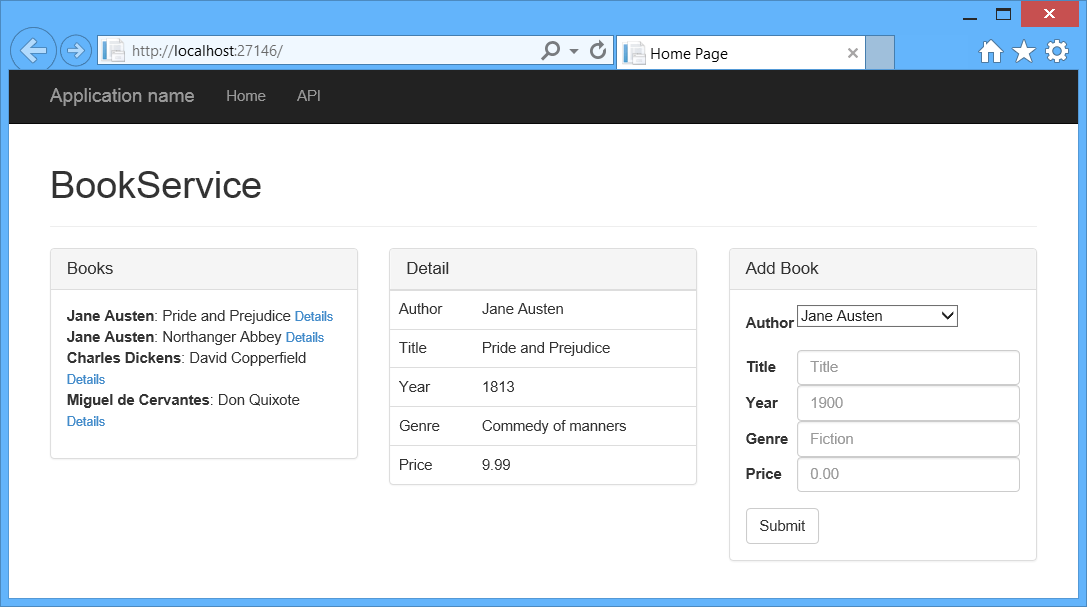
# Web API Lab 2

This tutorial uses 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.

[](https://media-www-asp.azureedge.net/media/4868628/ef22.png)

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 [Knockout.js](http://knockoutjs.com/), 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.
* Knockout.js data-binds the HTML elements to the JSON data.
* Entity Framework talks to the database.

See this App Running on Azure

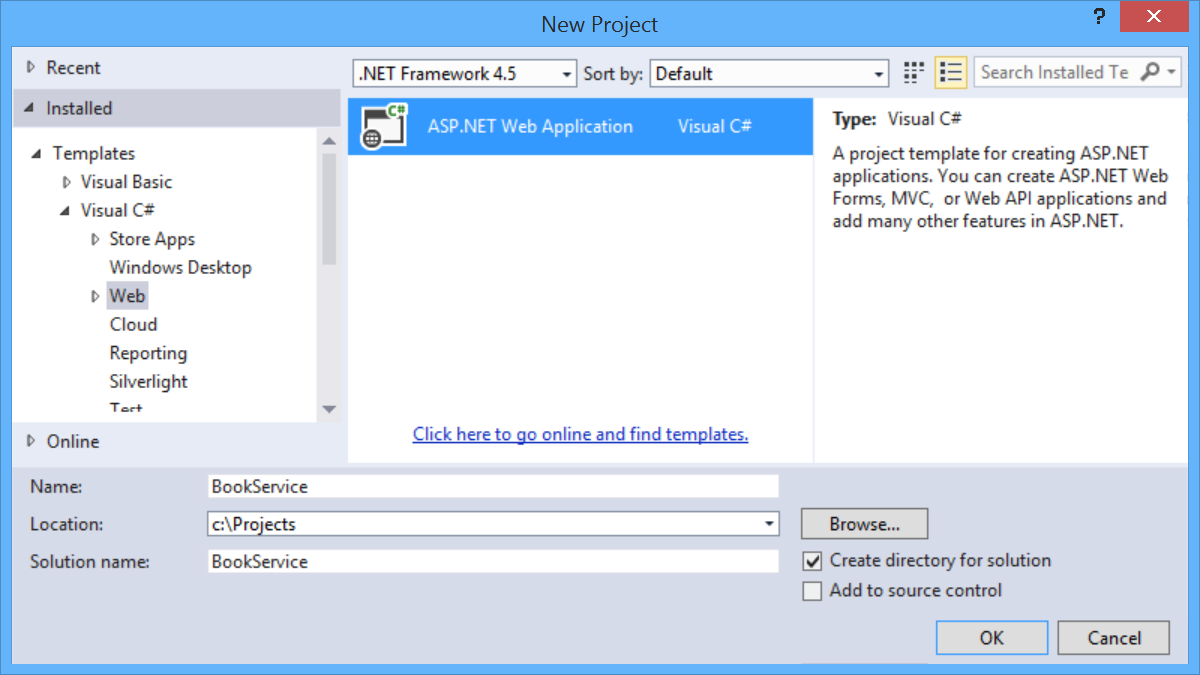
You need an Azure account to deploy this solution to Azure. If you do not already have an account, you have the following options:

* [Open an Azure account for free](http://azure.microsoft.com/en-us/pricing/free-trial/?WT.mc_id=A443DD604) - You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services.
* [Activate MSDN subscriber benefits](http://azure.microsoft.com/en-us/pricing/member-offers/msdn-benefits-details/?WT.mc_id=A443DD604) - Your MSDN subscription gives you credits every month that you can use for paid Azure services.

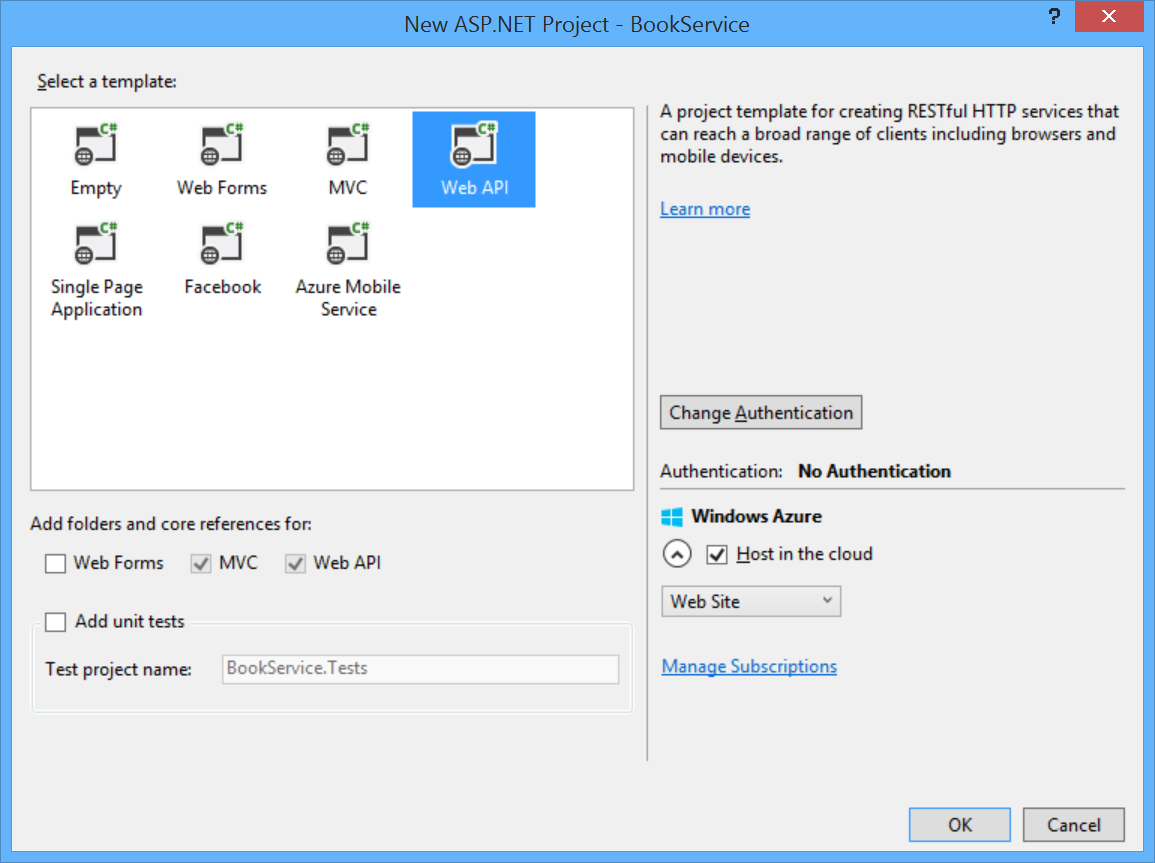
Create the Project

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**.

[](https://media-www-asp.azureedge.net/media/4868502/ef01.png)

In the **New ASP.NET Project** dialog, select the **Web API** template.

[](https://media-www-asp.azureedge.net/media/4868508/ef02.png)

If you want to host the project in a Azure App Service, leave the **Host in the cloud** box checked.

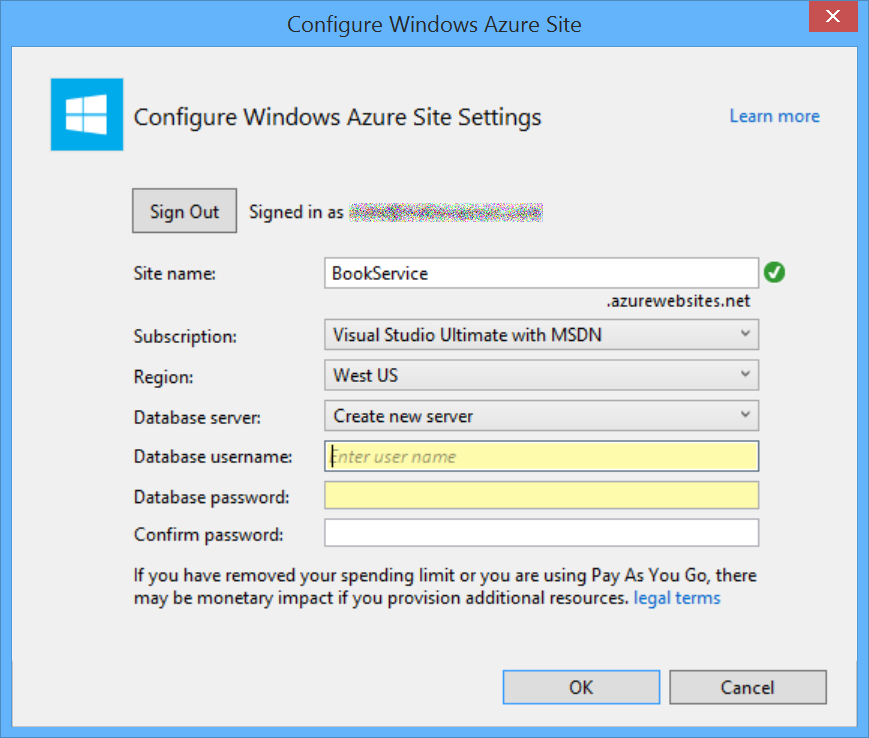
Click **OK** to create the project.

Configure Azure Settings (Optional)

If you left the **Host in Cloud** option checked, Visual Studio will prompt you to sign in to Microsoft Azure

[](https://media-www-asp.azureedge.net/media/4916468/ef02a.png)

After you sign in to Azure, Visual Studio prompts you to configure the web app. Enter a name for the site, select your Azure subscription, and select a geographical region. Under **Database server**, select **Create new server**. Enter an administrator username and password.

[](https://media-www-asp.azureedge.net/media/4916474/ef02b.png)

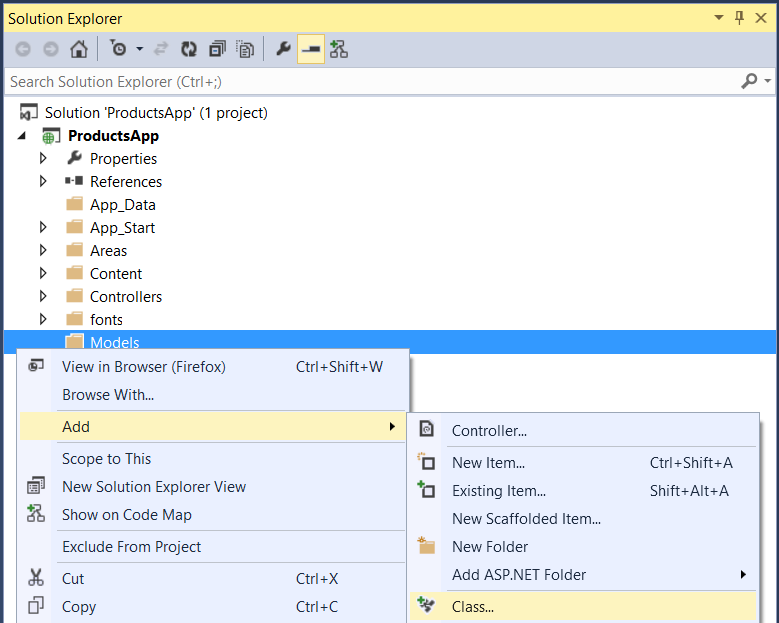
Add Model Classes

In this tutorial, we'll create the database by using the "Code First" approach to Entity Framework (EF). With Code First, you write C# classes that correspond to datbase tables, and EF creates the database.

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; }

}

}

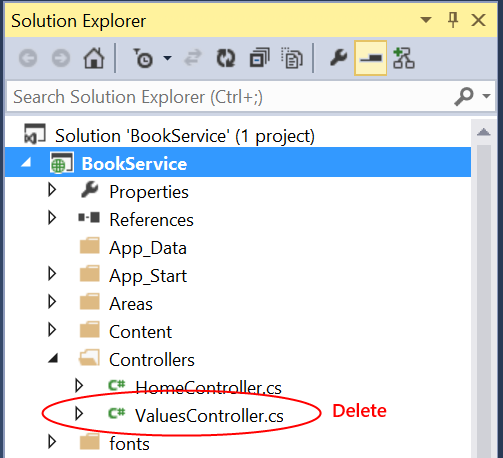
Entity Framework will use these models to create database tables. For each model, the Id property will become the primary key column of the database table.

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.

Add Web API Controllers

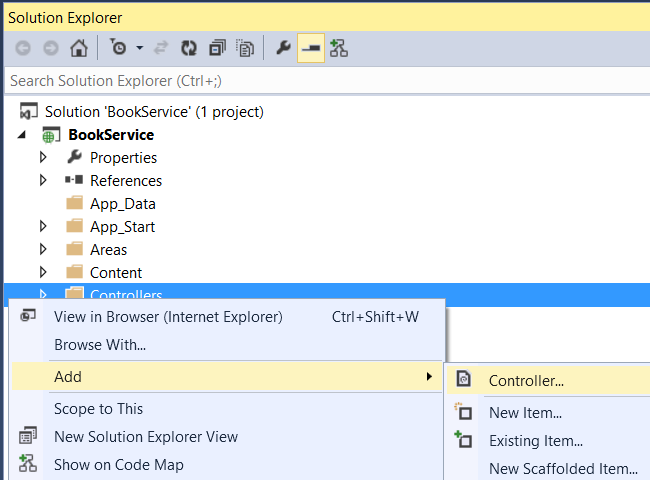
In this section, we’ll add Web API controllers that support CRUD operations (create, read, update, and delete). The controllers will use Entity Framework to communicate with the database layer.

First, you can delete the file Controllers/ValuesController.cs. This file contains an example Web API controller, but you don’t need it for this tutorial.

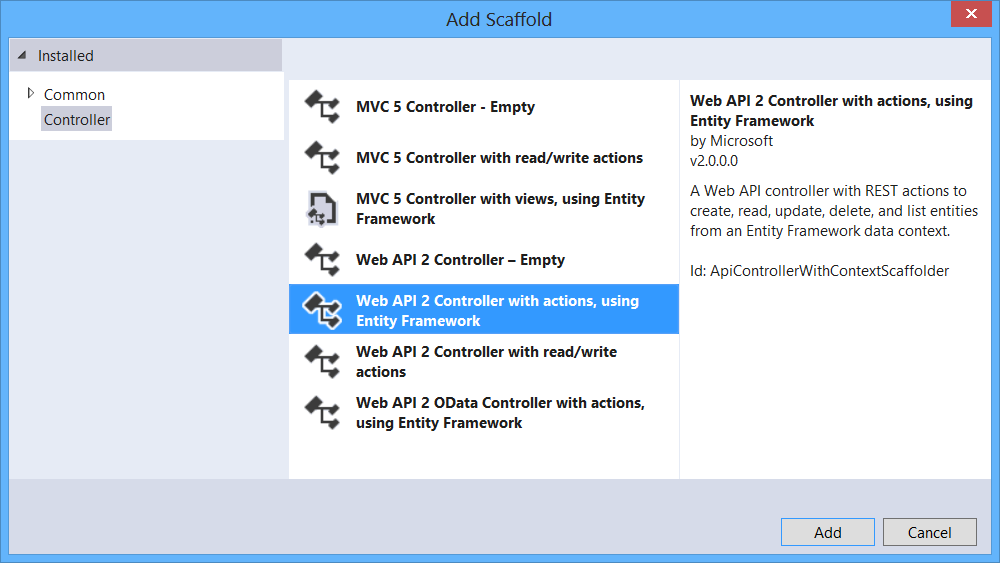


Next, build the project. The Web API scaffolding uses reflection to find the model classes, so it needs the compiled assembly.

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

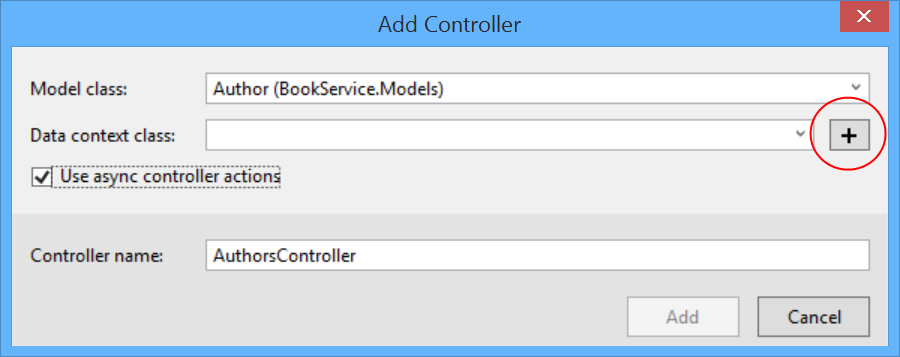


In the **Add Scaffold** dialog, select “Web API 2 Controller with actions, using Entity Framework”. Click **Add**.

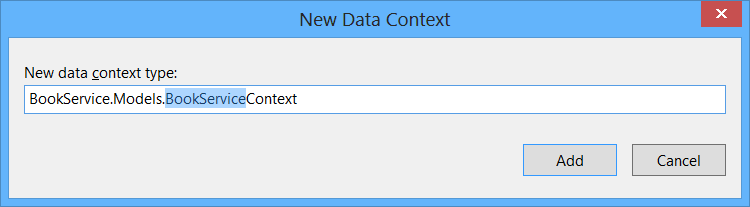


In the **Add Controller** dialog, do the following:

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**.

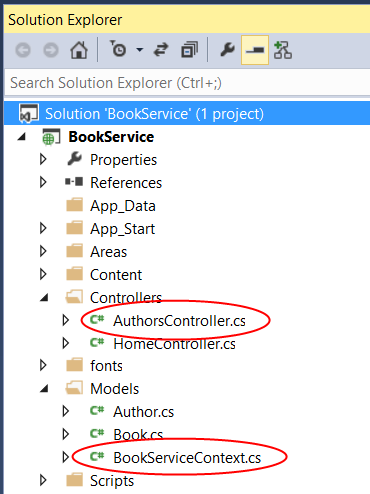


In the **New Data Context** dialog, leave the default name and click **Add**.

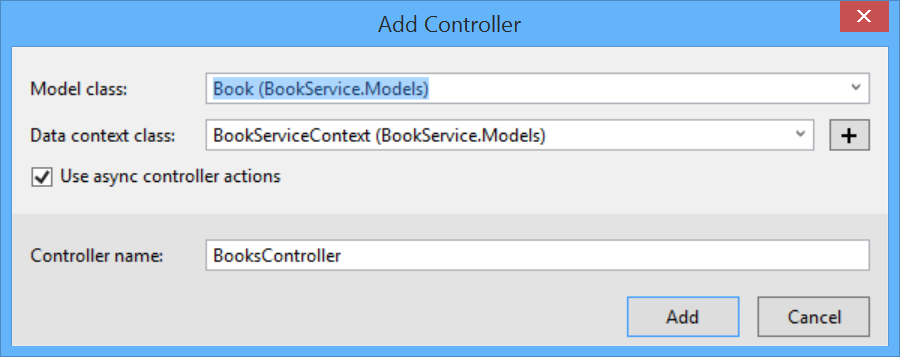


Click **Add** to complete the **Add Controller** dialog. The dialog adds two classes to your project:

* 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.



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.

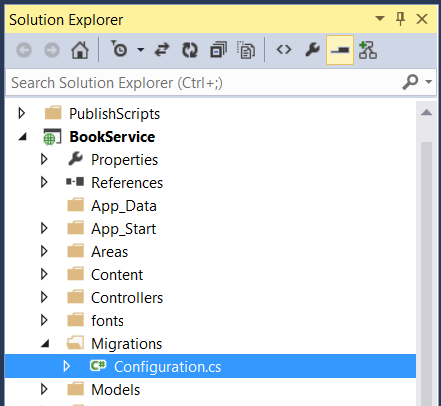


In this section, you will use [Code First Migrations](http://msdn.microsoft.com/en-us/data/jj591621) in EF to seed the database with test data.

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

Enable-Migrations

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" }

);

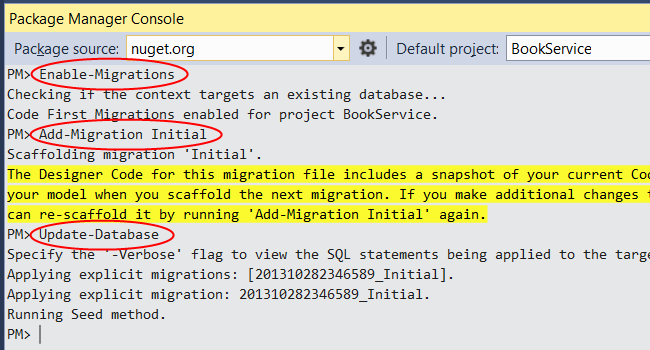
}

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

Add-Migration Initial

Update-Database

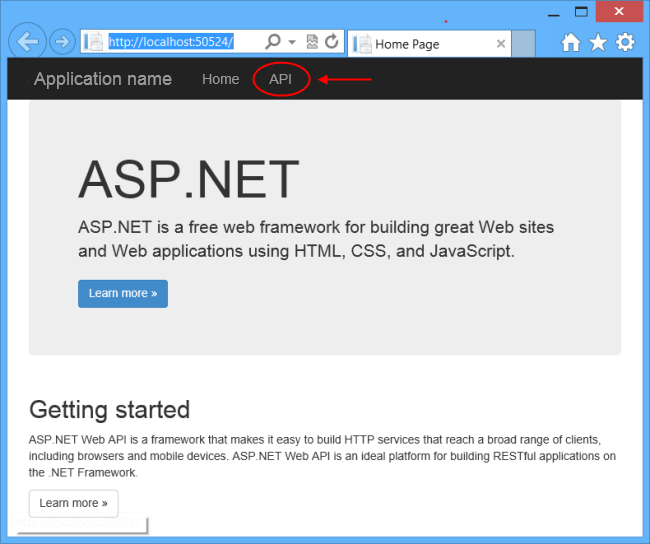
The first command generates code that creates the database, and the second command executes that code. The database is created locally, using [LocalDB](http://msdn.microsoft.com/en-us/library/hh510202.aspx).



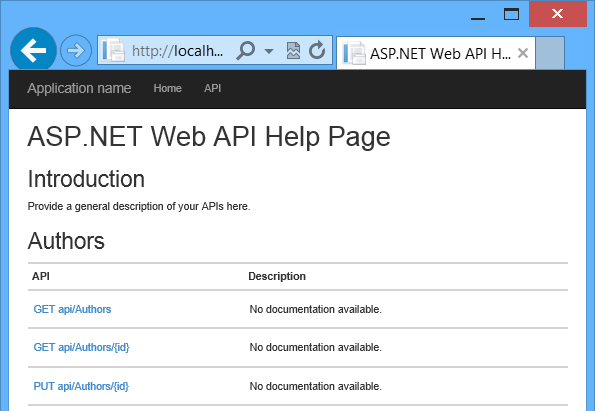
## Explore the API (Optional)

Press F5 to run the application in debug mode. Visual Studio starts IIS Express and runs your web app. Visual Studio then launches a browser and opens the app’s home page.

When Visual Studio runs a web project, it assigns a port number. In the image below, the port number is 50524. When you run the application, you'll see a different port number.



The home page is implemented using ASP.NET MVC. At the top of the page, there is a link that says “API”. This link brings you to an auto-generated help page for the web API. (To learn how this help page is generated, and how you can add your own documentation to the page, see [Creating Help Pages for ASP.NET Web API](https://www.asp.net/web-api/overview/creating-web-apis/creating-api-help-pages).) You can click on the help page links to see details about the API, including the request and response format.

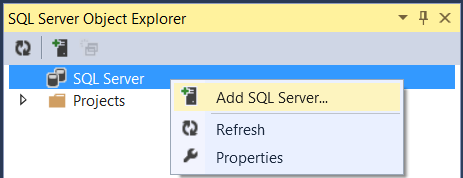


The API enables CRUD operations on the database. The following table summarizes the API.

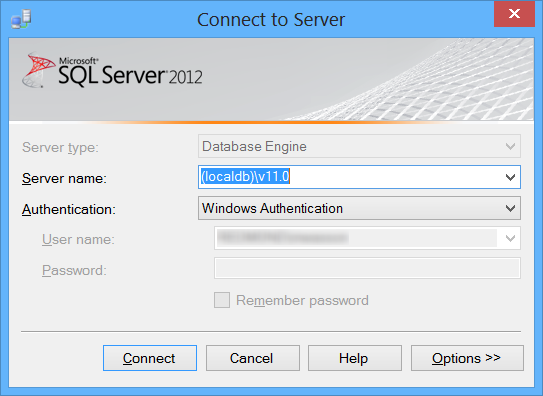
|  |  |
| --- | --- |
| Authors | |
| GET api/authors | Get all authors. |
| GET api/authors/{id} | Get an author by ID. |
| POST /api/authors | Create a new author. |
| PUT /api/authors/{id} | Update an existing author. |
| DELETE /api/authors/{id} | Delete an author. |
| Books | |
| GET /api/books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

## View the Database (Optional)

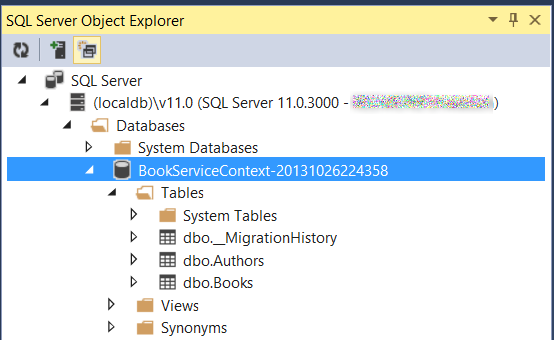
When you ran the Update-Database command, EF created the database and called the Seed method. When you run the application locally, EF uses [LocalDB](http://blogs.msdn.com/b/sqlexpress/archive/2011/07/12/introducing-localdb-a-better-sql-express.aspx). You can view the database in Visual Studio. From the **View** menu, select **SQL Server Object Explorer**.



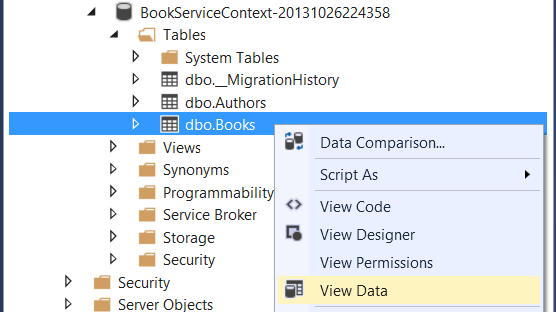
In the **Connect to Server** dialog, in the **Server Name** edit box, type “(localdb)\v11.0”. Leave the **Authentication** option as “Windows Authentication”. Click **Connect**.



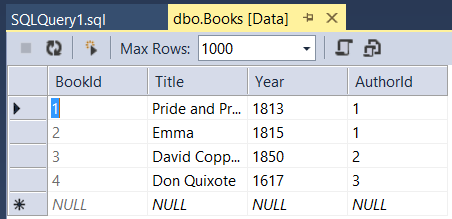
Visual Studio connects to LocalDB and shows your existing databases in the SQL Server Object Explorer window. You can expand the nodes to see the tables that EF created.



To view the data, right-click a table and select **View Data**.



The following screenshot shows the results for the Books table. Notice that EF populated the database with the seed data, and the table contains the foreign key to the Authors table.



This section describes some details of how EF loads related entities, and how to handle circular navigation properties in your model classes.

## Eager Loading versus Lazy Loading

When using EF with a relational database, it's important to understand how EF loads related data.

It's also useful to see the SQL queries that EF generates. To trace the SQL, add the following line of code to the BookServiceContextconstructor:

public BookServiceContext() : base("name=BookServiceContext")

{

// New code:

this.Database.Log = s => System.Diagnostics.Debug.WriteLine(s);

}

If you send a GET request to /api/books, it returns JSON like the following:

[

{

"BookId": 1,

"Title": "Pride and Prejudice",

"Year": 1813,

"Price": 9.99,

"Genre": "Comedy of manners",

"AuthorId": 1,

"Author": null

},

...

You can see that the Author property is null, even though the book contains a valid AuthorId. That's because EF is not loading the related Author entities. The trace log of the SQL query confirms this:

SELECT

[Extent1].[BookId] AS [BookId],

[Extent1].[Title] AS [Title],

[Extent1].[Year] AS [Year],

[Extent1].[Price] AS [Price],

[Extent1].[Genre] AS [Genre],

[Extent1].[AuthorId] AS [AuthorId]

FROM [dbo].[Books] AS [Extent1]

The SELECT statement takes from the Books table, and does not reference the Author table.

For reference, here is the method in the BooksController class that returns the list of books.

public IQueryable<Book> GetBooks()

{

return db.Books;

}

Let's see how we can return the Author as part of the JSON data. There are three ways to load related data in Entity Framework: eager loading, lazy loading, and explicit loading. There are trade-offs with each technique, so it's important to understand how they work.

### **Eager Loading**

With eager loading, EF loads related entities as part of the initial database query. To perform eager loading, use the **System.Data.Entity.Include** extension method.

public IQueryable<Book> GetBooks()

{

return db.Books

// new code:

.Include(b => b.Author);

}

This tells EF to include the Author data in the query. If you make this change and run the app, now the JSON data looks like this:

[

{

"BookId": 1,

"Title": "Pride and Prejudice",

"Year": 1813,

"Price": 9.99,

"Genre": "Comedy of manners",

"AuthorId": 1,

"Author": {

"AuthorId": 1,

"Name": "Jane Austen"

}

},

...

The trace log shows that EF performed a join on the Book and Author tables.

SELECT

[Extent1].[BookId] AS [BookId],

[Extent1].[Title] AS [Title],

[Extent1].[Year] AS [Year],

[Extent1].[Price] AS [Price],

[Extent1].[Genre] AS [Genre],

[Extent1].[AuthorId] AS [AuthorId],

[Extent2].[AuthorId] AS [AuthorId1],

[Extent2].[Name] AS [Name]

FROM [dbo].[Books] AS [Extent1]

INNER JOIN [dbo].[Authors] AS [Extent2] ON [Extent1].[AuthorId] = [Extent2].[AuthorId]

### **Lazy Loading**

With lazy loading, EF automatically loads a related entity when the navigation property for that entity is dereferenced. To enable lazy loading, make the navigation property virtual. For example, in the Book class:

public class Book

{

// (Other properties)

// Virtual navigation property

public virtual Author Author { get; set; }

}

Now consider the following code:

var books = db.Books.ToList(); // Does not load authors

var author = books[0].Author; // Loads the author for books[0]

When lazy loading is enabled, accessing the Author property on books[0] causes EF to query the database for the author.

Lazy loading requires multiple database trips, because EF sends a query each time it retrieves a related entity. Generally, you want lazy loading disabled for objects that you serialize. The serializer has to read all of the properties on the model, which triggers loading the related entities. For example, here are the SQL queries when EF serializes the list of books with lazy loading enabled. You can see that EF makes three separate queries for the three authors.

SELECT

[Extent1].[BookId] AS [BookId],

[Extent1].[Title] AS [Title],

[Extent1].[Year] AS [Year],

[Extent1].[Price] AS [Price],

[Extent1].[Genre] AS [Genre],

[Extent1].[AuthorId] AS [AuthorId]

FROM [dbo].[Books] AS [Extent1]

SELECT

[Extent1].[AuthorId] AS [AuthorId],

[Extent1].[Name] AS [Name]

FROM [dbo].[Authors] AS [Extent1]

WHERE [Extent1].[AuthorId] = @EntityKeyValue1

SELECT

[Extent1].[AuthorId] AS [AuthorId],

[Extent1].[Name] AS [Name]

FROM [dbo].[Authors] AS [Extent1]

WHERE [Extent1].[AuthorId] = @EntityKeyValue1

SELECT

[Extent1].[AuthorId] AS [AuthorId],

[Extent1].[Name] AS [Name]

FROM [dbo].[Authors] AS [Extent1]

WHERE [Extent1].[AuthorId] = @EntityKeyValue1

There are still times when you might want to use lazy loading. Eager loading can cause EF to generate a very complex join. Or you might need related entities for a small subset of the data, and lazy loading would be more efficient.

One way to avoid serialization problems is to serialize data transfer objects (DTOs) instead of entity objects. I’ll show this approach later in the article.

### **Explicit Loading**

Explicit loading is similar to lazy loading, except that you explicitly get the related data in code; it doesn't happen automatically when you access a navigation property. Explicit loading gives you more control over when to load related data, but requires extra code. For more information about explicit loading, see [Loading Related Entities](http://msdn.microsoft.com/en-us/data/jj574232#explicit).

## 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; }

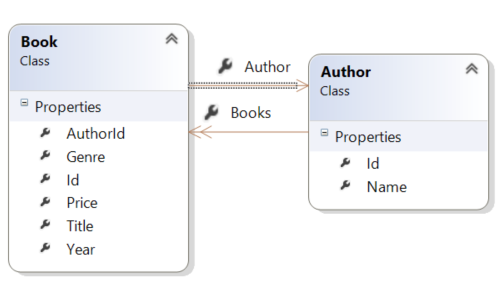
[Required]

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](https://www.asp.net/web-api/overview/formats-and-model-binding/json-and-xml-serialization#handling_circular_object_references).

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

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:

* Remove circular references (see previous section).
* 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](https://www.asp.net/web-api/overview/formats-and-model-binding/model-validation-in-aspnet-web-api) for a discussion of over-posting.)
* Decouple your service layer from your database layer.

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(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);

}

In this tutorial, we're converting to DTOs manually in code. Another option is to use a library like [AutoMapper](http://automapper.org/) that handles the conversion automatically.

Create the Javascript

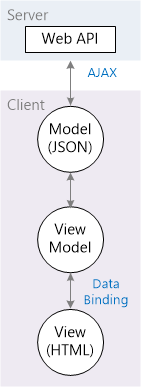
In this section, you will create the client for the application, using HTML, JavaScript, and the [Knockout.js](http://knockoutjs.com/) library. We'll build the client app in stages:

* Showing a list of books.
* Showing a book detail.
* Adding a new book.

The Knockout library uses the Model-View-ViewModel (MVVM) pattern:

* The **model** is the server-side representation of the data in the business domain (in our case, books and authors).
* The **view** is the presentation layer (HTML).
* The **view model** is a JavaScript object that holds the models. The view model is a code abstraction of the UI. It has no knowledge of the HTML representation. Instead, it represents abstract features of the view, such as "a list of books".

The view is data-bound to the view model. Updates to the view model are automatically reflected in the view. The view model also gets events from the view, such as button clicks.



This approach makes it easy to change the layout and UI of your app, because you can change the bindings, without rewriting any code. For example, you might show a list of items as a <ul>, then change it later to a table.

Add the Knockout Library

In Visual Studio, from the **Tools** menu, select **Library Package Manager**. Then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

Install-Package knockoutjs

This command adds the Knockout files to the Scripts folder.

Create the View Model

Add a JavaScript file named app.js to the Scripts folder. (In Solution Explorer, right-click the Scripts folder, select **Add**, then select **JavaScript File**.) Paste in the following code:

var ViewModel = function () {

var self = this;

self.books = ko.observableArray();

self.error = ko.observable();

var booksUri = '/api/books/';

function ajaxHelper(uri, method, data) {

self.error(''); // Clear error message

return $.ajax({

type: method,

url: uri,

dataType: 'json',

contentType: 'application/json',

data: data ? JSON.stringify(data) : null

}).fail(function (jqXHR, textStatus, errorThrown) {

self.error(errorThrown);

});

}

function getAllBooks() {

ajaxHelper(booksUri, 'GET').done(function (data) {

self.books(data);

});

}

// Fetch the initial data.

getAllBooks();

};

ko.applyBindings(new ViewModel());

In Knockout, the observable class enables data-binding. When the contents of an observable change, the observable notifies all of the data-bound controls, so they can update themselves. (The observableArray class is the array version of *observable*.) To start with, our view model has two observables:

* books holds the list of books.
* error contains an error message if an AJAX call fails.

The getAllBooks method makes an AJAX call to get the list of books. Then it pushes the result onto the books array.

The ko.applyBindings method is part of the Knockout library. It takes the view model as a parameter and sets up the data binding.

Add a Script Bundle

Bundling is a feature in ASP.NET 4.5 that makes it easy to combine or bundle multiple files into a single file. Bundling reduces the number of requests to the server, which can improve page load time.

Open the file App\_Start/BundleConfig.cs. Add the following code to the RegisterBundles method.

public static void RegisterBundles(BundleCollection bundles)

{

// ...

// New code:

bundles.Add(new ScriptBundle("~/bundles/app").Include(

"~/Scripts/knockout-{version}.js",

"~/Scripts/app.js"));

}

Create the View

In this section, you will start to define the HTML for the app, and add data binding between the HTML and the view model.

Open the file Views/Home/Index.cshtml. Replace the entire contents of that file with the following.

@section scripts {

@Scripts.Render("~/bundles/app")

}

<div class="page-header">

<h1>BookService</h1>

</div>

<div class="row">

<div class="col-md-4">

<div class="panel panel-default">

<div class="panel-heading">

<h2 class="panel-title">Books</h2>

</div>

<div class="panel-body">

<ul class="list-unstyled" data-bind="foreach: books">

<li>

<strong><span data-bind="text: AuthorName"></span></strong>: <span data-bind="text: Title"></span>

<small><a href="#">Details</a></small>

</li>

</ul>

</div>

</div>

<div class="alert alert-danger" data-bind="visible: error"><p data-bind="text: error"></p></div>

</div>

<div class="col-md-4">

<!-- TODO: Book details -->

</div>

<div class="col-md-4">

<!-- TODO: Add new book -->

</div>

</div>

Most of the div elements are there for [Bootstrap](http://getbootstrap.com/) styling. The important elements are the ones with data-bind attributes. This attribute links the HTML to the view model.

For example:

<p data-bind="text: error">

In this example, the "text" binding causes the <p> element to show the value of the error property from the view model. Recall that error was declared as a ko.observable:

self.error = ko.observable();

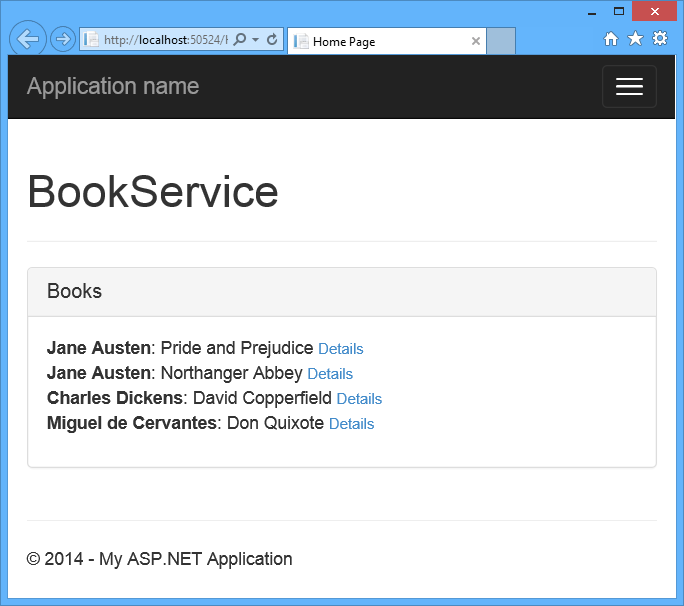
Whenever a new value is assigned to error, Knockout updates the text in the <p> element.

The foreach binding tells Knockout to loop through the contents of the books array. For each item in the array, Knockout creates a new <li> element. Bindings inside the context of the foreach refer to properties on the array item. For example:

<span data-bind="text: Author"></span>

Here the text binding reads the Author property of each book.

If you run the application now, it should look like this:



The list of books loads asynchronously, after the page loads. Right now, the "Details" links are not functional. We'll add this functionality in the next section.

Display Items Details

In this section, you will add the ability to view details for each book. In app.js, add to the following code to the view model:

self.detail = ko.observable();

self.getBookDetail = function (item) {

ajaxHelper(booksUri + item.Id, 'GET').done(function (data) {

self.detail(data);

});

}

In Views/Home/Index.cshtml, add a data-bind element to the Details link:

<ul class="list-unstyled" data-bind="foreach: books">

<li>

<strong><span data-bind="text: AuthorName"></span></strong>: <span data-bind="text: Title"></span>

<!-- New code -->

<small><a href="#" data-bind="click: $parent.getBookDetail">Details</a></small>

</li>

</ul>

This binds the click handler for the <a> element to the getBookDetail function on the view model.

In the same file, replace the following mark-up:

<div class="col-md-4">

<!-- TODO: Book details -->

</div>

with this:

<!-- ko if:detail() -->

<div class="col-md-4">

<div class="panel panel-default">

<div class="panel-heading">

<h2 class="panel-title">Detail</h2>

</div>

<table class="table">

<tr><td>Author</td><td data-bind="text: detail().AuthorName"></td></tr>

<tr><td>Title</td><td data-bind="text: detail().Title"></td></tr>

<tr><td>Year</td><td data-bind="text: detail().Year"></td></tr>

<tr><td>Genre</td><td data-bind="text: detail().Genre"></td></tr>

<tr><td>Price</td><td data-bind="text: detail().Price"></td></tr>

</table>

</div>

</div>

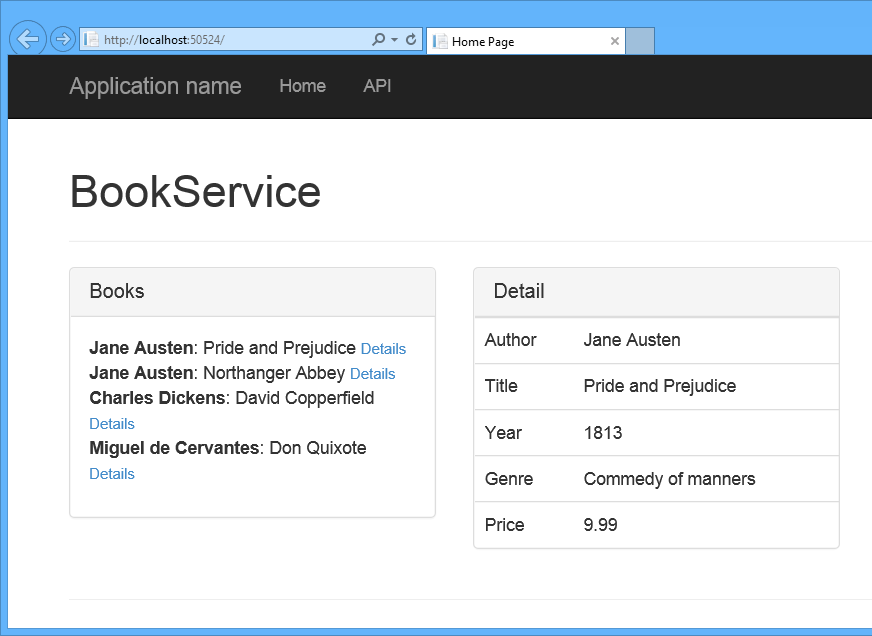
<!-- /ko -->

This markup creates a table that is data-bound to the properties of the detail observable in the view model.

The “<!-- ko -->" syntax lets you include a Knockout binding outside of a DOM element. In this case, the if binding causes this section of markup to be displayed only when details is non-null.

<!-- ko if:detail() -->

Now if you run the app and click one of the "Detail" links, the app will display the book details.

[](https://media-www-asp.azureedge.net/media/4916486/ef26.png)

Add a New Item to the Database

In this section, you will add the ability for users to create a new book. In app.js, add the following code to the view model:

self.authors = ko.observableArray();

self.newBook = {

Author: ko.observable(),

Genre: ko.observable(),

Price: ko.observable(),

Title: ko.observable(),

Year: ko.observable()

}

var authorsUri = '/api/authors/';

function getAuthors() {

ajaxHelper(authorsUri, 'GET').done(function (data) {

self.authors(data);

});

}

self.addBook = function (formElement) {

var book = {

AuthorId: self.newBook.Author().Id,

Genre: self.newBook.Genre(),

Price: self.newBook.Price(),

Title: self.newBook.Title(),

Year: self.newBook.Year()

};

ajaxHelper(booksUri, 'POST', book).done(function (item) {

self.books.push(item);

});

}

getAuthors();

In Index.cshtml, replace the following markup:

<div class="col-md-4">

<!-- TODO: Add new book -->

</div>

With:

<div class="col-md-4">

<div class="panel panel-default">

<div class="panel-heading">

<h2 class="panel-title">Add Book</h2>

</div>

<div class="panel-body">

<form class="form-horizontal" data-bind="submit: addBook">

<div class="form-group">

<label for="inputAuthor" class="col-sm-2 control-label">Author</label>

<div class="col-sm-10">

<select data-bind="options:authors, optionsText: 'Name', value: newBook.Author"></select>

</div>

</div>

<div class="form-group" data-bind="with: newBook">

<label for="inputTitle" class="col-sm-2 control-label">Title</label>

<div class="col-sm-10">

<input type="text" class="form-control" id="inputTitle" data-bind="value:Title"/>

</div>

<label for="inputYear" class="col-sm-2 control-label">Year</label>

<div class="col-sm-10">

<input type="number" class="form-control" id="inputYear" data-bind="value:Year"/>

</div>

<label for="inputGenre" class="col-sm-2 control-label">Genre</label>

<div class="col-sm-10">

<input type="text" class="form-control" id="inputGenre" data-bind="value:Genre"/>

</div>

<label for="inputPrice" class="col-sm-2 control-label">Price</label>

<div class="col-sm-10">

<input type="number" step="any" class="form-control" id="inputPrice" data-bind="value:Price"/>

</div>

</div>

<button type="submit" class="btn btn-default">Submit</button>

</form>

</div>

</div>

</div>

This markup creates a form for submitting a new author. The values for the author drop-down list are data-bound to the authors observable in the view model. For the other form inputs, the values are data-bound to the newBook property of the view model.

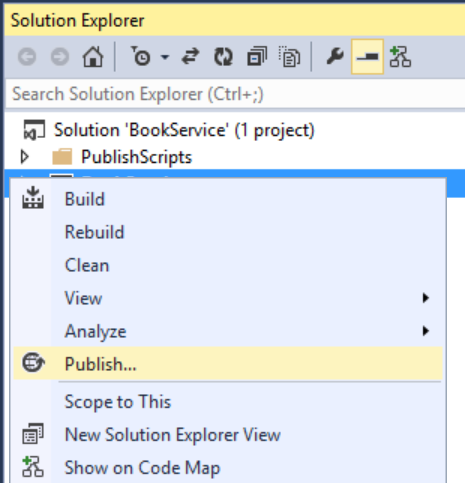
The submit handler on the form is bound to the addBook function:

<form class="form-horizontal" data-bind="submit: addBook">

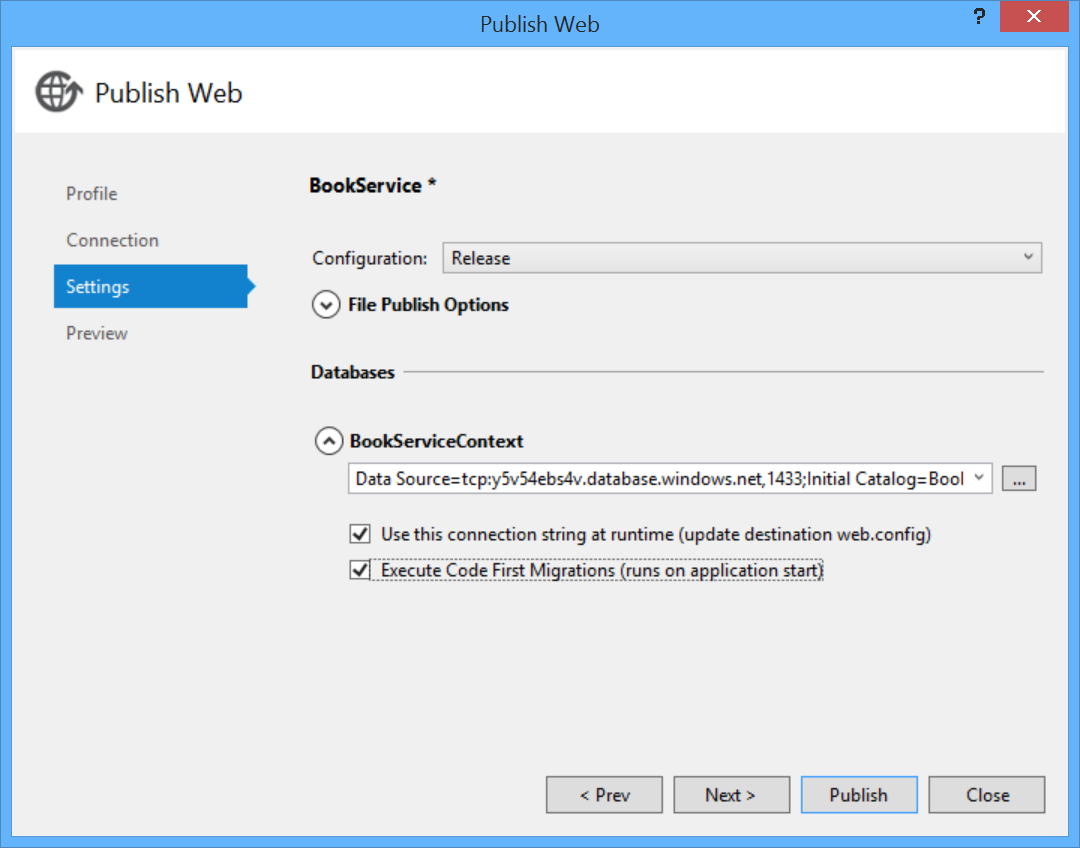
The addBook function reads the current values of the data-bound form inputs to create a JSON object. Then it POSTs the JSON object to /api/books.

Publish the App to Azure Azure App Service

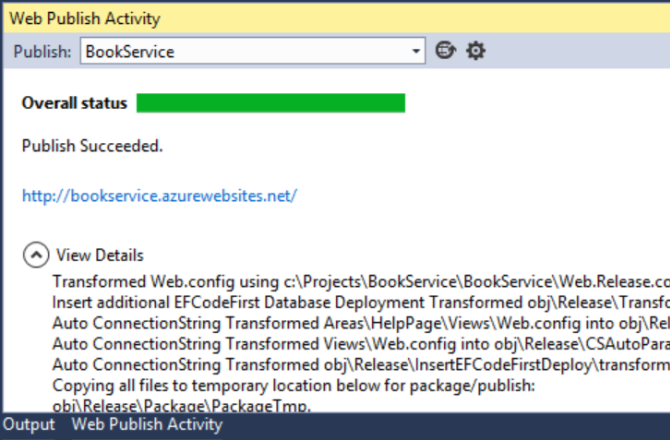
As the last step, you will publish the application to Azure. In Solution Explorer, right-click the project and select **Publish**.



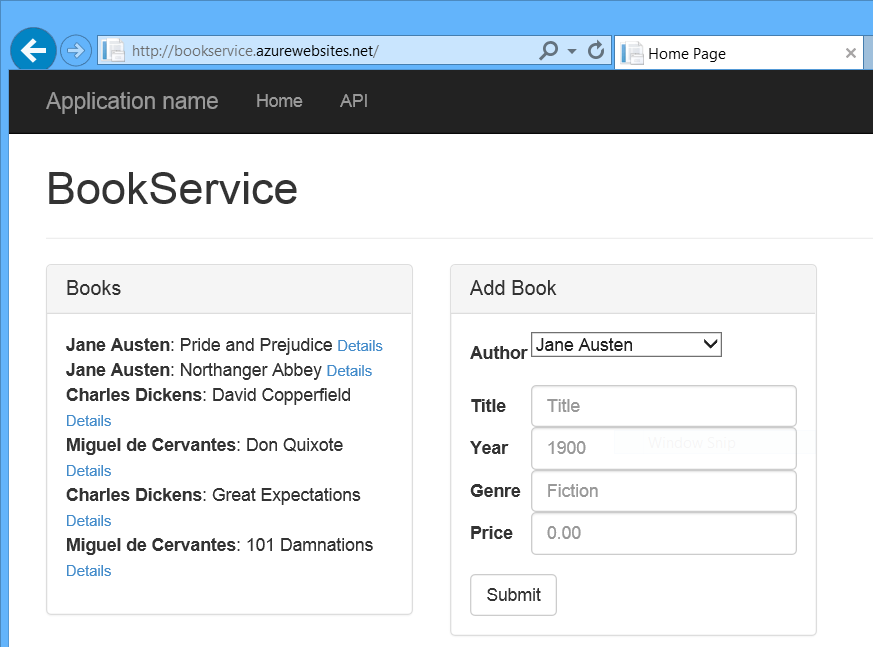
Clicking **Publish** invokes the **Publish Web** dialog. If you checked **Host in Cloud** when you first created the project, then the connection and settings are already configured. In that case, just click the **Settings** tab and check "Execute Code First Migrations". (If you didn't check **Host in Cloud** at the beginning, then follow the steps in the [next section](https://www.asp.net/web-api/overview/data/using-web-api-with-entity-framework/part-10#new-website).)

[](https://media-www-asp.azureedge.net/media/4917211/ef29.png)

To deploy the app, click **Publish**. You can view the publishing progress in the **Web Publish Activity** window. (From the **View** menu, select **Other Windows**, then select **Web Publish Activity**.)

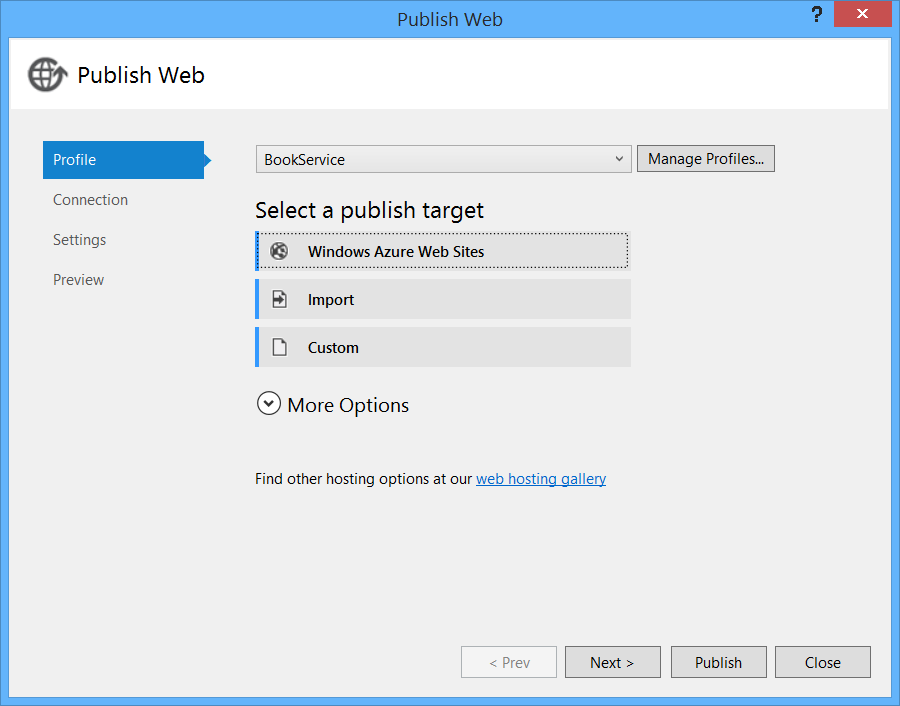


When Visual Studio finishes deploying the app, the default browser automatically opens to the URL of the deployed website, and the application that you created is now running in the cloud. The URL in the browser address bar shows that the site is being loaded from the Internet.

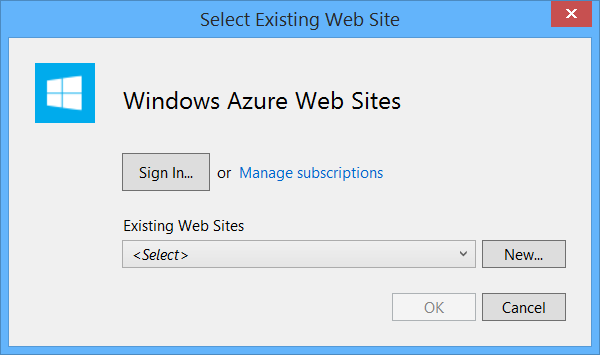
[](https://media-www-asp.azureedge.net/media/4919365/ef32.png)

## Deploying to a New Website

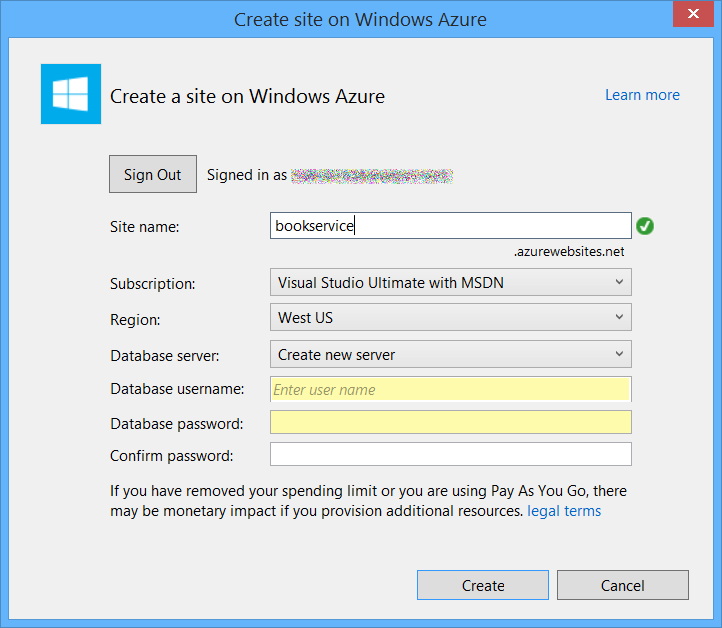
If you did not check **Host in Cloud** when you first created the project, you can configure a new web app now. In Solution Explorer, right-click the project and select **Publish**. Select the **Profile** tab and click **Microsoft Azure Websites**. If you aren't currently signed in to Azure, you will be prompted to sign in.

[](https://media-www-asp.azureedge.net/media/4919402/ef33.png)

In the **Existing Websites** dialog, click **New**.



Enter a site name. Select your Azure subscription and the region. Under **Database server**, select **Create New Server**, or select an existing server. Click **Create**.

[](https://media-www-asp.azureedge.net/media/4919408/ef34.png)

Click the **Settings** tab and check "Execute Code First Migrations". Then click **Publish**.