Configuring NHibernate with ASP.NET

A step-by-step tutorial on how to setup NHibernate with ASP.NET

Introduction

Object/Relational Mappers (OR/Ms) are getting increasing attention by .NET developers. By encapsulating many patterns that span from data access to data mapping and entity management, these tools often help developers overcome the so called *impedance mismatch* between the relational model at the base of the most used database engines and the object model supported by many programming languages.

One of the main benefits provided by OR/Ms is the increased speed in development that results from having a built-in implementation of many typical tasks related to data access, retrieval and mapping of data to objects – in a database-agnostic way.

One of the most used OR/M is NHibernate, which is port of the Hibernate engine, a mature open-source project for the Java platform.

Let's see what it takes to configure NHibernate in an ASP.NET Web Application project.

Assemblies and Dependencies

The first thing to do is download the NHibernate binaries from the official site. You'll get an archive with the `–bin.zip` extension.

In the `bin` folder of the package you'll find the following assemblies:

- **NHibernate.dll** – It's the main assembly.
- **Iesi.Collections.dll** – Provides the implementation for some data structures (such as sets) not provided by the .NET framework.
- **log4net.dll** – It's the logging framework used by NHibernate. When the project was born, support for logging wasn't built in the .NET framework. Thus, NHibernate acquired a dependency on this assembly.
- **Castle.Core**
- **Castle.DynamicProxy2** – These assemblies from the Castle project are mainly used for generating proxies, a capability that lets NHibernate support features such as *lazy loading*.

To proceed, you need to add references to these assemblies to your ASP.NET web application.

Configuration

Once the needed assemblies have been referenced, you have to configure NHibernate. This can be done either programmatically or declaratively through a configuration file.

The main goal of the configuration phase is to setup a *session factory*. As a result of the configuration, at runtime you'll be able to instantiate an object of type ISessionFactory.

One of the main uses of a session factory is creating a *session*. This is an object of type ISession that is used to interact with objects and the data store in a transparent way. For now, it's safe to assume that you will interact with the OR/M through an ISession object.

The Web.config file

In an ASP.NET web application, the configuration is read from the Web.config file.

The first thing you've got to do is tell ASP.NET that you're going to configure NHibernate. This is done by declaring a `section` element under `configSections`.

```
<configuration>
   <configSections>
      <section name="hibernate-configuration"
               type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate"
               />
   </configSections>
</configuration>
```

The previous step enables you to declare a `hibernate-configuration` element under the `configuration` node without any complaints by the ASP.NET configuration parser.

```
<configuration>
   <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
      <session-factory>
      </session-factory>
   </hibernate>
</configuration>
```

The `session-factory` element holds a set of properties that specify – among the many features available – those that we want equipped in the OR/M. Let's start by adding a basic set of properties used in almost every project.

```
<session-factory>
   <property name="connection.provider">
      NHibernate.Connection.DriverConnectionProvider
   </property>
   <property name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
   <property name="connection.connection_string">
      <!-- Insert the database connection string here -->
   </property>
</session-factory>
```

As you can see, you need to supply

- *Connection provider*, which is used to manage connections to the database.
- *Dialect*, which takes into account the differences in the SQL language implementation depending on the database engine used. In this case, we're going to target Microsoft SQL Server.
- *Connection string*, which provides the parameters used to perform a connection to the underlying database.

Here's another property that you are going to set in order to implement session management in ASP.NET:

```
<property name="current_session_context_class">web</property>
```

A value of `web` tells NHibernate to store an ISession object in the HttpContext.Items collection. The "Session Management" section explains why this is desirable in an ASP.NET application.

The Session Factory

Previously, we said that you open a session through a session factory. How do you instantiate and access it?

While sessions can be instantiated and disposed in an efficient manner, it's better to create a session factory once in an application. With ASP.NET, the Global.asax file could be the right place for this, until you decide to refactor to a more evolved design.

```
public class Global : System.Web.HttpApplication
{
    public static ISessionFactory SessionFactory { get; private set; }
    protected void Application_Start(object sender, EventArgs e)
    {
        var config = new Configuration();
        config.Configure();
        SessionFactory = config.BuildSessionFactory();
    }
}
```

In the `Application_Start` handler, you create an instance of the Configuration class. The `Configure` method is used to read the configuration settings into the Configuration instance.

At this point, you usually need to supply one or more *mapping* files. Mappings are the way in which OR/M tools are able to build objects from rows in a database; and to save their state back into the database. However, mappings are not covered in this article.

Finally, you build the session factory by invoking the `BuildSessionFactory` method on the Configuration instance. The ISessionFactory instance is stored through a static automatic property and can be accessed easily with `Global.SessionFactory`.

Session Management

In NHibernate, a session defines the context and the scope of an *interaction* between an application and the underlying database. To keep things simple, keep in mind that this communication usually consists of operations that result in objects being saved, loaded or deleted from the database.

The *context* and *scope* of a session can vary depending on the kind of application you're developing; or they can be dictated by business requirements. This is the so called *session management*.

With ASP.NET, operations are often performed in the context and for the duration of a request. In other terms, you create a session at the beginning of a request; and you dispose it when the request ends and a response is sent back to the client.

To show one of the possible ways to implement this behavior, we need the `current_session_context_class` property in conjunction with the ASP.NET HTTP Module shown in the following code.

```
public class NHibernateSessionModule : IHttpModule
{
    public void Dispose()
    {
    }
    public void Init(HttpApplication context)
    {
        context.BeginRequest +=
            delegate
                {
                    var session = Global.SessionFactory.OpenSession();
                    CurrentSessionContext.Bind(session);
                };
        context.EndRequest +=
            delegate
                {
                    CurrentSessionContext.Unbind(Global.SessionFactory);
                };
    }
}
```

When the `BeginRequest` even is fired, we open a session and *bind* it to the current session context using the `CurrentSessionContext.Bind` method. When the `EndRequest` event is fired, we *detach* or *unbind* the session with a call to the `CurrentSessionContext.Unbind` method, passing the session factory as an argument.

Finally, we need to register the HTTP Module in the Web.config file.

```
<configuration>
   <system.web>
      <httpModules>
         <add name="NHibernateSessionModule"
              type="MyAssemblyName.NHibernateSessionModule,
                     MyAssemblyName"/>
      </httpModules>
   </system.web>
</configuration>
```

To ensure that everything is setup correctly, let's use a simple ASP.NET page to perform a test.

```
<%@ Page Language="C#" AutoEventWireup="false" CodeBehind="Default.aspx.cs"
         Inherits="WebApplication1._Default" %>
<%@ Import Namespace="WebApplication1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <%= Global.SessionFactory.GetCurrentSession() %>
    </div>
    </form>
</body>
</html>
```

When developing with NHibernate, you'll always call `Global.SessionFactory.GetCurrentSession()` to get a reference to the current available session.

If you run the page, the type of the ISession instance - `NHibernate.Impl.SessionImpl` - should be printed on screen. This is the proof that NHibernate is up and running!

Summary

In this article you've learned how to perform a basic configuration of the NHibernate OR/M in ASP.NET.

References

- The Hibernate official site
- Homepage of nHibernate
- Kuaté, Harris, Bauer, King - NHibernate in Action – Manning, 2009
- NHibernate Reference

Original Url: http://dotnetslackers.com/articles/aspnet/Configuring-NHibernate-with-ASP-NET.aspx