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

\*\*\* End of new material and edits for 2nd edition \*\*\*

If your controllers are going to do anything useful, they will need to use functionality brought in from other classes. An obvious example of this would be a database repository—that is, an object from which you can query the database and also save changes back to the database. Another example might be an object that represents the current user context, from which you can grab the user’s name, email address, and so on. These are considered dependencies of your controller class. That is, your controller depends on them for functionality and behavior not implemented within the controller itself. And if you’re following the Single Responsibility Principle, your controller class won’t be doing much of anything, which means it should require at least a few external dependencies. In short, if your controller needs to do anything that isn’t simply responding to a web request, then that behavior should be pushed off to a separate class and used by the controller (through composition—not inheritance!). The following list illustrates some of the kinds of behaviors that would be used by a controller (in a general sense, not necessarily in the task-management service):

* Database repository
* Financial calculators
* Transaction posting
* DateTime adapter
* File adapter
* Environment adapter
* User management class
* Validation classes
* Logger
* Cache management
* Exception wrappers

Again, 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 remember the overall pattern is with these two separate steps:

1. Push all dependencies up to the constructor.
2. 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 novice. Think of it this way: a class should not use any behavior that does not come through the constructor. None. Zero. Zip. 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. It also includes the types of behaviors listed in the previous section. If your class is using anything that isn’t implemented in that class, it needs to be injected in through the constructor. Period. You even want to avoid the use of static properties and methods. If your code needs to use the static DateTime.Now property, for example, you should wrap it in an injectable adapter class. You can find an example of this in this book’s sample code (and also a bit later in this chapter in the DateTimeAdapter class).

Suppose that in the TasksController’s Post() method you want to know the current date and time, so that you can set the CreatedDate property on a newly created task. Rather than coupling yourself to the .NET DateTime class directly, you’re going to use DI to inject in a DateTime adapter. This is what the constructor code (and corresponding private field) might look like:

private readonly IDateTime \_dateTime;

public TasksController(IDateTime dateTime)

{

\_dateTime = dateTime;

}

The IDateTime interface, and corresponding default implementation that gets injected into the constructor (you’ll look at the DI configuration in a minute), look like this:

public interface IDateTime

{

DateTime UtcNow { get; }

}

public class DateTimeAdapter : IDateTime

{

public DateTime UtcNow

{

get { return DateTime.UtcNow; }

}

}

Somewhere in the controller code, you can then use the private \_dateTime field to grab the current date and time of the system:

var task = new Task

{

// …

CreatedDate = \_dateTime.UtcNow

};

You simply follow this same pattern for all other dependencies, even for other .NET Framework classes. Another classic example is the use of Environment.MachineName. This should also be wrapped in an adapter and injected in through the constructor. Not only do you decouple the code from something outside of the application (in this case, the name of the machine you’re on), but it also provides for much better testability. This is because you can now use a Mock during unit test execution and explicitly set the machine name property, as opposed to relying on the name of whatever machine the tests happen to be running on.

To summarize this section, remember that all functionality used by a class must be pushed up to the constructor as a dependency. Further, those dependencies should come in the form of an interface, whether you write it yourself (e.g., an adapter) or there is one already available for you to use (e.g., a log4net ILog interface).

Now that you’ve established the need and basic use of this very important pattern, let’s look at how you might configure a DI tool to give you what you need at runtime.

## Configuring Ninject Dependency Injection

For the task-management service, I’ve chosen to use the open source Ninject dependency injection tool. Ninject is easy to use and seems to be the most popular approach these days (within the .NET community). However, the same principles apply to all DI tools—you just need to account for the differences in syntax.

There are three things you 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 MVC 4 Web API service. Table 5-2 briefly describes each of these three activities.

Table 5-2. Three 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. |
| Container bindings | This is where you link interfaces to concrete implementations, such as IDateTime⮞DateTimeAdapter. |
| IDependencyResolver for Ninject | This tells MVC 4 / Web API to ask for all dependencies. 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 (as well as 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 three criteria:

* Be created as one of the first things that happens when the application starts up
* 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 simply install the Ninject.Web.Common NuGet package. If you were following the steps in Chapter 4, then you already did this. It handles creating and destroying a container instance—within those startup and shutdown methods. All of this happens within a class that gets added to the app\_start folder in the MVC project.

The bottom line: Installing the Ninject.Web.Common NuGet package is all you need to do to properly configure Ninject to work in an MVC 4 and Web API project. Pretty easy!

## Container Bindings

Once the container itself is configured to be around while the application is running, you need to give it the type mappings. This is essentially just mapping interface types to implementation types—and in some cases, to implementation methods. In the previous example, where you looked at how the IDateTime interface is injected into the TasksController, the implementation would be the DateTimeAdapter class. This particular mapping uses Ninject and would look like this:

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

Notice that you aren’t actually creating an instance of DateTimeAdapter—you’ll let Ninject do that for you as such instances are required.

You also need to map the log4net interface ILog to a specific logger. In this case, you’ll create an instance of the logger and tell Ninject to use it only—and not create its own instances. This is because the log4net logger must be created a certain way using the log4net LogManager class. Nevertheless, registering an instance is similar to registering a type mapping, as shown here:

log4net.Config.XmlConfigurator.Configure();

var loggerForWebSite = LogManager.GetLogger("Mvc4ServicesBookWebsite");

container.Bind<ILog>().ToConstant(loggerForWebSite);

Assuming that you used the log4net web.config section specified in Chapter 4, the first line (where you call the log4net Configuration object’s Configure() method) will read that configuration information and use it to wire up the loggers and their properties. Next, you can use the LogManager to get an instance of a logger—which you immediately stick into the Ninject container. Therefore all subsequent calls to container.Get<ILog>()—or, all constructor-injected ILog arguments—will use the loggerForWebSite instance. The main difference is that you are mapping the ILoginterface to the logger object you created. You do this using the ToConstant() method on the Ninject container.

You can make all of these mappings from within the NinjectWebCommon class that the Ninject.Web.Common package adds to the app\_start folder. The package installer creates a static method called RegisterServices() that takes the current container (called an IKernel in Ninject).You can use this method to make similar type mappings to those just shown. The code for the task-management service that goes along with this book includes a separate class to handle all of the type mapping. This code simply calls that class from within the stubbed RegisterServices() method. That class is called NinjectConfigurator, and it is shown in the next code snippet, with most of the mappings already present. Review the code to see how the Configure() method takes an IKernel, and then uses that to register all of the type mappings. And remember that this is called from the RegisterServices() method, which is called during application start-up. In short, all of these mappings are created during start-up before any of the controller methods are ever executed. As you’ll see in the next section, this is important because the controllers actually need these objects to be injected into them when they are activated by MVC.

Here’s the code for most of the Ninject mappings being done in the NinjectConfigurator class:

///<summary>

/// Class used to set up the Ninject DI container.

///</summary>

public class NinjectConfigurator

{

///<summary>

/// Entry method used by caller to configure the given

/// container with all of this application's

/// dependencies. Also configures the container as this

/// application's dependency resolver.

///</summary>

public void Configure(IKernel container)

{

// Add all bindings/dependencies

AddBindings(container);

// Use the container and the NinjectDependencyResolver as

// application's resolver

var resolver = new NinjectDependencyResolver(container);

GlobalConfiguration.Configuration.DependencyResolver = resolver;

}

///<summary>

/// Add all bindings/dependencies to the container

///</summary>

private void AddBindings(IKernel container)

{

ConfigureNHibernate(container);

ConfigureLog4net(container);

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

container.Bind<IDatabaseValueParser>().To<DatabaseValueParser>();

container.Bind<IHttpCategoryFetcher>().To<HttpCategoryFetcher>();

container.Bind<IHttpPriorityFetcher>().To<HttpPriorityFetcher>();

container.Bind<IHttpStatusFetcher>().To<HttpStatusFetcher>();

container.Bind<IHttpUserFetcher>().To<HttpUserFetcher>();

container.Bind<IHttpTaskFetcher>().To<HttpTaskFetcher>();

container.Bind<IUserManager>().To<UserManager>();

container.Bind<IMembershipAdapter>().To<MembershipAdapter>();

container.Bind<ICategoryMapper>().To<CategoryMapper>();

container.Bind<IPriorityMapper>().To<PriorityMapper>();

container.Bind<IStatusMapper>().To<StatusMapper>();

container.Bind<IUserMapper>().To<UserMapper>();

container.Bind<ISqlCommandFactory>().To<SqlCommandFactory>();

container.Bind<IUserRepository>().To<UserRepository>();

container.Bind<IUserSession>().ToMethod(CreateUserSession).InRequestScope();

}

///<summary>

/// Set up log4net for this application, including putting it in the

/// given container.

///</summary>

private void ConfigureLog4net(IKernel container)

{

log4net.Config.XmlConfigurator.Configure();

var loggerForWebSite = LogManager.GetLogger("Mvc4ServicesBookWebsite");

container.Bind<ILog>().ToConstant(loggerForWebSite);

}

///<summary>

/// Used to fetch the current thread's principal as

/// an <see cref="IUserSession"/> object.

///</summary>

private IUserSession CreateUserSession(IContext arg)

{

return new UserSession(Thread.CurrentPrincipal as GenericPrincipal);

}

///<summary>

/// Sets up NHibernate, and adds an ISessionFactory to the given

/// container.

///</summary>

private void ConfigureNHibernate(IKernel container)

{

// Build the NHibernate ISessionFactory object

var sessionFactory = FluentNHibernate

.Cfg.Fluently.Configure()

.Database(

MsSqlConfiguration.MsSql2008.ConnectionString(

c => c.FromConnectionStringWithKey("Mvc4ServicesDb")))

.CurrentSessionContext("web")

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

.BuildSessionFactory();

// Add the ISessionFactory instance to the container

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

// Configure a resolver method to be used for creating ISession objects

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

}

///<summary>

/// Method used to create instances of ISession objects

/// and bind them to the HTTP context.

///</summary>

private ISession CreateSession(IContext context)

{

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

if (!CurrentSessionContext.HasBind(sessionFactory))

{

// Open new ISession and bind it to the current session context

var session = sessionFactory.OpenSession();

CurrentSessionContext.Bind(session);

}

return sessionFactory.GetCurrentSession();

}

}

As you build the code for the task-management service—and as the code evolves over time—you will find yourself coming back to this NinjectConfigurator class over and over. This is because the classes you use 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 massage them as time goes on.

Now take a look at the third line in the Configure() method. This is where the code creates the dependency resolver that MVC will use to resolve all dependencies.

## IDependencyResolver for Ninject

So far you’ve configured two things with the container:

* You configured a Ninject container instance to be available to the application during its entire lifetime. You did this by installing the Ninject.Web.Common NuGet package into the MVC project.
* You registered all of the type mappings (at least, the ones you know about so far) in the Ninject container instance with the NinjectConfigurator class that is used from the NinjectWebCommon.RegisterServices() method.

Finally, you need to tell MVC to actually use this container instance when activating the controllers. This involves two main steps:

1. Create an implementation of IDependencyResolverin which you use the Ninject container to resolve the requested dependencies.
2. Register an instance of the IDependencyResolver implementation with MVC.

Here is the NinjectDependencyResolver you will find in the example code for this book. Note that it takes an instance of a Ninject container in its constructor

public class NinjectDependencyResolver : IDependencyResolver

{

private readonly IKernel \_container;

public IKernel Container

{

get { return \_container; }

}

public NinjectDependencyResolver(IKernel container)

{

\_container = 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()

{

// noop

}

}

The methods to note are the GetService() and GetServices(). All you’re really doing is using the Ninject container to get object instances for the requested service types. Note that, in the GetService() method, you are using the TryGet() method instead of the Get() method. This is because the NinjectDependencyResolver class will be used to resolve all dependencies, not just ones that you know about. Internally, the MVC Framework will be looking for other dependencies it needs (i.e., dependencies you don’t even know about). Thus, you need to make sure you don’t blow up if you’re asked for something you haven’t registered.

Once you have the IDependencyResolver class, you just need to run the following code to register it with MVC (as shown previously in the NinjectConfigurator class):

var resolver = new NinjectDependencyResolver(container);

GlobalConfiguration.Configuration.DependencyResolver = resolver;

And that’s it! At this point MVC will hit the configured Ninject container instance to resolve any dependencies needed during controller activation.

# NHibernate Configuration and Mappings

It’s time to turn your attention to configuring NHibernate to work against the database and with the domain model classes. You’ll be using the Fluent NHibernate library for both, instead of relying on XML files.

## Database Configuration

As with any approach to data access, at some point you must tell the underlying framework how to connect to the database. And because NHibernate is database-provider agnostic, you must also tell it which provider you’re using—and even which version of which provider. This allows NHibernate to load the appropriate driver for communicating with the database and when dynamically generating the DML (Data Manipulation Language). 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 you can—in theory—change database providers without having to change anything about your domain model or any code that uses it.

You would, of course, need to update the NHibernate configuration and mapping definitions. It is for this reason that the data layer is separated into two projects in Visual Studio:

* One is called Data, and it is purely database-provider agnostic.
* One is specific to the provider.

The Data project includes the entire domain model (i.e., all of the classes), as well as the repository interfaces. Neither of these 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 second project, named similar to Data.SqlServer or Data.Oracle, will contain the repository implementations and the domain model NHibernate mapping definitions. This approach is used because both of these will likely change when swapping out database providers.

The database configuration is generally just a small bit of code located somewhere in the application’s start-up logic. Or, rather than in code, it can exist as XML in the application’s config file. That means it is very easy to update this configuration, should you decide to switch from SQL Server to Oracle.

Let’s take a look at the database configuration for the task-management service. Again, you’re using Fluent NHibernate to make this work (instead of XML configuration files). You can find the following code in the NinjectConfigurator class discussed previously:

var sessionFactory = FluentNHibernate

.Cfg.Fluently.Configure()

.Database(

MsSqlConfiguration.MsSql2008.ConnectionString(

c => c.FromConnectionStringWithKey("Mvc4ServicesDb"))

.CurrentSessionContext("web")

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

.BuildSessionFactory();

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

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

In the preceding code, you set four properties, and then build the ISessionFactory object. Let’s take a closer look at those properties:

* The first property indicates that you are using SQL Server—and even that you’re using version 2008 (which is compatible with 2012).
* The second property specifies the database connection string and that you want it to load from the web.config file’s Mvc4ServicesDb connection string value.
* The third property tells NHibernate that you plan to use its web implementation to manage the current session object. You’ll explore session management more in the next section, but it essentially lets you scope 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. TheData.SqlServer project contains all of those mappings, so I just gave it a class name that exists in that assembly.

Finally, you call BuildSessionFactory(), which returns a fully configured NHibernate ISessionFactory instance. Next you put the ISessionFactory instance into the Ninject container with this statement:

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

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

## Model Mapping

Next, you need to provide all of the code that will map between the domain model classes (i.e., the Plain Old CLR Objects, or POCOs) and the database’s tables and columns. Depending on the database model you’re trying to map—and depending on much you are trying to abstract away the model itself—building these mappings can be anywhere from very simple to very complex. The task-management service will be on the very simple end of the scale to map. This is because the database has been modeled to be exactly what you want to expose in the service, so all of the tables and columns pretty much match what the service consumer will end up seeing. However, the mapping would be more involved if you were trying to build this REST service on some legacy database that had older style table and column names, and it wasn’t normalized the same way. Feel free to Google around and buy any one of several good NHibernate books if you need to go beyond the rather trivial mappings shown here. For a great getting-started guide on Fluent NHibernate, try reading the project’s Wiki at https://github.com/jagregory/fluent-nhibernate/wiki.

## The Mapping Classes

Since you’ve already gone through the model classes in Chapter 4, as well as the data model itself, these mapping definitions should be fairly self-explanatory. All of them are shown below, and I’ll point out a few key things following the code. Note that all of these mapping classes are located in the Mapping folder within the MVC4ServicesBook.Data.SqlServer project:

public class CategoryMap : VersionedClassMap<User>

{

public CategoryMap()

{

Id(x => x.CategoryId);

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

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

}

}

public class PriorityMap : VersionedClassMap<User>

{

public PriorityMap()

{

Id(x => x.PriorityId);

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

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

}

}

public class StatusMap : VersionedClassMap<User>

{

public StatusMap()

{

Id(x => x.StatusId);

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

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

}

}

public class TaskMap : VersionedClassMap<User>

{

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.DateCompleted).Nullable();

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

References(x => x.Priority, "PriorityId");

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

HasManyToMany(x => x.Users)

.Access.ReadOnlyPropertyThroughCamelCaseField(Prefix.Underscore)

.Table("TaskUser")

.ParentKeyColumn("TaskId")

.ChildKeyColumn("UserId");

HasManyToMany(x => x.Categories) .Access.ReadOnlyPropertyThroughCamelCaseField(Prefix.Underscore)

.Table("TaskCategory")

.ParentKeyColumn("TaskId")

.ChildKeyColumn("CategoryId");

}

}

public class UserMap : VersionedClassMap<User>

{

public UserMap()

{

Table("AllUsers");

Id(x => x.UserId).CustomType<Guid>();

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

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

Map(x => x.Email).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 allows you to automatically take advantage of NHibernate’s ability to check for dirty records in the database, based on a Rowversion column on each table. The definition of this base class looks like this:

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

{

protected VersionedClassMap()

{

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

}

}

That crazy-long statement can be broken down as follows:

* Use the Version property on each domain model class.
* The database column 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 just be null.

Again, all of this is to let NHibernate protect it (and you!) against trying to update dirty records (i.e., records that were updated by someone else after the data was read). Placing this statement in the constructor of the base class means it will automatically be executed by every ClassMap implementation in the Mapping folder. The IVersionedModelObject interface just ensures that the model class contains a Version property. Each of the model classes implements this interface.

The ClassMap<T> base class is defined in the Fluent NHibernate library, and it simply provides a means of configuring a model class’s mapping through code (as opposed to using XML files). You do this in the mapping class’s constructor. For example, the CategoryMap mapping class contains the mapping for the Category model class.

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 model class is used as the object identifier. This method also includes an overload to specify the table’s key column—if it happens to not match the property name given.

The Map() method is used in a similar fashion, although not for the object’s identifier. By default, NHibernate will assume the mapped column name is the same as the given property name. If it’s not, a different overload can be used to specify the column name. Additionally, because this is a fluent-style interface, you 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.

You should see one ClassMap<T> implementation for each model class.

## Project and File Organization

This could be confusing. To help visualize these classes, look at their file folders in Visual Studio (see Figures 5-1 and 5-2). Notice how the model classes are in the Data project, whereas the mapping classes are in the Data.SqlServer project.

Figure 5-1. The domain model classes

Figure 5-2. The NHibernate mapping classes

The Category, Priority, and Status mapping classes are pretty straightforward. They each have three properties mapped, the first of which is the class’s identifier. You don’t need to specify the column name because the column name happens to match the property name.

## Model Relationships

The TaskMap class is a little more complicated; it contains three many-to-one references and two many-to-many relationships. The many-to-one references are simple—the task has a reference to a Status, another reference to a Priority, and another reference to a User (to track who created the task).

The many-to-many relationships are a little 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 categories to a task, the TaskCategory table will contain a record for each category linked to that task. The categories will be loaded into the Categories collection property on the Task object.

Regarding collections, the Task class defines the Categories property like this:

private readonly IList<Category> \_categories = new List<Category>();

public virtual IList<Category> Categories

{

get { return \_categories; }

}

You define the property with only a getter, to prevent the developer from replacing the entire collection. You also want to create an empty collection upon class instantiation, which allows the developer to immediately call Add() on the property without having to create a new collection first. As such, the TaskMap class defines the Categories property map with this bit of code:

.Access.ReadOnlyPropertyThroughCamelCaseField(Prefix.Underscore)

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

Finally, note the use of the Table() function in the UserMap class. This is used because the name of the table is actually different from the name of the model class. In fact, in this particular case, the underlying table is actually a view. The Table() function tells NHibernate to use the AllUsers view when constructing SELECT, INSERT, UPDATE, and DELETE statements. But because AllUsers is actually a database view, you will only use NHibernate to fetch users. For saving user data, you will utilize a set of stored procedures.

You do this because, for the users in the system, you are supplementing ASP.NET Membership tables with your own, so it’s not a typical case of running just SELECT, INSERT, UPDATE, and DELETE statements on a single table. However, the rest of the model classes map directly to single tables, so NHibernate can be used for all SQL operations (including saving new or updated records).

Take a moment to recall the previous discussion on NHibernate configuration and the manner in which you tell NHibernate to find these mapping classes (i.e., the AddFromAssemblyOf<T>() method on the Fluent NHibernate configuration statement). NHibernate will look for and instantiate all ClassMap<T> implementations when the BuildSessionFactory() method is executed. At this point, the mapping code contained in the various constructors will run, as well.

That’s it for NHibernate database configuration and model mapping. Next, you're going to look at managing the NHibernate ISession object.

# 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.