.NET Framework: Developing Modern Web Apps with ASP.NET MVC - Workshop*PLUS*

Module 03: Controllers

Student Lab Manual

Instructor Edition (Book Title Hidden Style)

Version 1.0

**Conditions and Terms of Use**

Microsoft Confidential

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2017 Microsoft Corporation. All rights reserved.

**Copyright and Trademarks**

© 2017 Microsoft Corporation. All rights reserved.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see Use of Microsoft Copyrighted Content at  
<http://www.microsoft.com/en-us/legal/intellectualproperty/Permissions/default.aspx>

Azure, Microsoft, SQL Server, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other Microsoft products mentioned herein may be either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are property of their respective owners.

Contents

[Lab 3: Creating Controllers 1](#_Toc481002049)

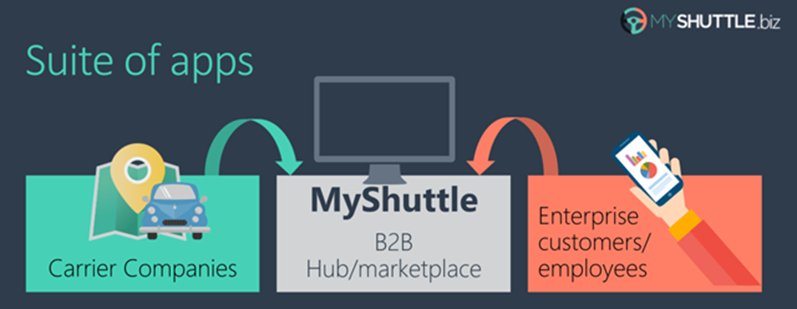
[Exercise 1: Create a Controller and Implement Action Methods 6](#_Toc481002050)

[Exercise 2: Running the MyShuttle.Web Application 13](#_Toc481002051)

# Lab 3: Creating Controllers

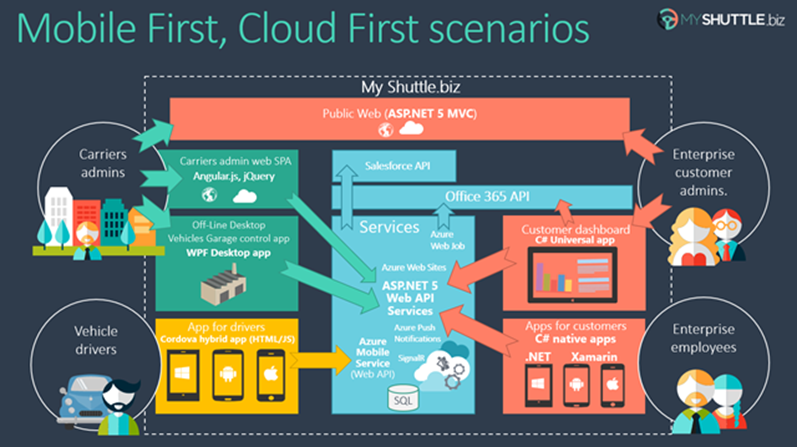
#### Introduction

MyShuttle is a B2B highly scalable multi-tenant software as a service (SaaS) solution that targets corporate scenarios in which carrier companies offer transport services to enterprise customers.



This multi-tenant SaaS system would allow any number of carrier companies who must be syndicated with the system, to provide their services (cabs/shuttles) directly to any number of customer-enterprises/companies who would also be registered in the MyShuttle.biz system. The final outcome is that any employee in those customer companies would be able to request a cab/shuttle at any time in any place/city without worrying about how to pay. Everything would take place underneath between their company and the carrier company for that ride.

The global scenario is described in the following image:

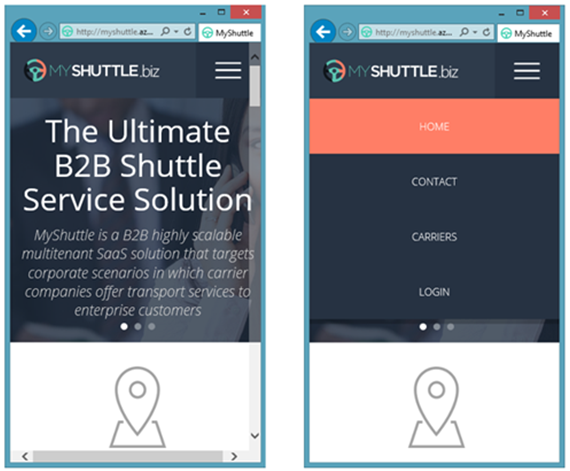


There are two web applications in the above scenario:

**Public Website:**

It is a typical public website. Its main purpose is to show information about the business but in a modern and clean way. It provides a responsive design and even if you resize the browser, you can see how it would also be perfectly valid for mobile devices, like a smartphone.

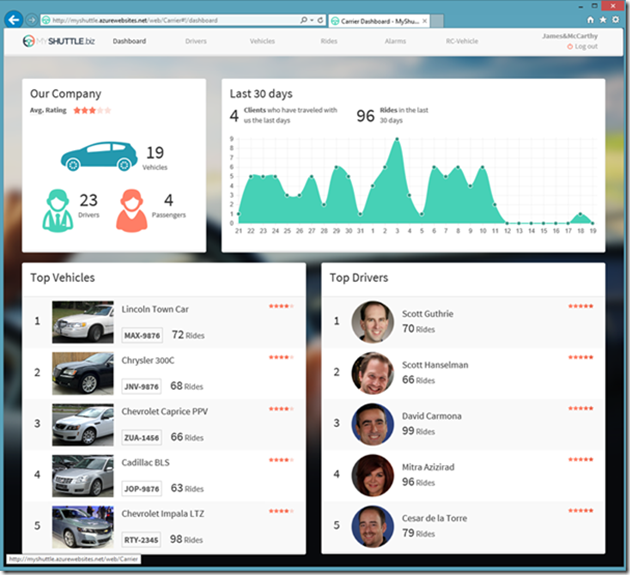




**Private Web Application:**

The second application is a Web Single Page Application (SPA), which you can access by logging in from the public website. However, in reality it is like a different web application, simulating a private web application especially made for the Carriers’ administrators.

This application is a data-driven and CRUD app so you can create and update information about your drivers, vehicles, etc. This application consumes the ASP.NET Web API Services using client-side frameworks.



In this series of labs, you will build the public website and some parts of the private web application above.

The Model View Controller (MVC) architectural pattern separates an application into three main components:

* **Models:** Model objects are the parts of the application that implement the domain logic. Often, model objects also retrieve and store model state in a database.
* **Views:** Views are the components that display the application's User Interface (UI). Usually, this UI is created from the model data. An example would be an edit view of a Products table that displays text boxes, drop-down lists, and check boxes based on the current state of a Product object.
* **Controllers:** Controllers are the components that handle user interaction, manipulate the model, and ultimately select a view to render the UI. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction.

The MVC pattern helps you to create applications that separate the different aspects of the application (such as input logic, business logic, and UI logic), while providing a loose coupling between these elements. This separation helps you manage complexity when you build an application, because it allows you to focus on one aspect of the implementation at a time. In addition to managing complexity, the MVC pattern makes it easier to test applications than it is to test a traditional ASP.NET Web application, encouraging the use of Test Driven Development (TDD) to create an application.

The ASP.NET MVC framework provides an alternative to the ASP.NET Web Forms pattern for creating MVC-based Web applications. The ASP.NET MVC framework is a lightweight, highly testable, presentation framework (as with Web Forms-based applications) that is integrated with existing ASP.NET features, such as master pages and membership-based authentication.

In addition, the loose coupling between the three main components of an MVC application also promotes parallel development. For instance, one developer can work on the view, a second developer can work on the controller logic, and a third developer can focus on the business logic in the model.

#### Overview

In this lab, you will:

* Explore the role of controllers.
* Understand how to create controllers.
* Implement action methods.

#### Objectives

After completing this lab, you will:

* Understand the basic principles of MVC.
* Learn how to implement a controller and action methods.
* Learn how to pass data to action methods.
* Learn how to implement asynchronous action methods.

#### Prerequisites

None

#### Scenario

Creating a Controller

#### Exercises

This hands-on lab includes the following exercises:

* Exercise 1: Create a Controller and Implement Action Methods.
* Exercise 2: Running the MyShuttle.Web Application

#### System Requirements

To complete this lab, you need:

* Microsoft Visual Studio 2017
* Microsoft SQL Server (any edition)

#### Hosted Lab Credentials

If the lab is exercised in Microsoft cloud environment, use the following user credentials to sign in:

* Username: aspnetuser
* Password: @Cir9hvc6!w

#### Estimated Time to Complete This Lab

60 minutes

#### For more information, (if applicable)

Refer the following blog post to see how MyShuttle application interoperates with a number of other client-side applications:

<http://blogs.msdn.com/b/cesardelatorre/archive/2014/11/30/myshuttle-biz-demo-apps-from-connect-visual-studio-and-azure-event.aspx>

Exercise 1: Create a Controller and Implement Action Methods

#### Introduction

In ASP.NET Web Form applications, user interaction is organized around pages, and around raising and handling events from those pages. In contrast, user interaction with ASP.NET MVC applications is organized around controllers and their action methods.

The ASP.NET MVC framework maps URLs to classes that are referred to as controllers. Controllers process incoming requests, handle user input and interactions, and execute appropriate application logic. A controller class typically calls a separate view component to generate the HTML markup for the request. In a MVC application, the view only displays information; the controller handles and responds to user input and interaction.

#### Objectives

Learn to *create new controllers* and different types of Action methods for your controllers.

#### Scenario

Create a new Controller.

In this exercise, you will:

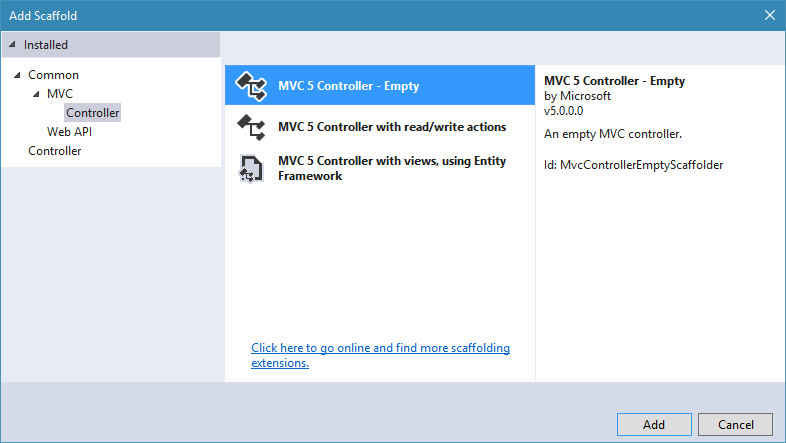
* Create Home Controller for MyShuttle application.
* Create CarrierList Controller for MyShuttle application

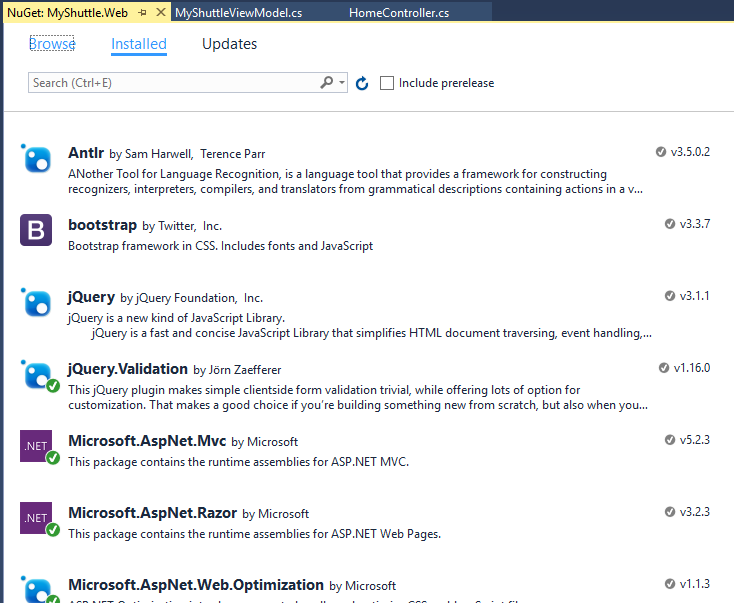
Task 1: Create Home Controller

Before creating any new classes, make sure your default namespace matches “MyShuttle.Web”, by going to the **Properties** tab on the MyShuttle.Web project, and selecting the **Default namespace setting** field in the **Application** tab. If not, change **Default namespace** to “MyShuttle.Web”.

**Note**: Controllers are classes that handle incoming browser requests, retrieve model data, and then specify view templates that return a response to the browser.

1. In **Solution Explorer**, open **MyShuttle.Web** and add a new folder named **Controllers**. This folder will be the home of our controllers.
2. Right-click the **Controllers** folder and select **Add** **>** **Add Controller**.  
   Select **MVC Controller – Empty** item and click the **Add** button.



1. In the next dialog, give **HomeController** as the name of your new controller.
2. Observe that Visual Studio has added the full set of standard MVC dependencies to the project. For now, ignore the instructions that may appear in **Readme.txt**.  
   Make note of these dependencies by right-clicking the **MyShuttle.Web** project in **Solution explorer** and selecting **Manage NuGet Packages**. Make sure you’re looking at the **Installed** tab:
3. It’s the right time to update all of NuGet packages in the solution. Right click the **MyShuttle solution** and select **Manage NuGet Packages for Solution**. Go to the **Updates** tab, select all packages and click **Update**.

**Note:** In the rare case when the latest version of NuGet package(s) introduce a breaking change, refer to the version(s) of NuGet packages this lab has been tested with in *End* folder.

1. In **Solution explorer** locate the pre-created folder **Models**. This folder will contain models used by the controller to be displayed or sent to the user requests. This type of model is called a ViewModel. By convention and best practices, you will suffix any model class name with “ViewModel”.
2. Right-click the **Models** folder and **Add** **>** **New Item**. Select **Class** and name it as **MyShuttleViewModel.cs**
3. Add a public property **MainMessage** of type String to **MyShuttleViewModel** class:

namespace MyShuttle.Web.Models

{

public class MyShuttleViewModel

{

public string MainMessage { get; set; }

}

}

1. Also add the following using statements in **HomeController.cs**:

using MyShuttle.Web.Models;

1. Refactor the **Index** action method of the **HomeController** as below to set the **MainMessage** property of the **MyShuttleViewModel** and return the **ViewResult**.

public ActionResult Index()

{

var model = new MyShuttleViewModel()

{

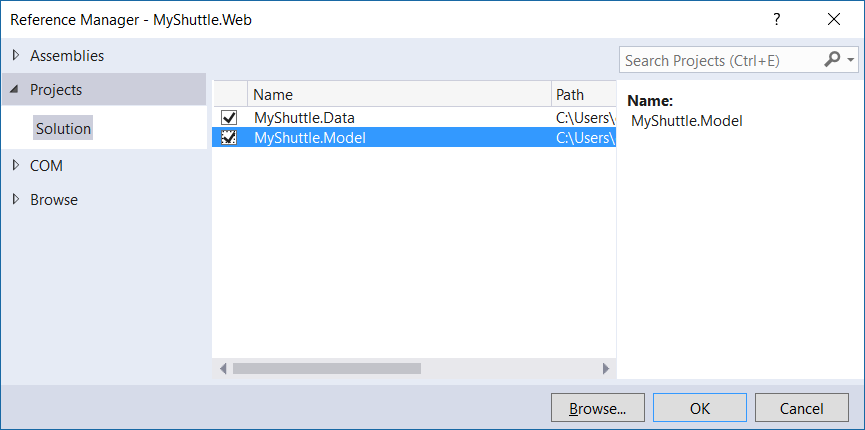
MainMessage = "The Ultimate B2B Shuttle Service Solution"

};

return View(model);

}

1. Compile the solution to ensure it compiles successfully.

Currently, the **HomeController** only displays a simple message through the **Index** action method. To demonstrate some advanced features, you would want this controller to interact with the carrier information between the user requests and the application data. Let us add references to the **MyShuttle.Model** and **MyShuttle.Data** projects by right-clicking the **MyShuttle.Web** project and selecting **Add >** **Reference**.  


1. Add the corresponding using directives in the **HomeController.cs** file:

using MyShuttle.Model;

using MyShuttle.Data;

1. Add the following code snippet in the **HomeController** class.

ICarrierRepository \_carrierRepository;

public HomeController(ICarrierRepository carrierRepository)

{

\_carrierRepository = carrierRepository;

}

You just implemented **Dependency Injection** (or DI for short) at the Controller level. **HomeController** now consumes the **CarrierRepository** service.  
However, ASP.NET 5 is only prepared to use a DI container, there’s none actually used by default. You need to add one first.

1. Right-click the **MyShuttle.Web** project in **Solution Explorer** and select **Manage NuGet Packages**. On the **Browse** tab, search for “SimpleInjector.MVC3”. Don’t get confused about the MVC3 name here; it refers to the type of container integration introduces in MVC3.
2. ASP.NET MVC will now use SimpleInjector to resolve dependencies of your containers. If you tried compiling the application now, you’d get an error in **SimpleInjectorInitializer.cs** at line ~32. This error is to call your attention to an important step here: before you can ask a DI container to resolve interfaces to actual types, you need to register these with the container. Now, replace the error’s line in the method **InitializeContainer** with this:

container.Register<MyShuttle.Data.ICarrierRepository, MyShuttle.Data.CarrierRepository>();

1. When an instance of the repository is needed, the framework will use SimpleInjector framework to inject a new instance into objects that need it (for example, the **HomeController**).
2. Navigate back to the **HomeController** class. Add another action method, which will be used for adding a new carrier by calling the carrier repository. Decorate it with [HttpPost] attribute.

[HttpPost]

public async Task<int> Post(Carrier carrier)

{

return await \_carrierRepository.AddAsync(carrier);

}

Your final **HomeController** class should look like the following:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using MyShuttle.Web.Models;

using MyShuttle.Model;

using MyShuttle.Data;

using System.Threading.Tasks;

namespace MyShuttle.Web.Controllers

{

public class HomeController : Controller

{

ICarrierRepository \_carrierRepository;

public HomeController(ICarrierRepository carrierRepository)

{

\_carrierRepository = carrierRepository;

}

// GET: Home

public ActionResult Index()

{

var model = new MyShuttleViewModel()

{

MainMessage = "The Ultimate B2B Shuttle Service Solution"

};

return View(model);

}

[HttpPost]

public async Task<int> Post(Carrier carrier)

{

return await \_carrierRepository.AddAsync(carrier);

}

}

}

1. Build the solution to ensure it compiles successfully.

Task 2: Create CarrierList Controller for the application

1. Let us add another controller to handle more specific information related to the carrier model. Right-click the **Controllers** folder and **Add** **>** **Controller**. Select **MVC 5 Controller - Empty**, click **Add**, and name the class as **CarrierListController**,and then click the **Add** button.
2. Replace the code in the new CarrierListController.cs with this:

using System.Threading.Tasks;

using System.Web.Mvc;

using MyShuttle.Data;

using MyShuttle.Web.Models;

namespace MyShuttle.Web.Controllers

{

public class CarrierListController : Controller

{

private ICarrierRepository \_carrierRepository;

public CarrierListController(ICarrierRepository carrierRepository)

{

\_carrierRepository = carrierRepository;

}

public async Task<ActionResult> Index(SearchViewModel searchVM)

{

string searchString = searchVM == null ? null : searchVM.SearchString;

var carriers = await \_carrierRepository.GetCarriersAsync(searchString);

var model = new CarrierListViewModel(carriers);

return View("Index", model);

}

}

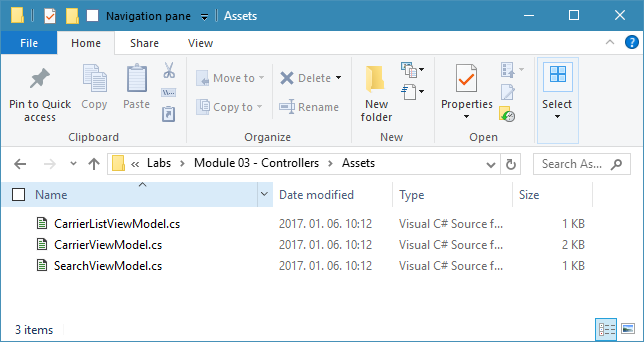
}

**Note**: At this point there will be some unresolved dependencies (*SearchViewModel* and *CarrierListViewModel*) – these will be addressed in the next few steps.

Sometimes, action methods may execute for a long time to get what the user needs. You may want to speed up the execution by running the long-running tasks in parallel to reduce the request execution time.

You will invoke *GetCarriersAsync* method to fetch all Carriers and pass the result to the View if the search criteria matches.

1. Right-click the **Models** folder in the **MyShuttle.Web** project and **Add** **>** **Existing** **Item**. Navigate to this lab assets location and add **CarrierListViewModel.cs**, **CarrierViewModel.cs** and **SearchViewModel.cs**:



1. To run and test the application before adding views, add the following code to **Global.asax.cs** to write text in the BeginRequest phase of the request pipeline.

protected void Application\_BeginRequest()

{

Response.Write("Hello World!");

Response.End();

}

**Note:** Normally, you would use MVC views to display a UI in your web app – this code is only there to show that you can use event handlers of the Application object to add request processing in special cases.

1. Build the solution to ensure it compiles successfully.

Exercise 2: Running the MyShuttle.Web Application

#### Objectives

In this exercise, you will:

* Test run the application and make sure there is no error.
* Look at how to add MVC into the hosting pipeline.

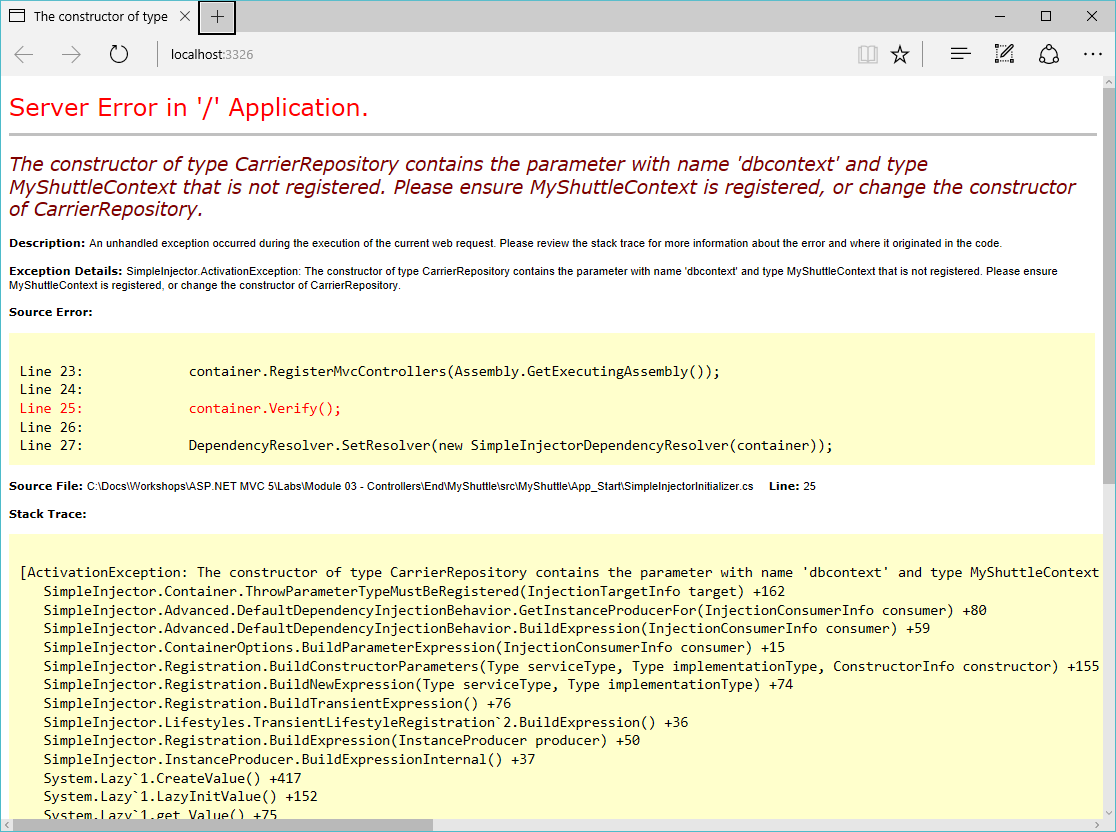
Task 1: Test run the application

So far, you have created the related models and controllers for our MVC application. You would want to ensure that everything is working properly.

1. Right-click the **MyShuttle.Web** project and select **Debug** **>** **Start New Instance.**

By default, the web application is hosted in **IIS Express** to serve the page.

Visual Studio launches the default browser to show the page. However, the app will not start as expected. You should instead an error message similar to this:



What happened? Can you figure it out from the exception message?

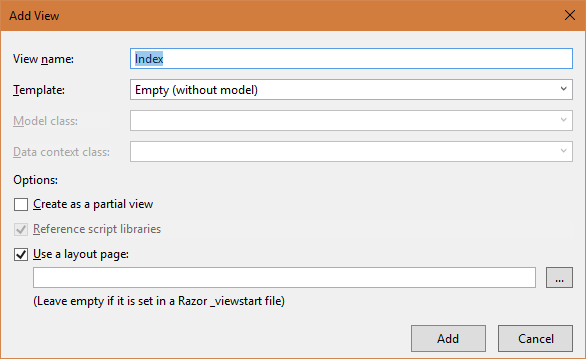
1. The error was caused by SimpleInjector’s verification of the DI container. It has realized that in order to create the registered **CarrierRepository**, its single dependency, **MyShuttleContext** must be created first – however, that type hadn’t been registered with the container yet. While DI containers – including SimpleInjector – usually support automatic registration (called auto wire-up), this feature can hide subtle dependency resolution problems, so you’re not using it right now. Even if in this concrete case the dependency is an actual class (and not an interface), which could’ve been instantiated directly, there’re other parameters of dependency injection like lifetime that do matter and shouldn’t be overlooked.
2. Stop debugging in Visual Studio.
3. Temporarily comment out **container.Verify()** in **SimpleInjectorInitializer.cs** under **App\_Start** folder within the **MyShuttle.Web** project.
4. Right-click the **MyShuttle.Web** project and select **Debug** **>** **Start New Instance.**
5. If everything is working properly,you should now see a blank page with **“Hello World!”** message.
6. Stop debugging in Visual Studio.
7. In the next task, you will look into how to enable MVC content to be processed and returned by the hosting server/process.

Task 2: Enabling MVC and run a simple test

Before you can test our MVC application, you need to setup a “View” for our controllers. You will go into the details about Views in the next lab. For testing purposes, you will just create a simple View page.

1. Comment out the **Application\_BeginRequest** method in **Global.asax.cs** (located under **Global.asax** – unfold it if not visible).
2. Observe the **MyShuttle.Web** project. It already has a **Views** folder and **Home** and **CarrierList** subfolders because you added identical namedcontrollers to the project.
3. By convention, the **Home** folder will contain Views related to our **HomeController** class. Again, that’s why it was created when you added our **HomeController** class to the project.

Right-click the **Home** folder and select **Add** **> View**.In the **Add View** dialog box, enter **Index** as the view’s name. Leave everything else on the defaults. Click the **Add** button.



1. Open the **Index.cshtml** file and add the following markup script and save the file:

@model MyShuttle.Web.Models.MyShuttleViewModel

@{

ViewBag.Title = "Home Page";

}

@Model.MainMessage

This will display the message that you have set in the **HomeController** from Exercise 1.

1. MVC requires a number of configuration / initialization steps, which are commonly placed into separate classes within the **App\_Start** folder. These have been already created for you when adding the first controller also installed the MVC NuGet packages. Let’s have a look at them.

* **BundleConfig** class is responsible for minification and packaging of JavaScript files and libraries used by the views.
* **FilterConfig** class contains code for registering global filters – ie. code that gets the chance to run for all requests. The default code adds a generic error handling filter.
* MVC needs to know which request it’s allowed to handle. These are called routes. In the MVC template defines a **RouteConfig** class for this purpose.

**Note**: Routes play a critical role in MVC, make sure you understand them.

The first statement asks MVC to ignore requests that refer to an .axd file – these represent HTTP handlers used for special purposes.

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

The second statement defines the default route and the way it maps to controllers, actions and a certain parameter. It even defines default values for these parts should they be missing from the requested URL:

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

**Note:** Make sure that the default route corresponds to this one above. When not using the Visual Studio wizard to create the MVC app (like in our case), the default value for controller (Home) might be missing.

1. Next, you need to let MVC actually know about these initialization classes by adding an event handler for **Application.Start** event in **Global.asax.cs** as follows:

protected void Application\_Start()

{

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

Add the missing namespaces as well:

using System.Web.Mvc;

using System.Web.Optimization;

using System.Web.Routing;

**Note**: You are not ready to run the application just yet. Remember that in the *HomeController*, you have injected the *CarrierRepository* service. So the controller is expecting the concrete service to be passed in. However, you have not setup all the data context configuration yet.

Only for this test run, you would need to comment out the HomeController constructor and the Post methodto avoid any dynamic data references.

1. Temporarily comment out the **HomeController** constructor and the Post method to avoid any dynamic data references and test MVC middleware.
2. Right-click the **MyShuttle.Web** project and select **Debug** **>** **Start New Instance.**

You should see a page with the message **“The Ultimate B2B Shuttle Service Solution”**.

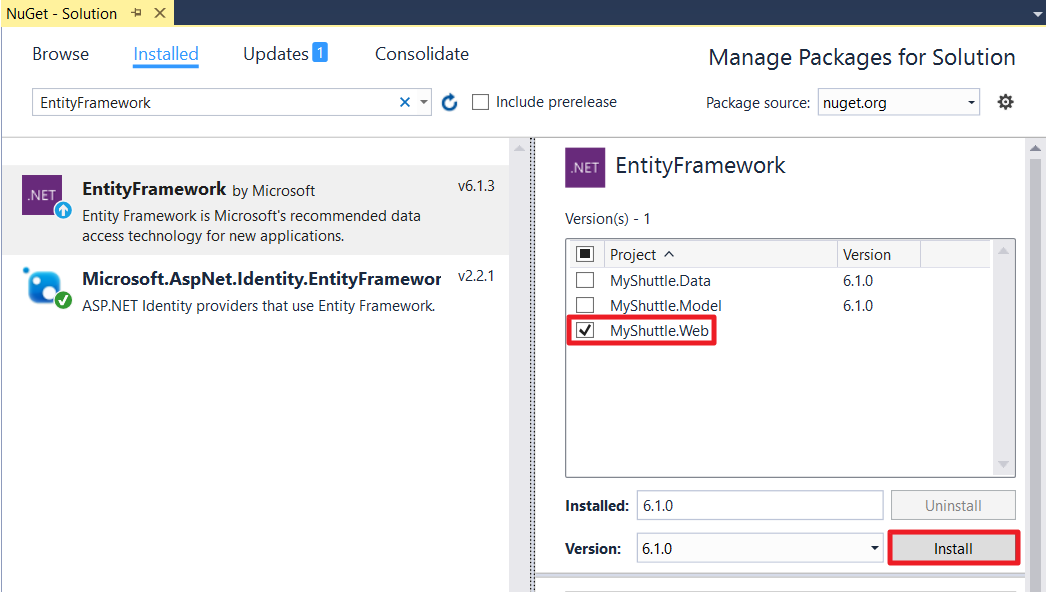
You have successfully got an ASP.NET MVC application running.

1. Stop debugging in Visual Studio.

**Note:** Before moving onto the next task, make sure to remove the comment for the constructor the Post method *in HomeController*, and for container.Verify(); in *SimpleInjectorInitializer.cs*.

Task 3: Setup the rest of the project for testing data driven features

1. Our next step is to test the data driven part of the application. But you need to setup properly all the data dependencies.
2. Install EntityFramework into the **MyShuttle.Web** project by right-clicking the **MyShuttle** solution and selecting **Manage NuGet Packages for Solution**. Go to the **Installed** tab, and select the **EntityFramework** package (it should be there as the **MyShuttle.Data** and **MyShuttle.Model** projects are already using it). In the right-hand pane, install it for the **MyShuttle.Web** project, too.



1. Add a Database connection string configuration.  
   Add the following in the **web.config** file in **MyShuttle.Web** under the **configSections** element (don’t confuse this web.config with the one under the Views folder):

<connectionStrings>

<add name="DefaultConnection" connectionString="Data Source=(LocalDb)\MSSQLLocalDB; Initial Catalog=MyShuttle;Integrated Security=True"

providerName="System.Data.SqlClient" />

</connectionStrings>

You will use the **LocalDB** to store our Carrier data.

1. Add the following in the **web.config** file under **entityFramework** element to ensure database initializer is triggered:

<contexts>

<context type="MyShuttle.Data.MyShuttleContext, MyShuttle.Data">

<databaseInitializer type="MyShuttle.Data.MyShuttleDataInitializer, MyShuttle.Data" />

</context>

</contexts>

1. Register the DataContext for dependency injection in **SimpleInjectorInitializer.cs** in the **InitializeContainer** method:

container.Register<MyShuttle.Data.MyShuttleContext>(() => new MyShuttle.Data.MyShuttleContext(), Lifestyle.Scoped);

1. Under the **Views** folder, right-click ***CarrierList***and select **Add** **>** **View**.In the **Add View** dialog box, enter **Index** as the view’s name. Leave everything else on the defaults. Click the **Add** button.Replace the code in the newly added **Index.cshtml** with this:

@model MyShuttle.Web.Models.CarrierListViewModel

@{

ViewBag.Title = "Index";

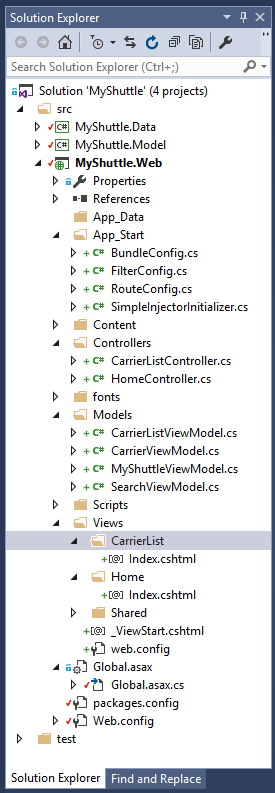
}

<h2>Index</h2>

@Html.Raw(@Model.CarrierList.Count)

**Note**: **CarrierList** folder will contain views related to **CarrierListController** class which was created when you added the controller itself.

1. The **MyShuttle.Web** project should look like this:



1. You finally can test our application again.
2. Right-click the **MyShuttle.Web** project and select **Debug** **>** **Start New Instance.**

You should see a page with the message **“The Ultimate B2B Shuttle Service Solution”**.  
In the browser address bar, append */carrierlist/index* and press the **Enter** key. You should see the number **1**.

1. Stop debugging in Visual Studio.