.NET Core: Developing Cross-Platform Web Apps with ASP.NET Core – Workshop*PLUS*

Module 02: Models

Student Lab Manual

Instructor Edition (Book Title Hidden Style)

Version 3.0

**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.

© 2019 Microsoft Corporation. All rights reserved.

**Copyright and Trademarks**

© 2019 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 2: Creating Model 1](#_Toc480185374)

[Exercise 1: Create MyShuttle Model 5](#_Toc480185375)

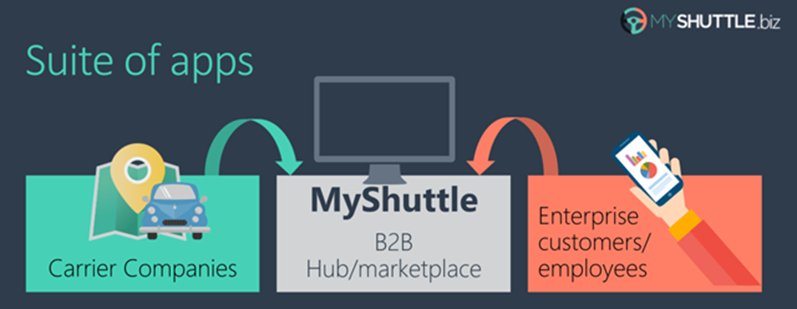
[Exercise 2: Implement MyShuttle Data CRUD Logic 15](#_Toc480185376)

[Exercise 3: Perform Unit Testing 22](#_Toc480185377)

# Lab 2: Creating Model

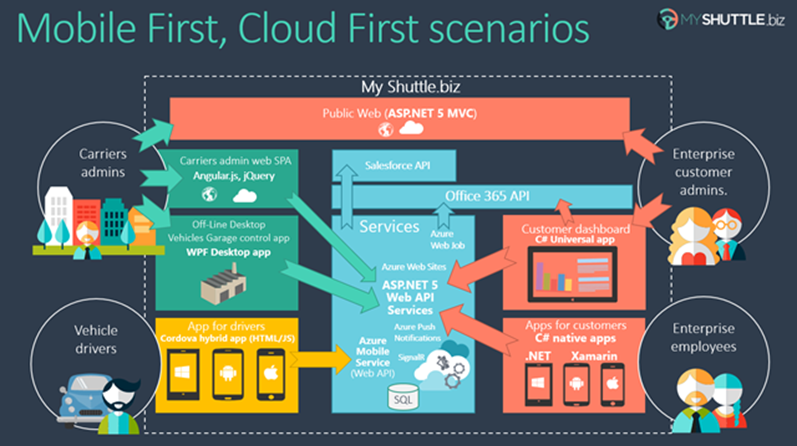
#### Introduction

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 between their company and the carrier company for that ride.

The global scenario is described in the following image:

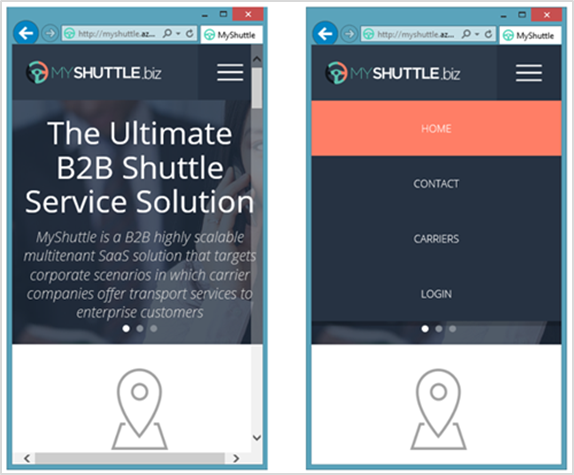


There are two web applications in the above scenario:

**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.

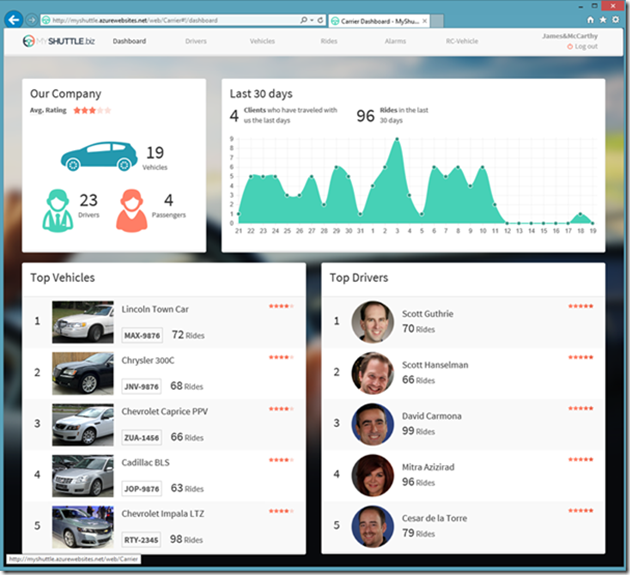




**Private Web Application:**

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 Core 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.

#### Objectives

This lab will show you how to:

* Create a new ASP.NET Core application in Microsoft Visual Studio 2019.
* Create Visual Studio projects for application model.
* Create application model using code-first technique.

#### System Requirements

To complete this lab, you need:

* Microsoft Visual Studio 2019 or higher

#### Hosted Lab Credentials

If the lab is performed 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 MyShuttle Model

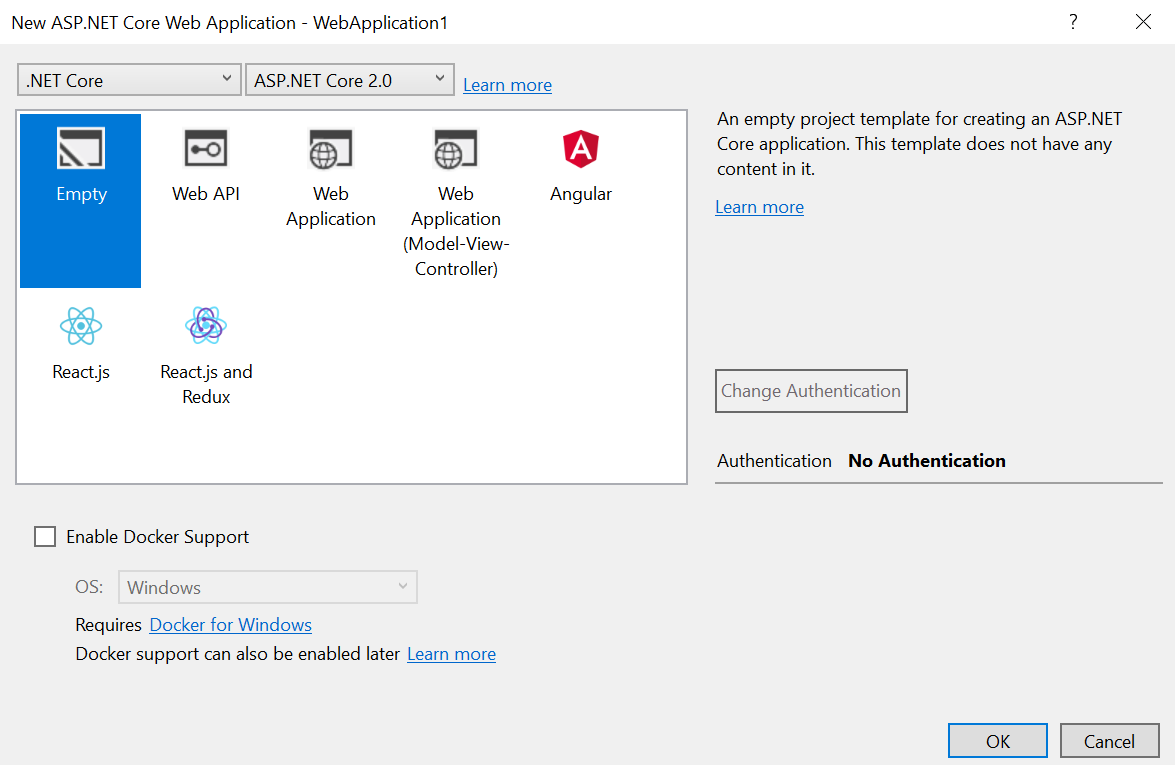
#### Objectives

In this exercise, you will:

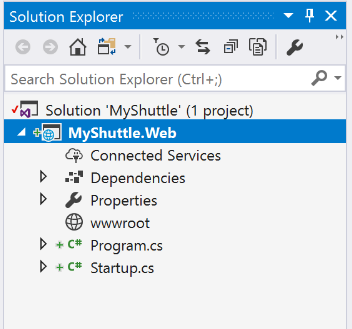
* Create a new Visual Studio solution for MyShuttle application.
* Create a project to implement MyShuttle model through code-first technique.

Task 1: Create the Visual Studio Solution

1. Open Visual Studio 2019.
2. Create a new ASP.NET Core application project by going to **File** > **New > Project**.
3. Name the project as **MyShuttle**, select the **Visual C#**/**Web**/**ASP.NET Core Web Application** template.   
     
   Create a new solution, and leave the **Create directory for solution** option selected, and then click **OK**.
4. Choose **Empty** project template.



1. Close or **Cancel the Configure Microsoft Azure Web App** pop-up window, if one appears.
2. Rename the default project as **MyShuttle.Web.**

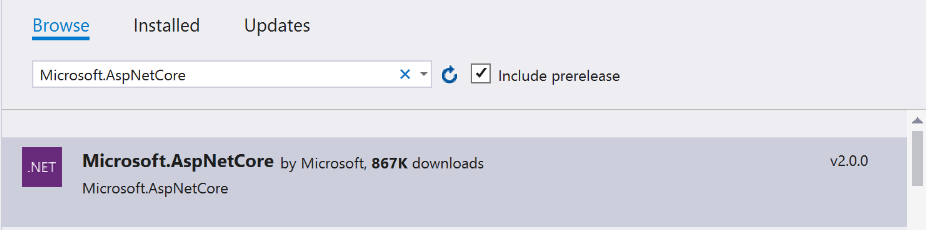


Since you chose .NET Core, the application only targets .NET Core out of the box.

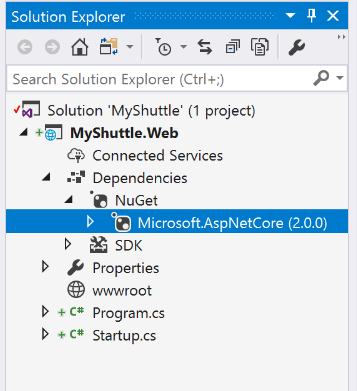
1. In our MyShuttle.Web project, under **Dependencies** > **NuGet**, remove the “**Microsoft.AspNetCore.All**” metapackage.

Then right-click on **MyShuttle.Web** > **Manage NuGet Packages** and under “**Browse**” search for and install “**Microsoft.AspNetCore**” instead.

Accept any license agreements dialogs that pop up.



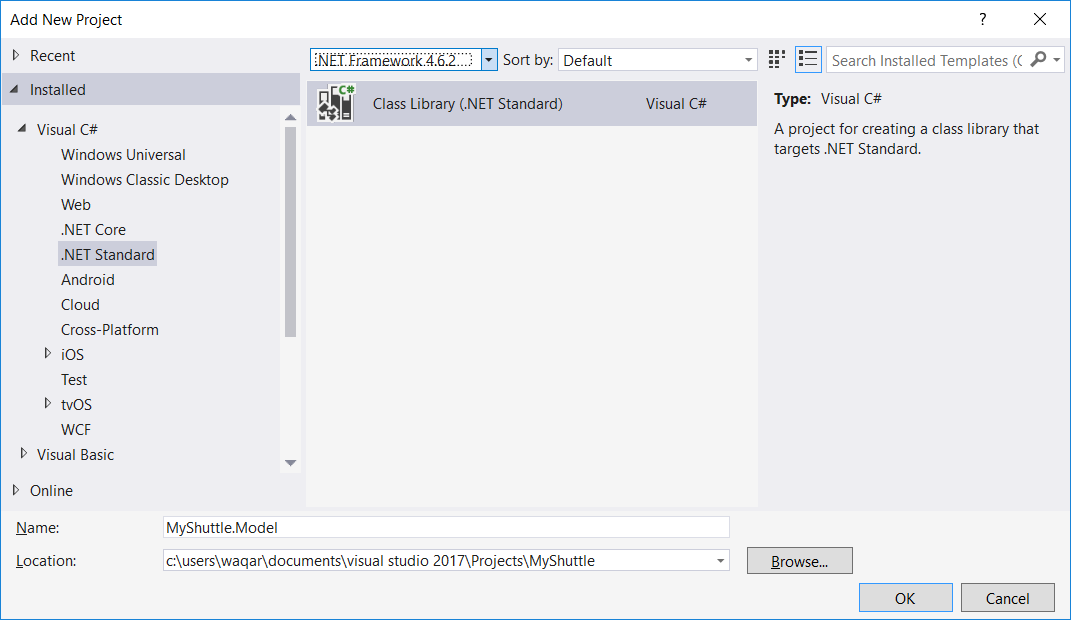
Your solution explorer should like this:



You’ve created your application, and now you are ready to start creating your models!

Task 2: Create the Model Project

1. Right-click **MyShuttle** solution in the **Solution Explorer** window and then click **Add** 🡪 **New Project**.
2. Choose **Visual C# 🡪 .NET Standard 🡪 Class Library (.NET Standard)** project template and name it **MyShuttle.Model**, and then click **OK**.



1. Delete Class1.cs.

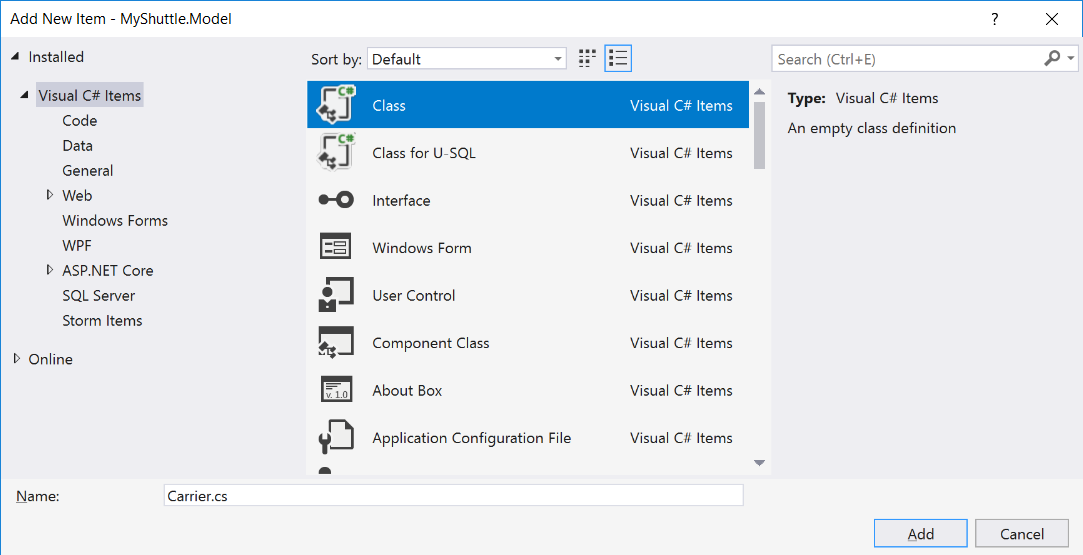
Note: It is recommended to use the Class Library (.NET Standard) over the Class Library (.NET Core), unless you need to use features that are only available in .NET Core. This lets you reuse your library across different types of applications.

We’ve finish creating a Class Library project to house our models. Let’s move on to implementing our models!

Task 3: Implement the Model Project

We need to implement a carrier model class to represent the carrier.

1. Right-click the **MyShuttle.Model** project and click **Add** 🡺 **New Item**.
2. Choose **Class** from available templates and name it **Carrier.cs**



1. Remove all the “using” statements above the namespace block, apart from **“System.Collections.Generic”** which will be used shortly.
2. Make sure the class is *public*.

public class Carrier

{

}

1. Make note of the carrier information in the database, such as its name, description, address, email, logo, etc.

Add the following fields to **Carrier** model class, to store carrier information. You will add foreign key collections later.

public class Carrier

{

public int CarrierId { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public string CompanyID { get; set; }

public string Address { get; set; }

public string ZipCode { get; set; }

public string City { get; set; }

public string State { get; set; }

public string Country { get; set; }

public string Phone { get; set; }

public string Email { get; set; }

public byte[] Picture { get; set; }

public double RatingAvg { get; set; }

}

1. We’ve just finished creating our first model class!

But you also need to create models for the enterprise company, company’s employees, carrier’s vehicles and drivers, and rides that enterprise company’s employees take. Additionally, you will also add model for performing analytics.

Instead of writing code for each model entity, add the following files from the respective ***…/Assets/MyShuttle.Model*** folder by **Add** 🡺 **Existing Item**.

ApplicationUser.cs: User of the application, connected to carrier;

Customer.cs: Customer enterprise company;

Driver.cs: Registered driver of the shuttle company who drives the vehicle;

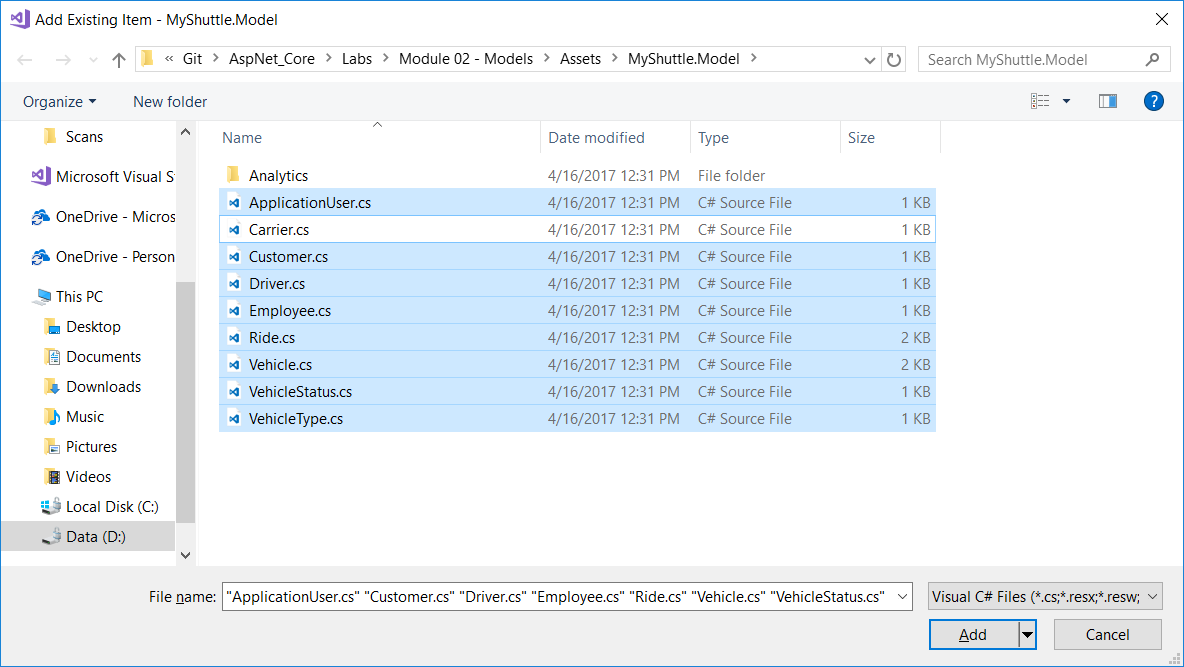
Employee.cs: Employee of the enterprise company who rides/rents a vehicle;

Ride.cs: Record of trip taken by an employee of a company in a vehicle driven by a driver;

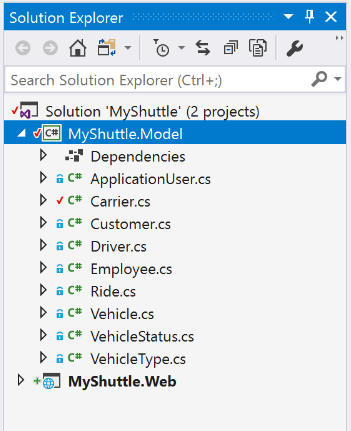
Vehicle.cs: Vehicle driven by a Driver;

VehicleStatus.cs: Current status of a vehicle, whether it’s available, busy, or unknown;

VehicleType.cs: Type of vehicle, such as compact, van, luxury, etc.



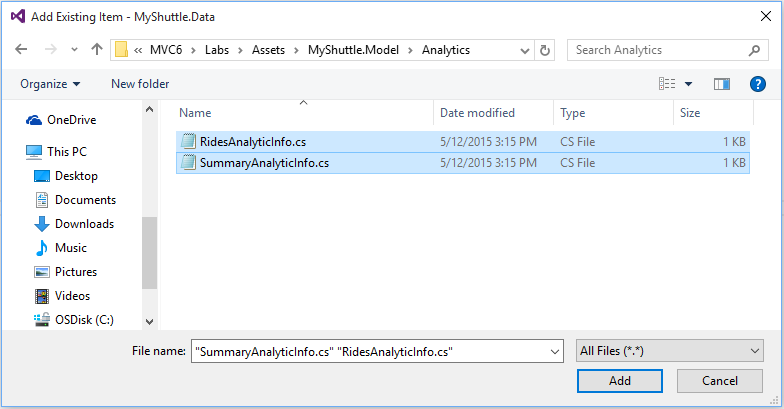
Your file structure should look like this:



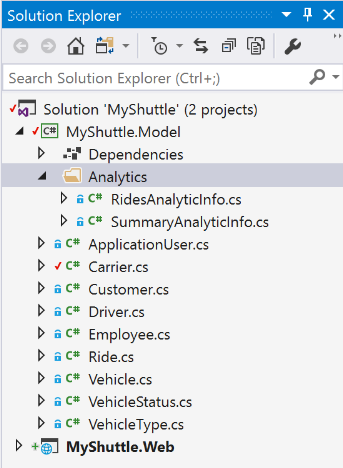
1. Now that you have all the model entities defined, you will also like to perform some analytics on the database records. To hold analytical data, we need to define model for it.

Add a new folder named **Analytics** under the **MyShuttle.Model** project.

1. Right-click the **Analytics** folder and click **Add** 🡺 **Existing Item**. Add the two files from the **…/Assets/MyShuttle.Model/Analytics** folder.

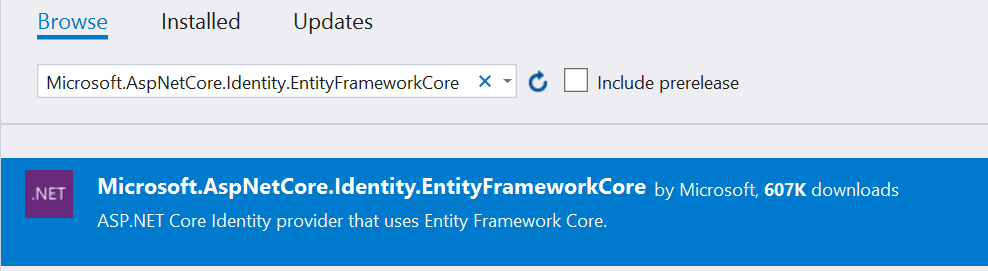


Your file structure should look like this:



1. We would like to add the **Identity** dependency for the project.   
     
   Before ASP.NET Core, you would have added either a direct assembly reference or a NuGet package. Beginning with ASP.NET Core, direct assembly reference is not possible. We will need to add a NuGet package instead. Use the NuGet Package Manager to install the dependency.

Right-click the **MyShuttle.Model** project in Solution Explorer and choose **Manage NuGet Packages**. Search for **Microsoft.AspNetCore.Identity.EntityFrameworkCore** (version 2.0.0). Click **Install** and accept the license agreement dialog box.



1. Each carrier has vehicles and drivers associated with it.   
     
   You did not add any collection references before because model classes for vehicles and drivers did not exist then. Now that they have been added to the project, add the following collections at the bottom of **Carrier** class.

public class Carrier

{

. . .

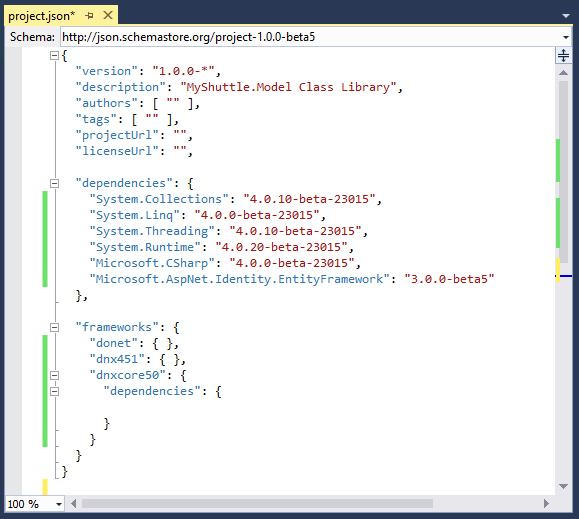
public ICollection<Vehicle> Vehicles { get; set; }

public ICollection<Driver> Drivers { get; set; }

}

1. Take the time to go through all the model classes and understand the attributes they have and how they are connected to each other.
2. The Core CLR is a lean and completely modular runtime for ASP.NET Core projects. This CLR has been re-designed into components so you have the flexibility to include only those features that you need in your app. You add the components as NuGet packages. When you are finished, your app is dependent only on required features.

By re-factoring the runtime into separate components, Microsoft can help deliver improvements to the components more quickly because each component is updated on its own schedule. The Core CLR is about 11 megabytes instead of around 200 megabytes for the full .NET CLR. The Core CLR can be deployed with your app and different versions of the Core CLR can run side-by-side. Hence, if possible, it is better to target Core CLR instead of full .NET CLR because of all the good reasons mentioned above.



1. Build the application to ensure that it compiles successfully.

You have now completed defining the model! Now you are ready to implement data repository pattern to perform database CRUD operations.

Exercise 2: Implement MyShuttle Data CRUD Logic

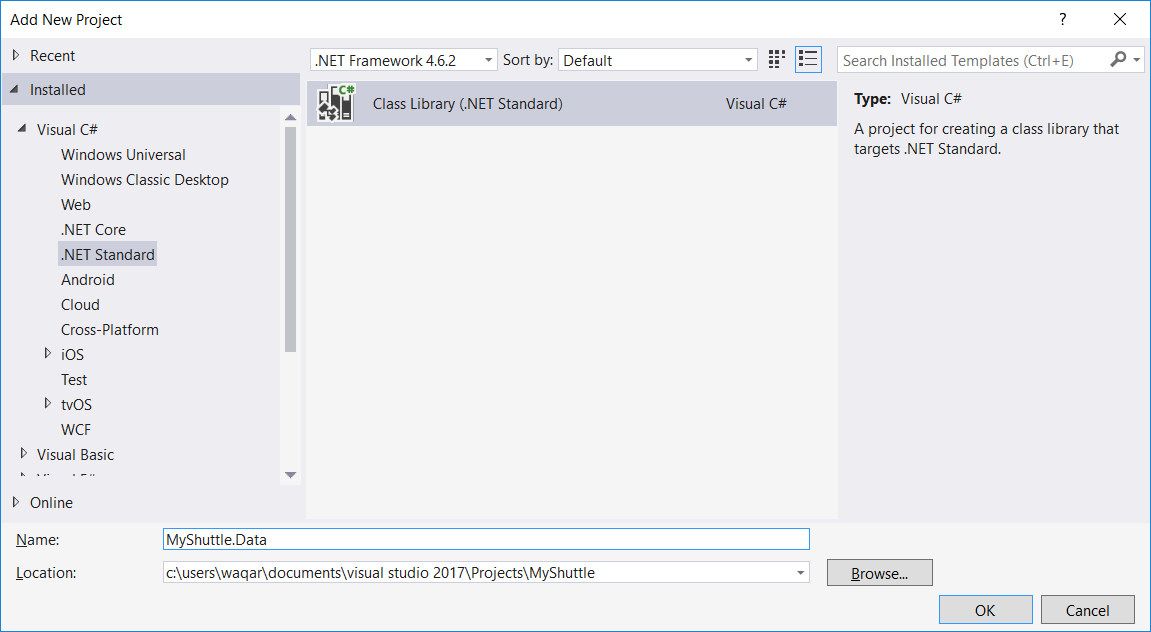
#### Objectives

In this exercise, you will:

* Create a project to implement repository pattern for database CRUD operations.
* Create dummy data for this app.

Task 1: Create the Data Project

1. If it is not already open, open **MyShuttle** solution in **Visual Studio 2019**.
2. Right-click **MyShuttle** solution in the **Solution Explorer** window and then click **Add > New Project**.
3. Choose **.NET Standard** > **Class Library (.NET Standard)** project template and name it **MyShuttle.Data**. Click **OK**.



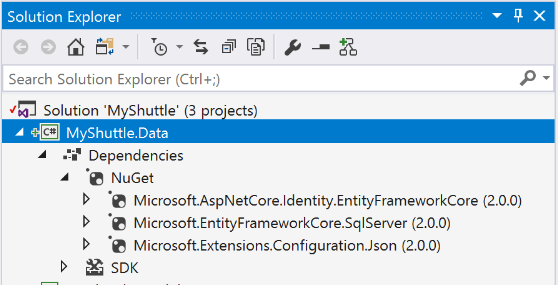
1. Delete Class1.cs.

Now we’re ready to implement the logic to initialize data.

Task 2: Implement the Data Initialization Logic

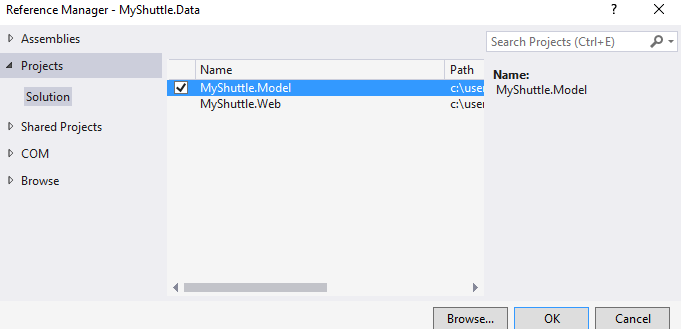
1. You would first like to define dependencies for this project. Add the dependencies from NuGet to the MyShuttle.Data project:
   * **Microsoft.AspNetCore.Identity.EntityFrameworkCore** (version 2.0.0)
   * **Microsoft.Extensions.Configuration.Json** (version 2.0.0)
   * **Microsoft.EntityFrameworkCore.SqlServer** (version 2.0.0)

Your dependencies node will look like this:

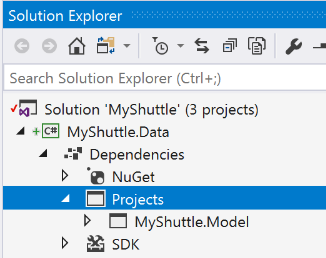


1. Add a reference to **MyShuttle.Model** to your project:

Right-click **MyShuttle.Data** > **Add** > **Reference**.   
Go to the **Projects** tab, and check the box for **MyShuttle.Model**.



You’ll now see a reference to it under the Dependencies/Projects node:



You’ll see that your dependencies will be neatly organized in categories underneath the Dependencies node.

1. Now let’s add the logic for our data class.   
     
   Right-click the **MyShuttle.Data** project and click **Add** 🡺 **Existing Item**.   
   Add the **MyShuttleContext.cs** file to the project root from **…/Assets/MyShuttle.Data** folder.   
     
   This class implements the database context of Entity Framework (EF) which declares all entities and their relationships using Code First approach.
2. Right-click the **MyShuttle.Data** project and click **Add** 🡺 **New Folder**. Name the folder as **Infrastructure**.
3. Add the following two existing items under the **Infrastructure** folder from the **…/Assets/MyShuttle.Data/Infrastructure** folder:
   * **FakeImages.cs**
   * **MyShuttleDataInitializer.cs**

**FakeImages.cs:** It contains images for drivers, vehicles, etc.

**MyShuttleDataInitializer.cs:** It contains dummy data and initialization logic.

We’ve finished implementing our data initialization part! We’ve created dummy data to fill out the database initially.

Let’s move on to implementing our data repositories.

Task 3: Implement Data Repositories

1. Right-click the **MyShuttle.Data** project and click **Add** > **New Folder**. Name the folder as **Interfaces**. This folder will contain all repository interfaces.
2. Right-click the **Interfaces** folder and click **Add** > **New** **Item** to add a new class. Name it **ICarrierRepository.cs**
3. Define the interface as following:

**Important!** See that the default namespace would be *MyShuttle.Data.Interfaces*, but we have renamed this to be ***MyShuttle.Data*** for simplicity.  
  
Also note that we’ve replaced the key word “class” with “interface”. And we’ve made it ”public” so we can access it from our other projects.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using MyShuttle.Model;

namespace MyShuttle.Data

{

public interface ICarrierRepository

{

Task<int> AddAsync(Carrier carrier);

Task<SummaryAnalyticInfo> GetAnalyticSummaryInfoAsync(int carrierId);

Task<Carrier> GetAsync(int carrierId);

Task<List<Carrier>> GetCarriersAsync(string filter);

Task UpdateAsync(Carrier carrier);

}

}

**AddAsync**: Adds a new carrier to the database.

**GetAnalyticSummaryInfoAsync**: Performs some analytics and returns summary analytics like rating, total drivers, passengers, and vehicles of a carrier.

**GetAsync**: Returns a carrier record matching the input Carrier ID.

**GetCarriersAsync**: Used by the search function to return all carriers that match the input criteria.

**UpdateAsync**: Updates carrier record in the database.

1. We’ve created our first repository interface. Now let’s add the interfaces for the other repositories. Add the following existing items from the **…/Assets/MyShuttle.Data/Interfaces/**folder.

* ICustomerRepository.cs
* IDriverRepository.cs
* IEmployeeRepository.cs
* IRidesRepository.cs
* IVehicleRepository.cs

1. After defining all the interfaces, it is time to implement them. You will implement CarrierRepository on your own. The rest of them will be added from the **Assets** folder.
2. Right-click the **MyShuttle.Data** project and click **Add** > **New Folder**. Name the folder as **Repositories**. This folder will contain all repository classes.
3. Right-click the **Repositories** folder and click **Add** > **New Item** to add a new class. Name it **CarrierRepository.cs**. Click Add.
4. **CarrierRepository** should implement the **ICarrierRepository** interface.

Implement **CarrierRepository.cs** with the following code.   
  
Note that the default namespace would be *MyShuttle.Data.Repositories*, but we have renamed this to be *MyShuttle.Data* for simplicity.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using MyShuttle.Model;

using Microsoft.EntityFrameworkCore;

namespace MyShuttle.Data

{

public class CarrierRepository : ICarrierRepository

{

MyShuttleContext \_context;

static readonly int DEFAULT\_PICTURE = 0;

public CarrierRepository(MyShuttleContext dbcontext)

{

\_context = dbcontext;

}

public async Task<int> AddAsync(Carrier carrier)

{

carrier.Picture = Convert.FromBase64String(FakeImages.Carriers[DEFAULT\_PICTURE]);

\_context.Carriers.Add(carrier);

await \_context.SaveChangesAsync();

return carrier.CarrierId;

}

public async Task<SummaryAnalyticInfo> GetAnalyticSummaryInfoAsync(int carrierId)

{

var passengers = await \_context.Rides.Where(r => r.CarrierId == carrierId).Select(r => r.EmployeeId).ToListAsync();

var rating = \_context.Rides.Where(r => r.CarrierId == carrierId).Select(r => r.Rating);

return new SummaryAnalyticInfo()

{

Rating = (rating.Count() > 0) ? rating.Average() : 0,

TotalDrivers = await \_context.Drivers.Where(r => r.CarrierId == carrierId).CountAsync(),

TotalPassengers = passengers.Distinct().Count(),

TotalVehicles = await \_context.Vehicles.Where(r => r.CarrierId == carrierId).CountAsync()

};

}

public async Task<Carrier> GetAsync(int carrierId)

{

return await \_context.Carriers

.Where(c => c.CarrierId == carrierId)

.SingleOrDefaultAsync();

}

public async Task<List<Carrier>> GetCarriersAsync(string filter)

{

var carriers = \_context.Carriers.AsQueryable();

if (!String.IsNullOrEmpty(filter))

{

carriers = carriers.Where(c => c.Name.ToLowerInvariant().Contains(filter.ToLowerInvariant()));

}

return await carriers.ToListAsync();

}

public async Task UpdateAsync(Carrier carrier)

{

\_context.Carriers.Update(carrier);

await \_context.SaveChangesAsync();

}

}

}

1. Review and understand the data CRUD logic implemented by the **CarrierRepository** class.

Note that we’ve implemented the ICarrierRepository interface for the CarrierRepository repository class:

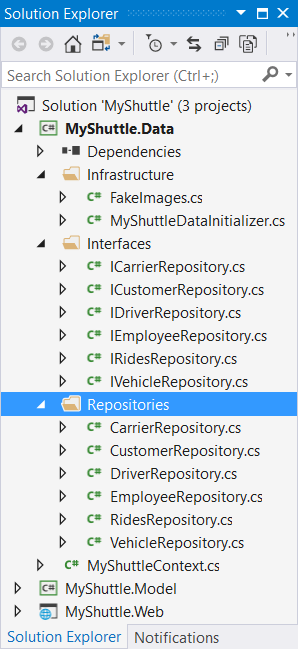
public class CarrierRepository : ICarrierRepository

We then defined each of the CRUD methods that was in our interface, and added logic to each one.

1. In order to add the remaining repositories, right-click the **Repositories** folder in the **MyShuttle.Data** project and click **Add** > **Existing Item** to add the following repositories from the folder **…/Assets/MyShuttle.Data/Repositories**:

* CustomerRepository.cs
* DriverRepository.cs
* EmployeeRepository.cs
* RidesRepository.cs
* VehicleRepository.cs

1. After adding all the repositories, the **MyShuttle.Data** project hierarchy should look like the following:



1. Compile the solution to ensure it builds successfully.

You’ve now finished implementing the repository pattern to perform database CRUD operations!

In the next lab, you will implement Controllers, which will handle user requests and respond with the model data. But we’ll create some unit tests first.

Exercise 3: Perform Unit Testing

#### Objectives

In this exercise, you will:

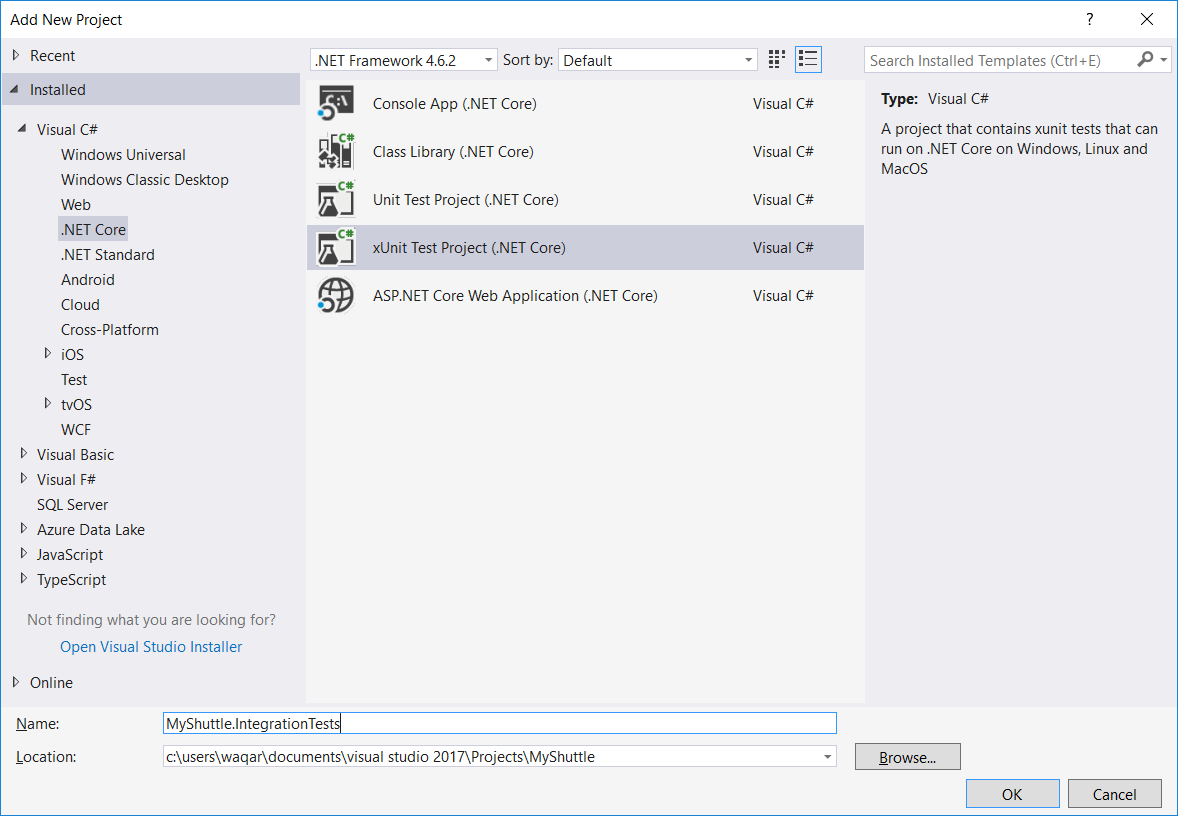
* Create a unit test to validate successful creation of the database.

Task 1: Create a Unit Test Project and Execute It

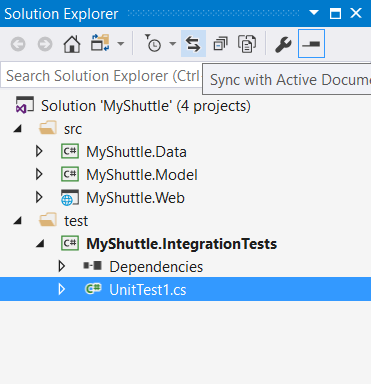
1. Create a Solution Folder under MyShuttle solution and name it **src**. All application projects will be placed under it.

Right-click the **MyShuttle** solution > **Add** > **New Solution Folder**. Name it **src**.

1. Move all three application projects under **src** folder.
2. Create a Solution Folder named **test**. All test projects will be placed under the test folder hierarchy.
3. Under the **test** folder, add a new project. Go to **Visual C# > .NET Core**, and create a new **xUnit Test Project (.NET Core)** with name “**MyShuttle.IntegrationTests**”



1. The Solution Explorer should look like the following:



1. Delete UnitTest1.cs
2. Add the following NuGet package to our **MyShuttle.IntegrationTests** project: **Microsoft.EntityFrameworkCore.InMemory** (version 2.0.0)
3. Add a project reference to **MyShuttle.Data**.
4. Now we’re ready to implement a unit test method for data context. Right-click **MyShuttle.IntegrationTests** and select **Add** > **Class**, name it ***MyShuttleContextTests.cs*** and then add the following code:

using Xunit;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using MyShuttle.Data;

namespace MyShuttle.IntegrationTests

{

public class MyShuttleContextTests

{

[Fact]

public async Task Db\_CreatedSuccessfully()

{

var optionsBuilder = new DbContextOptionsBuilder();

optionsBuilder.UseInMemoryDatabase("MyShuttleTestDb");

var context = new MyShuttleContext(optionsBuilder.Options);

var databaseCreated = await context.Database.EnsureCreatedAsync();

Assert.True(databaseCreated);

var databaseDeleted = await context.Database.EnsureDeletedAsync();

Assert.True(databaseDeleted);

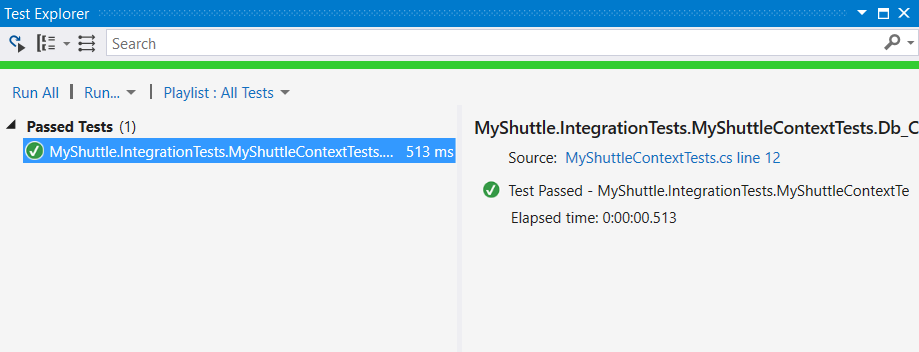
}

}

}

This code will test that we’ve successfully created a database.

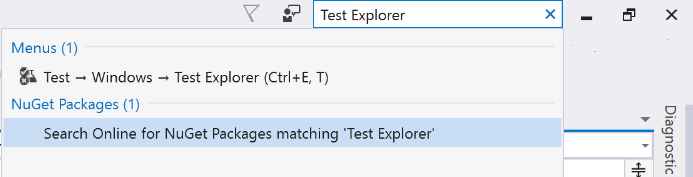
1. Build your solution. Run the test from “Test Explorer” using Visual Studio. Click “Run All”.



Note: If you can’t find your “Test Explorer” window, try searching for it in the Quick Launch bar. You can find it at the top right-hand corner of Visual Studio.

You can do this for any other windows as well.





We’ve now finished creating a test project and a unit test that validated a successful database creation.