Mastering KnockoutJS

Chapter No. 1
"Knockout Essentials"
In this package, you will find:

The author’s biography
A preview chapter from the book, Chapter no.1 "Knockout Essentials"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Timothy Moran has been working in the field of software for the last 4 years. He started with desktop development for .NET and moved on to web development for a variety of technologies. Timothy began using Knockout shortly after the release of Version 1.3 and has since used it in several projects personally and professionally. He also provides community support by answering questions on StackOverflow.

This is Timothy's first title.
Mastering KnockoutJS

Knockout is built around a pattern that started in Microsoft. This model is Model- View- ViewModel (MVVM), and I think introducing this pattern to newcomers is one of the biggest obstacles to wider adoption. Nearly every other JavaScript library or framework, along with most server-side frameworks, has been built around the Model-View-Controller (MVC) pattern, and the differences between the two are sometimes confusing, even for experienced developers. This problem is compounded by the fact that some larger frameworks, such as AngularJS, end up with a pattern that is nearly identical to MVVM.

Knockout's documentation is excellent, and its live examples and interactive tutorials are some of the best. When it comes to organizing full applications, though, more explanation is required. When I started writing this book, there was only one book on Knockout on Amazon, and it didn't have very favorable reviews. It seemed like a complete guide to using Knockout as the central piece of a frontend stack was missing.

I have been using Knockout for 3 years now, and I've been an active member of the community on StackOverflow and GitHub for 2 years. I've used Knockout in several professional applications as well as a dozen or so personal projects. It is, by far, my favorite JavaScript library, and I strongly prefer MVVM over MVC for developing client applications. Hopefully, you feel that this book gives you everything you need in order to be successful with Knockout.

A note on Knockout 3.2

Knockout 3.2 was released while this book was being written. Chapter 4, Application Development with Components and Modules, was rewritten to include the components feature, and some minor changes were made to other chapters in order to make them accurate. However, most of the code samples were written against Knockout 3.1, and so they do not take advantage of pure computed observables or other features that were released in Knockout 3.2.

What This Book Covers

Chapter 1, Knockout Essentials, covers the environment setup and basic use of the Knockout library. It also covers data binding, observables, binding handlers, and extenders, and demonstrates a simple Knockout Contacts List application.

Chapter 2, Extending Knockout with Custom Binding Handlers, gives you in-depth knowledge of how to create and use custom binding handlers. It includes simple single-property binding handlers as well as complex multiproperty binding handlers with templates.

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
Chapter 3, *Extending Knockout with Preprocessors and Providers*, teaches you how to use node and binding preprocessors and binding providers to customize Knockout's syntax. It also explores the Knockout Punches library.

Chapter 4, *Application Development with Components and Modules*, explains how to use RequireJS Asynchronous Module Definitions (AMDs) with Knockout to create organized, modular viewmodels. It also teaches you how to use the new Knockout components feature and how to continue working with the Contacts List demo application.

Chapter 5, *Durandal – the Knockout Framework*, explores the basics of the Knockout-based Durandal framework. This chapter covers composition, routing, modal dialogs, and custom widgets.

Chapter 6, *Advanced Durandal*, continues looking at the use of the Durandal framework. This chapter covers events, advanced composition, nested routers, custom dialogs, and the observable plugin.

Chapter 7, *Best Practices*, takes a deep dive into the inner workings of Knockout. It includes dependency detection and the publish/subscribe implementation, observable inheritance, the template engine, and a complete Knockout utility (ko.utils) reference.

Chapter 8, *Plugins and Other Knockout Libraries*, gives you an overview of the recommended patterns and practices for Knockout developers.

Chapter 9, *Under the Hood*, covers several popular Knockout plugins, including Knockout validation, Knockout mapping, and the new Knockout-ES5 plugin.

For More Information:  
www.packtpub.com/web-development/mastering-knockoutjs
Though it is expected that you have experience with both JavaScript and KnockoutJS, we will still be covering the basics to establish a common foundation. This book wouldn't be complete if we didn't cover at least the basics. After that, we will look at building a simple application to create and manage contact information. This application will be used throughout the book to explore new techniques in Knockout and see how they fit into the larger process of application development. In this chapter, you will learn how to:

- Define viewmodels
- Write standard bindings
- Use extenders
- Use templates
- Put all these pieces together into a functional application

This covers most of the standard functionalities in Knockout. In the next chapter, we will look at creating our own bindings to extend Knockout.

Even if you have used Knockout before and don't think you need a refresher, I encourage you to at least read the section that covers the Contacts List application example. It's something we will be using throughout the book as we explore more advanced concepts.

Before we get started, let's get our development environment set up.
The environment setup

We will be using a simple Node.js server to host our application because it will run on any operating system. If you haven't done so, install Node.js by following the instructions at http://nodejs.org/download.

We will be using Git to manage the code for each chapter. If you haven't done so, install Git by following the instructions at http://git-scm.com/book/en/Getting-Started-Installing-Git. The code for this book can be downloaded from http://www.packtpub.com. All the code needed to start each chapter can be found in a branch named cp[chapter#]-[sample]. For example, the first sample we will look at is going to be in the cp1-computeds branch.

To begin, clone the repository from https://github.com/tyrsius/MasteringKnockout. You can either use the provided download links or run the following command:

git clone git@github.com:tyrsius/MasteringKnockout

Then, check out the first sample using:

git checkout cp1

All the examples follow the same pattern. At the root is a server.js file that contains a boilerplate Node.js server. Inside the client directory is all the code for the application. To run the application, run this from the command line:

node server.js

Keep the command-line window open else the server will stop running. Then, open your web browser and navigate to http://localhost:3000. If you've set up your environment correctly, you should be looking at the empty Contacts List application, as shown in the following screenshot:

![Contacts List screenshot]

The cp1 branch contains a skeleton with some blank pages. Until we get to the Contacts application, most of the samples will not have the Contacts or Settings pages; they will just present the code on the home page.
Looking at the samples

Samples of running code are provided throughout the book. They are in branches in the Git repository. You can look at them by checking out the branch, using the following command:

```
git checkout [BranchName]
```

Since the repository is a functional app, most of the code is not relevant to the samples. The `client` directory contains the `index.html` and `shell.html` pages, as well as the `app`, `content`, and `lib` directories. The `app` directory is where our JavaScript is located. The `content` directory contains the included CSS and `lib` contains third-party code (Knockout, jQuery, and Twitter Bootstrap).

The included Node server has a very simple view composition that places the contents of a page in the `{{ body }}` section of the shell. If you have worked with any server-side MVC frameworks, such as Ruby on Rails or ASP.NET MVC, you will be familiar with this. The mechanism is not related to Knockout, but it will help us keep our code separated as we add files. The shell is in the `shell.html` file. You can take a look at it, but it's not directly related to the samples. The HTML for samples is in the `client/index.html` file. The JavaScript for samples is in the `client/app/sample.js` file.

JavaScript's compatibility

Throughout this book, we will be using code that relies on ECMAScript 5 features, which are supported on all modern browsers. I encourage you to run these examples using a compatible browser. If you cannot, or if you are interested in running them in an older environment, you can use a polyfill for them. A polyfill is a JavaScript library that adds standard features to old environments to allow them to run modern code. For the ECMAScript 5 functions, I recommend Sugar.js. For the CSS3 media query support, I recommend Respond.js.

An overview of Knockout

Knockout is a library designed for Model-View-ViewModel (MVVM) development. This pattern, a descendant of Martin Fowler's Presentation model, encourages the separation of User Interface (UI) from the business logic of the domain model. To facilitate this separation, Knockout provides the three necessary components for implementing this pattern, namely, a declarative syntax for the view (the data-bind HTML attribute), a mechanism to notify changes from the viewmodel (the observable object), and a data binder to mediate between the two (Knockout's binding handler).
Knockout Essentials

We will be covering the data-bind and observable object syntax here; the binding handler syntax and its use will be covered in the next chapter.

Using the MVVM pattern means your viewmodel operates on data with JavaScript, and your HTML view is described using the declarative data-binding syntax. Your JavaScript code should not be directly accessing or modifying the view — data-binding should handle that by translating your observable objects into HTML using binding handlers.

The best way to think about the separation between view and viewmodel is to consider whether two different views could use your viewmodel. While this is often not done, it is still helpful to keep it in mind because it forces you to maintain the separation between them. MVVM allows you to redesign the view without affecting the viewmodel.

Observables

Knockout follows a publish/subscribe pattern to keep data in sync between different parts of the application, such as the UI and the viewmodel. The publisher in Knockout is the observable object. If you've used MVVM before in Windows Presentation Foundation (WPF) development, then observable objects can be thought of as Knockout's INotifyPropertyChanged implementation.

To construct an observable, the observable function is called on the global ko object:

```javascript
this.property = ko.observable('default value');
```

The observable function returns a new observable. If ko.observable is called with a value, it returns an observable with that value.

The reason why Knockout observables are JavaScript functions instead of normal properties is to allow support for older browsers such as Internet Explorer 6, which did not support getters and setters on properties. Without that ability, setting properties would have no mechanism to notify subscribers about changes.

Observables are JavaScript functions that record subscribers reading their value, then call these subscribers when the value has been changed. This is done using Knockout's dependency tracking mechanism.

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
Observables are read by calling them without any parameters. To write to an observable, call it with the value as the first and only parameter (further parameters are ignored):

```javascript
var total = vm.total(); // read value
vm.total(50); // write new value
```

### Observables can contain any legal JavaScript value: primitives, arrays, objects, functions, and even other observables (though this wouldn't be that useful). It doesn't matter what the value is; observables merely provide a mechanism to report when that value has been changed.

### Observable arrays

Though standard observables can contain arrays, they aren't well suited to track changes in them. This is because the observable is looking for changes in the value of the array, a reference to the array itself, which is not affected by adding or removing elements. As this is what most people expect change notification to look like on an array, Knockout provides the `observableArray`:

```javascript
this.users = ko.observableArray(myUsers);
```

Like observables, arrays can be constructed with an initial value. Normally, you access an observable by calling it or setting its value by passing it a parameter. With observable arrays it's a little different. As the value of the array is its reference, setting that value would change the entire array. Instead, you usually want to operate on the array by adding or removing elements. Consider the following action:

```javascript
this.users().push(new User("Tim"));
```

By calling `this.users()`, the underlying array is retrieved before a new user is pushed to it. In this case, Knockout is not aware that the array was changed, as the change was made to the array itself and not the observable. To allow Knockout to properly track changes, these changes need to be made to the observable, not the underlying value.
To do this, Knockout provides the standard array methods on the observable, namely, `push`, `pop`, `shift`, `unshift`, `sort`, `reverse`, and `splice`. The call should look like this:

```javascript
this.users.push(new User("Tim");
```

Notice that instead of retrieving the array from the observable, we are calling `push` directly on the observable. This will ensure that subscribers are notified of the change with an updated array.

## Computed observables

Computed observables are properties that are set manually, either through your code or by bindings from the UI. Computed observables are properties that automatically update their value by responding to changes in their dependencies, as shown in the following code:

```javascript
var subtotal = ko.observable(0);
var tax = ko.observable(0.05);
var total = ko.computed(function() {
    return parseFloat(subtotal()) * (1 + parseFloat(tax()));
});
```

In this example, `subtotal` and `tax` are the dependencies of the `total` computed observable. For the first time, the computed observable calculates records of any other observables that were accessed and creates a subscription for them. The result is that whenever `subtotal` or `tax` are changed, the `total` is recalculated and notified to its subscribers. It helps to think of computed observables as declarative values; you define their value as a formula and they will keep themselves up to date.

The `parseFloat` calls are to ensure that they are treated as numbers instead of strings, which would cause concatenation instead of arithmetic. As Knockout binds data against HTML attributes, which are always strings, updates from data binding produce strings. When we discuss extenders, you will see another way to manage this issue.
You can see a sample of this on the `cp1-computeds` branch:

![Computed Observable Sample](Image)

Try changing some of the numbers and watch the `total` computed value update automatically. You can see that the viewmodel code contains just this sample by looking in the `client/app/sample.js` file.

**Writable computed observables**

The preceding `total` example is a read-only computed. While they are less common, it is also possible to make a computed observable writable. To do so, pass an object with a read and write function to `ko.computed`:

```javascript
var subtotal = ko.observable(0);
var tax = ko.observable(0.05);
var total  = ko.computed({
    write: function(newValue) {
        subtotal(newValue / (1 + parseFloat(self.tax())));
    },
    read: function() {
        parseFloat(subtotal()) * (1 + parseFloat(tax()));
    }
});
```

When something attempts to write to the `total` computed now, it will cause the `subtotal` observable to be updated by the `write` function. This is a very powerful technique, but it is not always necessary. In some cases being unable to write directly to `total` might be a good thing, such as when `total` might involve conditionally applying tax to a list of items. You should use writeable computeds only when it makes sense to do so.

You can see an example of this in the `cp1-writecomputed` branch. The `total` computed is now bound to an `input` element such as the `subtotal` and `tax` properties, and changes to the value will reflect back into the `subtotal` observable.

For More Information:  
Pure computed observables

Nonpure computed observables re-evaluate themselves whenever any of their dependencies change, even if there are no subscribers to receive the updated value. This re-evaluation can be useful if the computed also has intentional side effects, but it wastes memory and the processor's cycles if it has no side effects. Pure computed observables, on the other hand, do not re-evaluate when there are no subscribers.

Pure computed observables have two states: listening and sleeping. When a pure computed has subscribers, it will be listening and behaving exactly like a normal computed. When a pure computed has no subscribers, it will enter its sleeping state and dispose off all of its dependency subscriptions. When it wakes up, the pure computed will re-evaluate itself to ensure its value is correct.

Pure computed observables are useful when a value may go unused for an extended period of time, as they do not re-evaluate. However, since a pure computed always re-evaluates when accessed from a sleeping state, it can sometimes perform worse than a normal computed observable. Since normal computeds only re-evaluate when their dependencies change, a computed observable that is frequently woken from a sleeping state could potentially evaluate its dependencies more often.

There are two ways to create a pure computed: by using ko.pureComputed or by passing { pure: true } as the third parameter to ko.computed:

```javascript
var total = ko.pureComputed(function() {
    return parseFloat(subtotal()) * (1 + parseFloat(tax()));
});
//OR
var total = ko.computed(function() {
    return parseFloat(subtotal()) * (1 + parseFloat(tax()));
}, this, { pure: true });
```

Pure computed observables were introduced in Knockout 3.2, which was not released at the time this book was written. None of the code samples take advantage of pure computed observables, even though many of the samples would have benefited from them.

Manual subscriptions

Sometimes you need to do more than update a dependent value when an observable changes, such as make a web request for additional data based on the new value of your observable. Observables provide a subscribe function that lets you register a function to be called when the observable is updated.

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
Subscriptions use the same internal mechanism in Knockout that binding handlers and computed observables use to receive changes.

This is an example of setting up a subscription on an observable:

```javascript
var locationId = ko.observable();
locationId.subscribe(function (newLocationId) {
    webService.getLocationDetails(newLocationId);
});
```

This subscription will be called any time when the `locationId` is updated, whether it happens from a UI binding or from somewhere else in JavaScript.

The `subscribe` function also allows you to provide a target for the subscription and the name of the event you want to subscribe to. The target is the value of `this` for the subscription handler you provide. The event defaults to change, which receives the value after it has been updated, but can also be `beforeChange`, which is called with the old value before a change happens:

```javascript
locationId.subscribe(function (oldValue) {
    console.log("the location " + oldValue + " is about to change");
}, self, 'beforeChange');
```

Finally, you can stop a subscription from continuing to fire by capturing it and calling `dispose`. This can be useful if you want to stop the handler or to make subscriptions that only fire a single time:

```javascript
var subscription = locationId.subscribe(function (newValue) {
    console.log("the location " + oldValue + " is about to change");
    subscription.dispose();
});
```

Once a subscription has been disposed, it cannot be restarted. If you need it, you will have to recreate the subscription.

The `cp1-subscribe` branch has a subscription example that logs any changes to the `subtotal` observable on the JavaScript console, as well as a button that stops the subscription. Try changing the subtotal or total value and watch out for the console messages. Changing the total causes an update of the subtotal, which is why it still fires the subscription. Remember, changes from any source will cause an observable to report changes to all of its subscribers. This is the same reason updating the `total` computed causes the `subtotal` observable's input element to update; the input element is a subscriber to the viewmodel's property.

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
Defining viewmodels

Viewmodels are the objects whose properties your view binds with; they form the binding context. It is the representation of your data and operations for your view (we will cover them in detail in the Control flow bindings section later in this chapter). Like regular objects in JavaScript, there are many ways to actually create them, but Knockout introduces some specific challenges.

The this and self keywords

In JavaScript, this has a special meaning; it refers to the object calling the function. Functions called from an object get that object set to this. However, for functions that are anonymously called by code, that is merely the inside of an object, the behavior is different. Consider the following viewmodel:

```javascript
function Invoice() {
    this.subtotal = ko.observable();
    this.total = ko.computed(function() {
        return this.subtotal() * 1.08; //Tax Rate
    });
}
```

The function inside the computed observable is not a property of the Invoice object. As it runs in a different context, its value for this will be the window object, not the Invoice object. It will not be able to find the subtotal property. There are two ways to handle this.

The first is by using the second parameter of the ko.computed function to bind the function to this:

```javascript
function Invoice() {
    this.subtotal = ko.observable();
    this.total = ko.computed(function() {
        return this.subtotal() * 1.08; //Tax Rate
    }, this);
}
```

This gives the computed observable a reference to the Invoice that originally defined it, which allows the computed observable to call the supplied function in the correct context.

The second way to ensure the computed observable can reference the subtotal, is to capture the value of this in a closure. You can then use the closure to safely refer to the properties of the parent viewmodel. There are several conventional names for such a closure: that, _this, or self.

For More Information:

www.packtpub.com/web-development/mastering-knockoutjs
I prefer to use self as it is visually distinct from this while still carrying a similar meaning, but it's up to you:

```javascript
function Invoice() {
  var self = this;
  self.subtotal = ko.observable();
  self.total = ko.computed(function() {
    return self.subtotal() * 1.08; //Tax Rate
  });
}
```

I find the second method easier to remember. If you always use self to refer to the model, it will always work. If you have another anonymous function inside the computed, you will have to remember to bind that function as well; self continues to work as a closure no matter how many levels deep you nest. The self variable works as a closure inside any function defined in your viewmodel, including subscriptions. It's also easier to spot when self isn't being used, which is very helpful while debugging your code.

**Problems with prototypes**

If you are working with viewmodels that will be inherited by other viewmodels, you might think that putting all the base observable properties on the prototype is the way to go. In vanilla JavaScript, if you are inheriting an object, try to change the value of a property stored on the prototype; the property would be added to the inheriting object leaving the prototype intact. When using observables in Knockout though, this isn't the case. The observables are functions, and their values are set by calling them with a single parameter, not by assigning new values to them. Because prototypical inheritance would result in multiple objects referring to a single observable; observables cannot be safely placed on viewmodel prototypes. Nonobservable functions can still be safely included in prototypes. For example, consider the following objects:

```javascript
var protoVm = {
    name: ko.observable('New User')
};

var base1 = Object.create(protoVm);
var base2 = Object.create(protoVm);

base2.name("Base2");
```
The last line will cause the name of both objects to be updated, as it is referring to the same function. This example can be seen in the cp1-prototype branch, which includes two input elements bound to the name of each viewmodel. As they are really the same observable, changing one will affect the other.

**Serializing viewmodels**

When you are ready to send your viewmodels to the server, or really do anything that requires you to work with their values instead of observables, Knockout provides two very handy utility methods:

- **ko.toJS**: This function takes an object and does a deep copy, unwrapping all observables, into a new JavaScript object whose properties are normal (nonobservable) JavaScript values. This function is perfect to get copies of viewmodels.
- **ko.toJSON**: This function uses the output from ko.toJS with JSON.stringify to produce a JSON string of the supplied object. This function accepts the same parameters as JSON.stringify.

**The data-bind syntax**

Knockout takes advantage of the HTML5 data-* attribute specification to define its data-bind attribute. Though all HTML attributes are necessarily strings, Knockout parses them as name:value pairs. The name refers to the binding handler to be used and the value refers to the value the binding will use:

```html
<button data-bind="enable: canSave">Save</button>
```

The data-bind attribute can also contain multiple bindings separated by commas. This allows multiple properties to be bound on an element:

```html
<input data-bind="value: firstName, enable: canEdit" />
```

In the preceding example, the enable binding uses canEdit as a value. The binding will set the disabled attribute on the button element when canEdit is false, and remove the disabled attribute when canEdit is true. If canEdit is an observable, the enable binding will update whenever canEdit is updated. If canEdit is a literal value, such as true, it will only use the value to set the initial state.

For More Information:

Enable is a **one-way binding**; it will update the element with changes from the value but it will not update the value with changes from the element. This is because when enable is being used to control the element, Knockout assumes that nothing will be programmatically updating the element. Updates should happen in the viewmodel, and binding handlers should be responsible for ensuring the view is kept in sync.

When users update the UI of data-bound input elements, those changes need to be synced to the viewmodel. This is done with **two-way bindings**, such as the value binding:

```html
<input data-bind="value: firstName" />
```

This binding will set the initial value of the `input` element to the current value of the `firstName` property, and after that, it will ensure that any changes to either the element's value or the property cause the other to update. If the user types something into the input, the `firstName` property will receive the value. If the `firstName` property is updated programmatically, the input's value will be updated.

These are both examples of binding against a simple property on the viewmodel. This is the most common case, but Knockout supports more complex scenarios as well.

For a complete list of the standard Knockout binding handlers, see the Knockout documentation ([http://knockoutjs.com/documentation/introduction.html](http://knockoutjs.com/documentation/introduction.html)).

### Binding with nested properties

In the previous example, Knockout parsed the binding value for the name of a property and looked for that property on the current viewmodel. You can also provide deep property references. Consider the following object:

```javascript
var viewmodel = {
  user: {
    firstName: ko.observable('Tim'),
    age: ko.observable(27)
  }
};
```

We can bind directly against the `firstName` property of the viewmodel's user by using standard dot notation:

```html
<input data-bind="value: user.firstName" />
```

---

For More Information:  
www.packtpub.com/web-development/mastering-knockoutjs
**Binding against functions**

If you are using the `click` or `event` bindings to bind some UI event, the binding expects the property to be a function. Functions will receive the current model (the binding context) as their first parameter, and the JavaScript event as the second parameter (though you shouldn't need to do this very often).

In this example, the parent viewmodel receives the contact to be removed from the `click` binding because the `foreach` loop creates a nested binding context for each contact. The parent reference in the binding moves the context up to the parent viewmodel to get access to the remove function:

```html
<ul data-bind="foreach: contacts">
  <li>
    <span data-bind="text: name"></span> 
    <button data-bind="click: $parent.remove">Remove</button>
  </li>
</ul>
```

```javascript
var ViewModel = function() {
  var self = this;
  self.contacts = ko.observableArray([{ name: 'Tim' }, { name: 'Bob' }]);
  self.remove = function (contact) {
    self.contacts.remove(contact);
  }
};
```

**Binding with expressions**

In addition to property references, Knockout also supports the use of JavaScript expressions as binding values. For bindings that expect true or false values, such as `enable`, we can use Boolean expressions to set them:

```html
<button data-bind="enable: age > 18">Approve</button>
```

We can also use ternary expressions to control the result of the expression. This is useful in cases where Booleans are not expected, such as text bindings:

```html
Old enough to Drink in the U.S.
<span data-bind="text: age > 18 ? 'Yes' : 'No'" ></span>
```

Now the span will have `Yes` as content.

---

**For More Information:**

Both forms of expressions will use dependency tracking to rerun if they read from an observable the first time they are run. If age was an observable value, we could update it and the element’s binding would re-evaluate the expression, changing the text or enabled state if the result changed.

**Binding with function expressions**

The last method to set binding values is by using functions. You can call a function by referencing it in the binding:

```
<button data-bind="enable: canApprove(age)">Approve</button>
```

You can also write an anonymous function as a string directly in the binding. When creating a function for the `click` binding, the parameters are the binding context (viewmodel) and the JavaScript `click` event. If you bind against a viewmodel function using its property name, it would receive the same parameters:

```
<button data-bind="text: function(data) { console.log(data.age) }">Log Age</button>
```

Though this is possible, I wouldn't encourage it. It places logic directly in the view instead of in the viewmodel where it belongs. You should only use this last method in very special cases. It's much better to place the method on the viewmodel and just use a property reference.

**Using parentheses in bindings**

It can be confusing trying to figure out when to use parentheses in bindings to use an observable as a value. Knockout tries to be helpful by not requiring the parentheses in simple binding expressions like this one:

```
<input data-bind="value: firstName" />
```

In this example, the `firstName` property could be either an observable or a literal value, and it would work just fine. However, there are two cases when the parentheses are needed in bindings: when binding against a nested property and when binding with an expression. Consider the following viewmodel:

```javascript
var viewmodel = {
  user: ko.observable({
    firstName: ko.observable('Tim'),
    age: ko.observable(27)
  })
};
```

For More Information:

www.packtpub.com/web-development/mastering-knockoutjs
Knockout Essentials

The user object here is an observable property, as are each of its properties. If we wanted to write the same binding now, it would need to include parentheses on the `user` function but still not on the `firstName` property:

```html
<input data-bind="value: user().firstName" />
```

In cases where we are binding directly against a property, the parentheses of that property are never needed. This is because Knockout is smart enough to understand how to access the value of the observable that it is given in bindings.

However, if we are binding against an expression, they are always needed:

```html
<button data-bind="enable: user().age > 18">Approve</button>
<button data-bind="enable: user().age() > 18">Approve</button>
```

Neither of these bindings will cause errors, but the first one will not work as expected. This is because the first expression will try to evaluate on the `age` observable itself (which is a function, not a number) instead of the observable's value. The second one correctly compares the value of the observable to 18, producing the expected result.

**Debugging with ko.toJSON**

Because `ko.toJSON` accepts the spaces argument for `JSON.stringify`, you can use it in a text binding to get a live copy of your viewmodel with nice, readable formatting:

```html
<pre data-bind="text: ko.toJSON($root, null, 2)"></pre>
```

The `cp1-databind` branch has an interactive example of each of these bindings.

**Control flow bindings**

So far, we have looked at one-way and two-way bindings that set or sync data with an attribute on an HTML element. There is a different kind of binding that Knockout uses for modifying the DOM by adding or removing nodes. These are the control flow bindings, and they include `foreach`, `if`, `with`, and `template`.

All of the control flow bindings work by actually removing their content from the DOM and creating an in-memory template from it. This template is used to add and remove the content as necessary.

For More Information:

Control flow bindings (except `if`) also introduce a binding context hierarchy. Your root binding context is the viewmodel passed to `ko.applyBindings`. The `data-bind` attributes have access to properties in the current context. Control flow bindings (other than `if`) create a child-binding context, meaning that `data-bind` attributes inside the control flow binding's template have access to the properties of their context and not the root context. Bindings inside a child context have access to special properties to allow them to navigate the context hierarchy. The most commonly used are:

- `$parent`: This accesses the binding context of the immediate parent. In this example, `group` and `$parent.group` refer to the same property because `$parent` accesses the context outside of the person:
  
  ```html
  <span data-bind="text: group"></span>
  <div data-bind="with: person">
    <span data-bind="text: name"></span>
    <span data-bind="text: $parent.group"></span>
  </div>
  ```

- `$parents[n]`: This is an array of parent contexts. The `$parents[0]` array is same as `$parent`.
- `$root`: This is the root viewmodel, the highest context in the hierarchy.
- `$data`: This is the current viewmodel, useful inside `foreach` loops.

For a complete list of context properties, see the Knockout documentation for them at http://knockoutjs.com/documentation/binding-context.html.

### The if binding

The `if` binding takes a value or expression to evaluate and only renders the contained template when the value or expression is truthy (in the JavaScript sense). If the expression is falsy, the template is removed from the DOM. When the expression becomes true, the template is recreated and any contained `data-bind` attributes are reapplied. The `if` binding does not create a new binding context:

```html
<div data-bind="if: isAdmin">
  <span data-bind="text: user.username"></span>
  <button data-bind="click: deleteUser">Delete</button>
</div>
```
This div would be empty when isAdmin is false or null. If the value of isAdmin is updated, the binding will re-evaluate and add or remove the template as necessary.

There is also an ifnot binding that just inverts the expression. It's useful if you want to still use a property reference without needing to add a bang and parentheses. The following two lines are equivalent:

```html
<div data-bind="if: !isAdmin()">
  <div data-bind="ifnot: isAdmin">
```

The parentheses are needed in the first example because it is an expression, not a property name. They are not needed in the second example because it is a simple property reference.

## The with binding

The with binding creates a new binding context using the supplied value, which causes bindings inside the bound element to be scoped to the new context. These two snippets are functionally similar:

```html
<div>
  First Name:
  <span data-bind="text: selectedPerson().firstName"></span>
  Last Name:
  <span data-bind="text: selectedPerson().lastName"></span>
</div>
```

```html
<div data-bind="with: selectedPerson">
  First Name:
  <span data-bind="text: firstName"></span>
  Last Name:
  <span data-bind="text: lastName"></span>
</div>
```

While saving a few keystrokes and keeping your bindings easier to read is nice, the real benefit of the with binding is that it is an implicit if binding. If the value is null or undefined, the content of the HTML element will be removed from the DOM. In the cases where this is possible, it saves you from the need to make null checks for each descendant binding.
The foreach binding

The foreach binding creates an implicit template using the contents of the HTML element and repeats that template for every element in the array.

This viewmodel contains a list of people we need to render:

```javascript
var viewmodel = {
    people: [{name: 'Tim'}, {name: 'Justin'}, {name: 'Mark'}]
}
```

With this binding, we create an implicit template for the `li` element:

```html
<ul data-bind="foreach: people">
    <li data-bind="text: name"></li>
</ul>
```

This binding produces the following HTML:

```html
<ul>
    <li>Tim</li>
    <li>Justin</li>
    <li>Mark</li>
</ul>
```

The thing to note here is that the `li` element is binding against `name`, which is the property of a person. Inside the `foreach` binding, the binding context is the child element. If you need to refer to the child itself, you can either use `$data` or supply an alias to the `foreach` binding.

The `$data` option is useful when the array only contains primitives that you want to bind against:

```javascript
var viewmodel = {
    people: ['Tim', 'Justin', 'Mark']
}
...

<ul data-bind="foreach: people">
    <li data-bind="text: $data"></li>
</ul>
```

For More Information:

www.packtpub.com/web-development/mastering-knockoutjs
The alias option can clean up your code, but it is particularly useful when you have a nested context and want to refer to the parent. Refer to the following code:

```html
<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>
```

This can be achieved with $parent, of course, but it is more legible when using an alias.

### Template binding

The template binding is a special control flow binding. It has a parameter for each of the other control flow bindings. It might be more accurate to say that the other control flow bindings are all aliases for the template binding:

```html
<ul data-bind="foreach: { data: categories, as: 'category' }">
    <li>
        <ul data-bind="template: { foreach: categories, as: 'category' }">
            <div data-bind="template: { name: 'person-template', data: seller }"></div>
            <script type="text/html" id="person-template">
                <h3 data-bind="text: name"></h3>
                <p data-bind="text: credits"></p>
            </script>
        </ul>
    </li>
</ul>
```

Both of these are functionally equivalent. The template binding as has a parameter for if and data (which together make a with binding).

However, unlike the other control flow bindings, it can also generate its template from a named source using the name parameter. By default, the only source Knockout looks for is a `<script>` tag with an id parameter matching the name parameter:

```html
<div data-bind="template: { name: 'person-template', data: seller }"></div>
<script type="text/html" id="person-template">
    <h3 data-bind="text: name"></h3>
    <p data-bind="text: credits"></p>
</script>
```

To stop the script block from being executed as JavaScript, you need a dummy script type, such as text/html or text/ko. Knockout will not apply bindings to script elements, but it will use them as a source for templates.

Though it is much more common to use the inline templates seen in `foreach` or `with`, named templates have three very important uses.

---

*For More Information:*

Reusable templates

As templates can reference an external source for the HTML, it is possible to have multiple template bindings pointing to a single source:

```html
<div>
    <div data-bind="template: { name: 'person', data: father} "></div>
    <div data-bind="template: { name: 'person', data: mother} "></div>
</div>
...
<script type="text/html" id="person">
    <h3 data-bind="text: name"></h3>
    <strong>Age: </strong>
    <span data-bind="text: age"></span><br>
    <strong>Location: </strong>
    <span data-bind="text: location"></span><br>
    <strong>Favorite Color: </strong>
    <span data-bind="text: favoriteColor"></span><br>
</script>
```

The branch cp1-reuse has an example of this technique.

Recursive templates

Because templates participate in data-binding themselves, it is possible for a template to bind against itself. If a template references itself, the result is recursive:

```html
<div data-bind="template: { name: 'personTemplate', data: forefather} ">
    <script type="text/html" id="personTemplate">
        <h4 data-bind="text: name"></h4>
        <ul data-bind="foreach: children">
            <li data-bind="template: 'personTemplate'"></li>
        </ul>
    </script>
</div>
```

The template reference in the preceding template is using the shorthand binding, which just takes the name of the template directly. When using this shorthand, the current binding context is used for the template's data parameter, which is perfect inside a foreach loop like this one. This is a common technique when using recursive templates, as trees of information are the most common place to find visual recursion.

An example of this recursive template is in the cp1-recurse branch.

---

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
Dynamic templates

The name of the template in the previous example is a string, but it could be a property reference too. Binding the template name to an observable allows you to control which template is rendered. This could be useful to swap a viewmodel's template between a display and edit mode. Consider this template binding:

```html
<div data-bind="template: { name: template, data: father} "></div>
```

This template binding backed by a viewmodel property such as this one:

```javascript
self.template = ko.computed(function() {
    return self.editing() ? 'editTemplate' : 'viewTemplate';
});
```

If we update the `editing` property from `true` to `false`, the template will re-render from `viewTemplate` to `editTemplate`. This allows us to programmatically switch between them.

An example of a dynamic edit/view template is in the `cp1-dynamic` branch.

In an advanced scenario, you could use a technique such as this for creating a generic container on a page to display entirely different views. Switching the template name and the data at the same time would mimic navigation, creating a Single Page Application (SPA). We will take a look at a similar technique when we get to Chapter 4, Application Development with Components and Modules.

Containerless control flow

So far, we have looked at using the control flow bindings (`if`, `with`, `foreach`, and `template`) and the standard `data-bind` attribute on an HTML element. It is also possible to use control flow bindings without an element by using special comment tags that are parsed by Knockout. This is called containerless control flow.

Adding a `<!- ko -->` comment starts a virtual element that ends with a `<!- /ko -->` comment. This virtual element causes a control flow binding to treat all contained elements as children. The following block of code demonstrates how sibling elements can be grouped by a virtual comment container:

```html
<ul>
    <li>People</li>
    <li>Locations</li>
    <!-- ko if: isAdmin -->
    <li>Users</li>
    <li>Admin</li>
    <!-- /ko -->
</ul>
```

For More Information:

List elements only allow specific elements as children. The preceding containerless syntax applies the if binding to the last two elements in the list, which causes them to add or remove from the DOM based in the isAdmin property:

```html
<ul>
  <li>Nav Header</li>
  <!-- ko foreach: navigationItems -->
  <li><span data-bind="text: $data"></span></li>
  <!-- /ko -->
</ul>
```

The preceding containerless syntax allows us to have a foreach binding to create a list of items while maintaining a header item at the top of the list.

All of the control flow bindings can be used in this way. The preceding two examples can be seen in the cp1-containerless branch.

## Extenders

The last "basic" feature to cover is extenders (don't worry, there is still plenty of advanced stuff to cover). Extenders offer a way to modify individual observables. Two common uses of extenders are as follows:

- Adding properties or functions to the observable
- Adding a wrapper around the observable to modify writes or reads

## Simple extenders

Adding an extender is as simple as adding a new function to the ko.extenders object with the name you want to use. This function receives the observable being extended (called the target) as the first argument, and any configuration passed to the extender is received as the second argument, as shown in the following code:

```javascript
ko.extenders.recordChanges = function(target, options) {
    target.previousValues = ko.observableArray();
    target.subscribe(function(oldValue) {
        target.previousValues.push(oldValue);
    }, null, 'beforeChange');
    return target;
};
```

This extender will create a new previousValues property on the observable. This new property is as an observable array and old values are pushed to it as the original observable is changed (the current value is already in the observable of course).
The reason the extender has to return the target is because the result of the extender is the new observable. The need for this is apparent when looking at how the extender is called:

```javascript
var amount = ko.observable(0).extend({ recordChanges: true});
```

The `true` value sent to `recordChanges` is received by the extender as the `options` parameter. This value can be any JavaScript value, including objects and functions.

You can also add multiple extenders to an observable in the same call. The object sent to the `extend` method will call an observable for every property it contains:

```javascript
var amount = ko.observable(0).extend({ recordChanges: true,
                                         anotherExtender: { intOption: 1}});  
```

As the `extend` method is called on the observable, usually during its initial creation, the result of the `extend` call is what is actually stored. If the target is not returned, the `amount` variable would not be the intended observable.

To access the extended value, you would use `amount.previousValues()` from JavaScript, or `amount.previousValues()` if accessing it from a binding. Note the lack of parentheses after `amount`; because `previousValues` is a property of the observable, not a property of the observable's value, it is accessed directly. This might not be immediately obvious, but it should make sense as long as you remember that the observable and the value the observable contains are two different JavaScript objects.

An example of this extender is in the `cp1-extend` branch.

### Extenders with Options

The previous example does not pass any options to the `recordChanges` extender, it just uses `true` because the property requires a value to be a valid JavaScript. If you want a configuration for your extender, you can pass it as this value, and a complex configuration can be achieved by using another object as the value.

If we wanted to supply a list of values that are not to be recorded, we could modify the extender to use the options as an array:

```javascript
ko.extenders.recordChanges = function(target, options) {
  target.previousValues = ko.observableArray();
  target.subscribe(function(oldValue) {
    if (!options.ignore && options.ignore.indexOf(oldValue) !== -1)
      target.previousValues.push(oldValue)
  });
}
```

For More Information:

Then we could call the extender with an array:

```javascript
var history = ko.observable(0).extend({
    recordChanges: { ignore: [0, null] }
});
```

Now our `history` observable won't record values for 0 or `null`.

### Extenders that replace the target

Another common use for extenders is to wrap the observable with a computed observable that modifies reads or writes, in which case, it would return the new observable instead of the original target.

Let's take our `recordChanges` extender a step further and actually block writes that are in our `ignore` array (never mind that an extender named `recordChanges` should never do something like this in the real world!):

```javascript
ko.extenders.recordChanges = function(target, options) {
    var ignore = options.ignore instanceof Array ? options.ignore : [];
    // Make sure this value is available
    var result = ko.computed({
        read: target,
        write: function(newValue) {
            if (ignore.indexOf(newValue) === -1) {
                result.previousValues.push(target());
                target(newValue);
            } else {
                target.notifySubscribers(target());
            }
        }
    }).extend({ notify: 'always' });

    result.previousValues = ko.observableArray();

    // Return the computed observable
    return result;
};
```

That's a lot of changes, so let's unpack them.
First, to make ignore easier to reference, I've set a new variable that will either be the options.ignore property or an empty array. Defaulting to an empty array lets us skip the null check later, which makes the code a little easier to read. Second, I created a writable computed observable. The read function just routes to the target observable, but the write function will only write to the target if the ignore option doesn't contain the new value. Otherwise, it will notify the target subscribers of the old value. This is necessary because if a UI binding on the observable initiated the change, it needs the illegal change to be reverted. The UI element would already have updated and the easiest way to change it back is through the standard binding notification mechanism that is already listening for changes.

The last change is the notify: always extender that's on the result. This is one of Knockout's default extenders. Normally, an observable will only report changes to subscribers when the value has been modified. To get the observable to reject changes, it needs to be able to notify subscribers of its current unchanged value. The notify extender forces the observable to always report changes, even when they are the same.

Finally, the extender returns the new computed observable instead of the target, so that anyone trying to write a value does so against the computed.

The cpl-extendreplace branch has an example of this binding. Notice that trying to enter values into the input box that are included in the ignored options (0 or an empty string) are immediately reverted.

The Contacts List application

It's time to start putting these concepts together into a usable application. Isolated samples can only take you so far. We are going to cover the application in the cpl-contacts branch in detail. The application's functionality is all on the Contacts page, which you can get to from the navigation bar in your browser. Before we start digging into the code, I encourage you to play around with the application a bit (it does persist data). It will help in understanding the relationships in the code.

Overview

The application has three main JavaScript objects:

- The contact model
- The Contacts page viewmodel
- The mock data service

For More Information:

www.packtpub.com/web-development/mastering-knockoutjs
Chapter 1

The application only uses the HTML in the index.html file, but the two sections are mostly independent.

- The entry form (create and edit)
- The contacts list

The JavaScript code in the example follows the **Immediately-Invoked Function Expression (IIFE)** pattern (sometimes pronounced “iffy”) to isolate code from the global scope, and a namespace called app to share code between files:

```
(function(app, $, ko) {
    /* CODE IN HERE */
})(window.app = window.app || {}, jQuery, ko);
```

This is definitely not the only way to organize JavaScript code, and you may have a pattern you prefer. If you want to understand this pattern better, here are a few online resources:


**The contact model**

The client/app/contacts.js file defines our basic contact object. Let’s go over it piece by piece.

It starts with a standard declaration of observable properties with some default values. There are a lot of reasons to organize code in a variety of ways, but for the smaller models, I prefer to keep all of their persistable properties together at the top:

```
app.Contact = function(init) {
    var self = this;
    self.id = ko.observable(0);
    self.firstName = ko.observable('');
    self.lastName = ko.observable('');
    self.nickname = ko.observable('');
    self.phoneNumber = ko.observable('');
    /* More below */
```
Next is the `displayName` property, some simple logic to generate a nice "title" for UI display. The JavaScript or operator (`||`) is used here to ensure we don't try to read the `length` property on a null or undefined value by returning a default value in case all the names are empty. This essentially makes it a null-coalescing operator when used during an assignment:

```javascript
self.displayName = ko.computed(function() {
    var nickname = self.nickname() || '';
    if (nickname.length > 0)
        return nickname;
    else if ((self.firstName() || '').length > 0)
        return self.firstName() + ' ' + self.lastName();
    else
        return 'New Contact';
});
```

Next is a utility method to update the model that accepts an object and merges in its properties. I generally put a similar method onto all of my models so that I have a standard way of updating them. Once again, we are using `||` as a safety net, in case the method is called without a parameter (in the real world, you would want a stronger check, one that ensured `update` was an object and not a primitive value or an array):

```javascript
//Generic update method, merge all properties into the viewmodel
self.update = function(update) {
    data = update || {};
    Object.keys(data).forEach(function(prop) {
        if (ko.isObservable(self[prop]))
            self[prop](data[prop]);
    });
}

//Set the initial values using our handy-dandy update method.
self.update(init);
```

Also note that after defining the `update` function, the model calls it with the constructor argument. This lets the constructor provide the ability to create a new model from existing data and partial data as well. This is very useful when deserializing data, for example, JSON from an Ajax request.

Lastly, we have the `toJSON` method. The standard `JSON.stringify` method in JavaScript will look for this method to allow an object to control how it is serialized. As Knockout's `ko.toJSON` calls `JSON.stringify` underneath after it unwraps all the observables so that the serialization gets values and not functions.
As the serialized form of our model is the one we will try to persist, usually by sending it to the server with Ajax, we don't want to include things such as our computed display name. Our toJSON method override takes care of this by just deleting the property:

```javascript
//Remove unwanted properties from serialized data
self.toJSON = function() {
    var copy = ko.toJS(self);
    delete copy.displayName;
    return copy;
};
```

The copy with ko.toJS is important. We don't want to delete displayName from the actual model; we only want it removed from the serialized model. If we made the variable with copy = self, we would just have a reference to the same object. The ko.toJS method is a simple way to get a plain JavaScript copy that we can safely delete properties from without affecting the original object.

### The Contacts page viewmodel

The client/app/contactspage.js file defines the viewmodel for the Contacts page. Unlike our contacts model, the page does a lot more than expose some observable properties, and it isn't designed to be constructed from existing data either. Instead of taking an object to control its starting values, which doesn't make much sense for a page, the constructor's argument is designed for dependency injection; its constructor arguments take in its external dependencies.

In this example, dataService is a dependency used by the page viewmodel:

```javascript
app.ContactsPageViewmodel = function(dataService)
```

Very briefly, if you aren't familiar with dependency injection, it lets us define our page against an API (sometimes called a contract or interface) of methods to get and save data. This is especially useful for us, as in this sample application, we aren't using real Ajax but mocking it with an object that just writes to the DOM's local storage:

```javascript
ko.applyBindings
(new app.ContactsPageViewmodel(app.mockDataService));
```

For more information on the DOM local storage, see the page on the Mozilla Developer Network at https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Storage.
However, when we write the real Ajax service later, our ContactsPageViewModel doesn't need to change at all. We will just construct it with a different DataService parameter. As long as they expose the same methods (the same API) it will just work.

The first section inside the constructor is for the contacts list. We expose an observable array and get the contacts from our data service:

```javascript
self.contacts = ko.observableArray();

DataService.getContacts(function(contacts) {
    self.contacts(contacts);
});
```

We are passing callback to the getContacts call because our data service provides an asynchronous API. When the data service has finished getting our contacts, it will call the callback with them. All our callback needs to do is put them into the contacts array.

The next block of code is to control the CRUD (Create, Read, Update, Delete) operations for individual contacts. First, we expose an observable object that we will use for all edits:

```javascript
self.entryContact = ko.observable(null);

self.newEntry = function() {
    self.entryContact(new app.Contact());
};
self.cancelEntry = function() {
    self.entryContact(null);
};
```

Our UI is going to bind an edit form against the entryContact property. The entry contact property is pulling a double duty here; it contains the contact that is being created or edited, and it indicates that editing is occurring. If the entry contact is null, then we aren't editing; if it has an object, then we are editing. The UI will use `with` and `if` bindings to control which content to show based on this logic.

The `newEntry` and `cancelEntry` functions provide the UI with a means to switch between these two states.

For editing existing contacts, we just expose another function that takes a contact and sets the entry contact to it:

```javascript
self.editContact = function(contact) {
    self.entryContact(contact);
};
```

---

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
The last thing we need for real editing is the ability to persist our changes. As in the real world, we have three scenarios, namely creating new objects, saving existing objects, and deleting existing objects.

Creating and updating are both going to be done using the `entryContact` property, and we want to be able to bind the same form for both, which means we need to target a single function:

```javascript
self.saveEntry = function() {
    if (self.entryContact().id() === 0) {
        dataService.createContact(self.entryContact(), function() {
            self.contacts.push(self.entryContact());
            self.entryContact(null);
        });
    } else {
        dataService.updateContact(self.entryContact(), function() {
            self.entryContact(null);
        });
    }
};
```

Internally, our `saveEntry` method checks for a non-default `id` value to determine whether or not it's making a new object or updating an existing one. Both are calls to the data service using the entry contact with a callback to clear the `entryContact` property out (as we are done with editing). In the creation case, we also want to add the newly created contact to our local list of contacts before emptying the entry contact:

```javascript
self.contacts.push(self.entryContact());
self.entryContact(null);
```

You might think that the contact is going to be null out by the second line, but that is not the case. The `entryContact` property is an observable and its value is a contact. The first line reads this value and pushes it into the `contacts` array. The second line sets the value of the `entryContact` property to null; it does not affect the contact that was just pushed. It's the same as if we had set a variable to null after adding it to an array. The variable was a reference to the object, and making the variable null removes the reference, not the object itself.

The delete function is simple by comparison:

```javascript
self.deleteContact = function(contact) {
    dataService.removeContact(contact.id(), function() {
        self.contacts.remove(contact);
    });
};
```

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
It's going to take an existing contact, like `editContact` did, and call the data service. As we are deleting the contact, the only thing we need is the `id` property. The callback will remove the contact from the list of contacts when the service is done, using the `remove` function provided on all observable arrays by Knockout.

The last piece of functionality on the page is the search mechanism. It starts with an observable to track the search and a function to clear it out:

```javascript
self.query = ko.observable('');
self.clearQuery = function() { self.query(''); }
```

The `query` property is going to be used to filter out any contacts that don't have a matching or partially-matching property. If we wanted to be as flexible as possible, we could search against every property. However, since our list of contacts is only going to show our computed `displayName` and phone number, it would look odd to return results matching on properties we didn't show. This is the computed observable from the code sample that filters the contacts list:

```javascript
self.displayContacts = ko.computed(function() {
  // No query, just return everything
  if (self.query() === '')
    return self.contacts();
  var query = self.query().toLowerCase();
  // Otherwise, filter all contacts using the query
  return ko.utils.arrayFilter(self.contacts(), function(c) {
    return c.displayName().toLowerCase().indexOf(query) !== -1
       || c.phoneNumber().toLowerCase().indexOf(query) !== -1;
  });
});
```

If you want to filter all of the contact's properties, they are listed in the repository code as comments. They can easily be re-enabled by uncommenting each line.

First, we check to see whether the query is empty, because if it is, we aren't going to filter anything so we don't want to waste cycles iterating the contacts anyway.

Before starting, we call the `toLowerCase()` function on the query to avoid any case sensitivity issues. Then, we iterate on the contacts. Knockout provides several utilities methods for arrays (among other things) on the `ko.utils` object. The `arrayFilter` function takes an array and an iterator function, which is called on each element of the array. If the function returns `true`, `arrayFilter` will include that element in its return value; otherwise it will filter the element out. All our iterator needs to do is compare the properties we want to keep the filter on (remembering to put them in lowercase first).
Now if the UI binds against `displayContacts`, the search functionality will filter the UI.

However, we might experience poor performance with a large list of contacts if we are looping through them all every time the query is updated, especially if the query updates every time a key is pressed. To address this, we can use the standard Knockout `rateLimit` extender on our filtered computed to stop it from updating too frequently:

```javascript
self.displayContacts = ko.computed(function() {
    /* computed body */
}).extend({
    rateLimit: {
        timeout: 100,
        method: 'notifyWhenChangesStop'
    }
});
```

This extender has two modes: `notifyAtFixedRate` and `notifyWhenChangesStop`. These two options will throttle or debounce the computed.

If you aren't familiar with the throttling and debouncing functions, there is an excellent explanation with visuals at http://drupalmotion.com/article/debounce-and-throttle-visual-explanation.

This lets us control how often the computed re-evaluates itself. The preceding example will only re-evaluate the computed once all dependencies have stopped changing for 100 ms. This will let the UI update when the query typing settles down while still appearing to filter as the user types.

**A philosophical note on a model versus a viewmodel**

The line between model and viewmodel in client-server application can get blurry, and even after reading Knockout's documentation (http://knockoutjs.com/documentation/observables.html) it can be unclear whether or not our contact object is really a model or viewmodel. Most would probably argue that it is a viewmodel as it has observables. I like to think of these smaller objects, which are barely more than their persisted data, as models and to think of viewmodels as the objects containing operations and view representations, such as our Contacts page viewmodel `removeContact` operation or the `entryContact` property.
Mock data service

Normally, you would use an Ajax call, probably with jQuery, to retrieve data and submit data to and from the server. Because this is a book on Knockout and not Node.js, I wanted to keep the server as thin as possible. From the "Mastering Knockout" perspective, whether we call a JavaScript object making Ajax requests or store it in the DOM is immaterial. As long as we are working with something that looks and functions like an asynchronous service, we can explore how Knockout viewmodels might interact with it. That being said, there is some functionality in the data service that would be used in an Ajax data service object, and it is interesting from a Knockout application development perspective.

You might have noticed in the previous section that when the Contacts page view model communicated with the data service, it wasn't dealing with JSON but real JavaScript objects. In fact, not even plain JavaScript objects but our contact model. This is because part of the data service's responsibility, whether it's a mock or a real Ajax service, is to abstract away the knowledge of the service mechanisms. In our case, this means translating between JSON and our Knockout models:

```javascript
createContact: function(contact, callback) {
  $.ajax({
    type: "POST",
    url: "/contacts",
    data: ko.toJS(contact)
  })
  .done(function(response) {
    contact.id(response.id);
    callback();
  });
}
```

This is the createContact method from our mock data service if it was rewritten to use real Ajax (this code is in the mockDataService.js file as a comment). The data service is part of our application, so it knows that it's working with observable properties and that it needs to translate them into plain JavaScript for jQuery to properly serialize it, so it unwraps the contact that it's given with ko.toJS. Then, in the done handler, it takes the id that it gets back from the server's response and updates the contact's observable id property with it. Finally, it calls the callback to signify that it's done.
You might wonder why it doesn't pass contact as an argument to the callback. It certainly could, but it isn't necessary. The original caller already had the contact, and the only thing that the caller is going to need is the new id value. We've already updated the id, and as it's observable, any subscriber will pick that new value up. If we needed some special handling before setting the id value, that would be a different case and we could raise the callback with id as an argument.

The view

Hopefully, you have already played with the application a bit. If you haven't, now is the time. I'll wait.

You would have noticed that when adding or editing contacts, the contacts list is removed. What you might not have noticed is that the URL doesn't change; the browser isn't actually navigating when we switch between these two views. Though they are in the same HTML file, these two different views are mostly independent and they are controlled through a with and an ifnot binding.

The edit form

This is what is shown when adding or editing contacts:

```html
<form class="form" role="form" data-bind="with: entryContact, submit: saveEntry">
  <h2 data-bind="text: displayName"></h2>
  <div class="form-group">
    <label for="firstName" class="control-label">First Name </label>
    <input type="text" class="form-control" id="firstName" placeholder="First Name" data-bind="value: firstName">
  </div>
  <div class="form-group">
    <label for="lastName" class="control-label">Last Name </label>
    <input type="text" class="form-control" id="lastName" placeholder="First Name" data-bind="value: lastName">
  </div>
  <div class="form-group">
    <label for="nickname" class="control-label">Nickname </label>
    <input type="text" class="form-control" id="nickname" placeholder="First Name" data-bind="value: nickname">
  </div>
</form>
```

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
Because the with binding is also implicitly an if binding, the entire form is hidden when the entryContact property is null or undefined.

The rest of the form is pretty straightforward. A submit binding is used so that clicking the save button or hitting the enter key on any field calls the submit handler, a header showing the display name, value bindings for each field, a save button with type="submit" (so that it uses the submit handler), and a cancel button that binds to $parent.cancelEntry. Remember, the $parent scope is necessary because the with binding creates a binding context on the entry contact and cancelEntry is a function on ContactPageViewmodel.

Contacts list

The list starts with an ifnot binding on the entryContact property, ensuring that it only shows in the case that the previous form is hidden. We only want one or the other to be seen at a time:

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
The search input has a value binding as well as the valueUpdate option. The value update option controls when the value binding reports changes. By default, changes are reported on blur, but the afterkeydown setting causes changes to be reported immediately after the input gets a new letter. This would cause the search to update in real time, but remember that the display contacts have a rateLimit extender that debounces the updates to 100 ms.

Next to the search box is a button to add a new contact. Then, of course, the list of contacts is bound with a foreach binding on the displayContacts property. If it was bound against contacts directly, the list would not show the filtering. Depending on your application, you might even want to keep the unfiltered contacts list private and only expose the filtered lists. The best option really does depend on what else you’re doing, and in most cases, it’s okay to use your personal preference.

Inside the contacts list, each item shows the display name for the phone number, with a button to edit or delete the contact. As foreach creates a binding context on the individual contact and the edit and delete functions are on the parent, the click binding uses the $parent context property. The click binding also sends the current model to each of the edit and delete functions, so that these functions don’t have to try to find the right JavaScript object by looking through the full list.

That’s really all there is to the application. We’ve got a list view with searching that switches to a view that’s reused easily for both editing and creating.

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
Summary

In most of this chapter, we reviewed the use of standard Knockout. Hopefully, I didn't lose you in the weeds back there. The important thing is that before we move on to extending Knockout with custom functionality or building larger applications, you must feel comfortable with the basic use of observables and data binding. This includes:

- **Defining viewmodels**: This includes creating observables, binding functions, and handling serialization
- **Writing bindings**: This includes using properties, expressions, inline functions, and when to use parentheses
- **Extenders**: This includes creating extenders and extending observables
- **Templates**: This tells us how the flow of control works, what a binding context is, inline versus named templates, and containerless control flow

In the next chapter, we will be adding new functionalities to Knockout by creating our own binding handlers.

For More Information:
www.packtpub.com/web-development/mastering-knockoutjs
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.