How to Choose the Best JavaScript Framework for Your Team

A comparison of the top JavaScript frameworks available

(Excerpt from an e-book by Christian Gaetano)

Preface - About this E-book and why it exists

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.

Picking the right framework for your project requires a deep knowledge of *all* the 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. 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: Top Companies Love Frameworks

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 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.

Part 1. A Quick Look at JavaScript

1.1. About JavaScript as a Language

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 flexibility and uniqueness come from necessity, and learning to work *with* its quirks rather than trying to work *around* them is usually your best course of action.

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.

1.2. 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*.

JavaScript frameworks are all about separating out your concerns.

1.3. 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.

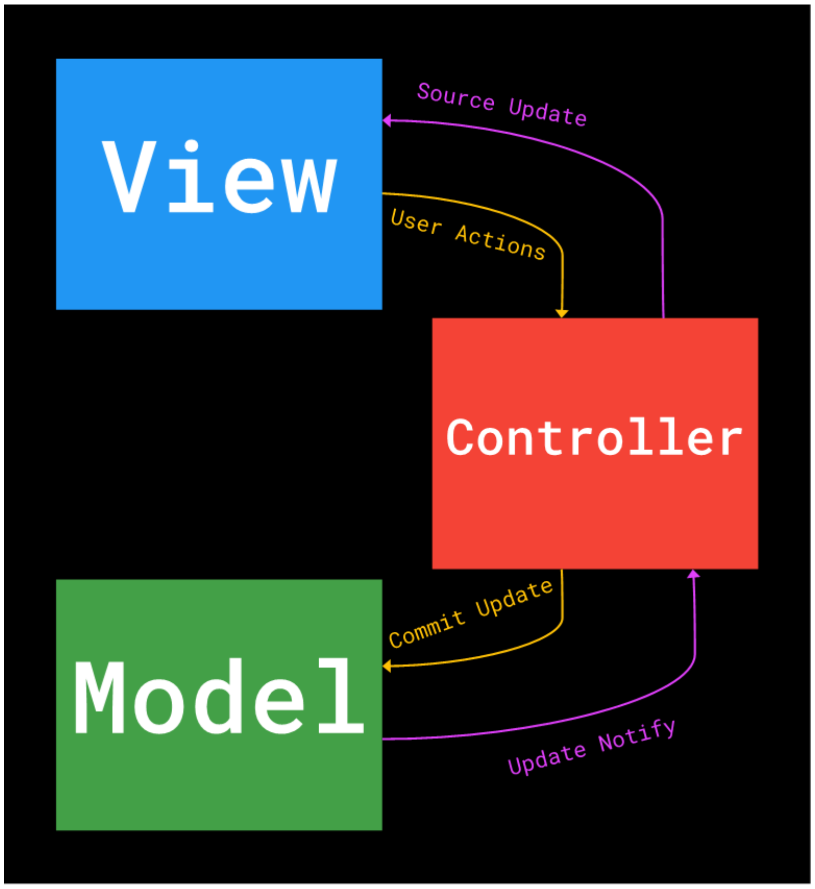

Fig 1. A simplistic depiction of the components of a typical MVC software architecture along with the actions associated with each.

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.