SMASHING MAGAZINE

REVIEW OF JS FRAMEWORKS

Journey Through The JavaScript MVC Jungle

By Addy Osmani

Published on July 27th, 2012 in Essentials, JavaScript, Tools with 95 Comments

When writing a Web application from scratch, it's easy to feel like we can get by simply by relying on a DOM[1] manipulation library (like jQuery[3][8][2]) and a handful of utility plugins. The problem with this is that it doesn't take long to get lost in a nested pile of jQuery callbacks and DOM elements without any real structure in place for our applications.

In short, we're stuck with spaghetti code[3]. Fortunately there are modern JavaScript frameworks that can assist with bringing structure and organization to our projects, improving how easily maintainable they are in the long-run.

What Is MVC, Or Rather MV*?

These modern frameworks provide developers an easy path to organizing their code using variations of a pattern known as MVC[4] (Model-View-Controller). MVC separates the concerns in an application down into three parts:

- **Models** represent the domain-specific knowledge and data in an application. Think of this as being a 'type' of data you can model — like a User, Photo or Note. Models should notify anyone observing them about their current state

(e.g Views).

- **Views** are typically considered the User-interface in an application (e.g your markup and templates), but don't have to be. They should know about the existence of Models in order to observe them, but don't directly communicate with them.
- **Controllers** handle the input (e.g clicks, user actions) in an application and Views can be considered as handling the output. When a Controller updates the state of a model (such as editing the caption on a Photo), it doesn't directly tell the View. This is what the observing nature of the View and Model relationship is for.

JavaScript 'MVC' frameworks that can help us structure our code don't always strictly follow the above pattern. Some frameworks will include the responsibility of the Controller in the View (e.g Backbone.js[105]) whilst others add their own opinionated components into the mix as they feel this is more effective.

For this reason we refer to such frameworks as following the MV* pattern, that is, you're likely to have a View and a Model, but more likely to have something else also included.

Note: There also exist variations of MVC known as MVP (Model-View-Presenter) and MVVM (Model-View ViewModel). If you're new to this and feel it's a lot to take in, don't worry. It can take a little while to get your head around patterns, but I've written more about the above patterns in my online book Learning JavaScript Design Patterns[6] in case you need further help.

When Do You Need A JavaScript MV* Framework?

When building a single-page application using JavaScript, whether it involves a complex user interface or is simply trying to reduce the number of HTTP requests required for new Views, you will likely find yourself inventing many of the pieces that make up an MV* framework like Backbone or Ember.

At the outset, it isn't terribly difficult to write an application framework that offers *some* opinionated way to avoid spaghetti code, however to say that it is equally as trivial to write something of the standard of Backbone would be a grossly incorrect assumption.

There's a lot more that goes into structuring an application than tying together a DOM manipulation library, templating and routing. Mature MV* frameworks typically not only include many of the pieces you would find yourself writing, but also include

solutions to problems you'll find yourself running into later on down the road. This is a time-saver that you shouldn't underestimate the value of.

So, where will you likely need an MV* framework and where won't you?

If you're writing an application that will likely only be communicating with an API or back-end data service, where much of the heavy lifting for viewing or manipulating that data will be occurring in the browser, you may find a JavaScript MV* framework useful.

Good examples of applications that fall into this category are GMail[7] and Google Docs[8]. These applications typically download a single payload containing all the scripts, stylesheets and markup users need for common tasks and then perform a lot of additional behavior in the background. It's trivial to switch between reading an email or document to writing one and you don't need to ask the application to render the whole page again at all.

If, however, you're building an application that still relies on the server for most of the heavy-lifting of Views/pages and you're just using a little JavaScript or jQuery to make things a little more interactive, an MV framework may be overkill. There certainly are complex Web applications where the partial rendering of views can* be coupled with a single-page application effectively, but for everything else, you may find yourself better sticking to a simpler setup.

The Challenge Of Choice: Too Many Options?

The JavaScript community has been going through something of a renaissance over the last few years, with developers building even larger and more complex applications with it as time goes by. The language still greatly differs from those more classic Software engineers are used to using (C++, Java) as well as languages used by Web developers (PHP, Python, .Net etc). This means that in many cases we are borrowing concepts of how to structure applications from what we have seen done in the past in these other languages.

In my talk "Digesting JavaScript MVC: Pattern Abuse or Evolution[9]", I brought up the point that there's currently too much choice when it comes to what to use for structuring your JavaScript application. Part of this problem is fueled by how different JavaScript developers interpret how a scalable JavaScript application should be organized — MVC? MVP? MVVM? Something else? This leads to more frameworks being created with a different take on MV* each week and ultimately more noise because we're still trying to establish the "right way" to do things, if that exists at all. Many developers believe it doesn't.

We refer to the current state of new frameworks frequently popping up as 'Yet Another Framework Syndrome' (or YAFS). Whilst innovation is of course something we should welcome, YAFS can lead to a great deal of confusion and frustration when developers just want to start writing an app but don't want to manually evaluate 30 different options in order to select something maintainable. In many cases, the differences between some of these frameworks can be very subtle if not difficult to distinguish.

TodoMVC: A Common Application For Learning And Comparison

There's been a huge boom in the number of such MV* frameworks being released over the past few years.

Backbone.js[10][5], Ember.js[3][12][0][11], AngularJS, Spine[12], CanJS[3][3][13] … The list of new and stable solutions continues to grow each week and developers can quickly find themselves lost in a sea of options. From minds who have had to work on complex applications that inspired these solutions (such as Yehuda Katz[14] and Jeremy Ashkenas[15]), there are many strong contenders for what developers should consider using. The question is, what to use and how do you choose?

We understood this frustration and wanted to help developers simplify their selection process as much as possible. To help solve this problem, we created TodoMVC[16] — a project which offers the same Todo application implemented in most of the popular JavaScript MV* frameworks of today — think of it as speed dating for frameworks.

Solutions look and feel the same, have a common feature set, and make it easy for us to compare the syntax and structure of different frameworks, so we can select the one we feel the most comfortable with or at least, narrow down our choices.

This week we're releasing a **brand new** version of TodoMVC[17], which you can find more details about lower down in the apps section.

In the near future we want to take this work even further, providing guides on how frameworks differ and recommendations for which options to consider for particular types of applications you may wish to build.

Our Suggested Criteria For Selecting A Framework

Selecting a framework is of course about more than simply comparing the Todo app implementations. This is why, once we've filtered down our selection of potential frameworks to just a few, it's recommend to spend some time doing a little due diligence. The framework we opt for may need to support building non-trivial features and could end up being used to maintain the app for years to come.

- **What is the framework really capable of?**
  Spend time reviewing both the source code of the framework and official list of features to see how well they fit with your requirements. There will be projects that may require modifying or extending the underlying source and thus make sure that if this might be the case, you've performed due diligence on the code.
- **Has the framework been proved in production?**
  i.e Have developers actually built and deployed large applications with it that are publicly accessible? Backbone has a strong portfolio of these (SoundCloud, LinkedIn) but not all frameworks do. Ember is used in number of large apps, including the user tools in Square. JavaScriptMVC has been used to power applications at IBM amongst other places. It's not only important to know that a framework works in production, but also being able to look at real world code and be inspired by what can be built with it.
- **Is the framework mature?**
  We generally recommend developers don't simply "pick one and go with it". New projects often come with a lot of buzz surrounding their releases but remember to take care when selecting them for use on a production-level app. You don't want to risk the project being canned, going through major periods of refactoring or other breaking changes that tend to be more carefully planned out when a framework is mature. Mature projects also tend to have more detailed documentation available, either as a part of their official or community-driven docs.
- **Is the framework flexible or opinionated?**

Know what flavor you're after as there are plenty of frameworks available which provide one or the other. Opinionated frameworks lock (or suggest) you to do things in a specific way (theirs). By design they are limiting, but place less emphasis on the developer having to figure out how things should work on their own.

- **Have you really played with the framework?**
  Write a small application without using frameworks and then attempt to refactor your code with a framework to confirm whether it's easy to work with or not. As much as researching and reading up on code will influence your decision, it's equally as important to write actual code using the framework to make sure you're comfortable with the concepts it enforces.
- **Does the framework have a comprehensive set of documentation?**
  Although demo applications can be useful for reference, you'll almost always find yourself consulting the official framework docs to find out what its API supports, how common tasks or components can be created with it and what the gotchas worth noting are. Any framework worth it's salt should have a detailed set of documentation which will help guide developers using it. Without this, you can find yourself heavily relying on IRC channels, groups and self-discovery, which can be fine, but are often overly time-consuming when compared to a great set of docs provided upfront.
- **What is the total size of the framework, factoring in minification, gzipping and any modular building that it supports?**
  What dependencies does the framework have? Frameworks tend to only list the total filesize of the base library itself, but don't list the sizes of the librarys dependencies. This can mean the difference between opting for a library that initially looks quite small, but could be relatively large if it say, depends on jQuery and other libraries.
- **Have you reviewed the community around the framework?**
  Is there an active community of project contributors and users who would be able to assist if you run into issues? Have enough developers been using the framework that there are existing reference applications, tutorials and maybe even screencasts that you can use to learn more about it?

Dojo And Rise Of The JavaScript Frameworks

As many of us know, the Dojo toolkit[18] was one of the first efforts to provide developers a means to developing more complex applications and some might say it in-part inspired us to think more about the needs of non-trivial applications. I sat down to ask Dojos Dylan Schiemann[19], Kitson Kelly, and James Thomas what their thoughts were on the rise of JavaScript MV* frameworks.

Q: Didn't Dojo already solve all of this? Why hasn't it been the dominent solution for developers wishing to build more structured (and more non-trivial) applications?

Years ago, while the JavaScript landscape evolved from adding simple Ajax and chrome to a page, Dojo was evangelizing a "toolkit" approach to building complex Web applications.

Many of those features were way ahead of most developers needs. With the emergence of the browser as the dominant application platform, many of the innovations pioneered in The Dojo Toolkit now appear in newer toolkits. MVC was just another package that Dojo has provided for quite some time, along with modular code packages, OO in JS, UI widgets, cross-browser graphics, templating, internationalization, accessibility, data stores, testing frameworks, a build system and much, much more.

JavaScript libraries shouldn't end at "query", which is why Dojo, early on, focussed on completing the picture for enterprise grade application development. This is the same focus that is has today with MVC, it's just another "tool in the arsenal".

Why is Dojo not the dominant toolkit? Its goal was never to be the only choice. The goal was to provide an open collection of tools that could be used with anything else, within projects, and liberally copied into other work as well. Dojo was criticized for being slow and even after that was addressed, it was criticized for being slow. Trying to shake that perception is challenging. It is very hard to document a feature-rich toolkit. There are 175 sub-packages in Dojo 1.8 and over 1,400 modules.

That is not only a challenge from a documentation purpose, it also means that there isn't one thing that Dojo does. Which is good if you are building software, but very difficult when you are starting out trying to figure out where to start. These are all things we have been trying to work on for Dojo 1.8, in the form of tutorials and significantly improved documentation.

Q: Why should developers still consider Dojo and what ideas do you have lined up for the future of the project? I hear 1.8 will be another major milestone.

In Dojo 1.8, dojox/mvc takes another step towards full maturity. There has been a lot of investment in time, effort, testing and community awareness into the package. It focuses on providing an MVC model that leverages the rest of Dojo. Coupled with dojox/app, an application framework that is designed to make it easier to build rich applications across desktop and mobile, it makes a holistic framework for creating a client side application.

In the typical Dojo way, this is just one of many viable ways in which to build applications with Dojo.

In 1.8, not only does the MVC sub-module become more mature, it is built upon a robust framework. It doesn't just give you markup language to create your views, express your models or develop a controller. It is far more then just wiring up some controls to a data source. Because it is leveraging the rest of Dojo, you can draw in anything else you might need.

In Dojo 2.0 we will be looking to take modularity to a new level, so that it becomes even easier to take a bit of this and a bit of that and string it all together. We are also exploring the concepts of isomorphism, where it should be transparent to the end-user where your code is being executed, be it client side or server side and that ultimately it should be transparent to the developer.

The TodoMVC Collection

In our brand new release, Todo implementations now exist for the most popular frameworks with a large number of other commonly used frameworks being worked on in Labs. These implementations have gone through a lot of revision, often taking

on board best practice tips and suggestions from framework authors, contributors and users from within the community.

Following on from comments previously made by Backbone.js author Jeremey Ashkenas and Yehuda Katz, TodoMVC now also offers consistent implementations based on an official application specification as well as routing (or state management).

We don't pretend that more complex learning applications aren't possible (they certainly are), but the simplicity of a Todo app allows developers to review areas such as code structure, component syntax and flow, which we feel are enough to enable a comparison between frameworks and prompt further exploration with a particular solution or set of solutions.

Our applications include:

- Backbone.js
- Ember.js[3][12][11]
- AngularJS[5][5][21]
- Spine.js[22]
- KnockoutJS[23] (MVVM)
- Dojo[24]
- YUI[25]
- Batman.js[26]
- Closure[27]
- Agility.js[28]
- Knockback.js[29]

For those interested in AMD versions:

- Backbone.js + RequireJS[5][6][3][23][0] (using AMD)
- Ember.js[3][12][11] + RequireJS[5][6][3][23][0] (using AMD)

And our Labs include:

- CanJS[33][13]
- Maria.js[34]
- cujo.js[35]
- Meteor[36]
- SocketStream[37] + jQuery[38][2]
- Ext.js[39]
- Sammy.js[40]
- JavaScriptMVC[41]
- Google Web Toolkit[42]
- TroopJS[43]

- Stapes.js[44]
- soma.js[45]
- DUEL[46]
- Fidel[47]
- Olives[48]
- PlastronJS[49]
- Dijon[50]
- rAppid.js[51]
- Broke[52]
- o_O[53]
- Fun[54]
- AngularJS[5521] + RequireJS[563230] (using AMD)

Note: We've implemented a version of our Todo application using just JavaScript and another using primarily jQuery conventions. As you can see, whilst these applications are functionally equivalent to something you might write with an MVC framework, there's no separation of concerns and the code becomes harder to read and maintain as the codebase grows.

We feel honored that over the past year, some framework authors have involved us in discussions about how to improve their solutions, helping bring our experience with a multitude of solutions to the table. We've also slowly moved towards TodoMVC being almost a defacto app that new frameworks implement and this means it's become easier to make initial comparisons when you're reviewing choices.

Frameworks: When To Use What?

To help you get started with narrowing down frameworks to explore, we would like to offer the below high-level framework summaries which we hope will help steer you towards a few specific options to try out.

I want something flexible which offers a minimalist solution to separating concerns in my application. It should support a persistence layer and RESTful sync, models, views (with controllers), event-driven communication, templating and routing. It should be imperative, allowing one to update the View when a model changes. I'd like some decisions about the architecture left up to me. Ideally, many large companies have used the solution to build non-trivial applications. As I may be building something complex, I'd like there to be an active extension community around the framework that have already tried addressing larger problems (Marionette[57], Chaplin[58], Aura[59], Thorax[60]). Ideally, there are also scaffolding tools (grunt-bbb[61], brunch[62]) available for the solution. **Use Backbone.js.**

I want something that tries to tackle desktop-level application development for the web. It should be opinionated, modular, support a variation of MVC, avoid the need to wire everything in my application together manually, support persistence, computed properties and have auto-updating (live) templates. It should support proper state management rather than the manual routing solution many other frameworks advocate being used. It should also come with extensive docs and of course, templating. It should also have scaffolding tools available (ember.gem, ember for brunch). **Use Ember.js.**

I want something more lightweight which supports live-binding templates, routing, integration with major libraries (like jQuery and Dojo) and is optimized for performance. It should also support a way to implement models, views and controllers. It may not be used on as many large public applications just yet, but has potential. Ideally, the solution should be built by people who have previous experience creating many complex applications. **Use CanJS.**

I want something declarative that uses the View to derive behavior. It focuses on achieving this through custom HTML tags and components that specify your application intentions. It should support being easily testable, URL management (routing) and a separation of concerns through a variation of MVC. It takes a different approach to most frameworks, providing a HTML compiler for creating your own DSL in HTML. It may be inspired by upcoming Web platform features such as Web Components and also has its own scaffolding tools available (angular-seed). **Use AngularJS.**

I want something that offers me an excellent base for building large scale applications. It should support a mature widget infrastructure, modules which support lazy-loading and can be asynchronous, simple integration with CDNs, a wide array of widget modules (graphics, charting, grids, etc) and strong support for internationalization (i18n, l10n). It should have support for OOP, MVC and the building blocks to create more complex architectures. **Use Dojo.**

I want something which benefits from the YUI extension infrastructure. It should support models, views and routers and make it simple to write multi-view applications supporting routing, View transitions and more. Whilst larger, it is a complete solution that includes widgets/components as well as the tools needed to create an organized application architecture. It may have scaffolding tools (yuiproject), but these need to be updated. **Use YUI.**

I want something simple that values asynchronous interfaces and lack any dependencies. It should be opinionated but flexible on how to build applications. The framework should provide bare-bones essentials like model, view, controller, events, and routing, while still being tiny. It should be optimized for use with CoffeeScript and come with comprehensive documentation. **Use Spine.**

I want something that will make it easy to build complex dynamic UIs with a clean underlying data model and declarative bindings. It should automatically update my UI on model changes using two-way bindings and support dependency tracking of model data. I should be able to use it with whatever framework I prefer, or even an existing app. It should also come with templating built-in and be easily extensible. **Use KnockoutJS.**

I want something that will help me build simple Web applications and websites. I don't expect there to be a great deal of code involved and so code organisation won't be much of a concern. The solution should abstract away browser differences so I can focus on the fun stuff. It should let me easily bind events, interact with remote services, be extensible and have a huge plugin community. **Use jQuery.**

What Do Developers Think About The Most Popular Frameworks?

As part of our research into MV* frameworks for TodoMVC and this article, we decided to conduct a survey to bring together the experiences of those using these solutions. We asked developers what framework they find themselves using the most often and more importantly, why they would recommend them to others. We also asked what they felt was still missing in their project of choice.

We've grouped some of the most interesting responses below, by framework.

EMBER.JS

> **Pros:** *The combination of live templates and observable objects has changed the way I write JavaScript. It can be a bit much to wrap your head around at first, but you end up with a nice separation of responsibility. I found that once I have everything set up, adding fairly complex features only takes a couple lines of code. Without Ember, these same features would've been hellish to implement.*
>
> **Cons:** *Ember has yet to reach 1.0. Many things are still in flux, such as the router and Ember data. The new website is very helpful, but there's still not as much documentation for Ember as there is for other frameworks, specifically Backbone. Also, with so much magic in the framework, it can be a little scary. There's the fear that if something breaks you won't be able to figure out exactly why. Oh, and the error messages that ember gives you often suck.*

Pros:
The key factors:

a) Features that let me avoid a lot of boilerplate (bindings, computer properties, view layer with the cool handlebars).

b) the core team: I'm a Rails developer and know the work of Yehuda Katz. I trust the guy =)

Cons: *Documentation. It's really sad that Ember doesn't have good documentation, tutorials, screencast like Backbone, Angular or other frameworks. Right now, we browse the code looking for docs which isn't ideal.*

Pros: *Convention over configuration. Ember makes so many small decisions for you it's by far the easiest way to build a client-side application these days.*

Cons: *The learning curve. It is missing the mass of getting started guides that exist for other frameworks like Backbone, this is partly because of the small community, but I think more because of the state of flux the codebase is in pre-1.0.*

Pros: *Simplicity, bindings, tight integration with Handlebars, ease of enabling modularity in my own code.*

Cons: *I'd like to have a stable integration with ember-data, and integrated localStorage support synced with a REST API, but hey that's fantasy that one day will surely come true ;-)*

BACKBONE.JS

Pros: *Simplicity — only 4 core components (Collection, Model, View, Router). Huge community (ecosystem) and lots of solutions on StackOverflow. Higher order frameworks like Marionette or Vertebrae with lots of clever code inside. Somebody might like "low-levelness" — need to write lots of boilerplate code, but get customized application architecture.*

Cons: *I don't like how extend method works — it copies content of parent objects into new one. Prototypal inheritance FTW. Sometime I miss real world scenarios in docs examples. Also there is a lot of research needed to figure out how to build a bigger app after reading the TODO tutorial.*

I'm missing official AMD support in projects from DocumentCloud (BB, _). [Note: this shouldn't be an issue with the new RequireJS shim() method in RequireJS 2.0].

Pros: *After the initial brain-warp of understanding how Backbone rolls, it is incredibly useful. Useful as in, well supported, lightweight, and constantly updated in a valid scope. Ties in with natural friends Underscore, jQuery/Zepto, tools that most of my studio's projects would work with.*

Cons: *The amount of tutorials on how to do things with Backbone is inconsistent and at different periods of Backbones lifespan. I've asked other devs to have a look at Backbone, and they would be writing code for v0.3. Un-aware. Whilst not a problem Backbone can fix itself, it is certainly a major dislike associated with the framework.*

I suppose in theory, you could apply this to anything else, but, Backbone is a recurrent one in my eyes. Hell, I've even seen month old articles using ancient Backbone methods and patterns.

Whatever dislikes I would have on the framework strictly itself, has been rectified by the community through sensible hacks and approaches. For me, that is why Backbone is great, the community backing it up.

Pros: *Provides just enough abstraction without unreasonable opinions — enabling you to tailor it to the needs of the project.*

Cons: *I would re-write (or possibly remove) Backbone.sync. It has baked in assumptions of typical client-initiated HTTP communications, and doesn't adapt well to the push nature of WebSockets.*

Pros: *It's extremely easy to get into, offering a nice gateway to MV* based frameworks. It's relatively customizable and there are also tons of other people using it, making finding help or support easy.*

Cons: *The fact that there's no view bindings by default (although you can fix this). Re-rendering the whole view when a single property changes is wasteful.*

The RESTful API has a lot of positives, but the lack of bulk-saving (admittedly a problem with REST itself, but still) and the difficulty in getting different URI schemes to work on different types of operations sucks.

ANGULARJS

Pros:

a) 2-way data binding is incredibly powerful. You tend to think more about your model and the state that it is in instead of a series of events that need to happen. The model is the single source of truth.

b) Performance. AngularJS is a small download. It's templating uses DOM nodes instead of converting strings into DOM nodes and should perform better.

c) If you are targeting modern browsers and/or are a little careful, you can drop jQuery from your dependencies too.

Cons: *I'd like to be able to specify transitions for UI state changes that propgate from a model change. Specifically for elements that use ng-show or ng-hide I'd like to use a fade or slide in in an easy declarative way.*

Pros: *It's very intuitive, has excellent documentation. I love their data binding approach, HTML based views, nested scopes. I switched from Backbone/Thorax to Angular and never looked back. A new Chrome extension Batarang integrates with Chrome Developer's Tools and provides live access the Angular data structures.*

Cons: *I'd like to have a built-in support to such functions as drag'n'drop, however this can be added using external components available on GitHub. I'd also like to see more 3rd party components available for reuse. I think it's just a matter of time for the ecosystem around AngularJS to get more mature and then these will be available just like they are in communities like jQuery.*

Pros: *It minimizes drastically the boilerplate code, allows for nice code reuse through components, extends the HTML syntax so that many complex features end up being as simple as applying a directive (attribute) in the HTML, and is super-easily testable thanks to a full commitment to dependency injection.*
You can write a non-trivial app without jQuery or without directly manipulating the DOM. That's quite a feat.

Cons: *Its learning curve is somewhat steeper than Backbone (which is quite easy to master), but the gain is appreciative. Documentation could be better.*

KNOCKOUTJS

Pros: *I don't necessarily use it all the time, but KnockoutJS is just fantastic for single page applications. Extremely easy subscribing to live sorting; much better API for so called "collection views" in Backbone using observable arrays. And custom event on observables for effects, etc.*

Cons: *Feel like the API is quite hard to scale, and would probably prefer to wrangle Backbone on the bigger applications. (But that's also partially due to community support).*

Pros: *I like the data binding mechanism and feel very comfortable using it. In particular I like how they have replaced templates with control flow binding.*

Cons: *I don't like that there is no guidance or best practice in terms of application structure. Aside from having a view model, the framework doesn't help you in defining a well structured view model. It's very easy to end up with a large unmaintainable function.*

DOJO

Pros: *Syntactically, Dojo is very simple. It allows for dynamic and robust builds, with the initial loader file being as low as 6k in some cases. It is AMD compatible, making it extremely portable, and comes out-of-the-box with a ton of features ranging from basic dom interactions to complex SVG, VML, and canvas functionality. The widget system, Dijit, is unmatched in it's ease-of-use and ability to be extended. It's a very well-rounded and complete toolkit.*
Cons: *The dojo/_base/declare functionality is not 100% strict mode compliant and there is currently some overhead due to backwards compatibility, though this will mostly go away in the Dojo 2.0 release.*

Pros: *Good components : tabs, datagrid, formManager... Renders the same cross browser. AMD compliant. Easy to test with mocks.Integrates well with other frameworks thks to amd (I ll integrate with JMVC)*
Cons: *Default design for components out of fashion. Not fully html5. So-so documentation Poor templating system (no auto binding).*

YUI

Pros: *YUI3 is a modular and use-at-will type of component library which includes all of the goodies of Backbone and more. It even (in my opinion) improves upon some of the concepts in Backbone by de-coupling some things (i.e. attribute is a separate module that can be mixed into any object – the event module can be mixed in similarly).*

Cons: *I'd love to see YUI3 support some of the auto-wiring (optional) of Ember. I think that is really the big win for Ember; otherwise, I see YUI3 as a superior component library where I can cherry-pick what I need. I'd also like to see a more AMD-compatible module loader. The loader today works very well; however, it would be nicer if I could start a new projects based on AMD modules and pull in certain YUI3 components and other things from other places that are also using AMD.*

JAVASCRIPTMVC

Pros: *Has all tools included, just need to run commands and start building. I have used for the last 6 months and it's been really good.*

Cons: *The only thing I would do is to speed up development of the next version. Developers are aware of problems and fixing issues but its going to be another ¾ months before some issues I want fixed are addressed, but then I could probably patch and do a pull request.*

MARIA

Pros: *Because Maria is a pure MVC framework that is focused on being just an MVC framework. No more and no less. Its clean and simple.*

Cons: *A little more usage documentation outside of the source code, plus a few more test cases. A tutorial that drives home the real use of MVC with Maria would be good too.*

CUJO.JS

Pros: *Real apps almost never fit perfectly into an MV* box, and the most important stuff is often outside the box. With cujo.js, you define the box.*

Yes, cujo.js has high-level MV-like features for creating views, models, controllers, etc., but every app is different, and no framework can ever be a 100% solution. Rather than try to be all things, cujo.js also provides lower level tools, architectural plumbling, and a rich plugin system that can even be used to integrate and extend other MV* frameworks.*

Create the architecture that best suits your application, rather than constraining your app to fit inside someone else's predefined architecture.

Cons: *The broader JavaScript community is totally unprepared and untrained to take on large-scale applications. Most of us don't even know that design patterns and architectural patterns exist.*

Since cujo.js is so different from other frameworks, it needs more than a simple API reference and code snippets. Without tutorials, educational materials, and step-by-step examples, cujo.js might look strange and overwhelming to the untrained eye but documentation is supposed to be coming soon.

EXTJS

Pros: *I think ExtJS works best in combination with Ext Designer. It gives it an edge beyond the other GUI frameworks by letting non-programmers mock up the UI so programmers can fill in the blanks. I think comparing it to MVC frameworks like Backbone doesn't do it justice – its strength lies in creating rich GUIs, not lean Web apps.*

For rich, commercial back-office applications I think ExtJS remains the best choice when it comes to JavaScript solutions (i.e. not GWT etc). For public-facing Web apps I'd rather have something that gives me more control over the markup (and ideally something that degrades gracefully).

Cons: *It has a steeper learning curve than many of the other modern structural frameworks. One can argue that if you're investing in ExtJS for the long-term this time spent learning will pay off, however I think solutions like it should aim to better minimize the time it takes to train teams up in using it.*

Pros: *I think a big feature of ExtJS 4 is that it throws you into the MVC mindset and the preferred filesystem structure right from the bat. With Dojo the initial tutorials seem to be mostly about augmenting existing websites whereas ExtJS assumes you're starting from scratch.*

Using ExtJS doesn't really "feel" like you're dealing with HTML at all. The component library is rich enough to let you go a long way without touching more HTML than what is needed to bootstrap your app.

It'd be interesting to see how both compare when Web components become more widely supported. This would finally allow manipulating the DOM without being afraid of breaking any widgets or causing your app's internal state to become inconsistent.

Cons: *The licensing is considered restrictive and difficult to understand by some. More people would be investing in ExtJS if it was clearer what the upfront and long-term costs of using it are. This isn't a concern with some other structural solutions but probably isn't as much a worry for larger businesses.*

Pros: *ExtJS is a fantastic package for rapidly building out RIAs for internal use. I for one, love to build with HTML and JavaScript, and for me there's great satisfaction in mucking around at that level. Even though ExtJS makes it feel like you're not really working with HTML it still offers a great deal of power, especially if you're using it to create a complex UI.*

Cons: *That said…I absolutely agree that it's very heavy and I don't think I'd recommend it for an external facing Web application. My biggest beef with the package overall is actually that it's more of a PITA to test with than I'd would like. Our tester actually ended up switching to Sikuli because it was becoming too much of a battle trying to work with it in Selenium.*

BATMAN

Pros: *It has a great and easy to use view bindings system. Plays with Rails very nicely and is all about convention over configuration.*

Cons: *The documentation could be a lot better and I feel Shopify won't be adding the features that they say that they will.*

Don't Be Afraid To Experiment

Whilst it's unlikely for a developer to need to learn how to use more than a handfull of these frameworks, I do encourage exploration of those you're unfamiliar with. There's more than mountain of interesting facts and techniques that can be learned in this process.

In my case: I discovered that Batman.js required the least hand-written lines of code for an implementation. I'm neither a frequent CoffeeScript nor Batman.js user but that in itself gave me some food for thought. Perhaps I could take some of what made this possible and bring it over to the frameworks I do use. Or, maybe I'd simply use Batman.js in a future project if I found the community and support around it improved over time.

Regardless of whether you end up using a different solution, at the end of the day all you have to gain from exploration is more knowledge about what's out there.

Going Beyond MV* Frameworks

Whilst the MV* family of patterns are quite popular for structuring applications, they're limited in that they don't address any kind of application layer, communication between Views, services that perform work or anything else. Developers may thus find that they sometimes need to explore beyond just MVC — there are times when you absolutely need to take what they have to offer further.

We reached out to developers that have been taking MVC further with their own patterns or extensions for existing frameworks to get some insights on where you need something more.

"In my case, I needed something Composite. I noticed that there were patterns in Backbone apps where developers realized there was a need for an object that coordinated various parts of an application. Most of the time, I've seen developers try to solve this using a Backbone construct (e.g a View), even when there isn't really a need for it. This is why I instead explored the need for an Application Initializer[63].

I also found that MVC didn't really describe a way to handle regions of a page or application. The gist of region management[64] is that you could define a visible area of the screen and build out the most basic layout for it without knowing what content was going to be displayed in it at runtime.

I created solutions for region management, application initialization and more in my extension project Marionette. It's one of a number of solutions that extend upon a framework (or architecture pattern) that developers end up needing when they're building single-page applications that are relatively complex.

There's even a TodoMVC Marionette app[65] available for anyone wishing to compare the standard Backbone application with one that goes beyond just MV*.

Derick Bailey — Author of Marionette

"While a good portion of problems can be decomposed into JavaScript MVC, there are some which simply cannot. For example, an application consumes a third party API at runtime, but is not given any information as to how the data will be structured.

I spent almost a year trying to solve that very problem, but eventually I came to the realization that shoehorning it into MV* was not a viable solution. I was dealing with an "amorphous model" and that's where it all fell apart. In other words, if you don't have a well-defined model, most modern JavaScript frameworks can't help you.

That's where Core J2EE Patterns[66] come in. I got turned on to them while reading PHP Objects, Patterns, and Practice[67] by Matt Zandstra, and I'm glad I did! The J2EE Patterns basically outline a request-driven process, where the URL drives the behavior of the application. In a nutshell, a request is created, modified, and then used to determine the view to render.

I've expanded on my experiences with request driven Javascript applications and J2EE patterns[68] for anyone who would like to learn more. "

Dustin Boston — co-author, Aura

Conclusions

While there are several choices for what to use for structuring your JavaScript Web applications these days, it's important to be **diligent** in the selection process – spend time thoroughly evaluating your options in order to make a decision which results in sustainable, **maintainable** code. Framework diversity fosters innovation, while too much similarity just creates noise.

Projects like TodoMVC can help narrow down your selections to those you feel might be the most interesting or most comfortable for a particular project. Remember to take your time choosing, don't feel too constrained by using a specific pattern and keep in mind that it's completely acceptable to build on the solution you select to best fit the needs of your application.

Experimenting with different frameworks will also give you different views on how to solve common problems which will in turn make you a better programmer.

Thanks to my fellow TodoMVC team-member Sindre Sorhus[69] for his help with tweaks and a technical review of this article.

FOOTNOTES

1 https://developer.mozilla.org/en/DOM/About_the_Document_Object_Model

2 http://jquery.com

3 http://en.wikipedia.org/wiki/Spaghetti_code

4 http://addyosmani.com/resources/essentialjsdesignpatterns/book/#detailmvcmvp

5 http://backbonejs.org

6 http://addyosmani.com/resources/essentialjsdesignpatterns/book/#detailmvcmvp

7 http://gmail.com

8 http://docs.google.com

9 http://addyosmani.com/blog/digesting-javascript-mvc-pattern-abuse-or-evolution/

10 http://backbonejs.org

11 http://emberjs.com

12 http://spinejs.com

13 http://canjs.us

14 http://github.com/wycats

15 http://github.com/jashkenas

16 http://todomvc.com

17 http://www.todomvc.com

18 http://dojotoolkit.org

19 http://dylanschiemann.com

20 http://emberjs.com

21 http://angularjs.org

22 http://spinejs.com

23 http://knockoutjs.com

24 http://dojotoolkit.org

25 http://yuilibrary.com

26 http://batmanjs.org

27 http://code.google.com/closure/library/

28 http://agilityjs.com

29 http://kmalakoff.github.com/knockback

30 http://requirejs.org

31 http://emberjs.com

32 http://requirejs.org

33 http://canjs.us

34 https://github.com/petermichaux/maria

35 http://cujojs.github.com

36 http://meteor.com

37 http://www.socketstream.org

38 http://jquery.com

39 http://www.sencha.com/products/extjs

40 http://sammyjs.org

41 http://javascriptmvc.com

42 https://developers.google.com/web-toolkit/

43 https://github.com/troopjs

44 http://hay.github.com/stapes

45 http://somajs.github.com/somajs

46 https://bitbucket.org/mckamey/duel/wiki/Home

47 https://github.com/jgallen23/fidel

48 https://github.com/flams/olives

49 https://github.com/rhysbrettbowen/PlastronJS

50 https://github.com/creynders/dijon-framework

51 http://www.rappidjs.com

52 https://github.com/brokenseal/broke

53 http://weepy.github.com/o_O

54 https://github.com/marcuswestin/fun

55 http://angularjs.org

56 http://requirejs.org

57 https://github.com/derickbailey/backbone.marionette

58 https://github.com/chaplinjs/chaplin

59 https://github.com/addyosmani/backbone-aura/

60 https://github.com/walmartlabs/thorax

61 https://github.com/backbone-boilerplate/grunt-bbb

62 http://brunch.io/

63 http://lostechies.com/derickbailey/2011/12/16/composite-javascript-applications-with-backbone-and-backbone-marionette/

64 http://lostechies.com/derickbailey/2011/12/12/composite-js-apps-regions-and-

region-managers/

65 https://github.com/derickbailey/todomvc/tree/marionette

66 http://java.sun.com/blueprints/corej2eepatterns/Patterns/

67 http://www.amazon.com/Objects-Patterns-Practice-Matt-Zandstra/dp/1590599098

68 http://dblogit.com/archives/3895

69 http://sindresorhus.com/

Hold on tiger! Thank you for reading the article. Did you know that we also publish printed books and run friendly conferences – crafted for pros like you? For example, Smashing Book 5, *packed* with smart responsive design patterns and techniques.

Addy Osmani

Addy Osmani is a Developer Programs Engineer on the Chrome team at Google. A passionate JavaScript developer, he has written open-source books like 'Learning JavaScript Design Patterns' and 'Developing Backbone Applications', having also contributed to open-source projects like Modernizr and jQuery. He is currently working on 'Yeoman' - an opinionated workflow for building beautiful applications.

With a commitment to quality content for the design community. Founded by Vitaly Friedman and Sven Lennartz. 2006-2012. Made in Germany. http://www.smashingmagazine.com