Building the Environment and Creating the Source Tree

It’s time to start working in Visual Studio! We’ve spent the first three chapters learning about REST and the ASP.NET Web API, as well as designing the task-management service and its underlying classes and database tables. More importantly, we’ve spent some time modeling the resource types and URLs we want to offer for the RESTful service.

In this chapter, we’ll walk through the process of creating the task-management source tree. This will include a specific folder structure (in Windows Explorer) and a Visual Studio 2013 solution and its projects. We will also configure some external libraries using NuGet, as well as create a few project references. Lastly, we will lay down some initial code for the data model classes, service-resource types, logging, and the database.

It is important to set up the source tree properly; or, rather, in a manner that allows for the benefits of separating architectural layers and components into discrete folders and projects. Think of it as the foundation on which we're going to build the “business” functionality. If done right, adding the task-management service operations and making sure they are fully testable will be simple. If done incorrectly, the service code will end up overly coupled and not conducive to clean and effective unit tests.

Speaking of the source code, feel free to download it from either Apress or from the corresponding GitHub repository at https://github.com/jamiekurtz/WebApi2Book.

Let’s start with a few basics that will help ensure your machine is ready for the code.

# Machine Configuration

In this section, you’ll learn about the software prerequisites for building your task-management service. The list is actually quite short, so this won’t take long. You may be able to get everything working by using a different bunch of software or versions. The specifications listed here simply note what has been tested (i.e., what is supported if you are going to utilize the example code that accompanies this book).

## Windows 8 64-bit with .NET Framework 4.51

The code in this book was written on 64-bit Windows 8 with .NET Framework 4.51 installed. Our recommendation would be to follow suit, though Windows 7 64-bit (with .NET 4.51) would probably work as well in case you're one of the many who haven't "upgraded" to Windows 8.

For the web site you’re going to build, you will use IIS Express during development, which is installed with Visual Studio 2013. Don’t worry about needing to use the Professional Edition of Windows 8 (that supports running IIS) unless, of course, you’d rather use IIS over IIS Express.

## SQL Server 2012

As discussed in Chapter 3, the task-management service will include a simple SQL Server database. Thus, you need to have some version of SQL Server installed on your local machine. We used SQL Server 2012 Developer Edition to write this book.

In general, we prefer to install SQL Server as the default instance (i.e., don’t use a named instance). To run the code as-is, you will need to do the same. That said, if you use a named instance (e.g., SQLEXPRESS) you can simply update the connection string(s) before trying to run the example code.

## Visual Studio 2013

Since you’re working with the ASP.NET Web API 2, you will need to install the 2012 or 2013 version of Visual Studio. This code will not work with any of the previous versions. We used 2013 to write the code accompanying this book. And in terms of a specific edition, we used the Ultimate Edition. The Professional and Premium editions will work fine, too.

One of the main reasons for using a non-Express edition of Visual Studio is that JetBrain’s ReSharper is only supported on the “full” editions. And there’s no way either of us would ever write code without ReSharper! For this book, we used ReSharper version 8.2; we highly recommend you do the same.

ReSharper

ReSharper is one of those tools that, once you’ve used it for a bit, you can’t go back to writing .NET code without it. Seriously, time and time again we hear developers refusing to code without ReSharper, even to the point where they will purchase their own personal copies if their employers won’t pony up. It’s that good!

So if you haven’t used it, we strongly encourage you to visit www.jetbrains.com and take a look - and buy it. It will save you tons of time and effort, especially if you write code according to today’s best practices with regard to dependency injection, unit tests, refactoring, variable naming, and so on.

## NuGet Package Manager 2.6

We will use NuGet to set up the various libraries used in your task-management service. This Visual Studio add-in allows a developer to download and add project references for third-party libraries, each with a single command in the NuGet Package Manager console (window). For example, assume you run the following command with your test project selected:

install-package nunit

This code downloads the latest version of NUnit and adds it to your source tree, as well as a reference to all necessary DLLs from within your test project.

NuGet also takes care of library dependencies automatically. For example, if the latest NUnit package required another library, it would be downloaded and referenced, as well.

This book, and the example code, takes advantage of a new feature added back in NuGet version 2.1 that allows you to specify a custom folder location for the downloaded packages. As you’ll see later, we prefer to put our libraries in a lib folder above the folder that holds the solution. By default, however, NuGet places the packages in the same folder as the solution file.

To ensure you have the 2.6 version (or greater) of the NuGet Package Manager, use the Extensions and Updates option under the Tools menu in Visual Studio. If you’re starting from a clean install of Visual Studio 2013, your NuGet Package Manager version should already be at 2.6. The version number will appear on the right-hand side when you click the extension itself. If you already have a greater version, that will work fine, too (e.g., we're using 2.8).

# Creating the Folder Structure

Part of the challenge of creating a source tree is making sure the top-level folder structure is created properly. That is, we want to create a set of folders and paths that allow for easy branching and merging, separation of libraries from source code, documents, and other types of artifacts, and are relatively easy and fast to type on the command line. We also want the folders to be intuitive to any developer who must look at the code.

While no real standard exists for a source code folder structure, the folders we’re going to create in this section are similar to what you can find in many of today’s open source projects. The structure we’ll use in this project is actually quite simple; we just want to have a root folder of some kind, with the following main folders under it: doc, lib, and src. Figure 4-1 shows what this would look like under a folder called WebApi2Book.

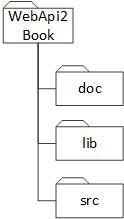


Figure 4-1. The source tree folder structure

What belongs in each of the folders just described should be fairly self-explanatory. But let’s leave no doubt; the following list describes the intended content for each folder:

* doc: Contains documents related to the code base; this might include developer documents, installation guides, tips, requirements, images, and wireframes.
* lib: Contains all third-party libraries and packages used by the application(s) in this source tree; as stated previously, you will configure NuGet to place downloaded packages in this folder.
* src: Contains all of the source code, including the Visual Studio solution file(s) and all project folders.

Even though the task-management service is fairly simple and doesn’t contain much in the way of application code, we think you’ll find that this structure makes it much easier to navigate the tree than if we simply piled everything into a single folder (e.g., the root WebApi2Book folder).

If you’re following along and have already completed the previous section for configuring your machine, then go ahead and create the folder structure from Figure 4-1 in a path similar to this:

C:\MyProjects\WebApi2Book\

At this point, you should now have a machine that contains all the software you need to build your task-management service. You should also have an empty source tree ready for creating an empty Visual Studio 2013 solution file.

# Creating the Solution

You’re now ready to create a blank Visual Studio solution file to which you can later add your projects. You create a blank solution first because you want the solution file to exist in the src folder. Unfortunately, Visual Studio doesn’t let you create a new solution file without also creating a new folder with the same name; it’s kind of a pain!

To put the solution file where you want it, follow these steps in Visual Studio:

* Create a new solution file in the src folder by selecting Project from the File ⮞ New menu.
* Under the Installed ⮞ Other Project Types ⮞ Visual Studio Solutions section, select Blank Solution.
* For this example, enter WebApi2Book for the solution Name.
* For the Location, enter the full path to the src folder you created a bit ago.
* Click OK.

This will create a new folder and solution in your src folder. Now either close Visual Studio or just close the solution. Then, using Windows Explorer, move the new solution file out of the folder that Visual Studio just created and into the src folder. Finally, delete the now-empty folder.

At this point, you should have something like Figure 4-2 in Windows Explorer.

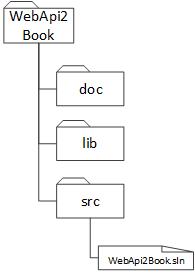


Figure 4-2. Folders with a blank solution file

Don’t re-open the solution file quite yet; you still need to make a small tweak to the NuGet configuration for this solution.

# NuGet Config File

The NuGet Package Manager was introduced in Visual Studio 2010 as a package-management system for .NET. It is similar to the Advanced Package Tool (APT) in many Linux distributions. The basic idea behind the tool is to provide a simple, reliable, and consistent mechanism for downloading libraries and their dependencies from a central repository, and then referencing them from Visual Studio projects. You will be using it to install most of the external libraries you need for your task-management service.

By default, NuGet downloads all packages to a folder called packages. This folder is created in the same folder where the solution file resides. But according to the folder structure shown in Figure 4-1, we want all of our external libraries to exist in the lib folder. As such, you need to provide NuGet with an override for the packages location.

To do this, create a new text file directly in the src folder (with Notepad or at the command line) and name this file nuget.config. Open the file and enter the following XML:

<settings>

<repositoryPath>..\lib</repositoryPath>

</settings>

Save and close the file. Now when you open your new WebApi2Book solution file, NuGet will be configured to place all downloaded libraries into your lib folder.

# Adding the Projects

In this section, we’ll walk through adding all the projects to the new solution, and then configure their dependencies. When building an application, one wouldn’t typically add all of the projects as a first step because it’s usually easier to build them as you go. In this case, though, we want to talk about all of them in one place.

Let’s get started by double-clicking the new solution file (created in the previous section) to open it in Visual Studio 2013. Once open, add the projects as the specified project types listed in Table 4-1.

Table 4-1. The Solution Projects

|  |  |
| --- | --- |
| Project Type | Project Name |
| Class library | WebApi2Book.Common |
| Class library | WebApi2Book.Data |
| Class library | WebApi2Book.Data.SqlServer |
| Class library | WebApi2Book.Web.Api.Models |
| Class library | WebApi2Book.Web.Common |
|  |  |
| ASP.NET Web Application.  Use the Empty project template, and only check the Web API option (Figure 4-3). | WebApi2Book.Web.Api |
| SQL Server Database Project  This can be found under Other Languages ⮞ SQL Server. | WebApi2BookDb |

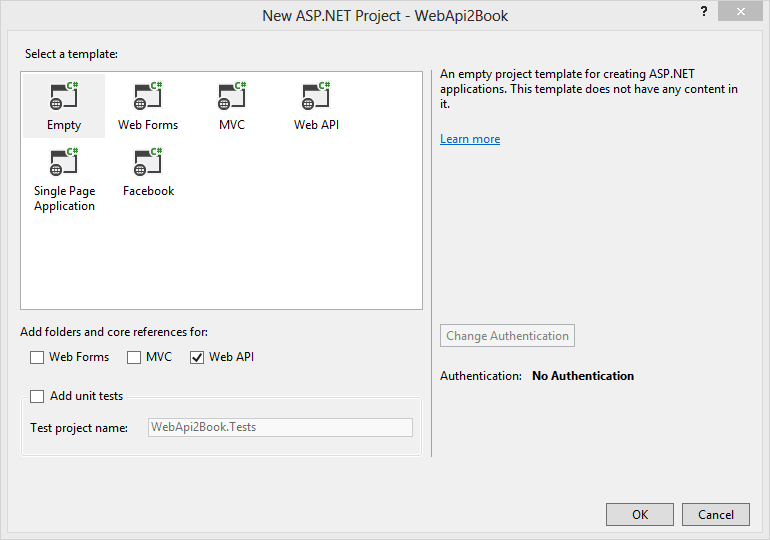


Figure 4- 3 Adding the ASP.NET Web Application

You also want to add a couple test projects to the solution. Begin by creating a new solution folder in Visual Studio called Tests, and then add the projects listed in Table 4-2 to that folder.

Table 4-2. The Solution Test Projects

|  |  |
| --- | --- |
| Project Type | Project Name |
| Class library | WebApi2Book.Common.Tests |
| Class library | WebApi2Book.Data.SqlServer.IntegrationTests |
| Class library | WebApi2Book.Data.Tests |
| Class library | WebApi2Book.Web.Api.IntegrationTests |
| Class library | WebApi2Book.Web.Api.Tests |
| Class library | WebApi2Book.Web.Common.Tests |

Notice that we didn’t add test projects for all projects. This is because not all projects have classes that need to be unit tested. For example, the WebApi2Book.Web.Api.Models project will only contain service model classes, none of which lend themselves to any kind of unit tests.

As mentioned previously, we highly recommend using JetBrains’ ReSharper when developing in .NET. Running unit tests is one of the benefits of this tool. It does a great job within the IDE of letting you run individual tests or all the tests in a class, category, project, or whatever. It also completely abstracts the underlying test framework, so the experience is the same whether you’re using NUnit or MSTest.

At this point, you might be wondering why you have so many projects for such a simple application. There are a plethora of reasons why this separation works well, some of which are beyond the scope of this book. The main goal here is to separate your dependencies; e.g., not require the WebApi2Book.Common project to depend on NHibernate, and not require you to add SQL Server-specific code to anything but the WebApi2.Data.SqlServer project. This approach helps during development, but it also helps keep deployments and updates/patches much cleaner. Table 4-3 illustrates what each project is used for and what it will contain.

Table 4-3. The Project Usage

|  |  |
| --- | --- |
| Project Name | Purpose and Contents |
| WebApi2Book.Common | Contains "framework-ish" functionality not specific to the API or the database. |
| WebApi2Book.Data | Contains the domain model Plain Old CLR Objects (POCOs); these are used by NHibernate to pull/push data from the database. Also contains the data access interfaces and helper classes. However, nothing in this project is specific to SQL Server. |
| WebApi2Book.Data.SqlServer | Contains data access implementations, as well as your NHibernate mappings. This project is what makes the Data project SQL Server–specific at runtime.  As you build up your services application, you should note that no code references any types contained in this project; i.e., the code only references the Data project. |
| WebApi2Book.Web.Api.Models | Contains the service’s REST resource types (or models).  We separate these into their own class library just to make unit testing a little easier. But remember that the client/caller never gets this DLL, because resource type definitions are not shared with REST service clients. |
| WebApi2Book.Web.Common | Contains functionality common to web and service applications. |
| WebApi2Book.Web.Api | This is the REST service application itself; it is hosted by IIS at runtime (though in development we use IISExpress). This project contains all of the Web API controllers and handlers, the REST routes, connection string(s), and so on. |
| WebApi2BookDb | Contains all the schema, code, and data for the SQL Server database. Once this project is compiled, use the output to deploy the database to your preferred target. This works whether you want to create a new database or upgrade an existing one. |
| WebApi2Book.Common.Tests | Unit tests for the classes in the WebApi2Book.Web.Common project. |
| WebApi2Book.Data.SqlServer  .IntegrationTests | Integration tests for the classes in the WebApi2Book.Data.SqlServer; these are used to test data access against the actual database. |
| WebApi2Book.Data.Tests | Unit tests for the classes in the WebApi2Book.Data project. |
| WebApi2Book.Web.Api  .IntegrationTests | Integration ("smoke") tests for the REST service. |
| WebApi2Book.Web.Api.Tests | Unit tests for the controllers and other classes in the WebApi2Book.Web.Api host project. |
| WebApi2Book.Web.Common.Tests | Unit tests for the classes in the WebApi2Book.Web.Common project. |

Now that you have all of your Visual Studio projects in place, you need to add their respective external libraries and references using the NuGet Package Manager Console. These commands will download the latest versions of the libraries (if needed), and then add appropriate references to the given projects. And because in a previous section you configured NuGet to download the packages to your lib folder, you can look there after running these commands to see what was downloaded.

From within the Visual Studio 2013 IDE, open the Package Manager Console window and run the commands listed in Table 4-4. You can find the names of these packages and their corresponding install commands on the NuGet web site at [www.nuget.org](http://www.nuget.org). Each command indicates which package to install and in which project to add the package reference.

Table 4-4. A List of NuGet Commands

|  |
| --- |
| NuGet Command |
| install-package automapper WebApi2Book.Common |
| install-package log4net WebApi2Book.Common |
| install-package nunit WebApi2Book.Common.Tests |
| install-package nunit WebApi2Book.Web.Api.Tests |
| install-package moq WebApi2Book.Web.Api.Tests |
| install-package nhibernate WebApi2Book.Data.SqlServer |
| install-package fluentnhibernate WebApi2Book.Data.SqlServer |
| install-package automapper WebApi2Book.Web.Api |
| install-package log4net WebApi2Book.Web.Api |
| install-package nhibernate WebApi2Book.Web.Api |
| install-package fluentnhibernate WebApi2Book.Web.Api |
| install-package ninject WebApi2Book.Web.Api |
| install-package ninject.web.common WebApi2Book.Web.Api |
| install-package log4net WebApi2Book.Web.Common |
| install-package nhibernate WebApi2Book.Web.Common |
| install-package ninject WebApi2Book.Web.Common |
| install-package ninject.web.common WebApi2Book.Web.Common |

You may need to add more libraries later, but this is a good start and something you can safely do on pretty much any ASP.NET Web API application. We'll be addressing further details regarding unit test project dependencies as we build out that code later in the book.

One of the key architecture considerations of building a web application such as those built with the ASP.NET Web API is managing the instantiation and lifetime of various objects. As stated previously, we are relying on Ninject as our DI container of choice. And to configure the dependency resolution process within ASP.NET Web API to use our Ninject container, we will be leveraging the Ninject.Web.Common extension, which essentially links a Ninject container to HttpContext, enabling object lifetimes to be scoped to a single web request. Further, we will be showing you how to build a custom Ninject-based dependency resolver that is used by ASP.NET. You’ll see how this works in the next chapter.

Finally, let’s add some project references that you already know about. More may be required later, but the ones listed in Table 4-5 are a good start. (We’ve omitted the first part of the projects’ names - WebApi2Book - so that their names will fit in the table).

Table 4-5. Project References

|  |  |
| --- | --- |
| Project | References |
| WebApi2Book.Common.Tests | WebApi2Book.Common |
| WebApi2Book.Web.Api.Tests | WebApi2Book.Common  WebApi2Book.Data  WebApi2Book.Web.Api  WebApi2Book.Web.Api.Models  WebApi2Book.Web.Common |
| WebApi2Book.Data.SqlServer | WebApi2Book.Common  WebApi2Book.Data |
| WebApi2Book.Web.Api | WebApi2Book.Common  WebApi2Book.Data  WebApi2Book.Web.Api  WebApi2Book.Web.Api.Models  WebApi2Book.Web.Common |

If you’ve followed the steps outlined so far, you should see something similar to Figure 4-4 in the Solution Explorer for the WebApi2Book solution. Don't concern yourself with the WebApi2Book.Web.Legacy.Api or WebApi2Book.Windows.Legacy.Client projects at this point; we'll introduce them later as part of a special section on supporting legacy clients.

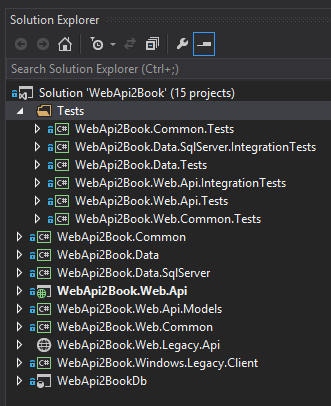


Figure 4-4. The solution in Visual Studio 2013

# Basic Components

At this point, the solution should build successfully, even though we haven’t added any real code yet. But with all the projects added and their libraries installed and referenced, we are ready to start building some of the easier components we’ll need later on:

* Domain model (the persistent entities)
* Service resource types (the service model)
* Logging
* Database

## Domain Model

In this section, we’re going to add the POCO classes that make up your application’s domain model. These will be used primarily to query and update the database.

Since these classes will be used by NHibernate, and we want to support lazy loading, we need to make every property virtual.

Note Lazy loading tells NHibernate to fetch related data only when it is needed – versus fetching all the data up front. For example, when a Task object is fetched from the database, lazy loading means that the Task object’s status and assignments won’t be fetched until code is executed that needs those values.

Other than that, they really are just POCOs. In other words, they don’t derive from some special base class, nor do they return any special types for their properties. They aren’t even tied to NHibernate at all, save for the virtual modifier to allow lazy loading.

Next, we’ll look at all the class definitions. Add them directly to the WebApi2Book.Data project in a folder called Entities. We use the folder name of Entities to more easily distinguish in the code between the persistent model types and the service model types; don't let the name "Entities" fool you into thinking this has anything to do with Entity Framework. The namespace for all of the classes that follow is WebApi2Book.Data.Entities. Feel free to download the code instead of typing all of this in manually:

public class Status

{

public virtual long StatusId { get; set; }

public virtual string Name { get; set; }

public virtual int Ordinal { get; set; }

public virtual byte[] Version { get; set; }

}

public class Task

{

private readonly IList<User> \_users = new List<User>();

public virtual long TaskId { get; set; }

public virtual string Subject { get; set; }

public virtual DateTime? StartDate { get; set; }

public virtual DateTime? DueDate { get; set; }

public virtual DateTime? CompletedDate { get; set; }

public virtual Status Status { get; set; }

public virtual DateTime CreatedDate { get; set; }

public virtual User CreatedBy { get; set; }

public virtual IList<User> Users

{

get { return \_users; }

}

public virtual byte[] Version { get; set; }

}

public class User

{

public virtual long UserId { get; set; }

public virtual string Firstname { get; set; }

public virtual string Lastname { get; set; }

public virtual string Username { get; set; }

public virtual byte[] Version { get; set; }

}

The Version byte array property on all of the domain model classes will be used by NHibernate to detect dirty data. As you’ll see later, the column in SQL Server that the Version property maps to will be of type rowversion. This value is automatically incremented by SQL Server every time a new row is added or updated in the database. In this way, the system can detect when an update to a row will overwrite a previous update.

## Service Model Types

Now let’s add the classes that will make up the service model. Most of these will be pretty similar to the domain model classes you just coded, but you need to remember that your domain model classes are only used internally; that is, they are never sent to the client. That’s what your service model types are used for; they represent the data that will be going back and forth between the client and the service.

All of these class definitions go right in the WebApi2Book.Web.Api.Models project; they use that name as their namespace, as well:

public class Link

{

public string Rel { get; set; }

public string Href { get; set; }

public string Method { get; set; }

}

public class Status

{

public long StatusId { get; set; }

public string Name { get; set; }

public int Ordinal { get; set; }

}

public class Task

{

private List<Link> \_links;

public long? TaskId { get; set; }

public string Subject { get; set; }

public DateTime? StartDate { get; set; }

public DateTime? DueDate { get; set; }

public DateTime? CreatedDate { get; set; }

public DateTime? CompletedDate { get; set; }

public Status Status { get; set; }

public List<User> Assignees { get; set; }

public List<Link> Links

{

get { return \_links ?? (\_links = new List<Link>()); }

set { \_links = value; }

}

public void AddLink(Link link)

{

Links.Add(link);

}

}

public class User

{

private List<Link> \_links;

public long UserId { get; set; }

public string Username { get; set; }

public string Firstname { get; set; }

public string Lastname { get; set; }

public List<Link> Links

{

get { return \_links ?? (\_links = new List<Link>()); }

set { \_links = value; }

}

public void AddLink(Link link)

{

Links.Add(link);

}

}

Recall that one of the tenets of REST is to avoid coupling the client to the server. This means you shouldn’t provide the DLL containing these resource types to callers of your API. These types are there simply to make it easier for the controller code to receive and respond to such data.

## Logging

In this section, you will configure the web.config file. We'll deal with initializing log4net later, when tackling the Ninject container configuration. For now, begin by adding the following code to the Web.Api project’s web.config file, near the top (and directly under the opening <configuration> tag). If the <configSections> section is already there, just add the log4net element:

<configSections>

<section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler, log4net" />

</configSections>

Next, directly under the closing </appSettings> tag, add the following log4net configuration section:

<log4net>

<appender name="LogFileAppender" type="log4net.Appender.RollingFileAppender">

<file type="log4net.Util.PatternString" value="..\\..\\logs\\WebApi2Book.Web.Api.log" />

<appendToFile value="true" />

<maxSizeRollBackups value="-1" />

<countDirection value="1" />

<maximumFileSize value="5MB" />

<rollingStyle value="Composite" />

<preserveLogFileNameExtension value="true" />

<staticLogFileName value="false" />

<lockingModel type="log4net.Appender.FileAppender+MinimalLock" />

<layout type="log4net.Layout.PatternLayout">

<conversionPattern value="%date %-5level [%thread] %logger - %message%newline%exception" />

</layout>

</appender>

<logger name="NHibernate">

<level value="ERROR" />

</logger>

<logger name="NHibernate.SQL">

<level value="ERROR" />

</logger>

<root>

<level value="ALL" />

<appender-ref ref="LogFileAppender" />

</root>

</log4net>

There are about 101 ways to configure logging with log4net. If you want to log a target other than a rolling log file, or if you are interested in modifying the behavior just covered, you should read the log4net configuration documentation to learn more. Here are a couple of useful links:

* http://logging.apache.org/log4net/release/manual/configuration.html
* http://logging.apache.org/log4net/release/sdk/log4net.Layout.PatternLayout.html

As-is, the preceding configuration logs to a file called WebApi2Book.Web.Api.YYYY-MM-DD.count.log (e.g., " WebApi2Book.Web.Api.2014-04-19.3.log") in a logs folder in the WebApi2Book root directory. Each new day the system will create a new log file, and it will roll over to a new file if the current file gets to be 5MB in size. This configuration also logs only errors from NHibernate.

## The Database

You explored the tables included in your database in Chapter 3, when you designed your service API. In this section, you will look at the SQL Server Database Project in Visual Studio 2013. However, it would require too much space to add the scripts for all tables and data used in this book. So, to get the real database content, please download the source code from either Apress or from the corresponding GitHub repository at https://github.com/jamiekurtz/WebApi2Book.

To start, you will have two folders in the project—all created manually in Visual Studio (see Figure 4-5).

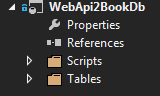


Figure 4-5. The database project

The Scripts folder will contain your deployment scripts for adding lookup data, permissions, and optional test data. For example, you need to have INSERT statements to populate your Status values.

The Tables folder should be self-explanatory.

I want to point out one thing regarding lookup data: the scripts in the project will run every time they are applied to a target database. In other words, you need to be very aware of existing data—and avoid INSERT statements that will cause primary key violations. For this reason, anytime you add or update lookup data, the SQL statements need to first check that the data doesn’t exist already, as in this snippet:

if not exists(select \* from dbo.Status where Name = 'Not Started')

insert into dbo.Status(Name, Ordinal) values('Not Started', 0);

if not exists(select \* from dbo.Status where Name = 'In Progress')

insert into dbo.Status(Name, Ordinal) values('In Progress', 1);

if not exists(select \* from dbo.Status where Name = 'Completed')

insert into dbo.Status(Name, Ordinal) values('Completed', 2);

# Summary

In this chapter, you learned how to configure a clean Windows 8 machine with the software required to build the task-management service. You also created the folder structure you need to start adding code, libraries, and documents to the source tree. Next, you created an empty solution and added to it all of the projects we plan on using, including various library and project references. Finally, you created a bunch of basic classes and the various application configuration settings needed to support the service.

You’ve also added all your solution projects, installed and referenced their libraries, and added your baseline classes. At this point, your solution should build successfully. You are now ready to start creating some of the framework-level components needed to manage controller and database session lifetimes, security, and the Ninject dependency injection container.