

Lesson 11

These exercises is an adaption for this course of the tutorial [Using Web API 2 with Entity Framework 6](#) by Mike Wasson. The exercises use ASP.NET Web API 2 with Entity Framework 6 to create a web application that manipulates a back-end database. Here is a screen shot of the application that you will create.

We will use ASP.NET Web API 2 with Entity Framework 6 to create a web application that manipulates a back-end database. Here is a screen shot of the application that you will create.

The screenshot shows a web browser window titled "Edit | MyBooks" with the address bar displaying "localhost:63830/#/books/3". The page has a header "MyBooks" and a main heading "Testing the Book Web API from Aurelia". Below the heading are three input fields: "Id" with the value "3", "AuthorId" with the value "2", and "Title" with the value "David Copperfield". Under these fields are four buttons: "getBooks() GET", "insertBook() PUT", "updateBook() POST", and "deleteBook() DELETE". Below the buttons is a section titled "Display Book Records" containing a table with four rows of book data. Each row has an "Id", "Title", "Author", and an "Edit" button.

Id	Title	Author	
1	Pride and Prejudice	Jane Austen	Edit
2	Northanger Abbey	Jane Austen	Edit
3	David Copperfield	Charles Dickens	Edit
4	Don Quixote	Miguel de Cervantes	Edit

The app uses a single-page application (SPA) design. "Single-page application" is the general term for a web application that loads a single HTML page and then updates the page dynamically, instead of loading new pages. After the initial page load, the app talks with the server through AJAX requests. The AJAX requests return JSON data, which the app uses to update the UI.

AJAX isn't new, but today there are JavaScript frameworks that make it easier to build and maintain a large sophisticated SPA application. This tutorial uses [Aurelia](#), but you can use any JavaScript client framework.

Here are the main building blocks for this app:

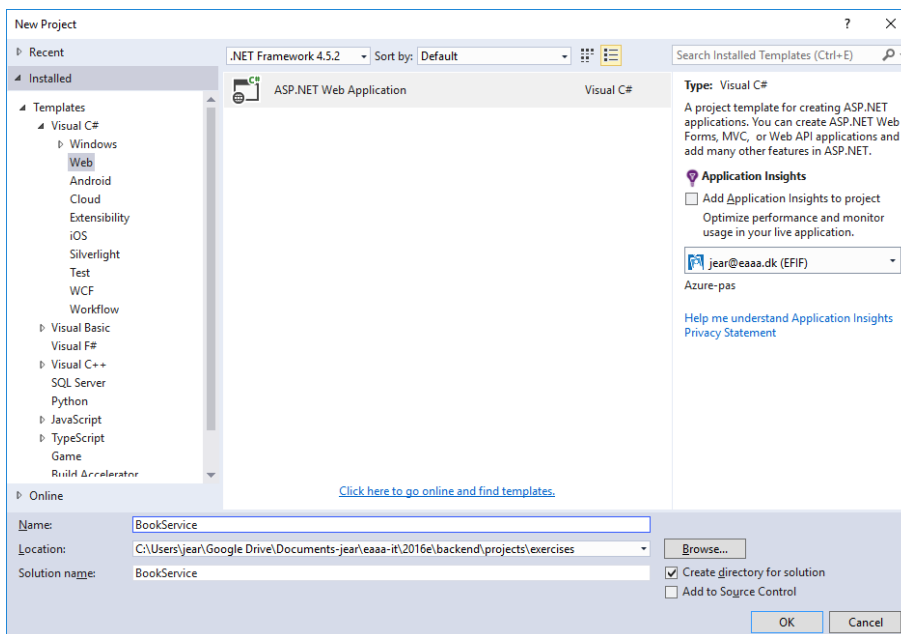
- ASP.NET MVC creates the HTML page.
- ASP.NET Web API handles the AJAX requests and returns JSON data.
- Aurelia data-binds the HTML elements to the JSON data.
- Entity Framework talks to the database.

Exercise 1, Create the project

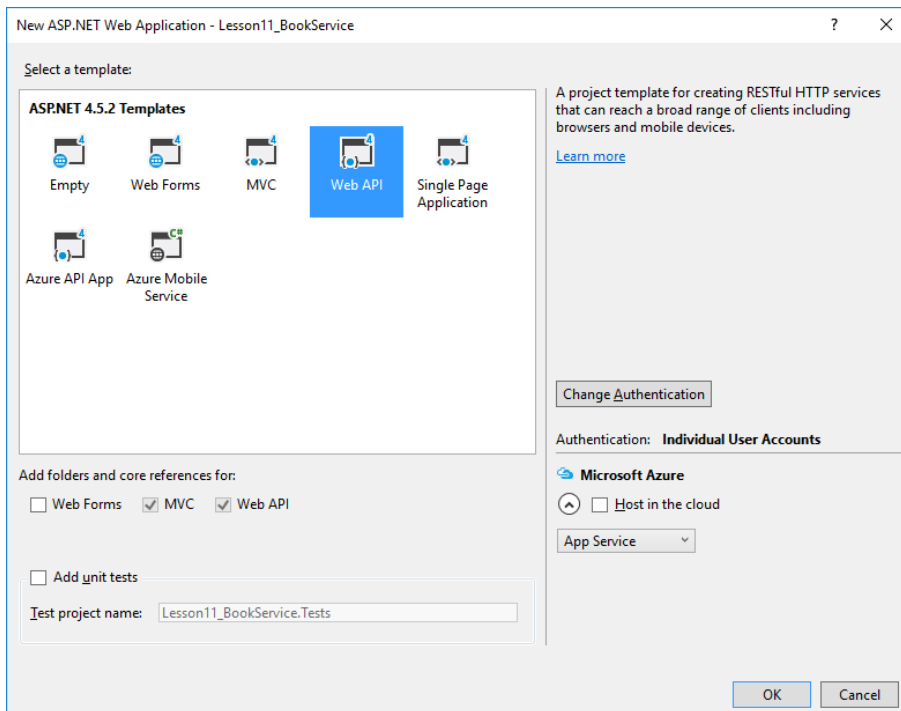
Open Visual Studio. From the **File** menu, select **New**, then select **Project**. (Or click **New Project** on the Start page.)

In the **New Project** dialog, click **Web** in the left pane and **ASP.NET Web Application** in the middle pane.

Name the project *BookService* and click **OK**.



In the **New ASP.NET Project** dialog, select an Empty project with folders and core references for **MVC** and **Web API**



We'll create a client website for the Web API service and we will use the MVC folder structure for that.

Exercise 2, Add models

In this exercise, you will add model classes that define the database entities. Then you will add Web API controllers that perform CRUD operations on those entities.

We start by defining our domain objects as POCOs (plain-old CLR objects). We will create the following POCOs:

- Author
- Book

In Solution Explorer, right click the Models folder. Select **Add**, then select **Class**. Name the class Author.

Replace all of the boilerplate code in Author.cs with the following code.

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace BookService.Models
{
    public class Author
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
    }
}
```

Add another class named Book, with the following code.

```
using System.ComponentModel.DataAnnotations;

namespace BookService.Models
{
    public class Book
    {
        public int Id { get; set; }
        [Required]
        public string Title { get; set; }
        public int Year { get; set; }
        public decimal Price { get; set; }
        public string Genre { get; set; }

        // Foreign Key
        public int AuthorId { get; set; }
        // Navigation property
        public Author Author { get; set; }
    }
}
```

Entity Framework will use these models to create database tables. For each model, the **Id** property will become the primary key column of the database table.

In the Book class, the AuthorId defines a foreign key into the Author table. (For simplicity, I'm assuming that each book has a single author.) The book class also contains a navigation property to the related Author. You can use the navigation property to access the related Author in code.

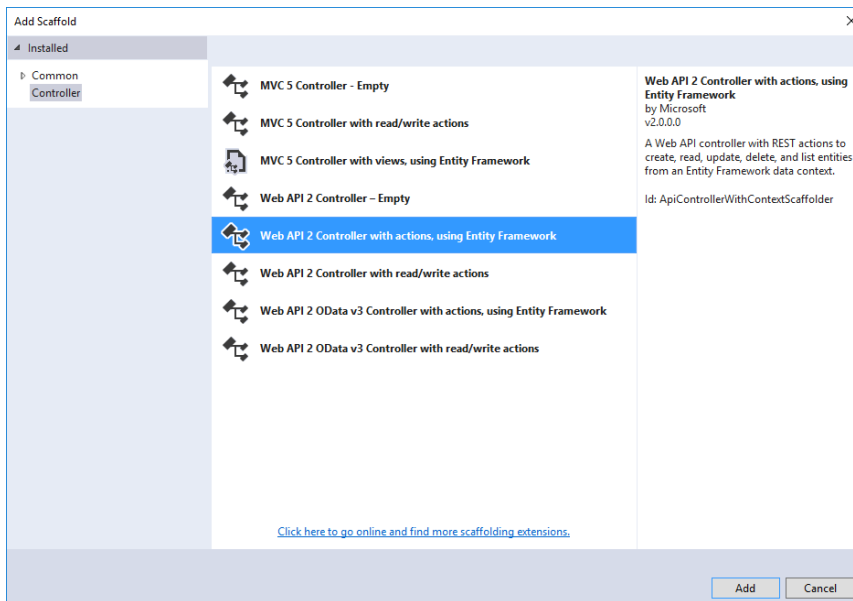
Exercise 3, Add Web API Controllers

In this exercise, we'll add Web API controllers that support CRUD operations (create, read, update, and delete). The controllers will use Entity Framework to communicate with the database layer.

First, build the project (CTRL+SHIFT+B). The Web API scaffolding uses reflection to find the model classes, so it needs the compiled assembly.

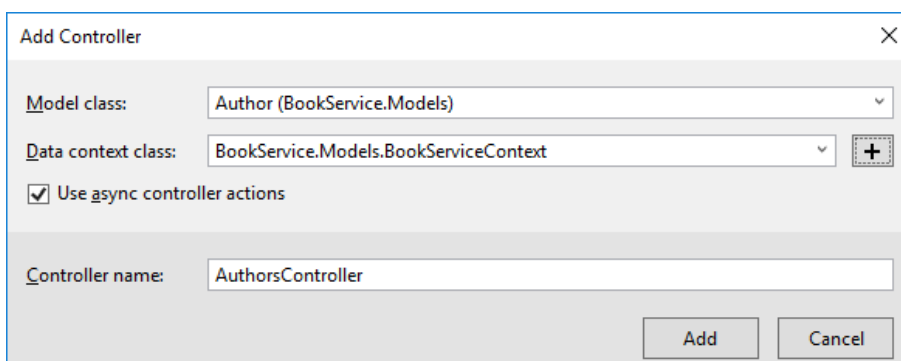
In the Solution Explorer, right-click the Controllers folder. Select **Add**, then select **Controller**.

In the **Add Scaffold** dialog, select "Web API 2 Controller with actions, using Entity Framework". Click **Add**.

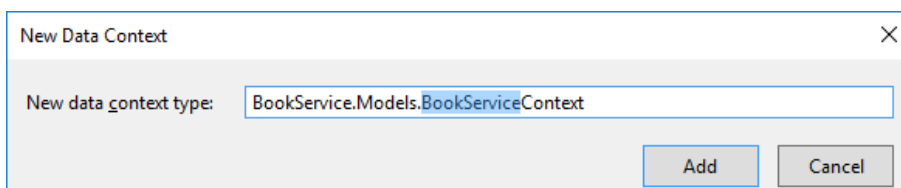


In the **Add Controller** dialog, do the following:

1. In the **Model class** dropdown, select the Author class. (If you don't see it listed in the dropdown, make sure that you built the project.)
2. Check "Use async controller actions".
3. Leave the controller name as "AuthorsController".
4. Click plus (+) button next to **Data Context Class**.

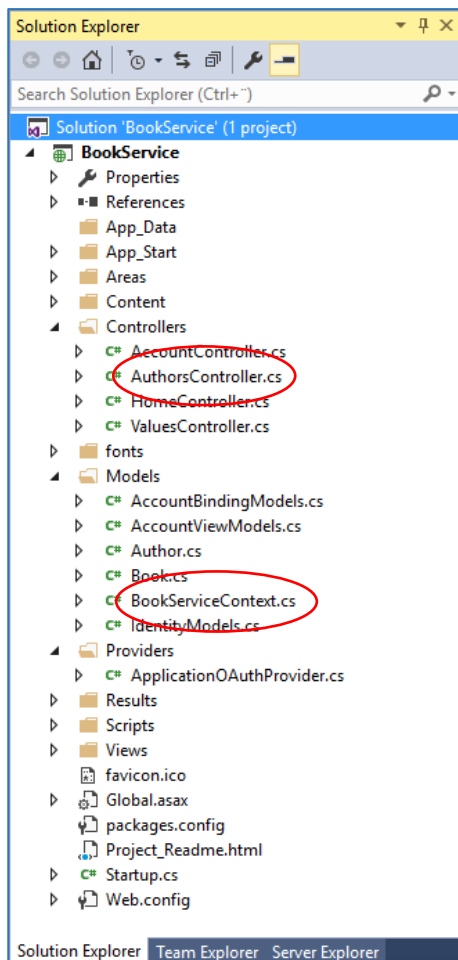


In the **New Data Context** dialog, leave the default name and click **Add**.



Click **Add** to complete the **Add Controller** dialog. The dialog adds two classes to your project:

- `AuthorsController` defines a Web API controller. The controller implements the REST API that clients use to perform CRUD operations on the list of authors.
- `BookServiceContext` manages entity objects during run time, which includes populating objects with data from a database, change tracking, and persisting data to the database. It inherits from `DbContext`.

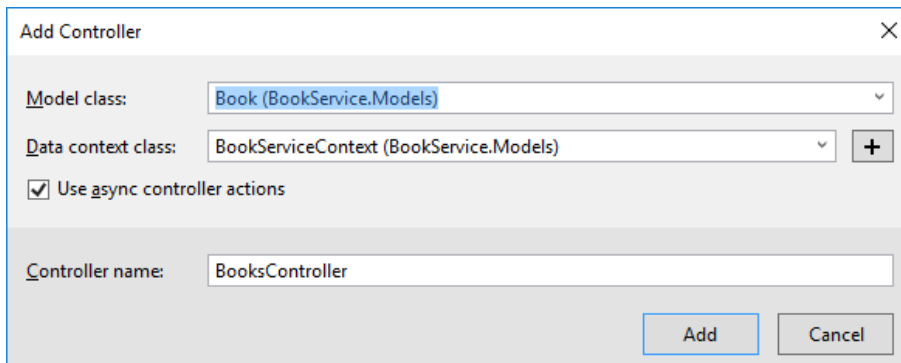


If you open the `web.config` file in the root folder, you can see the connection string the API controller scaffolder generated for you:

```
<add name="BookServiceContext" connectionString="Data Source=(localdb)\MSSQLLocalDB; Initial Catalog=BookServiceContext-20161121153909; Integrated Security=True; MultipleActiveResultSets=True; AttachDbFilename=|DataDirectory|BookServiceContext-20161121153909.mdf" providerName="System.Data.SqlClient" />
```

You can change the name of the database if you want it.

At this point, build the project again. Now go through the same steps to add an API controller for Book entities. This time, select **Book** for the model class, and select the existing `BookServiceContext` class for the data context class. (Don't create a new data context.) Click **Add** to add the controller.



Exercise 4, Use Code First migrations and seed the database

In this section, you will use [Code First Migrations](#) in EF to seed the database with test data.

From the **Tools** menu, select **Library Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

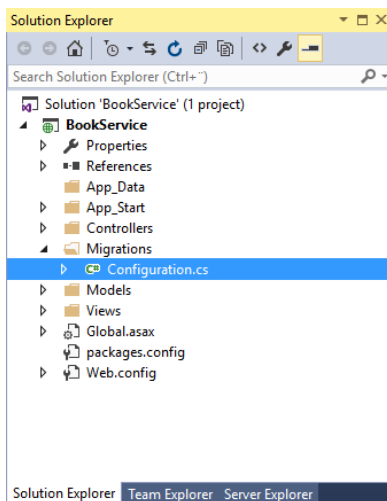
```
PM> Enable-Migrations
```

Or

```
PM> Enable-Migrations -ContextTypeName BookService.Models.BookServiceContext
```

If you have more than one DbContext type.

This command adds a folder named Migrations to your project, plus a code file named Configuration.cs in the Migrations folder.



Open the Configuration.cs file. Add the following **using** statement.

```
using BookService.Models;
```

Then add the following code to the **Configuration.Seed** method:

```
protected override void Seed(BookService.Models.BookServiceContext context)
{
    context.Authors.AddOrUpdate(x => x.Id,
        new Author() { Id = 1, Name = "Jane Austen" },
        new Author() { Id = 2, Name = "Charles Dickens" },
        new Author() { Id = 3, Name = "Miguel de Cervantes" }
    );

    context.Books.AddOrUpdate(x => x.Id,
        new Book()
        {
            Id = 1,
            Title = "Pride and Prejudice",
            Year = 1813,
            AuthorId = 1,
            Price = 9.99M,
            Genre = "Comedy of manners"
        },
        new Book()
        {
            Id = 2,
            Title = "Northanger Abbey",
            Year = 1817,
            AuthorId = 1,
            Price = 12.95M,
            Genre = "Gothic parody"
        },
        new Book()
        {
            Id = 3,
            Title = "David Copperfield",
            Year = 1850,
            AuthorId = 2,
            Price = 15,
            Genre = "Bildungsroman"
        },
        new Book()
        {
            Id = 4,
            Title = "Don Quixote",
            Year = 1617,
            AuthorId = 3,
            Price = 8.95M,
            Genre = "Picaresque"
        }
    );
}
```

In the Package Manager Console window, type the following commands:

```
PM> Add-Migration Initial
```

Open the initial migration file (201610201455534_Initial) in the migrations folder and inspect its content:

```
namespace BookService.Migrations
{
    using System;
```



```
using System.Data.Entity.Migrations;

public partial class Initial : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Authors",
            c => new
            {
                Id = c.Int(nullable: false, identity: true),
                Name = c.String(nullable: false),
            })
            .PrimaryKey(t => t.Id);

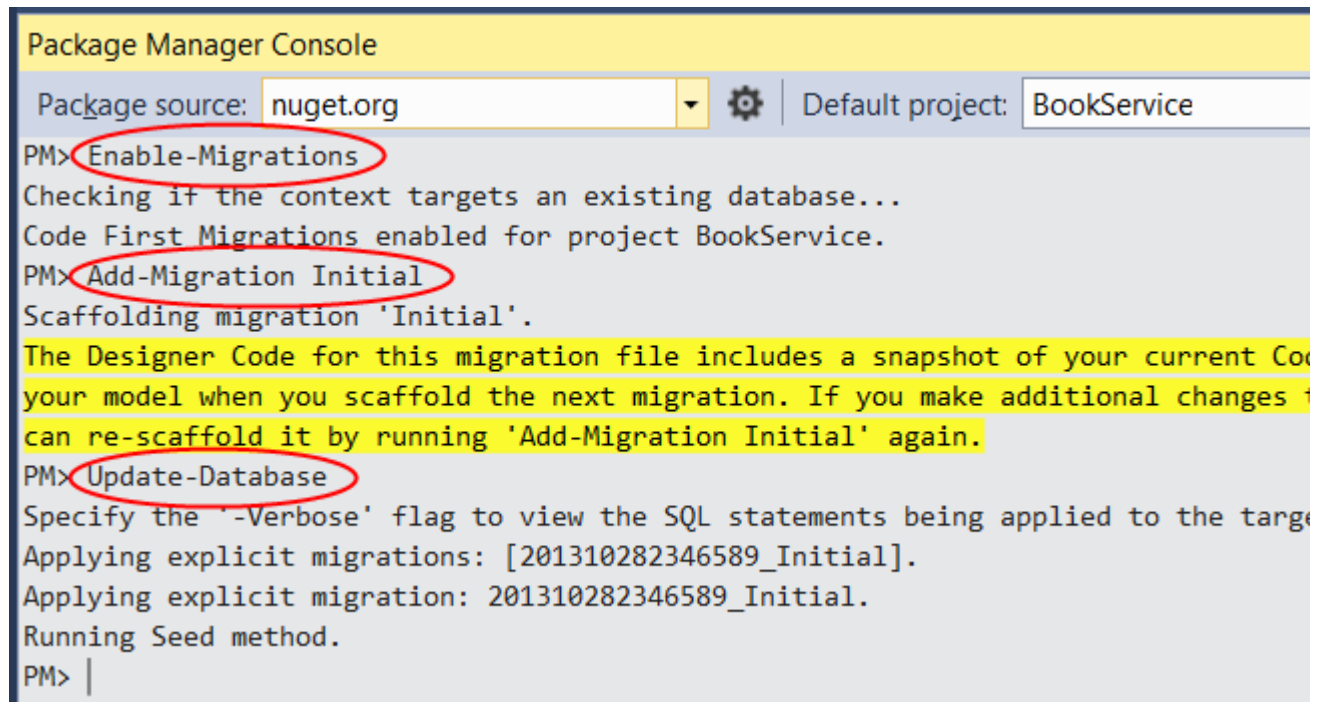
        CreateTable(
            "dbo.Books",
            c => new
            {
                Id = c.Int(nullable: false, identity: true),
                Title = c.String(nullable: false),
                Year = c.Int(nullable: false),
                Price = c.Decimal(nullable: false, precision: 18, scale: 2),
                Genre = c.String(),
                AuthorId = c.Int(nullable: false),
            })
            .PrimaryKey(t => t.Id)
            .ForeignKey("dbo.Authors", t => t.AuthorId, cascadeDelete: true)
            .Index(t => t.AuthorId);
    }

    public override void Down()
    {
        DropForeignKey("dbo.Books", "AuthorId", "dbo.Authors");
        DropIndex("dbo.Books", new[] { "AuthorId" });
        DropTable("dbo.Books");
        DropTable("dbo.Authors");
    }
}
```

Use the Update-Database statement to update the database with the new tables:

```
PM> Update-Database
```

The first command generates code that creates the database, and the second command executes that code. The database is created locally, using LocalDB.



```

Package Manager Console
Package source: nuget.org | Default project: BookService
PM> Enable-Migrations
Checking if the context targets an existing database...
Code First Migrations enabled for project BookService.
PM> Add-Migration Initial
Scaffolding migration 'Initial'.
The Designer Code for this migration file includes a snapshot of your current Code First Migrations model when you scaffold the next migration. If you make additional changes to your model, you can re-scaffold it by running 'Add-Migration Initial' again.
PM> Update-Database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying explicit migrations: [201310282346589_Initial].
Applying explicit migration: 201310282346589_Initial.
Running Seed method.
PM> |

```

Exercise 5, Explore the API

The API enables CRUD operations on the database. The following table summarizes the API.

Authors	
GET api/authors	Get all authors.
GET api/authors/{id}	Get an author by ID.
POST /api/authors	Create a new author.
PUT /api/authors/{id}	Update an existing author.
DELETE /api/authors/{id}	Delete an author.
Books	
GET /api/books	Get all books.
GET /api/books/{id}	Get a book by ID.
POST /api/books	Create a new book.

PUT /api/books/{id}	Update an existing book.
DELETE /api/books/{id}	Delete a book.

Run the application and test these relative urls:

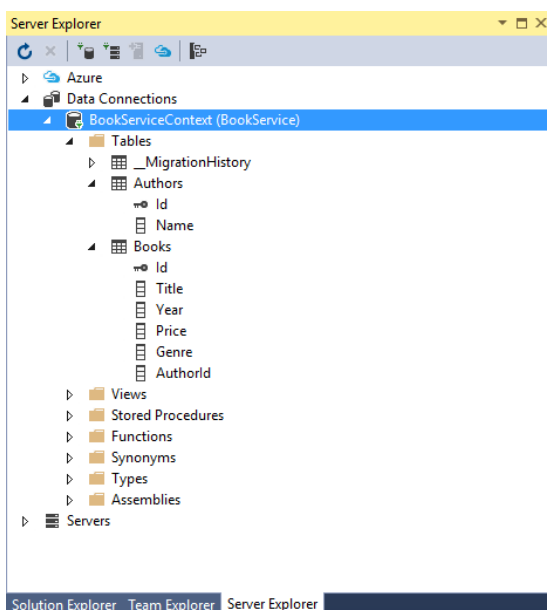
api/authors
api/authors/3

api/books
api/books/4

Exercise 5, View the Database (Optional)

When you ran the Update-Database command, EF created the database and called the **Seed** method. When you run the application locally, EF uses [LocalDB](#). You can view the database in Visual Studio.

In the Server Explorer, you can see the BookServiceContext connection and the newly created tables.



If you right-click on one of the table names you can choose Show table data to see table data that EF populated the database with using the seed method.

dbo.Books [Data]						
Max Rows: 1000						
	Id	Title	Year	Price	Genre	AuthorId
▶	1	Pride and Preju...	1813	9,99	Comedy of ma...	1
	2	Northanger Ab...	1817	12,95	Gothic parody	1
	3	David Copperfi...	1850	15,00	Bildungsroman	2
	4	Don Quixote	1617	8,95	Picaresque	3
*	NULL	NULL	NULL	NULL	NULL	NULL

Exercise 6 intro, Handling Entity Relations

This exercise how to handle circular navigation properties in your model classes. This section provides background knowledge, and is not required to complete the exercises, but has valuable information for better understanding of the next exercise.

Navigation Properties and Circular References

When I defined the Book and Author models, I defined a navigation property on the Book class for the Book-Author relationship, but I did not define a navigation property in the other direction.

What happens if you add the corresponding navigation property to the Author class?

```
public class Author
{
    public int AuthorId { get; set; }
    [Required]
    public string Name { get; set; }

    public ICollection<Book> Books { get; set; }
}
```

Unfortunately, this creates a problem when you serialize the models. If you load the related data, it creates a circular object graph.



When the JSON or XML formatter tries to serialize the graph, it will throw an exception. The two formatters throw different exception messages. Here is an example for the JSON formatter:

```
{
  "Message": "An error has occurred.",
  "ExceptionMessage": "The 'ObjectContent`1' type failed to serialize the response body for content type 'application/json; charset=utf-8'.",
  "ExceptionType": "System.InvalidOperationException",
  "StackTrace": null,
}
```

```
"InnerException": {
  "Message": "An error has occurred.",
  "ExceptionMessage": "Self referencing loop detected with type
'BookService.Models.Book'.
    Path '[0].Author.Books'.",
  "ExceptionType": "Newtonsoft.Json.JsonSerializationException",
  "StackTrace": "...
}"
}
```

Here is the XML formatter:

```
<Error>
  <Message>An error has occurred.</Message>
  <ExceptionMessage>The 'ObjectContent`1' type failed to serialize the response body
for content type
  'application/xml; charset=utf-8'.</ExceptionMessage>
  <ExceptionType>System.InvalidOperationException</ExceptionType>
  <StackTrace />
  <InnerException>
    <Message>An error has occurred.</Message>
    <ExceptionMessage>Object graph for type 'BookService.Models.Author' contains cycles
and cannot be
      serialized if reference tracking is disabled.</ExceptionMessage>
    <ExceptionType>System.Runtime.Serialization.SerializationException</ExceptionType>
    <StackTrace> ... </StackTrace>
  </InnerException>
</Error>
```

One solution is to use DTOs, which I describe in the next section. Alternatively, you can configure the JSON and XML formatters to handle graph cycles. For more information, see [Handling Circular Object References](#).

For this tutorial, you don't need the `Author.Book` navigation property, so you can leave it out.

Exercise 6 intro, Create a Data Transfer Object

Right now, our web API exposes the database entities to the client. The client receives data that maps directly to your database tables. However, that's not always a good idea. Sometimes you want to change the shape of the data that you send to client. For example, you might want to:

- Hide particular properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects, to make them more convenient for clients.
- Avoid “over-posting” vulnerabilities. (See [Model Validation](#) for a discussion of over-posting.)
- Decouple your service layer from your database layer.
- Remove circular references (see previous section).

To accomplish this, you can define a *data transfer object* (DTO). A DTO is an object that defines how the data will be sent over the network. Let's see how that works with the `Book` entity. In the `Models` folder, add two DTO classes:

```
namespace BookService.Models
{
```

```
public class BookDTO
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string AuthorName { get; set; }
}

namespace BookService.Models
{
    public class BookDetailDTO
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public int Year { get; set; }
        public decimal Price { get; set; }
        public string AuthorName { get; set; }
        public string Genre { get; set; }
    }
}
```

The BookDetailDTO class includes all of the properties from the Book model, except that AuthorName is a string that will hold the author name. The BookDTO class contains a subset of properties from BookDetailDTO.

Next, replace the two GET methods in the BooksController class, with versions that return DTOs. We'll use the LINQ **Select** statement to convert from Book entities into DTOs.

```
// GET api/Books
public IQueryable<BookDTO> GetBooks()
{
    var books = from b in db.Books
                select new BookDTO()
                {
                    Id = b.Id,
                    Title = b.Title,
                    AuthorName = b.Author.Name
                };

    return books;
}

// GET api/Books/5
[ResponseType(typeof(BookDetailDTO))]
public async Task<IHttpActionResult> GetBook(int id)
{
    var book = await db.Books.Include(b => b.Author).Select(b =>
        new BookDetailDTO()
        {
            Id = b.Id,
            Title = b.Title,
            Year = b.Year,
            Price = b.Price,
            AuthorName = b.Author.Name,
        }
    );
}
```

```
        Genre = b.Genre
    }).SingleOrDefaultAsync(b => b.Id == id);

    if (book == null)
    {
        return NotFound();
    }
    return Ok(book);
}
```

Here is the SQL generated by the new **GetBooks** method. You can see that EF translates the LINQ **Select** into a SQL SELECT statement.

```
SELECT
  [Extent1].[Id] AS [Id],
  [Extent1].[Title] AS [Title],
  [Extent2].[Name] AS [Name]
FROM   [dbo].[Books] AS [Extent1]
INNER JOIN [dbo].[Authors] AS [Extent2] ON [Extent1].[AuthorId] = [Extent2].[Id]
```

Finally, modify the **PostBook** method to return a DTO.

```
[ResponseType(typeof(BookDetailDTO))]
public async Task<IHttpActionResult> PostBook(Book book)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    db.Books.Add(book);
    await db.SaveChangesAsync();

    // New code:
    // Load author name
    db.Entry(book).Reference(x => x.Author).Load();

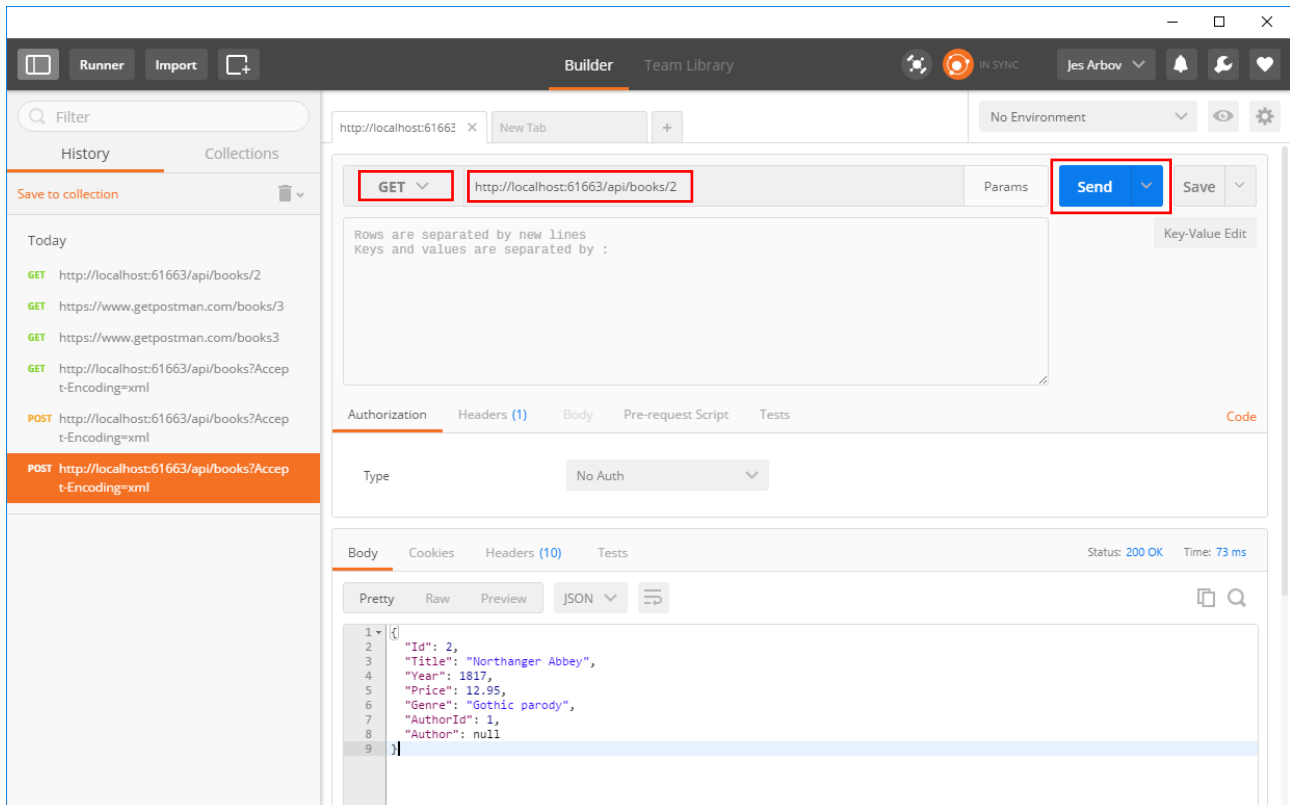
    var dto = new BookDTO()
    {
        Id = book.Id,
        Title = book.Title,
        AuthorName = book.Author.Name
    };

    return CreatedAtRoute("DefaultApi", new { id = book.Id }, dto);
}
```

Exercise 7, Use Postman to test the Book API

We can now test the API by using Postman which is excellent for testing Web APIs (<https://www.getpostman.com/>).

Use the GET method to retrieve a book by inserting the base URL with an endpoint and clicking **Send**:



You'll see the result in the result panel.

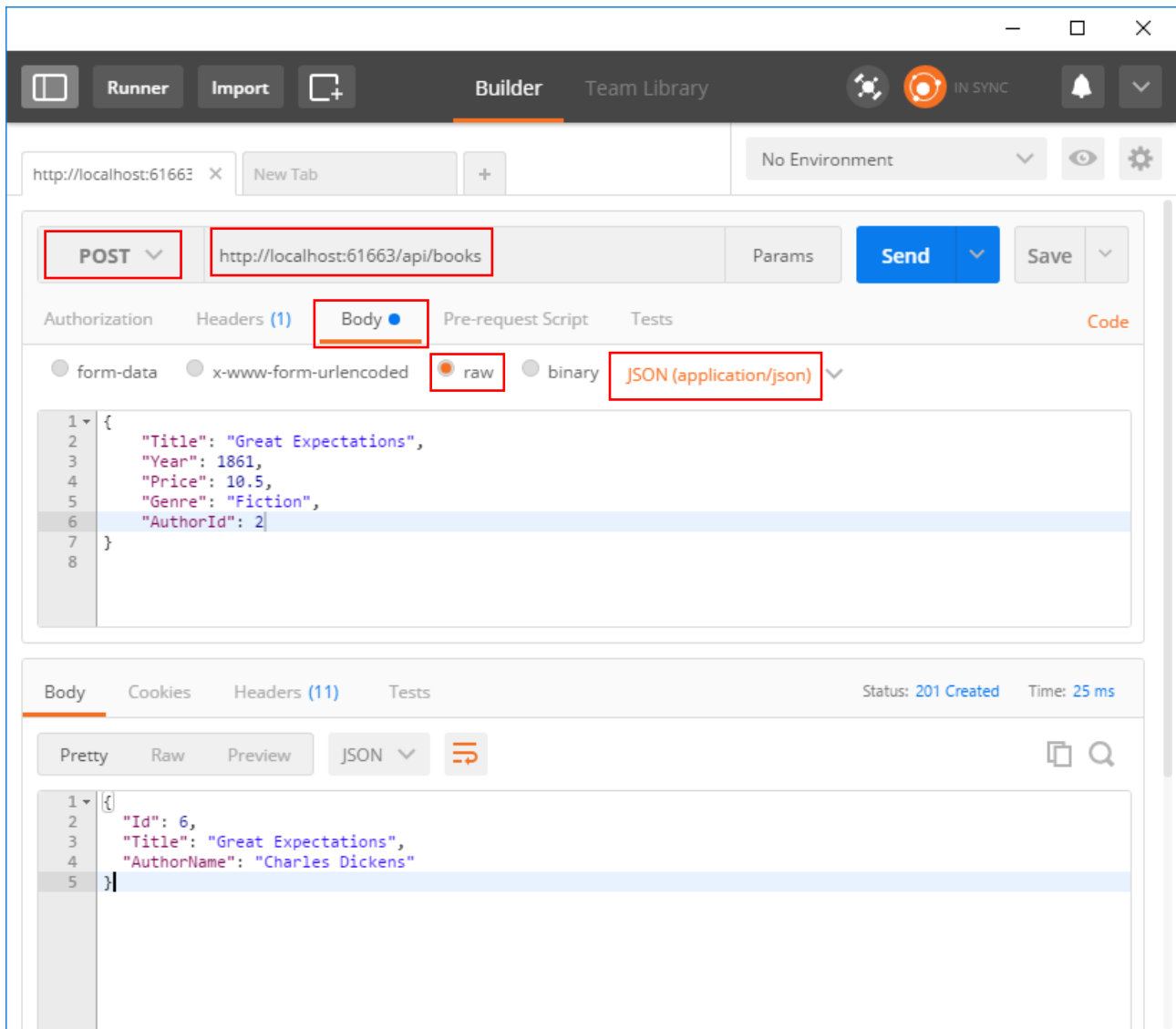
Create a new book

- Set the HTTP method to **POST**
- Tap the **Body** radio button
- Tap the **raw** radio button
- Set the type to **JSON**
- In the key-value editor, enter a Book item such as:

```
{
  "Title": "Great Expectations",
  "Year": 1861,
  "Price": 10.5,
  "Genre": "Fiction",
  "AuthorId": 2
}
```

- Tap **Send**

Tap the Headers tab and copy the **Location** header:



In the response panel, you'll see the returned newly created object from the PostBook method.

```
// POST: api/Books
[ResponseType(typeof(Book))]
public async Task<IHttpActionResult> PostBook(Book book)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

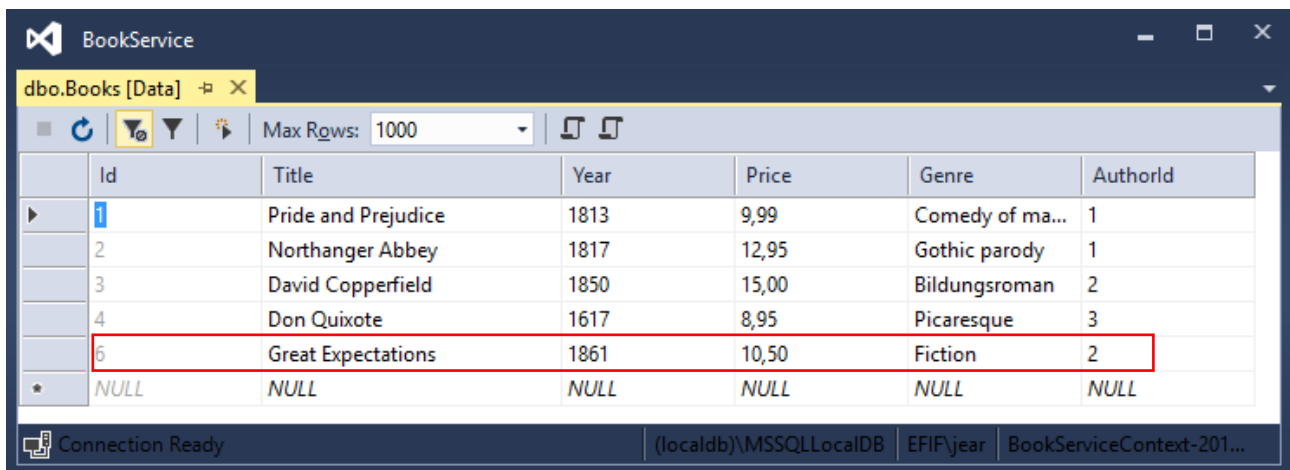
    db.Books.Add(book);
    await db.SaveChangesAsync();

    // New code:
    // Load author name
    db.Entry(book).Reference(x => x.Author).Load();
}
```

```
var dto = new BookDTO()
{
    Id = book.Id,
    Title = book.Title,
    AuthorName = book.Author.Name
};

return CreatedAtRoute("DefaultApi", new { id = book.Id }, dto);
}
```

Go to the database and see the book as a new row in the Book table.

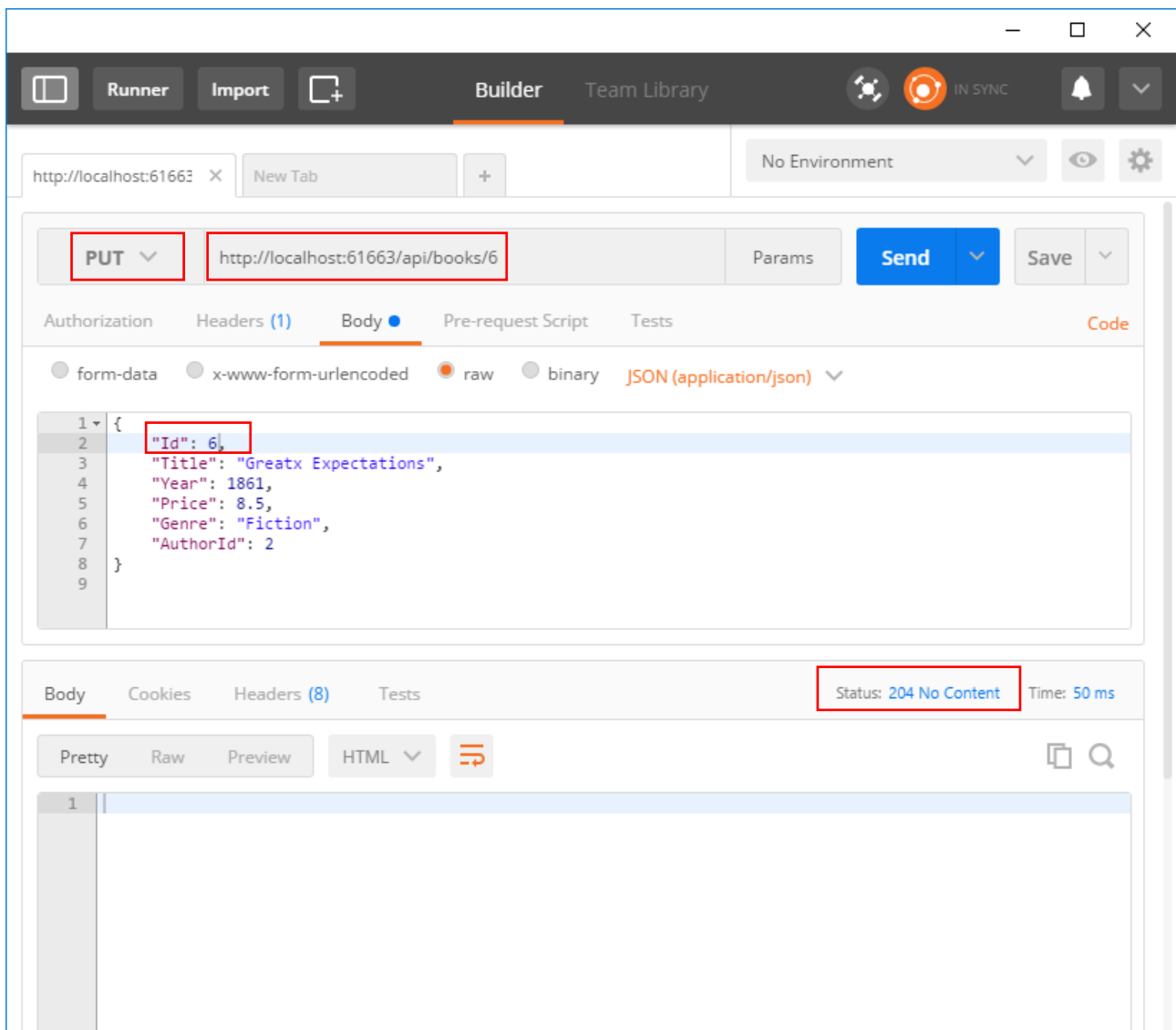


	Id	Title	Year	Price	Genre	AuthorId
▶	1	Pride and Prejudice	1813	9,99	Comedy of ma...	1
	2	Northanger Abbey	1817	12,95	Gothic parody	1
	3	David Copperfield	1850	15,00	Bildungsroman	2
	4	Don Quixote	1617	8,95	Picaresque	3
	6	Great Expectations	1861	10,50	Fiction	2
★	NULL	NULL	NULL	NULL	NULL	NULL

Connection Ready (localdb)\MSSQLLocalDB EFIF\year BookServiceContext-201...

Update an existing book

Update is similar to Create, but uses HTTP PUT. The response is **204 (No Content)**. According to the HTTP spec, a PUT request requires the client to send the entire updated entity, not just the deltas. To support partial updates, use HTTP PATCH.



The book is updated and no content is returned.

The task is handled by the PutBook method:

```
// PUT: api/Books/5
[ResponseType(typeof(void))]
public async Task<IHttpActionResult> PutBook(int id, Book book)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    if (id != book.Id)
    {
        return BadRequest();
    }

    db.Entry(book).State = EntityState.Modified;
```

```

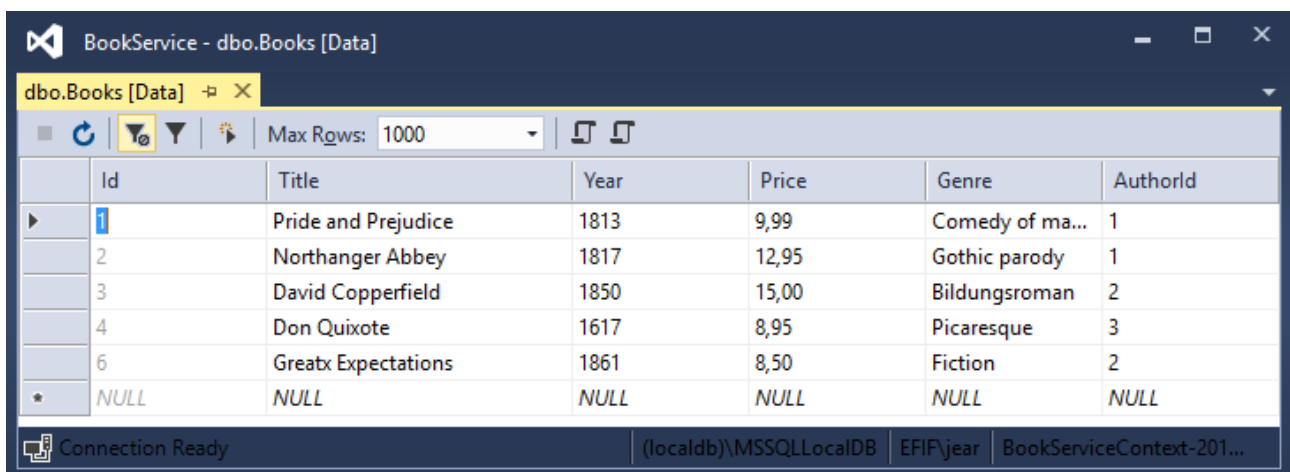
try
{
    await db.SaveChangesAsync();
}
catch (DbUpdateConcurrencyException)
{
    if (!BookExists(id))
    {
        return NotFound();
    }
    else
    {
        throw;
    }
}

return StatusCode(HttpStatusCode.NoContent);
}

```

As you see there is a check to see if the Book object's id match the id segment from the URL.

The price is updated in the Book table:



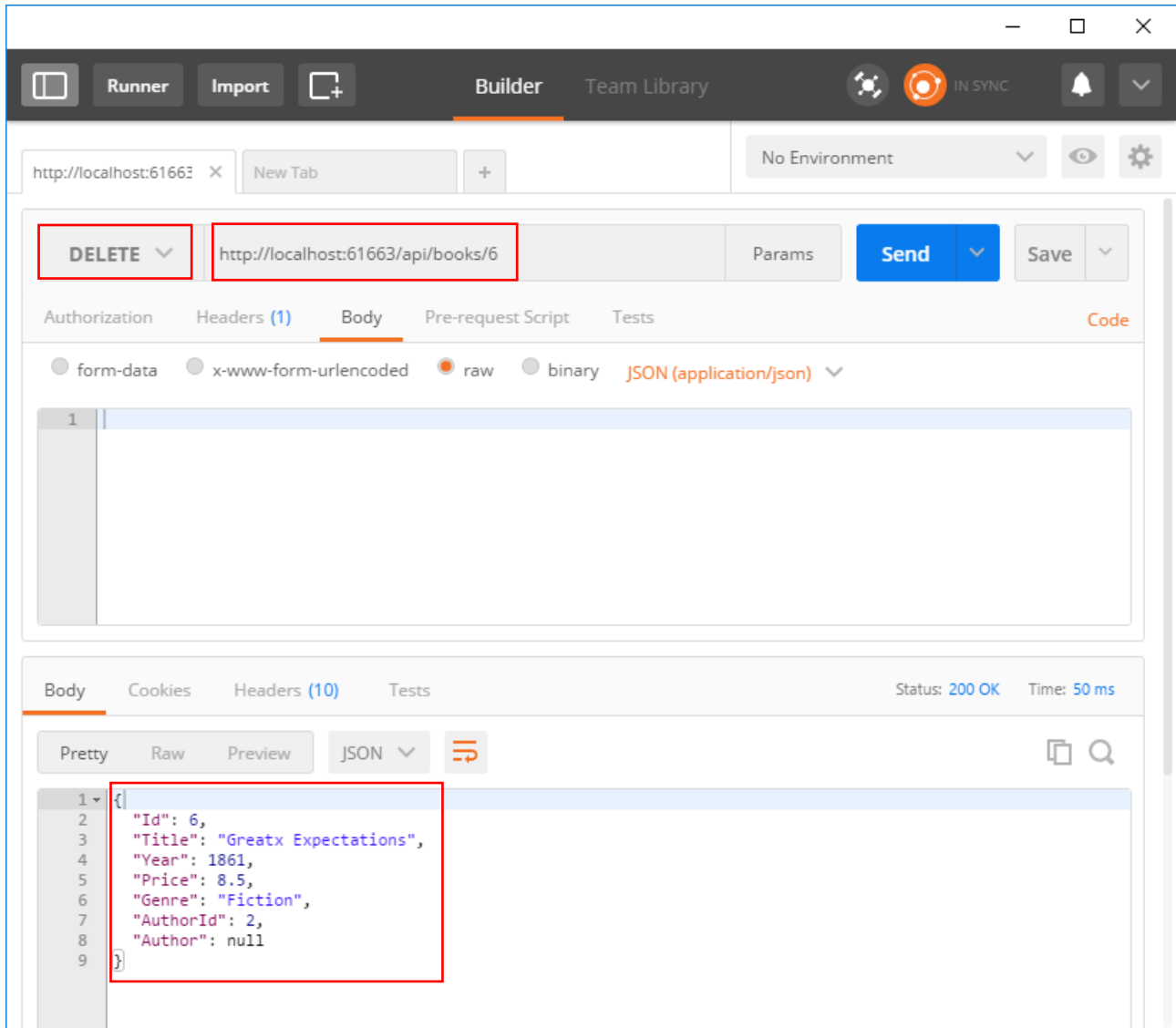
The screenshot shows a window titled 'BookService - dbo.Books [Data]'. Below the title bar is a toolbar with icons for refreshing, filtering, and zooming, along with a 'Max Rgws: 1000' dropdown. The main area displays a table with the following data:

	Id	Title	Year	Price	Genre	AuthorId
▶	1	Pride and Prejudice	1813	9,99	Comedy of ma...	1
	2	Northanger Abbey	1817	12,95	Gothic parody	1
	3	David Copperfield	1850	15,00	Bildungsroman	2
	4	Don Quixote	1617	8,95	Picaresque	3
	6	Greatx Expectations	1861	8,50	Fiction	2
★	NULL	NULL	NULL	NULL	NULL	NULL

At the bottom of the window, a status bar shows 'Connection Ready', '(localdb)\MSSQLLocalDB', 'EFIF\jean', and 'BookServiceContext-201...'.

Delete a book

You can delete a book by using HTTP PUT:



The deleted object is returned, as you also can see from the DeleteBook method:

```
// DELETE: api/Books/5
[ResponseType(typeof(Book))]
public async Task<IHttpActionResult> DeleteBook(int id)
{
    Book book = await db.Books.FindAsync(id);
    if (book == null)
    {
        return NotFound();
    }

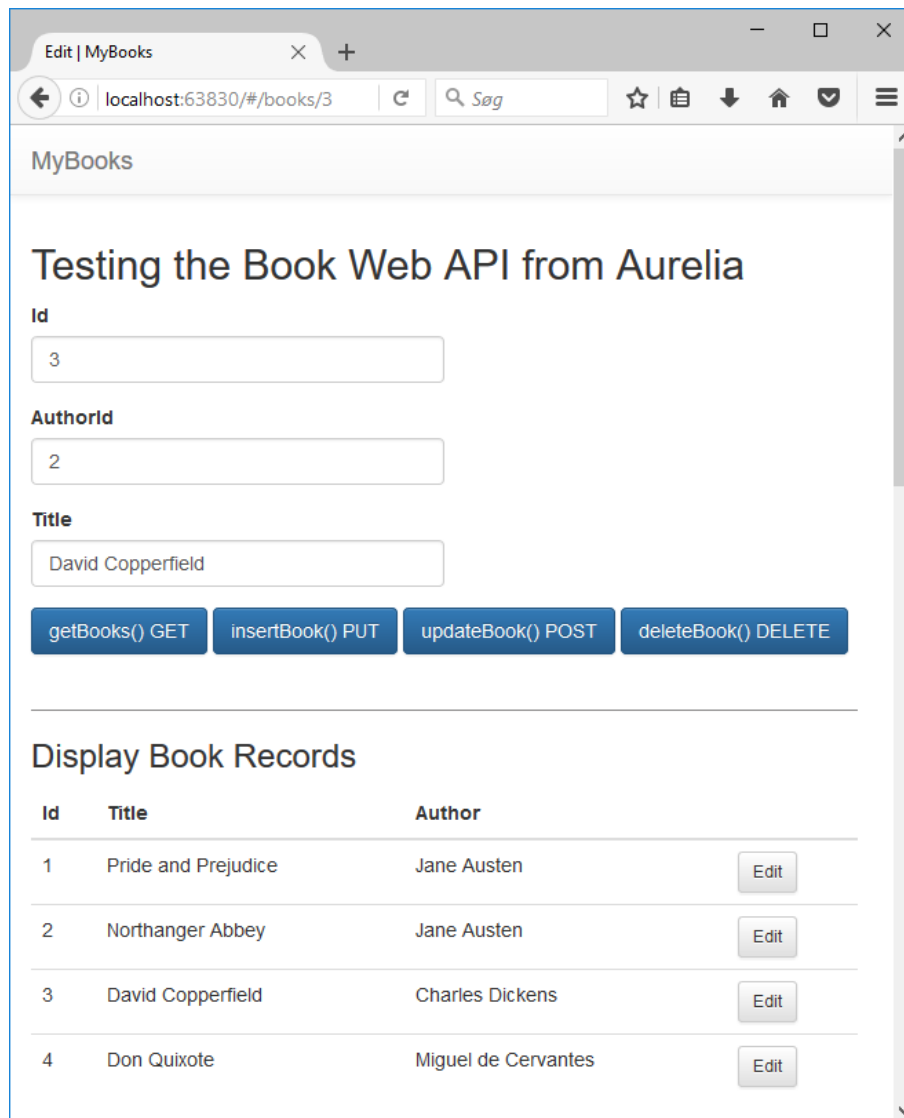
    db.Books.Remove(book);
    await db.SaveChangesAsync();

    return Ok(book);
}
```

}

Exercise 8, Create the JavaScript client (part 6-9), optional

In his tutorial, Mike Wasson uses Knockout MVVM framework for the client-side JavaScript code, but in this tutorial we'll use Aurelia instead. But you're welcome to use any JavaScript MV* framework you like, or you could can program a plain JavaScript/jQuery client, if you prefer that.



In Aurelia you can use the new ECMAScript 6 like modules and classes and even some ECMAScript 7 features as it compiles to ECMAScript 5. Beside that you need a

- Texteditor – we'll use Visual Studio
- A http server and server and some webservises like ASP.NET, php, node or rails. Well us ASP.NET Web API
- Node (nodejs.org) with npm (node package manager) and
- Additional tools:
 - jsmp.io manager
 - Aurelia

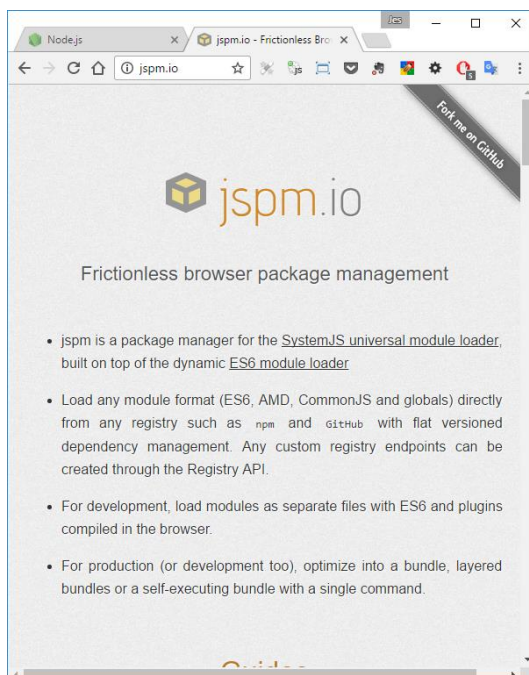
- jspm which is a polyfill for JavaScript 2015
- git command prompt client (jspm relies on that)

Installation of client side software

1. Download and install node (nodejs.org)



2. Install jspm package manager



```
> npm install jspm -g
```

3. Create a new project configuration. Open the BookService Web API *project folder* (with the BookService.csproj file) in file explore and open a command-prompt. Create a new project configuration

```
> jspm init
```

You can use most of the default setting but I recommend that you have the JavaScript config.js file inside a Scripts folder.

```
Package.json file does not exist, create it? [yes]:
Would you like jspm to prefix the jspm package.json properties under jspm?
[yes]:
Enter server baseURL (public folder path) [.]:
Enter jspm packages folder [./jspm_packages]:
Enter config file path [./config.js]: ./Scripts/config.js
Configuration file config.js doesn't exist, create it? [yes]:
Enter client baseURL (public folder URL) [/]:
Which ES6 transpiler would you like to use, Traceur or Babel? [babel]:
```

4. Install Aurelia:

```
> jspm install aurelia-framework
```

```
> jspm install aurelia-bootstrapper
```

5. Install libraries for communicating with web services:

```
> jspm install aurelia-fetch-client
```

```
> jspm install npm:fetch-polyfill
```

6. Install Bootstrap:

```
> jspm install bootstrap
```

7. Open the \Scripts.config.js file and update the babelOptions object:

```
babelOptions: {
  "optional": [
    "runtime",
    "optimisation.modules.system",
    "es7.decorators",
    "es7.classProperties"
  ]
},
```

Update paths to look in the src folder:

```
paths: {
  "*": "src/*",
  ...
},
```


Program the client using Aurelia and Web API

1. First we need an initial start page loading Aurelia Create index.html in the root folder:

index.html

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body aurelia-app>
  <script src="jspm_packages/system.js"></script>
  <script src="Scripts/config.js"></script>
  <script>
    System.import('aurelia-bootstrapper');
  </script>
</body>
</html>
```

The Aurelia application will be loaded into the body-element of the html page.

2. Create Src folder in the root project folder.
3. Create Src/app.js file for configuring the application:

Src/app.js

```
import {Router} from 'aurelia-router';

export class App{
  static inject() { return [Router]; }
  constructor(router) {
    this.router = router;
    this.router.configure(config => {
      config.title = 'MyBooks';
      config.map([
        { route: ['', 'start'], moduleId: 'start', nav: true,
title: 'Aurelia' },
        { route: ['books', 'books'], moduleId: 'books', nav: true,
title: 'Books' },
        { route: 'books/:id', name: 'booksList', moduleId:
'books', nav: false, title: 'Edit' }
      ]);
    });
  }
  message = 'Hello from Aurelia';
}
```

The main content of this file is the routing systems which determines which setup the relationship between URL's and modules.

4. Create a Src/app.html file for loading the initial content of the application:

Src\app.html

```
<template>
  <nav class="navbar navbar-default navbar-fixed-top" role="navigation">
    <div class="navbar-header">
      <a class="navbar-brand" href="#">
        <i class="fa fa-home"></i>
        <span>${router.title}</span>
      </a>
    </div>

    <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
      <ul class="nav navbar-nav">
        <li repeat.for="row of router.navigation"
          class="${row.isActive ? 'active' : ''}">
          <a href.bind="row.href">${row.title}</a>
        </li>
      </ul>
    </div>
  </nav>

  <!-- route will render here -->

  <div class="page-host">
    <router-view></router-view>
  </div>
</template>
```

This page is similar to the layout page in ASP.NET MVC. There is a navigation menu where all the routes with its nav property set to true will be rendered as links, and there is a router-view element in which the modules are displayed.

Src\start.js

```
export class Start{
  constructor() {
    this.heading = 'Welcome to Aurelia';
  }
}
```

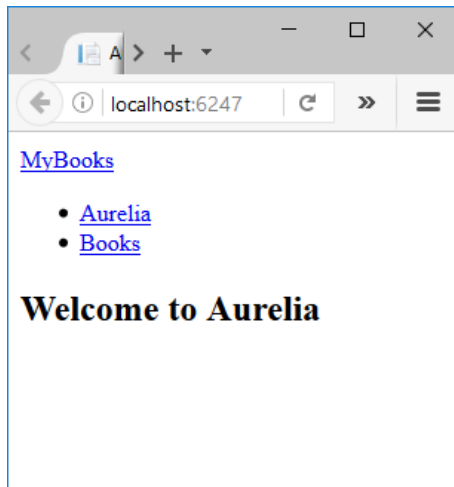
This is a very simple file with a class and a constructor setting a property.

Src\start.html

```
<template>
  <section>
    <h2>${heading}</h2>
  </section>
</template>
```

The template file simply displays the heading property inside a html heading element.

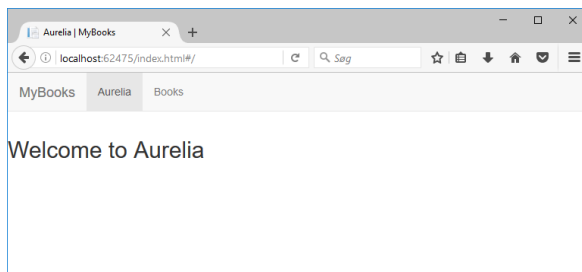
5. Run the index.html file from inside Visual Studio:



Add bootstrap css to the index.html file and create your own style.css file to prevent the bootstrap menu hide the header:

```
style.css
body {
  padding-top: 65px;
}
```

6. Reload the page



7. The next step is to create a book module:

```
Src\book.js
import 'fetch-polyfill';
import {HttpClient} from 'aurelia-fetch-client';

let httpClient = new HttpClient();

export class Books {

  constructor(data) {
    Object.assign(this, data);
    this.heading = 'Testing the Book Web API from Aurelia';
    this.bookList = null;
    this.status = null;
    this.id = 0;
    this.title = "";
    this.authorId = "";
  }
}
```

```
    this.BookId = 0;

    this.getBooks();
  }

  activate(path) {
    if(path && path.id) {
      this.BookId = path.id;
      this.getBook(this.BookId);
    }
  }

  getBooks()
  {
    httpClient.fetch('http://localhost:62475/api/books', {
      method: "GET"
    })
      .then(response => response.json())
      .then(data => {
        this.bookList = data;
        console.log(data);
      });
  }

  getBook(id) {
    var url = "http://localhost:62475/api/books/" + id;
    console.log(this.BookId);
    httpClient.fetch(url, {
      method: "GET"
    })
      .then(response => response.json())
      .then(data => {
        this.id = data.Id;
        this.title = data.Title;
        this.authorId = data.AuthorId;
      })
  }

  insertBook()
  {
    var book = {

      "Title": this.title,
      // uncomment for simplicity
      // "Year": this.id,
      // "Price": this.price,
      // "Genre": this.genre,
      "AuthorId": this.authorId
    };

    httpClient.fetch('http://localhost:62475/api/books', {
      method: "POST",
      headers: {
        'content-type': 'application/json'
      },
      body: JSON.stringify(book)
    })
  }
}
```

```
        .then(response => response.json())
        .then(data => {
            this.status = data;
            this.getBooks();
        });
    }

    updateBook()
    {
        var book = {
            "Id": this.id,
            "Title": this.title,
            // uncomment for simplicity
            // "Year": this.id,
            // "Price": this.price,
            // "Genre": this.genre,
            "AuthorId": this.authorId
        };

        var url = "http://localhost:62475/api/books/" + this.id;

        httpClient.fetch(url, {
            method: "PUT",
            headers: {
                'content-type': 'application/json'
            },
            body: JSON.stringify(book)
        })
        .then(response => response.json())
        .then(data => {
            this.status = data;
            this.getBooks();
        })
    }

    deleteBook() {
        var url = "http://localhost:62475/api/books/" + this.id;

        httpClient.fetch(url, {
            method: "DELETE"
        })
        .then(response => response.json())
        .then(data => {
            this.status = data;
            this.getBooks();
            console.log(data);
        });
    }
}
```

It's a lot of code, and I will not go into all details, but you have references to classes that enable XHR-request (Ajax), there is a constructor instantiating properties, there is `Activate` method assigning the path id to `BookId`, and calls the method `getBook` that retrieves a book, and there are methods for getting all the books (`getBooks`), and there are methods for insert. All CRUD-related methods make calls to the Web API Book service.

Please notice that the localhost port-number is related to my project. Your port-number will prolly be different and you must give the right number to make it work.

Src\books.html

```

<template>
  <div class="row">
    <div class="col-sm-12">
      <section>
        <h2>${heading}</h2>
        <form role="form" submit.delegate="start()">
          <div class="form-group">
            <label for="id">Id</label>
            <input type="text" value.bind="id" class="form-control"
id="id" placeholder="Id">
          </div>
          <div class="form-group">
            <label for="authorId">AuthorId</label>
            <input type="text" value.bind="authorId" class="form-control"
id="authorId" placeholder="Authorid">
          </div>
          <div class="form-group">
            <label for="title">Title</label>
            <input type="text" value.bind="title" class="form-control"
id="title" placeholder="Title">
          </div>

          <button click.delegate="getBooks()" class="btn btn-
primary">getBooks() GET</button>
          <button click.delegate="insertBook()" class="btn btn-
primary">insertBook() PUT</button>
          <button click.delegate="updateBook()" class="btn btn-
primary">updateBook() POST</button>
          <button click.delegate="deleteBook()" class="btn btn-
primary">deleteBook() DELETE</button>
        </form>
      </section>

      <br />
      <hr align="left" />

      <h3>Display Book Records</h3>
      <section>
        <table class="table" id="books">
          <thead>
            <tr>
              <th>Id</th>
              <th>Title</th>
              <th>Author</th>
              <th></th>
            </tr>
          </thead>
          <tbody>
            <tr repeat.for="book of bookList">

              <td>${book.Id}</td>
              <td>${book.Title}</td>
              <td>${book.AuthorName}</td>

              <td>
                <a route-href="route: booksList; params.bind:
{id:book.Id}"
class="btn btn-default btn-sm">Edit</a>

```

```

        </td>
      </tr>
    </tbody>
  </table>
</section>
</div>
</div>
</template>

```

The view template of the book module have a form for updating book info and each input field binds data from the model with the `value.bind` custom attribute that supports two way data-binding. Clicking the buttons activate the CRUD related methods in the model (`Src\books.js`) that calls the Book service on the server.

The table uses the `repeat.for` attribute that iterates over the `bookList` from the model and displays all books.

Each link is a reference to the route named `bookList` and it binds the `book.Id` as

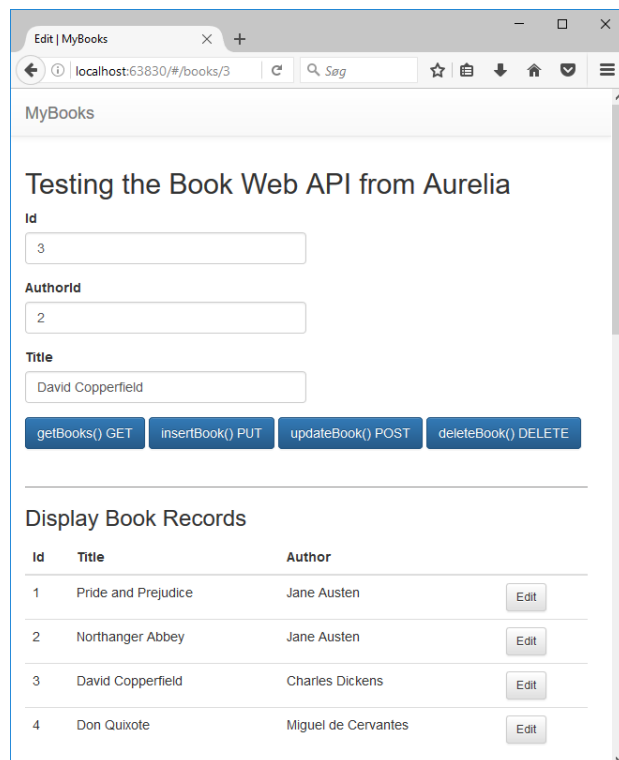
```

<td>
<a route-href="route: booksList; params.bind: {id:book.Id}" class="btn btn-default btn-sm">Edit</a>
</td>

```

Each link is a reference to the route and binds the Book's Id property to a url segment. The `activate` method in the model reads the id value and calls the `getBook` method that retrieves the book information from the server and updates the book model properties used for model binding.

The Aurelia books module gives you a nice interface with the functionality you need to run and test the ASP.NET Web API Book service:



There is still a small problem. Because the foreign key `AuthorId` is not retrieved from the server, the `authorId` input field is not updated in the form when you click the **Edit** button. To fix that, you must add an `authorId` property of type `int` to the `BookDetailDTO` class, and add the

```
AuthorId = b.Author.Id,
```

assignment to the `GetBook` method in the `BooksController` class. After compiling the ASP.NET Web API project, it should work.