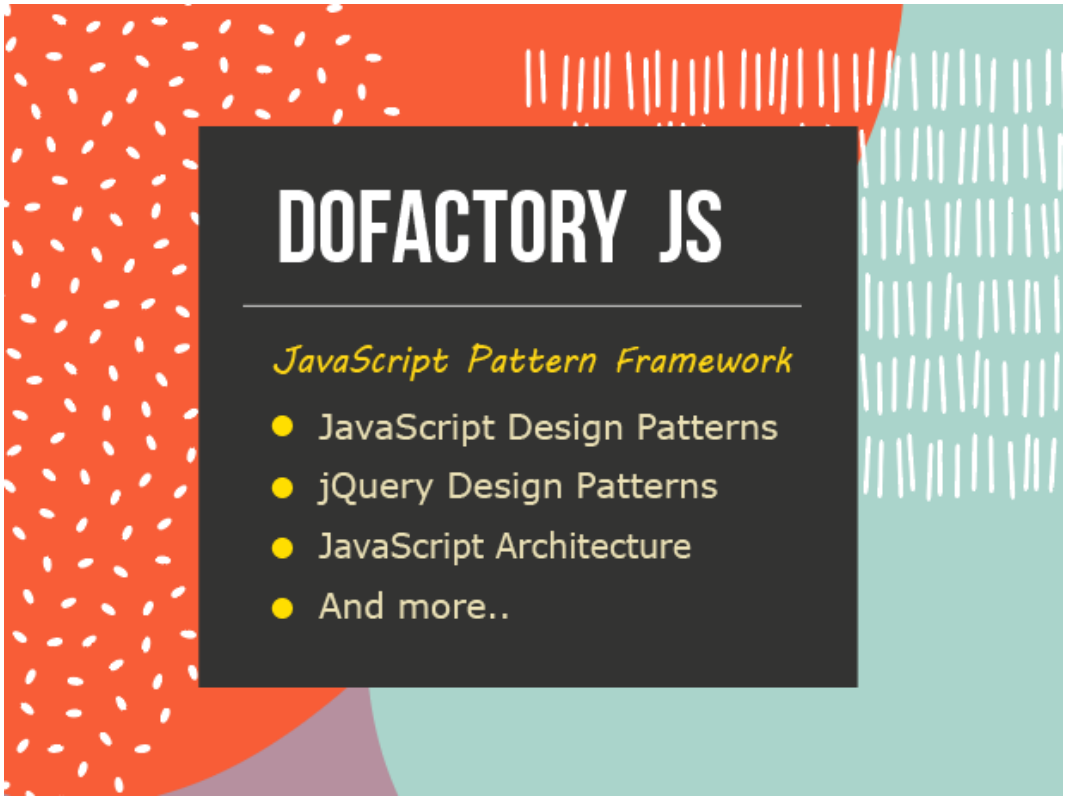
Dofactory JS 6.0

Model View JavaScript Patterns



by

Data & Object Factory, LLC

www.dofactory.com

5. Model View JavaScript Patterns

Index

Introduction

Suppose you are in charge of a small team that starts development on a new web application. Your initial thoughts are that on the client side all you need is a DOM manipulation library, such as jQuery, and perhaps a couple utility-type plugins and UI controls. With that you start crafting your app.

However, a few weeks into the project you realize that many pages in the app are becoming rather unwieldy and unstructured with dozens of event handlers and long reams of DOM manipulating JavaScript. You feel that you could use some extra help in getting your programs better organized. This is exactly what a Model-View (MV) Framework can do: to bring additional structure to your project.

What exactly is an MV Framework? An MV Framework is a JavaScript library that allows developers to organize their application in logical components to bring clarity and structure into their programs. These components represent 3 areas of concern which are called: Model, View and Controller. Together they make up the MVC design pattern.

The MVC pattern is one of a family of three related patterns which collectively are referred to as Model View (MV*) Patterns. The other two are refinements of MVC; they are MVP (Model View Presenter) and MVVM (Model View ViewModel). We will first review MVC, followed by MVP and MVVM. Finally, we provide a review of some popular open source MV Frameworks.

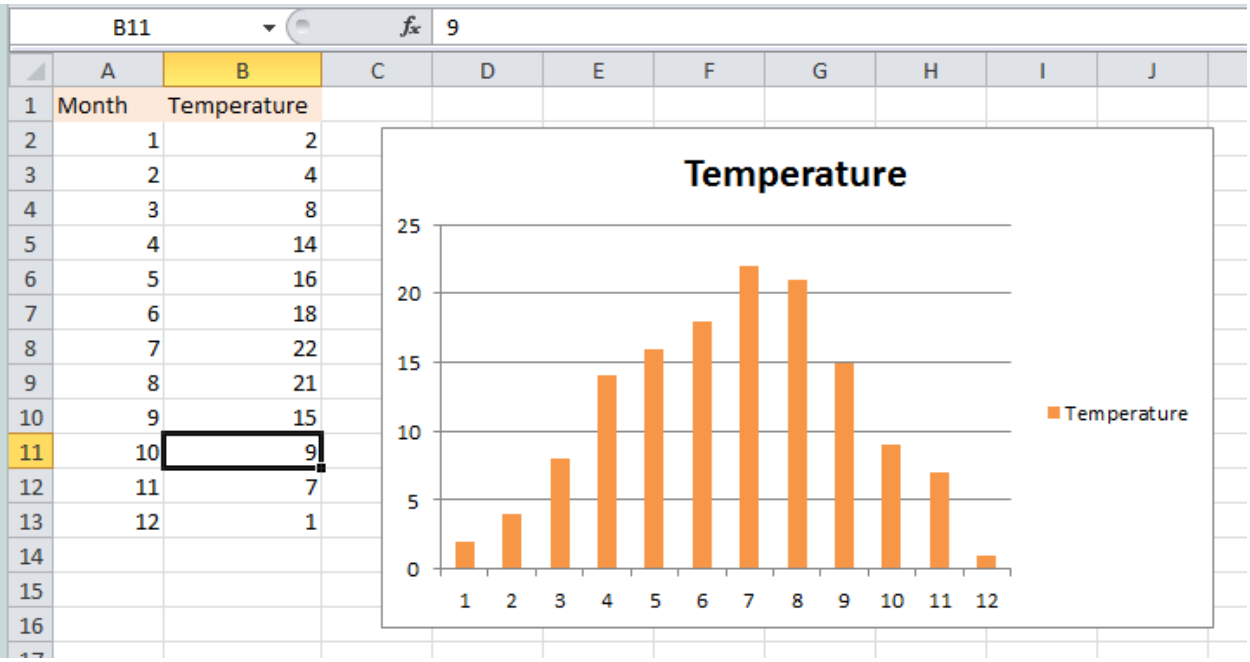
MVC

The Model View Controller (MVC) design pattern is one of the earliest documented patterns. The pattern was originally described in 1979 by Trygve Reenskaug, then working on Smalltalk.

MVC, as its name implies, comprises three parts: Model, View and Controller. Each one has its own role and responsibility (area of concern). This is referred to as 'separation of concerns', in which each part is responsible for a clearly defined task.

The MVC pattern is best explained with an application like Microsoft Excel. Understanding this will greatly help you understand the core principles behind MVC.

Say, you have a comma-separated file (CSV) with average monthly temperatures for Paris, France during the last year. You open the file in Excel and see two columns of data (month and temperature in °C) in your spreadsheet. Then you decide to create a column chart and show the temperature changes over time. You select all 12 observations and place a chart on the spreadsheet. It looks good and you are pleased with the results (see below).



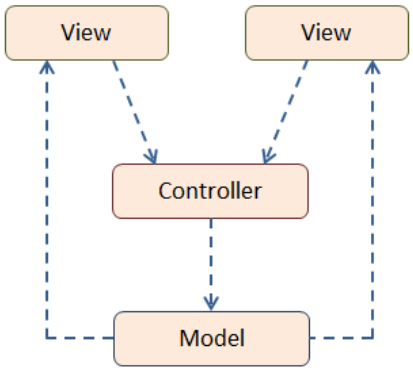
Month	Temperature
1	2
2	4
3	8
4	14
5	16
6	18
7	22
8	21
9	15
10	9
11	7
12	1

What is at work here is the MVC pattern. The data file with the temperature readings is the Model. The Grid (spreadsheet) and the Chart are two separate instances of a View. These Views represent two different ways of looking at the same data (the Model).

If the underlying data in the Model changes, we expect that **both** the Grid and the Chart to reflect this change. For example, if you change the temperature for October from 9 °C to 14 °C in the CSV file (Model) you expect to see this change in both the Grid and the Chart. And that is indeed what happens. The synchronization between the Views and the Model is the role of the Controller.

The Controller coordinates changes between the Model and one or more Views. Similarly, if a user changes the values in the Grid, you would expect the Chart to reflect

this immediately (as well as the Model data if you choose to save the file). Again, it is the Controller that is notified of the change, which in turn changes the Model, which then notifies the Views to update themselves with the new model values. Below is a diagram of the Model, View and Controller parts and the dataflow among them.

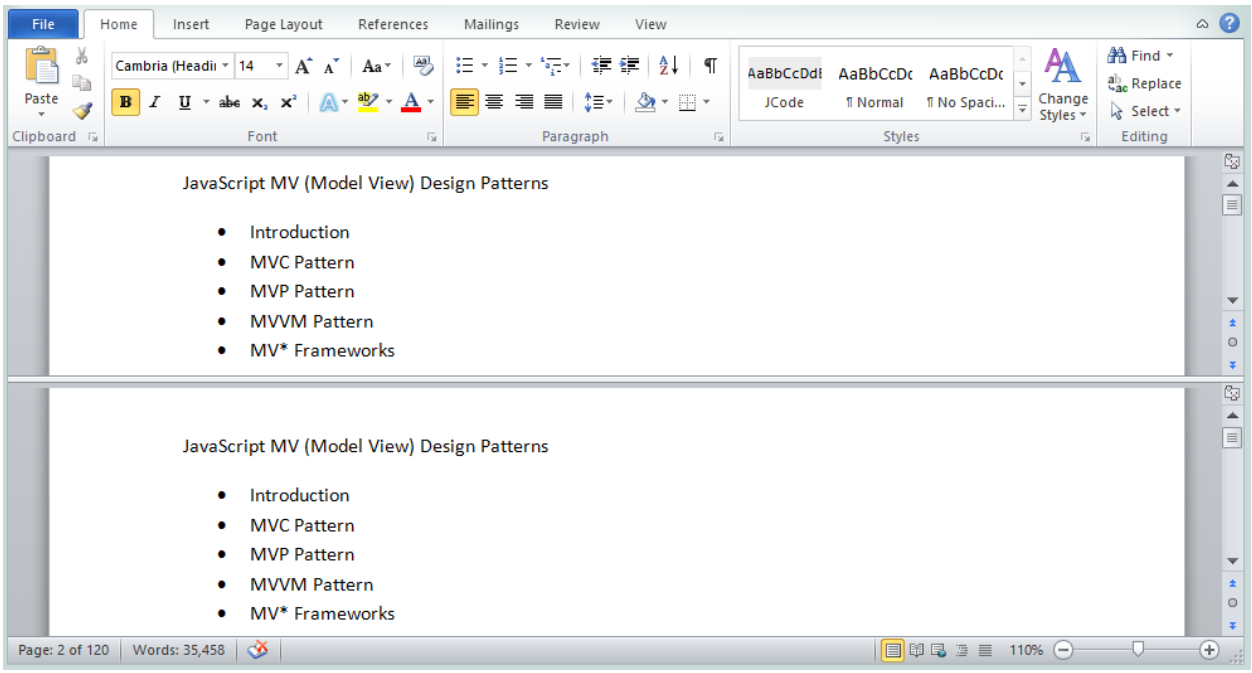


The dashed blue lines represent the data flow, which if you look closely is circular. It starts at the View where the user takes an action. The Controller is notified which then updates the Model. Finally, the Model changes are reported back to the View. The flow between the Controller and the Model is via method invocations, everything else is via events and event handlers.

Let's see how the data flow works in Excel. Suppose the user makes a data change in the Grid (the spreadsheet, which is a View). This triggers an event of which the Controller is notified. The Controller gets the changed data item and applies the same change to the Model. The Model then triggers another event of which *all* Views are notified. The Views get the data from the Model and change their displays accordingly.

The partitioning of the app using MVC helps developers to better organize a complex application like Excel. Long term it makes the application much easier to maintain and more extensible. For example, if a new Model needs to be supported (for example, a web service) or another View is introduced (for example, a report) then this will fit very easily into the existing structure.

As an aside: while preparing this document we used Microsoft Word. Word allows you to split the document which creates a second View (see below). When typing in one View, it immediately shows up in the other. Again, this is MVC in action.



Outside of desktop applications, MVC is also prominent on web servers with web development environments like ASP.NET MVC, Java/Spring MVC, PHP/CakePHP, and Ruby on Rails. More recently, MVC has also made big inroads on the client (browser) using JavaScript which is the topic of the next section.

MVC on the client

Let's recap the major parts of MVC and explore what it means to client-side web development:

Model: the model represents the domain specific data. These are business objects that are getting their data from a file or a database. For example, in an e-commerce app you may have model objects like Customer, Cart or Product. A Model notifies the page (the View) of any state changes that take place. So, if a special price of a Product changes, the View receives a data change event.

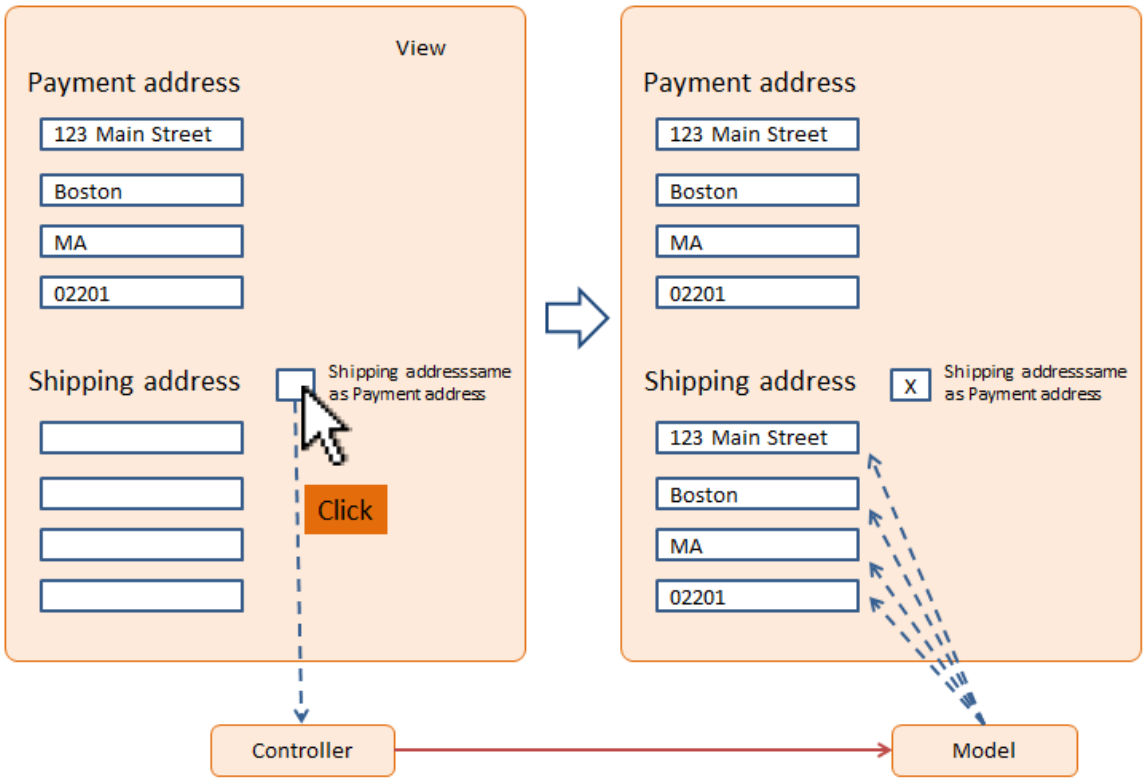
View: the view represents the user interface. In web apps this includes HTML elements and any templating you may have. A view receives data change events from the model and changes the DOM elements on the page accordingly. Say a new item is added to a shopping cart. The Cart model is updated with a new line item including product, quantity and cost and notifies the View. The View, in turn, will add a new row to a table

with the relevant product information. Views also communicate with the Controller. For example, when a user clicks a button to delete an item from the shopping cart, the Controller gets notified.

Controller: the controller coordinates changes between the Model and the View. It receives events from the View, such as click, change, and drag & drop events. It responds to these events by updating the Model. The Model in turn notifies the View of a data change.

Let's look at MVC in a real-world web scenario: an online credit card payment page. A customer enters credit card information and then clicks a checkbox to inform that the payment address is the same as the shipping address. The effect is that the 4 payment address fields get copied over to the shipping address fields, so the user will not have to type these in again.

In the figure below, left is the pre-click status of the page. To the right is what the page looks like after the user has clicked the checkbox.



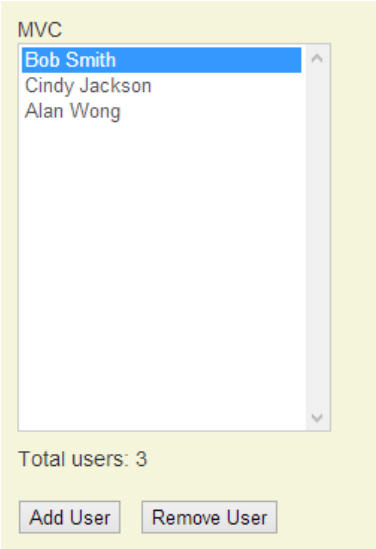
Here is what happens under the hood: when the user clicks the checkbox, the Controller gets notified of a click event. The Controller then retrieves all payment address values from the Model and copies these back into the Model's shipping address properties. The Model notifies the View of these changes and the View updates all shipping text fields with data from the Model.

The Controller is the intermediary between the page and the data, that is, the View and the Model. The typical flow in MVC is: View → Controller → Model → View. The user takes action on the View and the View notifies the Controller. The Controller makes a decision and takes action which frequently involves changing the Model. Then the Model notifies the View of its state change and the View refreshes accordingly. In this system, each component has a clearly defined role.

Next, we will explore how to build a web page from scratch using MVC in JavaScript.

Implementing MVC in JavaScript

We will use a simple web page which allows us to better focus on MVC. Our page has a list control with user names, and two buttons underneath; one to add a new user, and one to delete the currently selected user. We also display the total number of users in the list which is updated when the list changes. Here is what the page looks like:



Let's look at the HTML:

```
<div>
    MVC<br />
    <select id="nameList" size="15"></select>
    <div id="nameCount">Total users: 0</div>
    <button id="addButton">Add User</button>
    <button id="removeButton">Remove User</button>
</div>
```

To avoid clutter we removed style and class attributes.

Now let's explore the MVC JavaScript code. First we look at the Model:

```
var Model = function (names) {
    this.names = names;

    this.nameAdded = new Observer();
    this.nameRemoved = new Observer();
};

Model.prototype = {

    add: function (name) {
        this.names.push(name);
        this.nameAdded.notify(this.names);
    },

    remove: function (index) {
        this.names.splice(index, 1);
        this.nameRemoved.notify(this.names);
    },

    getNames: function () {
        return this.names;
    }

};


var Observer = function () {
    this.observers = [];
};

Observer.prototype = {
```

```
    attach: function (callback) {
        this.observers.push(callback);
    },

    notify: function (n) {
        for (var i = 0, len = this.observers.length; i < len; i++) {
            this.observers[i](n);
        }
    }
};
```

Remember that the Model is the domain or business object. It contains the data. The Model constructor function accepts an array of names which is assigned to a `names` property. The Model has two other properties: `nameAdded` and `nameRemoved`. These are observer objects that will notify any event handlers (callbacks) of any changes that take place in the `names` array. We will get back to the Observers shortly.

The Model's `add` method adds a new name to the list. Any callback that is registered with the `nameAdded` observer will be notified. Similarly, the Model's `remove` method removes a name from the list by index. It notifies any callback that is registered with the `nameRemoved` observer. Notice that the array of names is passed to either notify calls. This will allow the View, which is listening to Model changes, to refresh the list control with the entire data set.

Let's look at the Observer helper object, which by the way is an implementation of the GoF Observer design pattern. The observers maintain a list of callbacks that are notified when an event occurs. Callback methods register themselves with the `attach` method. The `notify` method triggers (executes) all registered callback methods while passing an event argument, named n, which in the case is a list of names.

So, the Model maintains the business data and notifies anyone that is interested of any data change that takes place in the list of names. We will see next that the View (the page) will be notified of these changes.

Here is the code for the View:

```
var View = function (elements) {
```

```
    this.elements = elements;
};

View.prototype = {

    init: function (model, controller) {

        var self = this;

        model.nameAdded.attach(function (n) {
            self.refresh(n);
        });

        model.nameRemoved.attach(function (n) {
            self.refresh(n);
        });

        this.elements.addButton.click(function () {
            var name = prompt('Add a new user name: ', '');
            if (name) controller.addName(name);
        });

        this.elements.removeButton.click(function () {
            var index = self.elements.nameList.get(0).selectedIndex;
            if (index != -1) {
                controller.removeName(index);
            } else {
                alert("No user was selected");
            }
        });

        this.refresh(model.getNames());
    },

    refresh: function (names) {

        this.elements.nameList.html('');

        for (var i = 0, len = names.length; i < len; i++)
            this.elements.nameList.append('<option>' + names[i] + '</option>');

        this.elements.nameCount.text("Total users: " + len);
    }
};
```

The View represents the user interface. Its constructor function accepts a list of DOM elements which it will interact with. They are stored in a property named elements. These elements are jQuery elements which makes modifying them very easy. The View has two methods: `init` and `refresh`.

The `init` method is called only once. Two arguments are passed: `model` and `controller`. It is in this method that all callback relationships and actions are established. In init the `this` value is assigned to a local variable called `self`. The 'self' variable is kept in a closure together with the arguments (model and controller) which ensures that all nested method bodies have access to the correct objects. Note that we could have also used `bind(this)`.

In `init` two callback functions are attached to the Model's `nameAdded` and `nameRemoved` events. Whenever the Model triggers these events the list on the page gets refreshed with the new set of names. The parameter `n` is the list of names which is passed on to the refresh method.

Next we are attaching click handlers to the 'add' and 'remove' buttons. The `add` handler receives a new user name and passes the information on to the Controller. The `remove` handlers checks which name is currently selected and passes the index on to the Controller.

The last line in `init` is a call to `refresh` with the current set of names. This call initializes the page with the starting set of names (there will be 3).

The View's `refresh` method accepts an array of names. It clears the list and then appends each name to the list control. As a final step the total number of users in the list is updated.

The Controller in a typical MVC implementation is usually fairly lightweight; other than some business decisions it frequently is a pass-through to the Model. Here is the Controller for our example:

```
var Controller = function (model) {
    this.model = model;
};

Controller.prototype = {
```

```
    addName: function (name) {
        this.model.add(name);
    },

    removeName: function (index) {
        this.model.remove(index);
    }
};
```

The Controller constructor function accepts the model and keeps a reference in a property named model. The two methods `addName` and `removeName` are accessed only from the View. The `addName` accepts a name parameter which is passed on to the Model's `add` method. Similarly, the `removeName` method accepts an `index` and passes it on to the Model's `remove` method.

Now that we have the fundamental MVC building blocks let's see how the program wires all this together. Here is the code that executes when the page is loaded.

```
$(function () {

    var model = new Model(['Bob Smith', 'Cindy Jackson', 'Alan Wong']);

    var controller = new Controller(model);

    var elements = {
        nameList: $('#nameList'),
        nameCount: $("#nameCount"),
        addButton: $('#addButton'),
        removeButton: $('#removeButton')
    };

    var view = new View(elements);

    view.init(model, controller);
});
```

A Model instance is created and initialized with 3 user names. Then a Controller is created with the model as its argument. Next, an object is created with jQuery elements that hold references to the 4 DOM elements we're interested in on this page. This is

assigned to the elements variable. A View instance is created that gets passed the elements object. All three components, i.e. Model, View and Controller, are wired together by a call to `view.init` which establishes the event handling and callback relationships. This call also refreshes the page which will render the 3 initial names in the Model.

You can now add and remove users; we have a full implementation of the MVC pattern in JavaScript.

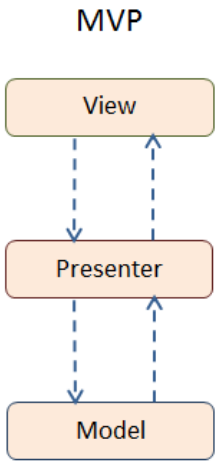
Next we will explore and develop the MVP pattern.

MVP

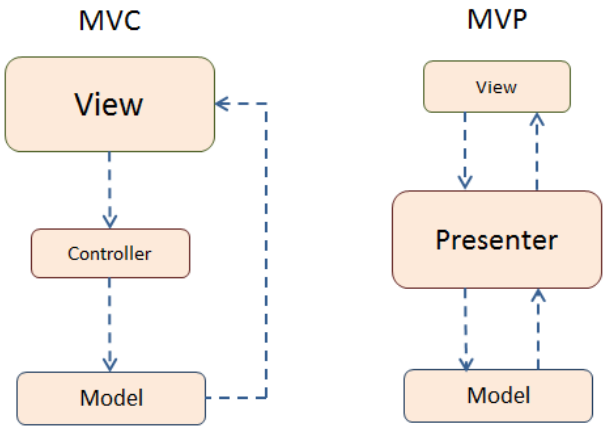
MVP is a variation of the MVC pattern. In MVP the Controller has been renamed Presenter because it takes on a much greater role. The Presenter makes decisions how to respond to events it receives from the View, but, more importantly, it can update the View directly. Hence its name 'Presenter': it presents the data and changes the presentation.

In MVP the role of the View is limited to notifying the Presenter of events that happen on the page, such as button clicks or data entry. The View does not have any business logic which is now the responsibility of the Presenter.

The Presenter communicates with the Model by updating its data. The Model does not notify the View but instead may notify the Presenter of any model changes. Below is a diagram of MVP and its data flow. We see that the Presenter sits in between the View and the Model and all communication must flow through the Presenter. In MVP there is no direct communication between the View and the Model.



To highlight the differences between MVC and MVP we show them side by side in a diagram. The size of the component boxes is a measure of their responsibilities, that is, the larger the box, the larger its role and responsibility. Notice the differences in data flow: in MVC it is circular and in MVP it is up and down the components.



When developing complex applications, MVP tends to be a more natural pattern than MVC. Developers like MVP because the UI is usually not the best place to implement business logic; it is better to maintain that at a central location.

Furthermore, MVP forces you to think in terms of reusability which improves the modeling and coding process. For example, when dealing with many pages, each with their own validation logic, it can be very difficult to find candidates for reuse. MVP helps in structuring you app from the onset.

Another benefit of MVP is testability: compared to MVC, MVP is much easier to test. User Interfaces are notorious difficult to test and having a relatively small and more passive view makes it easy to replace it with a Mock object and perform unit- or system-testing.

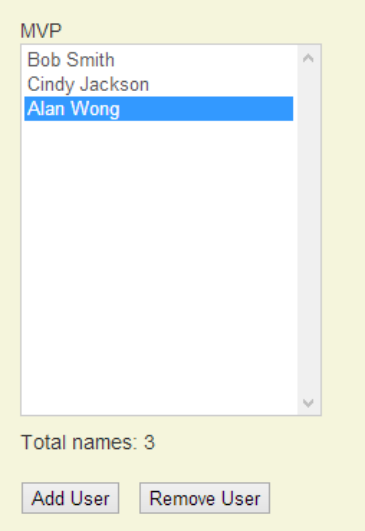
Next, we'll examine how to code MVP in JavaScript.

Implementing MVP in JavaScript

We will use the same page as MVC. This will allow us to focus on the pattern itself. Here is the HTML markup:

```
<div>
    MVP<br />
    <select id="nameList" size="15"></select>
    <div id="nameCount">Total users: 0</div>
    <button id="addButton">Add User</button>
    <button id="removeButton">Remove User</button>
</div>
```

It has a list control with user names and two buttons; one to add a new user, and one to delete the currently selected user. Also included is a div element that displays the total number of users. As a reminder, here is what the page looks like:



Let's now move on to the code starting with the Model and the Observer helper object.

```
var Model = function (names) {
    this.names = names;

    this.nameAdded = new Observer();
    this.nameRemoved = new Observer();
};

Model.prototype = {

    add: function (name) {
        this.names.push(name);
        this.nameAdded.notify(this.names);
    },

    remove: function (index) {
        this.names.splice(index, 1);
        this.nameRemoved.notify(this.names);
    },

    getNames: function () {
        return this.names;
    }

};


var Observer = function () {
    this.observers = [];
};

Observer.prototype = {

    attach: function (callback) {
        this.observers.push(callback);
    },

    notify: function (n) {
        for (var i = 0, len = this.observers.length; i < len; i++) {
            this.observers[i](n);
        }
    }
};
```

These are exactly the same as in MVC.

Recall that in MVC the `nameAdded` and `nameRemoved` Observers were used in notifying the View of any data changes. In MVP you will see that the Presenter gets notified instead.

This works well in asynchronous scenarios in which the Model gets updated from a remote server or database and the Observers notify any event handlers in the Presenter of any data changes. The Presenter then makes the business decisions and updates the View accordingly.

Next we will examine the View which is listed below.

```
var View = function (elements) {

    this.elements = elements;
};
```

Yes, that is all that is left of the View. It is an array of DOM elements that the Presenter interacts with. You can already see that this View is far easier to mock in testing scenarios compared with the View we had in MVC.

Let's move on to Presenter where the bulk of the logic resides:

```
var Presenter = function (model, view) {
    this.model = model;
    this.view = view;
    this.index = -1;
};

Presenter.prototype = {

    init: function () {

        var self = this;

        this.view.elements.addButton.click(function () {
            self.addName();
        });
```

```
        this.view.elements.removeButton.click(function () {
            self.removeName();
        });

        this.view.elements.nameList.change(function (e) {
            self.index = e.target.selectedIndex;
        });

        this.model.nameAdded.attach(function (n) {
            self.refresh(n);
        });

        this.model.nameRemoved.attach(function (n) {
            self.refresh(n);
        });


        this.refresh(this.model.getNames());
    },

    addName: function () {
        var name = prompt('Add a new user name: ', '');
        if (name) {
            this.model.add(name);
            this.index = -1;
        }
    },

    removeName: function () {
        if (this.index > -1) {
            this.model.remove(this.index);
            this.index = -1;
        }
        else {
            alert("No name was selected");
        }
    },

    refresh: function (names) {

        this.view.elements.nameList.html('');

        for (var i = 0, len = names.length; i < len; i++)
            this.view.elements.nameList.append('<option>' + names[i] + '</option>');

        this.view.elements.nameCount.text("Total names: " + len);
```

```
    }
};
```

You will recognize most of the logic, much of it from the original View in MVC. The initialization has migrated here, but more interestingly the `refresh` method also resides here. The `refresh` method updates the controls on the page: the View is not involved at all.

Let's go back to the `init` method. In it, the Presenter registers for events that occur on both the View and one the Model. It listens to the View's click events and the list's selection change events as well as the Model's `nameAdded` and `nameRemoved` events. The Presenter is the central hub in MVP.

Finally, let's run the system and review the startup code:

```
var model = new Model(['Bob Smith', 'Cindy Jackson', 'Alan Wong']);

var elements = {
    nameList: $('#nameList'),
    nameCount: $("#nameCount"),
    addButton: $('#addButton'),
    removeButton: $('#removeButton')
};

var view = new View(elements);

var presenter = new Presenter(model, view);
presenter.init();
```

From a separation of concerns' perspective this startup code feels right. First, the Model is created and initialized with three user names. There are no external dependencies. Next, the View is created with just the visual elements that we are interested in. Again, there are no dependencies. Then the Presenter is created with references to both the model and view instances, exactly the way you would expect it to be. The final call to `init` ties all components together and the page is ready to go!

MVP is an elegant pattern: each part has a clear responsibility and they are loosely coupled which is another good thing; it makes our code clean, clear, and highly maintainable.

As an aside: when reading up on MVP you may discover there are really two flavors of this pattern: one is called *Passive View* and the other is *Supervising Controller*. Our model is the Passive View which is the more common approach. The Supervising Controller simply adds bidirectional data flow between the Model and the View, so it a bit closer to MVC.

Whatever the intricate details of these patterns are it turns out that the JavaScript community has taken a lot of creative freedom with these patterns. From the 25 or so popular open source MV Frameworks only a few are easy to categorize as MVC, MVP, or MVVM; the rest is something like MV and 'Anything'. For example, the popular Backbone framework follows MVC, but the C stands for Collections, not Controller. Backbone also has a Router, which looks like a Controller.

As a JavaScript developer the main thing to understand is the separation of concerns, the loose binding, and the role that each component plays within each MV pattern.

Next we'll review the last of the MV patterns: MVVM.

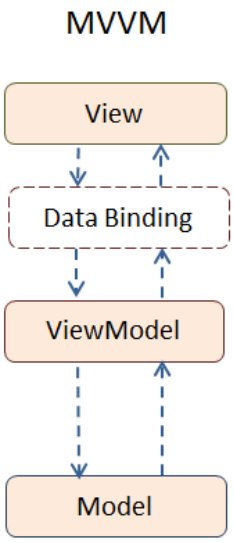
MVVM

MVVM (Model View ViewModel) is a more recent addition to the family of MV patterns. It was first documented by Microsoft in 2005 where it evolved when they began building rich UI applications based on WPF (Windows Presentation Foundation). Several built-in WPF features, including data binding and its commanding architecture, make it highly suitable for MVVM.

In addition to WPF and Silverlight, the MVVM pattern also took hold in the Adobe Flex community. More recently, MVVM is making inroads into the JavaScript community with MVVM frameworks like Knockout, Kendo MVVM, and Knockback. We will demonstrate MVVM with Knockout in a few moments.

In MVVM, the Controller has been renamed ViewModel, meaning that it exposes the relevant data from the Model to the View. The ViewModel provides 'a View into the Model'.

How does MVVM differ from the others? MVVM continues along the same line as MVP in which there is a clear separation between the presentation (View) and the business logic/behavior (ViewModel). The difference is that MVVM uses *data binding* between the presentation and the business logic which automates the communication between the two. The diagram below highlights the role of data binding in MVVM:



Data binding is a technique that binds two data sources together and synchronizes them. Say you have a textbox with a name on your page (the View). In your ViewModel you have a data item that stores the value entered in the textbox. Now, if the user changes the name, the data binding mechanism automatically updates the value in the ViewModel. Similarly, when the data item in the ViewModel changes it is automatically carried over to the page via data binding.

This example is called 2-way binding because it works in both directions. You may also run into 1-way bindings where the change is unidirectional.

Data binding usually involves declarative data binding annotations on the page. We will see examples of this in our implementation example.

Another important point about MVVM is that Views and ViewModels are paired, that is, a ViewModel is custom made for a particular View. It is not reusable. Typically,

ViewModels hold data that is highly tailored for the presentation requirements on the page. For example, you may have a formatted price field, such as a string with a "$145.22" value (ready to display), whereas in the Model, these are unformatted floating point values. So, each page (or partial page) has its own ViewModel.

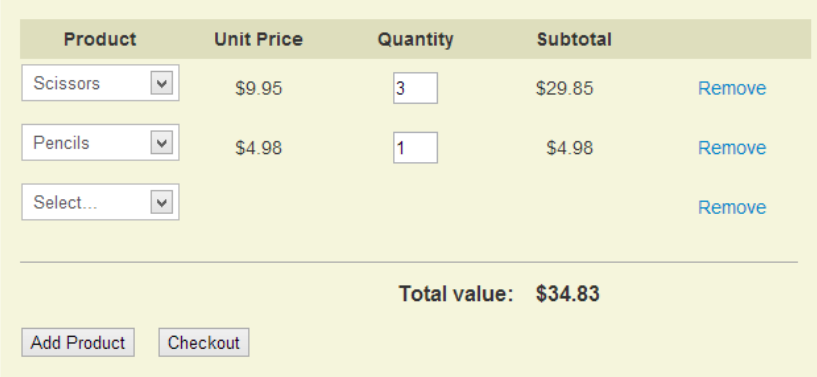
The ViewModel holds the data while the user is working on a page and it gets continually updated on any changes. But only when it gets notified, usually by a save click, will it update the Model which in turn saves the data to a backend database.

MVVM is a very loosely coupled. Unlike the Presenter in the MVP pattern, the ViewModel knows nothing about View or its visual components because this is all handled through data binding.

Implementing MVVM in JavaScript

The necessary plumbing required for data binding and templating is large. This prevents us from building MVVM from scratch. Instead, we will use Knockout, an open-source framework that follows the MVVM paradigm. It will provide you with a good feel for MVVM.

The example is a shopping cart where users can 1) add new products, 2) change quantities, and 3) remove line items. The grand total is maintained during these actions. Here is a screenshot of the page:



Initially the cart will be empty. Clicking the 'Add Product' button will insert a blank row into the cart with dropdown from which a product can be selected. Once selected, the following controls get updated on the same row: the unit price, the quantity, and the subtotal. Also the Total Value on the bottom right will be updated.

Changing the quantity will immediately update the Subtotal and Total Value fields. Clicking on remove will delete the row and the Total Value will be updated as well. Finally, clicking on Checkout will display the data to be sent to the server (in a real app we would first proceed to a checkout/payment page).

Here's the HTML markup -- for simplicity styles and class attributes have been removed.

```
<table>
    <thead>
        <tr>
            <th>Product</th>
            <th>Unit Price</th>
            <th>Quantity</th>
            <th>Subtotal</th>
            <th> </th>
        </tr>
    </thead>
    <tbody data-bind='foreach: items'>
        <tr>
            <td>
                <select data-bind='options: $root.products, optionsText: "name",
                        optionsCaption: "Select...", value: product'> </select>
            </td>
            <td data-bind='with: product'>
                <span data-bind='text: toMoney(price)'> </span>
            </td>
            <td>
                <input data-bind='visible: product,value: quantity,
                                  valueUpdate: "afterkeydown"' />
            </td>
            <td>
                <span data-bind='visible: product, text: toMoney(subtotal())'></span>
            </td>
            <td>
                 <a href='#' data-bind='click: $root.removeItem'>Remove</a>
            </td>
        </tr>
    </tbody>
</table>

<p>
   Total value: <span data-bind='text: toMoney(grandTotal())'> </span>
</p>
```

```
<button data-bind='click: addItem'>Add product</button>
<button data-bind='click: submit'>Checkout</button>
```

It has a <table> that displays the shopping cart, a with the grand total, and two <button>s, one to add a new product, and another to check out the order and complete the transaction. Let's look at this in more detail.

By the way, the purpose of the next few paragraphs is not to get into the details of the Knockout declarative data binding syntax, but rather to give you a flavor of building an app following the MVVM architecture.

The table has a regular header row. Its body contains several declarative data bindings. An important attribute in Knockout is `data-bind`. It has many options but almost always contains a reference to a property on the ViewModel which it will bind to. Note that we will associate (i.e. apply) the ViewModel with the View which we will see shortly when reviewing the JavaScript code.

The table's tbody element has a `data-bind` attribute with `'foreach: items'`. This tells Knockout to iterate over an array on the ViewModel called items. The items are the line items in the shopping cart.

In the first <td> we have a dropdown (<select>) that data-binds to `$root.products`. This is a reference to the root object which is the ViewModel; its `products` property is an array with all products for sale. The user selected value is stored in a property called `product` which is on the current line item (not the ViewModel).

The next <td> displays the unit price for the selected product. The `data-bind = 'with: product'` sets the context, which is the selected product on the line item. It has a `price` property which is bound and formatted on the next line. So, any time a different product is selected, its price changes with it.

The following <td> is where the quantity can be selected. It only is visible when a product is selected. Its default value is 1. The user can change this and the binding immediately changes the value in the ViewModel following each key hit, i.e. `"afterkeydown"`.

The next <td> is a subtotal for the line item. It is a computed value that is only visible when a product is selected. It is bound to a method on the line item object called `subtotal()`. Formatting is handled by a utility function called `toMoney`.

Finally, we have a <td> with a remove link that allows the user to delete the line item from the shopping cart. It is data bound to a `click` event which triggers the `$root.removeItem` method on the ViewModel which removes the current line item.

Below the table the grand total is displayed. It is data bound to `grandtotal()` which is a method on the ViewModel.

Finally, two buttons have their `click` events bound to `addItem` and `submit` respectively, which are methods on the ViewModel.

Next, we'll review the JavaScript code starting with the Model.

```
var Product = function (id, name, price) {
    this.id = id;
    this.name = name;
    this.price = price;
}

var Model = function Model() {
    this.products = [];
    this.products.push(new Product(1, "Paper", 4.95));
    this.products.push(new Product(2, "Scissors", 9.95));
    this.products.push(new Product(3, "Pencils", 4.98));
    this.products.push(new Product(4, "Pens", 19.50));
    this.products.push(new Product(5, "Eraser", 1.50));
    this.products.push(new Product(6, "Folders", 12.95));

    this.getProducts = function () {
        return this.products;
    };
}
```

There is nothing to indicate that this Model is used in the context of a MVVM project. It is a regular Model object, just like we used in MVC and MVP. It contains an array with all products that users can purchase. Each product has an `id`, a `name`, and a `price`. In most real-world situations this data comes straight from a database.

Before moving on to the ViewModel let's first review CartItem which represents a line item in the shopping cart.

```
var CartItem = function () {
    var self = this;

    this.product = ko.observable();
    this.quantity = ko.observable(1);

    this.subtotal = ko.computed(function () {
        return self.product() ?
               self.product().price * parseInt("0" + self.quantity(), 10) : 0;
    });
};
```

It has all the properties you expect to see in a line item on a shopping cart: a `product` (this is an object itself which contains product name and unit price), a `quantity`, and a `subtotal` value for the line.

The `product` is a method, an `observable` method to be precise. The variable `ko` is an instance of the Knockout *binding and dependency tracking engine*. The `observable` method monitors changes in the product and notifies any subscribers that a change has occurred.

The `quantity` is also an `observable` method. Its default value is 1 which is passed as an argument.

The `subtotal` is a computed value. The callback function passed into `ko.computed` computes the subtotal for the line. It automatically re-executes when product or quantity changes occur.

We're now ready to review the ViewModel.

```
var ViewModel = function () {
    var self = this;

    this.products = new Model().getProducts();

    this.items = ko.observableArray([new CartItem()]);
```

```
    this.grandTotal = ko.computed(function () {
        var total = 0;
        $.each(self.items(), function () { total += this.subtotal() })
        return total;
    });


    // Buttons actions
    this.addItem = function () {
        self.items.push(new CartItem())
    };

    this.removeItem = function (item) {
        self.items.remove(item)
    };

    this.submit = function () {
        var data = $.map(self.items(), function (item) {
            return item.product() ? {
                productId: item.product().id,
                quantity: item.quantity()
            } : undefined
        });
        alert("Info sent to server: " + JSON.stringify(data));
    };
};

var toMoney = function (value) {
    return "$" + value.toFixed(2);
};

ko.applyBindings(new ViewModel());
```

The first thing we do in ViewModel is to ensure that the nested functions will have a correct context by assigning the `this` value in a variable called `self` which then gets added to the function's closure (alternatively, we could have used `bind(this)` on all the methods).

The ViewModel assigns the Model's data to variable named `products` which, as you may recall, is data bound to the dropdown in the shopping cart.

The `items` property is an `observableArray` of shopping cart line items. It works like observable but tracks changes to the array itself, such as removal and addition of new items. This is the array that the shopping cart iterates over in the `data-bind = "foreach: items"` attribute in tbody.

The `grandTotal` is a computed dependency value that adds up all subtotals for all line items.

The `addItem` and `removeItem` methods add and remove line items from the shopping cart respectively.

The `submit` method is not involved in data binding. It iterates over the line items and collects the relevant information that can be submitted back to the server in JSON format.

A helper function called `toMoney` converts values to a monetary format. And this is all there is to the ViewModel.

As a final step we need to associate our ViewModel with the View. The last line shows how easy this is: simply call `ko.applyBindings` with a new instance of the ViewModel. This will start the data binding and data dependency tracking which continues throughout the lifetime of the page. The shopping cart is now ready for use.

MV Frameworks

At this point you have a good understanding of the MVC, MVP, and MVVM design patterns and appreciate that they may offer a solid architectural foundation for your JavaScript projects.

To build your own MV framework can be a daunting task. Fortunately, there is no reason to do so. There are many open-source projects available that have been field-tested in high-performance deployments, so all the work is done for you.

Assuming you have decided that you need the additional structure that an MV framework can offer, the only thing left is to choose the right framework for your project.

Unfortunately this is easier said than done because it takes time to do the research and determine which one is most appropriate for you.

When looking for a framework there are some criteria to keep in mind:

- Which MV model -- that is, MVC, MVP, MVVM, or MVA -- best fits your project style?
- Is the framework stable, mature, and has it plenty of successful real-world deployments.
- Does the framework have good documentation with good example code?
- Is the framework actively maintained; are bugs resolved quickly and effectively?
- Does it have an active community behind it that is willing to support each other?
- Is the framework opiniated or not? Are you willing to totally commit yourself?

The last point about *opiniated* frameworks is important. Some frameworks stay out of the way and let you use it when you need it. In other words, you can use it as much or as little as you like in your pages. Others however, require that you follow their structure to the letter for the entire project or else things stop working. These are called opiniated frameworks, i.e. they have strong opinions on standards, layout, code organization, and other conventions.

Below we have listed the most high-profile frameworks in order of popularity. Mind you that things change very quickly in this space and what is fashionable today, may be out of favor tomorrow. So, take this ordering with a grain of salt.

1. **Backbone**. Small framework, less than 1000 lines of code. MVC, with C standing for Collections. Includes rich Routing. Many successful deployments at high-profile sites. Mature.
2. **Ember**. Large framework, both in size and functionality. Inspired by Rails and Cocoa. Well thought out but opiniated. Intended to control entire web page. New.
3. **Angular**. Developed by Google. Includes templating and data binding. Designed with an eye towards the future: "this is where web development is going." MVVM like. Relatively new.
4. **Knockout**. Pure MVVM. Includes templating, declarative data binding, and observable models. No built-in routing. Not very opiniated: allows limited usage. Mature.

5. **Spine**. Very small framework. Derived from Backbone without Collections, but adds numerous small modifications. Uses Models and Controllers. Relatively new.
6. **Batman**. Primarily for Rails and CoffeeScript developers. Rich models, views, and controllers. Highly opiniated: you must follow their conventions. New.

The least opiniated ones are Backbone, Knockout, and Spine. They are reasonably agnostic and can be used on an as-needed basis in most environments. It is interesting to note that the highly opiniated frameworks, Ember and Batman, have close ties to Ruby on Rails which itself is an opiniated development framework that follows the *convention over configuration* rule. Rails developers will most certainly lean towards these frameworks.

Furthermore, Knockout has evolved from the .NET MVVM arena, so developers that have experience with WPF or Silverlight in .NET will feel quickly at home with Knockout.

In our 'Patterns in Action' section we make use of Backbone.