KnockoutJS Essentials

KnockoutJS is a Model View ViewModel (MVVM) framework that allows you to simplify the design of complex user applications. Knockout makes data manipulation simple and leads to maintainable web applications.

KnockoutJS Essentials gives you an insight into the best practices when designing and building MVVM web applications using the KnockoutJS framework. Packed with real-world tasks, this book starts with an introduction to the Knockout library, helping you make organizational and design decisions. You get a walk-through of the key concepts and terms that are common to development of Knockout applications. Finally, you will learn about the full-stack Knockout framework DurandalJS, which will give you all the tools you need to write complete single-page applications.

Who this book is written for

If you are a JavaScript developer who has been using DOM manipulation libraries such as Mootools or Scriptaculous, and you want to go further in modern JavaScript development with a simple and well-documented library, then this book is for you. Learning how to use Knockout will be perfect as your next step towards building JavaScript applications that respond to user interaction.

What you will learn from this book

- Organize and maintain your code when applications begin to grow
- Get to know JavaScript patterns and best practices to keep your code readable
- Bind JavaScript objects and DOM elements with practical examples
- Integrate and manage events to give the user a better experience
- Extend the framework to create your own components
- Create a single-page application using the Durandal JavaScript framework
- Split your application in modules to keep it engaging and maintainable
- Manage templates to keep your HTML as simple as possible

Implement a successful JavaScript-rich application with KnockoutJS, jQuery, and Bootstrap

Jorge Ferrando

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Refreshing the UI Automatically with KnockoutJS"
- A synopsis of the book’s content
- More information on KnockoutJS Essentials

About the Author

Jorge Ferrando is a frontend developer at Uni Micro AS, Norway. He develops economy systems using JavaScript. He has experience as a PHP and C# developer working for companies such as Vigilant and Dreamstar Cash.

Jorge grew up in Callosa d'en Sarrià, Spain. He attended the University of Alicante where he discovered that web applications are his passion. He started to develop websites when he was 16 years old for a little company in his hometown. He then started practicing at Vigilant as a junior PHP developer; there, he created his first full web application and discovered the potential of JavaScript. Hard work lead him to a bigger company, Dreamstar Cash, also as a PHP developer. There, he got to know several good developers and began to work more and more with JavaScript; thanks to the company, who wanted to use Node.js, Jorge fell in love with JavaScript and frontend development. A few months later, he moved to Norway with his girlfriend and began to work with a company that gave him new challenges every day, working with KnockoutJS, AngularJS, and Twitter Bootstrap.
Acknowledgments

I can't believe how fortunate I was to work on this book with so many great people. I would like to say thank you to Ellen Bishop, who gave me the opportunity to write it, and Owen Roberts and Shubhangi Dhamgaye, who have led me in my first adventure as a writer. Also to Christian Droulers, I have learned very much with his reviews.

Firstly, I would like to say thank you to my family; they have always supported my dreams and have given me the freedom to fail and succeed by myself, and they have always been by my side taking care of me.

To my girlfriend, Teresa, who supported me in those hard days when I thought all is going to fall because I had work all around and just 24 hours in a day. She is my stone and my stability.

I would like to extend my gratitude to all the people I have learned something from: Manuel Ciges, who gave me the responsibility of making a full production application, and Marcos Fernández, my partner through that trip at Vigilant. Rafael Crisanto from Rc Informática, who gave me my first opportunity as a web developer. Pedro García, Pablo Deeleman, Javier Gómez, Fran Iruela, and Alfonso Fernández, my partners at Dreamstar Cash. Guys you made that time the best of my life and you have made me a better developer.
KnockoutJS Essentials

One of the hardest problems to solve when we build the user interface is to synchronize data that developers manage behind the scenes, in the code, and data that is shown to the user. The first step that developers made was to separate the presentation and the logic. That separation allowed developers to manage both sides better and separately. But communication between these two layers still became hard. That was because JavaScript was considered a non-important language and we used to use it just for validation. Then jQuery gave us a clue about how powerful this language could be. But still data was managed in the server and we just displayed static presentations. That makes the user experience poor and slow.

Over the last few years, a new kind of architectural pattern has emerged. It is called the MVVM pattern. Libraries and frameworks that use this kind of pattern make it easy for developers to synchronize views and data. One of these libraries is Knockout, and the framework that uses Knockout is named Durandal.

Knockout is a fast and cross-browser-compatible library that helps us to develop client-side applications with a better user experience.

Developers don't need to worry any more about data synchronization. Knockout binds our code to the HTML elements showing the state of our code to the user in real time.

This dynamic binding makes us forget about coding synchronization, and we can focus our effort on coding the important features of our application.

Nowadays, managing these kinds of frameworks is a must for front-end developers. In this book, you will learn the basics of Knockout and Durandal and we will go into the best design practices and patterns of JavaScript.

If you want to improve the user experience of your applications and create fully operative frontend applications, Knockout and Durandal should be your choice.

What This Book Covers

Chapter 1, Refreshing the UI Automatically with KnockoutJS, teaches you about the Knockout library. You will create observables and make your templates reactive to changes.

Chapter 2, KnockoutJS Templates, shows how to create templates to reduce your HTML code. Templates will help you keep your designs maintainable and they can be adapted to your data.

Chapter 3, Custom Bindings and Components, shows how to extend the Knockout library to make your code more maintainable and portable.
Chapter 4, *Managing KnockoutJS Events*, teaches you how to communicate with isolated modules and libraries using jQuery events. Events will help you to send messages between different components or modules.

Chapter 5, *Getting Data from the Server*, shows how to communicate with the server from the client side using jQuery AJAX calls. You will also learn how to develop the client side without a server behind it using mock techniques.

Chapter 6, *The Module Pattern – RequireJS*, teaches you how to write well-formed modules using the module pattern and the AMD pattern to manage dependencies between libraries.

Chapter 7, *Durandal – The KnockoutJS Framework*, teaches you how the best Knockout framework works. You will learn about each part of the framework to have the capacity to make big applications with less code.

Chapter 8, *Developing Web Applications with Durandal – The Cart Project*, migrates the application built over the course of the book to Durandal. You will develop the same app with a few lines and will be able to add new features.
If you are reading this book, it is because you have discovered that managing web user interfaces is quite complex. DOM (short for Document Object Model) manipulation using only native JavaScript is very hard. This is because each browser has its own JavaScript implementation. To solve this problem, different DOM manipulation libraries have been born in the last few years. The library most frequently used to manipulate the DOM is jQuery.

It is increasingly common to find libraries that help developers to manage more and more features on the client side. As we have said, developers have obtained the possibility to manipulate the DOM easily and therefore to manage templates and format data. Also, these libraries provide developers with easy APIs to send and receive data from the server.

However, DOM manipulation libraries don't provide us with mechanisms to synchronize input data with the models in our code. We need to write code that catches user actions and updates our models.

When a problem occurs frequently in most projects, in almost all the cases, it can surely be solved in a similar way. It was then that libraries that manage the connection between the HTML files and JavaScript code began to appear. The pattern these libraries implement was named MV* (Model-View-Whatever). The asterisk can be changed by:

- Controller, MVC (for example, AngularJS)
- ViewModel, MVVM (for example, KnockoutJS)
- Presenter (MVP) (for example, ASP.NET)

The library we are going to use in this book is Knockout. It uses view-models to bind data and HTML, so it uses the MVVM pattern to manage the data binding issue.
In this chapter, you will learn the basic concepts of this library and you will begin a task to use Knockout in a real project.

**KnockoutJS and the MVVM pattern**

**KnockoutJS** is a very lightweight library (just 20 KB minified) that gives the ability to objects to become the nexus between views and models. It means that you can create rich interfaces with a clean underlying data model.

For this purpose, it uses declarative bindings to easily associate DOM elements with model data. This link between data and presentation layer (HTML) allows the DOM to refresh displayed values automatically.

Knockout set up chains of relationships between model data to transform and combine it implicitly. Knockout is also trivially extensible. It is possible to implement custom behaviors as new declarative bindings. This allows the programmer to reuse them in a just few lines of code.

The advantages of using KnockoutJS are many:

- It's free and open source.
- It is built using pure JavaScript.
- It can work together with other frameworks.
- It has no dependencies.
- It supports all mainstream browsers, even ancient ones such as IE 6+, Firefox 3.5+, Chrome, Opera, and Safari (desktop/mobile).
- It is fully documented with API docs, live examples, and interactive tutorials.

Knockout's function is specific: to join views and models. It doesn't manage DOM or handle AJAX requests. For these purposes, I would recommend jQuery. Knockout gives us the freedom to develop our code the way we want.
A real-world application – koCart

In order to demonstrate how to use Knockout in a real application, we are going to build a simple shopping cart called koCart.

First of all we are going to define the user stories. We just need a few sentences to know what we want to achieve, which are as follows:

- The user should be able to view the catalog
- We should have the ability to search the catalog
- The user can click on a button to add items to the catalog
- The application will allow us to add, update, and delete items from the catalog
- The user should be able to add, update, and delete items from the cart
- We will allow the user to update his personal information.
- The application should be able to calculate the total amount in the cart
- The user should be able to complete an order

Through user stories, we can see that our application has the following three parts:

- The catalog, which contains and manages all the products we have in the shop.
- The cart, which has responsibility for calculating the price of each line and the total amount of the order.
- The order, where the user can update his personal information and confirm the order.

Installing components

To develop our real-world project, we need to install a few components and set up our first layout.

These are all the components you need to download:

- Bootstrap: https://github.com/twbs/bootstrap/releases/download/v3.2.0/bootstrap-3.2.0-dist.zip
- jQuery: https://code.jquery.com/jquery-2.1.1.min.js
- KnockoutJS: http://knockoutjs.com/downloads/knockout-3.2.0.js
Refreshing the UI Automatically with KnockoutJS

Since we just work on the client side in the first chapters, we can mock data in the client and will not need a server side for now. So we can choose any place in our computer to start our project. I recommend you use the environment you usually employ to do your projects.

To start, we create a folder called ko-cart and then create three folders and a file inside it:

1. In the css folder, we will put all our css
2. In the js folder, we will put all our JavaScript
3. In the fonts folder, we will put all the font files needed by the Twitter Bootstrap framework
4. Create an index.html file

Now you should set up your files the same way as shown in the following screenshot:

```
<!DOCTYPE html>
<html>
<head>
  <title>KO Shopping Cart</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
</head>
<body>
  <script type="text/javascript" src="js/vendors/jquery.min.js"></script>
  <script type="text/javascript" src="js/vendors/bootstrap.min.js"></script>
</body>
</html>
```

Then we should set the content of the index.html file. Remember to set all the links to the files we will need using the `<script>` and `<link>` tags:
With these lines of code, we have all we need to start our application.

The view-model
The view-model is a pure code representation of the data and operations on a UI. It isn't the UI itself. It doesn't have any concept of buttons or display styles. It's not the persisted data model either. It holds the unsaved data the user is working with. View-models are pure JavaScript objects that have no knowledge of HTML. Keeping the view-model abstract in this way lets it stay simple, so you can manage more sophisticated behaviors without getting lost.

To create a view-model, we just need to define a simple JavaScript object:

```javascript
var vm = {};
```

Then to activate Knockout, we will call the following line:

```javascript
ko.applyBindings(vm);
```

The first parameter says which view-model object we want to use with the view. Optionally, we can pass a second parameter to define which part of the document we want to search for data-bind attributes.

```javascript
ko.applyBindings(vm, document.getElementById('elementID'));
```

This restricts the activation to the element with elementID and its descendants, which is useful if we want to have multiple view-models and associate each with a different region of the page.

The view
A view is a visible, interactive UI representing the state of the view-model. It displays information from the view-model, sends commands to the view-model (for example, when the user clicks on buttons), and updates whenever the state of the view-model changes. In our projects, views are represented by the HTML markups.

To define our first view, we are going to build an HTML to display a product. Add this new content to the container:

```html
<div class="container-fluid">
  <div class="row">
```

<div class="col-md-12">
  <!-- our app goes here →
  <h1>Product</h1>
  <div>
    <strong>ID:</strong> <span data-bind="text:product.id"/>
    <br/>
    <strong>Name:</strong> <span data-bind="text:product.name"/>
    <br/>
    <strong>Price:</strong> <span data-bind="text:product.price"/>
    <br/>
    <strong>Stock:</strong> <span data-bind="text:product.stock"/>
  </div>
</div>
</div>

Look at the data-bind attribute. This is called declarative binding. This attribute isn't native to HTML, though it is perfectly correct. But since the browser doesn't know what it means, you need to activate Knockout (the ko.applyBindings method) to make it take effect.

To display data from a product, we need to have a product defined inside our view-model:

```javascript
var vm = {
  product: {
    id: 1,
    name: 'T-Shirt',
    price: 10,
    stock: 20
  }
};
kо.applyBindings(vm);//This how knockout is activated
```

Add the view-model to the end of the script tags:

```html
<script type="text/javascript" src="js/viewmodel.js"></script>
```

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
This will be the result of our app:

![Product](image)

**The model**

This data represents objects and operations in your business domain (for example, products) and is independent of any UI. When using Knockout, you will usually make AJAX calls to some server-side code to read and write this stored model data.

Models and view-models should be separated from each other. In order to define our product model, we are going to follow some steps:

1. Create a folder inside our js folder.
2. Name it models.
3. Inside the models folder, create a JavaScript file called product.js.

The code of the product.js file is as follows:

```javascript
var Product = function (id, name, price, stock) {
    "use strict";
    var
        _id = id,
        _name = name,
        _price = price,
        _stock = stock;

    return {
        id: _id,
        name: _name,
        price: _price,
        stock: _stock
    };
};
```
This function creates a simple JavaScript object that contains the interface of the product. Defining the object using this pattern, called the revealing module pattern, allows us to clearly separate public elements from private elements.

To learn more about the revealing module pattern, follow the link https://carldanley.com/js-revealing-module-pattern/.

Link this file with your index.html file and set it at the bottom of all the script tags.

```html
<script type="text/javascript" src="js/models/product.js"></script>
</script>
```

Now we can use the product model to define the product in the view-model:

```javascript
var vm = {
    product: Product(1,'T-Shirt',10,20);
};
ko.applyBindings(vm);
```

If we run the code again, we will see the same result, but our code is more readable now. View-models are used to store and handle a lot of information, because of this view-models are commonly treated as modules and the revealing module pattern is applied on them. This pattern allows us in a clear manner to expose the API (public elements) of the view-model and hide private elements.

```javascript
var vm = (function(){
    var product = Product(1,'T-Shirt', 10, 20);
    return {
        product: product
    };
})();
```

Using this pattern when our view-model begins to grow helps us to clearly see which elements belong to the public part of the object and which ones are private.

## Observables to refresh the UI automatically

The last example shows us how Knockout binds data and the user interface, but it doesn't show the magic of the automatic UI refresh. To perform this task, Knockout uses observables.

**Observables** are the main concept of Knockout. These are special JavaScript objects that can notify subscribers about changes, and can automatically detect dependencies. For compatibility, `ko.observable` objects are actually functions.
To read an observable's current value, just call the observable with no parameters. In this example, `product.price()` will return the price of the product, and `product.name()` will return the name of the product.

```javascript
var product = Product(1,"T-Shirt", 10.00, 20);
product.price();//returns 10.00
product.name();//returns "T-Shirt"
```

To write a new value to the observable, call the observable and pass the new value as a parameter. For example, calling `product.name('Jeans')` will change the name value to 'Jeans'.

```javascript
var product = Product(1,"T-Shirt", 10.00, 20);
product.name();//returns "T-Shirt"
product.name('Jeans'); //sets name to "Jeans"
product.name();//returns "Jeans"
```

The complete documentation about observables is on the official Knockout website [http://knockoutjs.com/documentation/observables.html](http://knockoutjs.com/documentation/observables.html).

To show how observables work, we are going to add some input data into our template.

Add these HTML tags over `div` that contain product information.

```html
<div>
  <strong>ID:</strong> <input class="form-control" type="text"
  data-bind="value:product.id"/><br/>
  <strong>Name:</strong>  <input class="form-control" type="text"
  data-bind="value:product.name"/><br/>
  <strong>Price:</strong>  <input class="form-control" type="text"
  data-bind="value:product.price"/><br/>
  <strong>Stock:</strong>  <input class="form-control" type="text"
  data-bind="value:product.stock"/>
</div>
```

We have linked inputs to the view-model using the value property. Run the code and try to change the values in the inputs. What happened? Nothing. This is because variables are not observables. Update your `product.js` file, adding the `ko.observable` method to each variable:

```javascript
"use strict";
function Product(id, name, price, stock) {
```
"use strict";

var
    _id = ko.observable(id),
    _name = ko.observable(name),
    _price = ko.observable(price),
    _stock = ko.observable(stock)

    return {
        id:_id,
        name:_name,
        price:_price,
        stock:_stock
    };

Notice that when we update the data inside the inputs, our product values are updated automatically. When you change the name value to Jeans, the text binding will automatically update the text content of the associated DOM element. That's how changes to the view-model automatically propagate to the view.

Observable models are updated automatically
Managing collections with observables

If you want to detect and respond to changes in one object, you’d use observables. If you want to detect and respond to changes in a collection of things, use an observableArray. This is useful in many scenarios where you’re displaying or editing multiple values and need repeated sections of the UI to appear and disappear as items are added and removed.

To display a collection of products in our application, we are going to follow some simple steps:

1. Open the index.html file and remove the code inside the <body> tag and then add a table where we will list our catalog:

   ```html
   <h1>Catalog</h1>
   <table class="table">
   <thead>
   <tr>
   <th>Name</th>
   <th>Price</th>
   <th>Stock</th>
   </tr>
   </thead>
   <tbody>
   <tr>
   <td></td>
   <td></td>
   <td></td>
   </tr>
   </tbody>
   </table>
   ```

2. Define an array of products inside the view-model:

   ```javascript
   "use strict";
   var vm = (function () {

   var catalog = [
   Product(1, "T-Shirt", 10.00, 20),
   Product(2, "Trousers", 20.00, 10),
   Product(3, "Shirt", 15.00, 20),
   Product(4, "Shorts", 5.00, 10)
   ];

   return {
   catalog: catalog
   };
   })();
   ko.applyBindings(vm);
   ```
3. Knockout has a binding to repeat a piece of code for each element in a collection. Update the tbody element in the table:

```html
<tbody data-bind="foreach:catalog">
  <tr>
    <td data-bind="text:name"></td>
    <td data-bind="text:price"></td>
    <td data-bind="text:stock"></td>
  </tr>
</tbody>
```

We use the `foreach` property to point out that all that is inside this tag should be repeated for each item in the collection. Inside this tag we are in the context of each element, so you can just bind properties directly. Observe the result in your browser.

We want to know how many items we have in our catalog, so add this line of code above the table:

```html
<strong>Items:</strong> <span data-bind="text:catalog.length"></span>
```

### Inserting elements in collections

To insert elements in the products array, an event should occur. In this case, the user will click on a button and this action will fire an action that will insert a new product in the collection.

In future chapters, you will learn more about events. Now we will just need to know that there is a binding property named `click`. It receives a function as a parameter, and this function is fired when the user clicks on the element.

To insert an element, we need a form to insert the values of the new product. Write this HTML code just below the `<h1>` tag:

```html
<form class="form-horizontal" role="form" data-bind="with:newProduct">
  <div class="form-group">
    <div class="col-sm-12">
      <input type="text" class="form-control" placeholder="Name" data-bind="textInput:name">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-12">
      <input type="password" class="form-control" placeholder="Price" data-bind="textInput:price">
    </div>
  </div>
</form>
```
In this template, we find some new bindings:

- The `with` binding: This creates a new binding context so that descendant elements are bound in the context of a specified object, in this case `newProduct`.
  
  http://knockoutjs.com/documentation/with-binding.html

- The `textInput` binding: The `textInput` binding links a textbox (`<input>`) or text area (`<textarea>`) with a view-model property, providing two-way updates between the `viewModel` property and the element's value. Unlike the `value` binding property, `textInput` provides instant updates from the DOM for all types of user input, including autocomplete, drag-and-drop, and clipboard events. It is available from the 3.2 version of Knockout.
  
  http://knockoutjs.com/documentation/textinput-binding.html

- The `click` binding: The `click` binding adds an event handler so that your chosen JavaScript function is invoked when the associated DOM element is clicked. When calling your handler, Knockout will supply the current model value as the first parameter. This is particularly useful if you're rendering UI for each item in a collection, and you need to know which item's UI was clicked.
  
  http://knockoutjs.com/documentation/click-binding.html

- The `$parent` object: This is a binding context property. We use it to refer to data from outside the `foreach` loop.
For more information about binding context properties, read the Knockout documentation at http://knockoutjs.com/documentation/binding-context.html.

Now it is time to add the `newProduct` object to our view-model. First we should define a new product with empty data:

```javascript
var newProduct = Product("","","");
```

We have defined a literal object that will contain the information we want to put inside our new product. Also, we have defined a method to clear or reset the object once the insertion is done. Now we define our `addProduct` method:

```javascript
var addProduct = function (context) {
    var id = new Date().valueOf(); // random id from time
    var newProduct = Product(
        id,
        context.name(),
        context.price(),
        context.stock()
    );
    catalog.addProduct(id, newProduct, context);
    var vm = this.$parent;
    // Call the method
Chapter 1

context.stock();
catalog.push(newProduct);
newProduct.clear();

This method creates a new product with the data received from the click event.

The click event always sends the context as the first argument. Note also that you can use array methods such as `push` in an observable array. Check out the Knockout documentation (http://knockoutjs.com/documentation/observableArrays.html) to see all the methods available in arrays.

We should implement the private method that will clean data from the new product once it is added to the collection:

```javascript
var clearNewProduct = function () {
  newProduct.name("");
  newProduct.price("");
  newProduct.stock("");
};
```

Update the view-model:

```javascript
return {
  catalog: catalog,
  newProduct: newProduct,
  addProduct: addProduct
};
```

If you run the code, you will notice that when you try to add a new product nothing happens. This is because, despite the fact that our products have observable properties, our array is not an observable one. For this reason, Knockout is not listening to the changes. We should convert the array to an `observableArray` observable.

```javascript
var catalog = ko.observableArray([{
  Product(1, "T-Shirt", 10.00, 20),
  Product(2, "Trousers", 20.00, 10),
  Product(3, "Shirt", 15.00, 20),
  Product(4, "Shorts", 5.00, 10)
}]);
```

Now Knockout is listening to what is going on with this array, but not what is happening inside each element. Knockout just tells us about inserting or deleting elements in the array, but not about editing elements. If you want to know what is happening in an element, the object should have observable properties.
An observableArray observable just tracks which objects it holds, and notifies listeners when objects are added or removed.

Behind the scenes, the observableArray is actually an observable whose value is an array. So you can get the underlying JavaScript array by invoking the observableArray observable as a function with no parameters, just like any other observable. Then you can read information from that underlying array.

```
<strong>Items:</strong>
<span data-bind="text: catalog().length"></span>
```

### Computed observables

It is not weird to think that some values we show in our interface depend on other values that Knockout is already observing. For example, if we would like to search products in our catalog by name, it is evident that the products in the catalog that we show in the list are related to the term we have entered in the search box. In these cases Knockout offers us **computed observables**.

You can learn in detail about computed observables in the Knockout documentation at [http://knockoutjs.com/documentation/computedObservables.html](http://knockoutjs.com/documentation/computedObservables.html).

To develop the search function, define a textbox where we can write a term to search. We are going to bind it to the `searchTerm` property. To update the value as we write, we should use the `textInput` binding. If we use the value binding, the value will be updated when the element loses the focus. Put this code over the products table:

```
<div class="input-group">
    <span class="input-group-addon">
        <i class="glyphicon glyphicon-search"></i> Search</span>
    <input type="text" class="form-control" data-bind="textInput: searchTerm">
</div>
```

To create a filtered catalog, we are going to check all our items and test if the `searchTerm` is in the item's name property.

```
var searchTerm = ko.observable('');
var filteredCatalog = ko.computed(function () {
    //if catalog is empty return empty array
    if (!catalog()) {
        return [];
    }
    var filter = searchTerm().toLowerCase();
    //if filter is empty return all the catalog
    if (!filter) {
        return catalog();
    }
    return catalog().filter(function (item) {
        return item.name.toLowerCase().indexOf(filter) > -1;
    });
});
```
// filter data
var filtered = ko.utils.arrayFilter(catalog(), function (item) {
  var fields = ['name']; // we can filter several properties
  var i = fields.length;
  while (i--) {
    var prop = fields[i];
    var strProp = ko.unwrap(item[prop]).toLocaleLowerCase();
    if (strProp.indexOf(filter) !== -1) {
      return true;
    }
  }
  return false;
});

The ko.utils object is not documented in Knockout. It is an object used by the library internally. It has public access and has some functions that can help us with observables. There are a lot of unofficial examples about it on the Internet.

One of its helpful functions is ko.utils.arrayFilter. If you look at line 13, we have used this method to obtain a filtered array.

This function gets an array as the first parameter. Notice that we invoke the catalog array observable to get the elements. We don't pass the observable itself, but the contents of the observable.

The second parameter is the function that decides whether the item will be in the filtered array or not. It will return true if the item has the conditions to be in the filtered array. Otherwise it returns false.

On line 14 of this snippet, we can find an array called fields. This parameter will contain the fields that should comply with the criteria. In this case, we just check that the filter value is in the name value. If we are pretty sure that we are just going to check the name field, we can simplify the filter function:

```javascript
var filtered = ko.utils.arrayFilter(catalog(), function (item) {
  var strProp = ko.unwrap(item['name']).toLocaleLowerCase();
  return (strProp.indexOf(filter) > -1);
});
```

The ko.unwrap function returns the value that contains the observable. We use ko.unwrap when we are not sure if the variable contains an observable or not, for example:

```javascript
var notObservable = 'hello';
console.log(notObservable()) // this will throw an error.
```
Refreshing the UI Automatically with KnockoutJS

```javascript
console.log(ko.unwrap(notObservable)); //this will display 'hello');
```

Expose the filtered catalog into the public API. Notice that now we need to use the filtered catalog instead of the original catalog of products. Because we are applying the **revealing module pattern**, we can keep the original API interface and just update the value of the catalog with the filtered catalog. We don’t need to alert the view that we are going to use a different catalog or other element, as long as we always maintain the same public interface:

```javascript
return {
  searchTerm: searchTerm,
  catalog: filteredCatalog,
  newProduct: newProduct,
  addProduct: addProduct
};
```

Now, try to type some characters in the search box and see in your browser how the catalog updates the data automatically.

Wonderful! We have completed our first three user stories:

- The user should be able to view the catalog
- The user should be able to search the catalog
- The user should be able to add items to the catalog

Let's see the final result:

```
<table>
<thead>
<tr>
<th>Catalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Price</td>
</tr>
<tr>
<td>Stock</td>
</tr>
<tr>
<td><img src="link" alt="Add Product" /></td>
</tr>
<tr>
<td><img src="link" alt="Search" /></td>
</tr>
<tr>
<td>Items:1</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>T-Shirt</td>
</tr>
<tr>
<td>Trousers</td>
</tr>
<tr>
<td>Shirt</td>
</tr>
<tr>
<td>Shorts</td>
</tr>
</tbody>
</table>
```
Summary

In this chapter, you have learned the basics of the Knockout library. We have created a simple form to add products to our catalog. You have also learned how to manage observable collections and display them in a table. Finally, we have developed the search functionality using computed observables.

You have learned three important Knockout concepts:

- **View-model**: This holds the data that represents the state of the view. It is a pure JavaScript object.
- **Models**: This contains data from the business domain.
- **Views**: This displays the data we store in the view-model in a given instant of time.

To build reactive UIs, the Knockout library provides us with some important methods:

- **ko.observable**: This is used to manage variables.
- **ko.observableArray**: This is used to manage arrays.
- **ko.computed**: They respond to changes from observables that are inside them.

To iterate over the elements of an array, we use the `foreach` binding. When we use the `foreach` binding, we create a new context. This context is relative to each item. If we want to access out of this context we should use the `$parent` object.

When we want to create a new context relative to a variable, we can attach the `with` binding to any DOM element.

We use the `click` binding to attach the click event to an element. Click on event functions to always get the context as the first parameter.

To get values from a variable that we are not sure is an observable, we can use the `ko.unwrap` function.

We can use the `ko.utils.arrayFilter` function to filter collections.

In the next chapter, we are going to use templates to keep our code maintainable and clean. Template engines help us to keep our code arranged and allow us to update views in an easy way.

There is a copy of the code developed in this chapter at

Where to buy this book

You can buy KnockoutJS Essentials from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.