Getting Started with Knockout.js for .NET Developers

Unleash the power of Knockout.js to build complex ASP.NET web applications

Andrey Akinshin
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Integrating Knockout.js in ASP.NET MVC Applications'
- A synopsis of the book’s content
- More information on Getting Started with Knockout.js for .NET Developers

About the Author

Andrey Akinshin has a PhD in computer science, and he received a Microsoft MVP award in 2015. He works as a lead .NET Developer at Perpetuum Software and as a postdoctoral research fellow at the Weizmann Institute of Science. He is also the author and main contributor of the Knockout MVC library and has a wealth of experience in Knockout.js. He has experience in various IT areas, from competitive programming (silver medal at ACM ICPC 2009) to teaching (senior lecturer and the school coach of competitive programming and mathematics teams).

You can find more information about Andrey on his home page, http://aakinshin.net.
Getting Started with Knockout.js for .NET Developers

Development of a big web application is a hard task. Because of this, people try to use different useful and flexible approaches to build the architecture of their applications. One such approach is Knockout.js. It is a JavaScript library that provides you with a sophisticated way to communicate between your UI and the underlying data model to create rich web user interfaces based on the Model-View-ViewModel (MVVM) pattern. Knockout.js provides a simple two-way data binding mechanism between your data model and UI, which means that any change to your data model is automatically reflected in the UI and vice versa. Instead of using pure Knockout.js, you can use Knockout MVC. It is a library for ASP.NET MVC, which is a wrapper for Knockout.js that helps to move the entire business logic to the server side; the complete JavaScript code necessary on the client side will be generated.

This book will provide you with the skills you need to successfully create a Knockout.js-based application of varying complexity, from a simple Knockout.js web page in pure JavaScript to a complex ASP.NET web application. You will learn how you can use the MVVM design pattern, including the dependency tracking system and observable properties for creation of powerful sites with a clear separation of model, logic, and view layers.

What This Book Covers

Chapter 1, Introduction to Knockout.js, teaches basic Knockout.js concepts (overview), such as MVVM design pattern (including the creation of Model, ViewModel, and View), binding, observables, and subscribe machinery. Also, we'll consider an installation process for the library.

Chapter 2, Creating a Simple Knockout.js Application, covers how to use advanced Knockout.js features, such as working with observables arrays, special bindings, and computed observables.

Chapter 3, Integrating Knockout.js in ASP.NET MVC Applications, uses our Knockout.js experience to create a simple ASP.NET MVC application. We'll create a simple application in pure Knockout.js + ASP.NET MVC without external libraries. We'll cover how to create a Model in C# and connect it with the MVVM structure in JavaScript.

Chapter 4, Creating a Web Application with Knockout MVC, discusses how to move the entire business logic to the server side; the complete JavaScript code necessary on the client side will be generated automatically based on the described C# (or VB.NET) model.
Chapter 5, *Advanced Features of Knockout.js*, covers how to use advanced Knockout MVC features. The basic set will be enough for a very simple application. Any real application needs special concepts, such as regions, complex bindings, combined contexts, and so on. You may need to transfer some parameters to the server, write your own user scripts, or perform lazy loading of your data in the case of big data.

Chapter 6, *Advanced Features of Knockout MVC*, discusses how to use advanced Knockout MVC features. The basic set will be enough for a very simple application. Any real application needs special concepts, such as regions, complex bindings, combined contexts, and so on. You may need to transfer some parameters to the server, write your own user scripts, or make lazy loading of your data in the big data case.

Appendix, *A Brief on Knockout MVC References and Features*, lists some references and features that will be useful to readers.
Integrating Knockout.js in ASP.NET MVC Applications

In this chapter, we'll use our Knockout.js experience to create a simple ASP.NET MVC application. We'll create an application in pure Knockout.js plus ASP.NET MVC without external libraries. You'll also learn how to create a Model in C# and connect it with the MVVM structure in JavaScript.

We assume that you have some basic experience of developing ASP.NET MVC applications with Visual Studio, including familiarity with the Razor view engine. Also, we assume that you have got an understanding of the previous chapter, which explains the Knockout.js basics.

The list of topics that will be covered in this chapter are as follows:

- Creating an empty ASP.NET MVC application with Knockout.js support via NuGet
- Creating a C# Model and connecting it with JavaScript code
- Client-server interaction in a Knockout.js ASP.NET MVC application with the Razor view engine

Creating an application without Knockout.js

In this section, we will create a very simple ASP.NET MVC application with step by step instructions. We will use Visual Studio Express 2013 with Update 3 for Web (you can download for free it from the official site, http://www.visualstudio.com/), .NET Framework 4.5, and ASP.NET MVC 4 for this purpose, but you can use older or newer versions; the difference will be small.
Integrating Knockout.js in ASP.NET MVC Applications

The minimum requirements for this chapter are Visual Studio 2010, .NET Framework 4.0, and ASP.NET MVC 3.

Let's develop an application to store information about the home library. The application will allow users to manage their book lists, including standard Create-Read-Update-Delete (CRUD) operations (see [http://en.wikipedia.org/wiki/Create,_read,_update_and_delete](http://en.wikipedia.org/wiki/Create,_read,_update_and_delete) for more information). It should be noted that we will develop a study application; it shouldn't be considered a production-ready code. You will not find any interactions with a database, error handling, data validation, and other important aspects of a real application here. The source code will be written in a minimalist style because we need this example only for future work and to understand some key Knockout.js features. You can find the final version of the development solution in the HomeLibrary folder.

Creating a new project

In this section, you will learn how to create a new project in Visual Studio 2013. First of all, you should perform the following preparatory steps:

1. Launch the Visual Studio 2013 IDE.
2. Create a new ASP.NET MVC 4 Web Application project (go to the File menu and select New | Project…). Let's call the solution HomeLibrary and the project HomeLibrary1.
3. Select the **Basic** project template.

4. We don't want to work with the design of our pages in this example. Therefore, we will use the Bootstrap framework to make our site look nice. **Bootstrap** (http://getbootstrap.com/) is the popular HTML, CSS, and JS framework used to develop beautiful designs for web projects from ready components and blocks (or using ready CSS styles). We can install it via NuGet. **NuGet** (http://www.nuget.org/) is the package manager that will help us to add new dependencies with ease. There are two ways to install a library via NuGet.
Right-click on the project in **Solution Explorer**, select **Manage NuGet Packages...**, search for **Bootstrap**, and click on **Install**.

The another way is go to the **Tools** menu, select **NuGet Package Manager**, and then select **Package Manager Console**. Set **Default project** to **HomeLibrary1** and run the following command:

```
PM> Install-Package bootstrap
```

This is shown in the following screenshot:
5. Open the Views/Shared/_Layout.cshtml file (it is the file that contains the default layout logic for your web pages) and add links to the Bootstrap library in the head section:

```csharp
@Styles.Render("~/Content/bootstrap.css")
@Scripts.Render("~/Scripts/bootstrap.js")
```

Now we are ready to add some logic to our project.

**Adding models**

We will create two models: one for the book and one for the library. Go to Solution Explorer, right-click on the Models folder of your project, and select Add | Class.... Let's call it BookModel. The source code is as follows:

```csharp
namespace HomeLibrary1.Models
{
    public class BookModel
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string Author { get; set; }
        public int Year { get; set; }
    }
}
```

It is a really simple class. It contains four properties:

- **Id**: This is the identification number of the book
- **Title**: This is the title of the book
- **Author**: This is the author of the book
- **Year**: This is the publication year of the book

The second model will be called LibraryModel (you should create a corresponding file, as in the first case). The source code is as follows:

```csharp
using System.Collections.Generic;

namespace HomeLibrary1.Models
{
    public class LibraryModel
    {
```
private readonly List<BookModel> books = new List<BookModel>();

private int nextId = 1;

public string Name { get; set; }

public LibraryModel()
{
    Name = "My home library";
    AddBook(new BookModel { Title = "Oliver Twist", Author = "Charles Dickens", Year = 1837 });
    AddBook(new BookModel { Title = "Winnie-the-Pooh", Author = "A. A. Milne", Year = 1926 });
    AddBook(new BookModel { Title = "The Hobbit", Author = "J. R. R. Tolkien", Year = 1937 });
    AddBook(new BookModel { Title = "The Bicentennial Man", Author = "Isaac Asimov", Year = 1976 });
    AddBook(new BookModel { Title = "The Green Mile", Author = "Stephen King", Year = 1996 });
}

public IEnumerable<BookModel> GetBooks()
{
    return books;
}

public BookModel GetBook(int id)
{
    return books.Find(b => b.Id == id);
}

public void AddBook(BookModel book)
{
    book.Id = nextId++;
    books.Add(book);
}

public bool UpdateBook(BookModel book)
{
    var index = books.FindIndex(b => b.Id == book.Id);
    if (index == -1)
    {
        return false;
    }

    books[index] = book;
    return true;
}
books.RemoveAt(index);
b ooks.Insert(index, book);
return true;
}

public void RemoveBook(int id)
{
    books.RemoveAll(b => b.Id == id);
}

Let’s examine the contents of this class:

• books: This is the private data of the class, which is the collection of all books in the library
• nextId: This is the identification number that will be used for the books that will be added in the library
• Name: This is the name of the library
• LibraryModel: In this constructor, we fill data (Name and books) with the initial values for demonstration purposes
• GetBooks: This gets all the books in the library
• GetBook(int id): This gets a book with a specific identification number, id
• AddBook(BookModel book): This adds a new book in the library
• UpdateBook(BookModel book): This updates book data (book matching is performed by the identification number)
• RemoveBook(int id): This removes a book with a specific identification number, id

Adding views

Now, we will create two views: one for the library overview (Index) and one to edit the information about a specific book (Edit).

1. First of all, let’s create the Library subfolder (it will be the name of our controller) in the Views folder.
2. Next, right-click on the Library folder and click on Add | View....
3. Set View name to Index.
4. Set View engine to Razor (CSHTML).
5. Check the Create a strongly-typed view checkbox.

The source code is as follows:

```csharp
@model HomeLibrary1.Models.LibraryModel
@{ ViewBag.Title = "Library"; }
<div class="container">
    <h2 style="text-align: center">@Model.Name</h2>
    <table class="table table-bordered table-striped table-condensed table-hover">
        <thead>
            <tr>
                <th>Column1</th>
                <th>Column2</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>Value1</td>
                <td>Value2</td>
            </tr>
            <tr>
                <td>Value3</td>
                <td>Value4</td>
            </tr>
        </tbody>
    </table>
</div>
```
Let's look at the code more carefully:

- The first line sets the type of model for the view (`LibraryModel`).
- The second line sets `ViewBag.Title` (this property is used in `_Layout.cshtml` to render the title element).
- The remaining lines describe the main layout of the page.
- The CSS classes (such as `container`, `table-striped`, `btn-primary`, and so on) are described in the Bootstrap framework and are needed for pretty HTML rendering. You can just ignore them and focus on the substantive part of the code.
- The `h2` element contains the name of the library (the `LibraryModel.Name` property).
- The list of books is represented as a table with four columns: three for the data (`Title`, `Author`, and `Year`) and one for the action button.
Integrating Knockout.js in ASP.NET MVC Applications

- We have two buttons per book: Edit and Remove (we will consider them in more detail with the controller discussion).
- Also, you can see the Add action button at the end of the page to add a new book.

We will also need another view for book editing. Let's call it Edit. You should add the corresponding class, as in the Index case, with one difference: Model class will be BookModel (HomeLibrary1.Models). The source code is as follows:

```csharp
@model HomeLibrary1.Models.BookModel
@{ ViewBag.Title = "Edit book"; }
<div class="container">
  <h2 style="text-align: center">Edit book</h2>
  @using (Html.BeginForm("Edit", "Library", FormMethod.Post, new { @class = "form-horizontal", role = "form" }))
  {
    <div class="form-group">
      @Html.LabelFor(m => m.Title, new { @class = "col-sm-2 control-label" })
      <div class="col-sm-10">
        @Html.TextBoxFor(m => m.Title, new { @class = "form-control" })
      </div>
    </div>
    <div class="form-group">
      @Html.LabelFor(m => m.Author, new { @class = "col-sm-2 control-label" })
      <div class="col-sm-10">
        @Html.TextBoxFor(m => m.Author, new { @class = "form-control" })
      </div>
    </div>
    <div class="form-group">
      @Html.LabelFor(m => m.Year, new { @class = "col-sm-2 control-label" })
      <div class="col-sm-10">
        @Html.TextBoxFor(m => m.Year, new { @class = "form-control" })
      </div>
    </div>
    </div>
  </div>
</div>
```
It is a simple form with three fields: Title, Caption, and Year. You can see the Save button at the end of the form to save the results. The CSS classes were also taken from the Bootstrap framework.

**Adding the Controller**

It's time to create the Controller. It is the main class of our application because it is responsible for the main logic and interaction between models and views. Let's create a new Controller and call it LibraryController (right click on the Controllers folder of the project in Solution Explorer and select Add | Controller…). In our case, we need to select the Empty MVC controller template, as shown in the following screenshot:
You should write the following code for the Controller:

```csharp
using HomeLibrary1.Models;
using System;
using System.Web.Mvc;

namespace HomeLibrary1.Controllers
{
    public class LibraryController : Controller
    {
        private static readonly LibraryModel Library = new LibraryModel();

        public ActionResult Index()
        {
            return View(Library);
        }

        [HttpGet]
        public ActionResult Edit(int id)
        {
            return View(Library.GetBook(id));
        }

        [HttpPost]
        public ActionResult Edit(BookModel book)
        {
            Library.UpdateBook(book);
            return RedirectToAction("Index");
        }

        public ActionResult Add()
        {
            var book = new BookModel
            {
                Title = "New Book",
                Author = "Unknown",
                Year = DateTime.Now.Year
            };
        }
    }
}
```
Library.AddBook(book);
return RedirectToAction("Index");
}

public ActionResult Remove(int id)
{
    Library.RemoveBook(id);
    return RedirectToAction("Index");
}
}

Let's discuss it in detail. Library is the main model of our library; it is a static field for demonstration purposes (in the real application, it will likely be a database or another external data source). The model methods are as follows:

- **Index()**: This method returns the Index view with our library overview logic
- **[HttpGet] Edit(int id)**: This method returns the Edit view to edit the book with the identification number id
- **[HttpPost] Edit(BookModel book)**: This method takes the edited book's data, updates it in the library, and returns the Index view
- **Add()**: This method adds a book in the library and returns the Index view
- **Remove(int id)**: This method removes the book with the identification number id and return the Index view

### Running the application

Only a final touch remains. Open the App_start/RouteConfig.cs file and update the line with the defaults parameters with the Library controller (see [http://msdn.microsoft.com/en-us/library/vstudio/cc668201(v=vs.100).aspx](http://msdn.microsoft.com/en-us/library/vstudio/cc668201(v=vs.100).aspx) for more information):

```csharp
defaults: new { controller = "Library", action = "Index", id = UrlParameter.Optional }
```
Integrating Knockout.js in ASP.NET MVC Applications

This means that our `LibraryController` will be the default controller when an application starts. Now, you can choose your favorite browser on the Visual Studio toolbar (see the following screenshot):

The work is finished! Just press `Ctrl + F5` and enjoy! You can see the result in the following screenshot.

The following is the main view:
This is the edit view:

In this section, we obtained a complete working application that supports CRUD operations. Explore the source code and play with the rendered pages. In the next section, we will improve this example with the power of Knockout.js. You should be able to easily navigate through your code before moving on the next step.

Creating an application with Knockout.js

In this section, we will update a sample with Knockout.js from the previous section. For this purpose, we will create a new project with similar functionality and reprocessed source code. We will compare two samples (with and without Knockout.js) to understand how Knockout.js can help us develop a rich ASP.NET MVC application the easy way.

Motivation

The current state of our HomeLibrary project has one fatal drawback: Add and Delete actions entail a full page reload. This has a negative impact on the volume of traffic and the speed of query execution.
In the real application, it can be a serious problem. Just imagine: you have a very big page and the user wants to update only a small part of it. In this case, it will be very sad if you reload the full page. Modern applications should be responsive. They should take a small volume of traffic and immediately respond to user actions. This is particularly important in view of the increased number of mobile users, who have weak devices and slow, limited traffic.

The standard way in this case is **Asynchronous JavaScript and XML (AJAX)**. It is a group of techniques that help you perform a background request to the server and use the response to update only one part of the page. However, implementing the fully updated logic manually on pure JavaScript is routine activity.

Fortunately, we can use Knockout.js to dynamically update the target part of the page according to declarative bindings.

**Creating a new project**

Let's create a new project in our solution from the previous section and call it **HomeLibrary2**. It also will be an ASP.NET MVC4 application that is based on the Basic template. We need the following libraries: **bootstrap**, **knockoutjs**, and **jQuery**. You can install these libraries from **Package Manager Console** using the following commands with **HomeLibrary2** as a default project:

```
PM> Install-Package bootstrap
PM> Install-Package knockoutjs
PM> Install-Package jQuery
```

If you already have some of these libraries in the project template, the given command will update it to the latest version. In this book, we will work with **bootstrap 3.2.0**, **knockoutjs 3.1.0**, and **jQuery 2.1.1**, but you can work with other versions of these libraries, as well. Let's add a link to the libraries in the head section of the **Views/Shared/_Layout.cshtml** file:

```
@Styles.Render("~/Content/bootstrap.css")
@Scripts.Render("~/Scripts/bootstrap.js")
@Scripts.Render("~/Scripts/knockout-3.1.0.js")
@Scripts.Render("~/Scripts/jquery-2.1.1.js")
```

**Adding models**

We will use the same model as in the previous section. You can just copy the code of the **BookModel** and **LibraryModel** classes to a new project. Just do not forget to change the namespace to **HomeLibrary2**.
Adding views

The Edit view will be taken from HomeLibrary1. We will use the Index view to demonstrate applying the Knockout.js approach for our application. Let's implement the following source code for the Index view:

```html
@model HomeLibrary2.Models.LibraryModel
@{ ViewBag.Title = "HomeLibrary"; }
<div class="container">
    <h2 style="text-align: center"><span data-bind="text: Name"></span></h2>
    <table class="table table-bordered table-striped table-condensed table-hover">
        <thead>
            <tr>
                <th>Title</th>
                <th>Author</th>
                <th>Year</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody data-bind="foreach: Books">
            <tr>
                <td data-bind="text: Title"></td>
                <td data-bind="text: Author"></td>
                <td data-bind="text: Year"></td>
                <td>
                    <a href="#" data-bind="click: $root.edit" class="btn btn-primary btn-xs">Edit</a>
                    <a href="#" data-bind="click: $root.remove" class="btn btn-primary btn-xs">Remove</a>
                </td>
            </tr>
        </tbody>
    </table>
    <a href="#" data-bind="click: add" class="btn btn-primary">Add new book</a>
</div>

<script type="text/javascript">
    var libraryViewModel = function () {
        var self = this;
        self.Name = ko.observable();
        self.Books = ko.observableArray();

        // Initial data
        $.ajax({
            url: '@Url.Action("GetName")',
            success: function (data) {
                self.Name(data.Name);
                // Bind Books to data
            }
        });
    }
</script>
```
Integrating Knockout.js in ASP.NET MVC Applications

```javascript
$.ajax({
    url: '@Url.Action("GetBooks")',
    cache: false,
    type: 'GET',
    contentType: 'application/json; charset=utf-8',
    data: {},
    success: function (data) {
        self.Name(data);
    }
});

$.ajax({
    url: '@Url.Action("GetBooks")',
    cache: false,
    type: 'GET',
    contentType: 'application/json; charset=utf-8',
    data: {},
    success: function (data) {
        self.Books(data);
    }
});

// Remove
self.remove = function (book) {
    var id = book.Id;
    $.ajax({
        url: '@Url.Action("Remove")',
        cache: false,
        type: 'POST',
        contentType: 'application/json; charset=utf-8',
        data: JSON.stringify({ id: id }),
        dataType: "json",
        success: function (data) {
            self.Books(data);
        }
    });
}

// Add
self.add = function () {
    $.ajax({
        url: '@Url.Action("Add")',
        cache: false,
        type: 'GET',
        contentType: 'application/json; charset=utf-8',
        data: {},
        success: function (data) {
            self.Books(data);
        }
    });
}
```
// Edit
self.edit = function (book) {
    var id = book.Id;
    location.href = "Library/Edit/" + id;
}

// Applying bindings
ko.applyBindings(new libraryViewModel());
</script>

There are some distinctions from the previous version of the view. The first distinction is migrating to Knockout.js. Indeed, we use the foreach binding instead of Razor @foreach for enumeration of books, text binding instead of Razor rendering methods for the main data (Name for our library and Title, Author, and Year for its books), and click binding instead of Razor @Html.ActionLink for actions.

Also, we define a large amount of JavaScript source code. In the code, you can see the creation of a ViewModel to store the server LibraryModel data (it is called libraryViewModel) and apply it using the ko.applyBindings(new libraryViewModel()) statement. Let's discuss the declaration of libraryViewModel in detail. This declaration starts with the following lines:

    var self = this;
    self.Name = ko.observable();
    self.Books = ko.observableArray();

The first line is our usual technique to work with this via its local alias self. The second and third lines define observable properties of ViewModel: Name and Books. These properties correspond to original properties of the server LibraryModel class.

After that, we start to load data from the server with the following code:

    $.ajax({
        url: '@Url.Action("GetName")',
        cache: false,
        type: 'GET',
        contentType: 'application/json; charset=utf-8',
        data: {},
        success: function (data) {
            self.Name(data);
        }
    });
The preceding code describes the AJAX request to the server. $ is the main jQuery object, $.ajax is a special method to perform the AJAX request the easy way. This method works with the following properties:

- **url**: This is the URL address for the request; we define it via Razor @Url.Action
- **cache**: For this, a false value will force the requested page to not be cached by the browser
- **type**: In this case, we perform the GET request to get some data from the server
- **contentType**: In this case, we want to get data in the JSON format with the UTF-8 charset
- **data**: This is the data that transforms to the server; it is empty in this case
- **success**: This is the callback function for the success case; after performing a request, we should update the Name property of the ViewModel

Likewise, we load the list of books. We also declare three methods for the ViewModel: remove, add, and edit.

The logic of the add method is very similar to the logic of loading books, but with one distinction: we change the parameters of url to @Url.Action("Add"). The Add server methods add a new book and return the full book list in the JSON format.

The logic of the remove method is also similar to the logic of loading books, but it has more distinctions. In addition to changing the parameters of url (to @Url.Action("Remove")), this method should transform the identification number id of the deleted book to the server. It means that we should change the request type from GET to POST and set the data parameter to JSON.stringify({ id: id }) (it is a special way to form the id value in the right JSON format).

The edit method is very simple. In this case, we don't need to perform any AJAX request, we just change location.href of the page to the editing page (the Edit view) of the book with the given identification number id.

## Adding the controller

In this section, we will write a controller that will contain the main application logic and discuss this logic in detail. Let's implement a new logic for our Controller:

```csharp
using System;
using System.Web.Mvc;
```
using HomeLibrary2.Models;

namespace HomeLibrary2.Controllers
{
    public class LibraryController : Controller
    {
        private static readonly LibraryModel Library = new LibraryModel();

        public ActionResult Index()
        {
            return View(Library);
        }

        public JsonResult GetName()
        {
            return Json(Library.Name, JsonRequestBehavior.AllowGet);
        }

        public JsonResult GetBooks()
        {
            return Json(Library.GetBooks(), JsonRequestBehavior.AllowGet);
        }

        [HttpGet]
        public ActionResult Edit(int id)
        {
            return View(Library.GetBook(id));
        }

        [HttpPost]
        public ActionResult Edit(BookModel book)
        {
            Library.UpdateBook(book);
            return RedirectToAction("Index");
        }

        public JsonResult Add()
        {
            var book = new BookModel
            {
            }
This code is very similar to the LibraryController from HomeLibrary1, but it has some important distinctions:

- The Add and Remove methods return JsonResult instead of ActionResult. This means that they return only the target data in the JSON format instead of the full HTML page. In this case, the target data is the book list of the library. The JSON data is returned with the following code:

```csharp
return Json(Library.GetBooks(),
            JsonRequestBehavior.AllowGet);
```

- The model has two additional methods: GetName and GetBooks. These methods are used for the initial data loading and also to return values in the JSON format.

### Running the application

Set the default controller (in the same way as we did in HomeLibrary1) and run the application. The external behavior is identical to the previous project, but inside there were some changes. Open your favorite tool for network analysis (for example, the internal tool in the Firefox browser) and remove some books. You will see a new request to the server (Remove).
Check that the response contains only the book list instead of the full page (see the following screenshot):

<table>
<thead>
<tr>
<th>Method</th>
<th>Headers</th>
<th>Cookies</th>
<th>Params</th>
<th>Response</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 POST Remove</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Summary**

In this chapter, we started with a very simple ASP.NET MVC application to store information about the home library. The first version of the application (HomeLibrary1) was developed in pure ASP.NET MVC without additional JavaScript code. After that, we updated the internal logic with the Knockout.js library (HomeLibrary2). Thanks to the AJAX request, we were able to interact with the user without regular full page reloads. This approach reduces traffic, improves the performance, and makes the design more responsive. Knockout.js provides us with an easy way to create a view with declarative bindings. We shouldn't worry about writing JavaScript code for UI update; Knockout.js would take care of this for us.

In the next chapter, we will update the acquired application with the Knockout MVC library. We will learn how Knockout MVC can help us to simplify the development of applications based on Knockout.js.
Where to buy this book

You can buy Getting Started with Knockout.js for .NET Developers 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.