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

Module 5: Web API

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>

Internet Explorer, Microsoft, SQL Server, Visual Studio, and Windows 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 5: Implementing Web API 1](#_Toc480803621)

[Exercise 1: Setup the Web API Project 6](#_Toc480803622)

[Exercise 2: Implement the First API Controller 9](#_Toc480803623)

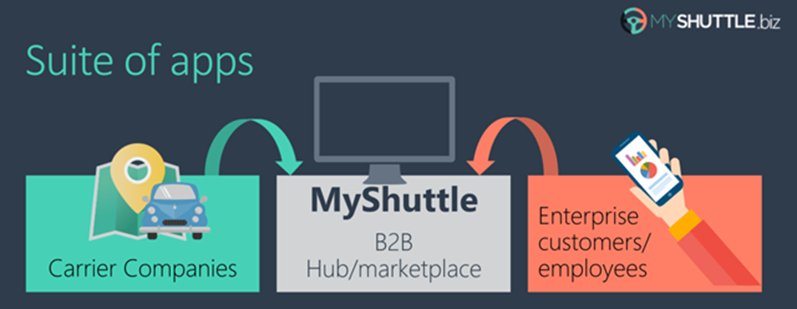
[Exercise 3: Implement the Analytics API Controller 15](#_Toc480803624)

# Lab 5: Implementing Web API

#### Introduction

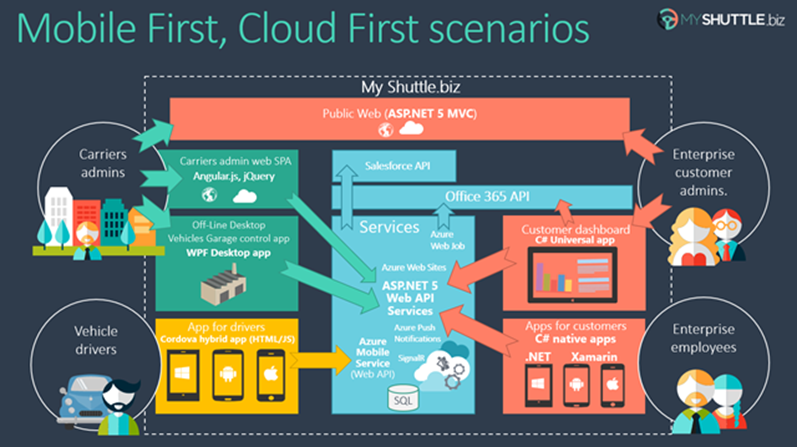
The aim of this exercise is to explore the role of views, how to create views and learn Razor syntax.

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:

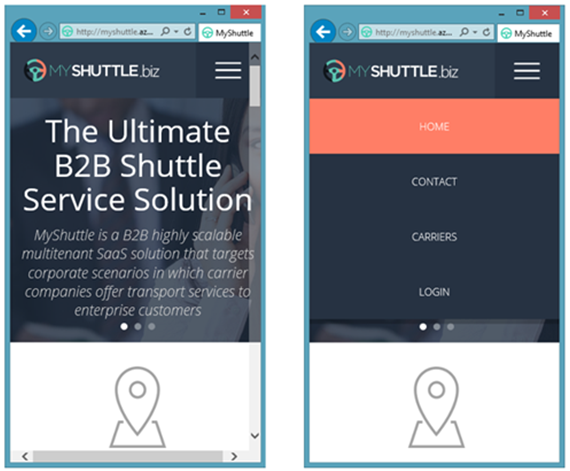


There are two web applications in the above scenario:

**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.

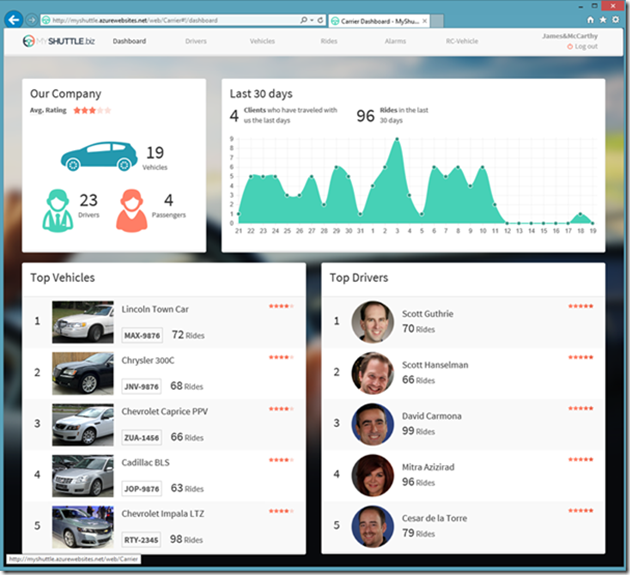




**Private Web Application:**

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.

#### Overview

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). Typically, 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 checkboxes 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 (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 enables 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) for creating an application.

Then, 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 that (as with Web-forms-based applications) 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.

#### Objectives

After completing this lab, you will be able to:

* Understand the basic principles of MVC.
* Learn how to implement a view in Razor view engines.
* Learn how to pass data from controllers to views.
* Learn how to implement GET, POST scenarios.

#### Prerequisites

None

#### Scenario

In this scenario, we will explore Web API Controllers.

#### 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

Exercise 1: Setup the Web API Project

#### Objectives

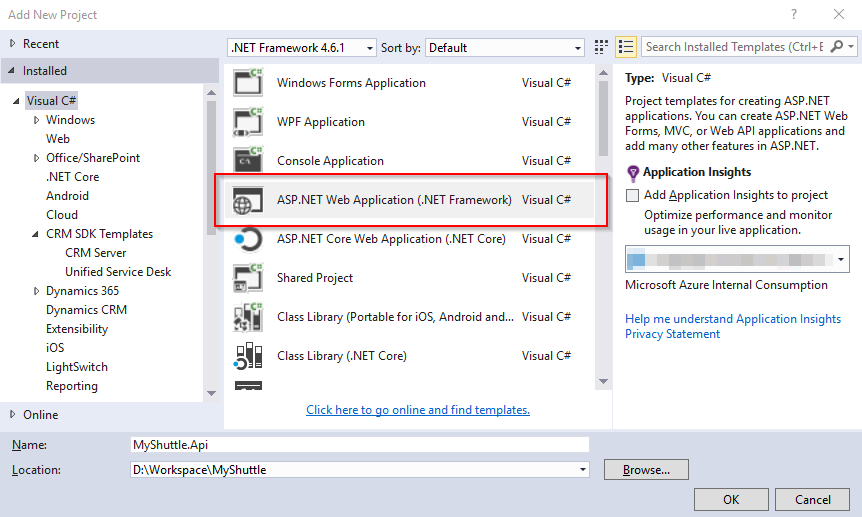
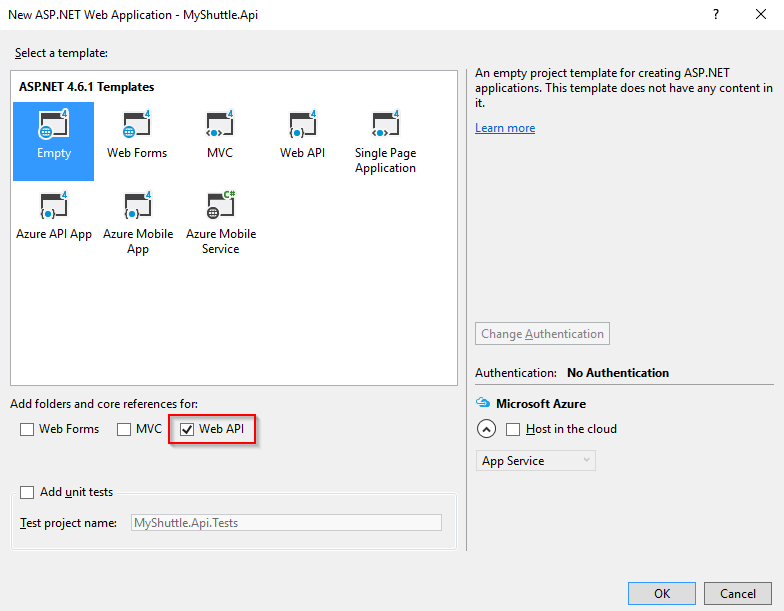
In this exercise, you will:

* Create a new ASP.NET Web API project in Visual Studio 2017.
* Setup the structure for the API project and setup the correct references between projects in the solution.

#### Scenario

Creating a new API class library and setting up the references.

Task 1: Create the new project

1. Open Visual Studio 2017.
2. Open the solution located in *Labs/Module 05 – WebApi/Begin/MyShuttle*. You should see three projects listed under the the solution: **MyShuttle.Data**, **MyShuttle.Model** and **MyShuttle.Web**.
3. Right-click **solution** and click **Add** > **New Project.**
4. Name the project as **MyShuttle.Api** and click **OK**.
5. Select Empty template and select Web API checkbox  
   
6. The API project itself will reference our Data and Model projects, so add these as references to the MyShuttle.Api project in the same way:
   1. Right-click **MyShuttle.Api** and click **Add** > **Reference.**
   2. Select **MyShuttle.Data** and **MyShuttle.Model**, and then click **OK.**
7. Add the structure for the API source files. Add three folders to the **MyShuttle.Api** project (Right-click **Project**, and click **Add** > **New Folder**), called:
   1. Constant  
      (for configuration files)
   2. Filters  
      (where we will store all our Http Filter classes)
   3. ServiceContainer  
      (where we will create the IoC container and DependancyResolver)
8. We need to add **AppDependencyResolver** and **AppServiceContainer** to the ServiceContainer folder:
   1. Right-click ServiceContainer and click Add > Existing Item…
   2. Browse to the ServiceContainer folder in assets (assets/MyShuttle.Api/ServiceContainer)
   3. Select AppDependencyResolver.cs and AppServiceContainer.cs then click Add.
9. To register our IoC container to be used as a dependency resolver we need to add the following code to **Global.asax** code behind in Application\_Start() event before line “GlobalConfiguration.Configure(WebApiConfig.Register);”:

var container = new AppServiceContainer();

container.AddService<IDriverRepository, DriverRepository>();

container.AddService<IVehicleRepository, VehicleRepository>();

container.AddService<ICarrierRepository, CarrierRepository>();

container.AddService<IRidesRepository, RidesRepository>();

GlobalConfiguration.Configuration.DependencyResolver = new AppDependencyResolver(container);

1. Add the following namespaces to the using block in **Global.asax** code behind:

using MyShuttle.Api.ServiceContainer;

using MyShuttle.Data;

1. adding the Entity Framework package to our project:
   1. Right-click **MyShuttle.Api** and selectManage Nuget Packages, Select Entity Framework and Click install.
   2. Go to web.config and add the code below inside the configuration Tag:

<connectionStrings>

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

providerName="System.Data.SqlClient" />

</connectionStrings>

Exercise 2: Implement the First API Controller

#### Objectives

In this exercise, you will:

* Create API Controllers.
* Implement a CRUD (Create, Read, Update, and Delete) API using HttpPost, HttpPut, HttpDelete and HttpGet actions.

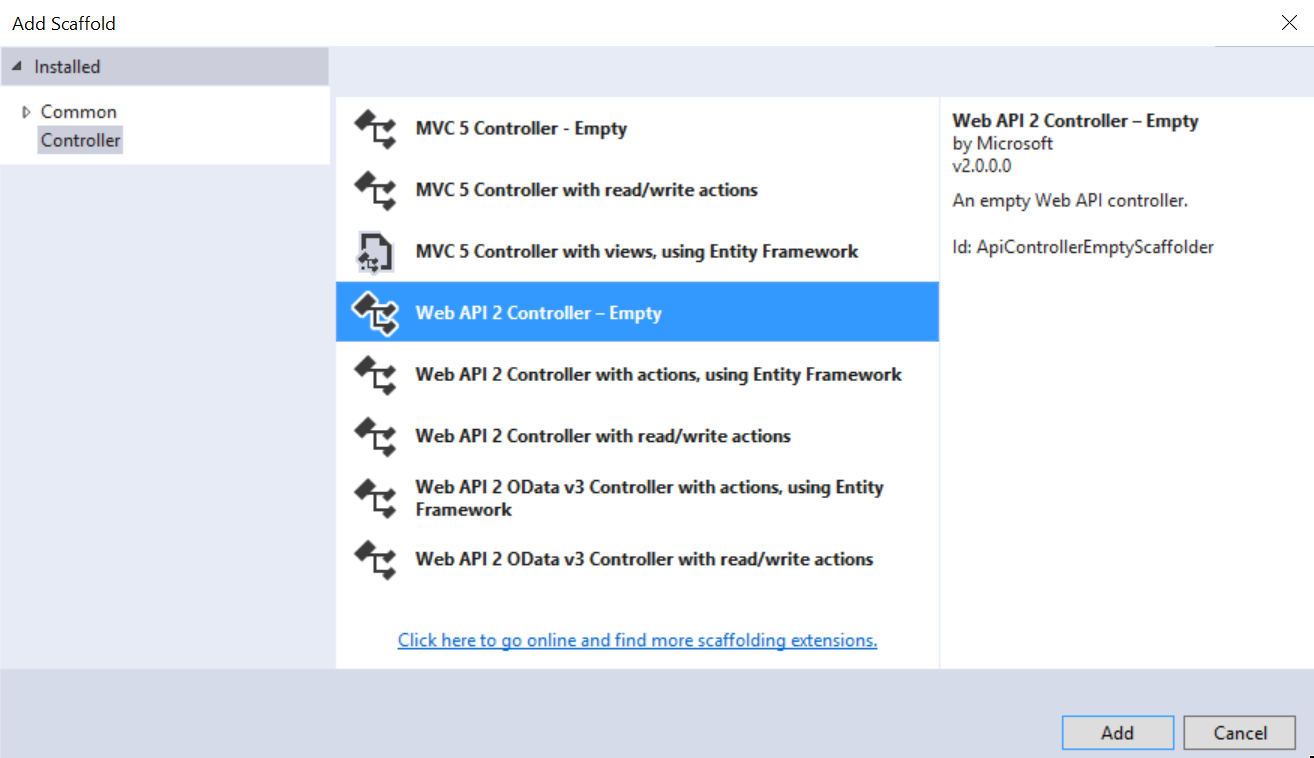
#### Scenario

Our API will support two resources: Analytics and Drivers. Analytics is a simple read-only API that returns dashboard level details about the data within our repositories. Drivers is a CRUD controller that will allow an external caller to read and write Driver data in the system.

Task 1: Implement DriversController

In a similar way to the UI part of the application, Controllers are used to handle requests coming in to the application.

Each controller represents a grouping of API functions that can be called via the routing framework. Usually a controller gives access to data and functionality scoped to a single type of resource. In our project, we will create a controller to access the Drivers in the system, and another for accessing Analytics data, to provide a feed to the dashboard.

1. We will now add our two new controllers to the class library.
   1. Right-click **Controllers** folder, and click **Add** > **Controller.**
   2. Select the **Web API 2 Controller - Empty** template.  
      
   3. Name the class **DriversController**, and click **Add**.
2. **DriversController** should derive from **ApiController** base class.

public class DriversController : ApiController

{ }

1. The new class derives from ApiController.
2. Add the using statement to your DriversController source file, and while we are there, add a reference to MyShuttle.Data and MyShuttle.Model as well:

using MyShuttle.Data;

using MyShuttle.Model;

using System.Threading.Tasks;

1. Now we can implement the constructor, which will use dependency injection to receive an instance of IDriverRepository, and setup the member properties:

IDriverRepository \_driverRepository;

private const int DefaultCarrierID = 0;

public DriversController(IDriverRepository driverRepository)

{

\_driverRepository = driverRepository;

}

1. The implementation of the CRUD operations thereafter is straightforward, with the note that these calls have been implemented using Async to ensure performance is optimal:

public async Task<Driver> Get(int id)

{

return await \_driverRepository.GetAsync(DefaultCarrierID, id);

}

[ActionName("search")]

public async Task<IEnumerable<Driver>> Get(string filter, int pageSize, int pageCount)

{

if (String.IsNullOrEmpty(filter))

filter = string.Empty;

return await \_driverRepository.GetDriversAsync(DefaultCarrierID, filter, pageSize, pageCount);

}

[ActionName("filter")]

public async Task<IEnumerable<Driver>> GetDriversFilter()

{

return await \_driverRepository.GetDriversFilterAsync(DefaultCarrierID);

}

[ActionName("count")]

public async Task<int> GetCount(string filter)

{

if (String.IsNullOrEmpty(filter))

filter = string.Empty;

return await \_driverRepository.GetCountAsync(DefaultCarrierID, filter);

}

[HttpPost]

public async Task<int> Post([FromBody]Driver driver)

{

driver.CarrierId = DefaultCarrierID;

return await \_driverRepository.AddAsync(driver);

}

[HttpPut]

public async Task Put([FromBody]Driver driver)

{

driver.CarrierId = DefaultCarrierID;

await \_driverRepository.UpdateAsync(driver);

}

[HttpDelete]

public async Task Delete(int id)

{

await \_driverRepository.DeleteAsync(id);

}

Task 2: Add Custom Action Routing

The current state of the code should compile and run. There are a couple of things that need attention though. If you run the app, you can now navigate to the new API URLs, for example try: <http://localhost:5000/api/drivers/GetCount>, you should see a page with nothing but the number of drivers displayed. The URL is based on the server address and port, the controller name, the action, which in this case is the method name GetCount, and the parameter list (string filter).

The action name can be overridden simply by using the ActionNameAttribute. The MVC routing engine will use the overridden action name if there is one, when parsing the URL to determine which method to call, rather than the method name.

1. For the methods which do something, other than the standard Get, Insert, Update, Delete operations, add an ActionNameAttribute to override the default Action. Add the route name "search" to the Get(string, int, int) method:

[ActionName("search")]

public async Task<IEnumerable<Driver>> Get(string filter, int pageSize, int pageCount)

{ … }

1. Similarly add the ActionName attribute with a value of "filter" to GetDriversFilter() and "count" to GetCount(string filter).

Now run the app again, and attempt to navigate to *http://localhost:5000/api/drivers/GetCount* - you should get an Http 404 (Not found) error. But try *http://localhost:5000/api/drivers/count* and your action will be executed successfully.

Task 3: Disable HttpCaching on all API Calls

Many browsers will use client side caching to improve performance and reduce the number of network calls that need to be made. This behavior is enabled by default in many browsers, so to disable it, we will use a filter on our controller to add a header to the response that our APIs will return.

1. Add the **NoCacheAttribute** class from the assets folder to your project's **Filters** folder.
2. Right-click Filters and click Add > Existing Item…
3. Browse to the Filters folder in assets (assets/MyShuttle.Api/Filters)
4. Select NoCacheAttribute.cs and click Add.
5. Understand the Filter's code – its straightforward: the class overrides "OnActionExecuted" of the base class (ActionFilterAttribute) – checks to see if the HttpStatus is OK (200), and no cacheControl header has been set already. It then adds two headers (Pragma and CacheControl) to disable the cache, and sets the Expires header value to yesterday. These tactics will disable caching for the majority of scenarios.
6. Add the attribute to the DriversController

[NoCache]

public class DriversController : Controller

{ … }

1. Add the following name space to the using block:

using MyShuttle.Api.Filters

1. Build your app to confirm it all still compiles correctly.

Exercise 3: Implement the Analytics API Controller

#### Objectives

In this exercise, you will complete the API by adding the final controller and its helper class.

#### Scenario

The AnalyticsController is reasonably simple – it uses the existing repository interfaces to query summary information and return structured data as the result.

Task 1: Implement Analytics controller

1. Add the existing **GlobalConfig** file from assets:
2. Right-click the **Constant** folder, and click **Add** > **Existing Item…**
3. Browse to the folder Assets\MyShuttle.Api\Constant, select **GlobalConfig.cs** and click **Add**.
4. The **GlobalConfig** is a simple helper class used by the **AnalyticsController** to provide a central place for configuration – in this case we are providing the default number of items to return when requesting the "Top" drivers or vehicles.
5. Add the Analytics Controller class to the project
   1. Right-click the **Controllers** folder, and click **Add** > **Existing Item…**
   2. Browse to the folder *Assets\MyShuttle.API\Controllers*, select **AnalyticsController.cs** and click **Add**.
6. Examine the **AnalyticsController** class – it should all make sense based on the **DriverController** code completed already. Note again the use of async for performance best practice.
   1. The constructor uses dependency injection to receive the repositories.
   2. The action names are overridden to provide an easier to consume URL for consumers.
   3. The controller builds models from the result of the repository calls, and returns them as structured data.
   4. Note there is no need to convert the data models to XML or JSON, the framework handles this on your behalf.
7. Build and run the app now that it is complete.
8. Navigate to *http://localhost:5000/api/analytics/summary* to see a JSON formatted output from your GetSummaryInfo() call. Note the structure matches the SummaryAnalyticInfo model in the MyShuttle.Models project.
9. The final project should look like the following screenshot:

