CHAPTER 5

Up and Down the Stack with 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 set up 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 need to deal with some of the more complex infrastructural concerns in the task-management service and highlight several great ASP.NET Web API features. We'll cover the following topics:

- Routing (convention 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
- Error handling
- `IHttpActionResult`

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 make sense in the end. Now let's get started.

CODING CONVENTION

Unless otherwise noted, we implement one public type per file. The file name should match the type name, and the file location should match the namespace name.

For example, the `WebApi2Book.Web.Common.Routing.ApiVersionConstraint` class is in a file named "`ApiVersionConstraint.cs`," which is located in a project folder named "Routing" in the `WebApi2Book.Web.Common` project.

We will initially provide detailed instructions to help you implement the solution according to this coding convention. However, to avoid becoming excessively repetitive and tedious, we will gradually provide less detail, under the assumption that you will have become familiar with the pattern.

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 you'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-specific and action-specific filters, and result conversion (i.e., it fully utilizes the ASP.NET Web API's extensible processing pipeline). Leveraging routes and associated controllers 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.

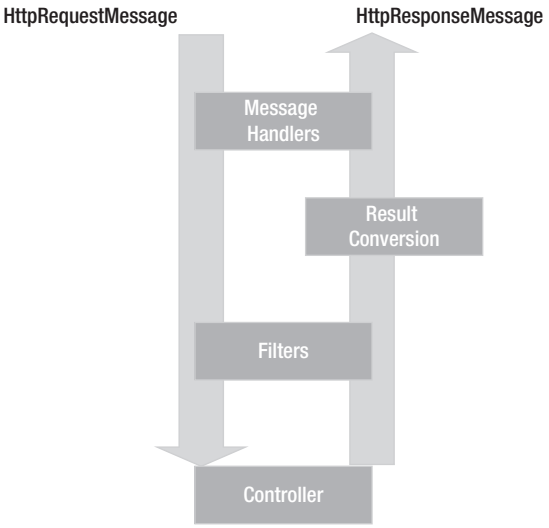

Figure 5-1. *ASP.NET Web API message processing pipeline*

Be sure to visit the official Microsoft site (`http://www.asp.net`) for this poster and many other useful resources.

When a service request comes over the network and into Internet Information Services (IIS), IIS routes it to a worker process that is hosting the ASP.NET runtime. Inside this process (or in an 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 (which 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 specific 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, 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 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 and associated conventions 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 trying to implement REST with ASP.NET MVC.

In addition to using arguments that come from the URL, you can add method arguments for data that arrives via the message body. Model binding, similar to that utilized with ASP.NET MVC, will be used to translate the incoming data into the appropriate model class. 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, as a matter of good design and clean code, you should be careful not to couple your controller 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 toward "convention over configuration," it just works! And as we discussed in Chapter 2, this lack of strict client-server contract is also a foundational principle with the RESTful architecture.

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 operations shown in Table 5-2.

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 the 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's data type. 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. By 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 2 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:

- First, the 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 you saw 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 the 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 titled "Attribute Routing in Web API 2" (`www.asp.net/web-api/overview/web-api-routing-and-actions/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 VERSUS API VERSIONING

Note that in this section we aren't talking about assembly or DLL versioning. The versioning of assemblies relates more to changes within the underlying implementation. Rather, we are interested here in the versioning of the public API (i.e., the interface). As such, changes to a DLL don't necessarily require a change to the API's version. But changes to URL, or breaking changes to a resource type, would require a new API version.

Currently within the software community, there are four basic approaches to versioning the RESTful way:

1. **URI Path** This approach takes the following form:

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

2. **URI Parameter** This approach takes the following form:

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

3. **Content Negotiation** This is done in the HTTP header.

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

4. **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 using 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 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". The API project should then look like Figure 5-2.

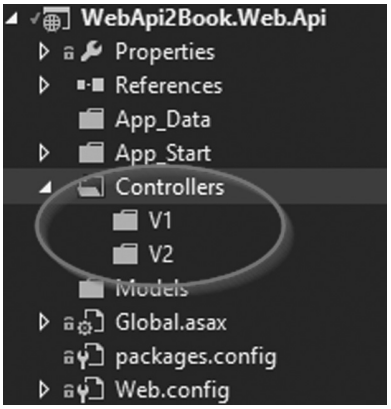

Figure 5-2. *API project with version-specific controller folders*

Add a new controller named `TasksController` to each folder using the empty Web API 2 controller template. (See Figures 5-3 and 5-4.)

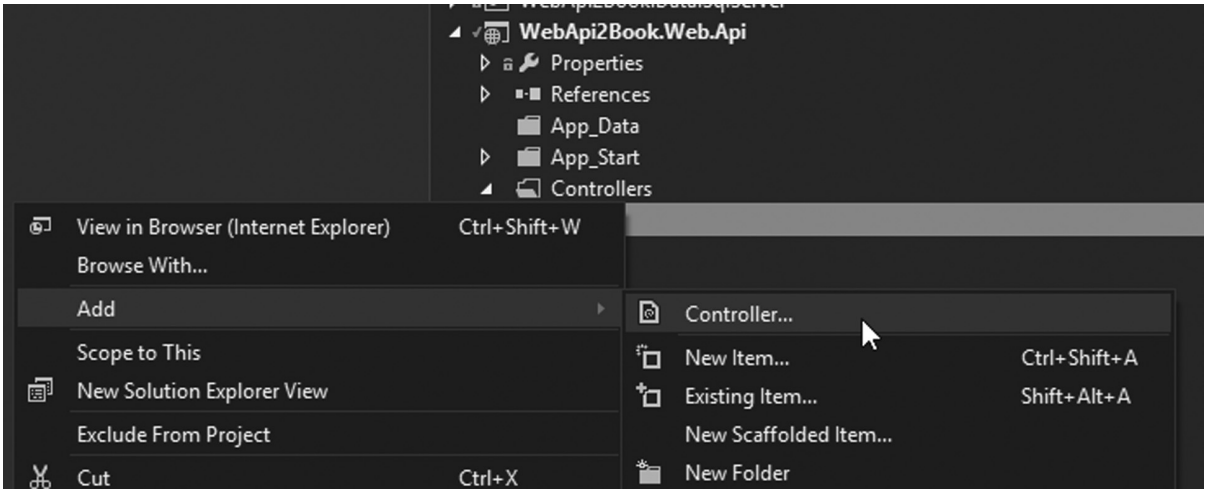

Figure 5-3. *Adding a controller*

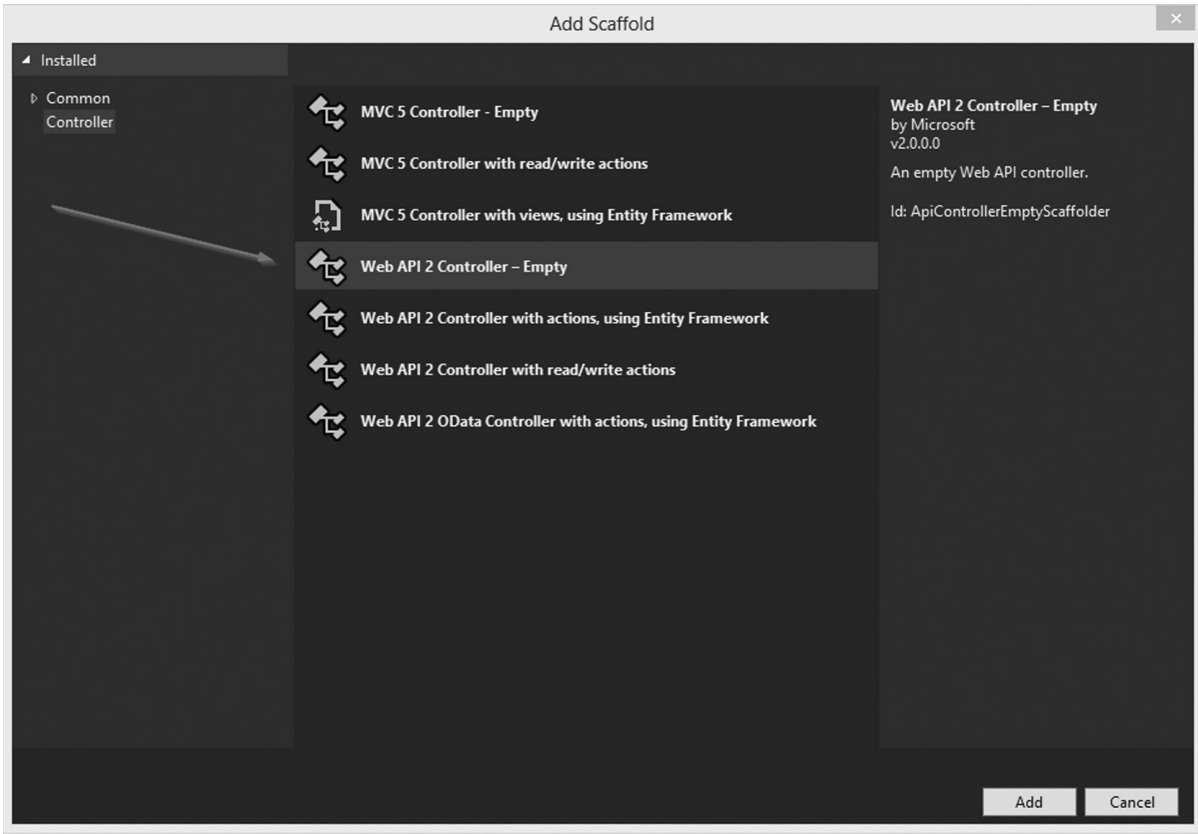

Figure 5-4. *Specifying the empty controller scaffold*

Now there will be two `TasksController` 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, and the V1 controller is the first match it finds. This is the case with both convention-based 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 that takes the namespace into account when looking for the matching controller class. First, let's deal with the constraint.

A Custom IHttpRouteConstraint

The first thing we need to do is add some dependencies to the `WebApi2Book.Web.Common` project. With the solution open, run the following commands in the Package Manager console:

```
install-package Microsoft.AspNet.WebApi WebApi2Book.Web.Common
```

Next, 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 over all of the controllers (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
    {
    }
}
```

Studying the `ApiVersion1RoutePrefixAttribute` class and the `TasksController` class, we can see that this `TasksController` 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 you learned in the "Attribute-Based Routing" section earlier in the chapter, you can now recognize that this `RoutePrefix` attribute is configured to match a URL path of `api/{apiVersion}/tasks`. You also can see that the `apiVersion` parameter is constrained by our custom `IHttpRouteConstraint` to a value of "v1". Again, using the custom `ApiVersion1RoutePrefix` attribute helps you avoid silly errors in routing caused by copy/paste/syntax mistakes.

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 implement 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 routing in any way. Also note that the two route names are unique between the two controllers. We need to ensure globally unique route names, as required by ASP.NET Web API.

OK, now we're almost ready to process a request. First, we'll implement the custom controller selector. (As we mentioned earlier, a custom controller selector is necessary because the framework only matches on the controller class name without regard to the controller class' namespace.) Then, after that, we will 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 "ASP.NET Web API: Using Namespaces to Version Web APIs" (`http://blogs.msdn.com/b/webdev/archive/2013/03/08/using-namespaces-to-version-web-apis.aspx`). 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 use of `IHttpRouteData` subroutes), but the basic design and implementation comes straight from CodePlex.

Anyway, go ahead and implement the custom controller selector as follows in the root of 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
{
    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've still got a long way to go, so 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, replacing the default implementation provided by Visual Studio):

```
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));
        }
    }
}
```

The first part of the `Register` method configures the version constraint. Our `ApiVersionConstraint` is registered with a constraint resolver, which the framework uses to find and instantiate the appropriate constraint at runtime. The last part of the method wires-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.

The Demo

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

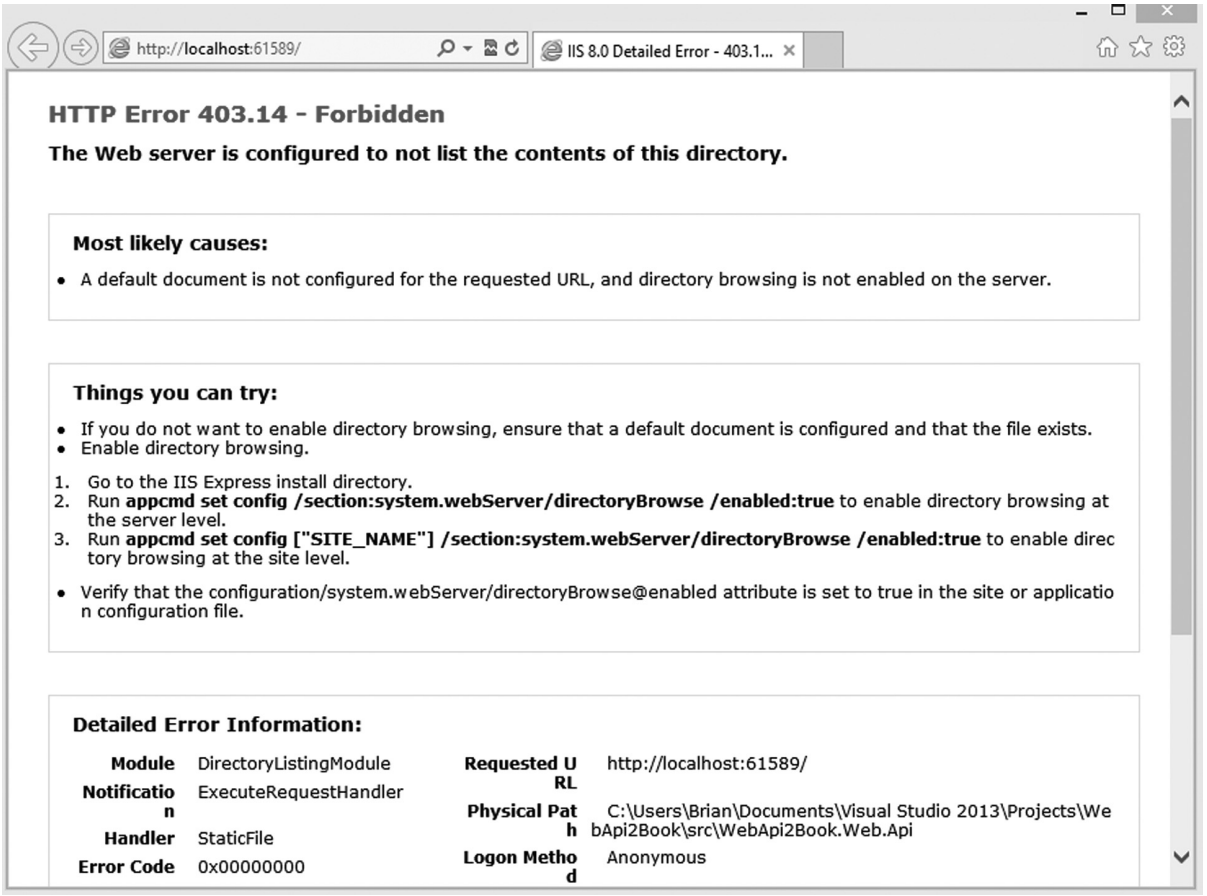

Figure 5-5. *Application error page*

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

Now we need to send an HTTP POST message via the V1 route, so go ahead and send the following request message using Fiddler (or your favorite web proxy debugging tool):

V1 Request

```
POST http://localhost:61589/api/v1/tasks HTTP/1.1
Content-Type: text/json

{"Subject":"Fix something important"}
```

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! You may also note that the `TaskId` is null, but that's because we're not actually persisting anything yet.

OK, 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. You learned about routing, including many of the new capabilities made available by ASP.NET Web API 2. You learned about constraints and controller selectors. And you successfully processed an HTTP request. However, we must admit that this is rather "hello world-ish"; our controller actions aren't doing anything meaningful. This leads us to our next topic: dependencies. Inside of these dependencies is where the real work gets done, at least from a business perspective. Anything related to data access, calculations, file I/O, date/time, etc., will all be handled by such dependencies.

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, they will depend heavily upon functionality provided by other classes. An example of this is a database accessor, which is an object that can be used to query and save changes back to the database. The database accessor is 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 the SOLID design principles, do yourself a huge favor and familiarize yourself with them. These principles were defined by Robert C. Martin in the early 2000s and have been reviewed and explained numerous times by quite a few people over the last decade.

The acronym SOLID stands for the following: Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, and Dependency inversion principle. You can read up on these five principles in the following related articles written by Bob Martin:

- S - `http://www.objectmentor.com/resources/articles/srp.pdf`
- O - `http://www.objectmentor.com/resources/articles/ocp.pdf`

- L - http://www.objectmentor.com/resources/articles/lsp.pdf
- I - http://www.objectmentor.com/resources/articles/isp.pdf
- D - http://www.objectmentor.com/resources/articles/dip.pdf

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 or all functionality, you need a pattern and appropriate tool(s) for configuring and obtaining those dependencies. The easiest way to approach this is summarized in these two points:

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

Constructor Injection of Dependencies

The concept of pushing 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.

To start, 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, 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, the `DateTime.Now` functionality should be wrapped in an injectable adapter class instead of used directly. There are exceptions to this "constructor injection mandate," such as when constructor injection isn't even possible (we'll see examples of this), but you get the point: use constructor injection wherever you can.

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. Of particular importance is the way in which controllers are activated by the runtime. All API requests will revolve around a controller method. As such, configuring our controllers to obtain their dependencies through their respective constructors provides the "root" we need to ensure all objects are managed properly.

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	Implementation	Description
Container configuration	`NinjectWebCommon`	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	NinjectConfigurator	This is where we bind or relate interfaces to concrete implementations so that the dependencies can be resolved at run time. For example, if a class requires an `IDateTime` object, the bindings tell the container to provide a `DateTimeAdapter` object.
`Dependency resolver` for Ninject	`NinjectDependencyResolver`	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. Without this resolver, ASP.NET won't use your configured Ninject container for dependencies.

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, you already did this. The package generates code that handles creating and destroying a container instance within the `Start` and `Stop` 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 percent of the way there by hooking into the ASP.NET application's startup and shutdown events. We'll take a look at it soon.

Container Bindings

Now that the container itself is configured to be available 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. This is an example of letting the container (and its associated configuration) manage the lifetime of an application's objects, thereby removing that burden from consumers of those objects.

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. We also sometimes use the `ToConstant` lifetime scope, which manages an application-level singleton instance that our code—not Ninject—has instantiated. And while we're on the subject of configuring dependencies with Ninject, we should mention that we occasionally use the `ToMethod` factory method to specify a method or delegate for Ninject to call whenever a new object of a given interface/abstract type is needed.

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:

IDateTime Interface

```
using System;

namespace WebApi2Book.Common
{
    public interface IDateTime
    {
        DateTime UtcNow { get; }
    }
}
```

DateTimeAdapter Class

```
using System;

namespace WebApi2Book.Common
{
    public class DateTimeAdapter : IDateTime
    {
        public DateTime UtcNow
        {
            get { return DateTime.UtcNow; }
        }
    }
}
```

Next, anticipating the need for logging (doesn't every significant application 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 log4net `LogManager.GetLogger` method. Remember, we want to push dependencies up to the controller, and not rely on specific static properties or methods.

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 call to `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. Go ahead and implement it in the root of the `WebApi2Book.Web.Common` project:

```
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);
        }
    }
}
```

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. We simply want to return null if we haven't explicitly registered a given type.

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, here are 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 our 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 the 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 the application accepts and processes any HTTP requests, and thus 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` directives outside of the namespace. There's no particular reason for this other than personal preference.

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 this class 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, or "entity," classes. We'll be using the Fluent NHibernate NuGet library that we installed in Chapter 4 for this.

To reiterate what we first mentioned in the previous chapter, we will continue to refer to the domain model classes as "entities" to more easily distinguish between the persistent domain model types (i.e., the entities) and the service model types.

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—i.e., SELECT, INSERT, UPDATE, and DELETE statements. 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 code is dependent on a specific database provider; that is, the entities will be the same whether you're working with SQL Server or Oracle.
- The `WebApi2Book.Data.SqlServer` project, which contains the NHibernate mapping definitions. 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 created in the previous chapter). 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.

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 after looking at the code; speaking of which, go ahead and add 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 as a concurrency (or, version) value.
- 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 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. For example, the Task class has a Status property that references an instance of a Status class. We then 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." That's right, NHibernate will use reflection to access the collection of users via your _users private field—as opposed to the public getter.

Database Configuration: Bringing It All Together

OK, now that we've laid the groundwork for it, let's finish off this section by hooking NHibernate up to the ASP.NET Web API. First, add the following using directives and methods, respectively, to the NinjectConfigurator class we discussed previously.

Additional Using Directives

```
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHibernate;
using NHibernate.Context;
using Ninject.Activation;
using Ninject.Web.Common;
using WebApi2Book.Data.SqlServer.Mapping;
```

Additional Methods

```
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();
}
```

Then modify the `AddBindings` method so that it is implemented as shown (adding the call to `ConfigureNHibernate`):

```
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);
```

Note that we've placed our newly created `ISessionFactory` in our container as a constant. When using NHibernate, it is important to only ever create a single `ISessionFactory` instance per application. The act of creating the factory is pretty compute intensive, as it is creating all mappings to the database. Further, we only need one, as opposed to one per request, per user, etc.

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 of using NHibernate is that it implements a unit of work with its `ISession` interface. In a service application like the task-management service, we need the database session object—as well as an associated database transaction—to span a complete service call. This provides support for three very important requirements for interacting with persistent data within a web request:

- Keep fetched domain objects in memory so that they are consistent across all operations within a single web request. NHibernate uses the `ISession` object to keep all fetched data and associated changes in memory. If some code used a different `ISession` instance, it would not see the same/updated data.
- 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.

In short, it is very important to ensure that every database operation within a given web request uses the same `ISession` object. This is what we mean 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, when creating the `ISessionFactory` object, we began leveraging this ability by using the `CurrentSessionContext("web")` call to tell NHibernate to use its web implementation (which relies on the ASP.NET `HttpContext`) to manage the current session object. In addition to that, though, we need to use a special class within NHibernate called the `CurrentSessionContext`. We 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 feels like a lot of code. But you only have to do this in one place, and you'll find that it works incredibly well.

It'll be easier to understand if we look at the code, so refer back to the NinjectConfigurator's `ConfigureNHibernate` method we implemented in the last section. We configured the `ISession` mapping with Ninject like this:

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

This tells Ninject to call the `CreateSession` method whenever an object (e.g., a controller) needs an `ISession` injected into its constructor. Now let's look at the `CreateSession` method.

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

This `CreateSession` method will be executed every time any dependent object requests an `ISession` object via Ninject (e.g., through constructor injection). Our implementation ensures that, for a single API request, we only ever create one `ISession` object.

At this point, we have code in place to create and manage a single database session for a given request. To then close and dispose of this `ISession` object, we're going to use an implementation of an `ActionFilterAttribute`. ASP.NET Web API uses these attributes, and derivations thereof, to execute pre and post behaviors around controller methods. Our attribute will decorate the controllers to ensure all controller actions are using a properly managed `ISession` instance. The attribute, and its `IActionTransactionHelper and WebContainerManager` dependencies, are implemented as follows. Go ahead and add these to the `WebApi2Book.Web.Common` project, and then we'll discuss them:

WebContainerManager Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;
using System.Web.Http.Dependencies;

namespace WebApi2Book.Web.Common
{
    public static class WebContainerManager
    {
        public static IDependencyResolver GetDependencyResolver()
        {
            var dependencyResolver = GlobalConfiguration.Configuration.DependencyResolver;
            if (dependencyResolver != null)
            {
                return dependencyResolver;
            }

            throw new InvalidOperationException("The dependency resolver has not been set.");
        }

        public static T Get<T>()
        {
            var service = GetDependencyResolver().GetService(typeof (T));

            if (service == null)
                throw new NullReferenceException(string.Format(
                    "Requested service of type {0}, but null was found.",
                    typeof (T).FullName));

            return (T) service;
        }
```

```
        public static IEnumerable<T> GetAll<T>()
        {
            var services = GetDependencyResolver().GetServices(typeof (T)).ToList();

            if (!services.Any())
                throw new NullReferenceException(string.Format(
                    "Requested services of type {0}, but none were found.",
                    typeof (T).FullName));

            return services.Cast<T>();
        }
    }
}
```

IActionTransactionHelper Interface

```
using System.Web.Http.Filters;

namespace WebApi2Book.Web.Common
{
    public interface IActionTransactionHelper
    {
        void BeginTransaction();
        void EndTransaction(HttpActionExecutedContext filterContext);
        void CloseSession();
    }
}
```

ActionTransactionHelper Class

```
using System.Web.Http.Filters;
using NHibernate;
using NHibernate.Context;

namespace WebApi2Book.Web.Common
{
    public class ActionTransactionHelper : IActionTransactionHelper
    {
        private readonly ISessionFactory _sessionFactory;

        public ActionTransactionHelper(ISessionFactory sessionFactory)
        {
            _sessionFactory = sessionFactory;
        }

        public bool TransactionHandled { get; private set; }

        public bool SessionClosed { get; private set; }
```

```
        public void BeginTransaction()
        {
            if (!CurrentSessionContext.HasBind(_sessionFactory)) return;

            var session = _sessionFactory.GetCurrentSession();
            if (session != null)
            {
                session.BeginTransaction();
            }
        }

        public void EndTransaction(HttpActionExecutedContext filterContext)
        {
            if (!CurrentSessionContext.HasBind(_sessionFactory)) return;

            var session = _sessionFactory.GetCurrentSession();

            if (session == null) return;
            if (!session.Transaction.IsActive) return;

            if (filterContext.Exception == null)
            {
                session.Flush();
                session.Transaction.Commit();
            }
            else
            {
                session.Transaction.Rollback();
            }

            TransactionHandled = true;
        }

        public void CloseSession()
        {
            if (!CurrentSessionContext.HasBind(_sessionFactory)) return;

            var session = _sessionFactory.GetCurrentSession();
            session.Close();
            session.Dispose();
            CurrentSessionContext.Unbind(_sessionFactory);
            SessionClosed = true;
        }
    }
}
```

UnitOfWorkActionFilterAttribute Class

```
using System.Web.Http.Controllers;
using System.Web.Http.Filters;

namespace WebApi2Book.Web.Common
{
    public class UnitOfWorkActionFilterAttribute : ActionFilterAttribute
    {
        public virtual IActionTransactionHelper ActionTransactionHelper
        {
            get { return WebContainerManager.Get<IActionTransactionHelper>(); }
        }

        public override bool AllowMultiple
        {
            get { return false; }
        }

        public override void OnActionExecuting(HttpActionContext actionContext)
        {
            ActionTransactionHelper.BeginTransaction();
        }

        public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext)
        {
            ActionTransactionHelper.EndTransaction(actionExecutedContext);
            ActionTransactionHelper.CloseSession();
        }
    }
}
```

And finally, add the following to the bottom of the NinjectConfigurator's `ConfigureNHibernate` method so that `IActionTransactionHelper` instances can be resolved at run time:

```
            container.Bind<IActionTransactionHelper>().To<ActionTransactionHelper>().
InRequestScope();
```

■ **Note** You'll also need to add a `using` directive for `WebApi2Book.Web.Common`.

We'll explore transaction control in the next section, so we won't discuss anything related to that quite yet. Instead, let's look at the attribute's `CloseSession` method that is called from the `OnActionExecuted` override. As you can see, the meaningful part of its implementation is in the `ActionTransactionHelper` class, so let's dive down into that. Once there, note that most of the `CloseSession` method implementation is similar to the `NinjectConfigurator` class' `CreateSession` method we just discussed. It first checks to see if an `ISession` object is currently bound to the `CurrentSessionContext`. If so, the method obtains the `ISession` object with the `GetCurrentSession` method, and then closes and disposes of it. Finally, it unbinds 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 that they, or the controllers themselves, are decorated with this attribute. We'll apply the attribute at the controller level because we want each action method to participate in a unit of work. So go ahead and add a `UnitOfWorkActionFilterAttribute` to each controller, like this for the V1 `TasksController`:

```
[ApiVersion1RoutePrefix("tasks")]
[UnitOfWorkActionFilter]
public class TasksController : ApiController
```

■ **Note** You'll also need to add a `using` directive for `WebApi2Book.Web.Common`.

Great! We've implemented the unit of work. Overall, this implementation provides a nice separation of concerns: the dependent code can use the injected `ISession` and simply assume it is active and being managed by something outside itself. It 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.

Before we conclude this section, let's circle back and discuss some of the other things—not related to transaction control—that are going on in these classes that we just added:

- The `ActionTransactionHelper` property in `UnitOfWorkActionFilterAttribute` provides access to the `IActionTransactionHelper` dependency. This is necessary because constructor injection isn't possible with an attribute. The runtime creates only a single instance of the attribute for an application, as opposed to one per web request. So we can't store any state or per-request objects.
- We overrode the `AllowMultiple` method in the attribute to prevent the filter from executing multiple times on the same call. (~~See http://stackoverflow.com/questions/18485479/webapi-filter-is-calling-twice?rq=1.~~)
- We added the `WebContainerManager` to provide access to dependencies managed by the `IDependencyResolver`. This is needed to serve areas in code where the resolver cannot automatically reach, such as in an attribute constructor.

Database Transaction Control

The last thing database-related to cover in this chapter is transaction control. Because we have a single `ISession` instance spanning all operations within a single web request, we also want to wrap all operations within a single database transaction by default.

To make this happen, we're using the custom attribute implemented in the previous section. We overrode both the `OnActionExecuting` and `OnActionExecuted` methods. These are called by ASP.NET Web API before and after the controller action method is executed, respectively. The interesting code, though, is in the `ActionTransactionHelper` class' `BeginTransaction` and `EndTransaction` methods, so let's study those.

In `BeginTransaction`, we first get the current `ISession` object (if it's available). Thanks to the `ISession` management code covered in the previous section, it has already been created on-demand and is accessible via the `ISessionFactory` object. We then use that `ISession` object to begin a new transaction. Pretty simple.

The `EndTransaction` method starts by obtaining a reference to the current `ISession` object, if it's available. It then checks to make sure there is an active transaction, because we don't want to try to commit or roll back a nonexistent transaction. If there is an active transaction, we 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.

We use the `filterContext.Exception` property for this check. If an exception doesn't exist, we flush the session (which forces NHibernate to write all in-memory model changes to the database) and commit the transaction. However, if an exception does exist, we roll back the active transaction.

But why do we have a check for an active session at the start of each `ActionTransactionHelper` method? Shouldn't we always expect an ISession to be available, especially after we've gone to the trouble of adding the `UnitOfWorkActionFilterAttribute` to our controllers? Why even bother with this `CurrentSessionContext.HasBind` test? Well, take a look at our two controllers. At this point, neither requires an `ISession`. And because neither requires an `ISession` instance, the framework will never hit our resolver to create one. Therefore, if you were to remove these checks and run the demo that we ran earlier in the "Implementing POST" section, you'd encounter an exception…and we don't like those.

At this point, we have configured everything our task-management service needs in terms of persistence, yet our controller implementations have not advanced beyond a persistence-free, "hello world-ish" state. Hang in there, we'll get to them shortly (really!). First we need to attend to a few more infrastructural items.

Diagnostic Tracing

When you are trying to troubleshoot—or gain deep knowledge of the inner workings of—an application, there is no substitute for a good set of trace logs. Fortunately, enabling tracing in ASP.NET Web API 2 is painless.

Start by installing the tracing package using the NuGet Package Manager console as follows:

```
install-package Microsoft.AspNet.WebApi.Tracing WebApi2Book.Web.Api
```

Then add the following code to the bottom of the `WebApiConfig` class' `Register` method:

```
config.EnableSystemDiagnosticsTracing();
```

And that's it. This code adds the `SystemDiagnosticsTraceWriter` class to the Web API pipeline. Because the `SystemDiagnosticsTraceWriter` class writes traces to `System.Diagnostics.Trace`, if you run the demo from the "Implementing POST" section you'll see trace statements written to Visual Studio's Output window. You can also register additional trace listeners; for example, to write traces to the Windows Event Log, a database, or a text file.

Though we could end this section now (after all, we've accomplished our goal of adding tracing), we want to take things a step further and demonstrate how to implement and register a custom trace writer that plugs into the ASP.NET Web API tracing infrastructure and writes to log4net.

First, we need to provide an `ITraceWriter` implementation. Implement the following `SimpleTraceWriter` class in the root of the `WebApi2Book.Web.Common` project:

SimpleTraceWriter Class

```
using System;
using System.Net.Http;
using System.Web.Http.Tracing;
using log4net;
using WebApi2Book.Common.Logging;

namespace WebApi2Book.Web.Common
{
    public class SimpleTraceWriter : ITraceWriter
    {
        private readonly ILog _log;
```

```
        public SimpleTraceWriter(ILogManager logManager)
        {
            _log = logManager.GetLog(typeof (SimpleTraceWriter));
        }

        public void Trace(HttpRequestMessage request, string category, TraceLevel level,
            Action<TraceRecord> traceAction)
        {
            var rec = new TraceRecord(request, category, level);
            traceAction(rec);
            WriteTrace(rec);
        }

        public void WriteTrace(TraceRecord rec)
        {
            const string traceFormat =
                "RequestId={0};{1}Kind={2};{3}Status={4};{5}Operation={6};{7}Operator={8};{9}
Category={10}{11}
                    Request={12}{13}Message={14}";

            var args = new object[]
            {
                rec.RequestId,
                Environment.NewLine,
                rec.Kind,
                Environment.NewLine,
                rec.Status,
                Environment.NewLine,
                rec.Operation,
                Environment.NewLine,
                rec.Operator,
                Environment.NewLine,
                rec.Category,
                Environment.NewLine,
                rec.Request,
                Environment.NewLine,
                rec.Message
            };

            switch (rec.Level)
            {
                case TraceLevel.Debug:
                    _log.DebugFormat(traceFormat, args);
                    break;
                case TraceLevel.Info:
                    _log.InfoFormat(traceFormat, args);
                    break;
                case TraceLevel.Warn:
                    _log.WarnFormat(traceFormat, args);
                    break;
```

```
                case TraceLevel.Error:
                    _log.ErrorFormat(traceFormat, args);
                    break;
                case TraceLevel.Fatal:
                    _log.FatalFormat(traceFormat, args);
                    break;
            }
        }
    }
}
```

Finally, add the following code to the bottom of the `WebApiConfig` class' `Register` method to register the writer with the framework:

```
//config.EnableSystemDiagnosticsTracing(); // replaced by custom writer
config.Services.Replace(typeof(ITraceWriter),
    new SimpleTraceWriter(WebContainerManager.Get<ILogManager>()));
```

■ **Note** You'll need to add the following `using` directives:

- `using System.Web.Http.Tracing;`
- `using WebApi2Book.Common.Logging;`

See, wasn't that easy? Now, next time you run the demo from the "Implementing POST" section, you'll find a new log file full of trace information inside of the `logs` folder of the `WebApi2Book` root directory that we created back in Chapter 4.

Error handling is next, and it's a little bit more involved.

Error Handling

With the release of ASP.NET Web API 2.1, developers finally have framework support for global handling of unhandled exceptions. The framework also now supports multiple exception loggers, each of which has access to the exception objects themselves and to the contexts in which the exceptions occur. In this section, we'll implement a global exception handler and a custom exception logger, as summarized in Table 5-5.

Table 5-5. *Error-Handling Classes*

Class	Purpose
SimpleExceptionLogger	This simple class' single responsibility is to log exceptions. Although exceptions will also be logged using the tracing we just added in the previous section, this class has access to additional—actually, essential—information not available via the tracing infrastructure.
GlobalExceptionHandler	This is a custom exception handler, which we will use to replace the ASP.NET Web API default exception handler. This allows us to customize the HTTP response that is sent when an unhandled application exception occurs.
SimpleErrorResult	This trivial `IHttpActionResult` implementation is used to add result information to the exception context. This information is later applied to the HTTP response by ASP.NET Web API.

EXCEPTION LOGGING WITH ELMAH

The ASP.NET Web API team has created a sample demonstrating the logging of all unhandled exceptions with the popular ELMAH framework. It's definitely worth checking out:

`http://aspnet.codeplex.com/SourceControl/latest#Samples/WebApi/Elmah/ReadMe.txt`

First, add the following class to a new `ErrorHandling` folder in the `WebApi2Book.Web.Common` project:

SimpleExceptionLogger Class

```
using System.Web.Http.ExceptionHandling;
using log4net;
using WebApi2Book.Common.Logging;

namespace WebApi2Book.Web.Common.ErrorHandling
{
    public class SimpleExceptionLogger : ExceptionLogger
    {
        private readonly ILog _log;

        public SimpleExceptionLogger(ILogManager logManager)
        {
            _log = logManager.GetLog(typeof (SimpleExceptionLogger));
        }

        public override void Log(ExceptionLoggerContext context)
        {
            _log.Error("Unhandled exception", context.Exception);
        }
    }
}
```

Note that the `SimpleExceptionLogger` class derives from the framework's `ExceptionLogger` class. Also note that the exception is obtained from the `ExceptionLoggerContext` argument. We could do a lot more with this, seeing that we have access to all of the context information, but for the sake of simplicity this will suffice.

To register our exception logger with the framework, add the following to the bottom of the `WebApiConfig` class' `Register` method:

```
config.Services.Add(typeof (IExceptionLogger),
    new SimpleExceptionLogger(WebContainerManager.Get<ILogManager>()));
```

■ **Note** You'll need to add the following `using` directives:

- `using System.Web.Http.ExceptionHandling;`
- `using WebApi2Book.Web.Common.ErrorHandling;`

That's it! However, to help prevent exceptions from escaping without being logged during the application's startup sequence (i.e., before this configuration is fully applied at run time), implement the following method in the `WebApiApplication` class. This will handle the base `HttpApplication` class' `Error` event:

```
protected void Application_Error()
{
    var exception = Server.GetLastError();
    if (exception != null)
    {
        var log = WebContainerManager.Get<ILogManager>().GetLog(typeof (WebApiApplication));
        log.Error("Unhandled exception.", exception);
    }
}
```

■ **Note** You'll need to add the following `using` directives:

- `using WebApi2Book.Common.Logging;`
- `using WebApi2Book.Web.Common;`

Again, we add this to handle any exceptions that are thrown in the application start-up sequence, before the application has fully initialized the global error handling configuration. As for the implementation, the main thing to note is that the exception that raised the `Error` event must be accessed by a call to the `GetLastError` method.

With global exception logging fully in place, let's now focus our efforts on something else made possible with the ASP.NET Web API 2.1 release: globally customizing behavior of unhandled exceptions. To make this more interesting, let's first implement a couple of custom exceptions that we'll need later: `RootObjectNotFoundException` and `ChildObjectNotFoundException`. Implement these in a new `Exceptions` folder in the `WebApi2Book.Data` project.

`RootObjectNotFoundExceptions` will be thrown by our various data access objects when the primary, or "aggregate root," data object is not found. Similarly, `ChildObjectNotFoundExceptions` will be thrown when a required child of a primary data object is not found (for example, a request is made to add a user to a task, but that user doesn't exist). This design allows our data access classes to be free of any concerns about dealing with forming proper HTTP error responses. And, as you'll soon see in our custom `ExceptionHandler`, it allows us to globally handle these exceptions by creating appropriate HTTP error responses in a class that *is* aware of HTTP.

RootObjectNotFoundException Class

```
using System;

namespace WebApi2Book.Data.Exceptions
{
    [Serializable]
    public class RootObjectNotFoundException : Exception
    {
        public RootObjectNotFoundException(string message) : base(message)
        {
        }
    }
}
```

ChildObjectNotFoundException Class

```
using System;

namespace WebApi2Book.Data.Exceptions
{
    [Serializable]
    public class ChildObjectNotFoundException : Exception
    {
        public ChildObjectNotFoundException(string message) : base(message)
        {
        }
    }
}
```

Next, let's create a simple `IHttpActionResult` implementation to help communicate the error information back to the caller. Add it to the `ErrorHandling` folder in the `WebApi2Book.Web.Common` project:

```
using System.Net;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
using System.Web.Http;

namespace WebApi2Book.Web.Common.ErrorHandling
{
    public class SimpleErrorResult : IHttpActionResult
    {
        private readonly string _errorMessage;
        private readonly HttpRequestMessage _requestMessage;
        private readonly HttpStatusCode _statusCode;

        public SimpleErrorResult(HttpRequestMessage requestMessage, HttpStatusCode statusCode,
            string errorMessage)
        {
            _requestMessage = requestMessage;
            _statusCode = statusCode;
            _errorMessage = errorMessage;
        }

        public Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
        {
            return Task.FromResult(_requestMessage.CreateErrorResponse(_statusCode, _errorMessage));
        }
    }
}
```

It doesn't get much simpler than this! This is an absolutely bare-bones `IHttpActionResult` implementation where we simply use the request message to create a response based on the status code and error message specified in the constructor. (Don't feel cheated; we'll get to a more interesting `IHttpActionResult` in the next section!) But what calls the constructor? You're about to find out.

Now we will replace the default exception handler so that we can fully control the HTTP response message that is sent when an unhandled exception occurs. Start off by referencing `System.Web` from the `WebApi2Book.Web.Common` project, and then implement the following custom `ExceptionHandler` class in the `ErrorHandling` folder of that same project:

```
using System.Net;
using System.Web;
using System.Web.Http.ExceptionHandling;
using WebApi2Book.Data.Exceptions;

namespace WebApi2Book.Web.Common.ErrorHandling
{
    public class GlobalExceptionHandler : ExceptionHandler
    {
        public override void Handle(ExceptionHandlerContext context)
        {
            var exception = context.Exception;

            var httpException = exception as HttpException;
            if (httpException != null)
            {
                context.Result = new SimpleErrorResult(context.Request,
                    (HttpStatusCode) httpException.GetHttpCode(), httpException.Message);
                return;
            }

            if (exception is RootObjectNotFoundException)
            {
                context.Result = new SimpleErrorResult(context.Request, HttpStatusCode.NotFound,
                    exception.Message);
                return;
            }

            if (exception is ChildObjectNotFoundException)
            {
                context.Result = new SimpleErrorResult(context.Request, HttpStatusCode.Conflict,
                    exception.Message);
                return;
            }

            context.Result = new SimpleErrorResult(context.Request, HttpStatusCode.InternalServerError,
                exception.Message);
        }
    }
}
```

Some items of note:

- The class derives from the framework's `ExceptionHandler` class.
- The `GlobalExceptionHandler` is what constructs the `SimpleErrorResult` instances. By examining the exception available in the framework-provided `ExceptionHandlerContext`, the `GlobalExceptionHandler` uses the `SimpleErrorResult` class to form a response with the proper `HttpStatusCode` and response message.

- For `HttpException` objects, the handler creates the response using the exception's status code and message. This ensures the appropriate code is returned in the response, and it ensures that only the exception's message—not its stack trace—is returned as well. Stack trace information is not lost, however. The full exception, including its stack trace, is logged using the `SimpleExceptionLogger` that we just implemented.

We'll add to this class as we get farther along in our controller implementation and have new types of exceptions that we need to handle at a global level.

To wrap things up, we still need to register our custom exception handler with the framework. Let's do that now, replacing the default exception handler with our custom exception handler. Add the following line to the bottom of the `Register` method in the `WebApiConfig` class:

```
config.Services.Replace(typeof (IExceptionHandler), new GlobalExceptionHandler());
```

At this point, with support for routing, dependency injection, persistence, tracing, and error handling in place, we're ready to advance our TasksControllers past their current "hello world-ish" implementation.

Persisting a Task and Returning IHttpActionResult

We're about to embark on another relatively long journey, but at the end of it we'll be able to persist `Task` objects to the database. We'll also have implemented a custom `IHttpActionResult` that automatically sets the `HttpStatusCode` and location header in the response messages. In addition, we'll explore several useful tools, techniques, and architectural patterns along the way. Good stuff! So let's take that first step.

New Service Model Type

The `Task` service model class we implemented in Chapter 4 contains several properties. Most of these are not necessary, or desired, for adding a new task to the task-management system. Therefore, we will introduce a new service model class, `NewTask`.

Before we do this, let's remove the `Models` folder from the `WebApi2Book.Web.Api` project. We'll remove it for a couple of reasons, the most important one being that it is unnecessary (the other being that it introduces ambiguity to the coding convention we stated at the beginning of the chapter).

Now we can implement `NewTask` as follows; by now, we assume you can infer where to locate it, based on the namespace:

```
using System;
using System.Collections.Generic;

namespace WebApi2Book.Web.Api.Models
{
    public class NewTask
    {
        public string Subject { get; set; }

        public DateTime? StartDate { get; set; }

        public DateTime? DueDate { get; set; }

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

Now modify the signature of the AddTask method in the V1 TasksController so that it appears as follows, accepting one of these NewTask objects as a parameter:

```
public Task AddTask(HttpRequestMessage requestMessage, NewTask newTask)
```

And that's it. However, we encourage you to follow along with the book's project code, where you'll see that we also added a NewTaskV2 class to use with the V2 controller. The point of NewTaskV2 is to illustrate that a service model may be changed (in this case simplified) based on user/developer feedback. In V2, the caller only has to provide a single User, not a list of them:

```
using System;

namespace WebApi2Book.Web.Api.Models
{
    public class NewTaskV2
    {
        public string Subject { get; set; }

        public DateTime? StartDate { get; set; }

        public DateTime? DueDate { get; set; }

        public User Assignee { get; set; }
    }
}
```

You can add NewTaskV2 to the project if you'd like, but at this point we're finished discussing things pertaining to V2 in the book. We've got to get on with persisting the task!

SECURITY IMPLICATIONS OF OVERPOSTING

Though we're going to cover the topic of security in the next chapter, we heartily recommend Badrinarayanan Lakshmiraghavan's excellent book, *Pro ASP.NET Web API Security: Securing ASP.NET Web API* (Apress, 2013). It goes into a level of detail that we can't possibly cover in one chapter of our book.

In short, an overposting attack is defined as an HTTP POST or PUT that successfully submits and updates the value of an underlying data property, even though that property wasn't part of the published, visible, or expected request message. For example, a web form in an HR application, or a REST endpoint in an HR-based Web API, might expect a caller to update an employee's first and last names. But if that server-side code is naively binding the incoming POST body to a model object that happens to include a Salary property, and the caller happens to specify a value for that Salary (either maliciously or unintentionally), the code may find itself updating the employee's salary in the database to the specified value.

According to Badrinarayanan, "The best approach to prevent overposting vulnerabilities in ASP.NET Web API is to never use entity classes directly for [service] model binding. Using a subset of the entity class that expects nothing more and nothing less for the scenario at hand is the best approach." And that's exactly what we've done by introducing the NewTask and NewTaskV2 classes.

Persisting the Task

As we said earlier in this chapter, controllers should not be doing much more than simply using the functionality offered by various dependencies. Now it's time to practice what we preach.

Let's begin by creating and configuring several new classes to do the heavy lifting in the `AddTask` method. First, add the following security-related types to the correct projects in the solution. The `IUserSession` and related user classes allow us to abstract away from the static `HttpContext.Current.User` property. Again, by now we assume you can infer where to locate these, based on the namespace (this message will not be repeated):

IUserSession Interface

```
namespace WebApi2Book.Common.Security
{
    public interface IUserSession
    {
        string Firstname { get; }
        string Lastname { get; }
        string Username { get; }
        bool IsInRole(string roleName);
    }
}
```

IWebUserSession Interface

```
using System;
using WebApi2Book.Common.Security;

namespace WebApi2Book.Web.Common.Security
{
    public interface IWebUserSession : IUserSession
    {
        string ApiVersionInUse { get; }
        Uri RequestUri { get; }
        string HttpRequestMethod { get; }
    }
}
```

UserSession Class

```
using System;
using System.Security.Claims;
using System.Web;
using System.Linq;

namespace WebApi2Book.Web.Common.Security
{
    public class UserSession : IWebUserSession
    {
        public string Firstname
        {
            get { return ((ClaimsPrincipal) HttpContext.Current.User).FindFirst(ClaimTypes.
GivenName).Value; }
        }
```

```
        public string Lastname
        {
            get { return ((ClaimsPrincipal) HttpContext.Current.User).FindFirst(ClaimTypes.Surname).
Value; }
        }

        public string Username
        {
            get { return ((ClaimsPrincipal) HttpContext.Current.User).FindFirst(ClaimTypes.Name).
Value; }
        }

        public bool IsInRole(string roleName)
        {
            return HttpContext.Current.User.IsInRole(roleName);
        }

        public Uri RequestUri
        {
            get { return HttpContext.Current.Request.Url; }
        }

        public string HttpRequestMethod
        {
            get { return HttpContext.Current.Request.HttpMethod; }
        }

        public string ApiVersionInUse
        {
            get
            {
                const int versionIndex = 2;
                if (HttpContext.Current.Request.Url.Segments.Count() < versionIndex + 1)
                {
                    return string.Empty;
                }

                var apiVersionInUse = HttpContext.Current.Request.Url.Segments[versionIndex].Replace(
                    "/", string.Empty);
                return apiVersionInUse;
            }
        }
    }
}
```

We'll discuss these further in Chapter 6, "Securing the Service." For now, just understand that these provide convenient access to data describing the current user and request. Before moving on, though, wire this up so that it can be injected as a dependency. This is done first by adding the following method to `NinjectConfigurator`:

```
private void ConfigureUserSession(IKernel container)
{
    var userSession = new UserSession();
    container.Bind<IUserSession>().ToConstant(userSession).InSingletonScope();
    container.Bind<IWebUserSession>().ToConstant(userSession).InSingletonScope();
}
```

Then modify the `AddBindings` method so that it appears as follows:

```
private void AddBindings(IKernel container)
{
   ConfigureLog4net(container);
   ConfigureUserSession(container);
   ConfigureNHibernate(container);

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

■ **Note** You'll need to add the following `using` directives:

- `using WebApi2Book.Common.Security;`
- `using WebApi2Book.Web.Common.Security;`

Also note that we are configuring an application-wide singleton `UserSession` to be injected for all objects requiring an `IUserSession` or an `IWebUserSession`. This works because `UserSession` implements both interfaces, and because it does not store any state.

Next, add the following types that we'll use to interact with the database via the NHibernate `ISession` object:

IAddTaskQueryProcessor Interface

```
using WebApi2Book.Data.Entities;

namespace WebApi2Book.Data.QueryProcessors
{
    public interface IAddTaskQueryProcessor
    {
        void AddTask(Task task);
    }
}
```

AddTaskQueryProcessor Class

```
using NHibernate;
using NHibernate.Util;
using WebApi2Book.Common;
using WebApi2Book.Common.Security;
```

```
using WebApi2Book.Data.Entities;
using WebApi2Book.Data.Exceptions;
using WebApi2Book.Data.QueryProcessors;

namespace WebApi2Book.Data.SqlServer.QueryProcessors
{
    public class AddTaskQueryProcessor : IAddTaskQueryProcessor
    {
        private readonly IDateTime _dateTime;
        private readonly ISession _session;
        private readonly IUserSession _userSession;

        public AddTaskQueryProcessor(ISession session, IUserSession userSession, IDateTime dateTime)
        {
            _session = session;
            _userSession = userSession;
            _dateTime = dateTime;
        }

        public void AddTask(Task task)
        {
            task.CreatedDate = _dateTime.UtcNow;
            task.Status = _session.QueryOver<Status>().Where(
                x => x.Name == "Not Started").SingleOrDefault();
            task.CreatedBy = _session.QueryOver<User>().Where(
                x => x.Username == _userSession.Username).SingleOrDefault();

            if (task.Users != null && task.Users.Any())
            {
                for (var i = 0; i < task.Users.Count; ++i)
                {
                    var user = task.Users[i];
                    var persistedUser = _session.Get<User>(user.UserId);
                    if (persistedUser == null)
                    {
                        throw new ChildObjectNotFoundException("User not found");
                    }
                    task.Users[i] = persistedUser;
                }
            }

            _session.SaveOrUpdate(task);
        }
    }
}
```

Here we see our new `IUserSession` being used, along with the `ISession` and `IDateTime` we discussed previously. Oh, and note the `ChildObjectNotFoundException`! Anyway, what's basically happening in `AddTask` is this:

- The system assigns the task's created date.
- The system assigns a status of Not Started to the task.
- The system uses the user session to indicate the user who created the task.
- The system populates the task's `Users` collection, forming associations with the `User` objects fetched from the database. It verifies that all users specified in the new task exist in the system.
- The system persists the task and all its relationships.

Before we move on, we'll wire this up so that it can be injected as a dependency. This is done by adding the following to the bottom of the `NinjectConfigurator AddBindings` method:

```
container.Bind<IAddTaskQueryProcessor>().To<AddTaskQueryProcessor>().InRequestScope();
```

■ **Note** You'll need to add the following `using` directives:

- `using WebApi2Book.Data.QueryProcessors;`
- `using WebApi2Book.Data.SqlServer.QueryProcessors;`

OK, time for something really cool. We've fallen in love with AutoMapper, a great tool developed by Jimmy Bogard that can be used to easily transfer data from one object to another. As you'll soon see, this comes in handy when mapping between the entity and service model types. We added the AutoMapper NuGet package in Chapter 4, but before we use it we will wrap the primary mapping functions so that we can inject them as dependencies. (Yes, we're a bit disappointed AutoMapper uses static methods.)

Start by adding the following types to the correct projects in the solution:

IAutoMapperTypeConfigurator Interface

```
namespace WebApi2Book.Common.TypeMapping
{
    public interface IAutoMapperTypeConfigurator
    {
        void Configure();
    }
}
```

IAutoMapper Interface

```
namespace WebApi2Book.Common.TypeMapping
{
    public interface IAutoMapper
    {
        T Map<T>(object objectToMap);
    }
}
```

AutoMapperAdapter Class

```
using AutoMapper;

namespace WebApi2Book.Common.TypeMapping
{
    public class AutoMapperAdapter : IAutoMapper
    {
        public T Map<T>(object objectToMap)
        {
            return Mapper.Map<T>(objectToMap);
        }
    }
}
```

To wire it in with Ninject, first add the following method to `NinjectConfigurator`:

```
private void ConfigureAutoMapper(IKernel container)
{
    container.Bind<IAutoMapper>().To<AutoMapperAdapter>().InSingletonScope();
}
```

Then modify the `AddBindings` method so that it appears as follows (adding the call to `ConfigureAutoMapper`):

```
private void AddBindings(IKernel container)
{
    ConfigureLog4net(container);
    ConfigureUserSession(container);
    ConfigureNHibernate(container);
    ConfigureAutoMapper(container);

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

    container.Bind<IAddTaskQueryProcessor>().To<AddTaskQueryProcessor>().InRequestScope();
}
```

■ **Note** You'll need to add a `using` directive to `WebApi2Book.Common.TypeMapping` to satisfy the compiler.

Excellent. Now let's configure some mappings. For this, we need to implement `IAutoMapperTypeConfigurator` for each mapping. Start by implementing the mapping from the `NewTask` service model type to the `Task` entity type using AutoMapper's fluent syntax:

```
using AutoMapper;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Web.Api.Models;
```

```
namespace WebApi2Book.Web.Api.AutoMappingConfiguration
{
    public class NewTaskToTaskEntityAutoMapperTypeConfigurator : IAutoMapperTypeConfigurator
    {
        public void Configure()
        {
            Mapper.CreateMap<NewTask, Data.Entities.Task>()
                .ForMember(opt => opt.Version, x => x.Ignore())
                .ForMember(opt => opt.CreatedBy, x => x.Ignore())
                .ForMember(opt => opt.TaskId, x => x.Ignore())
                .ForMember(opt => opt.CreatedDate, x => x.Ignore())
                .ForMember(opt => opt.CompletedDate, x => x.Ignore())
                .ForMember(opt => opt.Status, x => x.Ignore())
                .ForMember(opt => opt.Users, x => x.Ignore());
        }
    }
}
```

The `Mapper.CreateMap` method establishes a default mapping between the properties on the `NewTask` class and the properties on the `Task` class. We then fluently tweak this mapping by informing the mapper to ignore several properties on the target class (`Task`) because they aren't available on the source class (`NewTask`). To gain a deeper understanding of AutoMapper, we encourage you to visit the AutoMapper site on GitHub (`https://github.com/AutoMapper/AutoMapper`); however, this is the gist of what's happening in the code.

Now let's implement the mapping from the `Task` entity to the `Task` service model type. This mapping will be used to return the newly generated `Task` object data to the caller. This is more complicated because it requires an AutoMapper resolver. Add the following:

TaskEntityToTaskAutoMapperTypeConfigurator Class

```
using AutoMapper;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Data.Entities;

namespace WebApi2Book.Web.Api.AutoMappingConfiguration
{
    public class TaskEntityToTaskAutoMapperTypeConfigurator : IAutoMapperTypeConfigurator
    {
        public void Configure()
        {
            Mapper.CreateMap<Task, Models.Task>()
                .ForMember(opt => opt.Links, x => x.Ignore())
                .ForMember(opt => opt.Assignees, x => x.ResolveUsing<TaskAssigneesResolver>());
        }
    }
}
```

TaskAssigneesResolver Class

```
using System.Collections.Generic;
using System.Linq;
using AutoMapper;
using WebApi2Book.Common.TypeMapping;
```

```
using WebApi2Book.Data.Entities;
using WebApi2Book.Web.Common;
using User = WebApi2Book.Web.Api.Models.User;

namespace WebApi2Book.Web.Api.AutoMappingConfiguration
{
    public class TaskAssigneesResolver : ValueResolver<Task, List<User>>
    {
        public IAutoMapper AutoMapper
        {
            get { return WebContainerManager.Get<IAutoMapper>(); }
        }

        protected override List<User> ResolveCore(Task source)
        {
            return source.Users.Select(x => AutoMapper.Map<User>(x)).ToList();
        }
    }
}
```

The `TaskEntityToTaskAutoMapperTypeConfigurator` is fairly straightforward. It ignores the links, because they aren't present on the source (i.e., the entity), and it delegates the Assignees mapping to the resolver.

By examining the resolver's `ResolveCore` method, we can see that this is mapping users from the source (entity) representation to the target (service model) representation. If you get the sense that something is missing here, you are correct. What's missing is the mapping between entity and service model representations for users, which we'll get to now (please add the following):

UserToUserEntityAutoMapperTypeConfigurator Class

```
using AutoMapper;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.AutoMappingConfiguration
{
    public class UserToUserEntityAutoMapperTypeConfigurator : IAutoMapperTypeConfigurator
    {
        public void Configure()
        {
            Mapper.CreateMap<User, Data.Entities.User>()
                .ForMember(opt => opt.Version, x => x.Ignore());
        }
    }
}
```

UserEntityToUserAutoMapperTypeConfigurator Class

```
using AutoMapper;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Data.Entities;
```

```
namespace WebApi2Book.Web.Api.AutoMappingConfiguration
{
    public class UserEntityToUserAutoMapperTypeConfigurator : IAutoMapperTypeConfigurator
    {
        public void Configure()
        {
            Mapper.CreateMap<User, Models.User>()
                .ForMember(opt => opt.Links, x => x.Ignore());
        }
    }
}
```

Although it's not as obvious as the situation with `User` types, to complete our mapping we need to map the different `Status` types. If we don't, AutoMapper will blow up when trying to locate a `Status` mapping for the different `Task` class' `Status` property. Therefore, add the following trivial implementations:

StatusToStatusEntityAutoMapperTypeConfigurator Class

```
using AutoMapper;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.AutoMappingConfiguration
{
    public class StatusToStatusEntityAutoMapperTypeConfigurator : IAutoMapperTypeConfigurator
    {
        public void Configure()
        {
            Mapper.CreateMap<Status, Data.Entities.Status>()
                .ForMember(opt => opt.Version, x => x.Ignore());
        }
    }
}
```

StatusEntityToStatusAutoMapperTypeConfigurator Class

```
using AutoMapper;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Data.Entities;

namespace WebApi2Book.Web.Api.AutoMappingConfiguration
{
    public class StatusEntityToStatusAutoMapperTypeConfigurator : IAutoMapperTypeConfigurator
    {
        public void Configure()
        {
            Mapper.CreateMap<Status, Models.Status>();
        }
    }
}
```

Ah, but there are still some missing pieces regarding this AutoMapper stuff. One, we need a way to make sure the configurators get run somewhere in the application's start-up sequence. Two, we need to configure the `IAutoMapperTypeConfigurator` classes to be injected in as dependencies. First things first, so implement the following class in the `App_Start` folder:

```
using System.Collections.Generic;
using System.Linq;
using AutoMapper;
using WebApi2Book.Common.TypeMapping;

namespace WebApi2Book.Web.Api
{
    public class AutoMapperConfigurator
    {
        public void Configure(IEnumerable<IAutoMapperTypeConfigurator> autoMapperTypeConfigurations)
        {
            autoMapperTypeConfigurations.ToList().ForEach(x => x.Configure());

            Mapper.AssertConfigurationIsValid();
        }
    }
}
```

The `Configure` method, when invoked, will configure each of the `IAutoMapperTypeConfigurator` instances and then assert that the entire mapping scheme is valid. And how do we integrate this into the application's start-up sequence? Easily, by modifying the `WebApiApplication` to appear as follows:

```
using System.Web;
using System.Web.Http;
using WebApi2Book.Common.Logging;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Web.Common;

namespace WebApi2Book.Web.Api
{
    public class WebApiApplication : HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);

            new AutoMapperConfigurator().Configure(
                WebContainerManager.GetAll<IAutoMapperTypeConfigurator>());
        }

        protected void Application_Error()
        {
            var exception = Server.GetLastError();
```

```
            if (exception != null)
            {
                var log = WebContainerManager.Get<ILogManager>().GetLog(typeof(WebApiApplication));
                log.Error("Unhandled exception.", exception);
            }
        }
    }
}
```

Now, to make our `IAutoMapperTypeConfigurator` instances available for injection, we modify the `NinjectConfigurator` class' `ConfigureAutoMapper` method so that it appears as follows:

```
private void ConfigureAutoMapper(IKernel container)
{
    container.Bind<IAutoMapper>().To<AutoMapperAdapter>().InSingletonScope();

    container.Bind<IAutoMapperTypeConfigurator>()
        .To<StatusEntityToStatusAutoMapperTypeConfigurator>()
        .InSingletonScope();
    container.Bind<IAutoMapperTypeConfigurator>()
        .To<StatusToStatusEntityAutoMapperTypeConfigurator>()
        .InSingletonScope();
    container.Bind<IAutoMapperTypeConfigurator>()
        .To<UserEntityToUserAutoMapperTypeConfigurator>()
        .InSingletonScope();
    container.Bind<IAutoMapperTypeConfigurator>()
        .To<UserToUserEntityAutoMapperTypeConfigurator>()
        .InSingletonScope();
    container.Bind<IAutoMapperTypeConfigurator>()
        .To<NewTaskToTaskEntityAutoMapperTypeConfigurator>()
        .InSingletonScope();
    container.Bind<IAutoMapperTypeConfigurator>()
        .To<TaskEntityToTaskAutoMapperTypeConfigurator>()
        .InSingletonScope();
}
```

■ **Note** You'll need to add a `using` directive to `WebApi2Book.Web.Api.AutoMappingConfiguration` to satisfy the compiler.

OK, we're in the homestretch! Now we'll add the dependency that the controller will use to add the new task. Implement the following processor (which will in turn use the `IAddTaskQueryProcessor` we created earlier):

IAddTaskMaintenanceProcessor Interface

```
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing
{
    public interface IAddTaskMaintenanceProcessor
```

```
    {
        Task AddTask(NewTask newTask);
    }
}
```

AddTaskMaintenanceProcessor Class

```
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Data.QueryProcessors;
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing
{
    public class AddTaskMaintenanceProcessor : IAddTaskMaintenanceProcessor
    {
        private readonly IAutoMapper _autoMapper;
        private readonly IAddTaskQueryProcessor _queryProcessor;

        public AddTaskMaintenanceProcessor(IAddTaskQueryProcessor queryProcessor,
            IAutoMapper autoMapper)
        {
            _queryProcessor = queryProcessor;
            _autoMapper = autoMapper;
        }

        public Task AddTask(NewTask newTask)
        {
            var taskEntity = _autoMapper.Map<Data.Entities.Task>(newTask);

            _queryProcessor.AddTask(taskEntity);

            var task = _autoMapper.Map<Task>(taskEntity);

            return task;
        }
    }
}
```

This should be pretty easy to figure out. Using injected dependencies that we've implemented, `AddTask` maps the `NewTask` service model object to an entity object and then persists it. It then maps the new entity back to a full `Task` service model object and returns it to the caller.

And now, as you may have guessed, we need to wire it up with Ninject by adding the following to the `AddBindings` method in `NinjectConfigurator`:

```
container.Bind<IAddTaskMaintenanceProcessor>().To<AddTaskMaintenanceProcessor>()
        .InRequestScope();
```

■ **Note** You'll need to add a `using` directive to `WebApi2Book.Web.Api.MaintenanceProcessing` to satisfy the compiler.

At this point, we're ready to hook all of this up to the controller so that we can start actually persisting tasks. Reimplement the `TasksController` as follows:

```
using System.Net.Http;
using System.Web.Http;
using WebApi2Book.Web.Api.Models;
using WebApi2Book.Web.Common;
using WebApi2Book.Web.Common.Routing;
using WebApi2Book.Web.Api.MaintenanceProcessing;

namespace WebApi2Book.Web.Api.Controllers.V1
{
    [ApiVersion1RoutePrefix("tasks")]
    [UnitOfWorkActionFilter]
    public class TasksController : ApiController
    {
        private readonly IAddTaskMaintenanceProcessor _addTaskMaintenanceProcessor;

        public TasksController(IAddTaskMaintenanceProcessor addTaskMaintenanceProcessor)
        {
            _addTaskMaintenanceProcessor = addTaskMaintenanceProcessor;
        }

        [Route("", Name = "AddTaskRoute")]
        [HttpPost]
        public Task AddTask(HttpRequestMessage requestMessage, NewTask newTask)
        {
            var task = _addTaskMaintenanceProcessor.AddTask(newTask);

            return task;
        }
    }
}
```

In this implementation, we see the controller simply delegating its work to the `IAddTaskMaintenanceProcessor`. Yes, we practice what we preach. The benefits of this will be reemphasized in our discussion of legacy SOAP message support. And you can imagine the benefits this brings to the controller in terms of unit-testability.

Anyway, for now, go ahead and run the demo that we first ran back at the end of the "Implementing POST" section. What do you see? An exception??? Yes, an exception. As a consolation, open the log file and note how well-documented the exception appears! Also note that the client (we're using Fiddler) received a response prepared by our custom `GlobalExceptionHandler`. At least it's nice to know that our error handling is working!

The cause of the exception, as you may have deduced by examining the log file, is that there is no User in the session. This is because we haven't done any authentication or authorization. Since we won't get to that until the next chapter, and since we really want to see the persistence work, we're going to implement a temporary hack. Change the following line in `AddTaskQueryProcessor` from this

```
task.CreatedBy = _session.QueryOver<User>().Where(x => x.Username ==
                                _userSession.Username).SingleOrDefault();
```

to this:

```
task.CreatedBy = _session.Get<User>(1L); // HACK: All tasks created by user 1 for now
```

And then re-run the demo. You should see something similar to the following (abbreviated) response:

```
HTTP/1.1 200 OK
Content-Type: text/json; charset=utf-8

{"TaskId":8,"Subject":"Fix something important","StartDate":null,"DueDate":null,
"CreatedDate":"2014-05-03T23:02:53.2704726Z","CompletedDate":null,
"Status":{"StatusId":1,"Name":"Not Started","Ordinal":0},"Assignees":[],"Links":[]}
```

Furthermore, if you examine the task table, you will see that the `Task` has been persisted to the database. Great! However (isn't there always a "however"?), we've got to do something about the HTTP status code. A web API should be returning a 201 to indicate that a new resource was created, but ours is returning a 200. We should also be populating the response message's `Location` header. We'll address these functional gaps next by implementing a custom `IHttpActionResult`.

IHttpActionResult

Before we create the custom `IHttpActionResult` implementation, we need to implement some prerequisites. First, add the following interface:

```
using System.Collections.Generic;

namespace WebApi2Book.Web.Api.Models
{
    public interface ILinkContaining
    {
        List<Link> Links { get; set; }
        void AddLink(Link link);
    }
}
```

Next, modify the `Task` service model class to implement the interface. Fortunately, this is trivial, because the interface members are already implemented by `Task`; the interface just needs to be added to the class declaration:

```
public class Task : ILinkContaining
```

Now add the `Constants` class. This is just a convenience class aimed at reducing "magic numbers." We only need one of the members for now, but we'll go ahead and add the entire (projected) implementation so that we don't need to revisit it later:

```
namespace WebApi2Book.Common
{
    public static class Constants
    {
        public static class MediaTypeNames
        {
            public const string ApplicationXml = "application/xml";
            public const string TextXml = "text/xml";
            public const string ApplicationJson = "application/json";
            public const string TextJson = "text/json";
        }
```

```
        public static class Paging
        {
            public const int MinPageSize = 1;
            public const int MinPageNumber = 1;
            public const int DefaultPageNumber = 1;
        }

        public static class CommonParameterNames
        {
            public const string PageNumber = "pageNumber";
            public const string PageSize = "pageSize";
        }

        public static class CommonLinkRelValues
        {
            public const string Self = "self";
            public const string All = "all";
            public const string CurrentPage = "currentPage";
            public const string PreviousPage = "previousPage";
            public const string NextPage = "nextPage";
        }

        public static class CommonRoutingDefinitions
        {
            public const string ApiSegmentName = "api";
            public const string ApiVersionSegmentName = "apiVersion";
            public const string CurrentApiVersion = "v1";
        }

        public static class SchemeTypes
        {
            public const string Basic = "basic";
        }

        public static class RoleNames
        {
            public const string Manager = "Manager";
            public const string SeniorWorker = "SeniorWorker";
            public const string JuniorWorker = "JuniorWorker";
        }

        public const string DefaultLegacyNamespace = "http://tempuri.org/";
    }
}
```

The next prerequisite type to add is `LocationLinkCalculator`. This is just a little utility class encapsulating a trivial algorithm that returns the location link from a service model's collection of `Link` objects. Note that it is static (gasp!). While we generally discourage the use of statics because of the tight coupling they introduce, they can be appropriate for trivial utility functions. Does this calculator need to support polymorphism? Does it need to be mocked? No, and probably not, respectively. So implement it as follows:

```
using System;
using System.Linq;
using WebApi2Book.Common;
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing
{
    public static class LocationLinkCalculator
    {
        public static Uri GetLocationLink(ILinkContaining linkContaining)
        {
            var locationLink = linkContaining.Links.FirstOrDefault(
                x => x.Rel == Constants.CommonLinkRelValues.Self);
            return locationLink == null ? null : new Uri(locationLink.Href);
        }
    }
}
```

With prerequisites complete, we're now ready to implement the `IHttpActionResult` that indicates a new task has been created. Implement it as follows:

```
using System.Net;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
using System.Web.Http;
using Task = WebApi2Book.Web.Api.Models.Task;

namespace WebApi2Book.Web.Api.MaintenanceProcessing
{
    public class TaskCreatedActionResult : IHttpActionResult
    {
        private readonly Task _createdTask;
        private readonly HttpRequestMessage _requestMessage;

        public TaskCreatedActionResult(HttpRequestMessage requestMessage,
            Task createdTask)
        {
            _requestMessage = requestMessage;
            _createdTask = createdTask;
        }

        public Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
        {
            return System.Threading.Tasks.Task.FromResult(Execute());
        }
```

```
        public HttpResponseMessage Execute()
        {
            var responseMessage = _requestMessage.CreateResponse(
                HttpStatusCode.Created, _createdTask);

            responseMessage.Headers.Location = LocationLinkCalculator.GetLocationLink(_createdTask);

            return responseMessage;
        }
    }
}
```

Looking at the implementation, we see that the constructor accepts an `HttpRequestMessage` and a `Task` service model object. It caches those for use in the `Execute` method, which is invoked from the asynchronous `ExecuteAsync` method. `Execute` uses the request message to create a response with the proper `HttpStatusCode` and `Task` content. Then it also adds the `Location` header to the response.

At last, modify the `AddTask` method of the `TasksController` to return the custom result:

```
[Route("", Name = "AddTaskRoute")]
[HttpPost]
public IHttpActionResult AddTask(HttpRequestMessage requestMessage, NewTask newTask)
{
    var task = _addTaskMaintenanceProcessor.AddTask(newTask);
    var result = new TaskCreatedActionResult(requestMessage, task);
    return result;
}
```

Though links won't be fully developed until we implement our link services to add the actual links, we will still see the benefit in the response if we hack in a fake link. Therefore, modify the `AddTask` method of the `AddTaskMaintenanceProcessor` so that it appears as follows:

```
public Task AddTask(NewTask newTask)
{
    var taskEntity = _autoMapper.Map<Data.Entities.Task>(newTask);

    _queryProcessor.AddTask(taskEntity);

    var task = _autoMapper.Map<Task>(taskEntity);

    // TODO: Implement link service
    task.AddLink(new Link
    {
        Method = HttpMethod.Get.Method,
        Href = "http://localhost:61589/api/v1/tasks/" + task.TaskId,
        Rel = Constants.CommonLinkRelValues.Self
    });

    return task;
}
```

■ **Note** You'll need to add the following `using` directives:

- `using System.Net.Http;`
- `using WebApi2Book.Common;`

And now the moment we've been waiting for…it's demo time! Once again, execute the demo procedures described in the "Implementing POST" section. This time, you should see something similar to the following (abbreviated) response:

```
HTTP/1.1 201 Created
Content-Type: text/json; charset=utf-8
Location: http://localhost:61589/api/v1/tasks/10

{"TaskId":10,"Subject":"Fix something important","StartDate":null,"DueDate":null,
"CreatedDate":"2014-05-04T02:52:39.9872623Z","CompletedDate":null,
"Status":{"StatusId":1,"Name":"Not Started","Ordinal":0},"Assignees":[],
"Links":[{"Rel":"self","Href":"http://localhost:61589/api/v1/tasks","Method":"GET"}]}
```

Congratulations! But what's the point in returning this `IHttpActionResult`? Why not just return a `Task`? Well, our `IHttpActionResult` class' single responsibility is to encapsulate the logic of setting the response code and `Location` header in the response. We could have put the necessary logic in the controller, but we prefer to keep controllers "thin" and easy to unit test.

Summary

The task-management service we implemented is posting a task, using a message received via a versioned URL, and returning a proper result. We implemented the entire stack, and along the way we dealt with routing, controller selection, dependency management, persistence, type mapping, diagnostic tracing/logging, and error handling. To recap how all of this comes together at runtime, let's summarize it with some pseudo code:

1. Caller makes a web request.
2. ASP.NET Web API starts the activation of the appropriate controller, which is selected using our custom controller selector based on the URL routes registered at application start-up.
3. ASP.NET Web API uses the `NinjectDependencyResolver` to satisfy all of the dependencies of the controller, all of the dependencies each dependency requires, and so on. (It's recursive.)
4. If any object requires an `ISession` object, Ninject calls the `NinjectConfigurator.CreateSession` method to create the `ISession` instance.
5. The `CreateSession` method opens a new session and binds it to the web context so that it will be available for subsequent `ISession` requests.
6. ASP.NET Web API calls the custom unit of work attribute's `OnActionExecuting` override, which in turn starts a new database transaction.

7. ASP.NET Web API calls the controller action method, which uses dependencies that were injected in during controller activation to execute the actual business logic (i.e., the whole reason the method was called in the first place).

8. ASP.NET Web API calls the custom unit-of-work 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.

Wow, that's a lot going on for every web request! Yes, and the entire sequence is traced in great detail with the custom diagnostic tracing we configured. In fact, examining the trace log is an excellent way to understand what's going on under the hood!

We also demonstrated some architectural patterns inspired by SOLID design principles. Although all of this will be more meaningful once we add things like security, we've laid a great deal of the groundwork necessary to make much progress going forward.

In the next chapter, we will continue the exploration of "framework things" by examining 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.