KnockoutJS Demo Script

# Basics

1. Open default.html and run the page
2. Show person.js, talk about viewModel and ko.applyBindings()
3. Add the following data-bind commands to the two spans

data-bind=”text: firstName”

data-bind=”text: lastName”

1. Run the page.
2. Now let’s add some input fields on the page, and subscribe to the same fields

<p>First name: <input data-bind="value: firstName" /></p>

<p>Last name: <input data-bind="value: lastName" /></p>

1. Run the page and edit a box. Show that nothing changes. We need to tell the viewModel to watch these properties for changes and notify all subscribers about those changes.
2. Modify the viewModel to make properties observable:

var viewModel = {

firstName: ko.observable("Hugo"),

lastName: ko.observable("Reyes")

};

1. Run the page and change some textboxes
2. Now let’s say we want to combine the values of these two fields to display the users full name. For that knockout allows me to use dependent observables, which are observables that depend on other properties on a viewModel, i.e. they are not self-contained. Let’s add a fullName property to the viewModel

viewModel.fullName = ko.dependentObservable(function() {

return this.firstName() + " " + this.lastName();

}, viewModel);

1. And now a new field to display the full name:

<p>Full name: <span data-bind="text: fullName"></span></p>

1. Run and talk about how changes to the UI ripple through because Knockout handles dependency tracking for us
2. Now lets add one final behavior to illustrate using Knockout to respond to other user events. We’re going to add a function to capitalize the user’s last name. So add this function to the viewModel

capitalizeLastName: function() {

var currentVal = this.lastName(); // <-- Call observable values as functions

this.lastName(currentVal.toUpperCase()); // Pass the new value in as a parameter

}

1. Now, add a button to the page and wire up the click handler

<button data-bind="click: capitalizeLastName">Go caps</button>

# Templating

1. When you have repeating blocks of UI elements generated dynamically, you’ll want to look at using Templates with Knockout.
2. Open booking.html and booking.js and show out initial state on both
3. In booking.html, replace the tbody element with a template binding

<tbody data-bind="template: 'reservationTemplate'"></tbody>

1. Then define the template by adding a script block to the bottom of the page

<script type="text/x-jquery-tmpl" id="reservationTemplate">

{{each seats}}

<tr>

<td>${name || 'Anonymous'}</td>

<td>${meal().mealName}</td>

<td>${meal().price}</td>

</tr>

{{/each}}

</script>

1. Run and show the page
2. Because the array in question is an observable array, it’s easy to work with adding and removing elements, and have those cascade throughout the application.
3. Add a button called reserve another seat

<button data-bind="click: addSeat">Reserve another seat</button>

1. Now add an addSeat function to your ViewModel and run it again

, addSeat: function() {

this.seats.push(new seatReservation());

}

1. In using templating this way, we’re actually re-rendering the entire template each time. Show this by adding a date field to the template

<td>${**new** Date}</td>

1. We can make this more efficient and responsive with knockout’s help. First, modify the tbody tag like so:

<tbody data-bind="template: {name:'reservationTemplate', foreach: seats}"></tbody>

1. Then, remove the {{each}} tags from the template itself. Now rerun and show the timestamp.

# Knockout and ASP.NET MVC

1. Show the Create Speaker page and how we want to show a user profile view when the user enters data. But it doesn’t work.
2. Show the addSpeaker.js page with our viewModel and Create.cshml, \_CreateOrEdit.cshtml. Note that instead of using raw elements, we are using the strongly typed helpers. The good news is that data-bind still works fine with these, so you want to add an anonymous type with the data\_bind (notice the underscore) property on each.

Name: @Html.TextBoxFor(m => m.Name, new { data\_bind="value: name" }) <br />

Email: @Html.TextBoxFor(m => m.Email, new { data\_bind = "value: email" }) <br />

Bio: @Html.TextBoxFor(m => m.Bio, new { data\_bind = "value: bio" }) <br />

Twitter Handle: @Html.TextBoxFor(m => m.TwitterHandle, new { data\_bind = "value: twitterHandle" }) <br />

State: @Html.TextBoxFor(m => m.State, new { data\_bind = "value: state" }) <br />

Photo Url: @Html.TextBoxFor(m => m.PhotoUrl, new { data\_bind = "value: photoUrl" })

1. Now run the page and show everything working just fine and our bindings being picked up. Bottom line, it’s easy to use the strongly-typed helpers in MVC with knockout.
2. Chances are, if we’re doing all of this hard work to create a responsive ui, that we probably want to avoid a full form submit when we make changes. As you probably know, MVC provides facilities for working with JSON data, and knockout can help us send data back to the server.
3. Start by opening addSpeaker and add an addSpeaker function just before ko.applyBindings()

viewModel.addSpeaker = function () {

$.ajax({

url: "/speakers/create/",

type: 'post',

data: ko.toJSON(this),

contentType: 'application/json',

success: function (result) {

$('#notice').text(result).show(2000, function () {

$('#notice').hide(6000);

});

}

});

};

1. Then, open \_CreateOrEdit and add a data-bind attribute to the form:

data-bind="submit: addSpeaker"

1. Here, we’re just using KO to add a method that will respond to a form submission and, rather than posting back, it will call the addSpeaker method on our viewModel, which will perform a jquery ajax call back to the create post method on our controller. Let’s take a look at that.
2. Open SpeakersController.cs and place a breakpoint on it.
3. Run the page and save. It works!

# Knockout.Unobtrusive

1. To get KO.Unobtrusive, we’ll need to first go to <http://github.com/bsatrom/Knockout.Unobtrusive> to grab the source and add a reference
2. Then, the goal of KOU is to get the data-bind syntax out of my page and into my JavaScript. To do that, we’ll create a bindings object like so:

var bindings = {

value: ['name', 'bio', 'twitterHandle', 'state', 'photoUrl', 'languageToAdd', 'email', 'topicToAdd'],

text: [{ displayName: 'name' },

{ displayState: 'state' },

{ displayBio: 'bio' },

{ displayTwitterHandle: 'twitterHandle' },

{ languageList: 'languages' },

{ topicList: 'topics' },

{ displayPresentedBefore: 'presentedBefore'}],

options: ['languages', 'favoriteTopics'],

checked: ['presentedBefore'],

click: ['addLanguage', 'addTopic'],

custom: {

languageToAdd: 'valueUpdate: "afterkeydown"',

addLanguage: 'enable: enableAddLanguage',

photo: 'attr: {src: photoUrl, alt: name}',

twitterUrl: 'attr: {href: twitterUrl}'

}

};

1. This object centralizes all of our bindings, so we don’t have to litter our html with data-bind attributes if we don’t want to.
2. The next step we need to make this work is to remove all of our data-bind calls, and we’ll do that in the main part of the page:

Name: @Html.TextBoxFor(m => m.Name) <br />

Email: @Html.TextBoxFor(m => m.Email) <br />

Bio: @Html.TextBoxFor(m => m.Bio) <br />

Twitter Handle: @Html.TextBoxFor(m => m.TwitterHandle) <br />

State: @Html.TextBoxFor(m => m.State) <br />

Photo Url: @Html.TextBoxFor(m => m.PhotoUrl)

1. Finally, we need to call ko.unobtrusive to create our bindings for us from the bindings object.

ko.unobtrusive.createBindings(bindings);

1. Now run the site and show that things still work. View source and show the ID assigned by default. Go back to the viewModel and show how the Mapping object can define ids for both fields and viewModel or can specify both in an object literal. And how the properties on the object define the binding in question.