**Step 1**

In this first tutorial you'll experience some of the basics of building a web UI with the *Model-View-ViewModel* (MVVM) pattern using knockout.js.

You'll learn how to define a UI's appearance using **views** and **declarative bindings**, its data and behavior using **viewmodels** and **observables**, and how everything stays in sync automatically thanks to Knockout's**dependency tracking** (even with arbitrary cascading chains of data).

**Using bindings in the view**

In the bottom-right corner, you've got a *viewmodel* containing data about a person. In the top-right corner, you've got a *view* that's supposed to display the person data. Right now it just displays "*todo*", so let's fix that.

Modify the two **<strong>** elements in the view, adding **data-bind**attributes to display the person's name:

<**p**>First name: <**strong** data-bind="text: firstName"></**strong**></**p**> <**p**>Last name: <**strong** data-bind="text: lastName"></**strong**></**p**>

**data-bind** attributes are how Knockout lets you declaratively associate viewmodel properties with DOM elements. You just used the **text** binding to assign text to your DOM elements.

**Step 2**

**Making the data editable**

You're not limited to displaying static data. Let's use the **value** binding, along with some regular HTML **<input>** controls, to make the data editable.

Add the following markup to the bottom of your view (leaving the existing markup in place above it):

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

Now run the application. What happens when you edit the text in one of the text boxes?

Hmm... apparently nothing happens. Let's fix that...

**Introducing Observables**

Actually, when you edit one of those text boxes, it *does* update the underlying viewmodel data. But because the viewmodel properties are just plain JavaScript strings, they have no way of notifying anybody that they've changed, so the UI stays static. That's why Knockout has a concept of**observables** - these are properties that automatically will issue notifications whenever their value changes.

Update your viewmodel to make the **firstName** and **lastName** properties*observable* using **ko.observable**:

**function** **AppViewModel**() { **this**.firstName = ko.observable("Bert"); **this**.lastName = ko.observable("Bertington"); }

**Step 3**

**Defining computed values**

Very often, you'll want to combine or convert multiple observable values to make others. In this example, you might want to define *full name* as being*first name* plus *space* plus *last name*.

To handle this, Knockout has a concept of **computed properties** - these are *observable* (i.e., they notify on change) and they are *computed* based on the values of other observables.

Add a **fullName** property to your view model, by adding the following code inside **AppViewModel**, after **firstName** and **lastName** are declared:

**this**.fullName = ko.computed(**function**() { **return** **this**.firstName() + " " + **this**.lastName(); }, **this**);

As you can see, we're passing a callback function to the **ko.computed**which specifies how it should compute its value. Next, display the**fullName** value in your UI by adding markup at the bottom of your view:

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

If you run the application now and edit the text boxes, you'll see that all UI elements (including the full name display) stay in sync with the underlying data.

**How does it work?**

Things stay in sync because of automatic dependency tracking: the last**<strong>** there depends on **fullName**, which in turn depends on**firstName** and **lastName**, and either of those can be altered by editing the textboxes. Any changes ripple out through the object graph causing the minimal set of refreshes needed to bring your viewmodel and visible UI up-to-date.

**Step 4**

**Adding more behavior**

To finish this example, let's add one final behavior: a button that makes the "last name" value turn upper-case.

**Updating the viewmodel**

First, add a **capitalizeLastName** function to the viewmodel to implement this behavior:

**function** **AppViewModel**() { // ... leave firstName, lastName, and fullName unchanged here ... **this**.capitalizeLastName = **function**() { **var** currentVal = **this**.lastName(); // Read the current value **this**.lastName(currentVal.toUpperCase()); // Write back a modified value }; }

Notice that, to read or write an observable's value, you call it as a function.

**Updating the view**

Next, add a button to the view, using the **click** binding to associate clicks with the viewmodel function you just added:

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

**Good job!**

This was a very basic example, but it did illustrate some of the key points of MVVM:

* You've got a clean, object-oriented representation of your UI's data and behaviors (your viewmodel)
* Separately, you've got a declarative representation of how it should be displayed visibly (your view)
* You can implement arbitrarily sophisticated behaviors just by updating the viewmodel object. You *don't* have to worry about which DOM elements need to be changed/added/removed - the framework can take care of synchronizing things for you.