# Making Sense of Application Architecture Choices

**Summer 2014**

**Art Kay, Developer Relations Manager Sencha, Inc**

## Abstract

Just like functions and objects help developer efficiency by allowing code reuse and modularity at a micro level, application architectures help team efficiency at the macro level by organizing, abstracting and restricting how large pieces of code interact.

This paper summarizes the currently popular JavaScript application architectures and takes a deep dive into the latest application architecture that Sencha supports – Model-View-ViewModel (MVVM).

## Introduction

The Sencha Ext JS framework has become an industry standard for developing enterprise web applications thanks to its comprehensive widget library, powerful data package and robust tooling. Ext JS has proven to be a highly scalable and easily customizable framework, with over 60% of Fortune 100 companies and more than 2 million developers worldwide using it.

Since the original Ext JS 1.0 release in 2007, a lot has changed in our industry – and web applications are certainly larger and more complex than ever. In 2010, Sencha released Touch 1.0, delivering the industry's first JavaScript framework to support a Model-View-Controller pattern (MVC), addressing the architectural problems often faced by large enterprise web applications. We then applied that feature to Ext JS 4.0 in 2011, helping to organize application code in the new world of enterprise web apps.

Sencha has recently released Ext JS 5.0 with optional support for the MVVM architectural pattern. Ext JS 5 includes features such as two-way data binding and declarative configuration. We know that enterprise web applications can be extremely diverse, and that it's absolutely critical to choose the correct architecture at the beginning of a project to ensure its success. By supporting both MVC and MVVM, Ext JS 5 is the most flexible option among popular JavaScript frameworks for building enterprise web applications.

## The Evolution of Enterprise Applications

For most of its early history, the Internet was nothing more than a linked web of simple HTML pages. The term "web application" didn't yet exist, as web technologies (browsers, tooling, even the raw language specifications) couldn't conceivably envision "applications" as we know them today.

Beginning in 1995, JavaScript was useful for little more than form field validation and image rollovers. The popular server-side languages at that time (CGI, PHP, ASP, Cold Fusion, etc.) delivered static HTML pages. Each time the user clicked on a link or filled out a form, the server spit out a brand new page. Even the very coolest websites, created by graphic designers using tools like Photoshop, lacked any real functionality or interactivity.

 1700 Seaport Blvd. Suite 120, Redwood City, CA 94063 1 (800) 212–2802 www.sencha.com @sencha

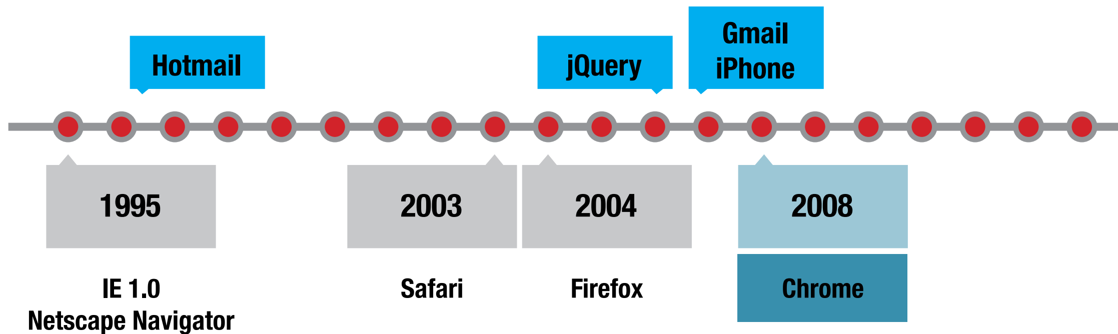
```
<html>
        <head></head>
        <body>
                <form onsubmit=”return validate(this);”>
                        <input type=”text” name=”name” id=”name”>
                </form>

                <script>
                        function validate(form) {
                                // DO STUFF
                        }
                </script>
        </body>
</html>
```

The little bit of JavaScript that was being written simply attached rudimentary functions to DOM nodes. HTML markup and JavaScript code existed in the same file – and this mess of unstructured code often resulted in memory leaks.

## Browser Wars: 1995 - 2008

This traditional webpage architecture began to change as new browsers appeared, better technologies emerged and faster internet connections became prevalent.




- In 1995, Internet Explorer 1.0 shipped and its only real competitor was Netscape Navigator 1.0.
- Several versions of Opera shipped in the late 1990s, though its market share and impact was minimal.
- In 2003, Apple first released Safari.
- In 2004, Mozilla first released Firefox.
- In 2006, the world saw jQuery for the first time.
- In 2007, Apple released the first iPhone which shipped with the first true "mobile" browser.
- In 2008, Google first released Chrome.

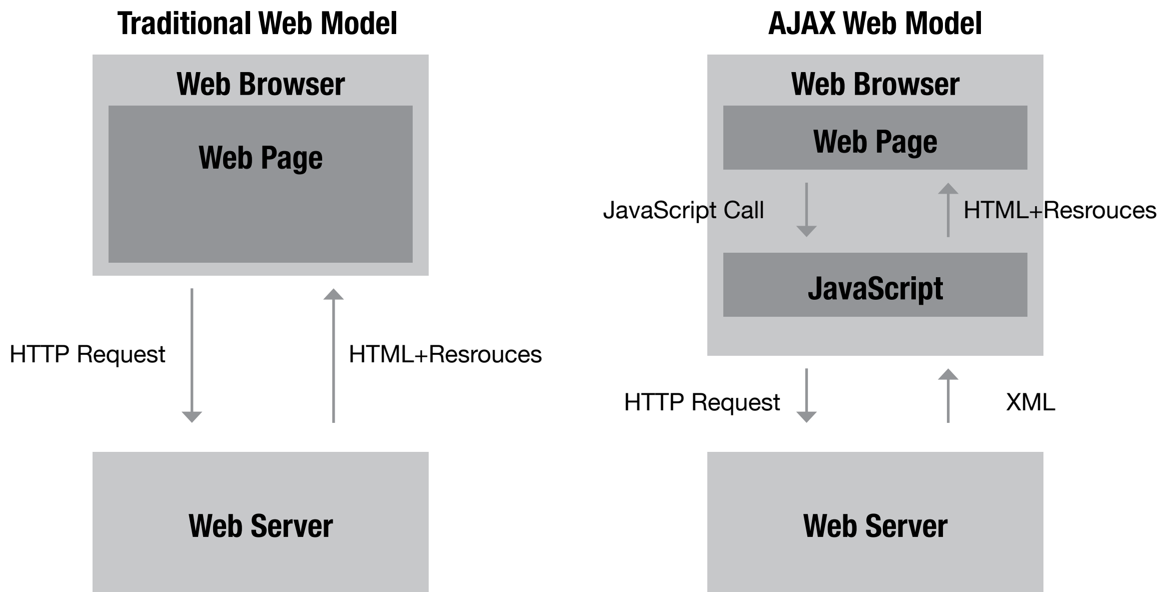
With more browsers, it became harder to deliver consistent websites because of variations in the JavaScript and rendering engines. To help combat the cross-browser differences, a few new JavaScript libraries appeared. jQuery, Prototype, and YUI all became very popular as developers faced a critical need to write code that would work on all platforms.

## The Rise of JavaScript and AJAX

In spite of the variations in behavior seen across the new multitude of browsers, the most significant benefit of the browser wars was that *every* browser received new features and improvements in capabilities. Although not every browser implemented things like CSS specs the same way, the fact remained that all browsers could do mostly the same things.

But not long after Google Maps shipped in February 2005 – the first mass-market AJAX application – the Internet began to completely change. The World Wide Web, which was previously dominated by static web pages, saw a complete paradigm shift in how "web applications" were engineered.

AJAX no longer required the server to completely re-render the page; the client-side applications instead requested data directly from server APIs and REST endpoints and managed the DOM themselves.




By 2006 or so, companies realized the potential behind the web. Company websites were no longer just static listings of their products; websites became products themselves, and customers expected to log in to their business applications from anywhere.

When the purpose of the Internet began to change, so did the teams who built those websites. Graphic designers were now augmented by more seasoned developers who could take the designs and add the interactions and business logic to create an application.

```
<html>
        <head></head>
        <body>
                <h1>I Haz Internet</h1>
                <p id=foo>Circa 1995</p>
        </body>
</html>
```

```
$(document).ready(function() {
        $(foo).on('click', function(e) {
        // Do Something //
        }})
        .addCls(bar)
        .html(<p>random text</p>)
        .slideUp(2000)
        .css(color, red)
        // etc…
});
```

Oddly enough, the underlying code for these new web applications didn't change much from what we had seen on static websites just a few years prior.

HTML pages did manage to shed some weight relating to frames and forms. Due to the rise of AJAX, the size of the HTML files actually began to shrink because client-side applications could request data directly from server APIs and REST endpoints.

JavaScript now managed more and more of the DOM. Libraries like jQuery certainly helped to iron out a lot of manual work required to deliver apps across browsers… but the code remained largely unorganized. JavaScript was still being attached to DOM nodes and often resulted in memory leaks. All of the logic was shoved into a single file leading to a gigantic mess of spaghetti code, and the convoluted method chains resulted in code that was difficult to maintain.

For smaller applications and simpler websites, the lack of any reasonable architecture was a headache, but definitely not a deal breaker. However, for enterprise applications, this presented a huge problem.

Enterprise companies were slow to adopt the new browsers and emerging web technologies. But as these companies migrated their applications to the web, they also noticed that libraries like *jQuery* didn't quite solve the problems they faced when building larger applications.

While jQuery's AJAX and CSS utilities were certainly useful, the complete lack of JavaScript architecture caused many problems for development and long-term maintenance. Memory leaks crippled applications, adding

features took months rather than weeks, and training new developers on legacy code became impossible. In short, enterprise web applications could not scale using the existing JavaScript libraries.

Furthermore, enterprise companies needed a standardized approach to building their applications. Having consistent APIs and server-side architectures helped reduce development and maintenance costs – but without anything like that on the client, enterprise companies struggled to find a solution they could rely on.

The enterprise desperately needed a solution which would bring order to web applications. Sure the companies with really large enterprise apps needed to organize their code to avoid a mess of spaghetti…but they also needed a consistent paradigm for building applications, one that could be taught to a diverse development team and scale as new features were added.

Ext JS 1.0 was released in April 2007, and it was among the first true JavaScript frameworks that attempted to solve these problems faced by the enterprise.

Ext JS first started as an extension of the YUI library (termed “yui-ext”) in early 2006, but it quickly grew into an independent library. Like other JavaScript libraries at the time, Ext JS 1.0 was a widget library created to help developers build Rich Internet Applications that functioned consistently across browsers. Later that year, Ext JS 2.0 was released as a self-contained framework featuring more widgets, superior documentation, better examples, and even contained its own base cross-browser abstraction layer.

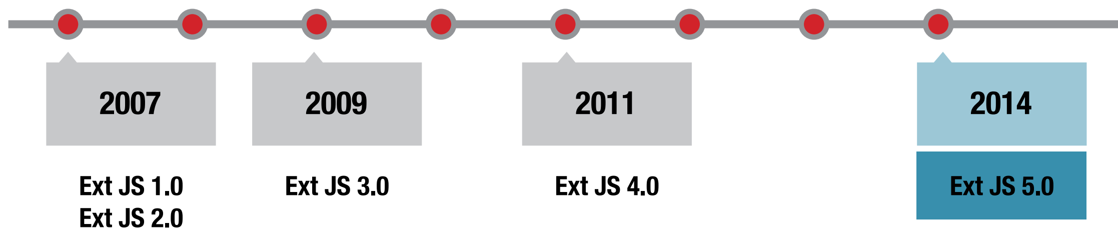
In 2007, Ext JS was very different from all other JavaScript libraries. Rather than attaching random JavaScript to DOM nodes, Ext JS introduced the approach of object oriented programming and inheritance to web applications by making JavaScript the focus of development rather than HTML. This helped conceptualize application programming, stopping the paradigm of endless closures and spaghetti code, and eliminating common memory leaks.

```html
<html>
        <head></head>
        <body>
        </body>
</html>
```

```javascript
var onClickHandler = function(e) {
        // Do Stuff //
};

var widget = new Ext.Container({
        height: 100,
        width: 300,
        renderTo: Ext.getBody(),
        listeners: {
                click: onClickHandler
        }
});
```

The most revolutionary part of Ext JS was the component lifecycle. Each widget in the framework had its own lifecycle – so developers could easily modify component state through custom events. Event logic was cleaned up properly upon component destruction, so memory leaks were no longer something to worry about. And because components were not tightly bound to the DOM, the developers could spend more time adding features and functionality rather than managing HTML and JavaScript spaghetti code.




Ext JS 3.0 was released in 2009, and it featured full support for REST communication, a charting package, and even more widgets.

In 2011, Sencha released Ext JS 4, and it added support for MVC architecture, a revised data package and an overhauled charting library. This was a revolutionary release, not just for our framework but also in comparison to other frameworks at that time, because Ext JS was the first to address the problems faced by enterprise web applications.

### 2014: Ext JS 5

Sencha approached the Ext JS 5 release a bit differently. We wanted to carefully engineer a product that solved the needs of an enterprise application, without breaking any of the hard work our customers had put into their existing apps.

On the one hand, we wanted Ext JS 5 to maintain full backwards compatibility with applications built with an MVC architecture in Ext JS 4. On the other hand, we wanted to help developers increase their productivity and decrease code complexity. We believe that adding support for MVVM architecture did just that.

In a nutshell, Ext JS has evolved over the years to help address the architectural issues faced by enterprise web applications. Now, let's take a deeper look at the MVC and MVVM patterns and see what specific problems they solve.

## Making Sense of MVC and MVVM

Application architecture is as much about providing structure and consistency as it is about actual classes and framework code. Building a good architecture unlocks a number of important benefits:

- Every application works the same way, so you only have to learn it once.
- It's easy to share code between apps, because they all work the same way.
- It's harder for developers to create overlapping and conflicting functionality.

These points are particularly important in the enterprise where development teams are both large and diverse – potentially consisting of hundreds of developers spread across the world. These "teams" can change frequently, and their projects can vary in duration from weeks to years. With so much time and money invested in the software development cycle, there is little doubt why enterprises want to create a standard way of building applications.

While many application architectural patterns exist, the two most popular in web development over the past several years have been MVC and MVVM. Although similar, the two have subtle differences, which can have significant architectural consequences if they're not clearly understood.

Let's first explore what the MVC and MVVM patterns are, and then examine how using Ext JS 5 helps the enterprise solve specific architectural problems.

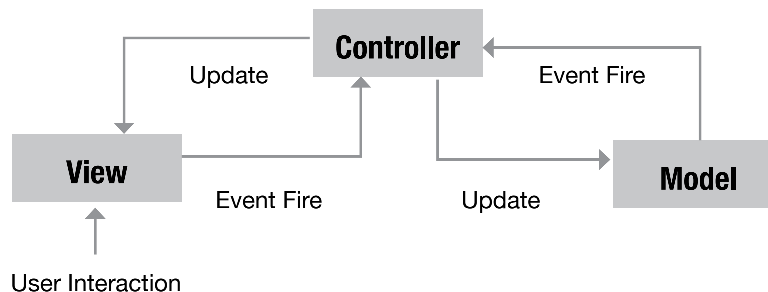
### What is MVC?

Model–View–Controller (MVC) is an architectural pattern for writing software. It divides the user interface of an application into three distinct parts, helping to organize the codebase into logical representations of information depending upon their function.

MVC was originally conceived in the 1970s and was one of the first approaches to describing software in terms of the conceptual "responsibilities" for individual pieces of a program. MVC has evolved since its initial introduction, and many variations (such as MVP, MVVM and others) have appeared.

As a result of these variations, there is often confusion surrounding exactly what MVC constitutes, but in essence: The *Model* describes a common format for the data being used in the application. It may also contain business rules, validation logic and various other functions.

- The *View* represents the data to the user. Multiple views may display the same model data in different ways (e.g. charts versus grids).
- The *Controller* is the central piece of an MVC application. It listens for events in the application and delegates commands between the *Model* and the *View*. It may also contain business rules.




In MVC architecture, every object in the program is either a *Model*, a *View* or a *Controller*. The user interacts with *Views,* which display data held in a *Model*; those interactions are monitored by a *Controller*, which then responds to the interactions by updating the *View* and *Model,* as necessary.

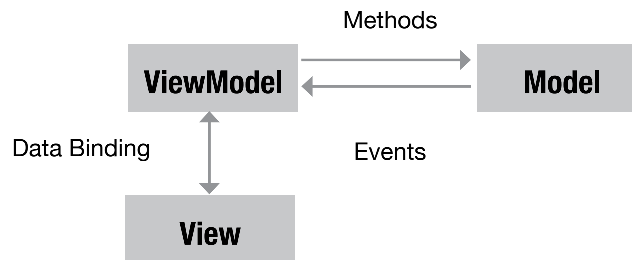
The View and the Model are mostly unaware of each other because the Controller has the sole responsibility of directing updates. Views typically have little (if any) business logic, and Models are simple interfaces to data – meaning that Controllers hold most of the application logic within an MVC application.

Perhaps the best part about using MVC architecture in web applications is that it helps to avoid enormous files full of spaghetti code. The goal of MVC is to clearly define the responsibilities of each piece of the application. Because every class (i.e. every file) has clearly defined responsibilities, they implicitly become ignorant of the larger environment – making the app easier to test, easier to maintain, and allowing code to be reused.

## What is MVVM?

Model-View-ViewModel (MVVM) is another architectural pattern for writing software that is largely based on the MVC pattern. It was conceived as a specialization of Martin Fowler's ***Presentation Model*** design pattern, in which the GUI configuration is clearly separated from any business logic.

The key difference between MVC and MVVM is that MVVM features an abstraction of a View (the ViewModel), which manages the changes between a Model's data and the View's representation of that data (i.e. data bindings) – something that typically is cumbersome to manage in traditional MVC applications.




The result is that the *Model* and framework perform as much work as possible, minimizing or eliminating application logic that directly manipulates the *View*.

The MVVM pattern includes the following elements:

- The Model describes a common format for the data being used in the application, just as in the classic MVC pattern.
- The View represents the data to the user, just as in the classic MVC pattern.
- The ViewModel is an abstraction of the View that mediates changes between the View and an associated Model. In the MVC pattern, this would have been the responsibility of a specialized Controller, but in MVVM, the ViewModel directly manages the data bindings and formulas used by the View in question.

## MVC and MVVM in Ext JS 5

Now that we have a basic understanding of both MVC and MVVM from a theoretical standpoint, let's examine how Sencha applications implement MVC and MVVM. We will also investigate some shortcomings of MVC and discuss how MVVM attempts to solve these issues.

### MVC in Ext JS

#### Client-Side vs. Server-Side

Because of the success we saw in the enterprise with server-side frameworks like Apache Struts and ASP.NET MVC, we modeled our approach to MVC on these paradigms. But when you read enough descriptions of MVC, MVP and the related architectural patterns, it becomes clear that there is no single definition of MVC, and certainly not everyone agrees.

When we first introduced MVC support for Ext JS 4 in 2011, It wasn't a surprise that there was some initial confusion surrounding our approach. Ext JS is a client-side application framework, and therefore a Sencha "application" runs inside what might be considered a "View" in terms of the traditional server-side architecture.

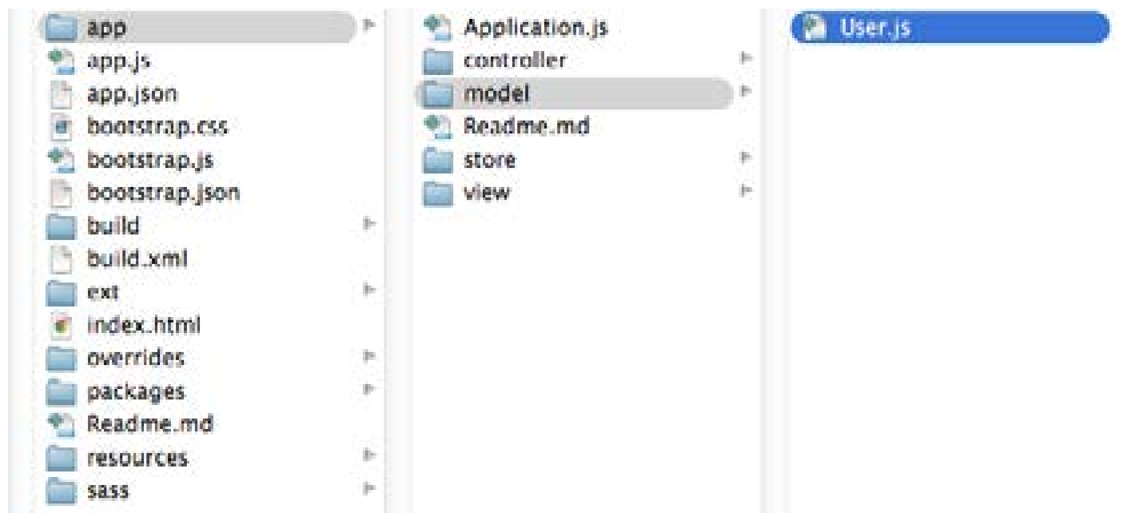
Because Sencha frameworks are server-side agnostic, we try to separate the idea of "architecture on the client" from "architecture on the server" because ultimately Sencha applications don't care about the lower tiers of the underlying project.

## Ext JS Application Architecture

Ext JS is a true “framework” in that it acts as a universal software platform for developing web applications. Beyond the architectural paradigm for structuring application code, Ext JS features many extensible widgets and utilities for manipulating data. As such, the framework controls the application lifecycle, allowing the developer to focus on the functionality of the program.

In an MVC application, Ext JS has specific classes to manage *Controllers* (Ext.app.Controller) and *Models* (Ext.data.Model), but *Views* are defined by extending any of the framework’s widgets (which all inherit from Ext.Component).

*Models*, *Views* and *Controllers* in an MVC application will follow a clearly defined naming convention (e.g. *MyApp.model.User*), which correlates to their location in the filesystem (e.g. ~/app/model/User.js).



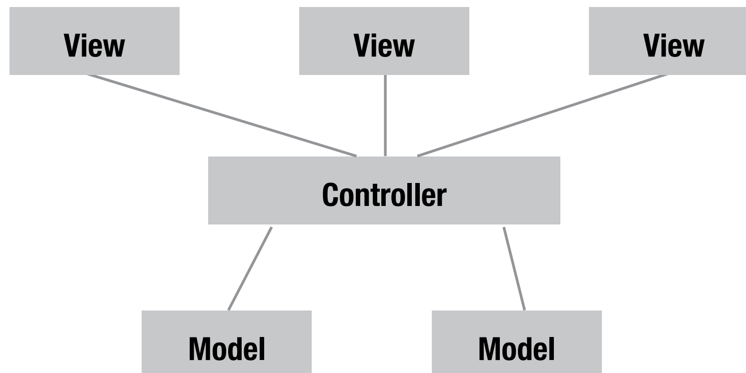

The Ext JS framework then dynamically loads all application dependencies at runtime (and can optionally compile them for deployment).

## Controllers: Pros and Cons

While MVC, in general, did help enterprises build applications with well-organized code, we also found that MVC had a few shortcomings.

Over time, our Professional Services and Support teams reported that many customer applications contained a very small number of Controllers that each spanned thousands of lines of code – a problem leading to poor performance and long-term maintenance problems.

In the Ext JS approach to MVC, *Controllers* are also globally scoped (to the application). This results in additional business logic to grab references to *Views*, *Models* and other objects. *Controllers* could be written to watch any object at any time, so any given *Controller* might have logic for both *View A* and *View B*, leading to additional confusion in large applications.




Finally, we found that writing unit tests against MVC applications was difficult. *Models*, *Views* and *Controllers* are supposed to be loosely coupled, but in reality, testing a *Controller* requires knowledge of both the relevant *Models* and *Views*. Because *Controllers* often listened for events across multiple *Views*, customers found it hard to test individual "units" without launching the entire application to satisfy these dependencies.

| Pros | Cons |
| --- | --- |
| Organized Code<br>Separation of Concerns | Extra Business Logic<br>Global Controllers<br>Refs<br>Testing Challenges |

In summary, MVC architecture has some very good benefits for web applications and Ext JS 5 will continue to support this pattern into the future. But MVC does have a few shortcomings which Ext JS 5 addresses by adding support for MVVM.

### MVVM in Ext JS

As noted in the previous section, the biggest complaint about MVC was that global (decoupled) *Controllers* seem great in theory, but they can be difficult to manage in practice. MVVM can mitigate this problem by coupling *Views* to a specific *ViewModel* (and/or *ViewController*) instance, resulting in superior organization, maintainability and testability.

### Ext JS Application Architecture

In Ext JS 5 with MVVM, not much has actually changed in terms of the overall architecture. *Models* still extend *Ext.data.Model*, and *Views* still extend any of the framework's widgets. The *ViewModel* works essentially the same way, using the new *Ext.app.ViewModel* class.

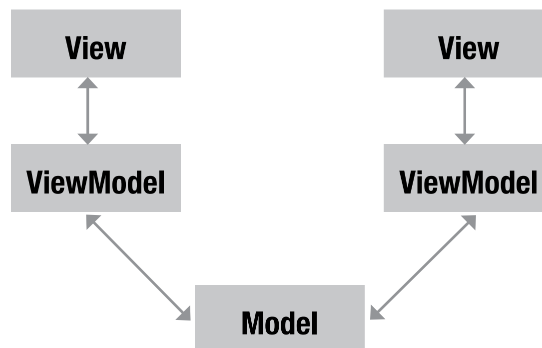
*Models*, *Views* and *ViewModels* in an MVVM application continue to follow the clearly defined naming convention used previously, where *MyApp.model.User* would correlate to ~/app/model/User.js.

As such, any application written in Ext JS 4.x should upgrade to Ext JS 5.x without any problems from an architectural standpoint.

### ViewModels

The major difference between MVC and MVVM comes down to the *ViewModel*. At a high level, MVC *Controllers* and MVVM *ViewModels* are very similar; however, the ViewModel facilitates updates in the Model and the View through data binding rather than through events.

The *ViewModel* also participates in the component lifecycle. For every instance of a *View*, unique instances of the configured *ViewModel* are also created. The *ViewModel* is then subsequently destroyed when the associated *View* is destroyed.




### ViewControllers

Despite the name Model-View-ViewModel, the MVVM pattern in Ext JS may still use Controllers – although one might choose to then call it a hybrid MVC+VM architecture. Confusing acronyms aside, the point is that each of these approaches has merit, and Ext JS 5 is flexible enough to support both.

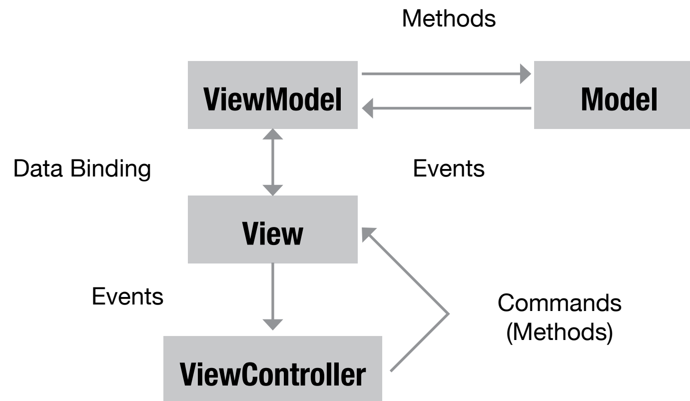
Ext JS 4 introduced MVC *Controllers* as application-wide event listeners (e.g. a publish/subscribe model or event bus), and Ext JS 5 still supports that concept. However, Ext JS 5 also supports a new variation termed *ViewController*.

A *ViewController* in Ext JS is similar in nature to a *ViewModel*. Both constructs are scoped directly to the related *View*, eliminating much of the overhead required in traditional MVC to manage object references and restore application state.

*ViewControllers* are also similar to traditional (i.e. application-wide) MVC *Controllers* from Ext JS 4 in that they listen for events and execute logic in response to those events. However, a major difference between *ViewControllers* and “traditional” *Controllers* is that individual *ViewControllers* are created for each related *View*, whereas *Controllers* are singular constructs listening globally across multiple *Views*.

*ViewControllers* also allow us to write declarative listeners, which reduces the complexity of our application code and helps keep the *View* free from business logic.

Lastly, ViewControllers and ViewModels participate in the Component lifecycle – meaning that for every instance of a View, unique instances of the configured ViewModel and ViewController are also created. The ViewModel and ViewController are subsequently destroyed when their associated View is destroyed.




The result of tightly coupled *Views* and *ViewControllers* (and/or *ViewModels)* is that Ext JS applications now become far easier to unit test. In MVVM, we can easily avoid writing bloated global *Controllers*, leading to components that can be completely isolated for testing.

It is important to keep in mind that not all Views in the Ext JS approach to MVVM require a ViewController or ViewModel – they are completely optional parts of the architecture.

## Conclusion

Ext JS 4 paved the way for enterprise web applications to begin using MVC – defining a consistent architecture for organized code. Ext JS 5 builds upon the success of MVC by adding support for MVVM while also maintaining backwards compatibility. Developers should have no problems upgrading all apps built using Ext JS 4 and MVC to the latest version, and will not have to re-architect their applications.

Client-side MVC was a huge step forward for enterprise web applications. It offered a solution to unorganized spaghetti code and helped applications scale easily. MVVM builds upon the foundations of MVC by incorporating data binding and declarative listeners, eliminating lots of code and making applications easier to unit test!

## Resources:

- Ext JS 5: MVC, MVVM and more!
- Ext JS 5: View Models and Data Binding
- Understanding Sencha Cmd Packages
- Model-View-ViewModel for iOS
- MVC or MVP Pattern - What's the Difference?
- StackOverflow: What are MVP and MVC and what is the difference?
- GUI Architectures