* 1. 

Hands-On Lab

ASP.NET MVC – Dependency Injection

* 1. Lab version: 1.1.0
  2. Last updated: 1/11/2011

Contents

[Overview 3](#_Toc282507711)

[Exercise 1: Injecting a Controller 5](#_Toc282507712)

[Task 1 – Running the Application 8](#_Toc282507713)

[Task 2 – Including Unity into MvcMusicStore Solution 9](#_Toc282507714)

[Task 3 – Adding a Unity Controller Factory 10](#_Toc282507715)

[Task 4 – Registering Unity in Global.asax.cs Application\_Start 12](#_Toc282507716)

[Task 5 – Running the Application 13](#_Toc282507717)

[Exercise 2: Injecting a View 14](#_Toc282507718)

[Task 1 – Creating a View that Consumes a Service 15](#_Toc282507719)

[Task 2 – Including a Custom Dependency Resolver and a Custom View Page Activator 16](#_Toc282507720)

[Task 3 – Registering for Dependency Injection in Global.asax.cs Application\_Start 19](#_Toc282507721)

[Task 4 – Running the Application 21](#_Toc282507722)

[Exercise 3: Injecting Action Filters 22](#_Toc282507723)

[Task 1 – Including the Tracking Filter in the Solution 22](#_Toc282507724)

[Task 2 – Registering and Enabling the Filter 26](#_Toc282507725)

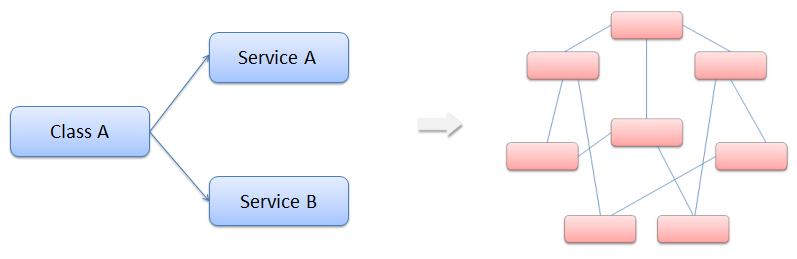
[Task 3 – Running the Application 27](#_Toc282507726)

[Summary 30](#_Toc282507727)

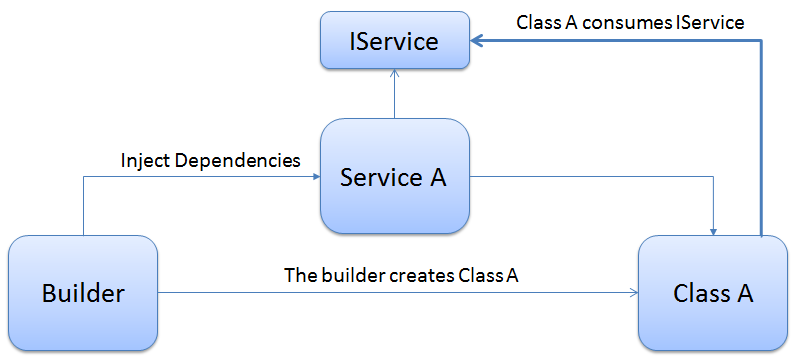
Overview

* 1. **Note:** This Hands-on Lab assumes you have basic knowledge of **ASP.NET MVC** and **ASP.NET MVC 3 filters**. If you have not used **ASP.NET MVC 3 filters** before, we recommend you to go over **ASP.NET MVC Custom Action Filters** and **MVC Global and Dynamic Action filters** Hand-on Lab.

In Object Oriented Programming paradigm, objects work together in a collaboration model where there are contributors and consumers. Naturally, this communication model generates dependencies between objects and components that could became difficult to manage when complexity increases .

* 1. 
  2. Figure
  3. Class dependencies and model complexity
  4. You have probably heard about the Factory Pattern and the separation between the interface and the implementation using services. However, the client objects are often responsible for service location.
  5. Before introducing the Dependency Injection Pattern, we will explain what Inversion of Control (IoC) principle is.
  6. With ***Inversion of Control (Ioc)***, consumer objects do not create the other objects on which they rely. Those objects come from an external source.

# The Dependency Injection (DI) Design Pattern

* 1. Dependency injection (DI) design pattern is based on separating component behavior from dependency resolution without object intervention.
  2. This pattern is a particular implementation of Inversion of Control, where the consumer object receives his dependencies inside constructor properties or arguments.
  3. DI requires a framework component behind to deal with class constructor.
  4. 
  5. Figure
  6. Overview – Dependency Injection diagram
  7. The advantages of using Dependency Injection pattern and Inversion of Control are the following:
  + Reduces class coupling
  + Increases code reusing
  + Improves code maintainability
  + Improves application testing
  1. **Note:** Depencency Injection is sometimes compared with Abstract Factory Design Pattern, but there is a slight difference between both approaches. DI has a Framework working behind to solve dependencies by calling the factories and the registered services.
  2. Now that you understand the Dependency Injection Pattern, you will learn through this lab how to apply it on ASP.NET MVC 3. You will start using Dependency Injection on **Controllers** to include a service for database access. Next you will use Dependency Injection on **Views** to use a service inside a view and display information. Then, you will extend DI to MVC 3 Filters concept and inject a **Custom Action Filter** in the solution.
  3. In this Hands-on Lab, you will learn how to:
  + Integrate MVC 3 with Unity Application Block for Dependency Injection
  + Use dependency injection inside an MVC Controller
  + Use dependency injection inside an MVC View
  + Use dependency injection inside an MVC Action Filter
  1. **Note:** This Lab proposes Unity Application Block as the dependency resolver framework, but it is posible to adapt any Dependency Injection Framework to work with MVC 3.

# System Requirements

* 1. You must have the following items to complete this lab:
  + ASP.NET and ASP.NET MVC 3
  + Visual Studio 2010
  + SQL Server Database (Express edition or above)
    1. **Note:** You can install the previous system requirements by using the Web Platform Installer 3.0: <http://go.microsoft.com/fwlink/?LinkID=194638>.

# Exercises

* 1. This Hands-On Lab is comprised by the following exercises:
  2. Exercise 1: Injecting a Controller
  3. Exercise 2: Injecting a View
  4. Exercise 3: Injecting Filters
  5. Estimated time to complete this lab: **30 minutes**.
  6. **Note:** Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

# Next Step

* 1. Exercise 1: Injecting a Controller

Exercise 1: Injecting a Controller

In this exercise, you will learn how to use Dependency Injection in MVC Controllers, by integrating Unity Application Block. For that reason you will include services into your MVC Music Store controllers to separate the logic from the data access. The service will create a new dependence into the controller constructor that will be resolved using Dependency Injection with the help of **Unity** application block.

With this approach you will learn how to generate less coupled applications, which are more flexible and easier to maintain and test. Additionally, you will also learn how to integrate MVC with Unity.

# About StoreManager Service

The MVC Music Store provided in the begin solution now includes a service that manages the Store Controller data, **StoreService**. Below you will find the Store Service implementation. Note that all the methods return Model entities.

* 1. C# - StoreService.cs
  2. using System;
  3. using System.Collections.Generic;
  4. using System.Linq;
  5. using System.Web;
  6. using MvcMusicStore.Models;
  7. namespace MvcMusicStore.Services
  8. {
  9. public class StoreService : MvcMusicStore.Services.IStoreService
  10. {
  11. **MusicStoreEntities storeDB = new MusicStoreEntities();**
  12. public IList<string> GetGenreNames()
  13. {
  14. var genres = from genre in storeDB.Genres
  15. select genre.Name;
  16. return genres.ToList();
  17. }
  18. public Genre GetGenreByName(string name)
  19. {
  20. var genre = storeDB.Genres.Include("Albums")
  21. .Single(g => g.Name == name);
  22. return genre;
  23. }
  24. public Album GetAlbum(int id)
  25. {
  26. var album = storeDB.Albums.Single(a => a.AlbumId == id);
  27. return album;
  28. }
  29. }
  30. }

Additionally, in the **StoreController** you will find in the begin solution now uses **StoreService**. All data references were removed from Store Controller, and therefore it is possible to modify the current data access provider without making changes at any method that consumes the Store Service.

You will find below that the **Store Controller** implementation has a dependency with the Store Service inside the class constructor.

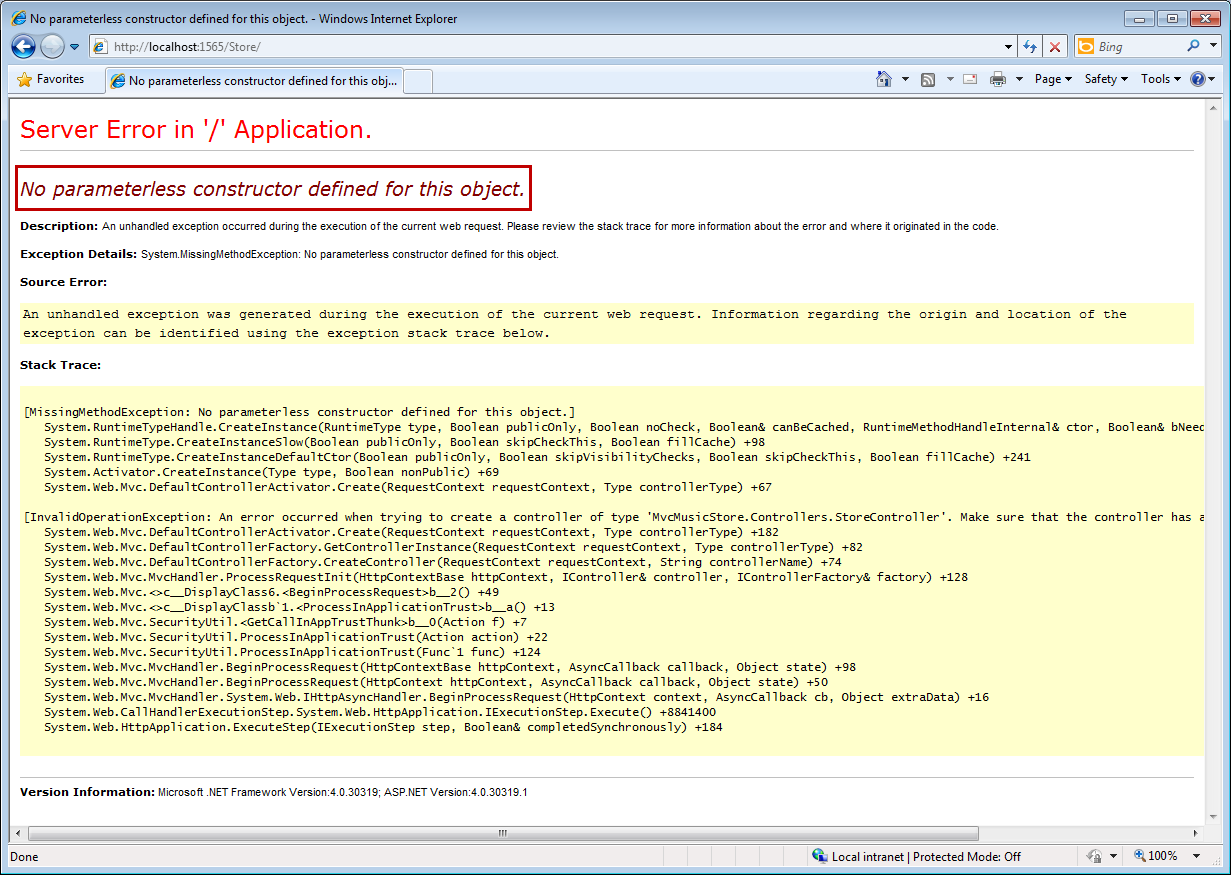
* 1. **Note:** The dependency introduced in this exercise is related to **MVC Inversion of Control** (IoC).
  2. The **StoreController** class constructor receives an **IStoreService** parameter, which is essential to perform service calls inside the class. However, **StoreController** does not implement the default constructor (with no parameters) that any controller must have to work with IoC.

To resolve the dependency, the controller should be created by an abstract factory (a class that returns any object of the specified type).

* 1. C# - StoreController.cs
  2. using System;
  3. using System.Collections.Generic;
  4. using System.Linq;
  5. using System.Web;
  6. using System.Web.Mvc;
  7. **using MvcMusicStore.ViewModels;**
  8. **using MvcMusicStore.Models;**
  9. **using MvcMusicStore.Services;**
  10. namespace MvcMusicStore.Controllers
  11. {
  12. public class StoreController : Controller
  13. {
  14. **private IStoreService service;**
  15. public StoreController(IStoreService service)
  16. {
  17. **this.service = service;**
  18. }
  19. //
  20. // GET: /Store/
  21. public ActionResult Index()
  22. {
  23. // Create list of genres
  24. **var genres = this.service.GetGenreNames();**
  25. // Create your view model
  26. var viewModel = new StoreIndexViewModel
  27. {
  28. Genres = genres.ToList(),
  29. NumberOfGenres = genres.Count()
  30. };
  31. return View(viewModel);
  32. }
  33. //
  34. // GET: /Store/Browse?genre=Disco
  35. public ActionResult Browse(string genre)
  36. {
  37. **var genreModel = this.service.GetGenreByName(genre);**
  38. var viewModel = new StoreBrowseViewModel()
  39. {
  40. Genre = genreModel,
  41. Albums = genreModel.Albums.ToList()
  42. };
  43. return View(viewModel);
  44. }
  45. //
  46. // GET: /Store/Details/5
  47. public ActionResult Details(int id)
  48. {
  49. **var album = this.service.GetAlbum(id);**
  50. return View(album);
  51. }
  52. }
  53. }

**Note:** You will get an error when a class tries to create this Store Controller without sending the service interface, because there is not a parameterless constructor declared.  
Through this lab you will learn how to deal with this problem using Dependency Injection with Unity.

Task 1 – Running the Application

* 1. In this task, you will run the Begin application, which is now including the service into the Store Controller that separates the data access from the application logic.
  2. After browsing to the store you will receive an exception since the controller service is not passed as a parameter by default:
  3. Open the begin solution **MvcMusicStore.sln** at **Source\Ex01-Injecting Controller\Begin**.
  4. Press **F5** to run the application.
  5. Browse to **/Store** to load Store Controller. You will get the error message “**No parameterless constructor defined for this object**”:
     1. 
     2. Figure
     3. Error while running MVC Begin Application
  6. Close the browser.
  7. In the following steps you will work on the Music Store Solution to inject the dependency this controller needs.

Task 2 – Including Unity into MvcMusicStore Solution

* 1. In this task, you will include Unity Application Block 2.0 into your solution.
  2. **Note:** The Unity Application Block (Unity) is a lightweight, extensible dependency injection container with optional support for instance and type interception. It’s a general-purpose container for use in any type of .NET application. It provides all the common features found in dependency injection mechanisms including: object creation, abstraction of requirements by specifying dependencies at runtime and flexibility, be deferring the component configuration to the container.
  3. You could read more about Unity 2.0 at [msdn](http://msdn.microsoft.com/en-us/library/ff663144.aspx).
  4. Open the begin solution **MvcMusicStore.sln** at **Source\Ex01-Injecting Controller\Begin**.
  5. In the **MvcMusicStore** project, add a reference to **Microsoft.Practices.Unity.dll**, which is included in **Source\Assets\Unity 2.0\** folder of this lab.

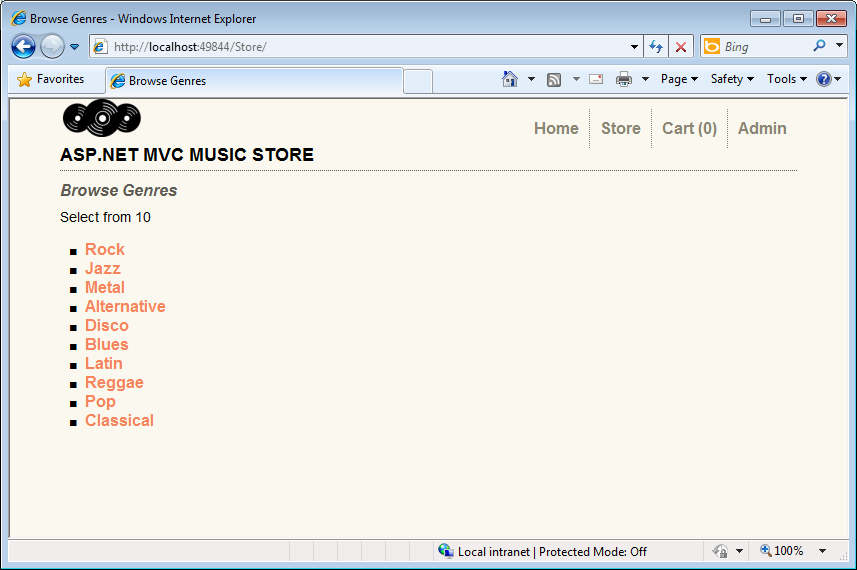
Task 3 – Adding a Unity Controller Factory

* 1. In this task, you will add to the solution a custom controller factory for **Unity**. This class implements **IControllerFactory** interface, extending **CreateController** and **ReleaseController** methods to work with Unity. This factory will create the instances of the controllers that work with Dependency Injection.
  2. **Note:** A controller factory is an implementation of the IControllerFactory interface, which is responsible both for locating a controller type and for instantiating an instance of that controller type.
  3. The following implementation of **CreateController** finds the controller by name inside the Unity container and returns an instance if it was found. Otherwise, it delegates the creation of the controller to an inner factory. One of the advantages of this logic is that controllers can be registered by name.
  4. You can find **IControllerFactory** interface reference at [msdn](http://msdn.microsoft.com/en-us/library/system.web.mvc.icontrollerfactory.aspx).
  5. In the **MvcMusicStore** project, create a new folder named **Factories**,and add the **UnityControllerFactory** class, which is included in the **Source\Assets** folder of this lab.
     1. C# - UnityControllerFactory.cs
     2. using System;
     3. using System.Collections.Generic;
     4. using System.Linq;
     5. using System.Web;
     6. using System.Web.Mvc;
     7. using Microsoft.Practices.Unity;
     8. using System.Web.Routing;
     9. namespace MvcMusicStore.Factories
     10. {
     11. public class UnityControllerFactory : IControllerFactory
     12. {
     13. private IUnityContainer \_container;
     14. private IControllerFactory \_innerFactory;
     15. public UnityControllerFactory(IUnityContainer container)
     16. : this(container, new DefaultControllerFactory())
     17. {
     18. }
     19. protected UnityControllerFactory(IUnityContainer container, IControllerFactory innerFactory)
     20. {
     21. \_container = container;
     22. \_innerFactory = innerFactory;
     23. }
     24. public IController CreateController(RequestContext requestContext, string controllerName)
     25. {
     26. try
     27. {
     28. return \_container.Resolve<IController>(controllerName);
     29. }
     30. catch (Exception)
     31. {
     32. return \_innerFactory.CreateController(requestContext, controllerName);
     33. }
     34. }
     35. public void ReleaseController(IController controller)
     36. {
     37. \_container.Teardown(controller);
     38. }
     39. public System.Web.SessionState.SessionStateBehavior GetControllerSessionBehavior(RequestContext requestContext, string controllerName)
     40. {
     41. return System.Web.SessionState.SessionStateBehavior.Default;
     42. }
     43. }
     44. }
     45. **Note:** This factory class can be reused in any project that uses Dependency Injection for Controllers.

Task 4 – Registering Unity in Global.asax.cs Application\_Start

* 1. In this task, you will register Unity library into **Global.asax.cs** Application Start.
  2. Open **Global.asax.cs** file.
  3. Include **Microsoft.Practices.Unity** Application Block, and references to the namespaces **Services**, **Factories** and **Controllers**:
     1. (Code Snippet – ASP.NET MVC Dependency Injection – Ex1 Injecting Controllers Global using– Csharp)
     2. C#
     3. using System;
     4. using System.Collections.Generic;
     5. using System.Linq;
     6. using System.Web;
     7. using System.Web.Mvc;
     8. using System.Web.Routing;
     9. **using Microsoft.Practices.Unity;**
     10. **using MvcMusicStore.Services;**
     11. **using MvcMusicStore.Factories;**
     12. **using MvcMusicStore.Controllers;**
  4. Create a new Unity Container in **Global.asax.cs** **Application\_Start** and register the Store Service and the Store Controller.
     1. (Code Snippet – ASP.NET MVC Dependency Injection –Ex1 Injecting Controllers Unity Container – Csharp)
     2. C#
     3. …
     4. protected void Application\_Start()
     5. {
     6. AreaRegistration.RegisterAllAreas();
     7. RegisterGlobalFilters(GlobalFilters.Filters);
     8. RegisterRoutes(RouteTable.Routes);
     10. **var container = new UnityContainer();**
     11. **container.RegisterType<IStoreService, StoreService>();**
     12. **container.RegisterType<IController, StoreController>("Store");**
     13. }
     14. …
  5. Register a UnityControllerFactory of the previous container inside MVC ControllerBuilder as the current factory for the controllers:
     1. (Code Snippet – ASP.NET MVC Dependency Injection – Ex1 Injecting Controllers Global application start)
     2. C#
     3. …
     4. protected void Application\_Start()
     5. {
     6. AreaRegistration.RegisterAllAreas();
     7. RegisterGlobalFilters(GlobalFilters.Filters);
     8. RegisterRoutes(RouteTable.Routes);
     10. var container = new UnityContainer();
     11. container.RegisterType<IStoreService, StoreService>();
     12. container.RegisterType<IController, StoreController>("Store");
     13. **var factory = new UnityControllerFactory(container);**
     14. **ControllerBuilder.Current.SetControllerFactory(factory);**
     15. }
     16. …
     17. **Note:** **ControllerBuilder** is an MVC class responsible for dynamically building a controller.
     18. You can read more about ControllerBuilder at [msdn](http://msdn.microsoft.com/en-us/library/system.web.mvc.controllerbuilder.aspx).

Task 5 – Running the Application

* 1. In this task, you will run the application to verify that the Store can now be loaded after including Unity.
  2. Press **F5** to run the application.
  3. Browse to **/Store**. This will invoke **StoreController**, which is now created by using **UnityControllerFactory**.
     1. 
     2. Figure
     3. MVC Music Store
  4. Close the browser.
  5. In the following exercises you will learn how to extend the Dependency Injection scope, and use it inside MVC Views and Action Filters.

# Next Step

* 1. Exercise 2: Injecting a View

Exercise 2: Injecting a View

* 1. In this exercise, you will learn how to apply Dependency Injection into a View by using new MVC 3 Features for Unity Integration. In order to do that, you will call a custom service inside the Store Browse View that shows a message with an image below. The service will introduce a new dependency inside the view as it has to be initialized in a point.
  2. Then, you will integrate the project with Unity Application Block and create a Custom Dependency Resolver to inject the dependencies.

Task 1 – Creating a View that Consumes a Service

* 1. In this task, you will create a view that performs a service call to generate a new dependence. The mentioned service is a simple messaging service example that is included in this solution.
  2. Open the begin solution **MvcMusicStore.sln** at **Source\Ex02-Injecting View\Begin**.
  3. Include **MessageService.cs** and **IMessageService.cs** classes from **Source\Assets\MessageService** folder inside **/Services** folder.
     1. **Note:** The **IMessageService** interface defines two properties that are implemented by the **MessageService** class. These properties are **Message** and **ImageUrl** and are defined to hold the message and Url of the image to be displayed.
  4. Create the folder **/Pages** at project root, and then add the class **MyBasePage.cs** from **Source\Assets\BasePage.** The base page you will inherit from has the following structure:
     1. (Code Snippet – ASP.NET MVC Dependency Injection – Ex02 Injecting Views – MyBasePage – CSharp)
     2. C#
     3. **using System;**
     4. **using System.Collections.Generic;**
     5. **using System.Linq;**
     6. **using System.Web;**
     7. **using Microsoft.Practices.Unity;**
     8. **using MvcMusicStore.Services;**
     9. **namespace MvcMusicStore.Pages**
     10. **{**
     11. **public class MyBasePage : System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreBrowseViewModel>**
     12. **{**
     13. **[Dependency]**
     14. **public IMessageService MessageService { get; set; }**
     15. **}**
     16. **}**

**Note:** On behalf of a technical reason coming from the **ASP.NET** engine, the **dependency** at **IMessageService** interface can’t be included into the respective **View Model** Class.

The class **MyBasePage** intercepts the relationship between the View and the View-Model so the dependency injection at MessageService can now be inherited by the View.

* 1. Open **Browse.aspx** view from **/Views/Store** project folder, and make it inherit from **MyBasePage.cs**:
     1. C#
     2. <%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" **Inherits="MvcMusicStore.Pages.MyBasePage"** %>
  2. Include in **Browse** view a call to **MessageService,** that will display an image and a message retrieved by the service:
     1. HTML
     2. <%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="MvcMusicStore.Pages.MyBasePage" %>
     3. <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
     4. Browse Albums
     5. </asp:Content>
     6. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     7. **<div>**
     8. **<%= this.MessageService.Message %>**
     9. **<br />**
     10. **<img alt="<%: this.MessageService.Message %>"src="<%: this.MessageService.ImageUrl %>" />**
     11. **</div>**
     12. …

Task 2 – Including a Custom Dependency Resolver and a Custom View Page Activator

* 1. In the previous task, you injected a new dependency inside a view to perform a service call inside it. Now, you will start solving that dependence by implementing the new MVC 3 Dependency Injection Interfaces **IViewPageActivator** and **IDependencyResolver**.
  2. You will include in the solution an implementation of **IDependencyResolver** that will deal with service retrieval by using Unity. Then you will include another custom implementation of **IViewPageActivator** interface that will solve the creation of Views.
  3. **Note: MVC 3** implementation for Dependency Injection had simplified the interfaces for service registration. **IDependencyResolver** and **IViewPageActivator** are a part of the new MVC3 features for Dependency Injection.
  4. **- IDependencyResolver** interface replaces the previous IMvcServiceLocator. Implementers of IDependencyResolver must return an instance of the service or a service collection.
  5. **public interface IDependencyResolver {**
  6. **object GetService(Type serviceType);**
  7. **IEnumerable<object> GetServices(Type serviceType);**
  8. **}**
  9. **- IViewPageActivator** interface provides more fine-grained control over how view pages are instantiated via dependency injection. The classes that implement **IViewPageActivator** interface must create the instance of a view having context information.
  10. **public interface IViewPageActivator {**
  11. **object Create(ControllerContext controllerContext, Type type);**
  12. **}**
  13. Copy the class **CustomViewPageActivator.cs** from **/Sources/Assets** to the **Factories** folder. This class implements the **IViewPageActivator** interface to hold the Unity Container.
      1. C# - CustomViewPageActivator
      2. using System;
      3. using System.Collections.Generic;
      4. using System.Linq;
      5. using System.Web;
      6. using System.Web.Mvc;
      7. using Microsoft.Practices.Unity;
      8. namespace MvcMusicStore.Factories
      9. {
      10. public class CustomViewPageActivator : IViewPageActivator
      11. {
      12. IUnityContainer container;
      13. public CustomViewPageActivator(IUnityContainer container)
      14. {
      15. this.container = container;
      16. }
      17. public object Create(ControllerContext controllerContext, Type type)
      18. {
      19. return this.container.Resolve(type);
      20. }
      21. }
      22. }
      23. **Note: CustomViewPageActivator** is responsive for managing the creation of a view by using a Unity container.
  14. Add the class **UnityDependencyResolver.cs,** which is included in the folder **/Sources/Assets**, in the project folder **/Factories.**
      1. C# - UnityDependencyResolver.cs
      2. using System;
      3. using System.Collections.Generic;
      4. using System.Linq;
      5. using System.Web;
      6. using System.Web.Mvc;
      7. using Microsoft.Practices.Unity;
      8. namespace MvcMusicStore.Factories
      9. {
      10. **public class UnityDependencyResolver : IDependencyResolver**
      11. {
      12. IUnityContainer container;
      13. IDependencyResolver resolver;
      14. public UnityDependencyResolver(IUnityContainer container, IDependencyResolver resolver)
      15. {
      16. this.container = container;
      17. this.resolver = resolver;
      18. }
      19. public object GetService(Type serviceType)
      20. {
      21. try
      22. {
      23. return this.container.Resolve(serviceType);
      24. }
      25. catch
      26. {
      27. return this.resolver.GetService(serviceType);
      28. }
      29. }
      30. public IEnumerable<object> GetServices(Type serviceType)
      31. {
      32. try
      33. {
      34. return this.container.ResolveAll(serviceType);
      35. }
      36. catch
      37. {
      38. return this.resolver.GetServices(serviceType);
      39. }
      40. }
      41. }
      42. }
      43. **Note: UnityDependencyResolver** class is a custom DependencyResolver for Unity. When a service can’t be found inside the Unity container, base resolver is invocated.
  15. In the following task both implementations will be registered, to let the model know where to locate the services and where to create the views.

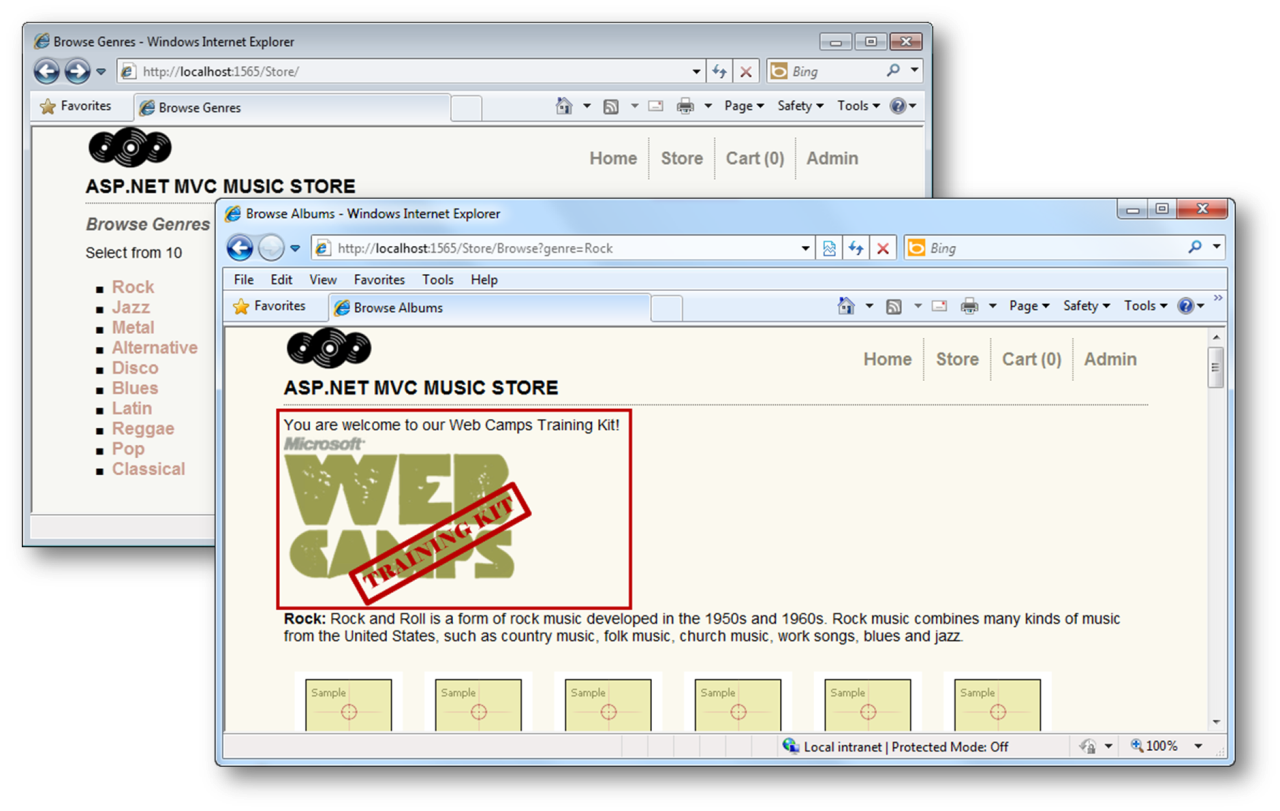
Task 3 – Registering for Dependency Injection in Global.asax.cs Application\_Start

* 1. In this task, you will put all the previous things together to make Dependency Injection Work.
  2. Up to now you have the following elements:
  + A **Browse** Viewthat inherits from **MyBaseClass** and consumes MessageService.
  + An intermediate **MyBaseClass** class that has dependency injection declared for the service interface.
  + A **MessageService** service and its Interface **IMessageService**.
  + A custom dependency resolver for Unity: **UnityDependencyResolver**,that deals with service retrieval.
  + A View Page activator: **CustomViewPageActivator,** that creates the page.
  1. To inject Browse View, you will now register the custom dependency resolver into the Unity container.
  2. Open **Global.asax.cs** at project root.
  3. Register an instance of **MessageService** into Unity container that will initialize the service:
     1. (Code Snippet – ASP.NET MVC Dependency Injection – Ex02 Injecting a View – GlobalAsax Registration – Csharp)
     2. C#
     3. protected void Application\_Start()
     4. {
     5. AreaRegistration.RegisterAllAreas();
     6. RegisterGlobalFilters(GlobalFilters.Filters);
     7. RegisterRoutes(RouteTable.Routes);
     9. var container = new UnityContainer();
     10. container.RegisterType<IStoreService, StoreService>();
     11. container.RegisterType<IController, StoreController>("Store");
     12. var factory = new UnityControllerFactory(container);
     13. ControllerBuilder.Current.SetControllerFactory(factory);
     14. **container.RegisterInstance<IMessageService>(new MessageService {**
     15. **Message = "You are welcome to our Web Camps Training Kit!",**
     16. **ImageUrl = "/Content/Images/logo-webcamps.png"**
     17. **});**
     19. }
  4. Register **CustomViewPageActivator** as a view page activator into Unity container:

(Code Snippet – ASP.NET MVC Dependency Injection – Ex02 Injecting a View – GlobalAsax Registration 2 – Csharp)

* + 1. C#
    2. protected void Application\_Start()
    3. {
    4. …
    5. container.RegisterInstance<IMessageService>(new MessageService {
    6. Message = "You are welcome to our Web Camps Training Kit!",
    7. ImageUrl = "/Content/Images/logo-webcamps.png"
    8. });
    9. **container.RegisterType<IViewPageActivator, CustomViewPageActivator>(new InjectionConstructor(container));**
    10. }
  1. Replace MVC 3 default dependency resolver with an instance of **UnityDependencyResolver**:
     1. (Code Snippet – ASP.NET MVC Dependency Injection – Ex02 Injecting a View – GlobalAsax Registration 3 – Csharp)
     2. C#
     3. protected void Application\_Start()
     4. {
     5. AreaRegistration.RegisterAllAreas();
     6. RegisterGlobalFilters(GlobalFilters.Filters);
     7. RegisterRoutes(RouteTable.Routes);
     9. var container = new UnityContainer();
     10. container.RegisterType<IStoreService, StoreService>();
     11. container.RegisterType<IController, StoreController>("Store");
     12. var factory = new UnityControllerFactory(container);
     13. ControllerBuilder.Current.SetControllerFactory(factory);
     14. container.RegisterInstance<IMessageService>(new MessageService {
     15. Message = "You are welcome to our Web Camps Training Kit!",
     16. ImageUrl = "/Content/Images/logo-webcamps.png"
     17. });
     18. container.RegisterType<IViewPageActivator, CustomViewPageActivator>(new InjectionConstructor(container));
     19. **IDependencyResolver resolver = DependencyResolver.Current;**
     20. **IDependencyResolver newResolver = new UnityDependencyResolver(container, resolver);**
     21. **DependencyResolver.SetResolver(newResolver);**
     22. }
     23. **Note:** ASP.NET MVC3 provides a default dependency resolver class. To work with custom dependency resolvers as the one we have created for unity, this resolver has to be replaced.

Task 4 – Running the Application

* 1. In this task, you will run the application to verify that the Store Browser consumes the service and shows the image and the message retrieved by it:
  2. Press **F5** to run the application, where Home page will load.
  3. Browse to **/Store** and enter to any of the genres that are shown below. In this example, we enter to “**Rock**”:
     1. 
     2. Figure

MVC Music Store - View Injection

* 1. Close the browser.

# Next Step

* 1. Exercise 3: Injecting Action Filters

Exercise 3: Injecting Action Filters

* 1. In a previous Lab about Custom Action Filters you have been working with filters customization and injection. In this exercise,you will learn how to inject filters with Dependency Injection by using Unity Application Block containers. To do that, you will add to the Music Store Solution a custom action filter that will trace site activity.

Task 1 – Including the Tracking Filter in the Solution

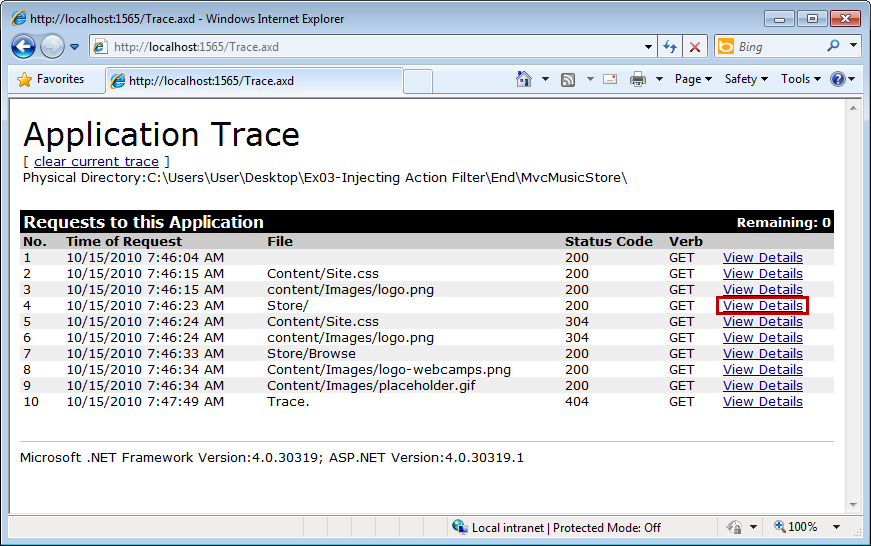
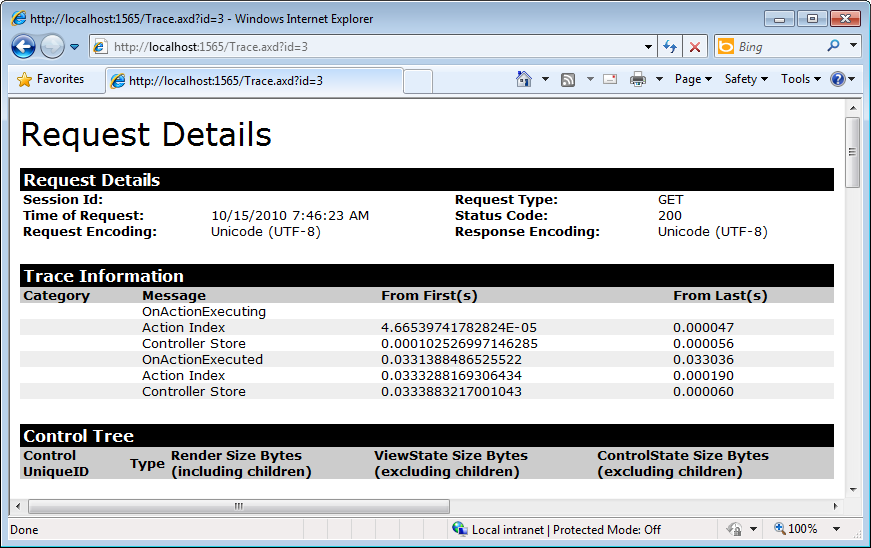
* 1. In this task, you will include in the Music Store a custom action filter for event tracing. As filters were treated in a previous Lab “Custom Action Filters”, you will include the filter class from the Assets folder and then create a Filter Provider for Unity:
  2. Open the begin solution at **/Source/Ex03 – Injecting Filters/Begin/MvcMusicStore.sln**.
  3. Create the folder **/Filters** at project root.
  4. Add the custom action filter **TraceActionFilter.cs** to the project in the folder **/Filters** that you can find it at**/Sources/Assets/TraceActionFilter.cs**.
     1. C# - TraceActionFilter.cs
     2. using System;
     3. using System.Collections.Generic;
     4. using System.Linq;
     5. using System.Web;
     6. using System.Web.Mvc;
     7. namespace MvcMusicStore.Filters
     8. {
     9. public class TraceActionFilter : IActionFilter
     10. {
     11. public void OnActionExecuted(ActionExecutedContext filterContext)
     12. {
     13. filterContext.HttpContext.Trace.Write("OnActionExecuted");
     14. filterContext.HttpContext.Trace.Write("Action " + filterContext.ActionDescriptor.ActionName);
     15. filterContext.HttpContext.Trace.Write("Controller " + filterContext.ActionDescriptor.ControllerDescriptor.ControllerName);
     16. }
     17. public void OnActionExecuting(ActionExecutingContext filterContext)
     18. {
     19. filterContext.HttpContext.Trace.Write("OnActionExecuting");
     20. filterContext.HttpContext.Trace.Write("Action " + filterContext.ActionDescriptor.ActionName);
     21. filterContext.HttpContext.Trace.Write("Controller " + filterContext.ActionDescriptor.ControllerDescriptor.ControllerName);
     22. }
     23. }
     24. }
     25. **Note:** This custom action filter performs ASP.NET tracing.
     26. You can check “Global and Dynamic Action Filters” Lab for more reference.
  5. Add the empty class **FilterProvider.cs** to the project in the folder **/Filters.**
  6. Include **System.Web.Mvc** and **Microsoft.Practices.Unity** namespaces in **FilterProvider.cs**.
     1. (Code Snippet – ASP.NET MVC Dependency Injection – Ex03 Injecting Action Filters – FilterProvider namespace – Csharp)
     2. C#
     3. using System;
     4. using System.Collections.Generic;
     5. using System.Linq;
     6. using System.Web;
     7. **using System.Web.Mvc;**
     8. **using Microsoft.Practices.Unity;**
     9. namespace MvcMusicStore.Filters
     10. {
     11. public class FilterProvider
     12. {
     13. }
     14. }
  7. Make the class inherit from **IFilterProvider** Interface.
     1. C#
     2. using System;
     3. using System.Collections.Generic;
     4. using System.Linq;
     5. using System.Web;
     6. using System.Web.Mvc;
     7. using Microsoft.Practices.Unity;
     8. namespace MvcMusicStore.Filters
     9. {
     10. public class FilterProvider : **IFilterProvider**
     11. {
     12. }
     13. }
  8. Add a **IUnityContainer** property in **FilterProvider** class, and then create a class constructor to set the container:
     1. (Code Snippet – ASP.NET MVC Dependency Injection – Ex03 Injecting Action Filters – IUnityContainer – Csharp)
     2. C#
     3. using System;
     4. using System.Collections.Generic;
     5. using System.Linq;
     6. using System.Web;
     7. using System.Web.Mvc;
     8. using Microsoft.Practices.Unity;
     9. namespace MvcMusicStore.Filters
     10. {
     11. public class FilterProvider : IFilterProvider
     12. {
     13. **IUnityContainer container;**
     14. **public FilterProvider(IUnityContainer container)**
     15. **{**
     16. **this.container = container;**
     17. **}**
     18. }
     19. }
     20. **Note:** The filter provider class constructor is not creating a **new** object inside. The container is passed as a parameter, and the dependency is solved by Unity.
  9. Implement in **FilterProvider** class the method **GetFilters** from **IFilterProvider** interface:
     1. (Code Snippet – ASP.NET MVC Dependency Injection – Ex03 Injecting Action Filters – GetFilters – Csharp)
     2. C#
     3. using System;
     4. using System.Collections.Generic;
     5. using System.Linq;
     6. using System.Web;
     7. using System.Web.Mvc;
     8. using Microsoft.Practices.Unity;
     9. namespace MvcMusicStore.Filters
     10. {
     11. public class FilterProvider : IFilterProvider
     12. {
     13. IUnityContainer container;
     14. public FilterProvider(IUnityContainer container)
     15. {
     16. this.container = container;
     17. }
     18. **public IEnumerable<Filter> GetFilters(ControllerContext controllerContext, ActionDescriptor actionDescriptor)**
     19. **{**
     20. **foreach (IActionFilter actionFilter in container.ResolveAll<IActionFilter>())**
     21. **yield return new Filter(actionFilter, FilterScope.First, null);**
     22. **}**
     23. }
     24. }
     25. **Note:** This implementation of **GetFilters** method is shorter than the one you have learned in “**Global and Dynamic Action Filters”** lab, where you ask if each controller or action was inside a list of accepted items.

Task 2 – Registering and Enabling the Filter

In this task, you will enable site tracking and then you will register the filter in **Global.asax.cs Application\_Start** method to start tracing:

* 1. Open **Web.config** at project root and enable trace tracking at System.Web group:
     1. XML
     2. <system.web>
     3. **<trace enabled="true"/>**
     4. <compilation debug="true" targetFramework="4.0">
  2. Open **Global.asax.cs** at project root and add a reference to the Filters namespace:
     1. C#
     2. using System;
     3. using System.Collections.Generic;
     4. using System.Linq;
     5. using System.Web;
     6. using System.Web.Mvc;
     7. using System.Web.Routing;
     8. using Microsoft.Practices.Unity;
     9. using MvcMusicStore.Services;
     10. using MvcMusicStore.Factories;
     11. using MvcMusicStore.Controllers;
     12. **using MvcMusicStore.Filters;**
     13. namespace MvcMusicStore
  3. Select **Application\_Start** method and register the filter in the Unity Container. You will have to register the filter provider and the action filter as well:
     1. (Code Snippet – ASP.NET MVC Dependency Injection – Ex03 Injecting Action Filters – Global asax unity registration - CSharp)
     2. C#
     3. protected void Application\_Start()
     4. {
     5. AreaRegistration.RegisterAllAreas();
     6. RegisterGlobalFilters(GlobalFilters.Filters);
     7. RegisterRoutes(RouteTable.Routes);
     9. var container = new UnityContainer();
     10. container.RegisterType<IStoreService, StoreService>();
     11. container.RegisterType<IController, StoreController>("Store");
     12. var factory = new UnityControllerFactory(container);
     13. ControllerBuilder.Current.SetControllerFactory(factory);
     14. IDependencyResolver resolver = DependencyResolver.Current;
     15. container.RegisterInstance<IMessageService>(new MessageService {
     16. Message = "You are welcome to our Web Camps Training Kit!",
     17. ImageUrl = "/Content/Images/logo-webcamps.png"
     18. });
     19. container.RegisterType<IViewPageActivator, CustomViewPageActivator>(new InjectionConstructor(container));
     20. **container.RegisterInstance<IFilterProvider>("FilterProvider", new FilterProvider(container));**
     21. **container.RegisterInstance<IActionFilter>("LogActionFilter", new TraceActionFilter());**
     22. IDependencyResolver newResolver = new UnityDependencyResolver(container, resolver);
     23. DependencyResolver.SetResolver(newResolver);
     24. }

Task 3 – Running the Application

* 1. In this task, you will run the application and test that the custom action filter is tracing the activity:
  2. Press **F5** to run the application.
  3. Browse to **/Store** and choose ‘**Rock’** genre. You can browse to more genres if you want to.
     1. 
     2. Figure
     3. Music Store
  4. Browse to **/Trace.axd** to see the Application Trace page, and then click the ‘**View Details’** link at the right column for **Store/**:
     1. 
     2. Figure
     3. Application Trace Log
     4. 
     5. Figure
     6. Application Trace – Request Details
  5. Close the browser.

# Next Step

Summary

Summary

* 1. By completing this Hands-On Lab you have learned how to use Dependency Injection in MVC 3 by integrating Unity Application Block. To achieve that purpose you have used Dependency Injection inside controllers, views and action filters.
  2. The following concepts were used:
  + MVC 3 Dependency Injection new features
  + Unity Application Block integration
  + Dependency Injection in Controllers
  + Dependency Injection in Views
  + Dependency injection of Action Filters