript frameworks?

1. Within the last 12 months alone, JavaScript framework usage has exploded astronomically. Using a framework when starting a new web project is the norm. From the smallest static websites to the largest stateful web apps, frameworks are utilized across the board for their unbeatable utility and software design principles. The recent explosion in popularity has diversified the features offered in the most commonly used frameworks. As such, picking the right framework for your project requires a deep knowledge of *all* available frameworks and how they compare.

2. Application development has always been a fast-moving field. The risks associated with development are low compared to the potential rewards, so developers feel freer to test new, sometimes radically different, features in the software they produce. This production speed is amplified even further in the modern web

development world, where updates are fetched simply by specifying a new CDN URL or running `npm install`.

In addition, web application development standards are shifting. While it's true that the HTML5, CSS3, and JavaScript ES2015 specifications have been standardized for some time now, the phenomenon of feature-based iteration has led to rapid change. Many analysts and tech company executives predict that web standards will adopt an official feature-based update schedule, allowing them to continue to evolve rapidly.[1] Point being: even if you're already using a JavaScript framework, *now* (and every minute after now) is a good time to re-evaluate your framework choice. For better or worse, web standards are going to continue changing quickly, and frameworks will change with them.

3. Keeping up with the evolving web landscape these days is *tough*. This e-book is the culmination of a lot of research on the current state of JavaScript frameworks. Whether you're just trying to support a variety of developers (like me!) or you're embarking on a new web project, it never hurts to have a quick, accessible guide that summarizes your options.

Significance by Association

Some of the top JavaScript frameworks discussed in this e-book are made by Google and Facebook. From Forbes to Airbnb, companies large and small use frameworks to

---

[1] *You Don't Know JS: Up & Going* by Kyle Simpson. http://shop.oreilly.com/product/0636920039303.do

revolutionize their web development workflows. We'll look more into why adoption has been so widespread later, but overall, it comes down to enabling novel user experiences and improving development workflows. The immediate takeaway is basic, but extremely powerful: if successful companies large and small are using JavaScript frameworks extensively, they must offer some benefit.

Before you grab a pot of your favorite coffee and dive in, take a look at the handy reading guide below. This e-book is intended for everyone, but depending on your experience, you may want to start at a different point.

Reading Guide

Web Development Experience	Try starting at...
Beginner	...the beginning! You should be able to learn the basics as you go, and if you're very new, read the appendix on JavaScript's history first. Don't be intimidated: this paper focuses more on software design than JavaScript syntax. (I think design can be much more fun!)
Intermediate: You know JavaScript and some front-end development.	...framework overviews. Skip all the JavaScript history and start at the point where we look at frameworks and their latest features.
Expert: You're already up-to-date on JavaScript frameworks.	...fitting a framework. This section talks about how to decide which framework is right for your projects. Even if you're already familiar with the latest and greatest in JavaScript

	frameworks, the concluding sections of the e-book may be useful in helping you start new projects utilizing them.

Introduction - A Shifting *View*

The Humble Beginnings of JavaScript

The Language Itself

If you haven't used JavaScript yet, don't fear! This is the best and most exciting time to learn this beautiful, dazzling, bewildering language. The reason this book exists is because of the booming web development field, and right now the focus of that field is JavaScript. Relatively recent developments like the release of the Node.js server platform, rich updates to browser APIs, and codification of new JavaScript language specifications have established JavaScript as a viable option for programming *entire applications*, even native ones! And, of course, all these leaps ahead have ultimately enabled the production of the frameworks we'll look at in the pages ahead.

Overall, JavaScript is somewhat of an anomaly among programming languages. Syntactically, JavaScript resembles a procedural language like C more than an object-oriented platform like Java. (Read more about the etymology in Appendix A). When you look a little deeper, you'll realize that JavaScript often doesn't act like any other programming language. From the precedence of scope to the behavior of "falsy" values, you might find yourself surprised during your first few run-ins with JavaScript.

Until very recently, many of the programming patterns used in JavaScript have been used because they've been proven to work well, rather than because of enforcement by

a standard. This has had both positive and negative impacts on the language, but **this flexibility has enabled quick adoption and unique usage of JavaScript**.

JavaScript's inherent flexibility derives from its most significant requirement: **you can run it in almost any environment.** From the beginning, developers have wrestled JavaScript into places where no programming language had gone before. At the most fundamental level, JavaScript's primary environment is a browser, and there are at least three different frequently-used browser environments to consider. Thus, JavaScript's flexibility and uniqueness come from necessity, and learning to work *with* its quirks rather than trying to work *around* them is your best course of action. At the end of the day, these "quirks" typically turn out to be very powerful features if you know how to use them to your advantage.

How to Write JavaScript

I highly recommend Kyle Simpson's "You Don't Know JS" series of books if you want to learn all the nuances of JavaScript. JavaScript is a procedural-style language with lots of quirky behaviors that developers leverage to develop powerful and elegant solutions on web, mobile and desktop. Frameworks maximize this leverage and typically work on all the platforms I just mentioned, and that's why they're so powerful. If you want to learn a bit about the history and current state of JavaScript, check out Appendix A.

What is a JavaScript Framework?

When I say "JavaScript framework,", I'm referring to a front-end "MVC" framework written in and designed to be used with JavaScript.

A warning: MVC can seem like a rather rigid term. Based on its original definition, not all JavaScript frameworks use the MVC software design pattern. Heck, maybe none of them do! MVC is more of a *philosophy* than a rigid pattern for writing code. Keep in mind that the overarching philosophy for each is to implement the principle of **separation of concerns** as much as possible. Don't get bogged down in the technical jargon; stay zen-like: *JavaScript frameworks are all about separating out your concerns, man*.

MVC and Other Software Design Patterns

Software designed with the MVC pattern is said to have three main parts: a **m**odel, a **v**iew, and a **c**ontroller. Many variations of this exact pattern exist, and frameworks utilize all kinds of different "parts." You could probably describe most JavaScript frameworks as having these parts, but rather than try to pin down exact implementations right now, just try to understand the concept.

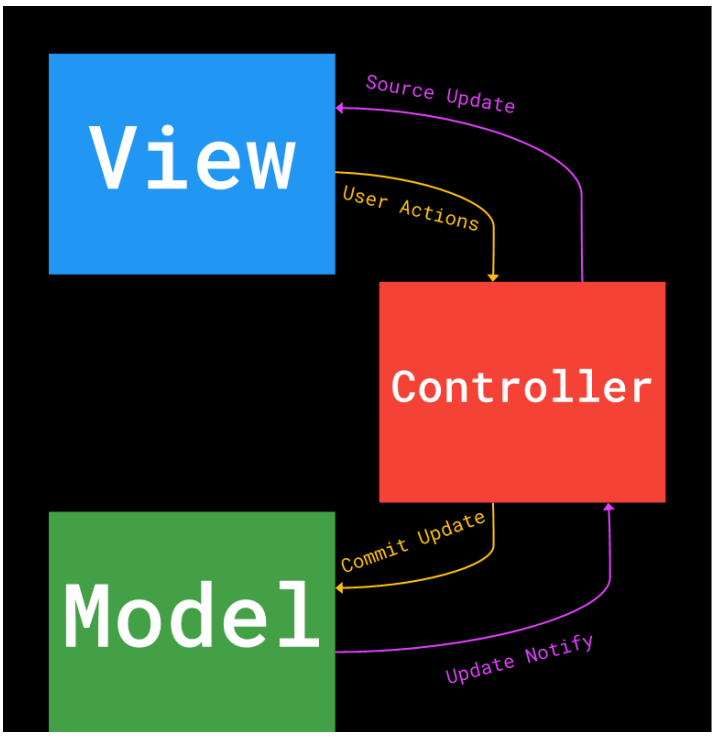


Fig 1. A simplistic depiction of the components of a typical MVC software architecture along with the actions associated with each.

The model

In MVC software, the model is a standalone unit that represents the data your application uses. If you're building an application that lets city residents register their pets online, for example, you might have a model for dogs and a model for cats:

```
Model Dog {
  string Name;
  number Age;
  string Breed;
  string Color;
  boolean PlaysFetch;
}

Model Cat {
  string Name;
  number Age;
  string Breed;
  string Color;
  boolean LikesYarn;
}
```

The model of an application is extremely important because it defines what the application will be able to do by dictating what information will be storable and retrievable. In addition, *the model is completely independent from the other parts of the application*. Other parts can interact with the model, but the model explicitly puts forth rules for such interaction. This is essential, as it allows you to abstract data management away from the user-facing parts of the app.

The view

The view of an application is the part with which users and developers are most familiar. The view is simply a compartment for holding controls and other UI elements that allow users to interact with your application. Many platforms refer to views as "forms" or "pages." Specifically, you can think of views as HTML pages on the web or XAML forms on Windows. Front-end code can be used to directly modify the view (e.g. JavaScript animations on the web), but in many cases, we should connect the view to some other component and allow it to do the heavy lifting.

This is really where the *reason* for MVC frameworks reveals itself. Understanding the **how** of frameworks requires knowledge of all their parts, but understanding **why** really only requires a brief examination of an app's view. Most developers have experienced the tendency of views to quickly become time-consuming components for a given project. When boiled down to the basics, software design reveals that the view is only needed to (1) receive input from the user and (2) show output to the user. From the design perspective, several other important factors make these user experiences enjoyable. From the development perspective, however, the view presents an interesting paradox. It may be a portal through which we can directly access the user, but how can this access be leveraged *and* allow designers and writers to work their magic freely?

Until the dawn of MVC frameworks, developers had a tough time tackling this issue. Many developers had no choice but to combine view markup with data access logic and other computations. As we all know, this leads to an unresponsive and unpleasant experience for the user. Luckily, MVC (and similar) frameworks solve this issue by abstracting away code that deals with data manipulation and other calculation. We've already seen how the model contributes to the solution by giving us another place to

worry about data, but now let's check out the controller's role in making our lives easier. (Computers do live to serve us, right? Right...?)

The controller

If we think about an MVC application as analogous to a human brain, the model would be the brain's memories, the view would be the senses and the voice (for communicating with the outside world), and the controller would be everything else. The controller of an application takes many different forms across various frameworks, but its purpose is universal: masterminding the operations of an app.

We'll talk about how controllers are implemented for each of the frameworks we cover. In general, consider a webpage with some JavaScript code loaded from a separate file. Specifically, think of an online "checkout" page for an ecommerce store. Even if we separate out the code for the model and the view into their own files, we still need additional logic to handle the checkout.

Check out this example model for a purchase made through the online store:

```
Model StorePurchase {
  string PurchaseId;
  string ProductId[];
  string CustomerId;
  number TotalCost;
  string PaymentMethod;
  boolean Coupon;
}
```

Our model is ready to store information about the purchase. We can assume the view is set up, too, with some form elements for the user to put in their name, payment

information, etc. Let's assume that all we need to do to charge the customer is call an API like this:

```
paymentGate.chargeEm();
```

Easy, isn't it? Even when those things are done for us, we still need a way to handle specific events in the view and coordinate all the logic. For example, what if we want to accept a coupon code from the customer? Even if those codes are stored in a database, we need a place to check and apply them to the final price. What if we want to automatically calculate discounts? Even if we're not interested in offering features (we just want to get the customer out the door), we need a place to charge the customer and *only* store the purchase if the charge goes through. That's where the controller comes in.

Generally, the controller will be linked to the view of an application so that it can handle events raised there, get input from the user, and ultimately display some response. For example, our controller could look something like this:

```
$view.onFormSubmit(function() {
  if (paymentGate.chargeEm()) {
    var newPurchase = new StorePurchase(...);
    newPurchase.save();
  }
});
```

The controller could also hold code for handling automatic discounts and other logic. The important thing to recognize here is that the controller is unlinked from the view—it is *not* an essential component. Even if we completely removed the controller, the user could still see the view as it would appear with a controller. Additionally, our data,

represented by the model, is also represented independent of the controller. Views and models willingly (and specifically) expose properties and methods that the controller should be able to access, and they allow it to do whatever it chooses with that information.

In general, this is a one-way street. For example, the controller can't tell the view when it's time to check out. The view *can*, however, provide a checkoutButtonClicked event to which the controller can subscribe and respond accordingly. Typically, the controller can also provide access to properties and methods that the view can access if it so chooses. The point is, the view has control over information flow.

That's the classic, original way of thinking that led to the development of frameworks. Many applications are still designed with the MVC pattern, but several popular variations exist. In fact, some of the frameworks we will discuss lend themselves more to other software design patterns.

Variations on Design Patterns

The important thing to denote here is that these patterns provide the developer with the tools they need to separate concerns, and they allow data to be considered independently of view layouts. They allow visual design to be decoupled from complex data processing. Some of the variations among patterns exist because some of the MVC components are unnecessary or can be utilized in different ways. For example, some frameworks emphasize a pattern that doesn't even have a controller. They package code back into the view but provide tools for doing so without blocking the UI thread. Other implementations take a two-way data binding approach, allowing the controller to directly manipulate and send data to the view. This approach allows the view and controller to be separated conceptually but connected in practice.

So What Makes the Web So Special?

The web varies from native application development quite a bit, and as such has always faced its own set of unique challenges. Aside from separating concerns, MVC and similar frameworks seek to simplify app development on the web by making it easier to **manage state** and **create dynamic views**.

Historically, it's been difficult to maintain state on a webpage, especially while dynamically updating elements in the view. In the past, modifying a view element has required reloading the webpage and, subsequently, losing state. Have you ever noticed your screen flashing while filling out a form on an older website? This is likely due to the "clunkiness" of handling state on the web.

Luckily, modern frameworks focus on using JavaScript to dynamically alter elements in the view by directly modifying the document object model (DOM). These frameworks abstract away the tough, tedious job of modifying the DOM directly. They provide developers with clear, efficient syntax for accessing utilities that make building dynamic apps on the web a breeze.

Here's a simple React.js demo showing how a developer can modify an existing a string and display it in the view:

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Christian',
  lastName: 'Gaetano'
};

const element = (
```

```
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

The code above displays this HTML element on the page:

```
<h1>Hello, Christian Gaetano!</h1>
```

That's how frameworks allow for easy manipulation of the DOM based on values "calculated" in code.

Other Utilities

In addition to simplifying software design and providing utilities for creating a dynamic view, JavaScript MVC frameworks typically provide other useful utilities. For example, almost all frameworks provide a built-in method for making AJAX calls to asynchronously retrieve data from APIs. Frameworks also provide engines (which can be overridden) for manually manipulating the DOM. Many frameworks offer utilities for interfacing with modern browsers via new APIs, such as local storage programming interface or web sockets.

Let's take one last look at the pros and cons of using a JavaScript framework.

Recap - General Pros and Cons of Framework Use

If some of these items seem unclear, skip to Appendix A, where TypeScript, ECMAScript, etc., are covered in detail.

Workflow

Pros

- Command line interfaces (CLIs) and project file management
- Deployment utilities
- Modular, so you can control how much weight frameworks add to your project
- Built-in shims and polyfills for older browser support
- Easily configurable unit testing
- Plugin support
- **UI libraries**: One thing I didn't get to talk about much yet is the usefulness of UI libraries. One of our projects, Wijmo, is a UI library that supports every JavaScript framework discussed in this e-book. Loading UI libraries into a framework project reduces developer workload and allows the workflow to focus on implementing new features rather than debugging UI controls.
- Automatic form logic handling
- Embedded media
- Content management and blogging
- Support for other JavaScript "flavors"
- TypeScript
- ES2015+

- CoffeeScript
- Elm
- Clojure

Cons

- Learning curve requires developers to learn ins and outs of new workflows
- Tooling and server setup requires time, and sometimes money

Performance

Pros

- More concise handling of complex UI functions like animation provides a smoother UX
- Non-blocking data operations via view-model binding
- Modularity means you can load only utilities you *need*
- Popularity on web means users may have CDN libraries pre-cached
- Some frameworks provide utilities for using web workers to improve performance

Cons

- Large libraries bring up bandwidth concerns - think the "next billion users"
- Client-side DOM manipulation dependent on client hardware

Features

Pros

- **Dynamic views!** This is the tenet of all JavaScript frameworks—they facilitate the creation of stateful, dynamic views
- Abstract data modeling and binding allows for separation of concerns
- Fun, useful utilities specific to each framework, such as animations, page transitions, and routing

Examining Popular Frameworks

But First, a Message from Our Sponsors

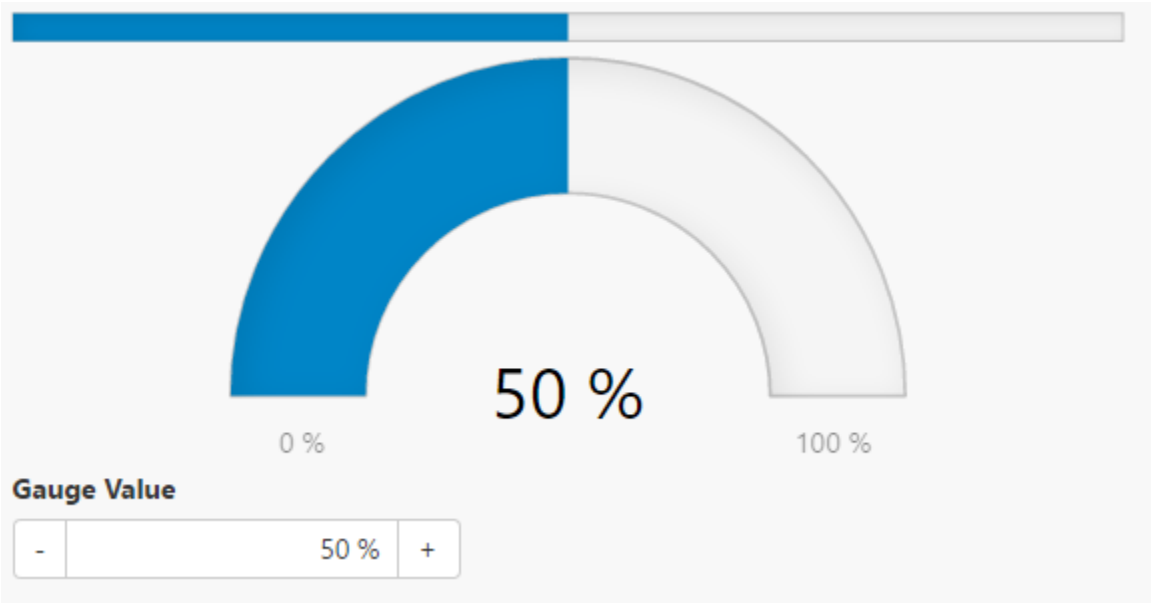
Before we get into the frameworks, let's look at how I use Wijmo to compare the implementation of different frameworks. Most JavaScript frameworks support a wide array of plugins, many of which are libraries of UI controls. Since the goal of most frameworks is to make working with a dynamic view easier, most frameworks differ in how they interact with the view.

For this reason, UI libraries and their controls are one of the most useful and obvious examples for comparing how different frameworks work. Wijmo is the only UI framework that ships with support for Angular v1 and v2, React.js, VueJS (1 and 2) and Knockout, all of which will be discussed in this e-book. Wijmo is a natural choice if you're looking for a fair platform to compare framework implementation. I can be cheap at times, but I'm not using Wijmo just because I have free access to it.

If you'd like to use Wijmo to follow along, you can download a free trial at any time.

Structure

To see how each framework handles the fundamental tasks of building a web app, we'll look at how frameworks can interact with the simple Wijmo gauge control:



2. A simple layout showing a Wijmo linear gauge control, radial gauge control and number input all bound to the same data value. We'll look at this example for each framework.

I can't stress enough that this example only demonstrates the syntactical differences between frameworks. The most important differences between frameworks are conceptual, and those examples would require building entire applications. In addition, not all frameworks are designed equally: some frameworks weren't really designed to handle jobs that other frameworks solve brilliantly. I'll focus on the conceptual differences between frameworks so that you can **choose a framework for your project**.

Pure JavaScript

Let's prepare a Wijmo gauge control with a number input that adjusts it.

HTML (*view*)

```
<!-- LinearGauge -->
<div id="gsLinearGauge"></div>

<!-- RadialGauge -->
<div id="gsRadialGauge"></div>

<!-- InputNumber -->
<div>
  <label>Gauge Value</label>
  <input id="gsValue" type="text" />
</div>
```

JavaScript ("*controller*" - technically a part of the view since we're in Pure JS)

```
// init Wijmo controls
var linearGauge = new wijmo.gauge.LinearGauge('#gsLinearGauge'),
    radialGauge = new wijmo.gauge.RadialGauge('#gsRadialGauge'),
    valueInput = new wijmo.input.InputNumber('#gsValue');

// LinearGauge - set properties
linearGauge.value = props.value;
linearGauge.min = props.min;
linearGauge.max = props.max;
linearGauge.format = props.format;

// Radial Gauge - set properties
radialGauge.value = props.value; radialGauge.min = props.min;

radialGauge.max = props.max;
radialGauge.format = props.format;

// InputNumber - set properties
valueInput.value = props.value;
valueInput.min = props.min;
valueInput.max = props.max;
valueInput.format = props.format;
```

```
  valueInput.step = props.step;

  // InputNumber valueChanged event
  valueInput.valueChanged.addHandler(function (sender) {

  // update Gauge.value when InputNumber.value changes
  linearGauge.value = sender.value;
  radialGauge.value = sender.value;
});
```

As you can see, the DOM must be accessed manually and the Wijmo controls initialized through them. We must also override the valueChanged event of the number input to manually change the gauge values—there is no true data binding.

Let's see how frameworks can simplify things.

AngularJS, aka Angular v1: The Choice for Design-Focused Teams Working with Dynamic Data

Note: As of 2017, Google has stated they're branding a single Angular framework with different "versions"—i.e., AngularJS is now Angular v1, and Angular 2 is Angular v2.

Premise and Main Concepts

Premise: HTML is intrinsically flawed in that it is not *designed* to represent dynamic views. The solution to this problem is to extend HTML to support dynamic web content.

AngularJS is one of the most popular JavaScript frameworks, and it was also one of the first. Google released AngularJS in 2010, and it has since been joined by its younger (hipper, edgier) sibling, Angular version 2. While some would claim that Angular v1 has been replaced by Angular v2, the more accurate view is that the two (happily) coexist. As you'll see, Angular v1 and Angular v2 work very differently, and Angular v1 is still being used for new projects.

The focus of AngularJS has always been "fixing" HTML to better support data binding and dynamically updating the view. Since the view is flawed, the best place to implement a fix is in the view itself. This becomes apparent when viewing the Angular v1 syntax, as it relies almost entirely on markup directives. For example, Angular v1 specifies that every application should have a controller, but even the controller is referenced in the application controller via a directive in the view. Therefore, model and controller constructs in Angular v1 are a result of supporting the extended HTML vocabulary.

Despite Angular v1's focus on fixing HTML, there is no imperative manipulation of the DOM: one can choose which parts of their view should be handled by Angular v1 by declaring them with the Angular directives. Furthermore, any code from the controller that interacts with the view must be bubbled up through global variables which Angular maintains (namely, `$scope`). Overall, this methodology emphasizes Angular's focus on improving and extending current tools rather than replacing them.

Use Cases

Because of its hyperfocus on HTML, Angular v1 is best utilized in applications that have a single fundamental need for creating a dynamic view. Angular v1 provides extremely effective functionality for building a dynamic view while remaining lightweight and unobtrusive. This is true both for the end-user and the developer.

Thus, Angular v1 is a great option for teams that want to stay quick and lightweight while adding new features and facilitating dynamic web form development. The learning curve for developers who already know HTML and JavaScript is low, so teams don't need to dedicate much time to transitioning their systems to Angular v1. I personally think

that Angular v1 is a great option for teams managing existing projects that focus on design and copy but now have the requirement of working with dynamic data.

Another strong suit of Angular v1 is its versatility. Because of how it works, Angular v1 can usually be easily mixed in with other frameworks and libraries.

Angular v1 in the Wild

Too many sites to count! WolframAlpha, NBC, Walgreens, and Intel all use Angular v1. Check out the full, official list.

Migration from Pure JS

The Gauge Example, Rewritten

Here's the Angular v1 version of the Wijmo gauge example:

HTML (*view*)

```
<body ng-app="app" ng-controller="appCtrl">

    <!-- Wijmo 5 LinearGauge directive -->
    <wj-linear-gauge
        value="props.value"
        min="{{ props.min }}"
        max="{{ props.max }}"
        format="{{ props.format }}">
    </wj-linear-gauge>

    <!-- Wijmo 5 RadialGauge directive -->
    <wj-radial-gauge
        value="props.value"
        min="{{ props.min }}"
        max="{{ props.max }}"
        format="{{ props.format }}">
    </wj-radial-gauge>

    <!-- Wijmo 5 InputNumber directive -->
    <div>
    <label>value</label>
```

```
    <wj-input-number
        value="props.value"
        min="{{ props.min }}"
        max="{{ props.max }}"
        format="{{ props.format }}"
        step="{{ props.step }}">
    </wj-input-number>
    </div>

</body>
```

JavaScript (*controller*)

```
// declare app module
var app = angular.module('app', ['wj']);

// app controller provides data
app.controller('appCtrl', function appCtrl($scope) {
    $scope.props = {
        value: 50,
        min: 0,
        max: 100,
        step: 10,
        format: 'n0'
    };
});
```

As you can see, the migration from Pure JS to AngularJS isn't too complicated. In fact, most of the code is taken away from the JavaScript altogether and moved to the markup. In a nutshell, that's the point of a framework.

Wijmo automatically provides special Angular directives for its components, allowing them to be easily loaded from the markup. All that's left to do is hook up the app to the controller's JavaScript and provide a data object that the controls can use (called `props`) to store and retrieve data. (Note that `$scope` is used to make `props` accessible in the view.) We don't even have to handle any events manually this time—Angular v1 automatically handles data changes for us—the epitome of “data binding”.

The main changes that you'd need to make when moving your Pure JS project to Angular v1 are enrolling your HTML elements in data binding and refactoring view logic to use `$scope`.

Feature Recap: Angular v1

- Lightweight and fast
- MVC architecture
- Two-way data binding
- Templates
- Directives
- Expressions
- Filters
- User-controlled DOM manipulation
- DOM Scope
- Deep linking
- Dependency injection

I recommend Angular v1 if you’re planning a new web app that requires only a shallow dynamic view integration, or if you’re retrofitting a more streamlined dynamic view framework to an old application.

Angular 2: When You're Looking for Full Team Workflow

Premise and Main Concepts

If Angular v1 is the lightweight champion of the framework arena, Angular v2 is its much "beefier" heavyweight counterpart. Even though Angular v2 retains some of the core philosophies of Angular v1, the differences are substantial.

Premise: Many companies are choosing to replace native apps with web apps. To ease that transition, the mindset and tooling that has been a hallmark of the native app development sphere should be available for the web. This gap in the development workflow transition can be filled by providing a full-fledged architecture for building responsive apps on the web.

That premise makes Angular v2 completely different from its older sibling. It's also why Angular v1 is still actively used and maintained; Angular v2 is not meant for everyone.

Fundamentally, Angular v2 is about providing a full-featured solution. Rather than acting as a lightweight framework that you can incorporate into your app, Angular v2 is designed to be a complete workflow solution for creating and maintaining your app from the beginning. One of the most prominent features of Angular v2 is the availability of a CLI. This isn't unique among frameworks, but it is important. The CLI allows you to quickly generate a file structure, package configuration, and other tedious elements for a new project. The CLI works with you throughout the project lifecycle, so you can continually add, remove, and modify different components in your application from the command line. This typifies the "holistic solution" approach taken by Angular v2.

By default, Angular v2 ships with much more going on under the hood. Where Angular v1 focused on providing a simple linkage between different components of an app, Angular v2 strives to provide an entire system that you can utilize to handle events, bind data, and more.

The holistic approach's major advantages involve abstracting away performance and overall speed concerns. While you still manage bandwidth and resource delivery on your own, Angular v2 automatically handles the job of optimizing code for working with large datasets and managing the DOM. These technologies still have intrinsic limitations, but the Angular v2 API utilizes the latest and greatest JavaScript features to run your code as efficiently as possible.

Angular v2 also departs from Angular v1 in terms of how the view is built and how view interactions are handled. Remember, Angular v1 focuses on "fixing" HTML by extending it. Angular v2 would probably agree with that sentiment, but it provides a different route to a "fix." In Angular v2, views are built as a collection of templates. You can think of a template as a custom HTML element in some ways. Rather than declaring a custom directive and mutating it via attributes (as in Angular v1), templates typically contain their own code, which allows them to mutate the view directly. When templates also contain their own logic, they are considered Angular v2 components. (As an example, the Wijmo controls are provided as Angular v2 components via an interop module.)

This difference between templates and components is significant, and components are one of the most powerful features of Angular v2. Rather than declaring some custom directive and then handling its functions in the general app controller, Angular v2 compartmentalizes each component's logic. This not only makes working in a team setting easier, but also provides performance advantages since you can selectively load

the code that is needed on a page at any given time. This takes some getting used to, but if you're coming from native programming it's probably familiar. Ever worked with a custom user control in .NET or a custom component in Android development? Components in Angular v2 work on the same premise.

Oh, and one more thing: of *course* they allow inheritance! Does it sound like JavaScript is getting kind of object-oriented in the Angular v2 context? You haven't heard the half of it yet.

TypeScript and Angular v2

AUTHOR'S NOTE: TypeScript is not required when using Angular v2. You can still write Angular v2 apps using good ol' JavaScript! But many people now consider Angular v2 and TypeScript synonymous, so we should talk about it.

TypeScript is a programming language developed and maintained by Microsoft, but it's a little odd in the way it works in the web field. Most programming languages are either compiled down into more low-level syntax or directly interpreted by an engine. Historically, on the web, the only major programming language for the client has been JavaScript, and JavaScript has used a combination of compilation and interpretation to run. TypeScript, on the other hand, is compiled into JavaScript which is *then* compiled/interpreted as usual.

If TypeScript ultimately turns into JavaScript, what's the point?

It's all in the name: *Type*Script is about types. Every developer knows a major pain point of JavaScript is its typing mechanism. In JavaScript, values have types but variables do not. Variables are simply containers for values, so the value (and its type!) associated with any given variable can change freely. TypeScript's original aim was to address this

issue head-on by designing a JavaScript-like language with type enforcement for variables.

In TypeScript, it's possible to create function definitions which return a specific type *and* take parameters with specific types. This approach allows for applications to grow larger, quicker, while also being properly maintained. (I feel that JavaScript's typing strategy is powerful when used correctly, but that's a topic for another e-book.)

In addition to adding type enforcement to code, TypeScript also provides many other patterns seen in the OOP world, such as inheritance and class constructors. Furthermore, since TypeScript compiles into JavaScript, it takes the guesswork out of JavaScript compatibility for developers and their workflows. For example, three development teams can target three different browser platforms while working on the same TypeScript codebase. When it comes time to package up the distributable files, each team can specify a JavaScript version it needs, and the TypeScript compiles to that version. There are caveats, but TypeScript will also help you work through those.

Where does TypeScript fit in with Angular v2?

By definition, TypeScript and Angular v2 are not linked, and neither *requires* the other. But Angular v2's strategies lend themselves to OOP patterns, especially typing and inheritance. For example, it's possible to write one Angular v2 component that inherits from another, and TypeScript naturally handles this paradigm. JavaScript can handle the same structure, but the meaning gets muddled, and the code becomes harder to understand.

Use Cases

Angular v2 is most suited to large-scale enterprises looking to create new web apps. Whether these apps replace old, native versions or provide some new functionality, Angular v2 is ready to take on the job. The availability of TypeScript and the inclusion of major workflow tools makes it easy for even the largest development teams to manage immense codebases. By designing its base units as components, Angular v2 also allows for development efforts to become highly compartmentalized.

Additionally, Angular v2's comprehensive form templating sets it apart in the framework world. This, combined with the workflow advantages, has propelled Angular v2 to the forefront of frameworks chosen by business enterprises. When it comes to building internal apps that require lots of user input, Angular v2 does the job best.

Even though Angular v2 facilitates moving away from old native applications, it provides officially-supported plugins that can be used to deploy an Angular v2 app as a native desktop or mobile application. While not as feature-rich as the web counterpart, the availability of native deployment is appealing to many organizations looking to future-proof their apps.

Overall, I think **Angular v2 can be epitomized as a *developer platform* rather than a simple *coding tool*.**

Angular v2 in the Wild

Companies like Thomson Reuters are using Angular v2 to serve their customers and manage their own operations. Here at GrapeCity, we've rewritten our entire Wijmo component suite in TypeScript to allow optimal integration with Angular v2 and easy maintenance going forward.

If you'd like to check out a more comprehensive list of companies using Angular v2, check out Google's official list.

Migration from Pure JS

The Gauge Example, Rewritten

Here's a simple Angular v2 version of the Wijmo gauge example:

HTML (app.html - *view*)

```
<body>
  <div class="container">
    <wj-linear-gauge class="lineargauge"
                     [(value)]="gauge.value"
                     [isReadOnly]="false"
                     [step]="gauge.step"
                     [min]="gauge.min"
                     [max]="gauge.max">
    </wj-linear-gauge>
    <wj-radial-gauge class="radialgauge"
                     style="display:block"
                     [(value)]="gauge.value"
                     [isReadOnly]="false"
                     [step]="gauge.step"
                     [min]="gauge.min"
                     [max]="gauge.max">
    </wj-radial-gauge>
    <wj-input-number [(value)]="gauge.value"
                     [min]="gauge.min"
                     [max]="gauge.max"
                     [format]="'n0'">
    </wj-input-number>
  </div>
</body>
```

TypeScript (app.ts - *controller*)

```
// Angular
import { Component, EventEmitter, Input, Inject, enableProdMode, NgModule}
from '@angular/core';
import { CommonModule } from '@angular/common';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { BrowserModule } from '@angular/platform-browser';
```

```
import { WjInputModule } from 'wijmo/wijmo.angular2.input';
import { WjGaugeModule } from 'wijmo/wijmo.angular2.gauge';
'use strict';

// The application root component.
@Component({
    selector: 'app-cmp',
    templateUrl: 'app.html'
})

export class AppCmp {

    gauge = {
        min: 0,
        max: 100,
        value: 25,
        step: 5,
        angles: [
            { start: -45, sweep: 270 },
            { start: 10, sweep: 340 },
            { start: 0, sweep: 90 },
            { start: 45, sweep: 90 }
        ]
    };

    constructor() {
    }
}

@NgModule({
    imports: [WjInputModule, WjGaugeModule, BrowserModule],
    declarations: [AppCmp],
    bootstrap: [AppCmp]
})
export class AppModule {
}



enableProdMode();
// Bootstrap application with hash style navigation and global services.
platformBrowserDynamic().bootstrapModule(AppModule);
```

Note: This example does not use any separate TypeScript component files. To further customize each of the displayed controls, you could create *.ts files for each of the components (the two gauges and one number input) and customize them there before loading them in the main app component.

If you're coming from a Pure JS environment, the most shocking change in Angular v2 is the focus on components and objects. JavaScript is highly procedural in nature, so any OOP language seems foreign when viewed through that lens. If you opt to avoid TypeScript, using JavaScript to interact with Angular v2 components will get confusing at times. And even if you do move to TypeScript, making the leap to a new programming language is likely the most difficult factor you'll face.

You'll also need to refactor code and markup so that they can be compartmentalized into components. This ultimately simplifies the development of your app and provides built-in performance improvements.

In the end, Angular v2 is a much easier framework to choose if you're starting work on a *new* app, but fitting it around the codebase of a preexisting app can be much more difficult. If you need to add some dynamic views to an old app, I suggest using Angular v1. If you need to add so many new features that starting from scratch is sensible, then Angular v2 is a viable option.

If you opt for Angular v2, stick with TypeScript even when it seems too different from the classical method of writing code for the web. Angular v2's and TypeScript's first-class integration with IDEs like Visual Studio make learning these new methodologies much easier, and the benefits justify the change management. If you're building an enterprise-level app staffed by very large teams, Angular v2 is a natural choice.

Feature Recap: Angular v2

- TypeScript support offers OOP patterns that compile into JavaScript
- Plugins offer native app deployment
- New browser and backend APIs supported out-of-the-box: web workers, multiple server types
- Code splitting via components and built-in view routing for multi-page apps
- CLI for creating project structure
- Highly compartmentalized
- Full development platforms offered including snippets, autocomplete, unit testing, animation APIs, and more

React.js: You Have a Need for Speed

Premise and Main Concepts

Premise: In the Pure JS days of yore, creating a dynamic view was made possible only by manually mutating the DOM via JavaScript. This approach works in many environments, but it's confusing, indirect, and makes it difficult to maintain state or bind data. The solution to this fundamental problem is to combine DOM markup with JavaScript logic in a single easily-generated and maintained package.

Based on various surveys and metrics analyses, React.js is currently the most popular front-end JavaScript framework. React owes its success to its subtle learning curve and easy integration. React.js is a highly pluggable framework that includes fundamental features of a JavaScript framework while staying very lightweight.

React.js is so lightweight, in fact, that it only focuses on the view of an application, leaving data handling to the developer. On the enterprise level, this can add time to development and may be a disadvantage. Overall, however, it offers the most pluggability and versatility of any framework, and it helps you stay true to the original focus of frameworks: dynamic views.

Like Angular v2, React's focus is on highly compartmentalized components, but that's where the similarity ends. React components are more lightweight and true to JavaScript than those in Angular v2. React doesn't provide a whole new programming language; instead, it extends JavaScript to make authoring components easier.

React uses a syntax known as JSX that essentially allows HTML to exist as a JavaScript variable. For example, the following is valid JSX:

```
var someEle = <h1>Hello, React!</h1>;
```

React places some special rules and restrictions on what constitutes valid JSX, but the main point is that *HTML can live within JavaScript in React*. This is the tenet of React.js, and it's the pinnacle of compartmentalization in the web world. Each piece of the view can be separated out as a component, and those components simultaneously contain both the markup *and* the JavaScript logic necessary to display and operate as expected. React's JSX components can even contain style information saved as JavaScript objects! And in React, *everything is a component*. Each control can be represented as a component; even containers on the page can be components. The *app* is a component. Bottom line: components are pretty important in React.

Since everything is a component in React, how do components communicate with one another or maintain state? One word: inheritance. Through its own defined hierarchical

system, React allows components to inherit from one another, although “inheritance” is a slightly altered concept.

Typically, one component will import another component’s definition, and then render that imported component within its **render()** function, thus allowing two-way communication between React components. Let’s dig deeper by understanding the difference between **state** and **props** in React.

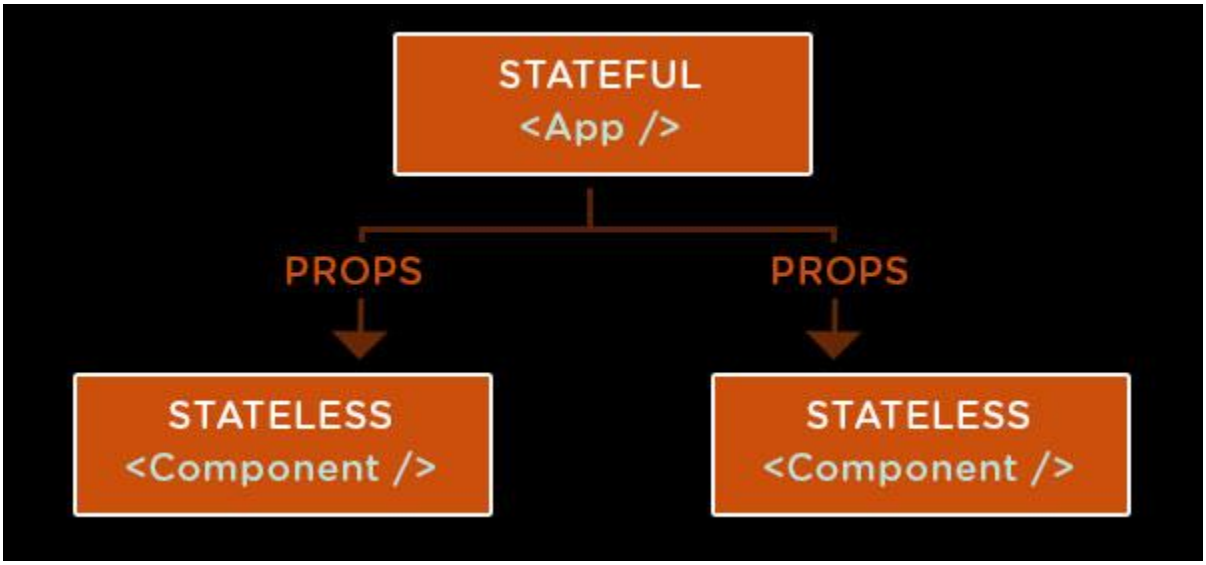
Props are object properties that are passed to a component via attributes, and they can be accessed on an object located at `this.props`. For example: suppose you imported a component called CustomButton into your React component (called CustomForm), and rendered it like so:

```
ReactDOM.render(<CustomButton reaction="happy" />);
```

In that example, the CustomButton component now has access to the 'reaction' property via `this.props.reaction`.

A similar system can be used for events and event handlers. For example, imagine a parent component defining an event handler function and then passing a reference to that function to the child component as a prop. The child component could then set the function reference as an event handler.

State works very similarly on the component level in React, but the React framework handles it differently in the background. A stateful component is indicated by defining a `getInitialState()` function on a component. This function should always return an object containing properties that will define the component's starting state. Then, from within the same component, state can be modified using `this.setState()`. One interesting facet of handling state in React is that state items can be modified individually, even though an object must always be passed or returned as the state



3. A generic visualization of a React.js programming pattern in which a stateful parent component is used to maintain communication between stateless child components.

parameter. React automatically aligns object property names and updates the supplied properties.

Since state can only be modified from within a component, a typical pattern in React is to provide a stateful "container" parent component with two stateless child components. The parent then exposes functions which alter or reveal state to the child components via props, and the child components access them as needed. Typically, one

child component is used to update the state and one is used to display the state (or respond to its value).

This concept can be a bit hard to wrap your head around at first, but React is actually the simplest framework in this text. The inheritance structure and simple state management allow React to become extremely powerful and versatile if needed, or subtle and convenient if only minor integration is required.

Another major consideration for React is the speed. React is *fast*. Like some other frameworks, React uses a virtual DOM, so the real DOM is only updated when it's absolutely necessary. React's far-reaching compartmentalization and component hierarchy optimize its virtual DOM handling—so much so that it performs noticeably faster than most frameworks. Because React is such a small and versatile tool, and because components insert *themselves* into the DOM using JavaScript, it can scale up and down to be as involved in your application as you want it to be.

Even though its lightweight nature makes it less of a workflow platform and more of a tool, React's support from Facebook and popularity in the community mean that it's surrounded by a rich ecosystem. Plugins like react-router and relay simplify view routing and data modeling, while countless others provide powerful UI base components and more. Like Angular v2, React has powerful plugins for developing native applications—perhaps the best in the frameworks arena.

Use Cases and React in the Wild

React is extremely versatile, and it scales well in both directions. Whether you're building a new, lightweight web form for a growing small business or a massive enterprise-level data management system, React can meet your needs. If you're looking for an out-of-

the-box workflow platform, it requires a bit more work. If you have time to set up tooling yourself, React can be configured to become a development partner that rivals Angular v2.

React in Native Mobile and Desktop

If you're considering a JavaScript framework to build a native mobile or desktop app, React.js is your best bet. The same speed and performance offered by React's web libraries are also represented in the native sphere. Furthermore, React Native strives to keep up with native mobile standards on Android and iOS, providing a UX that users expect. Apps like Airbnb and Instagram – which together have amassed *billions* of downloads across mobile platforms – are built using React Native.

If you want to check out a full listing of apps built with React, head to the official website.

Migration from Pure JS

The Gauge Example Rewritten

Here's the React.js version of the Wijmo gauge example:

App.js (JSX - *view*)

```
ReactDOM.render(
  <div>
    <Wj.LinearGauge
      value={ this.state.value }
      min={ this.state.min }
      max= { this.state.max }
      format={ this.state.format} />
    <Wj.RadialGauge
      value={ this.state.value }
      min={ this.state.min }
      max= { this.state.max }
      format={ this.state.format} />
```

```
    <div className="app-input-group">
      <label>value</label>
      <Wj.InputNumber
          value={ this.state.value }
          valueChanged={ this.valueChanged }
          min={ this.state.min }
          max= { this.state.max }
          step= { this.state.step }
          format={ this.state.format} />
    </div>
  </div>
);
```

Component.js (Example of code that would be included in each component)

```
getInitialState: function () {
    return {
        value: 50,
        min: 0,
        max: 100,
        format: 'n0',
        step: 10
    }
},

// Wijmo event handler
valueChanged: function (s, e) {
    this.setState({ value: s.value });
}
```

When migrating to React.js, the most obvious change is compartmentalizing all current markup and code into components. If you're trying to get an old app up and running with React, then it's okay to simply set state on the parent app component. If you're designing a *new* app, you may want to put some more thought into the hierarchy of the app and which components need to share state, etc.

At the most basic level, after separating your app out into components, you'll use the react and react-dom libraries to create the components (`React.createClass()`) and ultimately render them (`ReactDOM.render()`).

Ultimately, transitioning to React.js from Pure JS isn't too difficult, as React stays true to the core principles of JavaScript—it simply extends them a bit to support the creation of components.

Feature Recap: React

- Lightweight, fast, and scalable due to compartmentalization and virtual DOM
- Well-documented design patterns and simple philosophy
- Very rich ecosystem provides great plugin support for any deployment and build scenario
- High modularity simplifies integration into existing apps
- Supports server-side rendering for offloading computationally intense tasks

Vue.js: A Pared-Down Framework for the Minimalist

Premise and Main Concepts

Note: This section refers to Vue 2, since that's now the officially-supported version. This isn't a situation like Angular v2, where many people still use the first version. Most developers will be upgrading to Vue 2 for its enhancements.

Premise: Angular and React are well-established. They're widely used, have had multiple releases, and are built around the philosophy of making feature-adds easier for the user. They have their own learning curves, and complexity ranges from "somewhat" (React) to "very" (Angular 2). While Vue acknowledges that JavaScript frameworks bring a lot to the table in terms of features and workflow advantages, it considers the countless additions and updates to other frameworks to be a source of "feature creep" that ultimately obscures the ultimate goal. Thus, Vue focuses on *minimalism,* providing a

simple framework with very few built-in utilities that simply maps JavaScript objects to the view scope and handles specialized DOM tags.Vue is a bit of a latecomer to frameworks. It wasn't until its 2.0 release in September of 2016 that it began generating interest, and since then, it's been steadily climbing: a recent survey shows that more developers have said they're interested in learning about and trying Vue.js than any other framework.[2]

Why? Firstly, Vue is incredibly pluggable. The core of Vue.js works great on its own, but it's so minimalist that you'll likely *have* to use a plugin at some point in your Vue application's development lifecycle. Luckily, the Vue team maintains the major plugins themselves, making them easy to find and use. Interestingly, many of these official plugins support features shipped with other frameworks. To illustrate just how much Vue.js draws from other frameworks, consider its out-of-the-box JSX and rendering support. And you can easily add TypeScript support. Sound familiar?

Vue isn't stealing from other frameworks; it's learning from them. At its core, Vue.js is quite different from other frameworks: it utilizes **templates** as fundamental units that define dynamic behavior with a specialized syntax composed of Vue attribute tags and mustache-style variable replacement. This varies from JSX (which the Vue team concedes is sometimes necessary), as it keeps HTML and JavaScript separated. That split adds major readability and workflow enhancements, like allowing developers to use HTML pre-processors and more familiar CSS styling.

---

[2] "State of JS" by Sacha Greif. http://stateofjs.com/

Use Cases

Another major focus of Vue.js is a "scale-as-you-go" experience that encourages iteration-based app development. You can either create a new Vue app or transition an existing JavaScript project just by loading a single script on your page. By installing the officially-supported CLI—along with a host of other plugins that support features like routing and advanced state management—that single-script experience scales up to a fully-integrated workflow.

Overall, I agree with the development community that Vue is the most exciting up-and-coming framework in the JavaScript field right now. It packs plenty of features into a modular, customizable package while also maintaining speed and performance (it's even faster than React!). The Vue development team put together a thorough and helpful comparison guide that compares Vue to other frameworks.

Vue offers some exciting feature opportunities and performance gains for experienced framework users, and the straightforward JavaScript object syntax makes it easy to pick up for Pure JS developers as well. In the end, the only reasons Vue might not work for large enterprise applications are a lack of support for older browsers and the extra steps involved in setting up a full-on workflow.

If you're a developer building a cutting-edge web app, part of a startup, or have experience with other frameworks and are looking for something new, give Vue.js a try.

Vue.js in the Wild

While I wasn’t able to find a centralized list of sites using Vue, I did find a few companies and sites who use Vue in some capacity:

- GitLab
- Livestorm
- Citymoods

Vue is still trying to gain some traction and grab market share from the other frameworks that have been around for longer. Based on the amount of interest Vue has been generating, though, it won't be much longer until we see many more websites built with it.

Migration from Pure JS

The Gauge Example, Rewritten

Here's the Vue.js version of the Wijmo gauge example:

HTML (*view*)

```
<div id="app">
    <wj-linear-gauge
        :value="value"
        :min="min" :max="max"
        :format="format">
    </wj-linear-gauge>
    <wj-radial-gauge
        :value="value"
        :min="min" :max="max"
        :format="format">
    </wj-radial-gauge>
    <div class="app-input-group">
        <label>value</label>
        <wj-input-number
            :value="value"
            :value-changed="valueChanged"
            :min="min" :max="max"
            :format="format"
            :step="step">
        </wj-input-number>
    </div>
</div>
```

JavaScript (*controller*)

```
var app = new Vue({
    el: '#app',
    data: {
        value: 50,
        min: 0,
        max: 100,
        format: 'n0',
        step: 10
    },
    methods: {

        // Wijmo event handlers
        valueChanged: function (s, e) {
            this.value = s.value;
        }
    }
});
```

The great thing about Vue and its scalable, pluggable philosophy is that migrating from Pure JS can be as complicated or as simple as you want it to be. On the simplest level, you can import the Vue core script (which only adds about 23kb *with* the built-in compiler, by the way) and then map any unmapped existing data structures to JavaScript objects. Then use JavaScript to initialize the Vue instance in a DOM element, and you're ready to go.

Since Vue handles templating, data binding, and other directive-like markup behavior with plain JavaScript, transitioning your current JavaScript app to use the more concise and performant Vue systems is usually straightforward. The other advantage to using Vue when migrating is that you can specifically target *pieces* of your app for stepwise migration, allowing you to switch over and test one section of an app at a time.

Feature Recap: Vue.js

- Relatively small core compared to other frameworks (23 kb for Vue 2 core *with* compiler)
- Modularity is key with many officially maintained plugins
- Virtual DOM even faster than React's in most cases
- CLI via official plugin
- Routing via official plugin
- Advanced state management via official plugin
- TypeScript and JSX supported but optional
- Readily scales in both directions
- Clear separation between templates and markup/attribute directives
- Built-in transitions and animations
- Server-side rendering supported

Knockout: Build a Plug-In with Data Binding

Premise and Main Concepts

Premise: When you boil down JavaScript frameworks, the focus is always on data binding and dynamically updating a view. Knockout strives to be the ultimate minimalist framework by providing data binding capability without any additional features or overhead.

If Vue.js is where minimalism stands now, Knockout is where it started. Knockout is still updated and very widely used today, and its longevity can be attributed to its simplicity. In many ways, Knockout can be considered more of a library than a framework. It is *not*

a full-featured workflow solution, and it's not trying to be. If you're looking for a small plugin that you can mix into your current JavaScript application to get easy data binding, look no further than Knockout.

Knockout is simple. To set up data binding and dynamic views in Knockout, you need one HTML attribute, a simple JavaScript data structure, and even simpler JavaScript logic facilitated by the Knockout library. That's all! Knockout takes data binding options from the markup as defined by a `data-bind` attribute, uses those options to hook up some data stored in JavaScript to the DOM, and ultimately applies the data binding when you tell it to.

In addition, Knockout has extensive browser support (as in IE 6+), and it easily integrates with other JavaScript frameworks. It’s also agnostic of backend data sources, as long as the data provided can be formatted as a JavaScript object. So the real appeal of Knockout is that you can take an existing .NET web app, for example, and give it a dynamic view supported across all browsers in a matter of minutes.

Use Cases

The best use case for Knockout is when you're adding data binding to a preexisting application. Knockout can also be used for new projects that require comprehensive browser support, but keep in mind you'll have to add some other solutions if you want to automate other aspects of your app, like view routing. The good thing is that Knockout is so compact and concise that you won't have to worry about it interfering with the operation of other frameworks, so it’s a good place to start if you're unsure about which of the more full-featured frameworks would be right for you.

Knockout in the Wild

Because Knockout has been around for a while, and because it can operate alongside other frameworks and systems, it's being used in a plethora of projects out there on the web:

- Microsoft Azure
- JSFiddle
- AMC Theatres
- BMW USA

These large, complex websites use Knockout to take care of specific, pin-pointed data binding tasks. This is where Knockout really shines.

Migration from Pure JS

The Gauge Example, Rewritten

Here's the Knockout version of the Wijmo gauge example:

HTML (*view*)

```
<!-- Wijmo 5 LinearGauge directive -->
<div data-bind="wjLinearGauge: {
        value: props.value,
        min: props.min,
        max: props.max,
        format: props.format }"></div>

<!-- Wijmo 5 RadialGauge directive -->
<div data-bind="wjRadialGauge: {
        value: props.value,
        min: props.min,
        max: props.max,
        format: props.format }"></div>
```

```
<!-- Wijmo 5 InputNumber directive -->
<div class="app-input-group">
    <label>Gauge Value</label>
    <div data-bind="wjInputNumber: {
            value: props.value,
            min: props.min,
            max: props.max,
            format: props.format,
            step: props.step }"></div>
</div>
```

JavaScript (*controller*)

```
// create and apply application view model
function viewModel1() {
    this.props = {
        format: 'p0',
        max: 1,
        min: 0,
        value: ko.observable(0.5),
        step: 0.25
    };
};

(function () {
    ko.applyBindings(new viewModel1());
})();
```

Out of all the frameworks we've looked at, Knockout offers the smoothest transition from a pure JavaScript project. All you need to do is load the Knockout library, add a data-bind attribute to any databound HTML elements, and supply some data via a ViewModel from JavaScript.

The transition can be even easier if you're using standard HTML elements that are already managed using something like jQuery. Simply add the data-bind attribute (if you have data to bind to) and replace the jQuery event bindings with plain JavaScript objects and a call to **ko.observable**.

Feature Recap: Knockout

- 13 kb after compression and gzipping
- Supports virtually all browsers
- Written in pure JavaScript; can integrate with other frameworks
- Templates are supported but not required for advanced function
- Simple to learn and use
- Provides methods for dependency tracking in JavaScript

Summary

- **AngularJS** is a great starter framework if you want to add dynamic views to your application while also supporting custom components without compartmentalizing everything in your app. Challenge to transition from Pure JS: Intermediate
- **Angular v2** offers the most complete development workflow experience out of the frameworks discussed, but sacrifices development speed and size for it. Challenge to transition from Pure JS: Difficult
- **React.js** comes with the richest ecosystem offered by any framework and it offers considerable performance, compartmentalization, and state management, but everything must be a component. Challenge to transition from Pure JS: Intermediate
- **Vue.js** is the newest and the most broadly scalable of the frameworks discussed, making it a great fit for businesses of any size, especially growing ones—you don't have to worry about browser support. Challenge to transition from Pure JS: Intermediate

- **Knockout** provides the simplest solution to building a dynamic view with data binding and is small and flexible because of it, but lacks the workflow tools offered by the other frameworks. Challenge to transition from Pure JS: Easy

Choosing a Framework and Migration Concerns

While you can apply *any* framework to any application and development scenario, the evaluation methodology outlined below reflects my recommendations based on data collected from various sources. I draw on my firsthand experiences from working at GrapeCity, where we've gone to great lengths working with customers and framework teams to make Wijmo the only UI library to fully integrate with the five frameworks talked about here.

SPECing Out Your App

The **SPEC** ranking method, as I call it, requires you to rank these four criteria:

- **S**peed
- **P**roductivity
- **E**cosystem
- **C**ompatibility

Let’s look at a few key statements:

- No matter which framework you pick, you'll be using a top-notch development tool.

- The SPEC method requires you to think deeply about your project. You can't weigh all four elements equally; you must choose *one* and provide more focus to your developers.
- This method **does not** distinguish between Angular v1 and Angular v2 (more on why later). In brief, this method fits a framework to a project, not necessarily to a team. If you end up picking Angular, you'll need to take an extra step to assess which version is best for your team.

The SPEC Method

1. Rank the following in order of priority for your project and its development:
    - **Speed**: How important is your app's performance, especially in terms of client-based logic operations?
    - **Productivity**: How important is the degree of workflow management and project structure provided from the outset of development?
    - **Ecosystem**: How important is your development team's access to community support and plugins?
    - **Compatibility**: How important is it that your app can reach the most browsers on the most devices?
2. Use the lists below to formulate your choice. Each list represents one of the SPEC properties and ranks the framework options in respect to that property. The simplest way to decide is to find the list for your top priority selection and choose the #1 ranked framework.

- **Speed**
    1. Vue.js
    2. React.js

3. AngularJS and Angular v2
4. Knockout

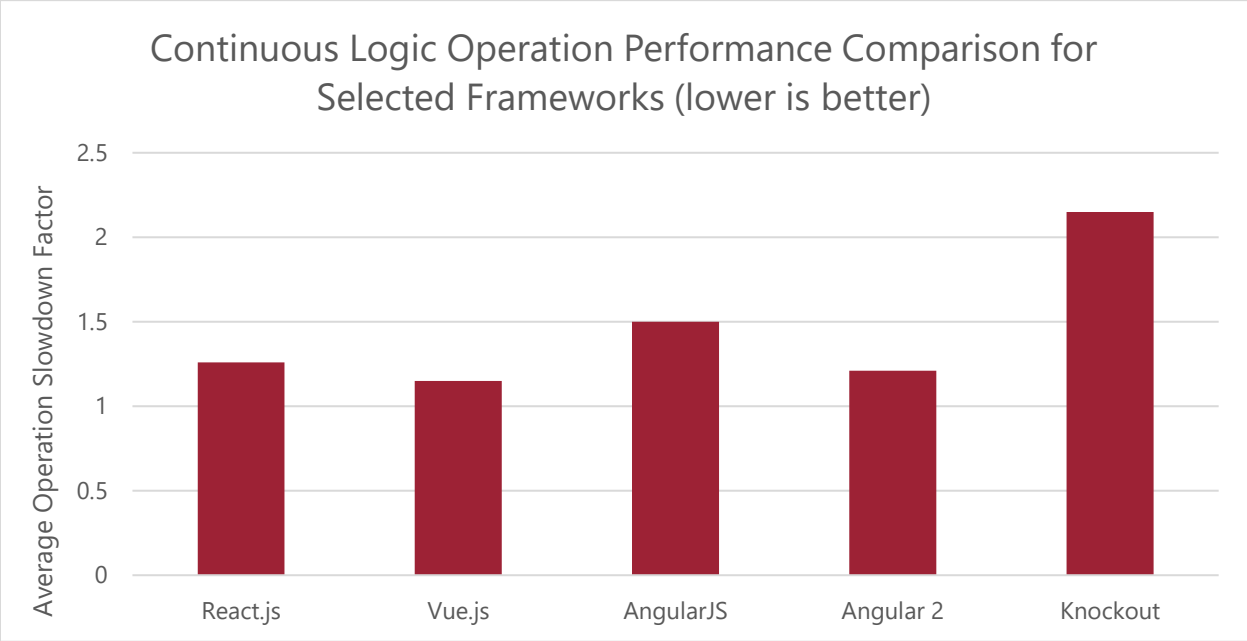
- **Productivity**
  1. AngularJS and Angular v2
  2. Vue.js
  3. React.js
  4. Knockout
- **Ecosystem**
  1. React.js
  2. AngularJS and Angular v2
  3. Vue.js
  4. Knockout
- **Compatibility**
  1. Knockout
  2. React.js
  3. AngularJS and Angular v2
  4. Vue.js

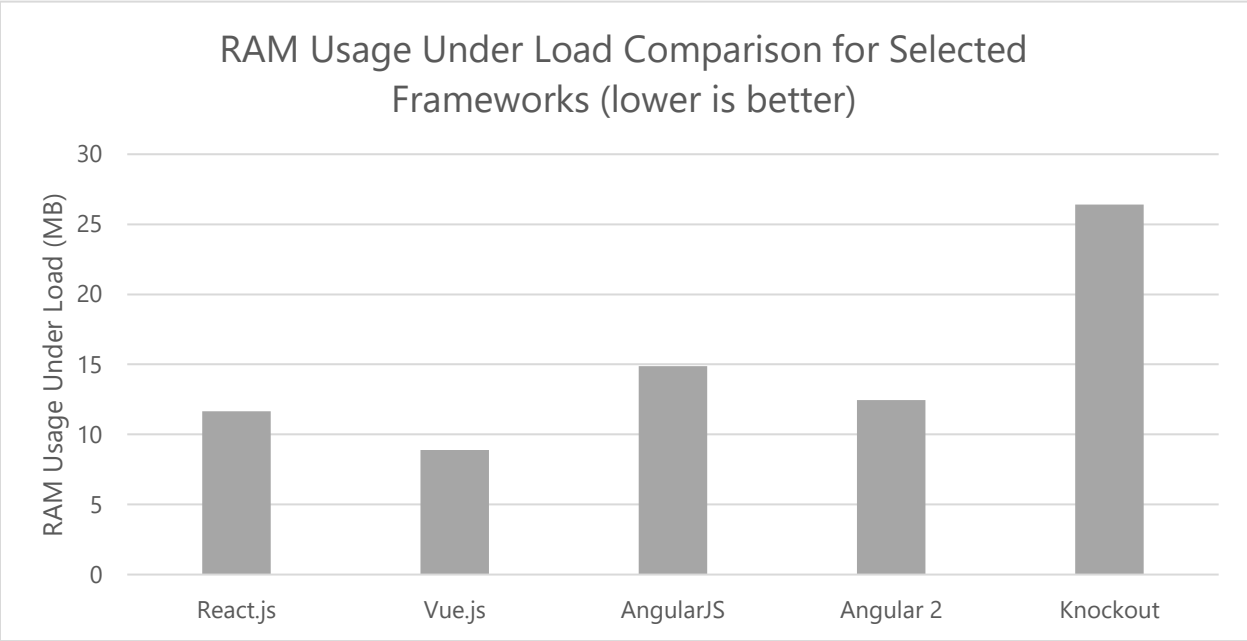
3. Download and set up your chosen framework!
4. Run tests and collect metrics to prove that your top priority item is meeting your project goals using the new framework.

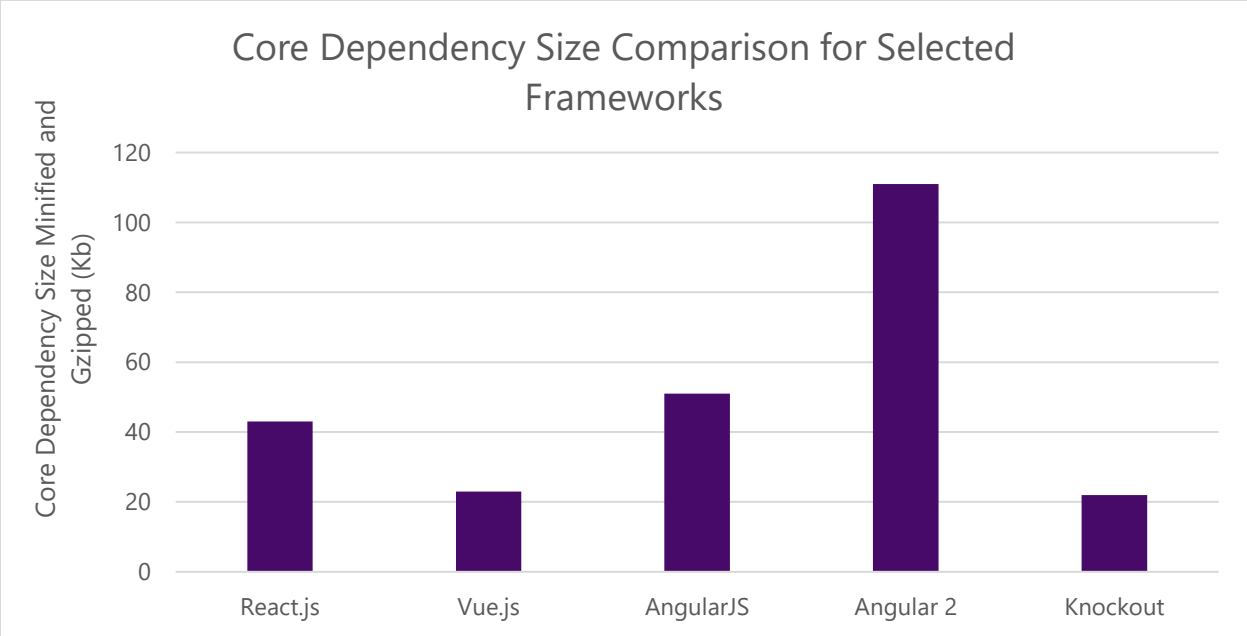
That's it! The framework rankings in step 2 are based on a collection of factors I talked about in the previous section of the paper. Here's the methodology in determining the ultimate decision.

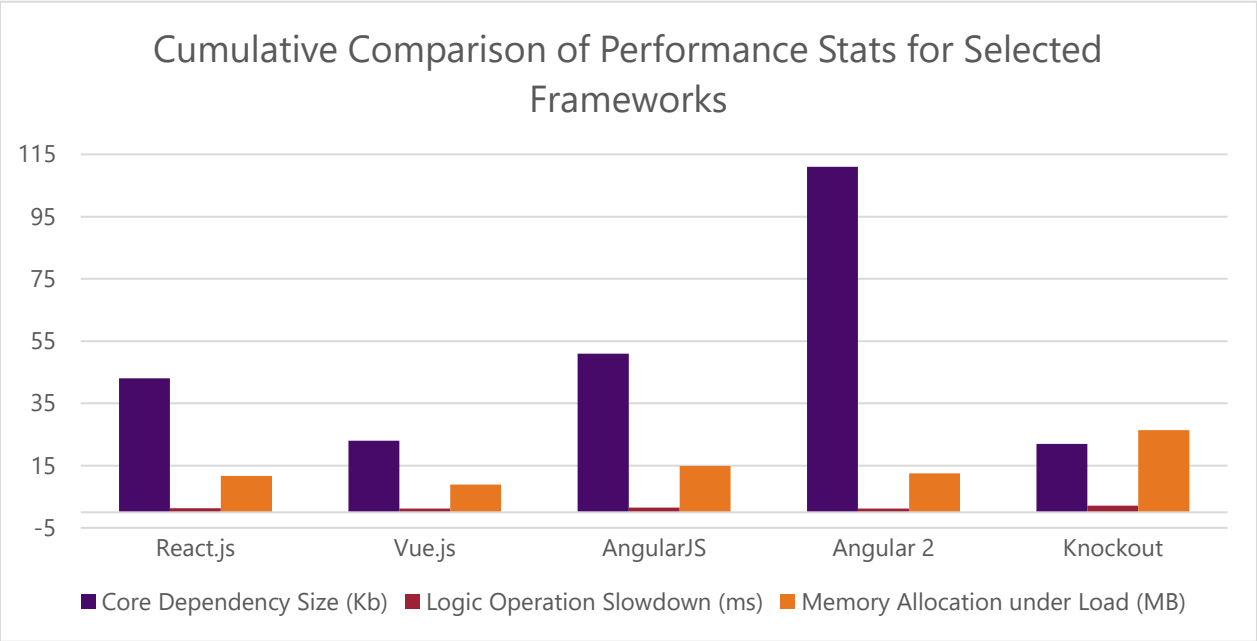
Justification

Speed and Performance




RAM Usage Under Load Comparison for Selected Frameworks (lower is better)
RAM Usage Under Load (MB)
30
25
20
15
10
5
0
React.js
Vue.js
AngularJS
Angular 2
Knockout


Core Dependency Size Comparison for Selected Frameworks
Core Dependency Size Minified and Gzipped (Kb)
120
100
80
60
40
20
0
React.js
Vue.js
AngularJS
Angular 2
Knockout



Before I get into discussing the graphs above, let's revisit the ranking of our five frameworks in terms of speed:

1. Vue.js
2. React.js
3. AngularJS and Angular v2
4. Knockout

Ranking was determined based on assessment of three factors: logic operation speed, RAM consumption under load, and core dependency size. At the end of the day, Vue.js comes out on top due to its unique combination of a virtual DOM and hot updating. React comes in a close second, followed by the Angular duo and Knockout in last. Angular v2, specifically, competes very closely with React and Vue. Its core dependency size is the highest of all the frameworks, but this only plays into network performance. The reason the Angular pair is placed in third is because of the slightly weaker

performance of Angular v1. It doesn't take advantage of some of the new performance-enhancing features of JavaScript and browser APIs, and its general structure causes it to be a little bit slower in DOM operations.

In general, **all frameworks perform well enough for production use.** I put performance in its own category mainly because many companies still use it as a crucial metric when justifying software choices, especially when the framework is used in a "mission critical" application and/or an application that will be under heavy load. These conditions justify using a framework just to save a few extra milliseconds in computation time. If your top priority for framework selection is performance, before you select Vue.js based on this factor alone, I encourage you to think deeply about *why* performance is so critical to your application, and whether you can afford to sacrifice a few milliseconds for other benefits.

Let’s look at the three factors that go into performance.

Continuous Logic Operation Performance

This data comes from an automated JavaScript framework benchmark suite developed and maintained by Stefan Krause. The benchmark runs tests for a slew of frameworks, including the five I've covered. The tests involve loading and displaying thousands of rows of data and then dynamically removing, formatting and updating certain rows once displayed in the DOM.

For each test on each framework, a slowdown value is reported. The slowdown is represented in milliseconds and is calculated by dividing the average duration of the test by the fastest iteration of the test for that framework. This is a relevant metric because it evidences a factor related to real-world performance. It shows how much a

framework can be slowed down *by its own logic*. For SPEC purposes, I report an average of slowdowns for all the tests.

Because of the calculation, you can think of 1 as a perfect score (no slowdown) and anything greater than that is a slowdown factor. E.g., a slowdown of 2.6 would mean that framework has the potential to perform 2.6 times slower than expected, simply due to its internal workings.

The results speak for themselves, but the fact is, none of the frameworks exhibit any major slowdown. Knockout is the only one that breaches the 2x slowdown mark, but even that's insignificant when the fastest operation is under 1 millisecond.

RAM Usage Under Load

This statistic measures the amount of client RAM each framework uses immediately after adding 1000 rows of data to the DOM. This data is also taken from Stefan Krause's benchmarks, and using as little RAM on the client computer as possible is ideal. Vue.js is the undisputed winner, using just under 9 MB under load. Knockout comes in last again, using over 25 MB for its processes. RAM Usage becomes a major concern if you're building your application with accessibility and responsiveness in mind. "The next billion users" of the internet are likely going to be accessing websites via less powerful smartphones, where memory consumption is a legitimate concern, so err on the side of caution and use as few resources as possible in your apps.

Core Dependency Size

These numbers represent the transported file size of each framework's core dependencies. This is representative of files you'll deliver to the end-user when they use your application.

As with the other metrics, this factor's importance has qualifications since the largest framework (Angular v2) is still only 111 Kb when transported. For users in urban areas, downloading this amount of data is typically a menial task. If your users don't have access to high-speed internet, or are slowed down by company network stacks, Core Dependency Size becomes more important. Because of its focused functionality, at 22 Kb, Knockout boasts the smallest core dependencies. At five times that size, Angular v2 is by far the largest framework. Even a 10 Kb difference in file size can make a 100 ms difference in loading time for users on a 2G connection. If you're thinking about "the next billion users," this is a vital consideration.

Productivity and Workflow

First, a quick reminder of the rankings:

1. AngularJS and Angular v2
2. Vue.js
3. React.js
4. Knockout

To determine a ranking for this section, I took a broad overview of the **out-of-the-box** workflow and project management solutions shipped with each framework. I also considered the official plugins available and the type of "development culture" each framework encourages.

With its first-class support for TypeScript and a built-in CLI that facilitates data model and app route auto-generation, Angular v2 emerges as the clear winner. Angular v2 also provides a built-in compiler and module loading system that encourages a specific format of compartmentalization and modularity, so your team spends less time making

productivity decisions and more time coding. Angular v2 brings along its older sibling, who shares design concepts like modularity handled by the framework itself. Both Angular versions have built-in compilation options that make deployment much easier. Although third-party task runners like Gulp and Grunt can elevate other frameworks to this level of automation, the functionality isn't built in like it is in Angular.

Vue.js and React.js essentially tie for second place because of the availability of workflow add-ons for both frameworks. Most of these add-ons, which increase functionality to include routing, CLIs and more, are optional, and are not considered out-of-the-box solutions. Vue.js and React may be better choices for teams who are migrating and bringing along their tooling, or for the developer who wants flexibility and options without compulsory workflow solutions.

Knockout doesn't offer many official plugins that deal with workflow. Despite this, its versatility and ability to insert into another framework’s project make it a valid option if you only want to extend the function of an existing app that already has a solid workflow.

Ecosystem

The ecosystem ranking from earlier:

1. React.js
2. AngularJS and Angular v2
3. Vue.js
4. Knockout

Ecosystem is similar to Productivity and Workflow, but it focuses more on the available extensibility of each framework and variety of plugins available.

Facebook has cultivated a rich ecosystem, and that puts React.js at the top of the list. Whether you're looking to build a full-scale data processing web app or a fun photo-sharing iPhone app, React has you covered. React offers plugins that implement routing, project management, deployment, native integration, and extensive data interfacing through novel data access paradigms like GraphQL (also developed by Facebook). React has also amassed community fervor that reaches far beyond that of any other framework. If you have trouble getting started with React, it's likely that you can find an answer to your specific problem with one Google search.

Angular offers considerable support in this area, and the Angular community is large (in fact, working with Wijmo, we've seen the biggest response from Angular users), but it doesn’t eclipse React’s community of startups and freelance web developers.

Vue.js falls in line with Angular, but its community is much smaller. It officially offers many significant plugins, like a CLI and routing, but the community hasn't shown much enthusiasm for developing plugins. This will likely change with time, but if you're looking for great community support right now, turn to React and Angular.

I hate to keep putting Knockout last because it’s a great tool! The problem is, it's more of a library than a framework, and it stands in its own category throughout this e-book. Knockout has strong community know-how and support, but since Knockout is sometimes considered an add-on itself, not many plugins interface with it. If you need routing or other advanced web app features, the most common solution would be to use something like ASP.NET MVC and simply integrate Knockout into your views. If

you're looking for a standalone framework with rich plugin availability, Knockout isn't your best option.

Compatibility and Browser Support

My rankings for framework browser support:

1. Knockout
2. React.js
3. AngularJS and Angular v2
4. Vue.js

And at last, Knockout comes out on top! When it comes to browser support, Knockout does it all. Supporting all the way back to Internet Explorer 6, Knockout covers over 99% of the global browser market share. If you have a mission critical Pure JS app and you want to give it some new tricks without sacrificing accessibility, integrating Knockout will ensure that you keep all your browser support.

React and Angular are close on browser support. They do support most major browsers, but only support IE back to version 9. This still covers most of the current browser market share, but may leave out vital businesses that still rely on *very* old IE support.

Vue also supports browsers back to IE 9, but its small community means fewer polyfills and shims are available for Vue-specific features that may be backported to older browsers.

If browser support is an absolute necessity for your project (perhaps you're building a web app for a business reliant on IE 8 or older), I suggest taking a Pure JS app and

slightly tweaking it with Knockout. If this isn't the case, you can still cover over 95% of browser market share with the other four frameworks.

Migration Concerns

If you're getting ready to use a JavaScript framework for the first time, this all might seem a little foreign to you. Don't be deterred: you have an advantage over someone who's looking to migrate from one framework to a different one. You’re starting fresh, and that means a lot in the framework world.

Even if you end up going with a framework that offers full-featured workflow solutions like Angular v2, you're going to need to spend time setting up tooling and *culture* to make the framework fit your needs. The time investment it takes to set up build tooling alone is enough to start "biasing" your team toward a particular framework. It can take weeks to get a proper build toolchain configured and automated, and that doesn't account for ongoing updates and maintenance applied over time. Therefore, if you've been using another framework to build your application and want to switch over, expect to spend at least a few weeks adjusting your tooling to work with the new framework.

In addition, be prepared to handle the friction that comes with changing a deeply ingrained development culture. Each framework establishes its own philosophy that filters down to developers’ mindset. For example, Angular v2 and its most popular components are built by some of the world's largest enterprises like Google and Microsoft, so it's no coincidence that Angular v2 is most often used by other large enterprise-level companies. The large-scale, efficiency-driven mindset that drove Angular v2's development "trickles down" to users and affects their own development

culture. The CLI has this effect by enforcing a specific file structure when used to create projects.

Neither of these transitory obstacles are much of a factor when moving to a framework from Pure JS because the frameworks themselves are built from Pure JS. You'll still need to add some new software to your computer and thought patterns to your mind, but the difference is you won't need to override anything.

Ultimately, if you're migrating to a new framework from any other platform you can expect to spend at least a month finding your stride. The total time depends on where you're coming from and where you're going, and the transition between certain frameworks may be easier than between others. If you're migrating a project from Pure JS to any framework, you'll experience the quickest overall migration time and the fewest number of "gotchas." Switching to Vue.js from any of the other four frameworks should also be relatively simple, since its core principles draw on philosophies established by the other frameworks that came before it. React.js, AngularJS and Knockout also work well as migration endpoints because they're tooling-agnostic; you can likely just bring over your existing tooling and easily hook it up to the new framework. Angular v2 is really the only option that will introduce more overhead to the migration process, simply because of the tooling and languages (TypeScript) that it encourages developers to use. The switch is worth it if you're working in a large team or want to further automate your development process. Plus, despite the differences, many pieces of programming know-how will always be transferable. Keeping in mind the philosophy of separation of concerns while developing is one of these transferable bits of knowledge. Another good example is knowing how to use a package manager to organize dependencies.

If you're worried about syntax, refer back to the "Migration from PureJS" sections earlier in the e-book. Each of these sections covers syntactical differences between the frameworks, and they provide a solid baseline for estimating how much you'll have to change in your code.

One more thing: consider the power and efficiency of using third-party libraries with multiple-framework support. As you've seen throughout this e-book, Wijmo provides an extensive library of UI controls for the web, and it ships with first-class support for all five of the frameworks we've talked about. You can build a web application using AngularJS and a UI driven by Wijmo. If your company grows and decides it would be better to switch to Angular v2, you only need to switch out your Wijmo reference and make minor syntactical tweaks to the UI. Rather than needing to author your own UI components twice, you *don't have to author them at all*. That's the power of interoperable tools like Wijmo. They increase your productivity at the start and throughout the life of your team's evolving needs.

The Big Picture

All of these frameworks are *incredibly* versatile. If one simply speaks to you and your team, your best bet may be to pick what feels natural. Even the frameworks with relatively low community support and ecosystem size have enough plugins to be customized to fit any app's needs. That being said: there's still only one *right* choice for your team. Logically, only one framework will work *best* for your team, and that depends on your project focus and your team's personality.

In the end, don't get bogged down in the decision-making process. The SPEC methodology is designed to make a lofty, unwieldy decision-making process quick and

easy. Overthinking it may lead you to make the *wrong* choice. Review the SPEC methodology and its suggestions, and then examine how you and your team feel about each framework.

In the end, this decision includes a vital "X factor" that isn't expressly included in the SPEC method. (SPEX, perhaps?) The personal preference of you and your team always enters into the equation. Even if all of the quantifiable signs (SPECs) are pointing to React.js, for example, if your team just doesn't get into it, you'll have to pass it by. If your team loves SPEC's recommendation, you'll have strong justification to devote time to get started with the framework. Even if it turns out that SPEC doesn't suggest the right framework for you, at the very least it will encourage you to reexamine your team's most important needs in the context of your new project. SPEC will also provide you with a versatile foundation for justifying a previous framework choice. Even if you've already started development with one framework, the SPEC criteria are great discussion points for hashing out your decision.

Final Thoughts: You can build an interactive web app without a framework. But why would you?

Even though we've focused primarily on comparing JavaScript frameworks, I hope you come away one singular, inherent message: **frameworks are powerful**. If you're not using a JavaScript framework in your web app now, I recommend that you integrate one as soon as you can. This paper examined just five of the most popular frameworks available, but you can explore plenty of others. With all the options available, there's a JavaScript framework out there that will fit the needs of your application, increase

performance, and improve your development workflow. Sure, it's possible to build an interactive web app without a framework, but why would you? It's also possible to travel from New York to Los Angeles by foot, but that’s not a time-efficient choice when we have non-perambulatory transportation. Similarly, the traditional HTML + CSS + JavaScript project design is not as time-efficient as using frameworks.

Nearly every JavaScript framework is surrounded by an entire ecosystem of add-ons and plugins that can help you meld a framework to fit your exact needs and ultimately make development even easier. For instance, Wijmo illustrates the power and versatility of using plugins in conjunction with frameworks, from easy and ready integration to feature-rich components that can be loaded into your project with just a few lines of code. Accelerating UI development gives an advantage to any workflow system, so you can see how building your app with a framework opens the door to countless opportunities for improvement beyond the framework itself. (Wijmo also works with pure JS, but the integration is inherently more complex.)

If you're already using a JavaScript framework, I hope you have some new information to think about. Maybe you were thinking about switching frameworks and now you have some better direction concerning which framework to move to, or maybe you were already switching and I validated your decision. In any case, I hope the SPEC method provides a solid process you make decisions in the future.

One last assignment: set up Hello, World, in at least one of the frameworks. It’s as fun here as it was the first time you tried it, and demonstrates how easy it is to set up. Let me know how it goes.

The Samples

You can access all the Wijmo Gauge control samples on the official Wijmo website: http://demos.wijmo.com/5/SampleExplorer/SampleExplorer/Sample/GaugeIntro.

Let Your Voice Be Heard

Take a minute to voice your opinion! Share whether you've used any frameworks, if I've missed some you think I should've included, and whether you have decided to try a new framework based on this e-book.

[[Poll links]]

APPENDIX A: The Rich History of JavaScript

JavaScript wasn't always the grand language used to build massive framework systems that it is today. For a long time following its inception, JavaScript was mostly used for gimmicky website effects, like firework animations. It's come a long way from its archaic beginnings. The best way to see this dramatic, if gradual, improvement is to look at the ECMAScript standardization of JavaScript.

If you know what JavaScript is, then you also know what ECMAScript is. These two titles both refer to the same programming language. The colloquial name "JavaScript" is a strategic misnomer. Even though JavaScript syntax bears some resemblance to Java, the languages vary widely on their core principles. Brendan Eich, a former Netscape employee credited with creating JavaScript in 1995, coined the name JavaScript due to Java's immense popularity at the time. Without this marketing ploy, JavaScript may not have been adopted by the community at large. On the other hand, that original name has left web developers with a confusing conundrum in modern-day usage of the language. Even though "JavaScript" is now universally recognized, its etymology often remains a mystery.

Shortly after JavaScript's creation, the European Computer Manufacturers Association (ECMA), which puts forth standards for many modern technological protocols and programming languages, was tasked with standardizing the ambiguously-named language. From this effort was borne the ECMAScript (ES) specification. Although related, ECMAScript and JavaScript are not synonymous. JavaScript is an *implementation* of the ECMAScript specification. (Other implementations of the ES specification exist, though they’re used much less widely than JavaScript.) Nonetheless, because of its widespread usage, JavaScript is the "poster child" of ECMAScript. Generally, and

especially in this e-book, any time you see a specific ECMAScript standard revision mentioned, you can think of that as "how JavaScript implements this ECMAScript standard revision."

The recently-finalized ES2015 standard is still undergoing adoption by most major browsers and JavaScript engines. As expected, adoption is occurring feature-by-feature rather than holistically. While this can muddle compatibility issues, the upside is that it allows us to see that JavaScript is now far separated from its original identity. No longer a simple gimmick of a programming language, JavaScript has adapted to address a major issue that has fueled debate over its usefulness: eloquence. With new features like block level scoping and generator functions shipping in ES2015, JavaScript can now hold its own among the traditional "refined" programming languages like Java and C#. Impressively, JavaScript is coming into this role while retaining its usefulness as a procedural, customizable language as well. All this evolution culminates to offer a platform that is powerful and useful, not only for the client-side web, but also for server-side and native apps.

Understanding this history is important to understanding the current boom in JavaScript usage. The changes included in the recent ES2015 standardization and the advent of other new technologies—like local storage through the browser and web sockets—have been especially important to enabling the "Dawn of Frameworks" that necessitated this e-book.

Where is JavaScript Now?

Although many have tried, it's difficult to generate reliable statistics on exactly what fraction of internet users have run some form of JavaScript. Fortunately, all modern web

browsers support and enable JavaScript out of the box. Even browsers that ship with smartphones have JavaScript support that mirrors or outdoes their desktop counterparts. So pretty much *anyone* who has a semi-modern device has access to JavaScript content on the web. Even if you look back several years, you'll find that the majority of web browsers shipped with JavaScript support enabled by default. Even if this support is not complete (due to compatibility issues with new syntax, etc.), most features can be polyfilled or shimmed to make them work in older browsers. Overall, estimates put JavaScript accessibility near 100% of internet users. (This doesn't account for the relatively small group of people who voluntarily turn off JavaScript.) I'll be the first to admit that we, as developers, have a responsibility to make content accessible to *everyone* who uses the internet. Luckily, most of the major JavaScript frameworks provide a check and fallback mechanism for users who can't access JavaScript-driven apps. As a developer, you're still responsible for providing static fallback content, but even that can be generated using JavaScript on the server.

The major takeaway here is that putting your money on JavaScript support when deciding to use a framework is a pretty safe bet to make. You can count on reaching virtually all internet users, and you can still use JavaScript to render a static fallback if you'd like.

The idea of using JavaScript to generate and serve static content brings up another interesting topic: server-side code. With the advent of Node.js, server-side JavaScript has become mainstream and commonplace. Node provides a powerful platform for writing reliable and fast server-side applications using JavaScript, and its scalability has made it a viable option for large and frequently used server-side APIs. While it may not be obvious, this is yet another reason to use JavaScript frameworks. The availability of

Node.js brings with it the possibility of a unified codebase. You can potentially integrate your entire development stack with the JavaScript language, building a backend with Node.js and a front-end with a JavaScript framework, to improve workflow and team communication.

Node.js has also introduced the web development world to Node Package Manager (NPM), which further facilitates JavaScript development. With thousands of handy pluggable modules at your fingertips, the ability to share package configurations for a project, and an easy install process, NPM has revolutionized the development world. In the scope of frameworks, NPM makes starting up a new project and adding any popular framework a breeze. In fact, the "Installation" pages of almost all major JavaScript frameworks begin immediately with an `npm install` instruction.

The combination of widespread browser support, server-side platforms, and package management solutions makes *now* the best time to start using a JavaScript framework. As a matter of fact, some recent surveys indicate that over 70% of JavaScript developers have already used a front-end framework, and over 95% have at least *heard of* one of the frameworks discussed in this e-book. And if everyone else is using them, there must be something to this whole framework thing, right?

The widespread usage of JavaScript frameworks not only indicates that they work well; it also means that a vast array of resources and plugins have been vetted and improved by the large JavaScript community. It always feels better to start something new when you know others have done it before you. So rest assured: many have traveled and survived the path to framework enlightenment on which you're about to embark, and they've left some helpful tools and tips along the way.

Thank You

And with that, my discussion of JavaScript front-end frameworks comes to a close. I have a lot of people to thank for offering up resources to write this e-book. But most of all, I want to **thank you**, the enduring reader. I hope you enjoyed your time here and learned something in the process, and I hope you'll come back for more in the future!

I also want to send out a huge thank you to everyone at GrapeCity for giving me the time and resources needed to tackle this topic. (Not to mention for taking the time to read and edit all of my long-winded sentences!)

Last but certainly not least, I want to thank:

- Sacha Greif for conducting perhaps the most fiery programming survey to date
- Stefan Krause for building an awesome JavaScript framework benchmark suite
- The Angular team at Google
- The React team at Facebook
- The Vue team
- The Knockout team
- The TypeScript team at Microsoft
- Kyle Simpson for showing me the JavaScript light

Get in Touch

If you want to talk about the e-book, JavaScript frameworks, or anything else at all, really, get in touch using the information below:

Christian Gaetano

christian.gaetano@grapecity.com

christiangaetano.com

[@cgatno](https://twitter.com/cgatno)