It’s now time to start dealing with some of the more complex concerns in the task-management service. In the previous chapter, you started with an empty folder, created a basic source-tree structure, added a new Visual Studio 2012 solution, and added the projects you know you’ll need for this little REST service. You also added some of the more basic code components, and setup the project and library references you anticipated needing. While each of these things is certainly important and critical to the overall design of the service, you’ve yet to address any of the following:

- Controller activation
- Dependencies
- NHibernate configuration and mappings
- Database unit of work management
- Database transaction control
- Security
- Logging of service calls and exceptions

You’ll learn about security and logging in a later chapter. For now you want to focus on the controllers and their dependencies, as well as working with the NHibernate ISession object and its lifetime.

Controller Activation

As you’ll see throughout this chapter, services in the ASP.NET MVC Framework (including the Web API) center on the controller. You don’t really have views, and the model is simply used to hold and move data around inside the application. However, the controller is used by the framework to respond to requests; it runs all of the logic
the service requires. In other words, when a web request comes over the network and into IIS (or IIS Express or a self-hosted application), it uses the routes configured in your application to determine which controller should respond to the request. When the appropriate controller class is found, the MVC Framework creates an instance of that class and forwards the web request to the appropriate controller method.

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

```csharp
config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new {id = RouteParameter.Optional});
```

Using the power of URL routing, the MVC engine will try to match the URLs of requests against this and other routes. In this particular route, MVC 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 either case, MVC 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, MVC will automatically append the word `Controller` to the name taken from the URL.

At this point, you may be asking the question, “Which controller method will get invoked?” The answer depends on whether you’re using the Web API. If not, the method has to be specified in the URL itself—also known as the `action`. In that case, when not using the Web API, the routes will look different—as they will need to have an `{action}` segment in addition to the `{controller}` segment. But since this is a book about the Web API, you will stick with the approach the Web API takes for method invocation.

You’re using the Web API, so the URL route just shown doesn’t include an `{action}` segment. This is because the Web API 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`, MVC (via the Web API) will invoke the `Get()` method on the `TasksController` controller class. If the caller were performing a POST instead, then MVC 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, then MVC will invoke the corresponding method that matches a signature containing a single argument. Table 5-1 shows a few examples based on the task-management service’s `TasksController`. 