**KnockoutJS**

It is a JavaScript library used for communication between your UI and the underlying data model.

It simplifies user interactions and makes interfaces fully responsive to any data source changes. Knockout provides a simple two-way binding mechanism to link a data model to an UI, thus making synchronization between them a breeze.

If you want to get the most benefits you should use it together with jQuery.

Knockout implements Model-View-View Model (MVVM) design pattern for JavaScript. In this pattern your application is split into three parts:

* A model that holds your application’s data. Either data entered by users or JSON data fetched from a web server.
* A view that serves as a connector and communication layer between Model and View. It holds data and operations for manipulating this data and display it in the UI. Every time when data model is changed corresponded UI parts updates, reflecting these changes. View Model in your application is represented by JavaScript code.
* A view that refers to all UI elements in your application. It is a representation of the structure and appearance for given UI. The view is responsible for displaying data and accepting user input. View is represented by HTML/CSS code in your application.

**They are three core concepts upon Knockout is built:**

1. Declarative Bindings: These allow you to connect parts of your UI to your data model in a simple and convenient way. When use JavaScript directly to manipulates DOM this may cause broken code if you later change the DOM hierarchy or element IDs. With declarative bindings even if you change the DOM all bound pieces stay connected. You can bind data to a DOM by simply including a data-bind attribute to any DOM element.
2. Dependency Tracking: Thankfully to the bindings and special type of variables called observables every time when your model data has changed all parts associated with it automatically being updated.
3. Templating: This comes in handy when your application becomes more complex and you need a way to display a rich structure of view model data, thus keeping your code simple.

Start with referencing the following libraries in your HTML document.

**<script type='text/javascript' src='jquery-1.7.1.min.js'></script>**

**<script type='text/javascript' src='knockout-2.0.0.js'></script>**

**<script type='text/javascript' src='application.js'></script>**

The data-bind attribute (explained later) isn’t native to HTML, and the browser doesn’t know what it means. So in order to take effect Knockout has to be activated by inserting ko.applyBindings() function at the end of the script. Also, if you use external JavaScript file or your script is placed in the head tag of your document you need to wrap the Knockout code in a jQuery ready function in order to work properly.

**$(document).ready(function(){**

**function viewModel() {**

**// Your code here**

**};**

**ko.applyBindings(new viewModel()); //bind our model to the UI**

**});**

You can even provide a DOM element if you only want to bind this view model to one part of your overall UI.

**ko.applyBindings() takes two parameters.**

* The first parameter says what view model object you want to use with the declarative bindings it activates.
* The second parameter is optional and it defines which part of the document you want to search for data-bind attributes.

For example, **ko.applyBindings(viewModel, document.getElementById('container'))** will restrict the activation to the element with ID container and its descendants. This is useful if you want to have multiple view models and associate each with a different region of the page.

With Knockout, you can bind data to a DOM element by including a data-bind attribute in the markup.

The code never has any reference to the DOM structure so you can freely change the HTML without breaking your bindings. e.g:

**// syntax: data-bind="bindingName: bindingValue"**

**<p>The day of the week is <span data-bind="text: dayOfWeek"></span>. It's time for <span data-bind="text: activity"></span></p>**

Then if we want to make the value of text to updates dynamically then we have to declare it in our view model as an observable (source object and target DOM elements to stay in sync – two-way binding).

**function viewModel() {**

**this.dayOfWeek = ko.observable('Sunday');**

**this.activity = ko.observable('rest');**

**};**

**ko.applyBindings(new viewModel());**

Knockout implements observable properties by wrapping object properties with a custom function named ko.observable(): e.g:

**this.property = ko.observable('value')**

To read the observable's current value, just call the observable with no parameters.

The following will return Sunday

**this.dayOfWeek()**

To write a new value to the observable, call the observable and pass the new value as a parameter.

The following will change the day of week to Monday

**this.dayOfWeek('Monday')**

To write values to multiple observable properties on a model object, you can use chaining syntax.

**this.dayOfWeek('Monday').activity('work')**

**No need to use observable properties if** you want DOM elements to receive values once but then not be updated when the values in the source object change, So in this case simple objects will be sufficient.

In some cases you may need **to combine the values of two or more observables into one new value**. This can be done with so-called **computed observables**. Computed observables are functions that are dependent on one or more other observables, and will automatically update whenever any of these dependencies change.

For example: the computed observable named fullDate will updates every time when one or more of the day, month and year observables change.

**<p>Day: <input data-bind="value: day" /></p><p>Month: <input data-bind="value: month" /></p><p>Year: <input data-bind="value: year" /></p>**

**<p>The current date is <span data-bind="text: fullDate"></span></p>**

**function viewModel() {**

**this.day = ko.observable('24');**

**this.month = ko.observable('02');**

**this.year = ko.observable('2012');**

**this.fullDate = ko.computed(function() {**

**return this.day() + "/" + this.month() + "/" + this.year();**

**},this);**

**};**

**ko.applyBindings(new viewModel());**

The **ko.computed()** takes a second parameter this. Without passing it in, it would not have been possible to refer to **this.day(), this.month() or this.year().**

In order to simplify things you can create a variable self, thus avoiding the addition of the second parameter. From now on we will use this approach in the code examples.

**function viewModel() {**

**var self = this;**

**self.day = ko.observable('24');**

**self.month = ko.observable('02');**

**self.year = ko.observable('2012');**

**self.fullDate = ko.computed(function() {**

**return self.day() + "/" + self.month() + "/" + self.year();**

**});**

**};**

**ko.applyBindings(new viewModel());**

When you dealing with one object you can easily track any changes to it by turn it into an observable. But what if you have multiple objects? In such cases Knockout has a special object called **ko.observableArray()**, which can detect and respond to changes of a collection of things. This makes possible to display and/or edit multiple values, for example, when you need repeated sections of UI to appear and disappear as items are added and removed.

You should bear in mind that an observable array tracks which objects are in the array, not the state of those objects. An observable array just tracks which objects it holds, and notifies listeners when objects are added or removed. ex:

**this.property = ko.observableArray();**

When you create an observable array you can leave it empty or populate it with some initial values.

**<p>Today is <span data-bind="text: daysOfWeek()[0]"></span></p>**

**function viewModel() {**

**var self = this;**

**self.daysOfWeek = ko.observableArray([**

**'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'**

**]);**

**alert("The week has " + self.daysOfWeek().length + " days");**

**};**

**ko.applyBindings(new viewModel());**

To read and write Knockout array you can use any native JavaScript functions. But Knockout has its own equivalent functions which syntax is a little bit more convenient:

**array().push('Some value'); // native JavaScript**

**array.push('Some value'); // Knockout**

The syntax for using the built-in bindings is to include the Knockout binding name and the view model property pairs inside of the data-bind attribute of an HTML element.

**// syntax: data-bind="bindingName: bindingProperty"**

**<span data-bind="text: msg"></span>**

If you want to data bind to more than one property in the HTML element, simply separate the bindings by a comma using this syntax:

**<span data-bind="text: msg, visible: toggle"></span>**

The most of the bindings attempt to convert any parameter to a boolean value. If you give a value that isn’t actually boolean, it will be interpreted loosely. This means that nonzero numbers and non-null objects and non-empty strings will all be interpreted as true, whereas zero, null, undefined, and empty strings will be interpreted as false.

**Type of data binding**

* **Simple Bindings**

We’ve already seen **text binding** when dealing with observables in the previous section. The text binding is often used to display values in a span or div element.

**<p>The tip of the day is: <span data-bind="text: tipOfTheDay"></span></p>**

**function viewModel() {**

**var self = this;**

**self.tipOfTheDay = ko.observable('Relax.Take it easy!')**

**};**

**ko.applyBindings(new viewModel());**

**Value binding** sets the value of the associated element to the value of your parameter. This is typically used for form elements like input, select and textarea. When the user edits the value in the associated form control, it updates the value on your view model. Likewise, when you update the value in your view model, this updates the value of the form control on screen. This is known as two-way binding

By default, Knockout updates your view model when the user transfers focus to another DOM node, on the change event, but you can control when the value is updated using the valueUpdate parameter.

"**change**" is the default event and it updates your view model when the user moves the focus to a different control, or in the case of <select> elements, immediately after any change.

"**afterkeydown**" – updates your view model as soon as the user begins typing a character. This works by catching the browser’s keydown event and handling the event asynchronously. If you want to keep your view model updated in real-time using "afterkeydown" will be the best choice.

"**keyup**" – updates your view model when the user releases a key

"**keypress**" – updates your view model when the user has typed a key. This updates repeatedly if the user holds a key down

**<input data-bind="value: name, valueUpdate: 'afterkeydown'"></input>**

**<p data-bind="text: name"></p>**

**function viewModel() {**

**var self = this;**

**self.name = ko.observable()**

**};**

**ko.applyBindings(new viewModel());**

The **html binding** isn’t used as often, but it’s very handy for rendering HTML content in your view model. This binding sets the HTML of the associated element to the value of your parameter and is the equivalent of setting the innerHTML property on the DOM element.

**<div data-bind="html: markup"></div>**

**function viewModel() {**

**var self = this;**

**self.markup = ko.observable('<p><strong>Knockout</strong> is so <em>cool</em>!</p>')**

**};**

**ko.applyBindings(new viewModel());**

You will surely encounter some situations for which none exist. For those, Knockout offers the **attr binding**, which allows you to data bind any attribute to a view model property. The parameter should be a JavaScript object where the property names are the attributes and the property values are the value that will be bound to the attribute. This is very useful in many common scenarios, such as binding the href and title of the a element or the src and alt of the img element.

**<img data-bind="attr: {src: url, alt: details}" />**

**function viewModel() {**

**var self = this;**

**self.url = ko.observable(images/logo.png)**

**self.details = ko.observable('This is logo')**

**};**

**ko.applyBindings(new viewModel());**

You can bind styles with Knockout using the css and the style built-in bindings.

**css binding** sets one or more CSS classes for the associated element. The parameter should be a JavaScript object where the property names correspond to the desired CSS classes and the ***property values evaluate to true or false indicating whether the class should be applied***. You can set multiple CSS classes at once.

**<style>**

**.colorize {color: red}**

**</style>**

**<p data-bind="css: { colorize: on }">Text</p>**

**function viewModel() {**

**var self = this;**

**self.on = ko.observable(true)**

**};**

**ko.applyBindings(new viewModel());**

**<p data-bind="css: { colorize: on() > 3 }">Text</p>**

While it is better to use css classes whenever possible, at times you might want to set a specific style as well.

The parameter should be an object whose properties correspond to CSS styles names, and the values correspond to the style values you wish to apply. Typically this parameter value is declared using JSON.

**<p data-bind="style: {color: on() > 3 ? 'red' : 'black'}">Text</p>**

**function viewModel() {**

**var self = this;**

**self.on = ko.observable(5)**

**};**

**ko.applyBindings(new viewModel());**

Note: When you have an attribute or CSS class whose name is not legal JavaScript variable name then you should wrap the identifier name in quotes so that it becomes a string literal. And if you want to apply style whose name isn’t a legal JavaScript identifier, you must use the JavaScript name for that style.

**//correct:**

**<div data-bind="attr: { 'data-something': someValue }">...</div>**

**<div data-bind="css: { 'my-class': someValue }">...</div>**

**<div data-bind="style: { fontWeight: someValue }">...</div>**

To use event binding, you pass an object literal containing name value pairs for the event name and the view model method, separated by commas.

**<p data-bind="event: { mouseover: hello, mouseout: goodbye }"> Mouse over me! </p>**

**<p data-bind="text: helloEnabled"></p>**

**<p data-bind="text: goodbyeEnabled"></p>**

**function viewModel() {**

**var self = this;**

**self.helloEnabled = ko.observable()**

**self.goodbyeEnabled = ko.observable()**

**self.hello = function() {**

**self.helloEnabled('Hello!');**

**self.goodbyeEnabled('');**

**}**

**self.goodbye = function() {**

**self.goodbyeEnabled('Goodbye!');**

**self.helloEnabled('');**

**}**

**};**

**ko.applyBindings(new viewModel());**

click binding, as you may guess, is handling the click event. Because it is the most-used binding for events, it’s simply a shortcut to the event binding.

**<button data-bind="click: writeMSG">Show</button>**

**<p data-bind="text: msg"></p>**

**function viewModel() {**

**var self = this;**

**self.msg = ko.observable()**

**self.writeMSG = function() {**

**self.msg('Hello!')**

**}**

**};**

**ko.applyBindings(new viewModel());**

**submit binding** is a shortcut for handling the submit event for the form element. When you use the submit binding on a form, Knockout will prevent the browser’s default submit action for that form. In other words, the browser will call your handler function but will not submit the form to the server. This is a useful default because when you use the submit binding, it’s normally because you’re using the form as an interface to your view model, not as a regular HTML form. If you do want to let the form submit like a normal HTML form, just return true from your submit handler.

**visible binding** sets the visibility of the associated element based on the binding parameter value.

**<button data-bind="click: show">Show Message</button>**

**<button data-bind="click: hide">Hide Message</button>**

**<p data-bind="visible: msg">Hello, Knockout!</p>**

**function viewModel() {**

**var self = this;**

**self.msg = ko.observable()**

**self.show = function() {**

**self.msg(true)**

**}**

**self.hide = function() {**

**self.msg(false)**

**}**

**};**

**ko.applyBindings(new viewModel());**

**enable/disable binding** sets the disabled attribute on the associated element based on the supplied value. This is typically used for form elements like the input, select and textarea.

**<input data-bind="value: val, valueUpdate: 'afterkeydown'">**

**<button data-bind="enable: val">Send</button>**

**function viewModel() {**

**var self = this;**

**self.val = ko.observable()**

**};**

**ko.applyBindings(new viewModel());**

The **hasfocus binding** is handy when you want the focus to be set to a specific element on a form

**<input data-bind="value: val, hasfocus: on">**

**<button data-bind="enable: on">Send</button>**

**function viewModel() {**

**var self = this;**

**self.val = ko.observable()**

**self.on = ko.observable(false)**

**};**

**ko.applyBindings(new viewModel());**

The **checked binding** should be bound to a property or expression that evaluates to true or false. For **radio buttons**, the binding compares the buttons value attribute to the binding parameter.

**<p>Let me choose my favorite car: <input type="checkbox" data-bind="checked: car" /></p>**

**<div data-bind="visible: car">**

**Preferred model:**

**<div><input type="radio" name="modelsGroup" value="ferrari" data-bind="checked: model" /> Ferrari</div>**

**<div><input type="radio" name="modelsGroup" value="lamborghini" data-bind="checked: model" /> Lamborghini</div>**

**<div><input type="radio" name="modelsGroup" value="bugatti" data-bind="checked: model" /> Bugatti</div>**

**</div>**

**function viewModel() {**

**var self = this;**

**self.car = ko.observable(),**

**self.model = ko.observable("lamborghini") // Initially selects Lamborghini**

**};**

**ko.applyBindings(new viewModel());**

The **options binding** identifies a list of values to display, usually from an array property on the view model.

**<p>Choose your destiny: <select data-bind="options: availableRoles"></select></p>**

**function viewModel() {**

**var self = this;**

**self.availableRoles = ko.observableArray(['an artist', 'an actor', 'an author'])**

**};**

**ko.applyBindings(new viewModel());**

The **selectedOptions binding** controls which elements in a multi-select list are currently selected. When the user selects or de-selects an item in the multi-select list, this adds or removes the corresponding value to an array on your view model.

**<p>Choose your destiny: <select data-bind="options: availableRoles, selectedOptions: selected" multiple="true"></select></p>**

**function viewModel() {**

**var self = this;**

**self.availableRoles = ko.observableArray(['an artist', 'an actor', 'an author'])**

**self.selected = ko.observableArray(['an author'])**

**};**

**ko.applyBindings(new viewModel());**

**ptionsCaption** is useful when you don’t want to have any prticular option selected by default

**<p>Locales: <select data-bind="options: locales, selectedOptions: selected, optionsCaption: 'Select your locale...', optionsText: 'country', optionsValue: 'code'"></select></p>**

**<p data-bind="text: selected"></p>**

**function viewModel() {**

**var self = this;**

**self.locales = [**

**{ country: 'USA', code: 'en\_US' },**

**{ country: 'Spain', code: 'es\_ES' },**

**{ country: 'French', code: 'fr\_FR' }**

**]**

**self.selected = ko.observableArray();**

**}**

**ko.applyBindings(new viewModel());**

**The foreach binding**

**<table>**

**<thead>**

**<tr><th>Title</th><th>Author</th></tr>**

**</thead>**

**<tbody data-bind="foreach: books">**

**<tr>**

**<td data-bind="text: title"></td>**

**<td data-bind="text: author"></td>**

**</tr>**

**</tbody>**

**</table>**

**<script type="text/javascript">**

**function viewModel() {**

**var self = this;**

**self.books = ko.observableArray([**

**{ title: 'The Secret', author: 'Rhonda Byrne' },**

**{ title: 'The Power', author: 'Rhonda Byrne' },**

**{ title: 'The Magic', author: 'Rhonda Byrne' }]);**

**}**

**ko.applyBindings(new viewModel());**

**</script>**

Sometimes you may need to refer to the array entry itself rather than just one of its properties. In that case, you can use the pseudovariable $data. It means "the current item", when is used within a foreach block.

**<ul data-bind="foreach: daysOfWeek">**

**<li>**

**<span data-bind="text: $data"></span>**

**</li>**

**</ul>**

**<script type="text/javascript">**

**function viewModel() {**

**var self = this;**

**self.daysOfWeek = ko.observableArray([**

**'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'**

**]);**

**};**

**ko.applyBindings(new viewModel());**

**</script>**