NOVEMBER 2013 VOLUME 28 NUMBER 11

ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET

By Mike Wasson (https://msdn.microsoft.com/en-us/magazine/mt149362?author=mike+wasson) | November 2013 | Get the Code (https://github.com/MikeWasson/MoviesSPA)

Single-Page Applications (SPAs) are Web apps that load a single HTML page and dynamically update that page as the user interacts with the app.

SPAs use AJAX and HTML5 to create fluid and responsive Web apps, without constant page reloads. However, this means much of the work happens on the client side, in JavaScript. For the traditional ASP.NET developer, it can be difficult to make the leap. Luckily, there are many open source JavaScript frameworks that make it easier to create SPAs.

In this article, I'll walk through creating a simple SPA app. Along the way, I'll introduce some fundamental concepts for building SPAs, including the Model-View-Controller (MVC) and Model-View-ViewModel (MVVM) patterns, data binding and routing.

About the Sample App

The sample app I created is a simple movie database, shown in **Figure 1**. The far-left column of the page displays a list of genres. Clicking on a genre brings up a list of movies within that genre. Clicking the Edit button next to an entry lets you change that entry. After making edits, you can click Save to submit the update to the server, or Cancel to revert the changes.

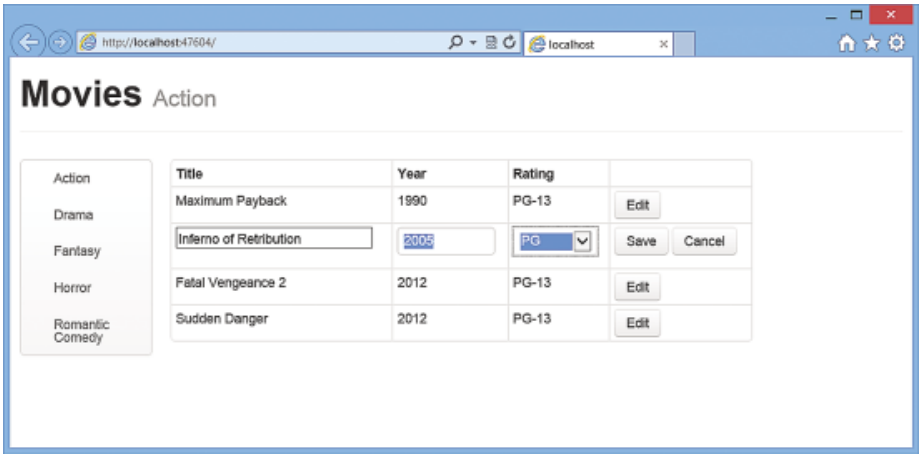

Figure 1 The Single-Page Application Movie Database App

I created two different versions of the app, one using the Knockout.js library and the other using the Ember.js library. These two libraries have different approaches, so it's instructive to compare them. In both cases, the client app was fewer than 150 lines of JavaScript. On the server side, I used ASP.NET Web API to serve JSON to the client. You can find source code for both versions of the app at github.com/MikeWasson/MoviesSPA (http://github.com/MikeWasson/MoviesSPA).

(Note: I created the app using the release candidate [RC] version of Visual Studio 2013. Some things might change for the released to manufacturing [RTM] version, but they shouldn't affect the code.)

Background

In a traditional Web app, every time the app calls the server, the server renders a new HTML page. This triggers a page refresh in the browser. If you've ever written a Web Forms application or PHP application, this page lifecycle should look familiar.

In an SPA, after the first page loads, all interaction with the server happens through AJAX calls. These AJAX calls return data—not markup—usually in JSON format. The app uses the JSON data to update the page dynamically, without reloading the page. **Figure 2** illustrates the difference between the two approaches.

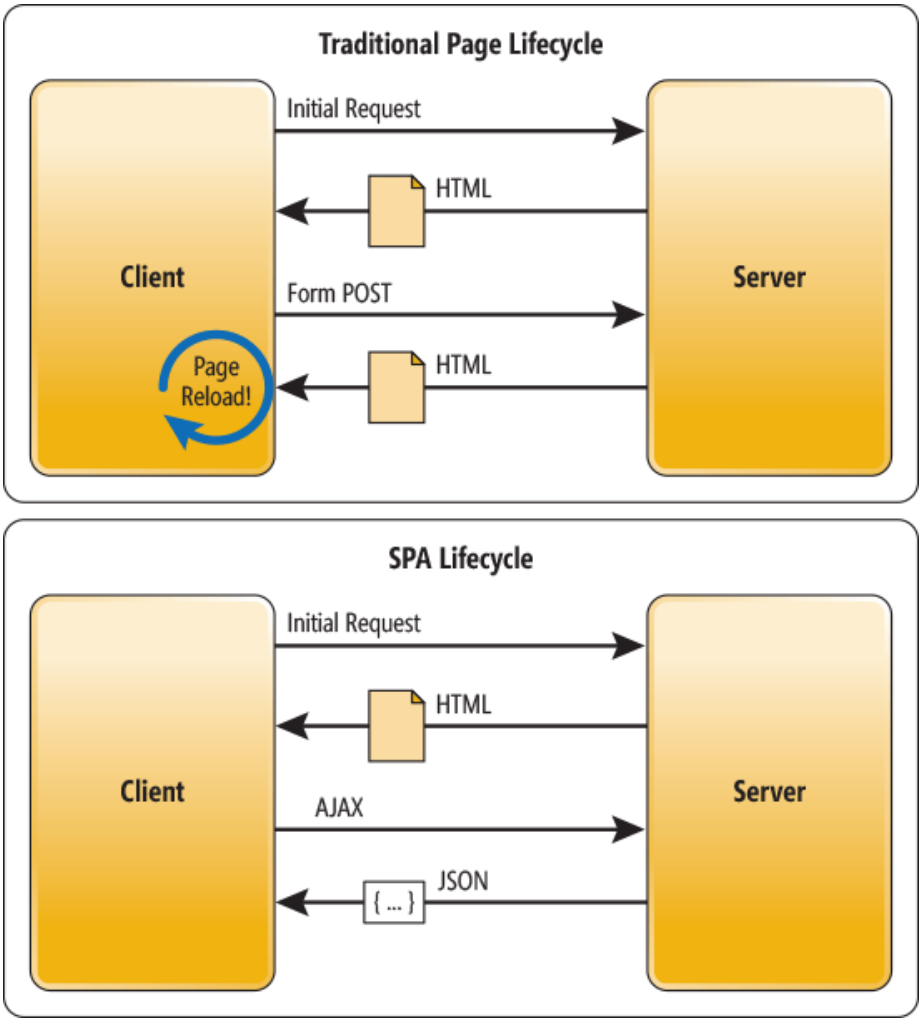

Figure 2 The Traditional Page Lifecycle vs. the SPA Lifecycle

One benefit of SPAs is obvious: Applications are more fluid and responsive, without the jarring effect of reloading and re-rendering the page. Another benefit might be less obvious and it concerns how you architect a Web app. Sending the app data as JSON creates a separation between the presentation (HTML markup) and application logic (AJAX requests plus JSON responses).

This separation makes it easier to design and evolve each layer. In a well-architected SPA, you can change the HTML markup without touching the code that implements the application logic (at least, that's the ideal). You'll see this in action when I discuss data binding later.

In a pure SPA, all UI interaction occurs on the client side, through JavaScript and CSS. After the initial page load, the server acts purely as a service layer. The client just needs to know what HTTP requests to send. It doesn't care how the server implements things on the back end.

With this architecture, the client and the service are independent. You could replace the entire back end that runs the service, and as long as you don't change the API, you won't break the client. The reverse is also true—you can replace the entire client app without changing the service layer. For example, you might write a native mobile client that consumes the service.

Creating the Visual Studio Project

Visual Studio 2013 has a single ASP.NET Web Application project type. The project wizard lets you select the ASP.NET components to include in your project. I started with the Empty template and then added ASP.NET Web API to the project by checking Web API under "Add folders and core references for:" as shown in **Figure 3**.

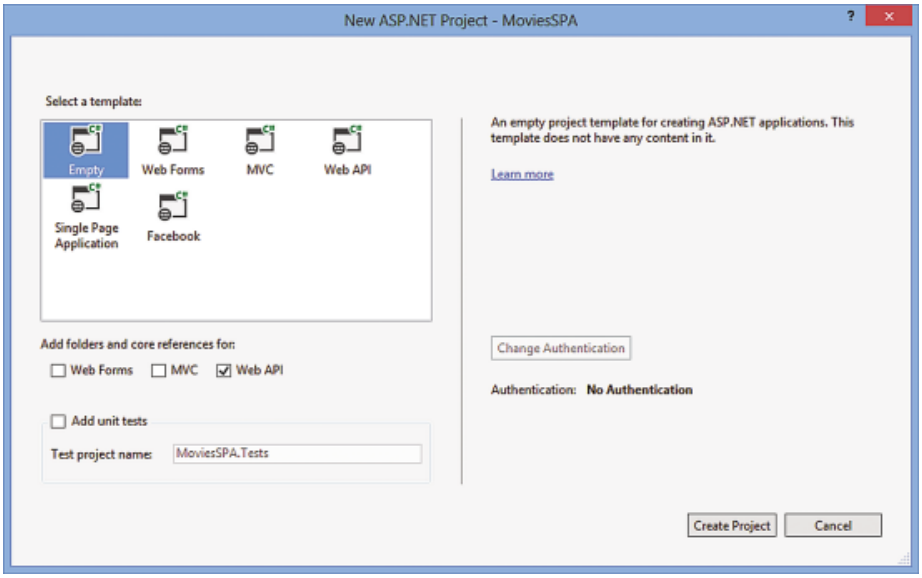

Figure 3 Creating a New ASP.NET Project in Visual Studio 2013

The new project has all the libraries needed for Web API, plus some Web API configuration code. I didn't take any dependency on Web Forms or ASP.NET MVC.

Notice in **Figure 3** that Visual Studio 2013 includes a Single Page Application template. This template installs a skeleton SPA built on Knockout.js. It supports log in using a membership database or external authentication provider. I didn't use the template in my app because I wanted to show a simpler example starting from scratch. The SPA template is a great resource, though, especially if you want to add authentication to your app.

Creating the Service Layer

I used ASP.NET Web API to create a simple REST API for the app. I won't go into detail about Web API here—you can read more at asp.net/web-api.

First, I created a Movie class that represents a movie. This class does two things:

- Tells Entity Framework (EF) how to create the database tables to store the movie data.
- Tells Web API how to format the JSON payload.

You don't have to use the same model for both. For example, you might want your database schema to look different from your JSON payloads. For this app, I kept things simple:

```
namespace MoviesSPA.Models
{
  public class Movie
  {
    public int ID { get; set; }
    public string Title { get; set; }
    public int Year { get; set; }
    public string Genre { get; set; }
    public string Rating { get; set; }
  }
}
```

Next, I used Visual Studio scaffolding to create a Web API controller that uses EF as the data layer. To use the scaffolding, right-click the Controllers folder in Solution Explorer and select Add | New Scaffolded Item. In the Add Scaffold wizard, select "Web API 2 Controller with actions, using Entity Framework," as shown in **Figure 4**.

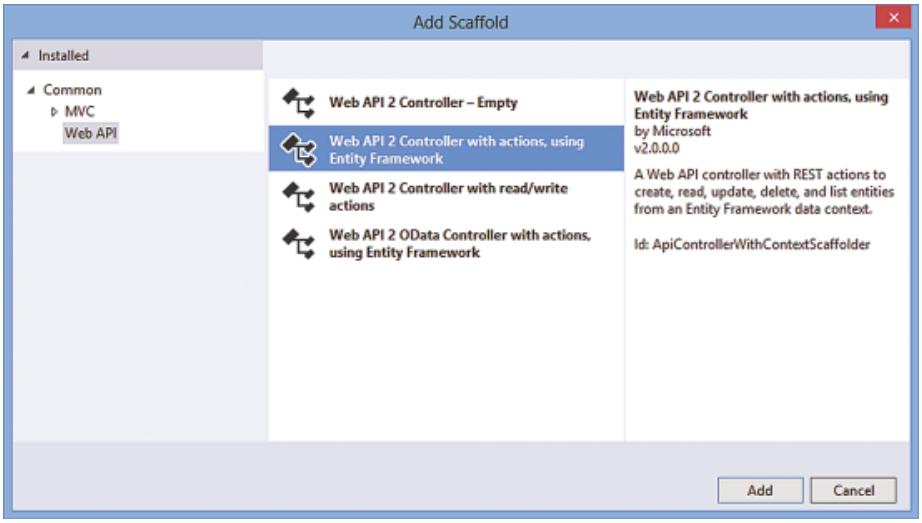

Figure 4 Adding a Web API Controller

Figure 5 shows the Add Controller wizard. I named the controller MoviesController. The name matters, because the URIs for the REST API are based on the controller name. I also checked "Use async controller actions" to take advantage of the new async feature in EF 6. I selected the Movie class for the model and selected "New data context" to create a new EF data context.

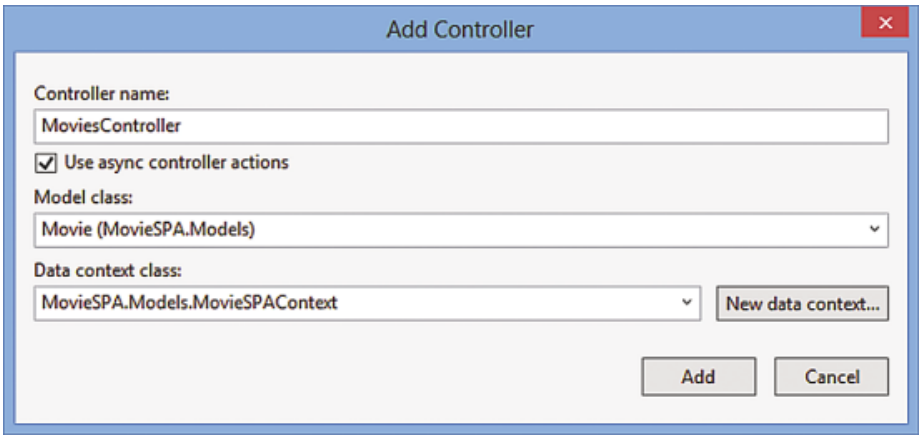

Figure 5 The Add Controller Wizard

The wizard adds two files:

- MoviesController.cs defines the Web API controller that implements the REST API for the app.
- MovieSPAContext.cs is basically EF glue that provides methods to query the underlying database.

Figure 6 shows the default REST API the scaffolding creates.

Figure 6 The Default REST API Created by the Web API Scaffolding

HTTP Verb	URI	Description
GET	/api/movies	Get a list of all movies
GET	/api/movies/{id}	Get the movie with ID equal to {id}
PUT	/api/movies/{id}	Update the movie with ID equal to {id}
POST	/api/movies	Add a new movie to the database
DELETE	/api/movies/{id}	Delete a movie from the database

Values in curly brackets are placeholders. For example, to get a movie with ID equal to 5, the URI is /api/movies/5.

I extended this API by adding a method that finds all the movies in a specified genre:

```
public class MoviesController : ApiController
{
  public IQueryable<Movie> GetMoviesByGenre(string genre)
  {
    return db.Movies.Where(m =>
      m.Genre.Equals(genre, StringComparison.OrdinalIgnoreCase));
  }
  // Other code not shown
```

The client puts the genre in the query string of the URI. For example, to get all movies in the Drama genre, the client sends a GET request to /api/movies?genre=drama. Web API automatically binds the query parameter to the genre parameter in the GetMoviesByGenre method.

Creating the Web Client

So far, I've just created a REST API. If you send a GET request to /api/movies?genre=drama, the raw HTTP response looks like this:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Date: Tue, 10 Sep 2013 15:20:59 GMT
Content-Length: 240
[{"ID":5,"Title":"Forgotten Doors","Year":2009,"Genre":"Drama","Rating":"R"}
```

Now I need to write a client app that does something meaningful with this. The basic workflow is:

- UI triggers an AJAX request
- Update the HTML to display the response payload
- Handle AJAX errors

You could code all of this by hand. For example, here's some jQuery code that creates a list of movie titles:

```
$.getJSON(url)
  .done(function (data) {
    // On success, "data" contains a list of movies
    var ul = $("<ul></ul>")
    $.each(data, function (key, item) {
      // Add a list item
      $('<li>', { text: item.Title }).appendTo(ul);
    });
  $('#movies').html(ul);
});
```

This code has some problems. It mixes application logic with presentation logic, and it's tightly bound to your HTML. Also, it's tedious to write. Instead of focusing on your app, you spend your time writing event handlers and code to manipulate the DOM.

The solution is to build on top of a JavaScript framework. Luckily, you can choose from many open source JavaScript frameworks. Some of the more popular ones include Backbone, Angular, Ember, Knockout, Dojo and JavaScriptMVC. Most use some variation of the MVC or MVVM patterns, so it might be helpful to review those patterns.

The MVC and MVVM Patterns

The MVC pattern dates back to the 1980s and early graphical UIs. The goal of MVC is to factor the code into three separate responsibilities, shown in **Figure 7**. Here's what they do:

- The model represents the domain data and business logic.
- The view displays the model.
- The controller receives user input and updates the model.

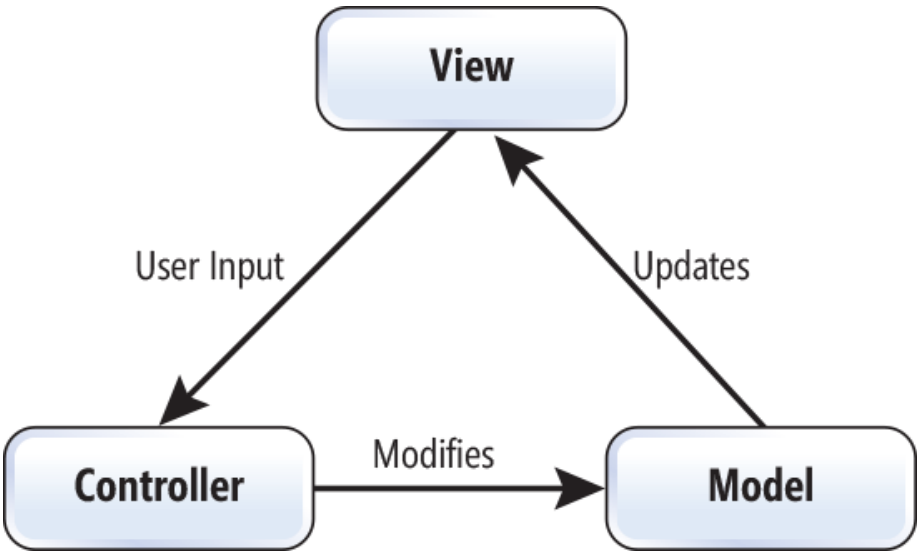

Figure 7 The MVC Pattern

A more recent variant of MVC is the MVVM pattern (see **Figure 8**). In MVVM:

- The model still represents the domain data.
- The view model is an abstract representation of the view.
- The view displays the view model and sends user input to the view model.

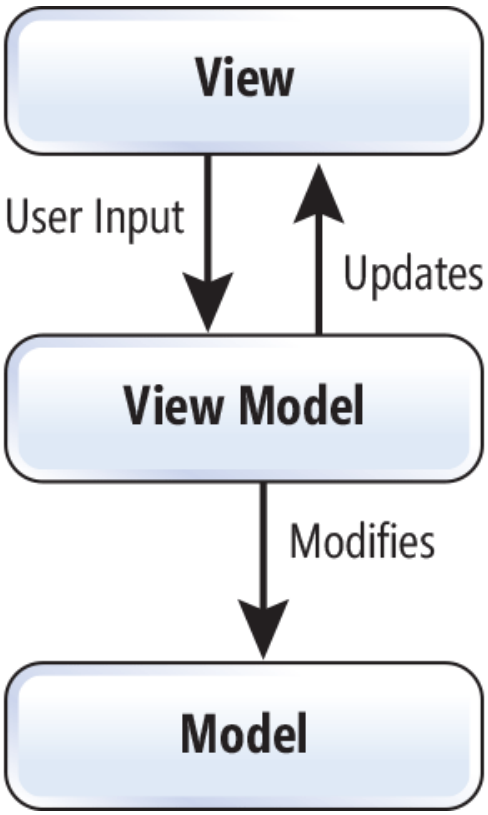

Figure 8 The MVVM Pattern

In a JavaScript MVVM framework, the view is markup and the view model is code.

MVC has many variants, and the literature on MVC is often confusing and contradictory. Perhaps that's not surprising for a design pattern that started with Smalltalk-76 and is still being used in modern Web apps. So even though it's good to know the theory, the main thing is to understand the particular MVC framework you're using.

Building the Web Client with Knockout.js

For the first version of my app, I used the Knockout.js library. Knockout follows the MVVM pattern, using data binding to connect the view with the view model.

To create data bindings, you add a special data-binding attribute to the HTML elements. For example, the following markup binds the span element to a property named genre on the view model. Whenever the value of genre changes, Knockout automatically updates the HTML:

```
<h1><span data-bind="text: genre"></span></h1>
```

Bindings can also work in the other direction—for example, if the user enters text into a text box, Knockout updates the corresponding property in the view model.

The nice part is that data binding is declarative. You don't have to wire up the view model to the HTML page elements. Just add the data-binding attribute and Knockout does the rest.

I started by creating an HTML page with the basic layout, with no data binding, as shown in **Figure 9**.

(Note: I used the Bootstrap library to style the app, so the real app has a lot of extra elements and CSS classes to control the formatting. I left these out of the code examples for clarity.)

Figure 9 Initial HTML Layout

```
<!DOCTYPE html>
<html>
<head>
  <title>Movies SPA</title>
</head>
<body>
  <ul>
    <li><a href="#"><!-- Genre --></a></li>
  </ul>
  <table>
    <thead>
      <tr><th>Title</th><th>Year</th><th>Rating</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td><!-- Title --></td>
        <td><!-- Year --></td>
        <td><!-- Rating --></td></tr>
    </tbody>
  </table>
  <p><!-- Error message --></p>
  <p>No records found.</p>
</body>
</html>
```

Creating the View Model

Observables are the core of the Knockout data-binding system. An observable is an object that stores a value and can notify subscribers when the value changes. The following code converts the JSON representation of a movie into the equivalent object with observables:

```
function movie(data) {
  var self = this;
  data = data || {};
  // Data from model
  self.ID = data.ID;
  self.Title = ko.observable(data.Title);
  self.Year = ko.observable(data.Year);
  self.Rating = ko.observable(data.Rating);
  self.Genre = ko.observable(data.Genre);
};
```

Figure 10 shows my initial implementation of the view model. This version only supports getting the list of movies. I'll add the editing features later. The view model contains observables for the list of movies, an error string and the current genre.

Figure 10 The View Model

```
var ViewModel = function () {
  var self = this;
  // View model observables
  self.movies = ko.observableArray();
  self.error = ko.observable();
  self.genre = ko.observable();  // Genre the user is currently browsing
  // Available genres
  self.genres = ['Action', 'Drama', 'Fantasy', 'Horror', 'Romantic Comedy'];
  // Adds a JSON array of movies to the view model
  function addMovies(data) {
    var mapped = ko.utils.arrayMap(data, function (item) {
      return new movie(item);
    });
    self.movies(mapped);
  }
  // Callback for error responses from the server
  function onError(error) {
    self.error('Error: ' + error.status + ' ' + error.statusText);
  }
  // Fetches a list of movies by genre and updates the view model
  self.getByGenre = function (genre) {
    self.error(''); // Clear the error
    self.genre(genre);
    app.service.byGenre(genre).then(addMovies, onError);
  };
  // Initialize the app by getting the first genre
  self.getByGenre(self.genres[0]);
}
// Create the view model instance and pass it to Knockout
ko.applyBindings(new ViewModel());
```

Notice that movies is an observableArray. As the name implies, an observableArray acts as an array that notifies subscribers when the array contents change.

The getByGenre function makes an AJAX request to the server for the list of movies and then populates the self.movies array with the results.

When you consume a REST API, one of the trickiest parts is handling the asynchronous nature of HTTP. The jQuery ajax function returns an object that implements the Promises API. You can use a Promise object's then method to set a callback that's invoked when the AJAX call completes successfully and another callback that's invoked if the AJAX call fails:

```
app.service.byGenre(genre).then(addMovies, onError);
```

Data Bindings

Now that I have a view model, I can data bind the HTML to it. For the list of genres that appears in the left side of the screen, I used the following data bindings:

```
<ul data-bind="foreach: genres">
  <li><a href="#"><span data-bind="text: $data"></span></a></li>
</ul>
```

The data-bind attribute contains one or more binding declarations, where each binding has the form "binding: expression." In this example, the foreach binding tells Knockout to loop through the contents of the genres array in the view model. For each item in the array, Knockout creates a new <li> element. The text binding in the sets the span text equal to the value of the array item, which in this case is the name of the genre.

Right now, clicking on the genre names doesn't do anything, so I added a click binding to handle click events:

```
<li><a href="#" data-bind="click: $parent.getByGenre">
  <span data-bind="text: $data"></span></a></li>
```

This binds the click event to the getByGenre function on the view model. I needed to use $parent here, because this binding occurs within the context of the foreach. By default, bindings within a foreach refer to the current item in the loop.

To display the list of movies, I added bindings to the table, as shown in **Figure 11**.

Figure 11 Adding Bindings to the Table to Display a List of Movies

```
<table data-bind="visible: movies().length > 0">
  <thead>
    <tr><th>Title</th><th>Year</th><th>Rating</th><th></th></tr>
  </thead>
  <tbody data-bind="foreach: movies">
    <tr>
      <td><span data-bind="text: Title"></span></td>
      <td><span data-bind="text: Year"></span></td>
      <td><span data-bind="text: Rating"></span></td>
      <td><!-- Edit button will go here --></td>
    </tr>
  </tbody>
</table>
```

In **Figure 11**, the foreach binding loops over an array of movie objects. Within the foreach, the text bindings refer to properties on the current object.

The visible binding on the <table> element controls whether the table is rendered. This will hide the table if the movies array is empty.

Finally, here are the bindings for the error message and the "No records found" message (notice that you can put complex expressions into a binding):

```
<p data-bind="visible: error, text: error"></p>
<p data-bind="visible: !error() && movies().length == 0">No records found.</
```

Making the Records Editable

The last part of this app is giving the user the ability to edit the records in the table. This involves several bits of functionality:

- Toggling between viewing mode (plain text) and editing mode (input controls).
- Submitting updates to the server.
- Letting the user cancel an edit and revert to the original data.

To track the viewing/editing mode, I added a Boolean flag to the movie object, as an observable:

```
function movie(data) {
  // Other properties not shown
  self.editing = ko.observable(false);
};
```

I wanted the table of movies to display text when the editing property is false, but switch to input controls when editing is true. To accomplish this, I used the Knockout if and ifnot bindings, as shown in **Figure 12**. The "<!-- ko -->" syntax lets you include if and ifnot bindings without putting them inside an HTML container element.

Figure 12 Enabling Editing of Movie Records

```
<tr>
  <!-- ko if: editing -->
  <td><input data-bind="value: Title" /></td>
  <td><input type="number" class="input-small" data-bind="value: Year" /></t
  <td><select class="input-small"
    data-bind="options: $parent.ratings, value: Rating"></select></td>
  <td>
    <button class="btn" data-bind="click: $parent.save">Save</button>
    <button class="btn" data-bind="click: $parent.cancel">Cancel</button>
  </td>
  <!-- /ko -->
  <!-- ko ifnot: editing -->
  <td><span data-bind="text: Title"></span></td>
  <td><span data-bind="text: Year"></span></td>
  <td><span data-bind="text: Rating"></span></td>
  <td><button class="btn" data-bind="click: $parent.edit">Edit</button></td>
  <!-- /ko -->
</tr>
```

The value binding sets the value of an input control. This is a two-way binding, so when the user types something in the text field or changes the dropdown selection, the change automatically propagates to the view model.

I bound the button click handlers to functions named save, cancel and edit on the view model.

The edit function is easy. Just set the editing flag to true:

```
self.edit = function (item) {
  item.editing(true);
};
```

Save and cancel were a bit trickier. In order to support cancel, I needed a way to cache the original value during editing. Fortunately, Knockout makes it easy to extend the behavior of observables. The code in **Figure 13** adds a store function to the observable class. Calling the store function on an observable gives the observable two new functions: revert and commit.

Figure 13 Extending ko.observable with Revert and Commit

Now I can call the store function to add this functionality to the model:

```
function movie(data) {
  // ...
  // New code:
  self.Title = ko.observable(data.Title).store();
  self.Year = ko.observable(data.Year).store();
  self.Rating = ko.observable(data.Rating).store();
  self.Genre = ko.observable(data.Genre).store();
};
```

Figure 14 shows the save and cancel functions on the view model.

Figure 14 Adding Save and Cancel Functions

```
self.cancel = function (item) {
  revertChanges(item);
  item.editing(false);
};
self.save = function (item) {
  app.service.update(item).then(
    function () {
      commitChanges(item);
    },
    function (error) {
      onError(error);
      revertChanges(item);
    }).always(function () {
      item.editing(false);
  });
}
function commitChanges(item) {
  for (var prop in item) {
    if (item.hasOwnProperty(prop) && item[prop].commit) {
      item[prop].commit();
    }
  }
}
function revertChanges(item) {
  for (var prop in item) {
    if (item.hasOwnProperty(prop) && item[prop].revert) {
      item[prop].revert();
    }
  }
}
```

Building the Web Client with Ember

For comparison, I wrote another version of my app using the Ember.js library.

An Ember app starts with a routing table, which defines how the user will navigate through the app:

```
window.App = Ember.Application.create();
App.Router.map(function () {
  this.route('about');
  this.resource('genres', function () {
    this.route('movies', { path: '/:genre_name' });
  });
});
```

The first line of code creates an Ember application. The call to Router.map creates three routes. Each route corresponds to a URI or URI pattern:

```
/#/about
/#/genres
/#/genres/genre_name
```

For every route, you create an HTML template using the Handlebars template library.

Ember has a top-level template for the entire app. This template gets rendered for every route. **Figure 15** shows the application template for my app. As you can see, the template is basically HTML, placed within a script tag with type="text/x-handlebars." The template contains special Handlebars markup inside double curly braces: {{ }}. This markup serves a similar purpose as the data-bind attribute in Knockout. For example, {{#linkTo}} creates a link to a route.

Figure 15 The Application-Level Handlebars Template

```
ko.observable.fn.store = function () {
  var self = this;
  var oldValue = self();
  var observable = ko.computed({
    read: function () {
      return self();
    },
    write: function (value) {
      oldValue = self();
      self(value);
    }
  });
  this.revert = function () {
    self(oldValue);
  }
  this.commit = function () {
    oldValue = self();
  }
  return this;
}
<script type="text/x-handlebars" data-template-name="application">
  <div class="container">
    <div class="page-header">
      <h1>Movies</h1>
    </div>
    <div class="well">
      <div class="navbar navbar-static-top">
        <div class="navbar-inner">
          <ul class="nav nav-tabs">
            <li>{{#linkTo 'genres'}}Genres{{/linkTo}} </li>
            <li>{{#linkTo 'about'}}About{{/linkTo}} </li>
          </ul>
        </div>
      </div>
    </div>
    <div class="container">
      <div class="row">{{outlet}}</div>
    </div>
  </div>
  <div class="container"><p>&copy;2013 Mike Wasson</p></div>
</script>
```

Now suppose the user navigates to /#/about. This invokes the "about" route. Ember first renders the top-level application template. Then it renders the about template inside the {{outlet}} of the application template. Here's the about template:

```
<script type="text/x-handlebars" data-template-name="about">
  <h2>Movies App</h2>
  <h3>About this app...</h3>
</script>
```

Figure 16 shows how the about template is rendered within the application template.

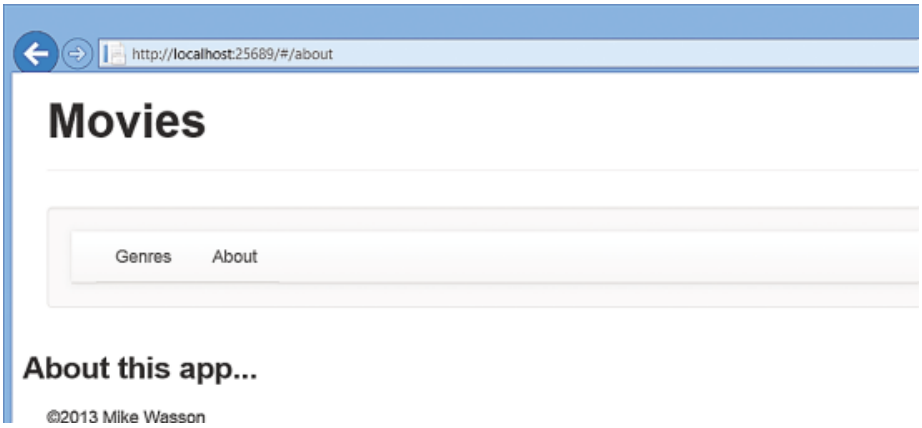

Figure 16 Rendering the About Template

Because each route has its own URI, the browser history is preserved. The user can navigate with the Back button. The user can also refresh the page without losing the context, or bookmark and reload the same page.

Ember Controllers and Models

In Ember, each route has a model and a controller. The model contains the domain data. The controller acts as a proxy for the model and stores any application state data for the view. (This doesn't exactly match the classic definition of MVC. In some ways, the controller is more like a view model.)

Here's how I defined the movie model:

```
App.Movie = DS.Model.extend({
  Title: DS.attr(),
  Genre: DS.attr(),
  Year: DS.attr(),
  Rating: DS.attr(),
});
```

The controller derives from Ember.ObjectController, as shown in **Figure 17**.

Figure 17 The Movie Controller Derives from Ember.ObjectController

```
App.MovieController = Ember.ObjectController.extend({
  isEditing: false,
  actions: {
    edit: function () {
      this.set('isEditing', true);
    },
    save: function () {
      this.content.save();
      this.set('isEditing', false);
    },
    cancel: function () {
      this.set('isEditing', false);
      this.content.rollback();
    }
  }
});
```

There are some interesting things going on here. First, I didn't specify the model in the controller class. By default, the route automatically sets the model on the controller. Second, the save and cancel functions use the transaction features built into the DS.Model class. To revert edits, just call the rollback function on the model.

Ember uses a lot of naming conventions to connect different components. The genres route talks to the GenresController, which renders the genres template. In fact, Ember will automatically create a GenresController object if you don't define one. However, you can override the defaults.

In my app, I configured the genres/movies route to use a different controller by implementing the renderTemplate hook. This way, several routes can share the same controller (see **Figure 18**).

Figure 18 Several Routes Can Share the Same Controller

```
App.GenresMoviesRoute = Ember.Route.extend({
  serialize: function (model) {
    return { genre_name: model.get('name') };
  },
  renderTemplate: function () {
    this.render({ controller: 'movies' });
  },
  afterModel: function (genre) {
    var controller = this.controllerFor('movies');
    var store = controller.store;
    return store.findQuery('movie', { genre: genre.get('name') })
    .then(function (data) {
      controller.set('model', data);
  });
  }
});
```

One nice thing about Ember is you can do things with very little code. My sample app is about 110 lines of JavaScript. That's shorter than the Knockout version, and I get browser history for free. On the other hand, Ember is also a highly "opinionated" framework. If you don't write your code the "Ember way," you're likely to hit some roadblocks. When choosing a framework, you should consider whether the feature set and the overall design of the framework match your needs and coding style.

Learn More

In this article, I showed how JavaScript frameworks make it easier to create SPAs. Along the way, I introduced some common features of these libraries, including data binding, routing, and the MVC and MVVM patterns. You can learn more about building SPAs with ASP.NET at asp.net/single-page-application (http://asp.net/single-page-application).

Mike Wasson *is a programmer-writer at Microsoft. For many years he documented the Win32 multimedia APIs. He currently writes about ASP.NET, focusing on Web API. You can reach him at mwasson@microsoft.com (mailto:mwasson@microsoft.com).*

Thanks to the following technical expert for reviewing this article: Xinyang Qiu (Microsoft)
Xinyang Qiu is a senior Software Design Engineer in Test on the Microsoft ASP.NET team and an active blogger for blogs.msdn.com/b/webdev (http://blogs.msdn.com/b/webdev). He's happy to answer ASP.NET questions or direct experts to answer your questions. Reach him at xinqiu@microsoft.com (mailto:xinqiu@microsoft.com).