Controllers, Dependencies, and Managing the Database Unit of Work

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):

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 TasksControllercontroller 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.

Table 5-1. Examples of URLs and 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) |

By default, the RESTful features of the Web API help ensure that you are coding to the HTTP verbs discussed in Chapters 2 and 3. This is much cleaner and much truer to the REST style of services than using ASP.NET MVC without the Web API. A little later, you’ll see some of the other benefits of using the Web API.

In addition to using arguments that come from the URL, you can add additional arguments for data that represents your model object, as well as a .NET type called HttpRequestMessage.

Here’s the signature of a Post() method that will exist on theCategoriesController:

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. You have 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. In this Post() method, you will use it to return a response that includes the 201 HTTP response code, so you can give the caller the location (i.e., the address) of the newly created category.

Note that you could instead use the ApiController’s Request property to access the same object. Since the Web API controllers all inherit from the ApiController base class, you can use this property anywhere you need within the controller code. However, you should be careful not to couple your code to anything going on in a base class. Doing so generally makes it much more difficult to test, and it increases the fragility of your code. This is why I prefer to have the request object passed into me when needed. That makes it’s much easier to mock up during tests; it also means you’re not coupled to the ApiController base class (not as much, at least).

Composition Over Inheritance

There have been many books written in the last 10 or 15 years talking about various evolutions in the practice of object-oriented programming. One of the major shifts of the past 15 years is captured in the oft-quoted phrase, “composition over inheritance.” In short, an application’s code will be much less coupled—and much more testable and maintainable—if it utilizes composition of many smaller classes instead of relying on a base class for desired behaviors. When a base class is packed with a wide variety of functionality, your class—even if it’s small—implicitly takes on the base class's full surface. Changes to the base class require full test regression of all classes that inherit from it—even if those classes aren’t using the base functionality that was modified.

Further, relying on a base class for behaviors forces the base class to take on more than one responsibility—and this is in direct conflict with the Single Responsibility Principle. You see, if you limit a base class’s responsibility (i.e., its functionality) to only one thing, but child classes require the use of more than a single behavior, then you are forced to use composition in providing these behaviors. And at that point, using a base class loses most of its intended appeal.

If the concept of “composition over inheritance” is new to you, I encourage you to read Head First Design Patterns by Eric Freeman et al (O’Reilly, 2004). I think you will find its principles and presentation rather freeing, especially if you come from a strong 1990’s style OO background.

## Adding a Model Object Argument

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

For example, if a caller submits the following JSON in the body of a POST request to /api/categories, MVC will convert the text to a Category object as the second parameter in the Post() method:

{"Name":"Blue","Description":"Deferred"}

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 id argument will contain 123, along with the category object from the request body, as well as the request object itself. This is quite amazing because you don’t need to do any special parsing of the JSON or XML content. Nor do you need to define any data contracts, as would normally be done in a WCF-based service. In line with the more recent trend towards “convention over configuration,” it just works!

# Dependencies

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

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

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

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

## Constructor Injection of Dependencies

The concept of pushing all of the dependencies up to the constructor is really quite simple, but it can be tough to grasp and put into practice for the Dependency Injection novice. Think of it this way: a class should not use any behavior that does not come through the constructor. None. Zero. Zip. This includes even seemingly harmless classes such as System.DateTime, System.IO.File, System.Environment and many other basic utility classes within the .NET Framework. It also includes the types of behaviors listed in the previous section. If your class is using anything that isn’t implemented in that class, it needs to be injected in through the constructor. Period. You even want to avoid the use of static properties and methods. If your code needs to use the static DateTime.Now property, for example, you should wrap it in an injectable adapter class. You can find an example of this in this book’s sample code (and also a bit later in this chapter in the DateTimeAdapter class).

A Short Bit on Dependency Injection

The topic of Dependency Injection (DI) is quite a bit larger than space allows. There are many tools out there to facilitate DI across many technologies, and many approaches and techniques. For example, rather than using constructor injection, you can use property or method injection. For a good overview of DI, check out Martin Fowler’s article from a few years ago:

http://martinfowler.com/articles/injection.html

This book simply illustrates the happy path of DI, which is quite sufficient for building just about any RESTful service with MVC 4. However, this book assumes you can look up other sources for a more in-depth education on the subject.

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

private readonly IDateTime \_dateTime;

public TasksController(IDateTime dateTime)

{

\_dateTime = dateTime;

}

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

public interface IDateTime

{

DateTime UtcNow { get; }

}

public class DateTimeAdapter : IDateTime

{

public DateTime UtcNow

{

get { return DateTime.UtcNow; }

}

}

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

var task = new Task

{

// …

CreatedDate = \_dateTime.UtcNow

};

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

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

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

## Configuring Ninject Dependency Injection

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

There are three things you need to take care of regarding Ninject in the service. The first two need to be done in any kind of application, while the third is somewhat unique to an MVC 4 Web API service. Table 5-2 briefly describes each of these three activities.

Table 5-2. Three Ninject–related activities

|  |  |
| --- | --- |
| Activity | Description |
| Container configuration | Make sure a DI container is created during application start-up and remains in memory until the application shuts down. |
| Container bindings | This is where you link interfaces to concrete implementations, such as IDateTime⮞DateTimeAdapter. |
| IDependencyResolver for Ninject | This tells MVC 4 / Web API to ask for all dependencies. This is the key that allows you to push dependencies up to the constructor on the controllers. |

## Container Configuration

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

* Be created as one of the first things that happens when the application starts up
* Be available at all times while the application is running
* Be destroyed as one of the last steps the application takes during shutdown

While it is certainly possible to wire up Ninject manually, the easiest and most reliable option for making sure the container is always available is to simply install the Ninject.Web.Common NuGet package. If you were following the steps in Chapter 4, then you already did this. It handles creating and destroying a container instance—within those startup and shutdown methods. All of this happens within a class that gets added to the app\_start folder in the MVC project.

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

## Container Bindings

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

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

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

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

log4net.Config.XmlConfigurator.Configure();

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

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

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

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

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

///<summary>

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

///</summary>

public class NinjectConfigurator

{

///<summary>

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

/// container with all of this application's

/// dependencies. Also configures the container as this

/// application's dependency resolver.

///</summary>

public void Configure(IKernel container)

{

// Add all bindings/dependencies

AddBindings(container);

// Use the container and the NinjectDependencyResolver as

// application's resolver

var resolver = new NinjectDependencyResolver(container);

GlobalConfiguration.Configuration.DependencyResolver = resolver;

}

///<summary>

/// Add all bindings/dependencies to the container

///</summary>

private void AddBindings(IKernel container)

{

ConfigureNHibernate(container);

ConfigureLog4net(container);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

}

///<summary>

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

/// given container.

///</summary>

private void ConfigureLog4net(IKernel container)

{

log4net.Config.XmlConfigurator.Configure();

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

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

}

///<summary>

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

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

///</summary>

private IUserSession CreateUserSession(IContext arg)

{

return new UserSession(Thread.CurrentPrincipal as GenericPrincipal);

}

///<summary>

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

/// container.

///</summary>

private void ConfigureNHibernate(IKernel container)

{

// Build the NHibernate ISessionFactory object

var sessionFactory = FluentNHibernate

.Cfg.Fluently.Configure()

.Database(

MsSqlConfiguration.MsSql2008.ConnectionString(

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

.CurrentSessionContext("web")

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

.BuildSessionFactory();

// Add the ISessionFactory instance to the container

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

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

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

}

///<summary>

/// Method used to create instances of ISession objects

/// and bind them to the HTTP context.

///</summary>

private ISession CreateSession(IContext context)

{

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

if (!CurrentSessionContext.HasBind(sessionFactory))

{

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

var session = sessionFactory.OpenSession();

CurrentSessionContext.Bind(session);

}

return sessionFactory.GetCurrentSession();

}

}

As you build the code for the task-management service—and as the code evolves over time—you will find yourself coming back to this NinjectConfigurator class over and over. This is because the classes you use for various behaviors will continue to change as the application evolves. Simply put, these mappings are not etched in stone, and you should expect to massage them as time goes on.

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

## IDependencyResolver for Ninject

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

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

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

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

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

public class NinjectDependencyResolver : IDependencyResolver

{

private readonly IKernel \_container;

public IKernel Container

{

get { return \_container; }

}

public NinjectDependencyResolver(IKernel container)

{

\_container = container;

}

public object GetService(Type serviceType)

{

return \_container.TryGet(serviceType);

}

public IEnumerable<object> GetServices(Type serviceType)

{

return \_container.GetAll(serviceType);

}

public IDependencyScope BeginScope()

{

return this;

}

public void Dispose()

{

// noop

}

}

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

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

var resolver = new NinjectDependencyResolver(container);

GlobalConfiguration.Configuration.DependencyResolver = resolver;

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

# NHibernate Configuration and Mappings

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

## Database Configuration

As with any approach to data access, at some point you must tell the underlying framework how to connect to the database. And because NHibernate is database-provider agnostic, you must also tell it which provider you’re using—and even which version of which provider. This allows NHibernate to load the appropriate driver for communicating with the database and when dynamically generating the DML (Data Manipulation Language). For example, creating a SELECT statement in SQL Server will be a little different in some cases than creating a SELECT statement in Oracle or MySQL. Indeed, one of the advantages of using an Object Relational Mapper (ORM) like NHibernate is that you can—in theory—change database providers without having to change anything about your domain model or any code that uses it.

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

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

The Data project includes the entire domain model (i.e., all of the classes), as well as the repository interfaces. Neither of these is dependent on a specific database provider; that is, the domain model will be the same whether you’re working with SQL Server or Oracle.

The second project, named similar to Data.SqlServer or Data.Oracle, will contain the repository implementations and the domain model NHibernate mapping definitions. This approach is used because both of these will likely change when swapping out database providers.

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

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

var sessionFactory = FluentNHibernate

.Cfg.Fluently.Configure()

.Database(

MsSqlConfiguration.MsSql2008.ConnectionString(

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

.CurrentSessionContext("web")

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

.BuildSessionFactory();

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

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

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

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

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

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

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

## Model Mapping

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

## The Mapping Classes

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

public class CategoryMap : VersionedClassMap<User>

{

public CategoryMap()

{

Id(x => x.CategoryId);

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

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

}

}

public class PriorityMap : VersionedClassMap<User>

{

public PriorityMap()

{

Id(x => x.PriorityId);

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

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

}

}

public class StatusMap : VersionedClassMap<User>

{

public StatusMap()

{

Id(x => x.StatusId);

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

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

}

}

public class TaskMap : VersionedClassMap<User>

{

public TaskMap()

{

Id(x => x.TaskId);

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

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

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

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

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

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

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

HasManyToMany(x => x.Users)

.Access.ReadOnlyPropertyThroughCamelCaseField(Prefix.Underscore)

.Table("TaskUser")

.ParentKeyColumn("TaskId")

.ChildKeyColumn("UserId");

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

.Table("TaskCategory")

.ParentKeyColumn("TaskId")

.ChildKeyColumn("CategoryId");

}

}

public class UserMap : VersionedClassMap<User>

{

public UserMap()

{

Table("AllUsers");

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

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

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

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

}

}

The first thing you might notice is that all of the mapping code is contained within each class’s constructor. Second, notice the use of the VersionedClassMap<T> base class for each of the map classes. This custom class allows you to automatically take advantage of NHibernate’s ability to check for dirty records in the database, based on a Rowversion column on each table. The definition of this base class looks like this:

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

{

protected VersionedClassMap()

{

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

}

}

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

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

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

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

The two main ClassMap<T> methods used in the application’s mapping classes are the Id() and Map() methods. The Id() method can be called only once, and it’s used to tell NHibernate which property on the model class is used as the object identifier. This method also includes an overload to specify the table’s key column—if it happens to not match the property name given.

The Map() method is used in a similar fashion, although not for the object’s identifier. By default, NHibernate will assume the mapped column name is the same as the given property name. If it’s not, a different overload can be used to specify the column name. Additionally, because this is a fluent-style interface, you can chain other property and column specifics together. For example, the UserMap class’s Firstname mapping also includes a specification telling NHibernate to treat the column as not nullable.

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

## Project and File Organization

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

Figure 5-1. The domain model classes

Figure 5-2. The NHibernate mapping classes

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

## Model Relationships

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

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

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

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

public virtual IList<Category> Categories

{

get { return \_categories; }

}

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

.Access.ReadOnlyPropertyThroughCamelCaseField(Prefix.Underscore)

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

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

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

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

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

# Managing the Unit of Work

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

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

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

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

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

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

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

private ISession CreateSession(IContext context)

{

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

if (!CurrentSessionContext.HasBind(sessionFactory))

{

var session = sessionFactory.OpenSession();

CurrentSessionContext.Bind(session);

}

return sessionFactory.GetCurrentSession();

}

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

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

private readonly ISession \_session;

public UserRepository(ISession session)

{

\_session = session;

}

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

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

public class LoggingNHibernateSessionAttribute : ActionFilterAttribute

{

public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext)

{

EndTransaction(actionExecutedContext);

CloseSession();

LogException(actionExecutedContext);

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

}

private void CloseSession()

{

var container = GetContainer();

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

if (CurrentSessionContext.HasBind(sessionFactory))

{

var session = sessionFactory.GetCurrentSession();

session.Close();

session.Dispose();

CurrentSessionContext.Unbind(sessionFactory);

}

}

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

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

[LoggingNHibernateSession]

public class TasksController : ApiController

And then watch the magic happen!

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

# Database Transaction Control

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

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

public override void OnActionExecuting(HttpActionContext actionContext)

{

LogAction(actionContext.ActionDescriptor, "ENTERING");

BeginTransaction();

}

public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext)

{

EndTransaction(actionExecutedContext);

CloseSession();

LogException(actionExecutedContext);

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

}

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

public void BeginTransaction()

{

var session = GetCurrentSession();

if (session != null)

{

session.BeginTransaction();

}

}

public void EndTransaction(HttpActionExecutedContext filterContext)

{

var session = GetCurrentSession();

if (session != null)

{

if (session.Transaction.IsActive)

{

if (filterContext.Exception == null)

{

session.Flush();

session.Transaction.Commit();

}

else

{

session.Transaction.Rollback();

}

}

}

}

private ISession GetCurrentSession()

{

var container = GetContainer();

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

var session = sessionFactory.GetCurrentSession();

return session;

}

private IKernel GetContainer()

{

var resolver = GlobalConfiguration.Configuration.DependencyResolver as NinjectDependencyResolver;

if (resolver != null)

{

return resolver.Container;

}

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

}

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

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

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

# Summary

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

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

Caller makes a web request.

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

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

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

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

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

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

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

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

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