addyosmani.com

Understanding MVC And MVP (For JavaScript And Backbone Developers)

20 min read • original

Last updated: 16th Jan, 2012.

Before exploring any JavaScript frameworks that assist in structuring applications, it can be useful to gain a basic understanding of architectural design patterns. Design patterns are proven solutions to common development problems and can suggest structural paradigms to help guide us in adding some organization to our application.

I think patterns are exciting as they're effectively a grass roots effort that build upon the collective experience of skilled developers who have previously faced similar problems as we do now. Although developers 10 or 20 years ago may not have been using the same programming languages for implementing patterns, there are many lessons we can learn from their efforts.

In this section, we're going to review two popular patterns – MVC and MVP. The context of our exploration will be how these patterns are related to the popular JavaScript framework Backbone.js, which will be explored in greater detail later on.

MVC

MVC (Model-View-Controller) is an architectural design pattern that encourages improved application organization through a separation of concerns. It enforces the isolation of business data (Models) from user interfaces (Views), with a third component (Controllers) (traditionally) managing logic, user-input and coordinating both the models and views. The pattern was originally designed by Trygve Reenskaug during his time working on Smalltalk-80 (1979) where it was initially called Model-View-Controller-Editor. MVC went on to be described in depth in "Design Patterns: Elements of Reusable Object-Oriented Software" (The "GoF" or "Gang of Four" book) in 1994, which played a role in popularizing its use.

Smalltalk-80 MVC

It's important to understand what the original MVC pattern was aiming to solve as it's mutated quite heavily since the days of it's origin. Back in the 70's, graphical user-interfaces were far and few between and a concept known as Separated Presentation began to be used as a means to make a clear division between domain objects which modelled concepts in the real world (e.g a photo, a person) and the presentation objects which were rendered to the user's screen.

The Smalltalk-80 implementaion of MVC took this concept further and had an objective of separating out the application logic from the user interface. The idea was that decoupling these parts of the application would also allow the reuse of models for other interfaces in the application. There are some interesting points worth noting about Smalltalk-80's MVC architecture:

- A Domain element was known as a Model and were ignorant of the user-interface (Views and Controllers)
- Presentation was taken care of by the View and the Controller, but there wasn't just a single view and controller. A View-Controller pair was required for each element being displayed on the screen and so there was no true separation between them
- The Controller's role in this pair was handling user input (such as key-presses and click events), doing something sensible with them.
- The Observer pattern was relied upon for updating the View whenever the Model changed

Developers are sometimes surprised when they learn that the Observer pattern (nowadays commonly implemented as a Publish/Subscribe system) was included as a part of MVC's architecture many decades ago. In Smalltalk-80's MVC, the View and Controller both observe the Model. As mentioned in the bullet point above, anytime the Model changes, the Views react. A simple example of this is an application backed by stock market data – in order for the application to be useful, any change to the data in our Models should result in the View being refreshed instantly.

Martin Fowler has done an excellent job of writing about the origins of MVC over the years and if you are interested in some further historical information about Smalltalk-80's MVC, I recommend reading his work.

MVC For JavaScript Developers

We've reviewed the 70's, but let us now return to the here and now. In modern times, the MVC pattern has been applied to a diverse range of programming languages including of most relevance to us: JavaScript. JavaScript now has a number of frameworks boasting support for MVC (or variations on it, which we refer to as the MV* family), allowing developers to easily add structure to their applications without great effort. You've likely come across at least one of these such frameworks, but they include the likes of Backbone,

Ember.js and JavaScriptMVC. Given the importance of avoiding "spaghetti" code, a term which describes code that is very difficult to read or maintain due to its lack of structure, it's imperative that the modern JavaScript developer understand what this pattern provides. This allows us to effectively appreciate what these frameworks enable us to do differently.

We know that MVC is composed of three core components:

Models

Models manage the data for an application. They are concerned with neither the user-interface nor presentation layers but instead represent unique forms of data that an application may require. When a model changes (e.g when it is updated), it will typically notify its observers (e.g views, a concept we will cover shortly) that a change has occurred so that they may react accordingly.

To understand models further, let us imagine we have a JavaScript photo gallery application. In a photo gallery, the concept of a photo would merit its own model as it represents a unique kind of domain-specific data. Such a model may contain related attributes such as a caption, image source and additional meta-data. A specific photo would be stored in an instance of a model and a model may also be reusable. Below we can see an example of a very simplistic model implemented using Backbone.

```
var Photo = Backbone.Model.extend({
    // Default attributes for the photo
    defaults: {
      src: "placeholder.jpg",
      caption: "A default image",
    viewed: false
    },
    // Ensure that each photo created has an `src`.
    initialize: function() {
       this.set({"src": this.defaults.src});
    }
});
```

The built-in capabilities of models vary across frameworks, however it is quite common for them to support validation of attributes, where attributes represent the properties of the model, such as a model identifier. When using models in real-world applications we generally also desire model persistence. Persistence allows us to edit and update models with the knowledge that its most recent state will be saved in either: memory, in a user's localStorage data-store or synchronized with a database.

In addition, a model may also have multiple views observing it. If say, our photo model contained meta-data such as its location (longitude and latitude), friends that were present in the a photo (a list of identifiers) and a list of tags, a developer may decide to provide a single view to display each of these three facets.

It is not uncommon for modern MVC/MV* frameworks to provide a means to group models together (e.g in Backbone, these groups are referred to as "collections"). Managing models in groups allows us to write application logic based on notifications from the group should any model it contains be changed. This avoids the need to manually observe individual model instances.

A sample grouping of models into a simplified Backbone collection can be seen below.

```
var PhotoGallery = Backbone.Collection.extend({
    // Reference to this collection's model.
    model: Photo,
    // Filter down the list of all photos
    // that have been viewed
    viewed: function() {
        return this.filter(function(photo){
           return photo.get('viewed');
        });
    },
    // Filter down the list to only photos that
    // have not yet been viewed
    unviewed: function() {
      return this.without.apply(this, this.viewed());
    }
});
```

Should you read any of the older texts on MVC, you may come across a description of models as also managing application 'state'. In JavaScript applications "state" has a different meaning, typically referring to the current "state" i.e view or sub-view (with specific data) on a users screen at a fixed point. State is a topic which is regularly discussed when looking at Single-page applications, where the concept of state needs to be simulated.

So to summarize, models are primarily concerned with business data.

Views

Views are a visual representation of models that present a filtered view of their current state. A view typically observes a model and is notified when the model changes, allowing the view to update itself accordingly. Design pattern literature commonly refers to views as 'dumb' given that their knowledge of models and controllers in an application is limited.

Users are able to interact with views and this includes the ability to read and edit (i.e get or set the attribute values in) models. As the view is the presentation layer, we generally present the ability to edit and update in a user-friendly fashion. For example, in the former photo gallery application we discussed earlier, model editing could be facilitated through an "edit" view where a user who has selected a specific photo could edit its meta-data.

The actual task of updating the model falls to controllers (which we'll be covering shortly).

Let's explore views a little further using a vanilla JavaScript sample implementation. Below we can see a function that creates a single Photo view, consuming both a model instance and a controller instance.

We define a `render()` utility within our view which is responsible for rendering the contents of the `photoModel` using a JavaScript templating engine (Underscore templating) and updating the contents of our view, referenced by `photoEl`.

The `photoModel` then adds our `render()` callback as one of it's subscribers so that through the Observer pattern we can trigger the view to update when the model changes.

You may wonder where user-interaction comes into play here. When users click on any elements within the view, it's not the view's responsibility to know what to do next. It relies on a controller to make this decision for it. In our sample implementation, this is achieved by adding an event listener to `photoEl` which will delegate handling the click behaviour back to the controller, passing the model information along with it in case it's needed.

The benefit of this architecture is that each component plays it's own separate role in making the application function as needed.

```
var buildPhotoView = function( photoModel, photoController ){
    var base        = document.createElement('div'),
        photoEl     = document.createElement('div');
     base.appendChild(photoEl);
     var render= function(){
        // We use a templating library such as Underscore
        // templating which generates the HTML for our
        // photo entry
        photoEl.innerHTML = _.template('photoTemplate', {src: photoModel.getSrc()
});
     }
     photoModel.addSubscriber( render );
     photoEl.addEventListener('click', function(){
        photoController.handleEvent('click', photoModel );
     });
     var show = function(){
        photoEl.style.display  = '';
```

```
    }
    var hide = function(){
       photoEl.style.display  = 'none';
    }
    return{
       showView: show,
       hideView: hide
    }
}
```

Templating

In the context of JavaScript frameworks that support MVC/MV*, it is worth briefly discussing JavaScript templating and its relationship to views as we briefly touched upon it in the last section.

It has long been considered (and proven) a performance bad practice to manually create large blocks of HTML markup in-memory through string concatenation. Developers doing so have fallen prey to inperformantly iterating through their data, wrapping it in nested divs and using outdated techniques such as `document.write` to inject the 'template' into the DOM. As this typically means keeping scripted markup inline with your standard markup, it can quickly become both difficult to read and more importantly, maintain such disasters, especially when building non-trivially sized applications.

JavaScript templating solutions (such as Handlebars.js and Mustache) are often used to define templates for views as markup (either stored externally or within script tags with a custom type – e.g text/template) containing template variables. Variables may be deliminated using a variable syntax (e.g {{name}}) and frameworks are typically smart enough to accept data in a JSON form (of which model instances can be converted to) such that we only need be concerned with maintaining clean models and clean templates. Most of the grunt work to do with population is taken care of by the framework itself. This has a large number of benefits, particularly when opting to store templates externally as this can give way to templates being dynamically loaded on an as-needed basis when it comes to building larger applications.

Below we can see two examples of HTML templates. One implemented using the popular Handlebars.js framework and another using Underscore's templates.

Handlebars.js:

```
<li class="photo">
  <h2>{{caption}}</h2>
  <img class="source" src="{{src}}"/>
  <div class="meta-data">
    {{metadata}}
```

```
  </div>
</li>
```

Underscore.js Microtemplates:

```
<li class="photo">
  <h2><%= caption %></h2>
  <img class="source" src="<%= src %>"/>
  <div class="meta-data">
    <%= metadata %>
  </div>
</li>
```

It is also worth noting that in classical web development, navigating between independent views required the use of a page refresh. In Single-page JavaScript applications however, once data is fetched from a server via Ajax, it can simply be dynamically rendered in a new view within the same page without any such refresh being necessary. The role of navigation thus falls to a "router", which assists in managing application state (e.g allowing users to bookmark a particular view they have navigated to). As routers are however neither a part of MVC nor present in every MVC-like framework, I will not be going into them in greater detail in this section.

To summarize, views are a visual representation of our application data.

Controllers

Controllers are an intermediary between models and views which are classically responsible for two tasks: they both update the view when the model changes and update the model when the user manipulates the view.

In our photo gallery application, a controller would be responsible for handling changes the user made to the edit view for a particular photo, updating a specific photo model when a user has finished editing.

In terms of where most JavaScript MVC frameworks detract from what is conventionally considered "MVC" however, it is with controllers. The reasons for this vary, but in my honest opinion it is that framework authors initially look at the server-side interpretation of MVC, realize that it doesn't translate 1:1 on the client-side and re-interpret the C in MVC to mean something they feel makes more sense. The issue with this however is that it is subjective, increases the complexity in both understanding the classical MVC pattern and of course the role of controllers in modern frameworks.

As an example, let's briefly review the architecture of the popular architectural framework Backbone.js. Backbone contains models and views (somewhat similar to what we reviewed earlier), however it doesn't actually have true controllers. Its views and routers act a little

similar to a controller, but neither are actually controllers on their own.

In this respect, contrary to what might be mentioned in the official documentation or in blog posts, Backbone is neither a truly MVC/MVP nor MVVM framework. It's in fact better to consider it a member of the MV* family which approaches architecture in its own way. There is of course nothing wrong with this, but it is important to distinguish between classical MVC and MV* should you be relying on advice from classical literature on the former to help with the latter.

Controllers in Spine.vs vs Backbone.js

Spine.js

We now know that controllers are traditionally responsible for updating the view when the model changes (and similarly the model when the user updates the view). As the framework we'll be discussing in this book (Backbone) doesn't have it's **own** explicit controllers, it can be useful for us to review the controller from another MVC framework to appreciate the difference in implementations. For this, let's take a look at a sample controller from Spine.js:

In this example, we're going to have a controller called `PhotosController` which will be in charge of individual photos in the application. It will ensure that when the view updates (e.g a user editd the photo meta-data) the corresonding model does too.

Note: We won't be delving heavily into Spine.js at all, but will just take a ten-foot view of what it's controllers can do:

```
// Controllers in Spine are created by inheriting from Spine.Controller
var PhotosController = Spine.Controller.sub({
  init: function(){
    this.item.bind("update", this.proxy(this.render));
    this.item.bind("destroy", this.proxy(this.remove));
  },
  render: function(){
    // Handle templating
    this.replace($("#photoTemplate").tmpl(this.item));
    return this;
  },
  remove: function(){
    this.el.remove();
    this.release();
  }
});
```

In Spine, controllers are considered the glue for an application, adding and responding to DOM events, rendering templates and ensuring that views and models are kept in sync (which makes sense in the context of what we know to be a controller).

What we're doing in the above example is setting up listeners in the `update` and `destroy` events using `render()` and `remove()`. When a photo entry gets updated , we re-render the view to reflect the changes to the meta-data. Similarly, if the photo gets deleted from the gallery, we remove it from the view. In case you were wondering about the `tmpl()` function in the code snippet: in the `render()` function, we're using this to render a JavaScript template called #photoTemplate which simply returns a HTML string used to replace the controller's current element.

What this provides us with is a very lightweight, simple way to manage changes between the model and the view.

Backbone.js

Later on in this section we're going to revisit the differences between Backbone and traditional MVC, but for now let's focus on controllers.

In Backbone, one shares the responsibility of a controller with both the Backbone.View and Backbone.Router. Some time ago Backbone did once come with it's own Backbone.Controller, but as the naming for this component didn't make sense for the context in which it was being used, it was later renamed to Router.

Routers handle a little more of the controller responsibility as it's possible to bind the events there for models and have your view respond to DOM events and rendering. As Tim Branyen (another Bocoup-based Backbone contributor) has also previously pointed out, it's possible to get away with not needing Backbone.Router at all for this, so a way to think about it using the Router paradigm is probably:

```
var PhotoRouter = Backbone.Router.extend({
  routes: { "photos/:id": "route" },
  route: function(id) {
    var item = photoCollection.get(id);
    var view = new PhotoView({ model: item });
    something.html( view.render().el );
  }
}):
```

To summarize, the takeaway from this section is that controllers manage the logic and coordination between models and views in an application.

What does MVC give us?

This separation of concerns in MVC facilitates simpler modularization of an application's functionality and enables:

- Easier overall maintenance. When updates need to be made to the application it is very clear whether the changes are data-centric, meaning changes to models and possibly controllers, or merely visual, meaning changes to views.
- Decoupling models and views means that it is significantly more straight-forward to write unit tests for business logic
- Duplication of low-level model and controller code (i.e what you may have been using instead) is eliminated across the application
- Depending on the size of the application and separation of roles, this modularity allows developers responsible for core logic and developers working on the user-interfaces to work simultaneously

Delving deeper

Right now, you likely have a basic understanding of what the MVC pattern provides, but for the curious, we can explore it a little further.

The GoF (Gang of Four) do not refer to MVC as a design pattern, but rather consider it a "set of classes to build a user interface". In their view, it's actually a variation of three other classical design patterns: the Observer (Pub/Sub), Strategy and Composite patterns. Depending on how MVC has been implemented in a framework, it may also use the Factory and Decorator patterns. I've covered some of these patterns in my other free book, JavaScript Design Patterns For Beginners if you would like to read into them further.

As we've discussed, models represent application data whilst views are what the user is presented on screen. As such, MVC relies on Pub/Sub for some of its core communication (something that surprisingly isn't cover in many articles about the MVC pattern). When a model is changed it notifies the rest of the application it has been updated. The controller then updates the view accordingly. The observer nature of this relationship is what facilitates multiple views being attached to the same model.

For developers interested in knowing more about the decoupled nature of MVC (once again, depending on the implement), one of the goal's of the pattern is to help define one-to-many relationships between a topic and its observers. When a topic changes, its observers are updated. Views and controllers have a slightly different relationship. Controllers facilitate views to respond to different user input and are an example of the Strategy pattern.

Summary

Having reviewed the classical MVC pattern, we should now understand how it allows us to cleanly separate concerns in an application. We should also now appreciate how JavaScript MVC frameworks may differ in their interpretation of the MVC pattern, which

although quite open to variation, still shares some of the fundamental concepts the original pattern has to offer.

When reviewing a new JavaScript MVC/MV* framework, remember – it can be useful to step back and review how it's opted to approach architecture (specifically, how it supports implementing models, views, controllers or other alternatives) as this can better help you grok how the framework expects to be used.

MVP

Model-view-presenter (MVP) is a derivative of the MVC design pattern which focuses on improving presentation logic. It originated at a company named Taligent in the early 1990s while they were working on a model for a C++ CommonPoint environment. Whilst both MVC and MVP target the separation of concerns across multiple components, there are some fundamental differences between them.

For the purposes of this summary we will focus on the version of MVP most suitable for web-based architectures.

Models, Views & Presenters

The P in MVP stands for presenter. It's a component which contains the user-interface business logic for the view. Unlike MVC, invocations from the view are delegated to the presenter, which are decoupled from the view and instead talk to it through an interface. This allows for all kinds of useful things such as being able to mock views in unit tests.

The most common implementation of MVP is one which uses a Passive View (a view which is for all intents and purposes "dumb"), containing little to no logic. MVP models are almost identical to MVC models and handle application data. The presenter acts as a mediator which talks to both the view and model, however both of these are isolated from each other. They effectively bind models to views, a responsibility which was previously held by controllers in MVC. Presenters are at the heart of the MVP pattern and as you can guess, incorporate the presentation logic behind views.

Solicited by a view, presenters perform any work to do with user requests and pass data back to them. In this respect, they retrieve data, manipulate it and determine how the data should be displayed in the view. In some implementations, the presenter also interacts with a service layer to persist data (models). Models may trigger events but it's the presenters role to subscribe to them so that it can update the view. In this passive architecture, we have no concept of direct data binding. Views expose setters which presenters can use to set data.

The benefit of this change from MVC is that it increases the testability of your application and provides a more clean separation between the view and the model. This isn't however without its costs as the lack of data binding support in the pattern can often mean having to take care of this task separately.

Although a common implementation of a Passive View is for the view to implement an interface, there are variations on it, including the use of events which can decouple the View from the Presenter a little more. As we don't have the interface construct in JavaScript, we're using more a protocol than an explicit interface here. It's technically still an API and it's probably fair for us to refer to it as an interface from that perspective.

There is also a Supervising Controller variation of MVP, which is closer to the MVC and MVVM patterns as it provides data-binding from the Model directly from the View. Key-value observing (KVO) plugins (such as Derick Bailey's Backbone.ModelBinding plugin) tend to bring Backbone out of the Passive View and more into the Supervising Controller or MVVM variations.

MVP or MVC?

MVP is generally used most often in enterprise-level applications where it's necessary to reuse as much presentation logic as possible. Applications with very complex views and a great deal of user interaction may find that MVC doesn't quite fit the bill here as solving this problem may mean heavily relying on multiple controllers. In MVP, all of this complex logic can be encapsulated in a presenter, which can simplify maintenance greatly.

As MVP views are defined through an interface and the interface is technically the only point of contact between the system and the view (other than a presenter), this pattern also allows developers to write presentation logic without needing to wait for designers to produce layouts and graphics for the application.

Depending on the implementation, MVP may be more easy to automatically unit test than MVC. The reason often cited for this is that the presenter can be used as a complete mock of the user-interface and so it can be unit tested independent of other components. In my experience this really depends on the languages you are implementing MVP in (there's quite a difference between opting for MVP for a JavaScript project over one for say, ASP.net).

At the end of the day, the underlying concerns you may have with MVC will likely hold true for MVP given that the differences between them are mainly semantic. As long as you are cleanly separating concerns into models, views and controllers (or presenters) you should be achieving most of the same benefits regardless of the pattern you opt for.

MVC, MVP and Backbone.js

There are very few, if any architectural JavaScript frameworks that claim to implement the MVC or MVC patterns in their classical form as many JavaScript developers don't view MVC and MVP as being mutually exclusive (we are actually more likely to see MVP strictly implemented when looking at web frameworks such as ASP.net or GWT). This is because it's possible to have additional presenter/view logic in your application and yet still consider it a flavor of MVC.

Backbone contributor Irene Ros (of Boston-based Bocoup) subscribes to this way of thinking as when she separates views out into their own distinct components, she needs something to actually assemble them for her. This could either be a controller route (such as a `Backbone.Router`, covered later in the book) or a callback in response to data being fetched.

That said, some developers do however feel that Backbone.js better fits the description of MVP than it does MVC . Their view is that:

- The presenter in MVP better describes the `Backbone.View` (the layer between View templates and the data bound to it) than a controller does
- The model fits `Backbone.Model` (it isn't greatly different to the models in MVC at all)
- The views best represent templates (e.g Handlebars/Mustache markup templates)

A response to this could be that the view can also just be a View (as per MVC) because Backbone is flexible enough to let it be used for multiple purposes. The V in MVC and the P in MVP can both be accomplished by `Backbone.View` because they're able to achieve two purposes: both rendering atomic components and assembling those components rendered by other views.

We've also seen that in Backbone the responsibility of a controller is shared with both the Backbone.View and Backbone.Router and in the following example we can actually see that aspects of that are certainly true.

Our Backbone `PhotoView` uses the Observer pattern to 'subscribe' to changes to a View's model in the line `this.model.bind('change',...)`. It also handles templating in the `render()` method, but unlike some other implementations, user interaction is also handled in the View (see `events`).

```
var PhotoView = Backbone.View.extend({
    //... is a list tag.
    tagName:  "li",
    // Pass the contents of the photo template through a templating
    // function, cache it for a single photo
    template: _.template($('#photo-template').html()),
    // The DOM events specific to an item.
    events: {
      "click img" : "toggleViewed"
```

```
    },
    // The PhotoView listens for changes to
    // its model, re-rendering. Since there's
    // a one-to-one correspondence between a
    // **Photo** and a **PhotoView** in this
    // app, we set a direct reference on the model for convenience.
    initialize: function() {
      _.bindAll(this, 'render');
      this.model.bind('change', this.render);
      this.model.bind('destroy', this.remove);
    },
    // Re-render the photo entry
    render: function() {
      $(this.el).html(this.template(this.model.toJSON()));
      return this;
    },
    // Toggle the `"viewed"` state of the model.
    toggleViewed: function() {
      this.model.viewed();
    }
});
```

Another (quite different) opinion is that Backbone more closely resembles Smalltalk-80 MVC, which we went through earlier.

As regular Backbone user Derick Bailey has previously put it, it's ultimately best not to force Backbone to fit any specific design patterns. Design patterns should be considered flexible guides to how applications may be structured and in this respect, Backbone fits neither MVC nor MVP. Instead, it borrows some of the best concepts from multiple architectural patterns and creates a flexible framework that just works well.

It *is* however worth understanding where and why these concepts originated, so I hope that my explanations of MVC and MVP have been of help. Call it **the Backbone way**, MV* or whatever helps reference its flavor of application architecture. Most structural JavaScript frameworks will adopt their own take on classical patterns, either intentionally or by accident, but the important thing is that they help us develop applications which are organized, clean and can be easily maintained.

Fast facts

Backbone.js

- Core components: Model, View, Collection, Router. It enforces its own flavor of MV*
- More complete documentation than some frameworks (at the time of writing e.g Ember.js) with more improvements on the way
- Used by large companies such as SoundCloud and Foursquare to build non-trivial applications

- Event-driven communication between views and models means more complete control over what's happening. It's relatively straight-forward to add event listeners to any attribute in a model meaning more control over what changes in the view
- Supports data bindings through manual events or a separate Key-value observing (KVO) library
- Great support for RESTful interfaces out of the box so models can be easily tied to a backend
- Extensive eventing system. It's trivial to add support for pub/sub in Backbone
- Prototypes are instantiated with the `new` keyword which some enjoy
- No default templating engine, however Underscore's micro-templating is often assumed for this. Also works well with solutions like Handlebars if you drop them in
- Doesn't support deeply nested models out of the box, but like many common problems there exist Backbone plugins such as this which can easily help
- Clear and flexible conventions for structuring applications. Backbone doesn't force usage of all of its components and can work with only those needed.

To continue reading, see Backbone Fundamentals.

References include Comparison of architecture patterns and will be documented in full in Backbone Fundamentals

---

Original URL:
http://addyosmani.com/blog/understanding-mvc-and-mvp-for-javascript-and-backbone-developers/