Introduction

Knockout is a JavaScript library that helps you to create rich, responsive display and editor user interfaces with a clean underlying data model. Any time you have sections of UI that update dynamically (e.g., changing depending on the user’s actions or when an external data source changes), KO can help you implement it more simply and maintainably.

Headline features:

Elegant dependency tracking - automatically updates the right parts of your UI whenever your data model changes.

Declarative bindings - a simple and obvious way to connect parts of your UI to your data model. You can construct a complex dynamic UIs easily using arbitrarily nested binding contexts.

Trivially extensible - implement custom behaviors as new declarative bindings for easy reuse in just a few lines of code.

Additional benefits:

Pure JavaScript library - works with any server or client-side technology

Can be added on top of your existing web application without requiring major architectural changes

Compact - around 13kb after gzipping

Works on any mainstream browser (IE 6+, Firefox 2+, Chrome, Safari, others)

Comprehensive suite of specifications (developed BDD-style) means its correct functioning can easily be verified on new browsers and platforms

Developers familiar with Ruby on Rails, ASP.NET MVC, or other MV\* technologies may see MVVM as a real-time form of MVC with declarative syntax. In another sense, you can think of KO as a general way to make UIs for editing JSON data… whatever works for you :)

OK, how do you use it?

The quickest and most fun way to get started is by working through the interactive tutorials. Once you’ve got to grips with the basics, explore the live examples and then have a go with it in your own project.

Is KO intended to compete with jQuery (or Prototype, etc.) or work with it?

Everyone loves jQuery! It’s an outstanding replacement for the clunky, inconsistent DOM API we had to put up with in the past. jQuery is an excellent low-level way to manipulate elements and event handlers in a web page. KO solves a different problem.

As soon as your UI gets nontrivial and has a few overlapping behaviors, things can get tricky and expensive to maintain if you only use jQuery. Consider an example: you’re displaying a list of items, stating the number of items in that list, and want to enable an ‘Add’ button only when there are fewer than 5 items. jQuery doesn’t have a concept of an underlying data model, so to get the number of items you have to infer it from the number of TRs in a table or the number of DIVs with a certain CSS class. Maybe the number of items is displayed in some SPAN, and you have to remember to update that SPAN’s text when the user adds an item. You also must remember to disable the ‘Add’ button when the number of TRs is 5. Later, you’re asked also to implement a ‘Delete’ button and you have to figure out which DOM elements to change whenever it’s clicked.

How is Knockout different?

It’s much easier with KO. It lets you scale up in complexity without fear of introducing inconsistencies. Just represent your items as a JavaScript array, and then use a foreach binding to transform this array into a TABLE or set of DIVs. Whenever the array changes, the UI changes to match (you don’t have to figure out how to inject new TRs or where to inject them). The rest of the UI stays in sync. For example, you can declaratively bind a SPAN to display the number of items as follows:

There are <span data-bind="text: myItems().count"></span> items

That’s it! You don’t have to write code to update it; it updates on its own when the myItems array changes. Similarly, to make the ‘Add’ button enable or disable depending on the number of items, just write:

<button data-bind="enable: myItems().count < 5">Add</button>

Later, when you’re asked to implement the ‘Delete’ functionality, you don’t have to figure out what bits of the UI it has to interact with; you just make it alter the underlying data model.

To summarise: KO doesn’t compete with jQuery or similar low-level DOM APIs. KO provides a complementary, high-level way to link a data model to a UI. KO itself doesn’t depend on jQuery, but you can certainly use jQuery at the same time, and indeed that’s often useful if you want things like animated transitions

The "foreach" binding

Purpose

The foreach binding duplicates a section of markup for each entry in an array, and binds each copy of that markup to the corresponding array item. This is especially useful for rendering lists or tables.

Assuming your array is an observable array, whenever you later add, remove, or re-order array entries, the binding will efficiently update the UI to match - inserting or removing more copies of the markup, or re-ordering existing DOM elements, without affecting any other DOM elements. This is far faster than regenerating the entire foreach output after each array change.

Of course, you can arbitrarily nest any number of foreach bindings along with other control-flow bindings such as if and with.

Example 1: Iterating over an array

This example uses foreach to produce a read-only table with a row for each array entry.

<table>

<thead>

<tr><th>First name</th><th>Last name</th></tr>

</thead>

<tbody data-bind="foreach: people">

<tr>

<td data-bind="text: firstName"></td>

<td data-bind="text: lastName"></td>

</tr>

</tbody>

</table>

<script type="text/javascript">

ko.applyBindings({

people: [

{ firstName: 'Bert', lastName: 'Bertington' },

{ firstName: 'Charles', lastName: 'Charlesforth' },

{ firstName: 'Denise', lastName: 'Dentiste' }

]

});

</script>

Example 2: Live example with add/remove

The following example shows that, if your array is observable, then the UI will be kept in sync with changes to that array.

People

Name at position 0: Bert Remove

Name at position 1: Charles Remove

Name at position 2: Denise Remove

Add

Source code: View

<h4>People</h4>

<ul data-bind="foreach: people">

<li>

Name at position <span data-bind="text: $index"> </span>:

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

<a href="#" data-bind="click: $parent.removePerson">Remove</a>

</li>

</ul>

<button data-bind="click: addPerson">Add</button>

Source code: View model

function AppViewModel() {

var self = this;

self.people = ko.observableArray([

{ name: 'Bert' },

{ name: 'Charles' },

{ name: 'Denise' }

]);

self.addPerson = function() {

self.people.push({ name: "New at " + new Date() });

};

self.removePerson = function() {

self.people.remove(this);

}

}

ko.applyBindings(new AppViewModel());

Parameters

Main parameter

Pass the array that you wish to iterate over. The binding will output a section of markup for each entry.

Alternatively, pass a JavaScript object literal with a property called data which is the array you wish to iterate over. The object literal may also have other properties, such as afterAdd or includeDestroyed — see below for details of these extra options and examples of their use.

If the array you supply is observable, the foreach binding will respond to any future changes in the array’s contents by adding or removing corresponding sections of markup in the DOM.

Additional parameters

None

Note 1: Referring to each array entry using $data

As shown in the above examples, bindings within the foreach block can refer to properties on the array entries. For example, Example 1 referenced the firstName and lastName properties on each array entry.

But what if you want to refer to the array entry itself (not just one of its properties)? In that case, you can use the special context property $data. Within a foreach block, it means “the current item”. For example,

<ul data-bind="foreach: months">

<li>

The current item is: <b data-bind="text: $data"></b>

</li>

</ul>

<script type="text/javascript">

ko.applyBindings({

months: [ 'Jan', 'Feb', 'Mar', 'etc' ]

});

</script>

If you wanted, you could use $data as a prefix when referencing properties on each entry. For example, you could rewrite part of Example 1 as follows:

<td data-bind="text: $data.firstName"></td>

… but you don’t have to, because firstName will be evaluated within the context of $data by default anyway.

Note 2: Using $index, $parent, and other context properties

As you can see from Example 2 above, it’s possible to use $index to refer to the zero-based index of the current array item. $index is an observable and is updated whenever the index of the item changes (e.g., if items are added to or removed from the array).

Similarly, you can use $parent to refer to data from outside the foreach, e.g.:

<h1 data-bind="text: blogPostTitle"></h1>

<ul data-bind="foreach: likes">

<li>

<b data-bind="text: name"></b> likes the blog post <b data-bind="text: $parent.blogPostTitle"></b>

</li>

</ul>

For more information about $index and other context properties such as $parent, see documentation for binding context properties.

Note 3: Using “as” to give an alias to “foreach” items

As described in Note 1, you can refer to each array entry using the $data context variable. In some cases though, it may be useful to give the current item a more descriptive name using the as option like:

<ul data-bind="foreach: { data: people, as: 'person' }"></ul>

Now anywhere inside this foreach loop, bindings will be able to refer to person to access the current array item, from the people array, that is being rendered. This can be especially useful in scenarios where you have nested foreach blocks and you need to refer to an item declared at a higher level in the hierarchy. For example:

<ul data-bind="foreach: { data: categories, as: 'category' }">

<li>

<ul data-bind="foreach: { data: items, as: 'item' }">

<li>

<span data-bind="text: category.name"></span>:

<span data-bind="text: item"></span>

</li>

</ul>

</li>

</ul>

<script>

var viewModel = {

categories: ko.observableArray([

{ name: 'Fruit', items: [ 'Apple', 'Orange', 'Banana' ] },

{ name: 'Vegetables', items: [ 'Celery', 'Corn', 'Spinach' ] }

])

};

ko.applyBindings(viewModel);

</script>

Tip: Remember to pass a string literal value to as (e.g., as: 'category', not as: category), because you are giving a name for a new variable, not reading the value of a variable that already exists.

Note 4: Using foreach without a container element

In some cases, you might want to duplicate a section of markup, but you don’t have any container element on which to put a foreach binding. For example, you might want to generate the following:

<ul>

<li class="header">Header item</li>

<!-- The following are generated dynamically from an array -->

<li>Item A</li>

<li>Item B</li>

<li>Item C</li>

</ul>

In this example, there isn’t anywhere to put a normal foreach binding. You can’t put it on the <ul> (because then you’d be duplicating the header item), nor can you put a further container inside the <ul> (because only <li> elements are allowed inside <ul>s).

To handle this, you can use the containerless control flow syntax, which is based on comment tags. For example,

<ul>

<li class="header">Header item</li>

<!-- ko foreach: myItems -->

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

<!-- /ko -->

</ul>

<script type="text/javascript">

ko.applyBindings({

myItems: [ 'A', 'B', 'C' ]

});

</script>

The <!-- ko --> and <!-- /ko --> comments act as start/end markers, defining a “virtual element” that contains the markup inside. Knockout understands this virtual element syntax and binds as if you had a real container element.

Note 5: How array changes are detected and handled

When you modify the contents of your model array (by adding, moving, or deleting its entries), the foreach binding uses an efficient differencing algorithm to figure out what has changed, so it can then update the DOM to match. This means it can handle arbitrary combinations of simulaneous changes.

When you add array entries, foreach will render new copies of your template and insert them into the existing DOM

When you delete array entries, foreach will simply remove the corresponding DOM elements

When you reorder array entries (retaining the same object instances), foreach will typically just move the corresponding DOM elements into their new position

Note that reordering detection is not guaranteed: to ensure the algorithm completes quickly, it is optimized to detect “simple” movements of small numbers of array entries. If the algorithm detects too many simultaneous reorderings combined with unrelated insertions and deletions, then for speed it can choose to regard a reordering as an “delete” plus an “add” instead of a single “move”, and in that case the corresponding DOM elements will be torn down and recreated. Most developers won’t encounter this edge case, and even if you do, the end-user experience will usually be identical.

Note 6: Destroyed entries are hidden by default

Sometimes you may want to mark an array entry as deleted, but without actually losing record of its existence. This is known as a non-destructive delete. For details of how to do this, see the destroy function on observableArray.

By default, the foreach binding will skip over (i.e., hide) any array entries that are marked as destroyed. If you want to show destroyed entries, use the includeDestroyed option. For example,

<div data-bind='foreach: { data: myArray, includeDestroyed: true }'>

...

</div>

Note 7: Post-processing or animating the generated DOM elements

If you need to run some further custom logic on the generated DOM elements, you can use any of the afterRender/afterAdd/beforeRemove/beforeMove/afterMove callbacks described below.

Note: These callbacks are only intended for triggering animations related to changes in a list. If your goal is actually to attach other behaviors to new DOM elements when they have been added (e.g., event handlers, or to activate third-party UI controls), then your work will be much easier if you implement that new behavior as a custom binding instead, because then you can use that behavior anywhere, independently of the foreach binding.

Here’s a trivial example that uses afterAdd to apply the classic “yellow fade” effect to newly-added items. It requires the jQuery Color plugin to enable animation of background colors.

<ul data-bind="foreach: { data: myItems, afterAdd: yellowFadeIn }">

<li data-bind="text: $data"></li>

</ul>

<button data-bind="click: addItem">Add</button>

<script type="text/javascript">

ko.applyBindings({

myItems: ko.observableArray([ 'A', 'B', 'C' ]),

yellowFadeIn: function(element, index, data) {

$(element).filter("li")

.animate({ backgroundColor: 'yellow' }, 200)

.animate({ backgroundColor: 'white' }, 800);

},

addItem: function() { this.myItems.push('New item'); }

});

</script>

Full details:

afterRender — is invoked each time the foreach block is duplicated and inserted into the document, both when foreach first initializes, and when new entries are added to the associated array later. Knockout will supply the following parameters to your callback:

An array of the inserted DOM elements

The data item against which they are being bound

afterAdd — is like afterRender, except it is invoked only when new entries are added to your array (and not when foreach first iterates over your array’s initial contents). A common use for afterAdd is to call a method such as jQuery’s $(domNode).fadeIn() so that you get animated transitions whenever items are added. Knockout will supply the following parameters to your callback:

A DOM node being added to the document

The index of the added array element

The added array element

beforeRemove — is invoked when an array item has been removed, but before the corresponding DOM nodes have been removed. If you specify a beforeRemove callback, then it becomes your responsibility to remove the DOM nodes. The obvious use case here is calling something like jQuery’s $(domNode).fadeOut() to animate the removal of the corresponding DOM nodes — in this case, Knockout cannot know how soon it is allowed to physically remove the DOM nodes (who knows how long your animation will take?), so it is up to you to remove them. Knockout will supply the following parameters to your callback:

A DOM node that you should remove

The index of the removed array element

The removed array element

beforeMove — is invoked when an array item has changed position in the array, but before the corresponding DOM nodes have been moved. Note that beforeMove applies to all array elements whose indexes have changed, so if you insert a new item at the beginning of an array, then the callback (if specified) will fire for all other elements, since their index position has increased by one. You could use beforeMove to store the original screen coordinates of the affected elements so that you can animate their movements in the afterMove callback. Knockout will supply the following parameters to your callback:

A DOM node that may be about to move

The index of the moved array element

The moved array element

afterMove — is invoked after an array item has changed position in the array, and after foreach has updated the DOM to match. Note that afterMove applies to all array elements whose indexes have changed, so if you insert a new item at the beginning of an array, then the callback (if specified) will fire for all other elements, since their index position has increased by one. Knockout will supply the following parameters to your callback:

A DOM node that may have moved

The index of the moved array element

The moved array element

For examples of afterAdd and beforeRemove see animated transitions.

Dependencies

None, other than the core Knockout library