Chapter 39
RESTful Services

39.1 Introduction

This chapter looks at RESTful Web services as implemented using the Play framework.

39.2 RESTful Services

As well as dynamic Web applications, many developers also want to be able to create RESTful services that can be invoked by Ajax style clients.

REST stands for Representational State Transfer and was a term coined by Roy Fielding in his Ph.D. to describe the lightweight, resource-oriented architectural style that underpins the Web. Fielding, one of the principle authors of HTTP, was looking for a way of generalising the operation of HTTP and the Web. He generalised the supply of Web pages as a form of data supplied on demand to a client where the client holds the current state of an exchange. Based on this state information the client requests the next item of relevant data sending all information necessary to identify the information to be supplied with the request. Thus the requests are independent and not part of an ongoing stateful conversation (hence state transfer). If you are interested in the background to this see

- http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

It should be noted that although Fielding was aiming to create a way of describing the pattern of behaviour within the Web, he also had an eye on producing lighter weight Web-based services (than those using either proprietary Enterprise Integration frameworks or SOAP-based services). These lighter weight HTTP-based Web services have become very popular and are now widely used in many areas. Systems which follow these principles are termed RESTful services.

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_39

A key aspect of a RESTful service is that all interactions between a client (whether some JavaScript running in a browser or a standalone application) are done using simple HTTP-based operations. HTTP supports four operations: HTTP GET, HTTP POST, HTTP PUT and HTTP DELETE. These can be used as verbs to indicate the type of action being requested. Typically these are used as follows:

- retrieve information (HTTP GET)
- create information (HTTP POST)
- update information (HTTP PUT)
- delete information (HTTP DELETE).

It should be noted that REST is not a standard in the way that HTML is a standard. Rather it is a Design Pattern that can be used to create Web applications that can be invoked over HTTP and that give meaning to the use of GET, POST, PUT and DELETE HTTP operations with respect to a specific resource (or type of data).

The advantage of using RESTful services as a technology, compared to some other approaches (such as SOAP-based services which can also be invoked over HTTP) is that

- the implementations tend to be simpler,
- the maintenance easier.
- They run over standard HTTP and HTTPS protocols and
- do not require expensive infrastructures and licences to use.

This means that there is lower server and server-side costs. There is little vendor or technology dependency, and clients do not need to know anything about the implementation details or technologies being used to create the services.

39.3 A RESTful API

A RESTful API is one in which you must first determine the key concepts or *resources* being represented or managed. These might be books, products in a shop, room bookings in hotels, etc. In our case we will assume a bookstore-related service in which the data being held represents types of books, CDs, DVDs, etc. This data is split into resources with books as one type of resource. We will ignore the other resources such as DVDs and CDs. Based on these books we will identify suitable URLs for these RESTful services. Note that although URLs are frequently used to describe a web page—that is just one type of resource. For example, we might develop a resource such as

/bookservice/book

from this we could develop a URL based API, such as

/bookservice/book/:isbn

Where ISBN indicates a unique number to be used to identify a specific book whose details will be returned using this URL.

We also need to design the representation or formats that the service can supply. These could include plain text, JSON, XML. JSON stands for the JavaScript Object Notation and it is a concise way to describe data that is to be transferred from a service running on a server to a client running in a browser. This is the format we will use in the next section. As part of this we might identify a series of operations to be performed by our services based on the type of HTTP method used to invoke our service and the contents of the URL provided. For example, for a simple BookService this might be:

- GET /book/{isbn}—used to retrieve a book for a given ISBN
- GET /book/list.xml—used to retrieve all current books in XML format
- GET /book/list.json—used to retrieve all current books in JSON format
- POST /book (XML or JSON in body of the message)—which supports creating a new book
- PUT /book (XML or JSON in body of message)—used to update the data held on an existing Book
- DELETE /book/{isbn}—used to indicate that we would like a specific book deleted from the list of books held.

Note that the *parameter* ISBN in the above URLs actually forms part of the URL path.

39.4 Creating the RESTful Web Application

We will create a new Play application for this. This is done in exactly the same way as before. In fact Play creates Web-based applications. These can be treated as websites or as RESTful services depending upon the type of data supplied and the way in which you expect the service to be invoked. Thus the distinction between a RESTful service and a Web page is in the eye of the beholder (or at least the client application). Indeed if you wish rather than unzipping the start project again you can choose to modify the simple application we created in the last chapter.

Given the Model–View–Controller structure described earlier we need the model for our RESTful services. This will be defined in a separate package model and comprise a Book case class and a Books object. This can be accessed by our controller as and when required. The class and object that comprise the model package are shown below:

```
package model

case class Book(val isbn: Int,
                val title: String,
                val author: String,
                val price: Double)

object Books {

  val bookList = List(Book(1, "XML", "S. Smith", 10.99),
    Book(2, "Java", "J. Jones", 12.99),
    Book(3, "Scala", "A. Adams", 11.99))

  def get(isbn: Int): Book = {
    val result = bookList.filter(_.isbn == isbn)
    result.head
  }

}
```

Note that the `Books` object defines both a property `list` and a method `get`. The `list` property holds the list of books currently available, and the `get` method returns a book given a specified ISBN.

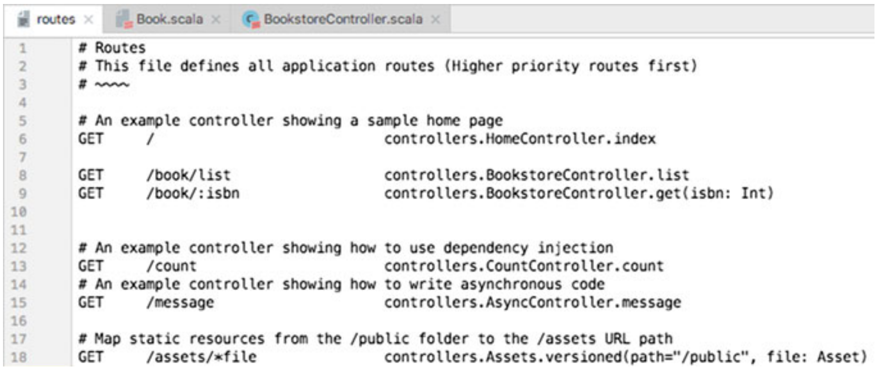
However, we now need to determine what the HTTP requests we will support will be. Our bookstore service will be a read-only service which will provide a list of books or a single book if an ISBN is supplied. As this is an access-oriented service we only need to support the HTTP GET requests. We therefore want to provide mappings from

- /book/list
- /book/:isbn

Both of these routes can be supported by the same object, the `controllers.Bookstore` object via methods `list` and `get`; thus we will map:

- /book/list -> controllers.BookstoreController.list
- /book/:isbn -> controllers.BookstoreController.get(isbn: Int)

Note that the routing information will be used to map the isbn information provided as part of the/book/:isbn URL to the parameter passed into the `get` method. We will thus update our routes file as illustrated below.



```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# An example controller showing a sample home page
GET     /                           controllers.HomeController.index

GET     /book/list                  controllers.BookstoreController.list
GET     /book/:isbn                 controllers.BookstoreController.get(isbn: Int)


# An example controller showing how to use dependency injection
GET     /count                      controllers.CountController.count
# An example controller showing how to write asynchronous code
GET     /message                    controllers.AsyncController.message

# Map static resources from the /public folder to the /assets URL path
GET     /assets/*file               controllers.Assets.versioned(path="/public", file: Asset)
```

Next we need to write the `BookstoreController`. As with the original HomeController controller in the last chapter, our `BookstoreController` extends the `Controller` type. The initial state of the object is shown below:

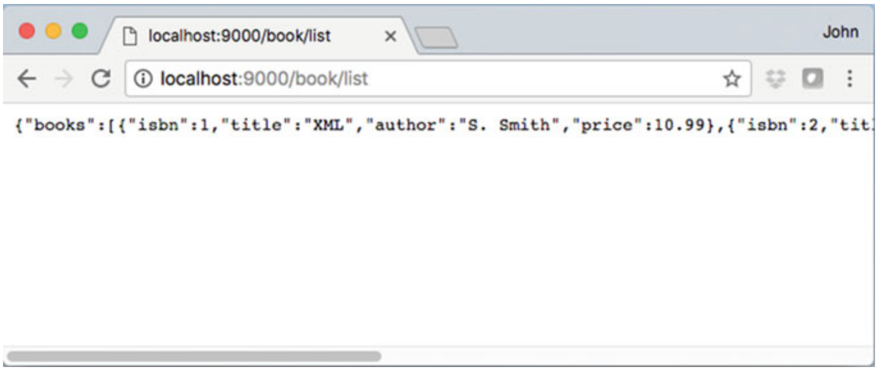
```
package controllers

import javax.inject.Inject

import play.api.mvc.AbstractController
import play.api.mvc.ControllerComponents

class BookstoreController
        @Inject()(cc: ControllerComponents)
                extends AbstractController(cc) {

}
```

The `list` method to be added to the `BookstoreController` will take no parameters and implement an `Action` that will return a list of Books. We will use the JSON data format as it is very widely used within Web-based services. However, we need to specify how an instance of the `Book` class should be converted into a JSON object. We will do this using the `toJson` method that is designed to convert a `Book` instance into a JSON object by extracting the isbn, title, author and price from a book and wrapping them up within an appropriate JSON type. The Play frameworks' JSON library is defined within `play.api.libs.json` and contains case classes for JSON types such as `JsString`, `JsNumber`, `JsBoolean`, `JsObject`, `JsArray` and `JsNull`.

```
package controllers

import javax.inject.Inject

import play.api.mvc.AbstractController
import play.api.mvc.ControllerComponents

import play.api.libs.json._

import model.Books
import model.Book

class BookstoreController @Inject()(cc: ControllerComponents) extends
AbstractController(cc) {

  // Utility methods
  private def booksAsJsonList: JsObject =
    JsObject("books" -> JsArray(Books.bookList.map { book => toJson(book)}) ::
Nil)

  private def toJson(book: Book) = {
    JsObject(
      "isbn" -> JsNumber(book.isbn) ::
        "title" -> JsString(book.title) ::
        "author" -> JsString(book.author) ::
        "price" -> JsNumber(book.price) :: Nil)
  }

  // Request Handling mehtods
  def list = Action {
    Ok(booksAsJsonList)
  }

  def get(isbn: Int) = Action {
    Ok(toJson(Books.get(isbn)))
  }

}
```

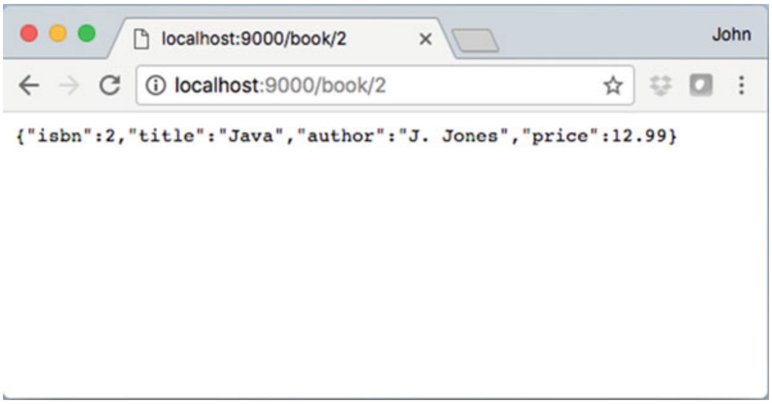
The `get` method will take an ISBN. The type of the ISBN needs to be determined. We will assume that all our ISBNs are integers and thus the type of the parameter for the `get` method will be `Int`. The `get` method uses this ISBN number to return a book with that ISBN number or a message indicating that no book with that number was found.

Note that the `JsObject` constructs a JSON object comprised of key-value pairs that are used to represent the actual `Book`. This type is obtained from the `play.api.libs.json` package. The `JsString` and `JsNumber` are also defined by that package.



```
{"books":[{"isbn":1,"title":"XML","author":"S. Smith","price":10.99},{"isbn":2,"tit
```

Fig. 39.1 Data returned from the book service



```
{"isbn":2,"title":"Java","author":"J. Jones","price":12.99}
```

Fig. 39.2 Retrieving a single book details from the book service

If we now invoke this RESTful service from a client browser, we can see the data returned. For example, if you enter into the browser:

http://www.localhost:9000/book/list

you should see the result presented in Fig. 39.1. The data returned is in JSON format and indicates that the property *books* relates to an array of book information. Each item of book information is comprised of an isbn, a title, an author and a price.

We can also obtain information on a single book using the url/books/:isbn routing information; here the isbn number forms part of the URL—thus indicating a (dynamically generated) resource, for example, using the URL:

http://www.localhost:9000/book/2

The information about the book with the ISBN 2 is returned as shown in Fig. 39.2.

Alternatively, we could construct a client using JavaScript that would be run within a browser. This client could invoke the RESTful service we have just created asynchronously when requested to do so by the user.

39.5 JavaScript and jQuery

The Web service we have created supplies data that can be consumed by a suitable client. In many cases these clients will be implemented in JavaScript and executed within the client-side browser (such as Chrome and Firefox). Note that the server-based Scala code will execute within the confines of the server, whereas the client will run within a browser potentially anywhere in the world.

A very common approach is to create a Web page containing some JavaScript that will, in response to user actions, such as clicking a button, asynchronously invoke the remote service and populate the current Web page with the data provided. For example, if you are requesting your recent bank transactions, the request for those transactions occurs in page without the need for the whole page to be refreshed.

JavaScript itself is a programming language that can be executed within a wide range of environments. The most common environment to run JavaScript in is within a browser, where a JavaScript engine interprets it. As such JavaScript is commonly treated as a client-side programming language to be embedded in a Web page and interacts with the contents of that Web page (represented as the Document Object Model or DOM of the page).

A common library to use with such applications is jQuery. This is because it simplifies many activities that you would need to develop from scratch if you were using JavaScript on its own (such as the look and feel generation, calendars and Ajax style programming). The basic concept behind Ajax is that data can be retrieved from a server asynchronously in the background, without interfering with the display and behaviour of the existing page. The name original came from the acronym for *Asynchronous JavaScript and XML* programming. However, Ajax style applications are very widely used with JSON rather than XML as the data interchanges format. It should be noted that Ajax as such is not a technology; rather it is a set of technologies that are used together in a particular "Design Pattern" or implementation strategy. This approach typically utilises:

- XHTML and CSS for presentation
- The Document Object Model (DOM) for dynamic display of and interaction with data
- XML, JSON or plain text for the interchange, manipulation and display of data
- Facilities in JavaScript for asynchronous communication
- JavaScript to bring these technologies together.

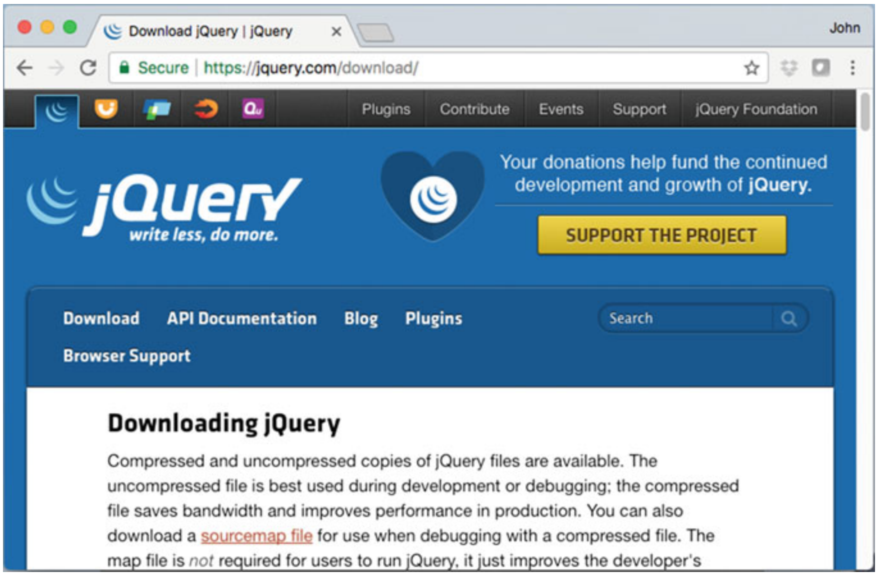
jQuery makes it easy to implement an Ajax style client and to use the data provided by remote services to change the current Web page.

39.6 The jQuery Client

We will create a simple jQuery-based client. This client will be loaded when an initial Web page is displayed to the user. The Web page will be created using the Play framework. This allows us to dynamically determine some of the information to display in this page and links the client to the back-end service.

39.6.1 Obtaining jQuery

jQuery can be downloaded from https://jquery.com/download/.



Downloading jQuery

Compressed and uncompressed copies of jQuery files are available. The uncompressed file is best used during development or debugging; the compressed file saves bandwidth and improves performance in production. You can also download a sourcemap file for use when debugging with a compressed file. The map file is *not* required for users to run jQuery, it just improves the developer's

jQuery is a fast, small and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation and Ajax much simpler with an easy-to-use API that works across a multitude of browsers.

39.6.2 Adding jQuery to the Application

We will first modify our `HomeController` class so that the string passed to the index view template provides the name of the bookstore. This is shown below:

```
package controllers

import javax.inject._
import play.api.mvc._

@Singleton
class HomeController @Inject()(cc: ControllerComponents) extends
AbstractController(cc) {

  def index = Action {
    Ok(views.html.index("Johns Bookstore"))
  }

}
```

Next, we will rewrite the `index.scala.html` template file such that it uses more of the templating features of the Play framework. The implementation of our view is presented below.

The head of the template still indicates the information being provided to it. This is the string passed to the view from the `Application` controller. However, this string is now used within an HTML Web page. The Web page uses the string in the welcome heading presented to the user. The rest of the template is presented below:

```
@(message: String)

<html lang="en">
<head>
 <meta charset="utf-8">
 <title>jQuery.getJSON demo</title>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
 <link rel="stylesheet" href="@routes.Assets.versioned("stylesheets/main.css")"
 type="text/css" media="screen"></link>
 <link rel="stylesheet" href="@routes.Assets.versioned("stylesheets/bookshop-
style.css")"
      type="text/css"></link>
 <script src="@routes.Assets.versioned("javascripts/jquery-3.2.1.js")"></script>
  <script type="text/javascript"
src="@routes.Assets.versioned("javascripts/myscript.js")"></script>
</head>

<body>

<h2>Welcome @message</h2>

<div class="button" id="show"> Show </div>
<div id="books"></div>

</body>
</html>
```

An HTML page can be divided into a header and a body. The *header* tells the browser information about the page, and the *body* provides the data to be displayed to the user.

The above Web page contains a simple HTML body that merely displays a welcome message and places a string shown in a divider and an empty divider called books within the page. Note that use of `@message` indicates that this is a template where the value of the message will be provided by the Application controller at runtime. However, the interesting parts are in the HTML header. This header includes:

```
<link rel="stylesheet" href="@routes.Assets.versioned("stylesheets/main.css")"
 type="text/css" media="screen"></link>
 <link rel="stylesheet" href="@routes.Assets.versioned("stylesheets/bookshop-
style.css")"
      type="text/css"></link>
 <script src="@routes.Assets.versioned("javascripts/jquery-
3.2.1.js")"></script>
  <script type="text/javascript"
src="@routes.Assets.versioned("javascripts/myscript.js")"></script>
</head>
```

The links specify the CSS style sheet to use. This style sheet defines various colours and formats to be used with elements within the Web page. For example, the bookshop-style.css file defines the button style. This style is used within the body of the HTML page in:

```
<div class="button" id="show"> Show </div>
```

which indicates that the button CSS style class will be used with the text Show. Note that the location of the style sheets is relative to the Web application we are writing and thus the `@routes.Assets.versioned()` function is used to generate the actual path to the style sheet directory (this is also used below to access the jQuery files).

The notable aspect of the header is that it specifies two scripts to use. The first is the jQuery library itself (version 3.2.1), and the second is the file containing the custom code that implements our jQuery client.

The contents of the myscript.js file are presented below. This jQuery program does two things. It first links the hover function to the HTML element with the id “show” such that when the user moves the cursor over the *button* the cursor changes—which provides some useful feedback to the user. The second thing it does is specifies what should happen when the user *clicks* on the button.

```
$(function() {
  $(function() {
  $("div#show").hover(function() {
      $(this).addClass("hover");
    }, function() {
    $(this).removeClass("hover");
  });
  $("div#show").click(
    function() {
      $.get('book/list', function(data) {
     $('div#books').empty();
        $(data.books).each(
       function() {
         var $book = $(this);
         var html = "<div class='book'>";
         html += "<h3 class='title'>"
              + $book.attr('title') + "</h3>";
         html += "Author: " + $book.attr('author');
         html += "<br>Price: "
              + $book.find('price').text();
         html += "<br>ISBN: "
           + $book.find('ISBN').text();
         html += '</div>';
         $('div#books').append($(html));
      });
    });
    });
  });
});
```

The function defined for the user ‘click’ event on the element called *show* performs an asynchronous get request to the URL books/list. When the data returns, the associated second function executes. The data returned from the RESTful service will be supplied to the variable *data*. The second function initially removes anything that is currently being displayed by the element *books*. It then looks through the array of books indexed by the value books in the JSON object structure. For each book in that array it creates some HTML to format the book information and retrieves the appropriate data items from the book structure (such as title, author, price). The resulting information is then appended to the *books* element within the Web page.

The above files are stored under the *public* area of Web application project structure. This is where static assets of the project are location such as images, style sheets, JavaScript files and HTML files. In our case the style sheet is under the *stylesheets* directory and the jQuery and myscript.js files are under the *javascript* directory. This is illustrated in Fig. 39.3.

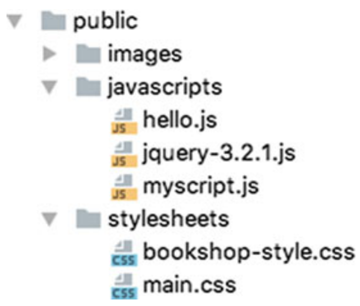
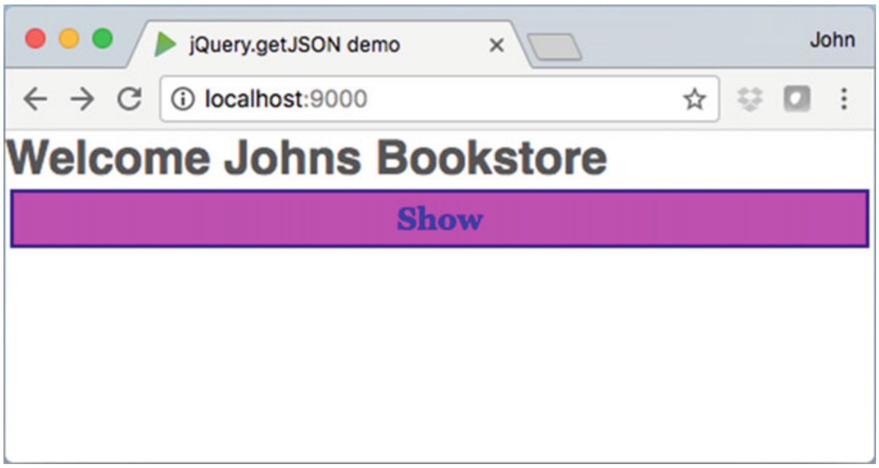


Fig. 39.3 Location of the assets used in the web application

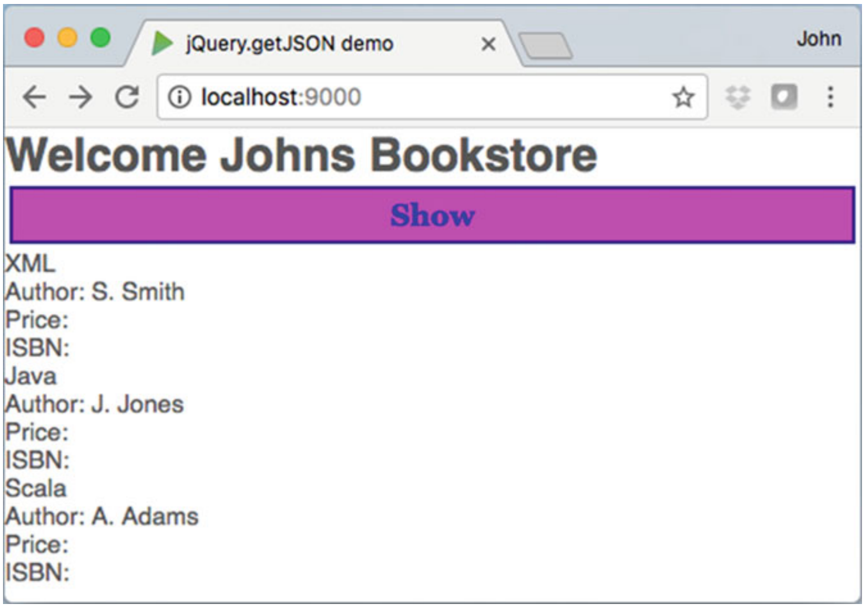
As Play automatically reloads the changes you make, all you need to do to see the behaviour of your jQuery client is to change the URL in your browser such that the address is:

http://www.localhost:9000

This will run the index (or default) page of your application that should now display the simplified Web page welcoming you to Johns Bookstore:



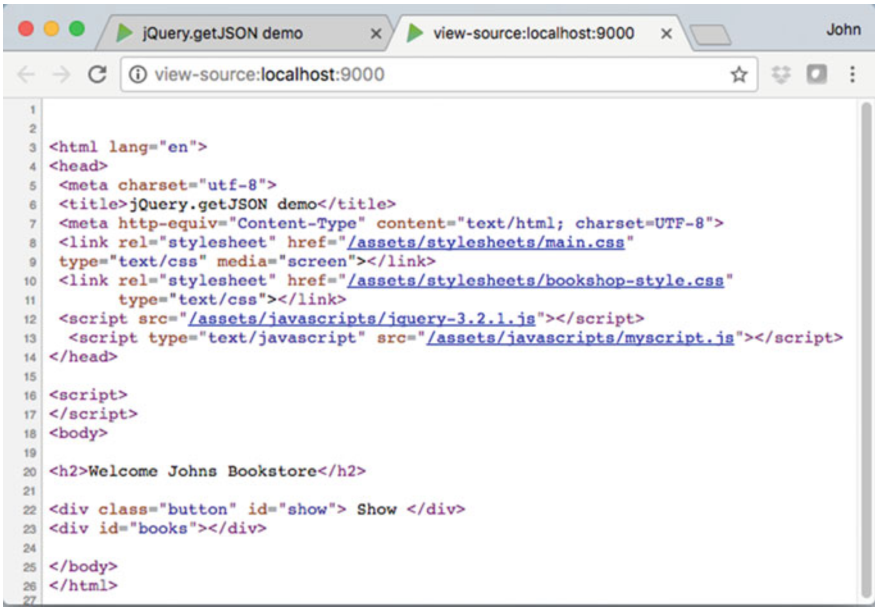
If you move your mouse over the *show* button you should see it change. Now click on the show button, and the page should (in the background) request the data from the bookstore service and display each of the books within the current page. This is shown below:



Welcome Johns Bookstore

XML
Author: S. Smith
Price:
ISBN:
Java
Author: J. Jones
Price:
ISBN:
Scala
Author: A. Adams
Price:
ISBN:

Interestingly if you view the source code for this page you will see that there are no books listed—this is because they were dynamically obtained from the server and added to that page by the jQuery based functions and are thus not part of the static HTML of the page. This is illustrated here:



```


<html lang="en">
<head>
 <meta charset="utf-8">
 <title>jQuery.getJSON demo</title>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
 <link rel="stylesheet" href="/assets/stylesheets/main.css"
 type="text/css" media="screen"></link>
 <link rel="stylesheet" href="/assets/stylesheets/bookshop-style.css"
       type="text/css"></link>
 <script src="/assets/javascripts/jquery-3.2.1.js"></script>
  <script type="text/javascript" src="/assets/javascripts/myscript.js"></script>
</head>

<script>
</script>
<body>

<h2>Welcome Johns Bookstore</h2>

<div class="button" id="show"> Show </div>
<div id="books"></div>

</body>
</html>
```

JQuery Online References

http://www.jquery.com—jQuery homepage
http://docs.jquery.com/Tutorials—Tutorials
http://www.learningjquery.com/—jQuery tutorial blog
http://docs.jquery.com/Sites_Using_jQuery—jQuery Success Stories