Implementing a Post

In the previous chapter, we started with an empty folder, created a basic source-tree structure, added a new Visual Studio 2013 solution, and added the projects we know we'll need for this little REST service. We also added some of the more basic code components, and setup the project and library references we anticipate needing. We wrapped things up by creating the database.

In this chapter we will implement our first controller method (or "action method"). Along the way we will deal with some of the more complex infrastructural concerns in the task-management service and highlight some great ASP.NET Web API features. We'll cover:

* Routing (conventional and attribute-based)
* API versioning using attribute-based routing and a custom controller selector
* Management of dependencies
* NHibernate configuration and mappings
* Database unit of work management
* Database transaction control
* Diagnostic Tracing/Logging
* Error handling
* IHttpActionResult
* Input validation
* Content negotiation

You may be wondering why security is missing from this infrastructure-heavy chapter. Well, security merits its own chapter; so don't worry, we will get to it soon. For now we want to focus on getting the basic infrastructure in place so that we can begin implementing our task-management business logic.

Yes, this is a lot of material to cover in this rather long chapter. We'll take it step-by-step so that it will end up making sense in the end. Now let's get started…

# Routing

Although a request to an ASP.NET Web API-based service can be processed by a message handler without any need for a controller (and we'll see an example of this in a later chapter when we discuss processing SOAP messages for legacy callers), ASP.NET Web API-based services are normally configured to route messages to controllers for processing. Such an arrangement allows services to benefit from model binding, controller- and action-specific filters, and result conversion; i.e., it makes full utilization of the ASP.NET Web API's extensible processing pipeline. This is the kind of arrangement we will be discussing in this chapter.

HTTP Message Lifecycle in ASP.NET Web API

The official Microsoft ASP.NET Web API site has an excellent poster illustrating the complete ASP.NET Web API HTTP message lifecycle. The poster is available at <http://www.asp.net/posters/web-api/ASP.NET-Web-API-Poster.pdf>. A highly-simplified version illustrating some main elements of the processing pipeline is shown in Figure 5-1, below.



Figure 5-1. ASP.NET Web API Message Processing Pipeline

Be sure to visit the official Microsoft site for this poster and many other useful resources.

When a service request comes over the network and into IIS, it routes it to an instance of the ASP.NET worker process. Inside this process (or equivalent host process for self-hosted applications and applications running in IIS Express) the ASP.NET Web API framework uses the routes configured in the application to determine which controller should respond to the request. When the appropriate controller class is found, the ASP.NET Web API framework creates an instance of that controller class and forwards the web request to the appropriate controller action.

Let’s look at some examples. Suppose you have the following route configured in the WebApiConfig.cs file (this is actually the default route set up by Visual Studio when you create a new Web API project):

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new {id = RouteParameter.Optional});

Using the power of URL routing, the framework will try to match the URLs of requests against this and other routes. In this particular route, the framework will use the portion of the URL specified after api/ to determine the appropriate controller to activate. Assume you were to make either of the following calls against the service:

/api/tasks

/api/tasks/123

In both cases, the framework would activate the controller class called TasksController. Note that even though the URL specifies only tasks, the class name to be activated will be TasksController. By convention, the framework will automatically append the word Controller to the name taken from the URL.

At this point, you may be asking the question, “Which controller action method will get invoked?” Well, unlike with ASP.NET MVC, the URL route doesn’t have to include an {action} segment. This is because the ASP.NET Web API framework automatically invokes controller methods based on the HTTP verb the caller is using. For example, if the caller performs a GET on the URL /api/tasks, the framework will invoke the Get() method on the TasksController class. If the caller were performing a POST instead, then the framework would invoke the Post() method on the controller.

As you can see in the preceding route configuration, there is an optional {id} segment in the URL mapping. If the caller includes some sort of identifier at the end of the URL, the framework will select invoke the corresponding controller method that matches a signature containing a single argument. Table 5-1 shows a few examples based on the task-management service’s TasksController.

Table 5-1. Examples of URLs, Verbs, and Matching Controller Methods

|  |  |  |
| --- | --- | --- |
| URL | Verb | Controller Method |
| /api/tasks | GET | Get() |
| /api/tasks/123 | GET | Get(long id) |
| /api/tasks/123 | DELETE | Delete(long id) |
| /api/tasks | POST | Post() |
| /api/tasks/123 | PUT | Put(long id) |

The RESTful features of ASP.NET Web API help ensure that you are coding to the HTTP verbs discussed in Chapters 2 and 3. This is much cleaner and much truer to the REST style of services than using, say, ASP.NET MVC or WCF.

In addition to using arguments that come from the URL, you can add method arguments for data that arrives via the message body. You can even add an argument of .NET type HttpRequestMessage, which the framework will provide automatically. As an example, here’s the signature of the Post() method that existed on the CategoriesController in the previous edition of this book:

Post(HttpRequestMessage request, Category category)

Let’s examine each of these arguments…

## Adding an HttpRequestMessage Argument

The HttpRequestMessage is an object that you can use to examine all kinds of properties of the incoming request. It provides access to the request headers, the body, the URL used to invoke the call, client certificates, and many other valuable properties. You can also use the HttpRequestMessage object to create a response that is pre-wired to the given request object.

Note that you could instead use the controller's Request property to access the request object, because all ASP.NET Web API controllers inherit the property from the framework's ApiController base class. However, you should be careful not to couple your code to anything going on in a base class. Doing so generally makes it much more difficult to test, and it increases the fragility of your code. This is why we prefer to have the request object passed in as an argument when needed.

## Adding a Model Object Argument

The second argument, the Category object, is also inserted auto-magically by the framework. When the caller puts a JSON or XML representation of a specific model object into the body of an HTTP request, the framework will do the work of parsing the textual data into an instance of that model type.

The same applies to a PUT request where the URL also contains an identifier. Suppose you have the following CategoriesController method:

Put(HttpRequestMessage request, long id, Category category)

Now suppose the caller submits a PUT request to the following URL:

/api/categories/123

The framework will invoke the controller's Put method with a framework-provided HttpRequestMessage as the request parameter, 123 as the id, and a category object parsed from the message body as the category. This is quite amazing because you don’t need to do any special parsing of the JSON or XML content; the framework does it for you. Nor do you need to define any data contracts, as would normally be required in a WCF-based service. In line with the more recent trend towards “convention over configuration,” it just works!

## Attribute-Based Routing

While certainly powerful, the convention-based routing in ASP.NET Web API version 1 had some limitations. For example, let's say we needed to support the following:

Table 5-2. URLs, Verbs, and Controller Methods for Attribute-based Routing Example

|  |  |  |
| --- | --- | --- |
| URL | Verb | Controller Method |
| /api/tasks/123 | GET | Get(long id) |
| /api/tasks/abc | GET | Get(string taskNum) |

Here's the controller implementation:

public class TasksController : ApiController

{

public string Get(int id)

{

return "In the Get(int id) overload, id = " + id;

}

public string Get(string taskNum)

{

return "In the Get(string taskNum) overload, taskNum = " + taskNum;

}

}

If convention-based routing were our only option we'd be out of luck. The framework picks the first action method based on the route and verb, and it ignores other method overloads that would be more appropriate based on the parameter type(s). To illustrate, here are some [excerpted] HTTP message requests and responses captured using Fiddler, a popular web debugging proxy (that you can download from http://www.telerik.com/fiddler):

Request #1

GET http://localhost:50101/api/tasks/123 HTTP/1.1

Response #1

HTTP/1.1 200 OK

"In the Get(int id) overload, id = 123"

Request #2

GET http://localhost:50101/api/tasks/abc HTTP/1.1

Response #2

HTTP/1.1 400 Bad Request

{"Message":"The request is invalid.","MessageDetail":"The parameters dictionary contains a null entry…

Things are even worse if we rename the taskNum parameter to id in order to match the configured route, so that the controller appears as follows:

public class TasksController : ApiController

{

public string Get(int id)

{

return "In the Get(int id) method, id = " + id;

}

public string Get(string id)

{

return "In the Get(string id) method, id = " + id;

}

}

So now we've not only compromised some of the semantic meaning in our business domain by renaming "taskNum" to "id" (in this example they are distinct concepts), but we've also put ourselves in a situation where we have multiple actions matching the configured route. Not taking the parameter types on the methods into account, the framework has no way to determine which route is correct; therefore, it just gives up and responds with an internal server error:

HTTP/1.1 500 Internal Server Error

{"Message":"An error has occurred.","ExceptionMessage":"Multiple actions were found that match the request: …

The good news is that attribute-based routing is now available, and it solves this and many other routing problems. It is enabled by default for ASP.NET Web API applications. Let's take a look at the controller code and those same HTTP messages when attribute-based routing is used.

First, the controller code. Note the Route attributes over the methods. Not only do they specify the path to be matched against the URL, but they also contain constraints describing the action parameters. This enables the framework to make a more informed action method selection:

public class TasksController : ApiController

{

[Route("api/tasks/{id:int}")]

public string Get(int id)

{

return "In the Get(int id) overload, id = " + id;

}

[Route("api/tasks/{tasknum:alpha}")]

public string Get(string taskNum)

{

return "In the Get(string taskNum) overload, taskNum = " + taskNum;

}

}

Request #1

GET http://localhost:50101/api/tasks/123 HTTP/1.1

Response #1

HTTP/1.1 200 OK

"In the Get(int id) overload, id = 123"

Request #2

GET http://localhost:50101/api/tasks/abc HTTP/1.1

Response #2

HTTP/1.1 200 OK

"In the Get(string taskNum) overload, taskNum = abc"

That certainly looks better! With that victory behind us, we're now ready to dive into a much more complex example. Let's look at a controller that uses the ASP.NET Web API's RoutePrefixAttribute and a mix of attribute-based and convention-based routing. We'll also add a new convention-based route so that we can avoid conflating "taskNum" and "id".

First, the convention-based route configuration, this time showing the entire WebApiConfig class (note the new FindByTaskNumberRoute route):

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// Enables attribute-based routing

config.MapHttpAttributeRoutes();

// Matches route with the taskNum parameter

config.Routes.MapHttpRoute(

name: "FindByTaskNumberRoute",

routeTemplate: "api/{controller}/{taskNum}",

defaults: new { taskNum = RouteParameter.Optional }

);

// Default catch-all

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

}

}

And next, the controller class:

[RoutePrefixAttribute("api/employeeTasks")]

public class TasksController : ApiController

{

[Route("{id:int:max(100)}")]

public string GetTaskWithAMaxIdOf100(int id)

{

return "In the GetTaskWithAMaxIdOf100(int id) method, id = " + id;

}

[Route("{id:int:min(101)}")]

[HttpGet]

public string FindTaskWithAMinIdOf101(int id)

{

return "In the FindTaskWithAMinIdOf101(int id) method, id = " + id;

}

public string Get(string taskNum)

{

return "In the Get(string taskNum) method, taskNum = " + taskNum;

}

}

There are a lot of things happening here (gee, that seems to be the theme of this chapter!):

First, controller class' RoutePrefixAttribute is overriding the default behavior where the framework determines the controller class by the route name. The normal route to activate this controller is api/tasks, as we've seen earlier, but this attribute has changed it to api/employeeTasks for all methods except the non-attributed, convention-based Get method.

Now look at the GetTaskWithAMaxIdOf100 method. The method name begins with Get, which is normal for controller action methods that implement GET requests. However, the Route attribute contains a constraint limiting id to an integer with a maximum value of 100.

The FindTaskWithAMinIdOf101 method is even more interesting. Note that the method name does not begin with Get (or any other HTTP method name for that matter), so we've added an HttpGet attribute to the method to inform the framework that this is an action method suitable for GET requests. Also note the Route attribute contains a constraint limiting id to an integer with a minimum value of 101.

And last but not least, the Get method. This is plain old vanilla, convention-based routing. But do note that we had to add that route named FindByTaskNumberRoute to the WebApiConfig class to enable the framework to match this action method with its nonstandard "taskNum" parameter name.

We'll wrap up this section on routing by looking at the [excerpted] HTTP message requests and responses, captured using Fiddler, with this highly-customized routing in place…

Request #1

GET http://localhost:50101/api/employeeTasks/100 HTTP/1.1

Response #1

HTTP/1.1 200 OK

"In the GetTaskWithAMaxIdOf100(int id) method, id = 100"

Request #2

GET http://localhost:50101/api/employeeTasks/101 HTTP/1.1

Response #2

HTTP/1.1 200 OK

"In the FindTaskWithAMinIdOf101(int id) method, id = 101"

Request #3

GET http://localhost:50101/api/tasks/abc HTTP/1.1

Response #3

HTTP/1.1 200 OK

"In the Get(string taskNum) method, taskNum = abc"

Excellent! Just what we expected. And though we've reconfigured paths, added constraints, and changed controller method names, we've been able to maintain the characteristics of a RESTful interface throughout the course of this little exercise.

At this point, we've touched on some of the main capabilities of attribute-based routing, and we know enough to move forward with our task service implementation. If you'd like to dive deeper, we recommend you visit the official Microsoft ASP.NET Web API site. There you'll find an excellent piece by Mike Wasson entitled Attribute Routing in Web API 2. Be sure to check it out!

# Versioning

In this section we are going to implement the first controller action method in our task-management service. Before we start slinging code, though, we need to consider the api design we documented in Chapter 3. It is lacking an important feature; one that should be addressed before we "break ground". Security? Well, yes, but we're going to cover that later. Localization? Ok, yes, but let's assume that's not a requirement. How about versioning? Correct! And in case it's not totally obvious from the title of this section, we will create our first controller action method in a way that supports API versioning.

Implementation versioning, api versioning, content versioning

TODO:

Here we need a short explanation that explains/contrasts implementation versioning, API versioning, and content versioning.

Currently there are four basic approaches to versioning the RESTful way:

1. URI Path.

http://api/v2/Tasks/{TaskId}

1. URI Parameter.

http://api/Tasks/{TaskId}?v=2

1. Content Negotiation. This is done in the HTTP header.

Content Type: application/vnd.taskManagerApp.v2.param.json

1. Request Header. This is also done in the HTTP header.

x-taskManagerApp-version: 2

Out there on the Web you can find passionate arguments for each of these, and even combinations of these, different approaches. We encourage you to research this on your own and determine what best fits your current project. However, for the sake of maintaining focus on the ASP.NET Web API we have decided to use the first option in our task-management service. We are combining API and content versioning, so a change to the resource content (e.g., changing properties on a Web model class) constitutes a change to the API.

With that as an introduction, we will be implementing a controller action method to match the request shown in Table 5-3, so go ahead and open the solution in Visual Studio:

Table 5-3. URL and HTTP Verb for Versioned POST

|  |  |  |
| --- | --- | --- |
| URI | Verb | Description |
| /api/{apiVersion}/tasks | POST | Creates a new task; returns the new task in the response |

## Implementing POST

Add two folders to the Controllers folder; "V1" and "V2", respectively. The API project should then look like this:



Figure 5-1. API Project with Version-specific Controller Folders

Add a new controller named TasksController to each (see Figures 5-2 and 5-3).



Figure 5-2. Adding a Controller



Figure 5-3. Specifying the Empty Controller Scaffold

Now there will be two TaskController classes in the project, but the project will compile because they are in different namespaces. However, requests will always be routed to the controller in the WebApi2Book.Web.Api.Controllers.V1 namespace because the framework only matches on the controller class name without regard to the controller class' namespace. This is the case with both convention- and attribute-based routing.

To work around this shortcoming (and, more importantly, to show off some ASP.NET Web API 2 goodness) we will use attribute-based routing with custom constraints, and we will also add a custom controller selector. First, let's deal with the constraint…

### A Custom IHttpRouteConstraint

Add a folder named Routing to the WebApi2Book.Web.Common project, and then add a class named ApiVersionConstraint to the new folder. Implement the class as follows:

using System.Collections.Generic;

using System.Net.Http;

using System.Web.Http.Routing;

namespace WebApi2Book.Web.Common.Routing

{

public class ApiVersionConstraint : IHttpRouteConstraint

{

public ApiVersionConstraint(string allowedVersion)

{

AllowedVersion = allowedVersion.ToLowerInvariant();

}

public string AllowedVersion { get; private set; }

public bool Match(HttpRequestMessage request, IHttpRoute route, string parameterName,

IDictionary<string, object> values, HttpRouteDirection routeDirection)

{

object value;

if (values.TryGetValue(parameterName, out value) && value != null)

{

return AllowedVersion.Equals(value.ToString().ToLowerInvariant());

}

return false;

}

}

}

This class implements the IHttpRouteConstraint.Match method. Match will return true if the specified parameter name equals the AllowedVersion property, which is initialized in the constructor. But where does the constructor get this value? It gets it from a RoutePrefixAttribute, which we'll implement now.

### A Custom RoutePrefixAttribute

Add a class named ApiVersion1RoutePrefixAttribute to the WebApi2Book.Web.Common.Routing folder. Implement it as follows:

using System.Web.Http;

namespace WebApi2Book.Web.Common.Routing

{

public class ApiVersion1RoutePrefixAttribute : RoutePrefixAttribute

{

private const string RouteBase = "api/{apiVersion:apiVersionConstraint(v1)}";

private const string PrefixRouteBase = RouteBase + "/";

public ApiVersion1RoutePrefixAttribute(string routePrefix)

: base(string.IsNullOrWhiteSpace(routePrefix) ? RouteBase : PrefixRouteBase + routePrefix)

{

}

}

}

The main purpose of this attribute class is to encapsulate the "api/v1" part of the route template so that we don't have to copy and paste it all over the place (we will be using it a lot); it's just a bit of syntactic sugar to enhance the RoutePrefixAttribute base class. Oh, and it also allows us to demonstrate that cool new ASP.NET Web API 2 constraint we just added (note the constraint in the RouteBase string constant).

Let's add an ApiVersion1RoutePrefixAttribute to the appropriate TasksController, and then we'll review what's going on with all this. Here's the controller with the attribute applied to it:

using System.Web.Http;

using WebApi2Book.Web.Common.Routing;

namespace WebApi2Book.Web.Api.Controllers.V1

{

[ApiVersion1RoutePrefix("tasks")]

public class TasksController : ApiController

{

}

}

Looking at the ApiVersion1RoutePrefixAttribute class and the TasksController class we can see that this controller implementation is equivalent to the following:

using System.Web.Http;

using WebApi2Book.Web.Common.Routing;

namespace WebApi2Book.Web.Api.Controllers.V1

{

[RoutePrefix("api/{apiVersion:apiVersionConstraint(v1)}/tasks")]

public class TasksController : ApiController

{

}

}

Recalling what we learned in the Attribute-Based Routing section earlier in the chapter we now recognize that the RoutePrefix attribute is configured to match a URL path of api/{apiVersion}/tasks, where the apiVersion parameter is constrained by our custom IHttpRouteConstraint to a value of "v1". Make sense?

Now let's finish up stubbing out the controllers. We'll flesh out the real implementation later; for now, we're trying to demonstrate that we can properly support versioned routes. First, "implement" the V1 controller…

using System.Net.Http;

using System.Web.Http;

using WebApi2Book.Web.Api.Models;

using WebApi2Book.Web.Common.Routing;

namespace WebApi2Book.Web.Api.Controllers.V1

{

[ApiVersion1RoutePrefix("tasks")]

public class TasksController : ApiController

{

[Route("", Name = "AddTaskRoute")]

[HttpPost]

public Task AddTask(HttpRequestMessage requestMessage, Task newTask)

{

return new Task

{

Subject = "In v1, newTask.Subject = " + newTask.Subject

};

}

}

}

… and now the V2 controller:

using System.Net.Http;

using System.Web.Http;

using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.Controllers.V2

{

[RoutePrefix("api/{apiVersion:apiVersionConstraint(v2)}/tasks")]

public class TasksController : ApiController

{

[Route("", Name = "AddTaskRouteV2")]

[HttpPost]

public Task AddTask(HttpRequestMessage requestMessage, Models.Task newTask)

{

return new Task

{

Subject = "In v2, newTask.Subject = " + newTask.Subject

};

}

}

}

Note that for the V2 controller we're using the RoutePrefix attribute directly rather than subclassing it. The purpose is to emphasize that the custom ApiVersion1RoutePrefixAttribute is merely syntactic sugar; it doesn't affect the processing in any way. Also note that the two route names are unique. We're not particularly fond of runtime exceptions, so we will ensure globally unique route names, as required by ASP.NET Web API.

Ok, now we're almost ready to process a message. First, we need to implement the custom controller selector, and then we need to wire up the custom constraint and the custom controller selector with the ASP.NET Web API framework. So without further ado…

### A Custom IHttpControllerSelector

Our controller selector implementation was inspired by Mike Wasson's MSDN blog post entitled, ASP.NET Web API: Using Namespaces to Version Web APIs. The implementation borrowed heavily from the official Microsoft CodePlex NamespaceControllerSelector sample referenced in that blog post. We made some simplifications (e.g., eliminated the code that checked for duplicate paths) and we made some enhancements required to address changes in the ASP.NET Web API framework (e.g., note the usage of IHttpRouteData subroutes), but the basic design and implementation comes straight from CodePlex. Thanks, Microsoft! Although it makes one wish this were already part of the framework…

Anyway, go ahead and implement the custom controller selector as follows in the WebApi2Book.Web.Common project:

using System;

using System.Collections.Generic;

using System.Globalization;

using System.Linq;

using System.Net;

using System.Net.Http;

using System.Web.Http;

using System.Web.Http.Controllers;

using System.Web.Http.Dispatcher;

using System.Web.Http.Routing;

namespace WebApi2Book.Web.Common

{

/// <summary>

/// Controller selector implementation that uses the controller namespace

/// as part of the selection decision.

/// </summary>

public class NamespaceHttpControllerSelector : IHttpControllerSelector

{

private readonly HttpConfiguration \_configuration;

private readonly Lazy<Dictionary<string, HttpControllerDescriptor>> \_controllers;

public NamespaceHttpControllerSelector(HttpConfiguration config)

{

\_configuration = config;

\_controllers = new Lazy<Dictionary<string, HttpControllerDescriptor>>(InitializeControllerDictionary);

}

public HttpControllerDescriptor SelectController(HttpRequestMessage request)

{

var routeData = request.GetRouteData();

if (routeData == null)

{

throw new HttpResponseException(HttpStatusCode.NotFound);

}

var controllerName = GetControllerName(routeData);

if (controllerName == null)

{

throw new HttpResponseException(HttpStatusCode.NotFound);

}

var namespaceName = GetVersion(routeData);

if (namespaceName == null)

{

throw new HttpResponseException(HttpStatusCode.NotFound);

}

var controllerKey = String.Format(CultureInfo.InvariantCulture, "{0}.{1}", namespaceName, controllerName);

HttpControllerDescriptor controllerDescriptor;

if (\_controllers.Value.TryGetValue(controllerKey, out controllerDescriptor))

{

return controllerDescriptor;

}

throw new HttpResponseException(HttpStatusCode.NotFound);

}

public IDictionary<string, HttpControllerDescriptor> GetControllerMapping()

{

return \_controllers.Value;

}

private Dictionary<string, HttpControllerDescriptor> InitializeControllerDictionary()

{

var dictionary = new Dictionary<string, HttpControllerDescriptor>(StringComparer.OrdinalIgnoreCase);

var assembliesResolver = \_configuration.Services.GetAssembliesResolver();

var controllersResolver = \_configuration.Services.GetHttpControllerTypeResolver();

var controllerTypes = controllersResolver.GetControllerTypes(assembliesResolver);

foreach (var controllerType in controllerTypes)

{

var segments = controllerType.Namespace.Split(Type.Delimiter);

var controllerName =

controllerType.Name.Remove(controllerType.Name.Length -

DefaultHttpControllerSelector.ControllerSuffix.Length);

var controllerKey = String.Format(CultureInfo.InvariantCulture, "{0}.{1}", segments[segments.Length - 1],

controllerName);

if (!dictionary.Keys.Contains(controllerKey))

{

dictionary[controllerKey] = new HttpControllerDescriptor(\_configuration, controllerType.Name,

controllerType);

}

}

return dictionary;

}

private T GetRouteVariable<T>(IHttpRouteData routeData, string name)

{

object result;

if (routeData.Values.TryGetValue(name, out result))

{

return (T)result;

}

return default(T);

}

private string GetControllerName(IHttpRouteData routeData)

{

var subroute = routeData.GetSubRoutes().FirstOrDefault();

if (subroute == null) return null;

var dataTokenValue = subroute.Route.DataTokens.First().Value;

if (dataTokenValue == null) return null;

var controllerName =

((HttpActionDescriptor[])dataTokenValue).First()

.ControllerDescriptor.ControllerName.Replace("Controller", string.Empty);

return controllerName;

}

private string GetVersion(IHttpRouteData routeData)

{

var subRouteData = routeData.GetSubRoutes().FirstOrDefault();

if (subRouteData == null) return null;

return GetRouteVariable<string>(subRouteData, "apiVersion");

}

}

}

There's a lot going on in that controller selector, but it's explained well in Wasson's blog post so we will refer you to it rather than continue to dwell on this topic: <http://blogs.msdn.com/b/webdev/archive/2013/03/08/using-namespaces-to-version-web-apis.aspx>.

We're going to move on now to the configuration step, where all of this routing and versioning finally comes together.

### Configuration

We need to register our constraint with ASP.NET Web API so that it gets applied to incoming requests. We also need to configure our custom controller selector. We accomplish this by implementing the WebApiConfig class as follows (go ahead and type/copy it in):

using System.Web.Http;

using System.Web.Http.Dispatcher;

using System.Web.Http.Routing;

using WebApi2Book.Web.Common;

using WebApi2Book.Web.Common.Routing;

namespace WebApi2Book.Web.Api

{

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

var constraintsResolver = new DefaultInlineConstraintResolver();

constraintsResolver.ConstraintMap.Add("apiVersionConstraint", typeof (ApiVersionConstraint));

config.MapHttpAttributeRoutes(constraintsResolver);

config.Services.Replace(typeof (IHttpControllerSelector),

new NamespaceHttpControllerSelector(config));

}

}

}

In the first three lines of the Register method are used to configure the version constraint. Our ApiVersionConstraint is registered with a constraint resolver, which the framework uses to instantiate the constraint at runtime. The last two lines in the method are used to wire-in our custom controller selector, replacing the default, namespace-unaware, controller selector.

With that in place, we are now finally ready to build and test the app.

### Demo

With the WebApi2Book.Web.Api project configured as the startup project in Visual Studio, we'll hit F5 to start the application. If you're following along you'll see it load an error page in your browser (Figure 5-4):



Figure 5-4. Application Error Page

Don't worry, this is expected. There are no routes configured in our application that match this base address.

Now we'll use Fiddler to send an HTTP POST message via the V1 route:

V1 Request

POST http://localhost:61589/api/v1/tasks HTTP/1.1

Content-Type: text/json

{"Subject":"Fix something important"}

Go ahead and send the message using Fiddler or your favorite Web proxy debugging tool. You should see the following response:

V1 Response (abbreviated)

HTTP/1.1 200 OK

Content-Type: text/json; charset=utf-8

{"TaskId":null,"Subject":"In v1, newTask.Subject = Fix something important","StartDate":null,"DueDate":null,"CreatedDate":null,"CompletedDate":null,"Status":null,"Assignees":null,"Links":[]}

Excellent! Note the "Subject" value in the response… just as we implemented it! Now do the same for V2…

V2 Request

POST http://localhost:61589/api/v2/tasks HTTP/1.1

Content-Type: text/json

{"Subject":"Fix something important"}

V2 Response (abbreviated)

HTTP/1.1 200 OK

Content-Type: text/json; charset=utf-8

{"TaskId":null,"Subject":"In v2, newTask.Subject = Fix something important","StartDate":null,"DueDate":null,"CreatedDate":null,"CompletedDate":null,"Status":null,"Assignees":null,"Links":[]}

Perfect! Ok, this is great. We've learned about routing, including many of the new capabilities made available by ASP.NET Web API 2. We've learned about constraints and controller selectors. And we've successfully processed an HTTP request. However, we must admit that this is rather "hello-worldish"; our controller actions aren't doing anything meaningful. Which leads us to our next topic: Dependencies. Inside of dependencies is where the "real" work gets done, at least from a business perspective.

# Dependencies

If the controllers are going to do anything useful, and if they are going to be implemented in a well-architected manner using SOLID design principles, then they will depend heavily upon functionality provided by other classes. An example of this would be a database repository, which is an object that can be used to query the database and save changes back to the database. The database repository would be considered a dependency of the controller class that uses it; that is, the controller depends on it for functionality not implemented within the controller itself.

SOLID Design Principles

If you don't know with what we mean by SOLID Design Principles, then please do yourself a huge favor and familiarize yourself with them. Chad Myers has written a concise introduction to SOLID, referencing source material from Robert Martin and others, available here for your perusal on the Los Techies website:

http://lostechies.com/chadmyers/2008/03/08/pablo-s-topic-of-the-month-march-solid-principles/

The main idea is that the methods in the controllers should not be doing much more than simply using the functionality offered by various dependencies. And that brings us to the point of this section: managing dependencies within the application. Once you adopt the approach of using dependencies for most/all functionality, you need a pattern and tool for configuring and obtaining those dependencies. The easiest way to approach this is summarized in these two points:

* Push all dependencies up to the constructor.
* Configure the application to use dependency injection.

## Constructor Injection of Dependencies

The concept of pushing all of the dependencies up to the constructor is really quite simple, but it can be tough to grasp and put into practice for the Dependency Injection (DI) novice. Don't worry if you are new to DI, because you will become quite familiar with it as we continue to implement the task service.

Think of it this way: a class should not use any behavior that does not come through the constructor in the form of an abstraction (i.e., an interface). This includes even seemingly harmless classes such as System.DateTime, System.IO.File, System.Environment and many other basic utility classes within the .NET Framework. If a class is using the services of another class, then that other class needs to be injected in through the constructor. This also applies to static properties and methods. For example, if some piece of code needs to use the static DateTime.Now property, then the DateTime.Now functionality should be wrapped in an injectable adapter class instead of used directly.

Again, you will see working examples of all of this later. But first, let's configure a DI tool to provide our code with the dependencies it needs at runtime.

## Configuring Ninject Dependency Injection

As we mentioned in Chapter 3, we've chosen to use the open source Ninject DI tool. The same principles apply to all DI tools, though, so if you prefer a different one you probably just need to account for the differences in syntax.

There are three things we need to take care of regarding Ninject in the service. The first two need to be done in any kind of application, while the third is somewhat unique to an ASP.NET Web API service. Table 5-4 briefly describes each of these activities.

Table 5-4. Ninject–related activities

|  |  |
| --- | --- |
| Activity | Description |
| Container configuration | Make sure a DI container is created during application start-up and remains in memory until the application shuts down. (You can think of the container as the object that contains the dependencies.) |
| Container bindings | This is where we link interfaces to concrete implementations so that the dependencies can be resolved at run time. |
| IDependencyResolver for Ninject | This tells ASP.NET Web API to ask Ninject for all dependencies required at run time by the dependent objects. This is the key that allows you to push dependencies up to the constructor on the controllers. |

## Container Configuration

In order for the DI container to be useful for creating objects and injecting them into constructors, and to control the lifetime of those objects, the container must be available for the entire duration that the application is running. In other words, a single container instance must meet these criteria:

* Be created early in the application start up process
* Be available at all times while the application is running
* Be destroyed as one of the last steps the application takes during shutdown

While it is certainly possible to wire up Ninject manually, the easiest and most reliable option for making sure the container is always available is to install the Ninject.Web.Common.WebHost NuGet package. If you were following the steps in Chapter 4, then you already did this. The package generates code that handles creating and destroying a container instance within the startup and shutdown methods of the NinjectWebCommon class it adds to the app\_start folder of the WebApi2Book.Web.Api project. The generated NinjectWebCommon class does require some tweaking to establish the container bindings and to register itself with the Web API framework's global configuration, but hey, it gets you 99% of the way there. We'll take a look at it soon.

## Container Bindings

Now that the container itself is configured to be around while the application is running, we need to give it the type mappings so that it can instantiate and help inject dependencies into the objects that require them. This step is essentially just mapping interface types to implementation types, and in some cases, to implementation methods or variables. In a previous example we mentioned wrapping DateTime.Now in an adapter class and injecting the interface into the dependent class(es). The particular mapping to accomplish this is as follows:

container.Bind<IDateTime>().To<DateTimeAdapter>().InSingletonScope();

Note that this isn't actually creating an instance of DateTimeAdapter; Ninject does that as such instances are required. Also note that by specifying InSingletonScope we are directing Ninject to provide a shared instance to all dependent objects for the entire lifetime of the application. Another lifetime scope we'll be using very frequently is InRequestScope, which provides a shared instance to all dependent objects processing the same HTTP request message. We also sometimes use the ToConstant lifetime scope, which manages an application-level singleton instance that our code - not Ninject - has instantiated.

Armed with this vast knowledge about dependencies, DI, and Ninject, let's get back to writing some code! First off, let's go ahead and implement that DateTimeAdapter class and corresponding interface in the root of the WebApi2Book.Common project as follows:

DateTimeAdapter class

using System;

namespace WebApi2Book.Common

{

public class DateTimeAdapter : IDateTime

{

public DateTime UtcNow

{

get { return DateTime.UtcNow; }

}

}

}

IDateTime interface

using System;

namespace WebApi2Book.Common

{

public interface IDateTime

{

DateTime UtcNow { get; }

}

}

Next, anticipating the need for logging (doesn't every significant app need some form of diagnostic logging?), add a project folder named Logging to the WebApi2Book.Common project. Then add the following interface and adapter class to that folder; this is to prevent tight coupling to the static LogManager.GetLogger method:

ILogManager interface

using System;

using log4net;

namespace WebApi2Book.Common.Logging

{

public interface ILogManager

{

ILog GetLog(Type typeAssociatedWithRequestedLog);

}

}

LogManagerAdapter class

using System;

using log4net;

namespace WebApi2Book.Common.Logging

{

public class LogManagerAdapter : ILogManager

{

public ILog GetLog(Type typeAssociatedWithRequestedLog)

{

var log = LogManager.GetLogger(typeAssociatedWithRequestedLog);

return log;

}

}

}

And now we're ready to configure our first actual dependency bindings. So let's add a new class, named NinjectConfigurator, to the App\_Start folder of the WebApi2Book.Web.Api project. Implement it as follows:

using log4net.Config;

using Ninject;

using WebApi2Book.Common;

using WebApi2Book.Common.Logging;

namespace WebApi2Book.Web.Api

{

public class NinjectConfigurator

{

public void Configure(IKernel container)

{

AddBindings(container);

}

private void AddBindings(IKernel container)

{

ConfigureLog4net(container);

container.Bind<IDateTime>().To<DateTimeAdapter>().InSingletonScope();

}

private void ConfigureLog4net(IKernel container)

{

XmlConfigurator.Configure();

var logManager = new LogManagerAdapter();

container.Bind<ILogManager>().ToConstant(logManager);

}

}

}

See the IDateTime and ILogManager bindings? Hopefully they make sense now. However, how does this get invoked? We see that AddBindings calls ConfigureLog4net (by the way, that XmlConfigurator.Configure() is required to configure log4net), and we see that Configure calls AddBindings, but what calls Configure? The answer to this question lies in the aforementioned NinjectWebCommon class, which we will explore at the end of this section on dependencies.

## IDependencyResolver for Ninject

Here is the NinjectDependencyResolver we will use for our task-management service. Note that it takes an instance of a Ninject container in its constructor.

using System;

using System.Collections.Generic;

using System.Web.Http.Dependencies;

using Ninject;

namespace WebApi2Book.Web.Common

{

public sealed class NinjectDependencyResolver : IDependencyResolver

{

private readonly IKernel \_container;

public NinjectDependencyResolver(IKernel container)

{

\_container = container;

}

public IKernel Container

{

get { return \_container; }

}

public object GetService(Type serviceType)

{

return \_container.TryGet(serviceType);

}

public IEnumerable<object> GetServices(Type serviceType)

{

return \_container.GetAll(serviceType);

}

public IDependencyScope BeginScope()

{

return this;

}

public void Dispose()

{

GC.SuppressFinalize(this);

}

}

}

Go ahead and implement it in the root of the WebApi2Book.Web.Common project. The methods to note are GetService() and GetServices(). All they really do is delegate to the Ninject container to get object instances for the requested service types. Also note that in the GetService() method we are using the TryGet() method instead of the Get() method. This is to prevent Ninject from blowing up if it is asked for a dependency that it can't provide because the dependency - or one of its dependencies - was never registered.

And now it's time to put it all together; it's time for us to complete the implementation of the NinjectWebCommon class.

## Completing NinjectWebCommon

Take a look at the NinjectWebCommon class that the Ninject.Web.Common.WebHost NuGet package added, and then modify the implementation so that it appears as follows:

using System;

using System.Web;

using System.Web.Http;

using Microsoft.Web.Infrastructure.DynamicModuleHelper;

using Ninject;

using Ninject.Web.Common;

using WebActivatorEx;

using WebApi2Book.Web.Api;

using WebApi2Book.Web.Common;

[assembly: WebActivatorEx.PreApplicationStartMethod(typeof (NinjectWebCommon), "Start")]

[assembly: ApplicationShutdownMethod(typeof (NinjectWebCommon), "Stop")]

namespace WebApi2Book.Web.Api

{

public static class NinjectWebCommon

{

private static readonly Bootstrapper Bootstrapper = new Bootstrapper();

public static void Start()

{

DynamicModuleUtility.RegisterModule(typeof (OnePerRequestHttpModule));

DynamicModuleUtility.RegisterModule(typeof (NinjectHttpModule));

IKernel container = null;

Bootstrapper.Initialize(() =>

{

container = CreateKernel();

return container;

});

var resolver = new NinjectDependencyResolver(container);

GlobalConfiguration.Configuration.DependencyResolver = resolver;

}

public static void Stop()

{

Bootstrapper.ShutDown();

}

private static IKernel CreateKernel()

{

var kernel = new StandardKernel();

try

{

kernel.Bind<Func<IKernel>>().ToMethod(ctx => () => new Bootstrapper().Kernel);

kernel.Bind<IHttpModule>().To<HttpApplicationInitializationHttpModule>();

RegisterServices(kernel);

return kernel;

}

catch

{

kernel.Dispose();

throw;

}

}

private static void RegisterServices(IKernel kernel)

{

var containerConfigurator = new NinjectConfigurator();

containerConfigurator.Configure(kernel);

}

}

}

Our modified version is different in some subtle ways. First, the significant changes that we made:

* We modified the Start method to register our dependency resolver with the Web API configuration. In doing so, we have directed the framework to hit the configured Ninject container instance to resolve any dependencies that are needed.
* We modified the RegisterServices method to configure the container bindings using the NinjectConfigurator class. So now we've finally answered the question about what calls the Configure method: NinjectWebCommon.RegisterServices does!

It's important to note that registration of our dependency resolver with Web API and configuration of container bindings by the NinjectConfigurator.Configure method are both called (the former directly, the latter indirectly) from the Start method, which is called during application start-up. In this way, all of this setup is completed before any of the controllers - which rely on dependencies being injected into them - are ever created.

Now, for completeness, here are the insignificant changes that we made:

* We removed the comments. Nothing against comments, we're just pressed for space!
* We changed the namespace to WebApi2Book.Web.Api. It's common practice to use this namespace for files in the App\_Start folder. Case in point: look at the namespace of the WebApiConfig class that Visual Studio automatically added to the WebApi2Book.Web.Api project.
* We moved the using statements outside of the namespace. No particular reason, other than the fact that this is how ReSharper is configured by default.

As we build the task-management service we will find ourselves coming back to the NinjectConfigurator fairly often. This is because the classes used for various behaviors will continue to change as the application evolves. Simply put, these mappings are not etched in stone, and you should expect to modify them as time goes on.

# NHibernate Configuration and Mappings

We now turn our attention to configuring NHibernate to work with the database and with the domain model (entity) classes. We’ll be using the Fluent NHibernate library that we installed in Chapter 4 for this.

## Database Configuration - Overview

As with any approach to data access, at some point the underlying framework must be told how to connect to the database. And because NHibernate is database-provider agnostic, we must also tell it which provider we’re using, and even which version of which provider. This allows NHibernate to load the appropriate driver for dynamically generating the DML (Data Manipulation Language) it needs to interact with the database. For example, creating a SELECT statement in SQL Server will be a little different in some cases than creating a SELECT statement in Oracle or MySQL. Indeed, one of the advantages of using an Object Relational Mapper (ORM) like NHibernate is that one can, in theory, change database providers without having to change anything about the domain model or any code that uses it. One would most likely need to update the NHibernate configuration and mapping definitions, however. It is for this reason that we have split the data layer into two separate projects:

* The WebApi2Book.Data project, which includes the entire domain model; i.e., all of the entity classes. None of this is dependent on a specific database provider; that is, the domain model will be the same whether you’re working with SQL Server or Oracle.
* The WebApi2Book.Data.SqlServer project, which contains the domain model NHibernate mapping definitions, and which will contain the query processors once we get to them later in the book. These could possibly change when swapping out database providers.

Actually wiring-in the database configuration with the ASP.NET Web Api framework requires a small bit of code located in the application’s start-up logic, along with some supporting classes (that are easy to isolate) and some config file-based configuration. That means deciding to switch from SQL Server to Oracle, for example, can be accomplished relatively noninvasively.

So let's begin database configuration for the task-management service. You'll notice that although the wire-in itself doesn't require many lines of code, it does involve many related "moving parts". So let's go through this carefully!

## Adding Concurrency Support to Entities

The first thing we'll do is add concurrency support to the entities that we introduced back in Chapter 4. Start this off by adding a new interface to the WebApi2Book.Data.Entities namespace:

IVersionedEntity Interface

namespace WebApi2Book.Data.Entities

{

public interface IVersionedEntity

{

byte[] Version { get; set; }

}

}

Then, have each of the entity classes implement that interface. This is trivial, because the Version property is already defined in each entity class. As an example, the Status class should now look like this:

namespace WebApi2Book.Data.Entities

{

public class Status : IVersionedEntity

{

public virtual long StatusId { get; set; }

public virtual string Name { get; set; }

public virtual int Ordinal { get; set; }

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

}

}

## Entity Mapping

Next, we need to provide all of the code that will map between the entities and the database’s tables and columns. Depending on the database model you’re trying to map, and depending on how much you are trying to abstract away the domain model itself, building these mappings can be anywhere from very simple to very complex. We're focusing on the ASP.NET Web API, so we have designed the task-management service to be on the very simple end of the scale to map.

Since we’ve already gone through the entity classes and the data model in Chapter 4, these mapping definitions should be fairly self-explanatory. We'll point out a few key things following the code; speaking of which, go ahead and all the following classes to a new folder named "Mapping" in the WebApi2Book.Data.SqlServer project:

VersionedClassMap class

using FluentNHibernate.Mapping;

using WebApi2Book.Data.Entities;

namespace WebApi2Book.Data.SqlServer.Mapping

{

public abstract class VersionedClassMap<T> : ClassMap<T> where T : IVersionedEntity

{

protected VersionedClassMap()

{

Version(x => x.Version).Column("ts").CustomSqlType("Rowversion").Generated.Always().UnsavedValue("null");

}

}

}

StatusMap class

using WebApi2Book.Data.Entities;

namespace WebApi2Book.Data.SqlServer.Mapping

{

public class StatusMap : VersionedClassMap<Status>

{

public StatusMap()

{

Id(x => x.StatusId);

Map(x => x.Name).Not.Nullable();

Map(x => x.Ordinal).Not.Nullable();

}

}

}

TaskMap class

using FluentNHibernate.Mapping;

using WebApi2Book.Data.Entities;

namespace WebApi2Book.Data.SqlServer.Mapping

{

public class TaskMap : VersionedClassMap<Task>

{

public TaskMap()

{

Id(x => x.TaskId);

Map(x => x.Subject).Not.Nullable();

Map(x => x.StartDate).Nullable();

Map(x => x.DueDate).Nullable();

Map(x => x.CompletedDate).Nullable();

Map(x => x.CreatedDate).Not.Nullable();

References(x => x.Status, "StatusId");

References(x => x.CreatedBy, "CreatedUserId");

HasManyToMany(x => x.Users)

.Access.ReadOnlyPropertyThroughCamelCaseField(Prefix.Underscore)

.Table("TaskUser")

.ParentKeyColumn("TaskId")

.ChildKeyColumn("UserId");

}

}

}

UserMap class

using WebApi2Book.Data.Entities;

namespace WebApi2Book.Data.SqlServer.Mapping

{

public class UserMap : VersionedClassMap<User>

{

public UserMap()

{

Id(x => x.UserId);

Map(x => x.Firstname).Not.Nullable();

Map(x => x.Lastname).Not.Nullable();

Map(x => x.Username).Not.Nullable();

}

}

}

The first thing you might notice is that all of the mapping code is contained within each class’s constructor. Second, notice the use of the VersionedClassMap<T> base class for each of the map classes. This custom class leverages NHibernate’s ability to check for dirty records in the database, based on a Rowversion column on each table. The crazy-long statement in the VersionedClassMap implementation can be broken down as follows:

* Use the Version property on each entity class.
* The database column supporting versioning is named ts.
* The SQL data type is a Rowversion.
* NHibernate should always let the database generate the value, as opposed to you or NHibernate supplying the value.
* Prior to a database save, the in-memory value of the Version property will be null.

Again, all of this is to let NHibernate protect against trying to update dirty records. Placing this statement in the constructor of the base class means it will automatically be executed by every ClassMap implementation in the Mapping folder. Implementing the IVersionedEntity interface just ensures that the class contains a Version property, and because of what we implemented in the previous section, we know that each of the entity classes implements this interface.

ClassMap<T>, the base class of VersionedClassMap<T>, is defined in the Fluent NHibernate library, and it simply provides a means of configuring entity-to-database mapping through code (as opposed to using XML files). This mapping code is placed in the each mapping class’s constructor. For example, the StatusMap mapping class constructor contains all of the mapping for the Status entity.

The two main ClassMap<T> methods used in the application’s mapping classes are the Id and Map methods. The Id method can be called only once, and it’s used to tell NHibernate which property on the entity class is used as the object identifier.

The Map is used to configure individual properties on the entities. By default, NHibernate will assume the mapped column name is the same as the given property name. If it’s not, an overload can be used to specify the column name. Additionally, because this is a fluent-style interface, we can chain other property and column specifics together. For example, the UserMap class’s Firstname mapping also includes a specification telling NHibernate to treat the column as not nullable.

We have implemented one ClassMap<T> for each entity. The StatusMap and UserMap mapping classes are straightforward. They each have their properties mapped, the first of which is the class’s identifier. We don’t need to specify the column name because the column name happens to match the property name.

The TaskMap class is slightly more complicated; it is used to map the Task's relationships to other entities. We'll explore it next.

## Mapping Relationships

Mapping many-to-one references is actually relatively simple. We see in TaskMap that a Task has a reference to a Status. The corresponding column name in the Task table is StatusId. The reference to the user that created the task is similar.

Many-to-many relationships are more complicated because you must identify the linking table in the database, as well as the linking table’s parent and child columns. For example, to link a set of users to a task, the TaskUser table will contain a record for each user linked to that task. The users will be loaded into the Users collection property on the Task object.

Regarding that collection, the Task class defines the Users property with only a getter, to prevent the developer from replacing the entire collection. Note that we also create an empty collection upon class instantiation, which allows us to immediately call Add on the property without having to create a new collection first. As such, the TaskMap class defines the Users property map with this bit of code:

.Access.ReadOnlyPropertyThroughCamelCaseField(Prefix.Underscore)

This tells NHibernate to “access the Users read-only property through a camel-cased field that is named with an underscore prefix.” Make sense?

## Database Configuration - Bringing it all Together

Okay, now that we've layed the groundwork for it, let's finish off this section by hooking NHibernate up to the ASP.NET Web Api. First, add the following methods to the NinjectConfigurator class we discussed previously…

private void ConfigureNHibernate(IKernel container)

{

var sessionFactory = Fluently.Configure()

.Database(

MsSqlConfiguration.MsSql2008.ConnectionString(

c => c.FromConnectionStringWithKey("WebApi2BookDb")))

.CurrentSessionContext("web")

.Mappings(m => m.FluentMappings.AddFromAssemblyOf<TaskMap>())

.BuildSessionFactory();

container.Bind<ISessionFactory>().ToConstant(sessionFactory);

container.Bind<ISession>().ToMethod(CreateSession).InRequestScope();

}

private ISession CreateSession(IContext context)

{

var sessionFactory = context.Kernel.Get<ISessionFactory>();

if (!CurrentSessionContext.HasBind(sessionFactory))

{

var session = sessionFactory.OpenSession();

CurrentSessionContext.Bind(session);

}

return sessionFactory.GetCurrentSession();

}

… and then modify the AddBindings method so that it is implemented as shown:

private void AddBindings(IKernel container)

{

ConfigureLog4net(container);

ConfigureNHibernate(container);

container.Bind<IDateTime>().To<DateTimeAdapter>().InSingletonScope();

}

Finally, add the following code to the Web.Api project’s web.config file, right below the closing configSections tag that we added in Chapter 4.

<connectionStrings>

<add name="WebApi2BookDb" providerName="System.Data.SqlClient" connectionString="Server=.;

initial catalog=WebApi2BookDb;Integrated Security=True;Application Name=WebApi2Book API Website" />

</connectionStrings>

Let's review what we just did, starting with the ConfigureNHibernate method. This method sets four properties and then builds the ISessionFactory object:

* The first property indicates that we are using a version of SQL Server that is compatible with SQL Server 2012.
* The second property specifies the database connection string, and that it should be loaded from the web.config file’s WebApi2BookDb connection string value.
* The third property tells NHibernate that we plan to use its web implementation to manage the current session object. We’ll explore session management more in the next section, but this essentially scopes a single database session to a single web request; i.e., one database session per call.
* The fourth property tells NHibernate which assembly to use to load mappings. The WebApi2Book.Data.SqlServer project contains all of those mappings, so we just gave it a class name that exists in that assembly.

Then the method calls BuildSessionFactory, which returns a fully configured NHibernate ISessionFactory instance. The ISessionFactory instance is then placed into the Ninject container with this statement:

container.Bind<ISessionFactory>().ToConstant(sessionFactory);

The statement that follows, where we tell Ninject how to get ISession objects, will be discussed in the next section. For now, just understand that we’ve configured NHibernate to be able to talk to the database. This is good progress!

# Managing the Unit of Work

As discussed in Chapter 3, one of the key benefits in using NHibernate is that it allows for separation between a repository and its unit of work, the ISession object. In a service application like the task-management service, you want the database session object to span a complete service call. This provides support for three very important aspects of database class within a web request:

* Keep fetched domain objects in memory, so that they are consistent across all operations within a single web request.
* Use in-memory objects to facilitate caching.
* Track all changes made to domain objects, so that saving the changes in an ISession instance will save all changes made during a single web request; this is especially important for updates that involve foreign key relationships.

Because none of these aspects are tied to a repository or specific set of domain model classes, you can use many repositories and any of the model classes throughout a web request, as long as only one ISession instance is used. As such, it is important to ensure that every database operation within a web request uses the same ISession object. This is what is meant by “managing the unit of work.”

Fortunately, NHibernate comes equipped with the ability to utilize the ASP.NET HttpContext to manage instances of ISession. In the previous section, you used the CurrentSessionContext("web") call to tell NHibernate that you intend to use the HttpContext object. Beyond that, you need to use a special class within NHibernate called the CurrentSessionContext. You can use this class to manually bind an instance of ISession to the underlying HttpContext, and then turn around and unbind it when the request is complete.

It’ll be easier just to look at the code. You configure the ISession mapping with Ninject like this:

container.Bind<ISession>().ToMethod(CreateSession);

This tells Ninject to call the CreateSession() method whenever an object needs an ISession injected into its constructor (e.g., a controller or a repository). That method looks like this:

private ISession CreateSession(IContext context)

{

var sessionFactory = context.Kernel.Get<ISessionFactory>();

if (!CurrentSessionContext.HasBind(sessionFactory))

{

var session = sessionFactory.OpenSession();

CurrentSessionContext.Bind(session);

}

return sessionFactory.GetCurrentSession();

}

First, you obtain an instance of the ISessionFactory that you configured during application start-up (covered in the previous section). You then use that ISessionFactory object to check whether an existing ISession object has already been bound to the CurrentSessionContext object. If not, then you open a new session (using the ISessionFactory object), and then immediately bind it to the context. (By the way, opening a session in NHibernate is somewhat similar to opening a connection to the database.) Finally, you return the currently bound ISession object.

This code will be executed every time any object requests an ISession object via Ninject (e.g., through constructor injection). This approach ensures that, for a single request, you only ever create a single ISession object. Here’s an example from the UserRepository constructor:

private readonly ISession \_session;

public UserRepository(ISession session)

{

\_session = session;

}

The repository can then use the injectedISession and simply assume it is active and being managed by something outside itself. In other words, the repository doesn’t need to worry about session lifetime, database connections, or transactions. It just has to use the ISession to access the database and let other components take care of the rest.

To close out the ISession object, you’re going to use an implementation of an MVC ActionFilterAttribute—which will decorate the controllers to ensure all calls to them are using a properly managed ISession instance. Here are the relevant parts of the custom attribute:

public class LoggingNHibernateSessionAttribute : ActionFilterAttribute

{

public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext)

{

EndTransaction(actionExecutedContext);

CloseSession();

LogException(actionExecutedContext);

LogAction(actionExecutedContext.ActionContext.ActionDescriptor, "EXITING");

}

private void CloseSession()

{

var container = GetContainer();

var sessionFactory = container.Get<ISessionFactory>();

if (CurrentSessionContext.HasBind(sessionFactory))

{

var session = sessionFactory.GetCurrentSession();

session.Close();

session.Dispose();

CurrentSessionContext.Unbind(sessionFactory);

}

}

You’ll explore transaction control and logging in a bit. For now, let’s look at the CloseSession() method that is called from the OnActionExecuted() override. Similar to the CreateSession() method just discussed, the CloseSession() method first obtains the ISessionFactory instance from the container (using a GetContainer() method that isn’t shown here). It then uses that object to check whether an ISession object is currently bound to the CurrentSessionContext. If so, it obtains the ISession object with the GetCurrentSession() method, and then closes and disposes of it. Finally, you need to unbind the ISession object from the current ISessionFactory instance.

To make sure the controller methods take advantage of this “automatic” ISession disposal, you simply need to make sure they are decorated with the custom attribute, like this:

[LoggingNHibernateSession]

public class TasksController : ApiController

And then watch the magic happen!

This chapter has covered a lot of complex material pretty quickly. Thus, I strongly encourage you to read along in the example code supplied with this book.

# Database Transaction Control

The last thing to cover in this chapter is transaction control for the database operations. Much as you can have a single ISession instance span all operations within a single web request, you also want to wrap all operations within a single database transaction (by default).You also don’t want the controller or repository code to worry about transactions at all. It should just work!

To make this happen, you’re going to use the custom attribute discussed in the previous section (where you used it to close and dispose of the ISession object). Let’s look at the relevant attribute code. First, you want to override both the OnActionExecuting() and OnActionExecuted() methods. These are called by MVC before the controller action (i.e., method) is executed, and again after it is executed. Here’s the code for those two methods:

public override void OnActionExecuting(HttpActionContext actionContext)

{

LogAction(actionContext.ActionDescriptor, "ENTERING");

BeginTransaction();

}

public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext)

{

EndTransaction(actionExecutedContext);

CloseSession();

LogException(actionExecutedContext);

LogAction(actionExecutedContext.ActionContext.ActionDescriptor, "EXITING ");

}

Let’s ignore the log-related code for now. In the OnActionExecuting() method, you call BeginTransaction(), and then you call EndTransaction() in the OnActionExecuted() method. The definition of those two methods is as follows:

public void BeginTransaction()

{

var session = GetCurrentSession();

if (session != null)

{

session.BeginTransaction();

}

}

public void EndTransaction(HttpActionExecutedContext filterContext)

{

var session = GetCurrentSession();

if (session != null)

{

if (session.Transaction.IsActive)

{

if (filterContext.Exception == null)

{

session.Flush();

session.Transaction.Commit();

}

else

{

session.Transaction.Rollback();

}

}

}

}

private ISession GetCurrentSession()

{

var container = GetContainer();

var sessionFactory = container.Get<ISessionFactory>();

var session = sessionFactory.GetCurrentSession();

return session;

}

private IKernel GetContainer()

{

var resolver = GlobalConfiguration.Configuration.DependencyResolver as NinjectDependencyResolver;

if (resolver != null)

{

return resolver.Container;

}

throw new InvalidOperationException("NinjectDependencyResolver not being used as the MVC dependency resolver");

}

To begin a new transaction, you first get the current ISession object (which, thanks to the ISession management code covered in the previous section, has already been created and is accessible via the ISessionFactory object). You then use that ISession object to begin a new transaction. Pretty simple.

After the controller action is executed, the OnActionExecuted() override calls EndTransaction(). This method starts by obtaining a reference to the current ISession object. It then checks to make sure there is an active transaction—as you don’t want to try to commit or rollback a non-existent transaction. If there is an active transaction, then you want to do one of two things: commit it or roll it back. This is dependent on whether an exception occurred somewhere in the execution of the controller action.

You can use the filterContext.Exception property for this check. If an exception doesn’t exist, you flush the session and commit the transaction. However, if an exception does exist, you want to roll back the active transaction.

# Summary

That’s it for this chapter. You’ve finally learned about all of the key aspects of MVC Web API controllers and their dependencies, NHibernate configuration, and ISession and database transaction management.

To recap how all of this comes together at runtime, let’s outline their usage with some pseudo code:

Caller makes a web request.

MVC starts activation of the appropriate controller, based on the URL routes registered at application start-up.

MVC uses the NinjectDependencyResolver to satisfy all of the dependencies of controller, all of the dependencies each dependency requires, and so on.

If any object requires an ISession object, Ninject calls the NinjectConfigurator.CreateSession() method to create the ISession instance.

The CreateSession() method opens a new session and binds it to the web context, so that it will be available for subsequent ISession requests.

MVC calls the custom attribute’s OnActionExecuting() override, which in turn starts a new database transaction.

MVC calls the controller method, which presumably uses dependencies that were injected in during controller activation.

MVC calls the custom attribute’s OnActionExecuted() method, which first ends (either commits or rolls back) the database transaction, and then closes and disposes of the current ISession object.

Phew! That’s a lot going on for every web request! I hope you see, though, that each responsibility has been separated into small, unique classes. And most importantly, the controller and repository code don’t have to worry at all about the lifetime management of ISession objects or database transactions, nor do they need to be worried about making database calls.

In the next chapter, you will complete the exploration of “framework things” by examining logging and security. And you’ll see again that you can easily pull these concerns into their own classes and wire them up to happen automatically on every web request.