Animating Knockout

Ryan Hodson on Jan 12th 2013 with 0 Comments

Tutorial Details

-
- **Difficulty:** Intermediate
- **Completion Time:** 30 Minutes

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

*This entry is part 9 of 9 in the **Knockout Succinctly** Session - Show All*

« Previous

Knockout.js is *not* an animation library. All of Knockout.js' automatic updates are *immediately* applied whenever the underlying data changes. In order to animate any of its changes, we need to dig into Knockout.js' internals and manually create animated transitions using another JavaScript framework like jQuery or MooTools. This lesson sticks with jQuery's animation routines, but the concepts presented apply to other animation libraries as well.

Return of the Shopping Cart

For this lesson, we’ll return to a simplified version of our shopping cart example. Create a new HTML file with the following contents. We won’t be making any AJAX requests, so feel free to put this anywhere on your computer. We will, however, be using jQuery’s animation routines, so be sure to include a link to your copy of the jQuery library.

```
<html lang='en'>
<head>
  <title>Animating Knockout.js</title>
  <meta charset='utf-8' />
  <link rel='stylesheet' href='style.css' />
</head>
<body>
  <h2>
  <table>
    <thead><tr>
      <th>Product</th>
      <th>Price</th>
      <th></th>
    </tr></thead>
    <tbody data-bind='foreach: items'>
      <tr>
        <td data-bind='text: name'></td>
        <td data-bind='text: price'></td>
        <td><button data-bind='click: $root.removeProduct'>Remove<
      </tr>
    </tbody>
  </table>

  <button data-bind='click: addProduct'>Add Beer</button>

  <script src='knockout-2.1.0.js'></script>
  <script src='jquery-1.7.2.js'></script>
  <script>
    function Product(name, price, tags, discount, details) {
      this.name = ko.observable(name);
      this.price = ko.observable(price);
    }
    function ShoppingCart() {
      var self = this;
      this.instructions = ko.observable("");
      this.hasInstructions = ko.observable(false);

      this.items = ko.observableArray([
        new Product("Beer", 10.99),
        new Product("Brats", 7.99),
        new Product("Buns", 1.49)
      ]);
```

```

        this.addProduct = function() {
          this.items.push(new Product("More Beer", 10.99));
        };

        this.removeProduct = function(product) {
          self.items.destroy(product);
        };

      };
      ko.applyBindings(new ShoppingCart());
    </script>
  </body>
  </html>
```

Hopefully, this is all review by now. We have an observable array containing a bunch of products, a `foreach` binding that displays each one of them, and a button to add more items to the shopping cart.

List Callbacks

Knockout.js is a powerful user interface library on its own, but once you combine it with the animation capabilities of a framework like jQuery or MooTools, you're ready to create truly stunning UIs with minimal markup. First, we'll take a look at animating lists, and then the next section presents a more generic way to animate view components.

The `foreach` binding has two callbacks named `beforeRemove` and afterAdd. These functions are executed before an item is removed from the list or after it's been added to the list, respectively. This gives us an opportunity to animate each item before Knockout.js manipulates the DOM. Add the callbacks to the <tbody> element like so:

```
<tbody data-bind='foreach: {data: items,
     beforeRemove: hideProduct,
     afterAdd: showProduct}'>
```

Instead of a property, our `foreach` binding now takes an object literal as its parameter. The parameter's `data` property points to the array you would like to render, and the beforeRemove and afterAdd properties point to the desired callback functions. Next, we should define these callbacks on the ShoppingCart ViewModel:

```
this.showProduct = function(element) {
  if (element.nodeType === 1) {
    $(element).hide().fadeIn();
  }
};

this.hideProduct = function(element) {
  if (element.nodeType === 1) {
   $(element).fadeOut(function() { $(element).remove(); });
  }
};
```

The `showProduct()` callback uses jQuery to make new list items gradually fade in, and the `hideProduct()` callback fades them out, and then removes them from the DOM. Both functions take the affected DOM element as their first parameter (in this case, it's a <tr> element). The conditional statements make sure that we're working with a full-fledged element and not a mere text node.

The end result should be list items that smoothly transition into and out of the list. Of course, you're free to use any of jQuery's other transitions or perform custom post-processing in either of the callbacks.

Custom Bindings

The `foreach` callbacks work great for animating lists, but unfortunately other bindings don't provide this functionality. So, if we want to animate other parts of the user interface, we have to create *custom* bindings that have the animation built right into them.

Custom bindings work just like Knockout.js' default bindings. For example, consider the following form fields:

```
<div>
  <p>
    <input data-bind='checked: hasInstructions'
           type='checkbox' />
    Requires special handling instructions
  </p>
<div>

<textarea data-bind='visible: hasInstructions,
                         value: instructions'>
</textarea>
```

The check box acts as a toggle for the `<textarea>`, but since we're using the `visible` binding, Knockout.js abruptly adds or removes it from the DOM. To provide a smooth transition for the `<textarea>`, we'll create a custom binding called visibleFade:

```
<textarea data-bind='visibleFade: hasInstructions,
                           value: instructions'>
```

Of course, this won't work until we add the custom binding to Knockout.js. We can do this by adding an object defining the binding to `ko.bindingHandlers` as shown in the following code sample. This also happens to be where all of the built-in bindings are defined, too.

```
ko.bindingHandlers.visibleFade = {
  init: function(element, valueAccessor) {
    var value = valueAccessor();
    $(element).toggle(value());
  },
  update: function(element, valueAccessor) {
    var value = valueAccessor();
    value() ? $(element).fadeIn() : $(element).fadeOut();
  }
}
```

The `init` property specifies a function to call when Knockout.js first encounters the binding. This callback should define the initial state for the view component and perform necessary setup actions (e.g., registering event listeners). For `visibleFade`, all we have to do is show or hide the element based on the state of the ViewModel. We implemented this using jQuery's toggle() method.

The `element` parameter is the DOM element being bound, and `valueAccessor` is a function that will return the ViewModel property in question. In our example, element refers to <textarea>, and valueAccessor() returns a reference to the hasInstructions observable.

The `update` property specifies a function to execute whenever the associated observable changes, and our callback uses the value of `hasInstructions` to transition the <textarea> in the appropriate direction. Remember that you need to call the observable to get its current value (i.e. value(), not value). However, if hasInstructions were a normal JavaScript property instead of an observable, this would not be the case.

Summary

In this lesson, we discovered two methods of animating Knockout.js view components. First, we added callback methods to the `foreach` binding, which let us delegate the addition and removal of items to a user-defined function. This gave us the opportunity to integrate jQuery's animated transitions into our Knockout.js template. Then, we explored custom bindings as a means to animate arbitrary elements.

This lesson presented a common use case for custom bindings, but they are by no means limited to animating UI components. Custom bindings can also be used to filter data as it is collected, listen for custom events, or create reusable widgets like grids and paged content. If you can encapsulate a behavior into an `init` and an update function, you can turn it into a custom binding.

Conclusion

> This series covered the vast majority of the Knockout.

Knockout.js is a pure JavaScript library that makes it incredibly easy to build dynamic, data–centric user interfaces. We learned how to expose ViewModel properties using observables, bind HTML elements to those observables, manage user input with interactive bindings, export that data to a server–side script, and animate components with custom bindings. Hopefully, you’re more than ready to migrate this knowledge to your real–world web applications.

This series covered the vast majority of the Knockout.js API, but there are still a number of nuances left to discover. These topics include: custom bindings for aggregate data types, the `throttle` extender for asynchronous evaluation of computed observables, and manually subscribing to an observable’s events. However, all of these are advanced topics that shouldn’t be necessary for the typical web application. Nonetheless, Knockout.js provides a plethora of extensibility opportunities for you to explore.

> If you’d prefer to re–read this session in this book">eBook form, be sure to check out Syncfusion’s website. Additionally, they offer a variety of free eBooks, just like this one!

> This lesson represents a chapter from *Knockout Succinctly*, a free eBook from the team at Syncfusion.

Tags: knockout

By Ryan Hodson

Ryan Hodson has worked in a diverse collection of programming fields, building everything from websites to e-publishing platforms, touch-screen thermostats, and natural language processing tools. These experiences have led to a love of exploring new software and a proficiency in several languages and frameworks.In 2012, Ryan founded an independent publishing firm called RyPress and published his first book, Ry's Friendly Guide to Git. Since then, he has worked as a freelance technical writer for well-known software companies, including Syncfusion and Atlassian. Ryan continues to publish high-quality software tutorials via RyPress.com.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more