IoC and Mocking

Estimated time for completion: 60 minutes

Overview:

This lab is based around a project management application. In this (cut down) Visual Studio project there is a WCF service that fronts an Entity Framework based repository. That repository has one table, WorkItems. A work item details work to be done on a project and has several fields including an Id, Title, Description, DateCreated and DateFinished.

Goals:

Lab Notes:

N/A

Part 1: Using and Configuring IoC

You're now going to look at testing the WCF service. Currently the service only has one method, GetItems(int limit), this method uses the repository and returns a list of WorkItems. At the moment it is impossible to write unit tests for this method as it uses hard coded references to the repository. Your first task is to initialize an IoC container to set up the repository and then to use the IoC container in the code. You will use Unity which is part of the Patterns and Practices Enterprise Library.

1. As you do not yet have unit tests in place for the project to test it works you need to run the server and client applications.
   a. The solution has a directory called 'sql', this contains the SQL needed to set up the database for the project. Open this file now and execute the sql.
   b. Run the server, **WorkItemServiceHost**,
   c. If the service fails to start add a reservation for the service
      i. Run the VisualStudio Command Prompt as an admin
      ii. Enter the following
      iii. netsh http add urlacl url=http://+:9000/WorkTrek/ user=student
   d. Run the client, **WorkTrekServiceClient**, – the client simply prints out the number of items in the database which should be 4
2. You will now change the project to use IoC.
3. Open the **WorkItemServiceImpl.cs** file in the **WorkItemsService** project. The **GetItems** method gets the connection string from the application configuration file then creates an instance of the repository using that connection string. You are going to replace the concrete creation with use of IoC

4. In the **WorkItemServiceHost** project open the **app.config** file. In this file you need to add configuration for Unity, the code snippet here shows the outline for the configuration.

```
<configSections>
   <section name="unity"
   type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
         Microsoft.Practices.Unity.Configuration" />
</configSections>
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
</unity>
```

5. This configuration file should contain an entry for the namespace and the assemblies used in the project. It should also contain a mapping for the repository which maps **IWorkItemRepository** to **WorkItemRepository**. This mapping should take the connection string as a constructor parameter.

```
<configSections>
   <section name="unity"
   type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
         Microsoft.Practices.Unity.Configuration" />
</configSections>
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
   <namespace name="WorkTrekRepository"/>
   <assembly name="WorkTrekRepository"/>
   <container>
     <register type="IWorkItemRepository" mapTo="WorkItemRepository" >
      <constructor>
        <param name="connectionString"
               value="COPY VALUE FROM CONNECTION STRING SECTION"/>
         </constructor>
       </register>
   </container>
</unity>
```

6. There are different ways of specifying the configuration, for example you could remove the namespace and assembly entries and add aliases instead

```
<alias alias="IWorkItemRepository"
       type="WorkTrekRepository.IWorkItemRepository, WorkTrekRepository"/>
<alias alias="WorkItemRepository"
       type="WorkTrekRepository.WorkItemRepository, WorkTrekRepository"/>
```

7. The **IoC** project is a helper project that parses IoC configuration and sets up a singleton reference to the IoC container. To use this; in the **WorkItemServiceHost** project add a reference to the **IoC** project
8. The **IoC** project is not complete. In the project open the **Container.cs** file. In this file, in the constructor you need to initialize the container:

```
UnityConfigurationSection section = ConfigurationManager.GetSection("unity")
   as UnityConfigurationSection;
if (section != null)
{
    section.Configure(unityContainer);
}
```

9. Once this is in place you can now change the service to use the IoC container. The first thing to do is to add references to the Unity libraries to the **WorkItemsService** project.
10. In the **WorkItemsService** project, go to the Add Reference dialog and browse to the project's lib directory. In there are the two Unity libraries, add these to the project.
11. In the **WorkItemsService** project open the **WorkItemServiceImpl.cs** file.
12. Unity provides many extension methods, to take advantage of these you need to add a new using statement to this file:

```
using Microsoft.Practices.Unity
```

13. In the **GetItems** method, remove the lines that get the connection string and create a repository.
14. In this method add a call to **IoC.Container.Resolve** to resolve the **IWorkItemRepository** that you just configured:

```
IWorkItemRepository repository = IoC.Container.Instance.Resolve<IWorkItemRepo
   sitory>();
IEnumerable<WorkItem> list = repository.Entities.Take(limit).ToList();
return list;
```

15. Now that this is in place you can re-run the service and the client and check that the service still works.

Part 2: Testing the Service and Mocking

In this part of the lab you are going to test the WCF service. To do this you will do refactor the service to accept the repository as a constructor parameter, add a test project to test the service and write a test using the Rhino mocking framework.

1. Open the **WorkItemsService** project.
2. You will add two constructors to this class, a default constructor so that the WCF runtime can create an instance of the class and a constructor that takes an IWorkItemRepository as a single parameter so that we can easily test the class.
3. Add a local variable of type **IWorkItemRepository** and call it **_repository**.
4. In the **WorkItemServiceImpl** class add a default constructor, in the **GetItems** method you added a call to the IoC containers Resolve method, move this call to the default constructor and assign the loaded repository to the **_repository** variable.
5. Change the **GetItems** method to use the **_repository** variable.
6. You may want to run the server and client to check they still work
7. Add a new test **project** called **WorkItemsServiceTest**.

8. The solution has a folder called Rhino, this contains the Rhino mocks dll, add a reference to this DLL to the new project.
9. The test will check that the **GetItems** method returns the correct number of items
10. Change the name of the test to **ShouldReturnTwoItems_WhenGetItems_IsAskedForTwoItems**
11. In the test you will create a mock repository and stub out the repositories **Entities** property, the stubbed property will contain three items and the **GetItems** method will be asked for two of these.
12. To the test project add references to the **WorkItemsService**, **Repository**, **WorkTrekRepository** project and the **WorkTrekModel** project
13. In the test method add a call to Rhino's **MockRepository.GenerateStub<IWorkItemRepository>()** method.

```
IWorkItemRepository repository = MockRepository.GenerateStub<IWorkItemReposit
   ory>();
```

14. In the method create a **List** of 3 **WorkItem** objects, these will be used in the test.

```
List<WorkItem> items = new List<WorkItem> {
    new WorkItem("", "", DateTime.Now),
    new WorkItem("", "", DateTime.Now),
    new WorkItem("", "", DateTime.Now),
};
```

15. On the stub you just created add a call to the **Stub** method. This takes an **Action** specifying the call to stub out (that will be the **Entities** property). This stub should return an **IQueryable** so the call to the **Stub** method should **Return** the list as an **IQueryable**

```
repository.Stub(r => r.Entities).Return(items.AsQueryable());
```

16. Finally create the service instance and call the **GetItems** method passing 2 as the parameter.
17. Add an Assert that two items are returned.

```
[TestMethod]
public void ShouldReturnTwoItems_WhenGetItems_IsAskedForTwoItems()
{
    IWorkItemRepository repository =
                         MockRepository.GenerateStub<IWorkItemRepository>();
    List<WorkItem> actual = new List<WorkItem> {
        new WorkItem("", "", DateTime.Now),
        new WorkItem("", "", DateTime.Now),
        new WorkItem("", "", DateTime.Now),
    };

    repository.Stub(r => r.Entities).Return(actual.AsQueryable());

    WorkItemServiceImpl service = new WorkItemServiceImpl(repository);
    int limit = 2;
    IEnumerable<WorkItem> expected = service.GetItems(limit);
    Assert.AreEqual(expected.Count(), limit);
}
```

18. Run the test, it should succeed.

Solutions

after\ ProjectManagement.sln