23

Data access with NHibernate

This chapter covers:

* Decoupling data access from the core and UI
* Configuring NHibernate mappings
* Bootstrapping NHibernate
* Invoking data access from ASP.NET MVC

Even though the ASP.NET MVC Framework is focused on the presentation layer, many developers work on small applications that do not need several layers of business logic and separation between the presentation layer and the data store. For these small applications, simpler separation patterns may be appropriate; however, many small applications grow much larger than originally anticipated. When this happens, separation of concerns is critical to the long-term maintainability of the software. NHibernate is a popular object-relational mapper. It makes data access with relational databases trivial. As with anything new, there is a learning curve associated with understanding the method of configuring the mapping between objects and tables. This chapter demonstrates how to configure and leverage NHibernate when developing an application whose UI takes advantage of the ASP.NET MVC Framework. This example is equally application in ASP.NET MVC 1 and 2.

23.1 Functional overview of reference implementation

Our reference implementation is on top of the default project template. The functionality that is added is the capability to track visits to the site. Each page tracks visitors. The site tracks the following pieces of data:

* URL
* Login name
* Browser
* Date and time
* IP Address

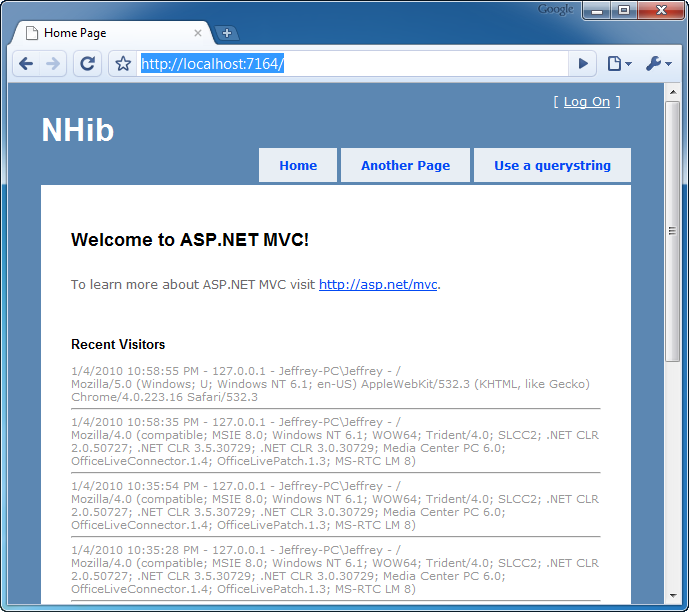
We see in figure 23.1 that by running the application the most recent visits are displayed at the bottom of the page. Each page displays the recent visits. 

Figure 23.1 Recent visitors are displayed at the bottom of every page.

We have intentionally kept the scope of this application small so we can focus on the usage of NHibernate as the data access library that allows us to persist and retrieve Visitor objects. Before we go into each layer of the application, let’s review the architecture of this application at a high level.

23.2 Application architecture overview

At a broad level, this application uses DDD inside an Onion Architecture. At a high level, the application is composed of a domain model at its core. Figure 23.2 shows a reference layout of Onion Architecture.

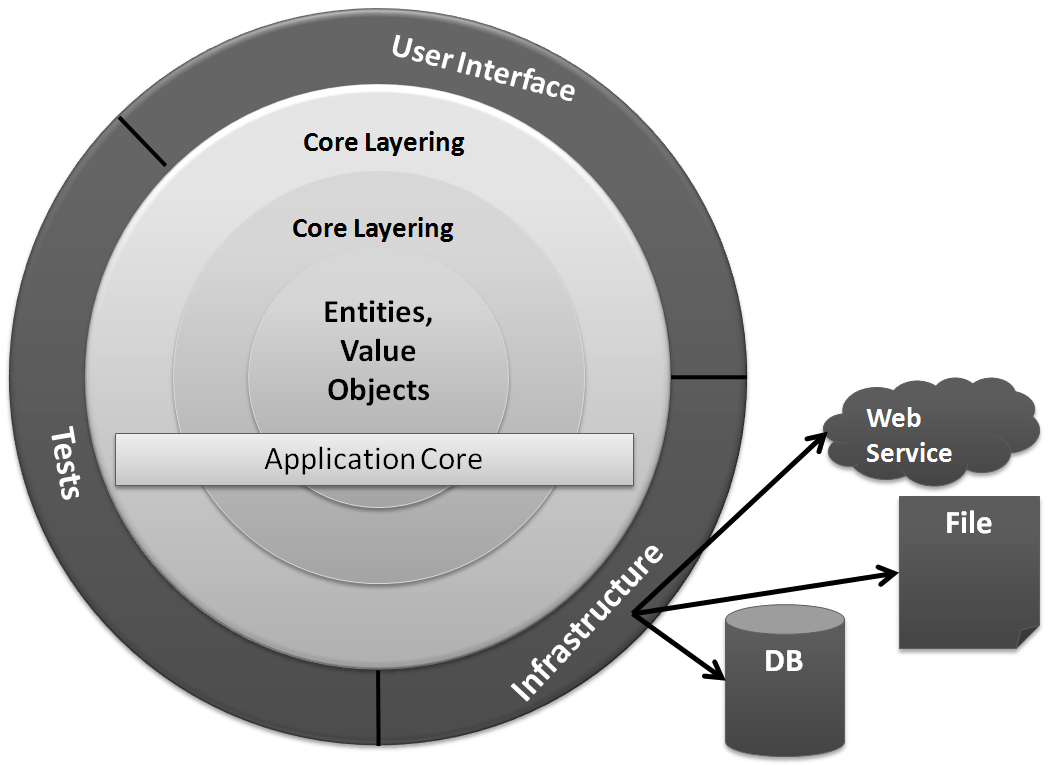


Figure 23.2 The Onion Architecture uses the concept of an application core that doesn’t depend on external libraries, such as NHibernate.

The solution structure implements the decoupling strategy that Onion Architecture requires. In figure 23.3, you can see the solution structure with the Core project’s references expanded. This application has a simple core, and the libraries referenced to implement the core are straightforward. Notice that there is no reference to NHibernate.dll from the Core project. It’s important that the Core remain portable and not coupled to external libraries that will change over time. As with everything in software, this is a trade-off. You may feel comfortable coupling to some libraries, but evaluate the consequences carefully. This example employs the Inversion of Control principle through abstract factories and dependency injection.

Inversion of Control is a principle, not a tool

With the popularity of IoC containers, many developers are not aware of how to implement Inversion of Control without a library like StructureMap. Many developers have experience with dependency injection, but only by the use of an IoC container.

This example employs Inversion of Control through liberal use of dependency injection via constructor injection. The decoupling mechanism employs the abstract factory pattern with start-up time bootstrapping code to initialize the abstract factories.

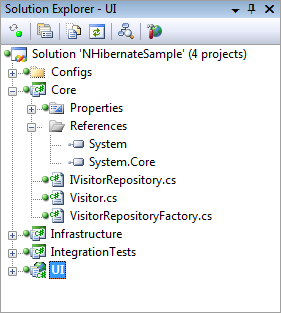


Figure 23.3 The Core project has minimal references and no external dependencies.

If we expand more of the projects, we see that no project references the Infrastructure project except for IntegrationTests, which is not deployed to production anyway. Figure 23.4 shows the solution fully expanded. Only the Infrastructure project references NHibernate.dll; no project references Infrastructure. When we examine the UI project, we’ll see how the application is organized at runtime to function properly. Note that this example is not focused on automated testing, so many of the necessary automated tests are omitted for the sake of brevity.

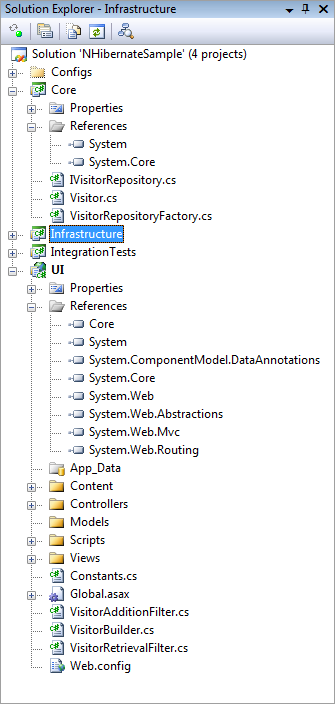


Figure 23.4 No project references Infrastructure. This arrangement is important for decoupling.

Now that we understand how the application is structured at a high level, we’ll explore each layer bit by bit. We’ll begin with the domain model.

23.3 Domain model–the application core

The domain model is the most important part of the application. Without the domain model, all of the pertinent concepts would be represented only in the UI. Our particular domain model contains a single aggregate made up of a single entity, the Visitor. The code for the Visitor class is shown in listing 23.1.

Listing 23.1 Visitor class is the domain model for this example

using System;

namespace Core

{

public class Visitor

{

public virtual Guid Id { get; set; }

public virtual string PathAndQuerystring { get; set; }

public virtual string LoginName { get; set; }

public virtual string Browser { get; set; }

public virtual DateTime VisitDate { get; set; }

public virtual string IpAddress { get; set; }

}

}

We have no business logic here, and at first glance it looks just like a data structure. All other concerns have been left out in an effort to include only abstractions and logic that are necessary for leveraging NHibernate in a loosely coupled way.

The Visitor class contains properties for all of the pieces of information that we want to record. The Id property exists as an identifier for the particular visit. We could certainly use Int32 as the ID, but in a data persistence environment, that forces a dependency on the data store for the generation of a unique Int32 value. Sometimes this is appropriate, but in DDD, the developer errs on the side of giving responsibility to the domain model, not the data store. In line with that, the Id is a Guid, and the application will generate a Guid before attempting to save to the database.

The mechanism for persisting or retrieving a Visitor is called a repository. The repository will save our entity as well as retrieve it. It can also represent filtering operations. In our domain model, we have an IVisitorRepository. This interface is seen in listing 23.2

Listing 23.2 The repository defines the persistence operations

namespace Core

{

public interface IVisitorRepository

{

void Save(Visitor visitor);

Visitor[] GetRecentVisitors(int numberOfVisitors);

}

}

With our repository, we are able to save a visitor as well as get the most recent visitors. We can ask for a specific number of recent visitors. In figure 23.4, you see that the Core project doesn’t contain any class that implements IVisitorRepository. This is important because the class that actually does the work represented by the interface will be responsible for the persistence, which is not a domain model concern. Persistence is infrastructure. I could imagine that this functionality would work equally well if I persisted the data to a file instead of the database. The mechanism of persistence is not a concern for the domain model; therefore, the class responsible for it is not in the Core project.

The concern that is in the Core project, however, is an abstract factory that is capable of locating or creating an instance of IVisitorRepository. The VisitorRepositoryFactory is responsible for returning an instance of our repository. Listing 23.3 shows that the knowledge of how to create the repository doesn’t reside with the factory. This factory merely represents the capability to return the repository.

Listing 23.3 Factory offers capability to get repository

using System;

namespace Core

{

public class VisitorRepositoryFactory

{

public static Func<IVisitorRepository> RepositoryBuilder = |#A

CreateDefaultRepositoryBuilder; |#A

private static IVisitorRepository CreateDefaultRepositoryBuilder()

{

throw new Exception("No repository builder specified."); #B

}

public IVisitorRepository BuildRepository()

{

IVisitorRepository repository = RepositoryBuilder(); #C

return repository;

}

}

}

#A Initialize at application startup

#B Throw if factory not initialized

#C Use delegate to build repository

To even the inexperienced eye, this class doesn’t seem useful alone. When BuildFactory() is called, an exception will be thrown. Out of the box, the domain model doesn’t know the implementation of IVisitorRepository that will be used, so there is no way to embed this knowledge into compiled code. The public static RepositoryBuilder property will have to be set to something useful before the factory will work properly. We’ll see how this is accomplished after all the pieces have been introduced.

This explicit factory is not necessary if you’re using an Inversion of Control container, which has been left out for the sake of simplicity. This domain model is simple. This is all there is. The next step is to understand how we configure NHibernate to automatically persist our entity to the database.

23.4 NHibernate configuration–infrastructure of the application

There is little code to write in order to leverage NHibernate for seamless persistence. NHibernate is a library, not a framework; the difference is important. Frameworks provide templates of code in which you then fill in the gaps to create something useful. Libraries are usable without providing templates. NHibernate doesn’t require your entities to derive from a specific base class or the implementation of a specific interface. NHibernate can persist any type of object as long as the configuration is correct.

This section will walk through the configuration of NHibernate so that we can save and retrieve the Visitor object. For this chapter, we are using NHibernate 2.1 with FluentNHibernate 1.0 for configuration help.

Before we dive into the configuration, let’s examine the implementation of the IVisitorRepository interface specified in the domain model. The purpose for starting with this class is to demonstrate how little code is actually written when calling NHibernate to perform a persistence operation. Listing 23.4 shows the VisitorRepository class located in the Infrastructure project.

Listing 23.4 Repository implementation coupled to NHibernate APIs

using System.Collections.Generic;

using System.Linq;

using Core;

using NHibernate;

namespace Infrastructure

{

public class VisitorRepository : IVisitorRepository

{

public void Save(Visitor visitor)

{

ISession session = GetSession();

session.SaveOrUpdate(visitor);

}

public Visitor[] GetRecentVisitors(int numberOfVisitors)

{

IList<Visitor> visitors = GetSession()

.CreateQuery( #A

"select v from Visitor v order by v.VisitDate desc"

).SetMaxResults(numberOfVisitors)

.List<Visitor>();

return visitors.ToArray(); #B

}

private ISession GetSession()

{

var cache = new SessionCache(); |#C

ISession session = cache.GetSession(); |#C

return session;

}

}

}

#A Use HQL to select Visitors

#B Return array of Visitors

#C Retrieve session from cache

This class is a total of 34 lines long, and many lines are largely white space. The code that leverages the NHibernate APIs is limited. Now that we see what it looks like to call NHibernate, we’ll walk through the configuration process of NHibernate and explore each step. We’ll start with the main configuration.

23.4.1 Like any .Net application NHibernate needs configuration

The beginning of the configuration process is the hibernate.cfg.xml file. This file is the same name as that used by the Hibernate library in Java. Because NHibernate started as a port from Hibernate, this is just one of the many similarities. Knowledge of one largely translates directly to the other.

The contents of the hibernate.cfg.xml file can also be put into the web.config file or app.config file. For simple applications, embedding this information into the .NET configuration file may be adequate; we should emphasize that this example stresses separation so that when applied to a medium-sized application, the code and configuration don’t run together. We have seen web.config files grow large, and it’s trivial to store the NHibernate configuration in a dedicated file. Listing 23.5 shows the contents of the hibernate.cfg.xml file.

Listing 23.5 Hibernate.cfg.xml file contains database connection information

<hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">

<session-factory>

<property name="connection.driver\_class"> #2

NHibernate.Driver.SqlClientDriver

</property>

<property name="connection.connection\_string"> #1

server=.\SQLExpress;database=NHibernateSample;

Integrated Security=true;

</property>

<property name="show\_sql">false</property>

<property name="dialect"> #3

NHibernate.Dialect.MsSql2005Dialect

</property>

<property name="adonet.batch\_size">100</property>

<property name="proxyfactory.factory\_class"> #4

NHibernate.ByteCode.LinFu.ProxyFactoryFactory,

NHibernate.ByteCode.LinFu

</property>

</session-factory>

</hibernate-configuration>

This is a simple configuration, and there are many other options documented with the NHibernate documentation (<http://nhforge.org/doc/nh/en/index.html>). The most obvious piece of information is the connection string **(1)**. Also, the driver class **(2)** and dialect **(3)** specify the details of the database engine used. This sample uses SQL Server 2005, but these values would change if you wanted to use any version of Oracle, SQLite, or the many other database engines supported out of the box.

The show\_sql property will output each SQL query to the Console as the statement is being sent to the database. This is useful for debugging. The adonet.batch\_size controls how many updates, deletes, or inserts will be sent to the database in a single batch. It’s more efficient to send multiple statements in a single network call than to make a separate network call for each statement. NHibernate will do this automatically. The last **(4)** configuration item is the proxy factory to use for mappings using lazy loading, which is the default. If we were using XML mapping files, we would also configure the assembly in which NHibernate could find the embedded mappings, but that is not necessary here since we are using code-based mappings with FluentNHibernate.

23.4.2 The NHibernate mapping–simple but powerful

NHibernate requires at least one mapping. Figure 23.5 shows the Infrastructure project. Here, you can see that there is a code file named VisitorMap.cs.

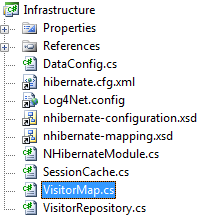


Figure 23.5 The Infrastructure project contains the NHibernate mapping for Visitor.

We are about to explore the VisitorMap.cs file, which contains the mapping information for the Visitor class. First, notice the four files that are linked into the project:

* Hibernate.cfg.xml
* Log4Net.config
* Nhibernate-configuration.xsd
* Nhibernate-mapping.xsd

These files do not belong to the project directly; they are linked from elsewhere. We do this because multiple projects need the same copy of these files. The first example that needs linked files is the IntegrationTests. It will contain tests for all data access. To test the data access, the tests need to leverage the same configuration as the application.

We have already covered the hibernate.cfg.xml file. The Log4Net.config file contains Log4Net configuration information that is broadly applicable to any type of application. If you’re not familiar with Log4Net, you can find more information at <http://logging.apache.org/log4net/index.html>. The two files provide the schema for the NHibernate configuration and the NHibernate mapping files. When added to the project, they enable Visual Studio to provide XML IntelliSense when you are editing these files. It makes the editing process smooth. In larger applications, you will have a mix of code-based mappings and XML mappings. Because the XML mappings are the most comprehensive and documented, you will have to use them in some situations. Without this XML IntelliSense, it would be cumbersome to maintain these XML files.

Let’s now turn to the mapping for the Visitor class. We’ll open the VisitorMap.cs file and examine its structure as shown in listing 23.6.

Listing 23.6 VisitorMap.cs file contains mapping for the Visitor class

using Core;

using FluentNHibernate.Mapping;

namespace Infrastructure

{

public class VisitorMap : ClassMap<Visitor>

{

public VisitorMap()

{

Table("Visitor"); #1

DynamicUpdate();

Id(x => x.Id).GeneratedBy.GuidComb(); #2

Map(x => x.PathAndQuerystring).Length(4000).Not.Nullable();

Map(x => x.LoginName).Length(255).Not.Nullable();

Map(x => x.Browser).Length(4000).Not.Nullable();

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

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

}

}

}

/\*<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"

namespace="Core"

assembly="Core">

<class name="Visitor" table="Visitors"

dynamic-update="true">

<id name="Id" column="Id" type="Guid">

<generator class="guid.comb"/>

</id>

<property name="PathAndQuerystring" length="4000"

not-null="true"/>

<property name="LoginName" length="255" not-null="true"/>

<property name="Browser" length="4000" not-null="true"/>

<property name="VisitDate" not-null="true"/>

<property name="IpAddress" not-null="true"/>

</class>

</hibernate-mapping>

\*/

The first line **(1)** is pretty standard and specifies the table to use. The Id method **(2)** is special, and it has to be the first property mapped on an entity. This will become the primary key on the table, and there are many ways to handle it. The generator node has many options, including SQL Server “identity” and Oracle “sequence” functionality. We want the object to have a value in the Id property before being persisted, so we are configuring NHibernate to generate a Guid for us before issuing the INSERT statement to the database. The GuidComb() generator is special, and it generates GUIDs in sequential order so that the clustered index on the primary key column has little to do when absorbing a new record inserted into the table. This sequencing sacrifices a bit of uniqueness in the GUID algorithm, but in this context, the only thing that is important is that the GUID be unique for this particular table. You can read more about the COMB GUID from the inventor, Jimmy Nilsson: <http://www.informit.com/articles/article.aspx?p=25862>.

The rest of the properties are largely self-explanatory. They have names, constraints, and the strings can have a length specified. If you’re all right with the column name being the same as the property name on the class, then a column attribute is unnecessary. When you have all the properties mapped, you’re ready to move on. If you have a more complex class structure, you will want to review all your mapping options in the documentation at: <http://nhforge.org/doc/nh/en/index.html> and <http://fluentnhibernate.org/>. The equivalent XML mapping is included in commented for for reference.

23.4.3 Initializing the configuration

There are two main abstractions in NHibernate: ISessionFactory and ISession. A session factory creates a session. A session is meant to be used for a single transaction. You should use and then quickly dispose of NHibernate sessions. The session factory is intended to be kept for the life of the application so that it can be used to create all sessions. The interface is the abstraction, but the implementation provided by NHibernate requires some understanding. The code in listing 23.7 shows how to create the session factory that will be used for the life of the application.

Listing 23.7 A Configuration object creates a session factory

public class DataConfig

{

public static ISessionFactory SessionFactory;

public void PerformStartup()

{

InitializeLog4Net();

InitializeNHibernateSessionFactory();

InitializeRepositories();

}

private void InitializeNHibernateSessionFactory()

{

Configuration configuration = |#A

BuildConfiguration(); |#A

SessionFactory = |#B

configuration.BuildSessionFactory(); |#B

}

public static Configuration BuildConfiguration()

{

Return |#C

Fluently.Configure( |#C

new Configuration().Configure()) |#C

.Mappings( |#C

cfg => |#C

cfg.FluentMappings.AddFromAssembly( |#C

typeof (VisitorMap).Assembly)) |#C

.BuildConfiguration(); |#C

}

private void InitializeLog4Net()

{

string configPath = Path.Combine(

AppDomain.CurrentDomain.BaseDirectory,

"Log4Net.config");

var fileInfo = new FileInfo(configPath);

XmlConfigurator.ConfigureAndWatch(fileInfo);

}

private void InitializeRepositories()

{

Func<IVisitorRepository> builder =

() => new VisitorRepository();

VisitorRepositoryFactory.RepositoryBuilder =

builder;

}

public void StartSession()

{

ISession session = SessionFactory.OpenSession();

session.BeginTransaction();

var cache = new SessionCache();

cache.CacheSession(session);

}

public void EndSession()

{

var cache = new SessionCache();

ISession session = cache.GetSession();

ITransaction transaction = session.Transaction;

if (transaction.IsActive)

{

transaction.Commit();

}

session.Dispose();

}

}

#A Configure NHibernate using XML configuration

#B Build and cache session factory

#C Apply Fluent Nhibernate mappings

The session factory is expensive to create. By expensive, we mean that it accesses the file system and parses XML from embedded resources inside DLLs. The configuration object is going to read the hibernate.cfg.xml file (out-of-process call), and then it will build the session factory using this configuration. When building the session factory, it will apply all the properties found in the configuration file. If an assembly was included for embedded XML mappings, it will retrieve all those mapping files from within the DLLs (out-of-process call). Each mapping file will be parsed using the XML DOM. Regardless if you use code mappings or XML mappings, NHibernate will use reflection on all the types to ensure that every property declared in the mapping actually exists on the types referenced. If lazy loading is enabled (the default), it will also check that all public properties and methods are marked as virtual. If you prefer not to mark them virtual, disable lazy loading. With most applications, it takes at least a full second (or more) to create the session factory; this operation is not something you want to do often. If you create the session factory for every web request, your web application will slow down dramatically. We push the session factory instance in a static variable so we can hold on to it for the life of the application.

The NHibernate session, on the other hand, is cheap. We’ll create and destroy many of these objects. In a stateful application, we’ll use a session for a single transaction or user operation. For a web application, we’ll use one session per web request. We’ll cover the web application usage is just a bit. The code for the creation of a session is shown in listing 23.8.

Listing 23.8 The session is inexpensive to create

ISession session = SessionFactory.OpenSession(); #A

#A Session factory provides the session

Before we can move on to the code that uses all this, we have to have a database. We have declared our connection string, and with the mapping, NHibernate knows the table structure. We can proceed to create our database schema manually, or we can get NHibernate to help us out. After creating an empty database named “NHibernateSample” inside SQL Server Express, as declared by the connection string, we can execute the code shown in listing 23.9 to have NHibernate create our schema.

Listing 23.9 NHibernate generates database from mappings

using Infrastructure;

using NHibernate.Tool.hbm2ddl;

using NUnit.Framework;

namespace IntegrationTests

{

[TestFixture]

public class DatabaseTester

{

[Test, Explicit]

public void CreateDatabaseSchmea()

{

var export = new SchemaExport(

DataConfig.BuildConfiguration());

export.Execute(true, true, false);

}

}

}

We are using an NUnit test fixture as an easy launching point for this code. It makes it trivial to run the code snippet. After running this test inside Visual Studio using the TestDriven.net add-in (<http://testdriven.net/>), we’ll see the output in the Output windows similar to listing 23.10.

Listing 23.10 Output from schema export shows table DDL

------ Test started: Assembly: IntegrationTests.dll ------

if exists (select \* from dbo.sysobjects where id = object\_id(N'Visitors') and OBJECTPROPERTY(id, N'IsUserTable') = 1) drop table Visitors

create table Visitors (

Id UNIQUEIDENTIFIER not null,

PathAndQuerystring NVARCHAR(4000) not null,

LoginName NVARCHAR(255) not null,

Browser NVARCHAR(4000) not null,

VisitDate DATETIME not null,

IpAddress NVARCHAR(255) not null,

primary key (Id)

)

1 passed, 0 failed, 0 skipped, took 6.86 seconds.

The NUnit test lives in the IntegrationTests project. This project also links in the hibernate.cfg.xml file to leverage the same configuration. Figure 23.6 shows the IntegrationTests project structure. We have kept it minimal for the sake of simplicity.

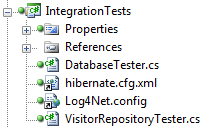


Figure 23.6 The IntegrationTests project contains tests for all the mappings and repositories.

Notice the VisitorRepositoryTester class. You can probably guess what this class does. That's right! It contains the automated testing necessary to ensure that the repository implementation functions as expected. We cannot write unit tests for data access. Data access, by its very nature, is an integration test concern. Not only are we integrating a third-party library, NHibernate, but we are also expecting another process to be running on our network, server, or workstation. SQL Server must be up and running. It also must contain the correct schema. If anything is wrong along the way, the tests will fail. Because of this arrangement, these integration tests are larger than we would expect for simple unit tests. Even so, keep them as small as possible, and only test the data access. Listing 23.11 shows the code for the VisitorRepositoryTester.

Listing 23.11 Integration test verifies mappings and database are correct

using System;

using System.Collections.Generic;

using System.Linq;

using Core;

using Infrastructure;

using NHibernate;

using NUnit.Framework;

using NUnit.Framework.SyntaxHelpers;

namespace IntegrationTests

{

[TestFixture]

public class VisitorRepositoryTester

{

[SetUp]

public void Setup()

{

new DatabaseTester().CreateDatabaseSchmea();

}

[Test]

public void When\_saving\_should\_write\_to\_database()

{

var config = new DataConfig(); #A

config.PerformStartup();

config.StartSession();

var visitor = new Visitor #B

{

Browser = "1",

IpAddress = "2",

LoginName = "3",

PathAndQuerystring = "4",

VisitDate =

new DateTime(2000, 1, 1)

};

var repository = new VisitorRepository();

repository.Save(visitor); #C

config.EndSession(); |#D

config.StartSession(); |#D

ISession session = new SessionCache().GetSession(); #E

var loadedVisitor = session.Load<Visitor>(visitor.Id);

Assert.That(loadedVisitor, Is.Not.Null); |#F

Assert.That(loadedVisitor.Browser, Is.EqualTo("1")); |#F

Assert.That(loadedVisitor.IpAddress, Is.EqualTo("2")); |#F

Assert.That(loadedVisitor.LoginName, Is.EqualTo("3")); |#F

Assert.That(loadedVisitor.PathAndQuerystring, |#F

Is.EqualTo("4")); |#F

Assert.That(loadedVisitor.VisitDate, |#F

Is.EqualTo(new DateTime(2000, 1, 1))); |#F

}

[Test]

public void Should\_get\_two\_most\_recent\_visitors()

{

var config = new DataConfig();

config.PerformStartup();

Visitor visitor1 =

CreateVisitor(new DateTime(2000, 1, 1));

Visitor visitor2 =

CreateVisitor(new DateTime(2000, 1, 2));

Visitor visitor3 =

CreateVisitor(new DateTime(2000, 1, 3));

config.StartSession();

using (

ISession session1 =

new SessionCache().GetSession())

{

session1.SaveOrUpdate(visitor1);

session1.SaveOrUpdate(visitor2);

session1.SaveOrUpdate(visitor3);

session1.Flush();

config.EndSession();

}

config.StartSession();

var repository = new VisitorRepository();

Visitor[] recentVisitors =

repository.GetRecentVisitors(2);

config.EndSession();

Assert.That(recentVisitors.Length, Is.EqualTo(2));

IEnumerable<Guid> idList =

recentVisitors.Select(x => x.Id);

Assert.That(idList.Contains(visitor3.Id), Is.True);

Assert.That(idList.Contains(visitor2.Id), Is.True);

Assert.That(idList.Contains(visitor1.Id), Is.False);

}

private Visitor CreateVisitor(DateTime visitDate)

{

return new Visitor

{

Browser = "1",

IpAddress = "2",

LoginName = "3",

PathAndQuerystring = "4",

VisitDate = visitDate

};

}

}

}

#A Configure NHibernate

#B Create new Visitor

#C Save Visitor

#D Create new session

#E Reload Visitor

#F Assert correct data

These tests are essential to ensuring that every query generated by NHibernate is tested and retested with every build. Because configuration changes will change the queries that are generated, tests are important for the stability of the application. When we run this test, we see that it passes as shown in figure 23.7.

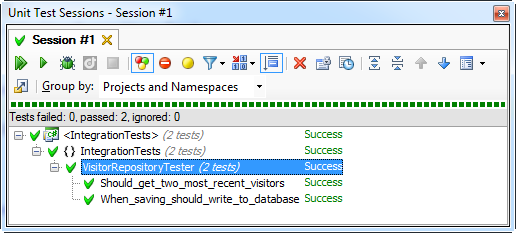


Figure 23.7 When the repository test passes, we know the mapping is correct (ReSharper test runner)

You now know the basics of persisting with NHibernate. All NHibernate coupling should remain in the Infrastructure project. Remember that none of the other projects have a reference to Infrastructure, so the rest of the code is not coupled to this particular data access library. This decoupling is important because data access methods change very frequently. You do not want to couple your application to infrastructural concerns when they are likely to change frequently.

NOTE

Some of the authors of this book have been using NHibernate since version 0.8 with .Net 1.1, but with object-relational mappers becoming mainstream, the writing is on the wall that other, more well-funded ORM libraries will supersede NHibernate in a few years. With that as a likely event, it is important to keep the core of the application from creating a coupling to this particular library.

Now that we have covered both the Core and Infrastructure, we’ll see how this ties together in the UI.

23.5 UI leverages domain model

Now that the domain model and the NHibernate infrastructure are set up and functioning, we can turn our attention once again to the ASP.NET MVC project. We have left the project very close to the default project template in an effort to keep it simple as well as call out the additions necessary to enable the saving of every visitor to the site. Figure 23.8 shows the structure of the UI project.

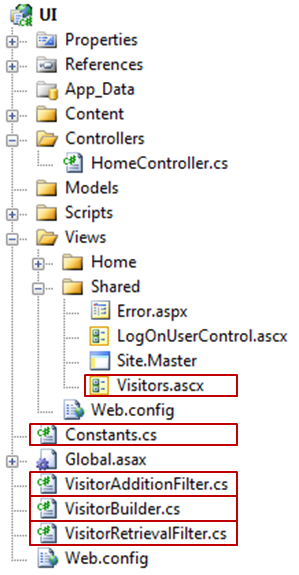


Figure 23.8 The additions to the project are boxed. We have added several files to support the capture and display of visitors.

From figure 23.1 you can recall that each page on the site shows the most recent visitors to the site at the bottom. To share this view on each page, we have wired up a partial view to the master page, Site.Master. We have covered this capability in previous chapters, so we won’t cover it in depth here.

At the highest level, we have added an action filter attribute to each controller. If the site contains many controllers, we would consider introducing a custom ControllerActionInvoker for all controllers and adding the filter for all controllers. In this example, the project contains only the HomeController. The HomeController is show in listing 23.12. Notice the action filters applied at the class level.

Cueballs in code and text

Listing 23.12 Action filters applied to controller to keep concerns separated

using System.Web.Mvc;

namespace UI.Controllers

{

[HandleError]

[VisitorAdditionFilter(Order = 0)] #1

[VisitorRetrievalFilter(Order = 1)] #2

public class HomeController : Controller

{

public ActionResult Index()

{

ViewData["Message"] = "Welcome to ASP.NET MVC!";

return View();

}

public ActionResult About()

{

return View();

}

}

}

We have introduced two filters, the VisitorAdditionFilter (1), and the VisitorRetrievalFilter (2). We have applied the Order optional parameter to ensure that they are executed in the intended order. It may be confusing, but the order the attributes are applied to the class is not the execution order. Officially, the order is not guaranteed to be deterministic.

We want to persist a new visitor before we get the list of recent visitors and pass the objects to a view. Listing 23.13 shows both of the action filters.

Cueballs in code and text. Note there are two 1s and two 2s widely separated.

Listing 23.13 Action filters interact with domain model.

using System.Web.Mvc;

using Core;

namespace UI

{

public class VisitorAdditionFilter : ActionFilterAttribute

{

private readonly IVisitorRepository \_repository;

public VisitorAdditionFilter(IVisitorRepository repository)

{

\_repository = repository;

}

public VisitorAdditionFilter() :

this(new VisitorRepositoryFactory().BuildRepository()) #1

{

}

public override void OnResultExecuting( #2

ResultExecutingContext filterContext)

{

var builder = new VisitorBuilder();

Visitor visitor = builder.BuildVisitor(); |#3

\_repository.Save(visitor); |#3

}

}

}

using System.Web.Mvc;

using Core;

namespace UI

{

public class VisitorRetrievalFilter : ActionFilterAttribute

{

private readonly IVisitorRepository \_repository;

public VisitorRetrievalFilter(IVisitorRepository repository)

{

\_repository = repository;

}

public VisitorRetrievalFilter() : this(

new VisitorRepositoryFactory().BuildRepository()) #1

{

}

public override void OnResultExecuting( #2

ResultExecutingContext filterContext)

{

Visitor[] visitors = \_repository.GetRecentVisitors(10); |#4

filterContext.Controller |#4

.ViewData[Constants.ViewData.VISITORS] = visitors; |#4

}

}

}

Each of the filters is simple. Most of the code is just for managing the dependency of the IVisitorRepository and building the repository from the factory (1). The three lines of work that are interesting are in the OnResultExecuting method (2). We build the visitor and save it (3). Then we get the recent visitors and push them into view data (4). The VisitorBuilder class is not shown, but it’s a simple one that constructs a Visitor and populates it with information from the HttpRequest. The next interesting file is the Visitors.ascx partial view located in /Views/Shared/Visitors.ascx. Listing 23.14 shows this partial view.

Listing 23.14 A partial view is strongly-typed and displays recent visitors

<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<Visitor[]>" %>

<%@ Import Namespace="Core"%>

<div style="text-align:left">

<h3>Recent Visitors</h3>

<%foreach (var visitor in ViewData.Model){%>

<%=visitor.VisitDate%> -

<%=visitor.IpAddress%> -

<%=visitor.LoginName%> -

<%=visitor.PathAndQuerystring%><br />

<%=visitor.Browser%><hr />

<%}%>

</div>

This partial is added to the page via the master page. The array of visitors is expected to be in ViewData.Model so that the array can be rendered the default way. At the bottom of the master page, the following code passes just the visitor array to the partial: <%Html.RenderPartial(Constants.Partials.VISITORS, ViewData[Constants.ViewData.VISITORS]); %>.

We use the constants so that the views do not contain duplicate string literals. Since logging and displaying visitor information are cross-cutting concerns for the application, we have taken steps to keep the logic factored out so that it can be shared across all controllers in the application. Let's review what we have done:

* Kept the persistence logic behind an interface that doesn’t belong to the UI project
* Leveraged action filters so that no single controller is responsible for knowing how to interact with IVisitorRepository
* Created a partial view to own the layout of the recent visitors
* Delegated to the partial view from the master page so that individual views don’t have to care about it

All the pieces are now in place to be pulled together.

23.6 Pulling it together

If you have been keeping a close eye on the code up to this point, you have noticed that we do not have a default way to create the NHibernate repository instance of IVisitorRepository that lives in the Infrastructure project. Our UI project doesn’t reference the Infrastructure project at all. This section walks through the process of wiring up these decoupled pieces.

The first piece is in the web.config file. Inside the httpModules node, we have registered an extra module:

<add name="StartupModule" type="Infrastructure.NHibernateModule, Infrastructure, Version=1.0.0.0, Culture=neutral"/>

This module kicks off the process of creating the session factory. It also hooks the BeginRequest and EndRequest events and creates and destroys NHibernate sessions for each web request. Listing 23.15 shows the code for NHibernateModule.cs, which lives in the Infrastructure project.

Listing 23.15 NHibernateModule kick-starts NHibernate

using System;

using System.Web;

namespace Infrastructure

{

public class NHibernateModule : IHttpModule

{

private static bool \_startupComplete = false;

private static readonly object \_locker = new object();

public void Init(HttpApplication context)

{

context.BeginRequest += context\_BeginRequest;

context.EndRequest += context\_EndRequest;

}

private void context\_BeginRequest(object sender, EventArgs e)

{

EnsureStartup();

new DataConfig().StartSession(); #A

}

private void context\_EndRequest(object sender, EventArgs e)

{

new DataConfig().EndSession(); #B

}

private void EnsureStartup()

{

if (!\_startupComplete)

{

lock (\_locker)

{

if (!\_startupComplete)

{

new DataConfig().PerformStartup();

\_startupComplete = true;

}

}

}

}

public void Dispose()

{

}

}

}

#A Open session when request starts

#B End session when request ends

The DataConfig class (listing 23.7) will create a session and store it in the cache. Listing 23.16 shows the SessionCache.cs file as well as an important method from DataConfig.cs.

Listing 23.16 Session cache keeps session in HttpContext items

using System.Collections;

using System.Web;

using NHibernate;

namespace Infrastructure

{

public class SessionCache

{

private const string SESSION\_KEY = "NHIBERNATE\_SESSION";

private static readonly IDictionary \_cacheStore = new Hashtable();

public ISession GetSession()

{

var session = (ISession) GetCacheStore()[SESSION\_KEY];

return session;

}

public void CacheSession(ISession session)

{

GetCacheStore()[SESSION\_KEY] = session;

}

private static IDictionary GetCacheStore()

{

if (HttpContext.Current != null)

return HttpContext.Current.Items;

return \_cacheStore;

}

}

}

//DataConfig.cs

…

private void InitializeRepositories() #A

{

Func<IVisitorRepository> builder =

() => new VisitorRepository();

VisitorRepositoryFactory.RepositoryBuilder = builder;

}

…

#A Part of DataConfig.cs

Now that we have a session factory, and we have a session, our application can call NHibernate and communicate with the database. Aside from the NHibernate initialization, we have the initialization of the VisitorRepositoryFactory. Many applications use StructureMap as an IoC container. IoC tools provide these factories automatically, but because this sample doesn’t leverage the IoC container, we had to provide this startup logic explicitly. There are several ways to do that. Another popular way is to declare an interface for the factory and keep an implementation around. Use your judgment when choosing a technique. The important thing is that neither the Core project nor the UI project should reference the Infrastructure project or libraries that are purely infrastructural in nature. We have kept NHibernate completely off to the side so that the rest of the application doesn’t care how the data access is happening.

There is one final missing piece required before we can run this application from Visual Studio using CTRL+F5. The web.config file refers to a class in the Infrastructure project, but because there is no reference, the Infrastructure assembly will not be in the bin folder of the website. We could copy it explicitly every time we compile, but that will get tiresome. The solution is to have Visual Studio copy it every time it’s compiled by adding the following lines to the Infrastructure.csproj file as a post-build event such as the command in listing 23.17.

Listing 23.17 Post-build event copies assemblies and config files

xcopy /y ".\\*.dll" "..\..\..\UI\bin\"

xcopy /y ".\\*.dll" "..\..\..\IntegrationTests\bin\$(ConfigurationName)"

xcopy /y ".\log4net.config" "..\..\..\UI\"

xcopy /y ".\hibernate.cfg.xml" "..\..\..\UI\bin\"

By setting up the four commands shown in listing 23.17, we have configured the Infrastructure project to copy two important configuration files as well as the necessary binaries to the UI project’s bin folder and the test folder. Not only will the Infrastructure assembly be copied, the NHibernate assemblies will be copied as well. This ensures that when the UI project is run from Visual Studio, you will be greeted with a running application that is saving and showing visitors as in figure 23.9.



Figure 23.9 The application works as expected after being wired together.

Because of this post-build step, the application has all the required assemblies and configuration files. This reduces the pain of copying these files manually every time they change. This is just one type of automation required when you truly commit to decoupling your applications.

23.7 Summary

In this chapter, you have seen how to structure your solution, configure NHibernate, use the repository pattern from domain-driven design, and wire up loosely coupled code at runtime. This chapter presents a vastly simplified example, but the de-coupling patterns contained within are appropriate in medium to large applications as well.

Configuring and using NHibernate is easy. It’s also easy to couple to it and get in trouble. Whether it’s NHibernate or any other data access library, do not couple your application to it. Keep your core clean and your UI separated. All data access should be behind abstractions and tested separately. For more advanced usages of NHibernate with ASP.NET MVC, you can open the CodeCampServer solution, which is included with this book’s code download.

Now that you understand all of the concepts in ASP.NET MVC as well as how to tie it together into a full application with a database, you will move on to Part 4, which will dive into more cross-cutting topics such as route debugging, customizing Visual Studio, overall testing practices.