Premier Developer

Boot Camp

ASP.NET Core Hackathon

Lab guide

March 2017

© 2017 Microsoft Corporation. All rights reserved. This document is confidential and proprietary to Microsoft. Internal use only. This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

This document is provided "as-is." Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples are for illustration only and are fictitious. No real association is intended or inferred.

Contents

[ASP.NET Core Development Hackathon 5](#_Toc480248127)

[Overview 5](#_Toc480248128)

[Hackathon Prerequisites 5](#_Toc480248129)

[Install Visual Studio 2017 and .NET Core SDK 5](#_Toc480248130)

[Create the Contoso Claim Service Solution 6](#_Toc480248131)

[Exercise 1: Complete the ContosoClaimService.Model Project. 6](#_Toc480248132)

[Task 1: Create Entity Classes for Claims Service 6](#_Toc480248133)

[Exercise 2: Complete the ContosoClaimService.Data Project. 7](#_Toc480248134)

[Tasks to complete 7](#_Toc480248135)

[Task 1: Configure Project Dependencies 7](#_Toc480248136)

[Task 2: Create A DBContext Object for The Claim Service 7](#_Toc480248137)

[Task 3: Implement A Repository Pattern 8](#_Toc480248138)

[Task 4: Create A Database Initializer to Create & Seed Claims Database 8](#_Toc480248139)

[Exercise 3: Complete the ContosoClaimService.API Project. 9](#_Toc480248140)

[Tasks to complete 9](#_Toc480248141)

[Task 1: Configure Project Dependencies 9](#_Toc480248142)

[Task 2: Configure Services and Middlewares in the .NET Core Startup Class 10](#_Toc480248143)

[Task 3: Create ViewModels and Mapping Profiles. 11](#_Toc480248144)

[Task 4: Create API Controller and Configure Routing. 11](#_Toc480248145)

[Task 5: Run and Verify Solution 12](#_Toc480248146)

[Task 6: Publish the ContosoClaimService to Local IIS. 12](#_Toc480248147)

[Create the Contoso Member Service Solution 12](#_Toc480248148)

[Exercise 1: Create the Solution using the dotnet CLI. 12](#_Toc480248149)

[Tasks to complete 12](#_Toc480248150)

[Task 1: Create the solution using dotnet sln command. 13](#_Toc480248151)

[Task 2: Create the API and Data projects using the dotnet new command 13](#_Toc480248152)

[Task 3: Add the projects to the solution using cli commands 13](#_Toc480248153)

[Task 4: Restore packages, build and run the solution using cli commands 13](#_Toc480248154)

[Exercise 2: Complete the ContosoMemberService.Data Project. 13](#_Toc480248155)

[Task 1: Configure Project Dependencies 13](#_Toc480248156)

[Task 2: Create Members Database from Script 14](#_Toc480248157)

[Task 3: Create A Model Class for the Member Object 14](#_Toc480248158)

[Task 4: Create A DBContext Object for The Project 14](#_Toc480248159)

[Task 5: Implement A Repository Pattern 14](#_Toc480248160)

[Exercise 3: Complete the ContosoMemberService.API Project. 15](#_Toc480248161)

[Tasks to complete 15](#_Toc480248162)

[Task 1: Configure Project Dependencies 15](#_Toc480248163)

[Task 2: Configure Services and Middlewares in the .NET Core Startup Class 16](#_Toc480248164)

[Task 3: Create API Controller and Configure Routing. 16](#_Toc480248165)

[Task 4: Run and Verify Solution 17](#_Toc480248166)

[Task 5: Publish the ContosoMemberService to Local IIS. 17](#_Toc480248167)

[Create the Contoso Member Dashboard Solution 17](#_Toc480248168)

[Exercise 1: Create and configure the Dashboard Solution 17](#_Toc480248169)

[Tasks to complete 17](#_Toc480248170)

[Task 1: Configure Project Dependencies 17](#_Toc480248171)

[Task 2: Create ViewModels, services and register with the asp.net core services container 19](#_Toc480248172)

[Task 3: Create controllers and views for dashboard and claim detail 21](#_Toc480248173)

[Task 4: Implement cookie middleware authentication 24](#_Toc480248174)

[Task 5: Publish the ContosoDashboard to Local IIS. 26](#_Toc480248175)

# ASP.NET Core Development Hackathon

## Overview

Contoso Patient Portal is a modern, electronic health record (EHR) patient portal which provides members with 24x7 access to health information recorded during doctor visits. Within the portal, Contoso members can securely view health records, manage claims, review medications, diagnoses and more. The portal integrates patient data from a network of related APIs and third-party services.

Contoso Claim Service is one such service which will be created, registered and consumed by the patient portal. The appointment management service includes several operations to view, create, modify and cancel appointments.

In this Hackathon, you will construct the Contoso patient portal and related claims service using .NET Core, ASP.NET Core and Entity Framework Core. You’ll implement MVC features such as View Components and Tag Helpers as well as middlewares such as Logging, Authentication, Exception Handling and Telemetry. You’ll configure and register services in the Startup class, leverage the built-in .NET Core dependency injection framework and implement environment variables. Finally, you’ll explore framework targeted and self-hosted deployment models, side by side deployment capability and cross-platform support for windows, linux and iOS platforms.

## Hackathon Prerequisites

## **Install Visual Studio 2017 and .NET Core SDK**

Install Visual Studio Community 2017 or any other 2017 version. Select the Community download and the default installation. Skip this step if you have Visual Studio 2017 installed.

* Install [Visual Studio 2017 IDE](https://www.visualstudio.com/en-us/visual-studio-homepage-vs.aspx)
* Install [.NET Core SDK and Runtime](https://www.microsoft.com/net/download/core)
  + .NET Core SDK (contains .NET Core 1.0 and 1.1)
  + .NET Core 1.1.1 runtime (Current)
* Install [SQL Server Express with Management Tools](https://www.microsoft.com/en-us/sql-server/sql-server-downloads)
  + SQL Server Express 2014
* Local Internet Information Services (IIS) enabled
* Optional: Create a [Visual Studio Team Services](https://www.visualstudio.com/team-services/) project to facilitate revision control and deployment.

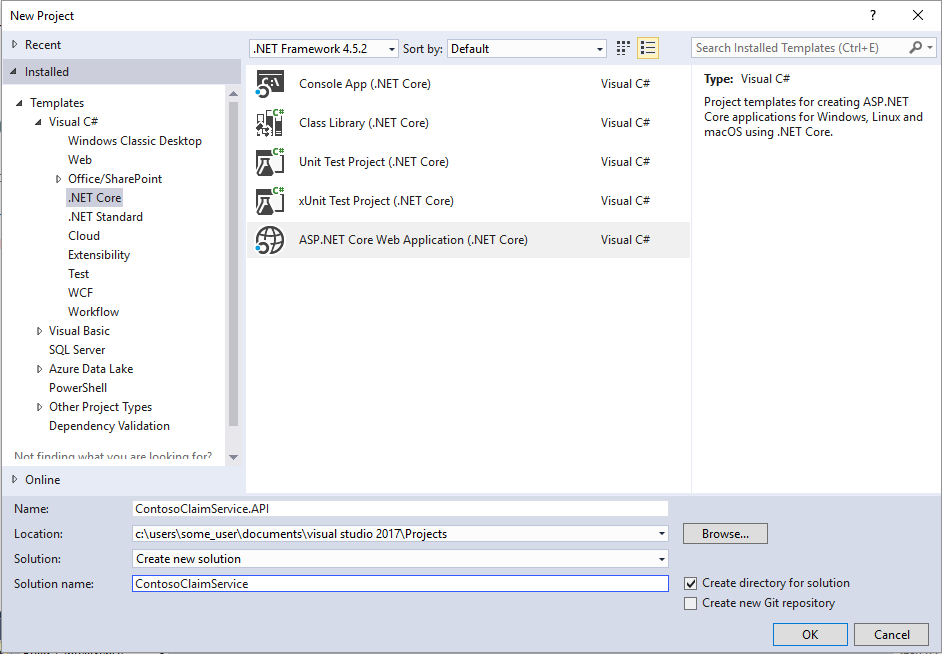
**Twelve-Factor App: I. Codebase**

A twelve-factor app is always tracked in a version control system, such as Git, Mercurial, or Subversion. A copy of the revision tracking database is known as a code repository. Track your code from idea to release with [Visual Studio Team Services](https://www.visualstudio.com/team-services/git/), a powerful DevOps environment, which supports Git and TFVC source control.

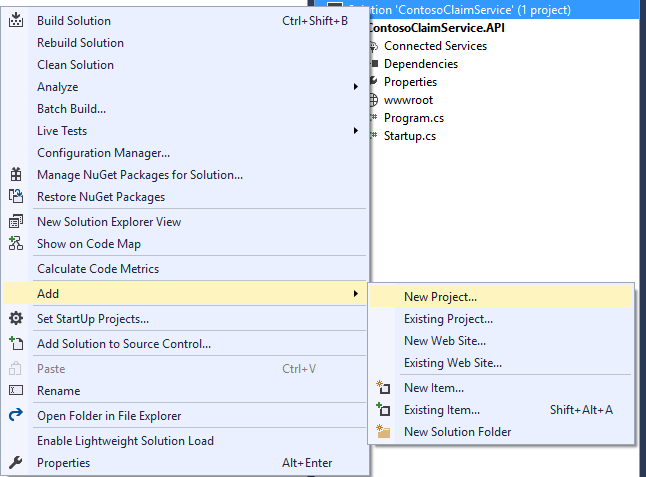
<https://12factor.net/codebase>

## **Create the Contoso Claim Service Solution**

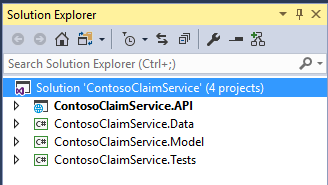
1. Create an empty ASP.NET Core Web Application with a project name of **ContosoClaimService.Api** and a solution name of **ContosoClaimService**



1. Right-click the solution name in the Solution Explorer to Add a .NET Standard Class Library project named **ContosoClaimService.Model**



1. Add a .NET Standard Class Library project named **ContosoClaimService.Data**
2. Add a .NET Core Unit Test project named **ContosoClaimService.Tests**
3. After creating these four projects, your solution should resemble the following:



**Introducing the .NET Standard**

.NET Standard solves the code sharing problem for .NET developers across all platforms by bringing all the APIs that you expect and love across the environments that you need: desktop applications, mobile apps & games, and cloud services:

* .NET Standard is a set of APIs that all .NET platforms have to implement. This unifies the .NET platforms and prevents future fragmentation.
* .NET Standard 2.0 will be implemented by .NET Framework, .NET Core, and Xamarin. For .NET Core, this will add many of the existing APIs that have been requested.
* .NET Standard 2.0 includes a compatibility shim for .NET Framework binaries, significantly increasing the set of libraries that you can reference from your .NET Standard libraries.
* .NET Standard will replace Portable Class Libraries (PCLs) as the tooling story for building multi-platform .NET libraries.

## Exercise 1: Complete the ContosoClaimService.Model Project.

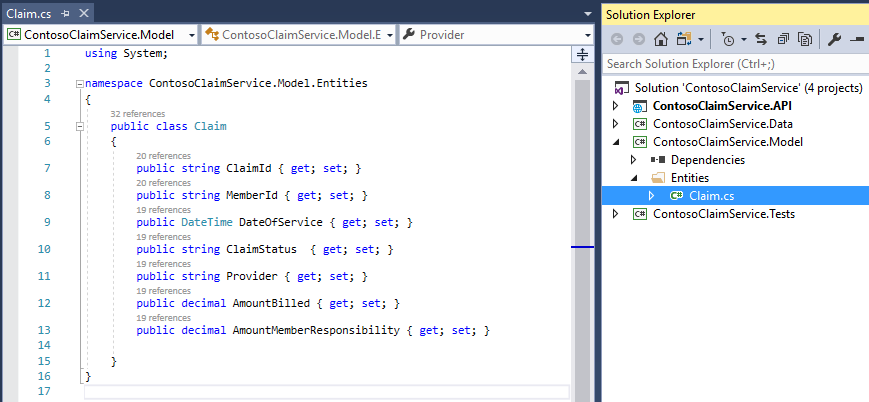
Duration: 5 Minutes

### Task 1: Create Entity Classes for Claims Service

1. Add the following entities class to a new folder called **Entities:**

Claim

ClaimId(string), MemberId(string), DateOfService(DateTime), ClaimStatus (string), Provider(string), AmountBilled(decimal) and AmountMemberResponsibility(decimal)



## Exercise 2: Complete the ContosoClaimService.Data Project.

Duration: 20 Minutes

### Tasks to complete

* Configure Project Dependencies
* Create A DBContext Object for The Claim Service
* Implement A Repository Pattern
* Create A Database Initializer to Create & Seed Claims Database

### Task 1: Configure Project Dependencies

1. Install the **Microsoft.EntityFrameworkCore** (version 1.1.2) NuGet package
2. Install the **Microsoft.EntityFrameworkCore.SqlServer** (version 1.1.2) NuGet package

**Twelve-Factor App: II. Dependencies**

Libraries installed through a packaging system can be installed system-wide (known as “site packages”) or scoped into the directory containing the app (known as “vendoring” or “bundling”). **A twelve-factor app never relies on implicit existence of system-wide packages.** It declares all dependencies, completely and exactly, via a dependency declaration manifest.

Using Nuget for library dependencies goes a long way toward dependency declaration and isolation. The .csproj file associated to each project can be read as a dependency declaration.

<https://12factor.net/dependencies>

1. Add a Project reference to the **ContosoClaimService.Model** project

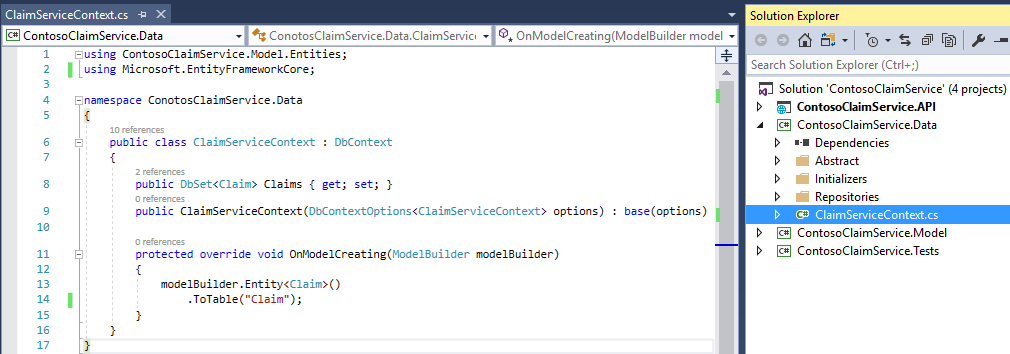
### Task 2: Create A DBContext Object for The Claim Service

**Note**: DbContext is an important part of Entity Framework. It is a bridge between your domain or entity classes and the database. DbContext is the primary class that is responsible for interacting with data as object.

1. Create a ClaimServiceContext class which inherits from DbContext
   1. Implement DbSet for the Claim entities.
   2. Implement an empty constructor with DbContextOptions as a parameter and inherit from the base object.
   3. Override **OnModelCreating** method with the following modelBuilder configuration:

modelBuilder.Entity<Claim>().ToTable("Claim");

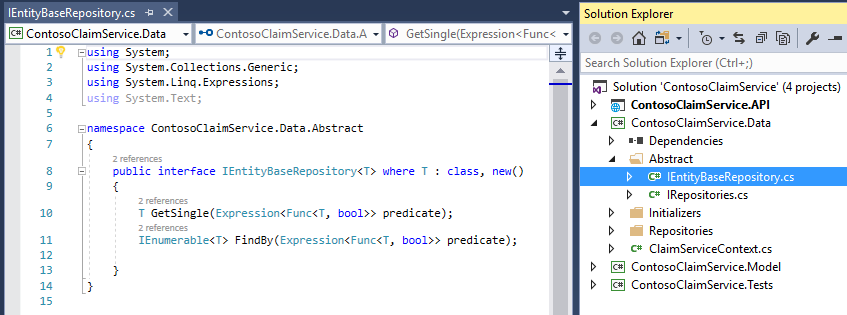
* 1. Your completed ClaimServiceContext class should resemble the following:



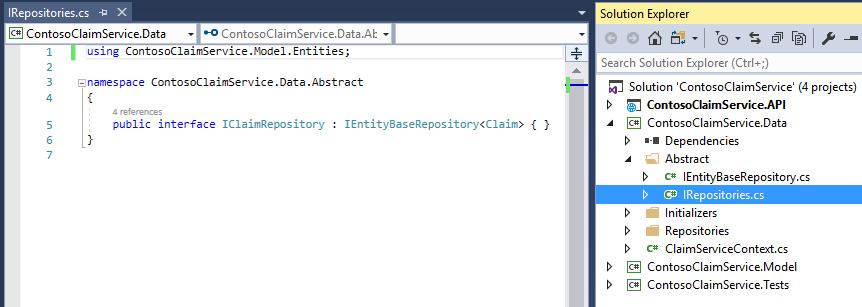
### Task 3: Implement A Repository Pattern

1. Implement the repository pattern by creating an IEntityBaseRepository interface in a new folder called **Abstract**.
   1. Add operation: T GetSingle(Expression<Func<T, bool>> predicate);
   2. Add operation: IEnumerable<T> FindBy(Expression<Func<T, bool>> predicate);
2. Implement the IEntityBaseRepository interfaceby creating the EntityBaseRepository class in a new folder called **Repositories**
   1. Inject the ClaimServiceContext into the EntityBaseRepository constructor
   2. Implement the GetSingle() method using the context to return T.
   3. Implement the FindBy() method using the context to return IEnumerable<T>
3. Create a **IRepositories** class in the **Abstract** folder
   1. Within the class, create an interface called IClaimRepository inheriting from IEntityBaseRepository, where T is Claim
4. Implement theIClaimRepository interfaceby creating the ClaimRepository class in the **Repositories** folder.
   1. In addition to implementing the IClaimRepository interface, inherit from the EntityBaseRepository class, where T is Claim
   2. Inject the ClaimServiceContext into the ClaimRepository constructor
   3. The inherited methods GetSingle() and FindBy() methods will be sufficient for data access needs.
5. Your repository classes and interfaces should resemble the following:

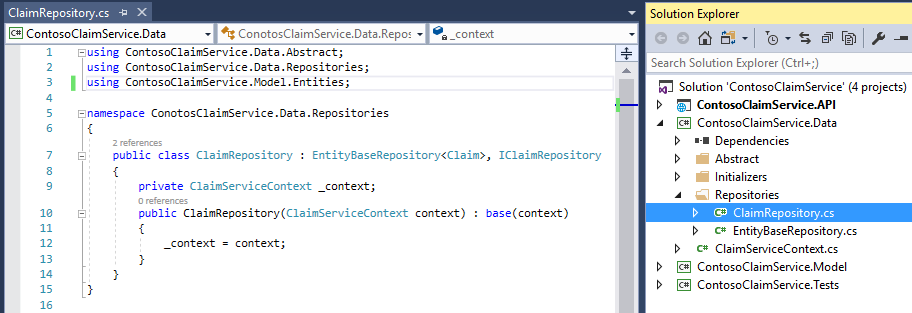
IEntityBaseRepository



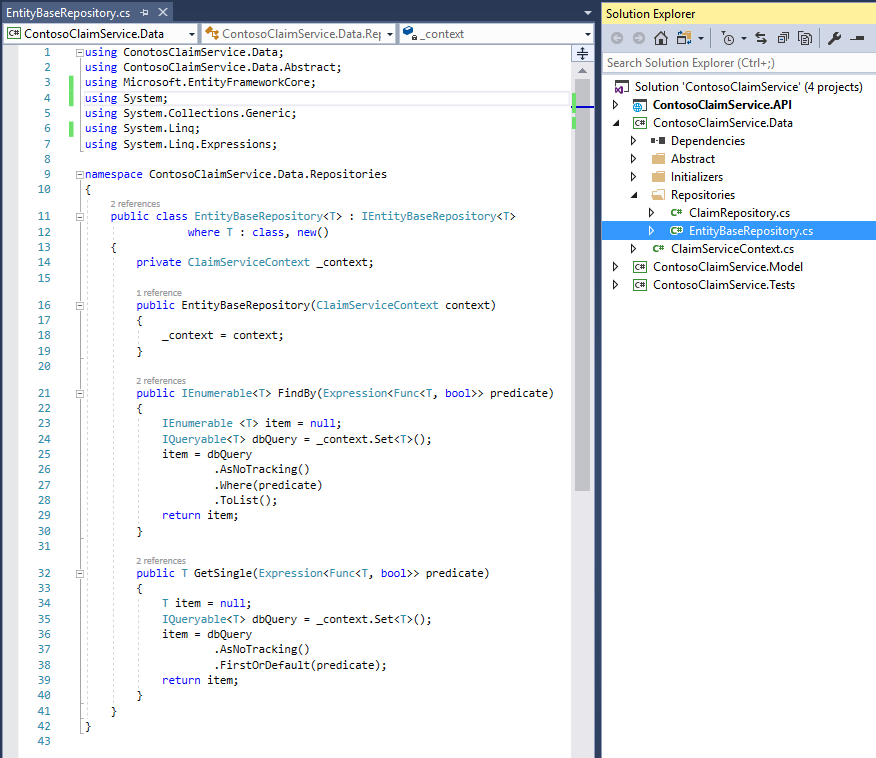
IRepositories



ClaimRepository



EntityBaseRepository



### Task 4: Create A Database Initializer to Create & Seed Claims Database

1. Finally, create a static ClaimDbInitializer class inside a new folder called **Initializers** to generate and seed the related claims database.
   1. Inject the ClaimServiceContext into a static **Initialize** method with a single parameter of IServiceProvider
   2. Set the context equal to the serviceProvider.GetService return object casted to ClaimServiceContext**.**
   3. Create a second static method called **InitializeClaims** and call it from the **Initialize** method
   4. Within the **InitializeClaims** method
      * Call the Database.EnsureCreated() operation from the context
      * Check if the claims database has already been seeded and exit processing if so.
      * Create Claims and seed data to populate the database.

IList<Claim> claims = new List<Claim>{

new Claim() {

ClaimId = "840000120001",

MemberId = "1",

DateOfService = new DateTime(2017, 3, 8),

ClaimStatus = "In Progress",

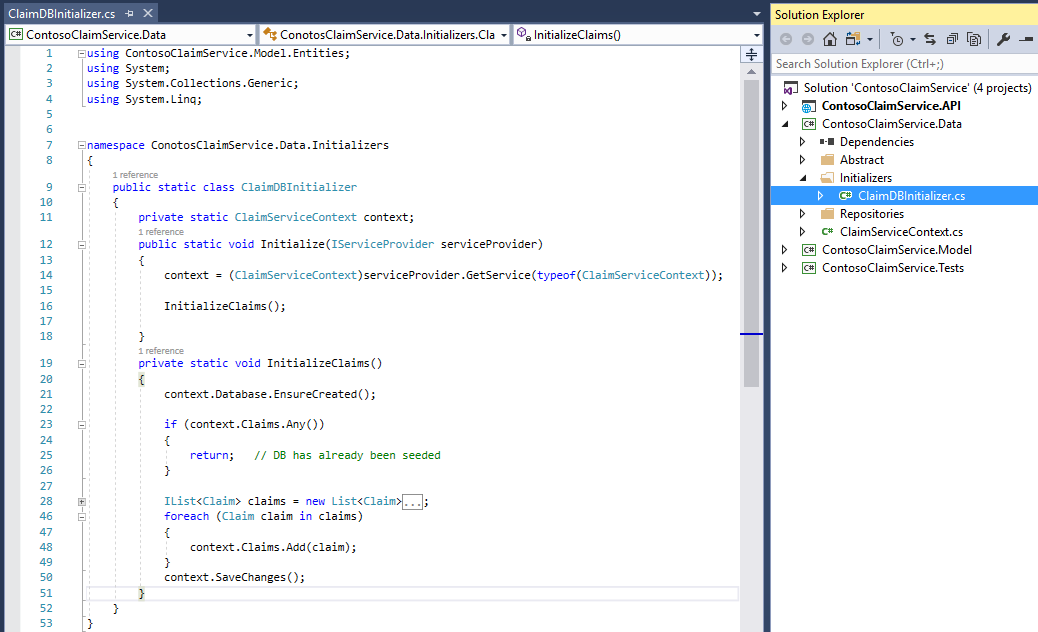
Provider = "Walgreens #12273",

AmountBilled = 378.00m,

AmountMemberResponsibility = 24.75m },

...}

* + - Call context.SaveChanges();
  1. Your completed ClaimDbInitializer should resemble the following:



## Exercise 3: Complete the ContosoClaimService.API Project.

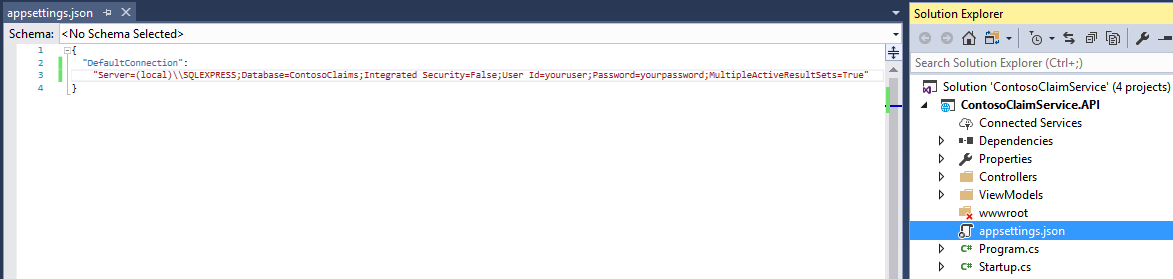
Duration: 30 Minutes

### Tasks to complete

* Configure Project Dependencies
* Configure Services and Middlewares in the .NET Core Startup Class
* Create ViewModels and Mapping Profiles
* Create API Controller and Configure Routing
* Run and Verify Solutions

### Task 1: Configure Project Dependencies

1. Install the **Microsoft.EntityFrameworkCore** (version 1.1.2) NuGet package
2. Install the **Microsoft.EntityFrameworkCore.SqlServer** (version 1.1.2) NuGet package
3. Install the **Microsoft.AspNetCore.Mvc** (version 1.1.3) NuGet package.
4. Install the **AutoMapper** (version 6.1.0) NuGet package
5. Add a Project reference to the **ContosoClaimService.Model** project
6. Add a Project reference to the **ContosoClaimService.Data** project
7. Create an **appsettings.json** file in the project and specify your local database connectionstring: "DefaultConnection": "Data Source=(localdb)\\MSSQLLocalDB; Initial Catalog=ContosoClaims; Integrated Security=True;"



**Twelve-Factor App: III. Config**

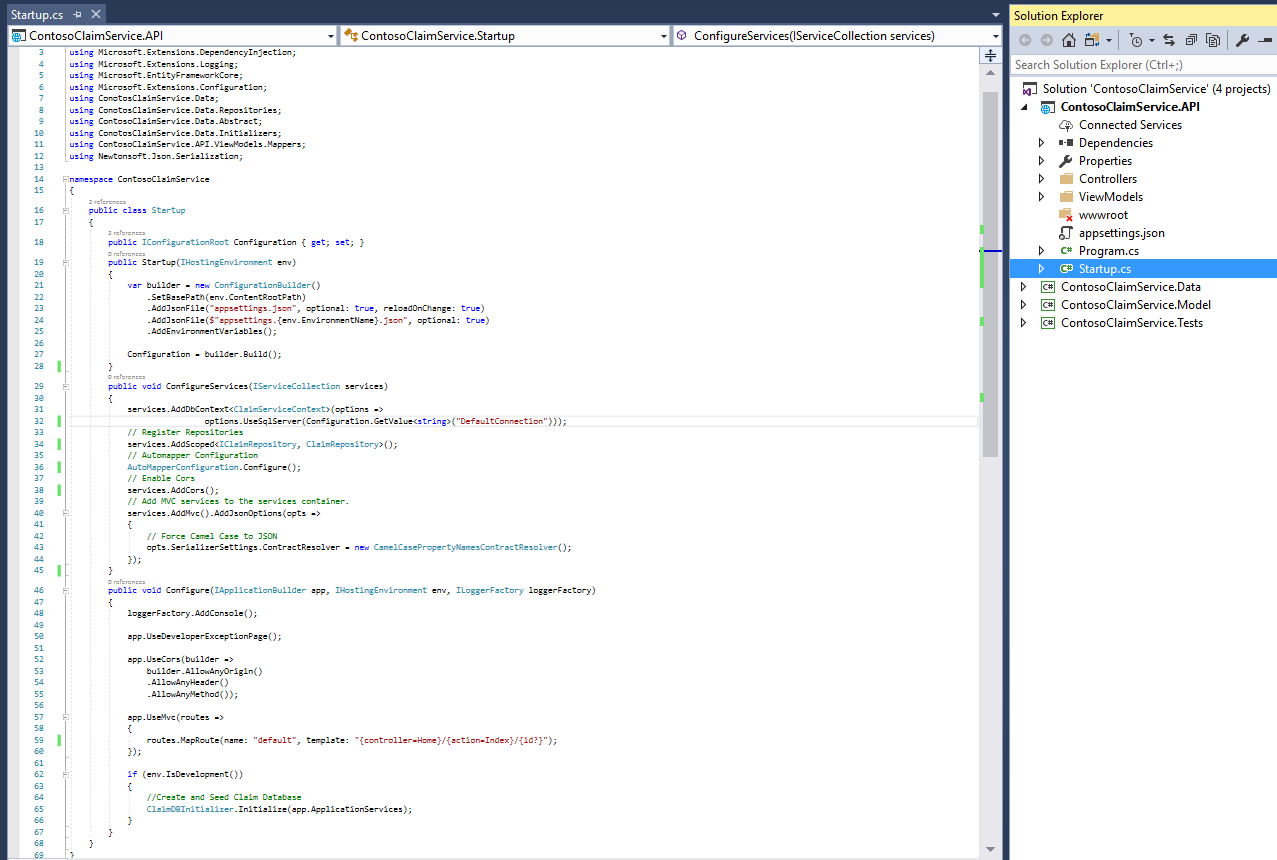
An app’s config is everything that is likely to vary between [deploys](https://12factor.net/codebase) (staging, production, developer environments, etc). Config varies substantially across deploys, code does not. Leveraging ASP.NET Core’s ConfigurationBuilder to separate environment variables from code adheres to twelve-factor.

<https://12factor.net/config>

### 

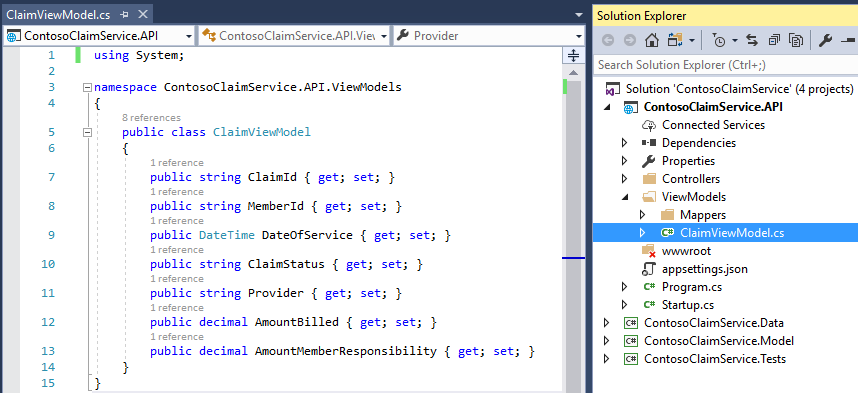
### Task 2: Configure Services and Middlewares in the .NET Core Startup Class

1. Configure the Startup class:
   1. Create a Startup class constructor with a parameter of IHostingEnvironment
   2. Create a class level IConfigurationRoot property call **Configuration.**
   3. Within the Startup constructor, implement a ConfigurationBuilder object which references the **appsettings.json** configuration file.
   4. Build the configuration and set it the **Configuration** class level property.
2. Register services to the .NET Core container in the **ConfigureServices** method of the **Startup** class:
   1. Register and add a DbContext of type ClaimServiceContext using SqlServer options and configuration value specified in the DefaultConnection property.
   2. Register the ClaimRepository service with a scoped lifetime.
   3. Register CORS support.
   4. Register the MVC service and configure camel case Json serialization.
3. Configure application middleware in the **Configure** method of the Startup class.
   1. Configure CORS to allow any origin, header and method.
   2. Configure MVC routing (use default MapRoute)
   3. Within development environment conditional, call the ClaimDbInitializer.Initialize method passing app.ApplicationServices as a parameter.

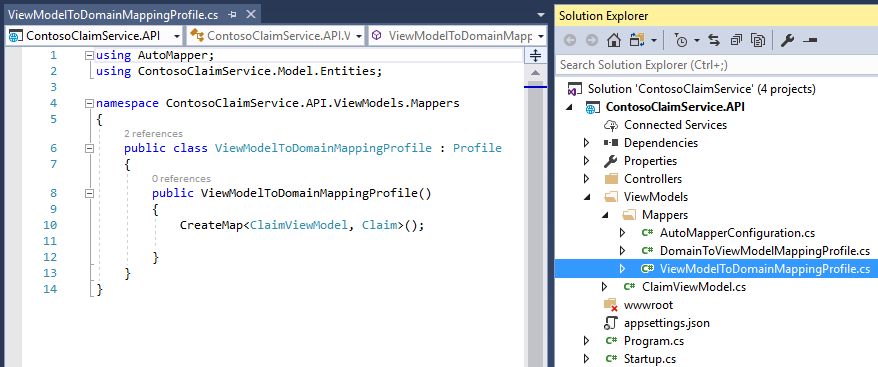


### Task 3: Create ViewModels and Mapping Profiles.

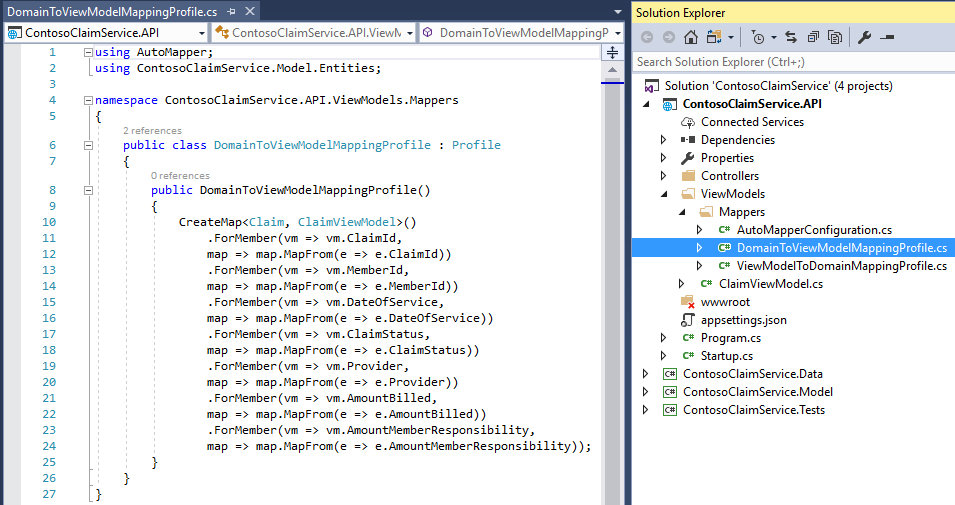
1. Create a new **ViewModels** folder. Create classes (or DTOs) for mapping domain entities to the API object model.
   1. ClaimViewModel



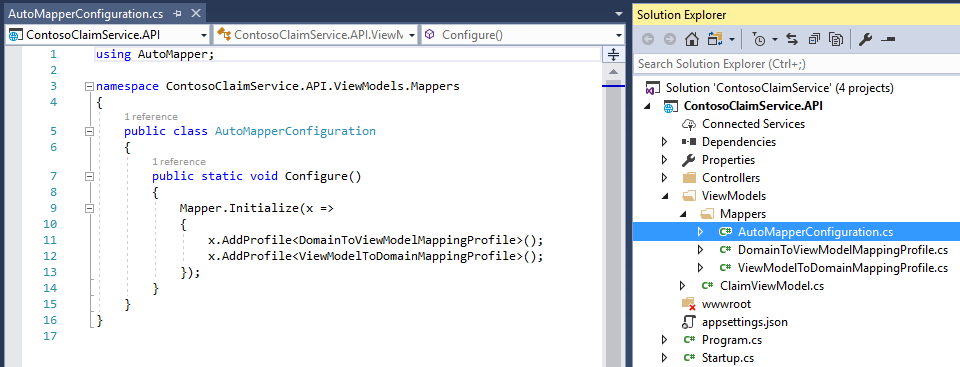
1. Create Automapper mapping profile classes within a **Mappers** folder inside the **ViewModel** folder.
   1. ViewModelToDomainMappingProfile



* 1. DomainToViewModelMappingProfile



* 1. Create the **AutoMapperConfiguration** class to initialize the mapping profiles inside a static **Configure** method.
  2. Add AutoMapperConfiguration.Configure(); to **ConfigureServices** method in the **Startup** class.

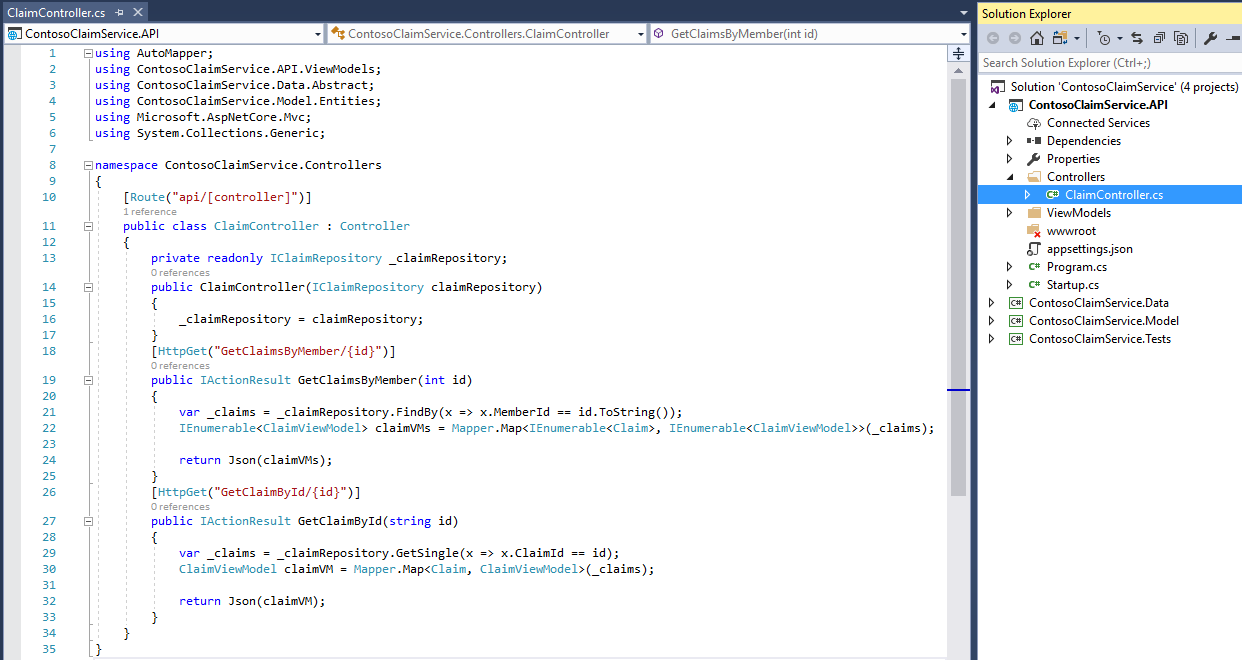


### Task 4: Create API Controller and Configure Routing.

1. Create a **ClaimController** class inside a new folder called **Controllers**.
   1. Set the route of the controller to “api/[controller]”
   2. Inject the ClaimRepository container service into the controller constructor.
   3. Create a controller action named GetClaimsByMember and set the HttpGet attribute to "GetClaimsByMember/{id}"**.**
   4. Within the GetClaimsByMember method call the claim repository to return all claims associated to the incoming id parameter.
   5. Reference the AutoMapper map method to convert the entity to a viewmodel.

IEnumerable<ClaimViewModel> claimVMs = Mapper.Map<IEnumerable<Claim>, IEnumerable<ClaimViewModel>>(\_claims);

* 1. Return the resulting claims in Json format.
  2. Create a controller action named GetClaimById, set the HttpGet attribute to "GetClaimById/{id}" and reference the claim repository to return a single claim based on the incoming id.

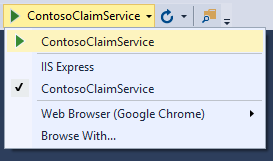


### Task 5: Run and Verify Solution

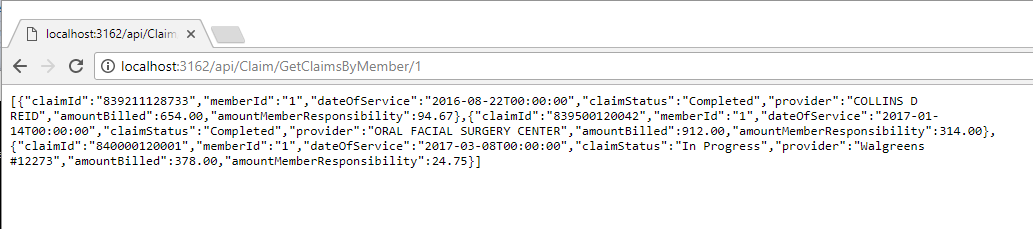
**Kestrel** is a cross-platform web server for ASP.NET Core based on libuv, a cross-platform asynchronous I/O library. Kestrel is the web server that is included by default in ASP.NET Core new project templates.

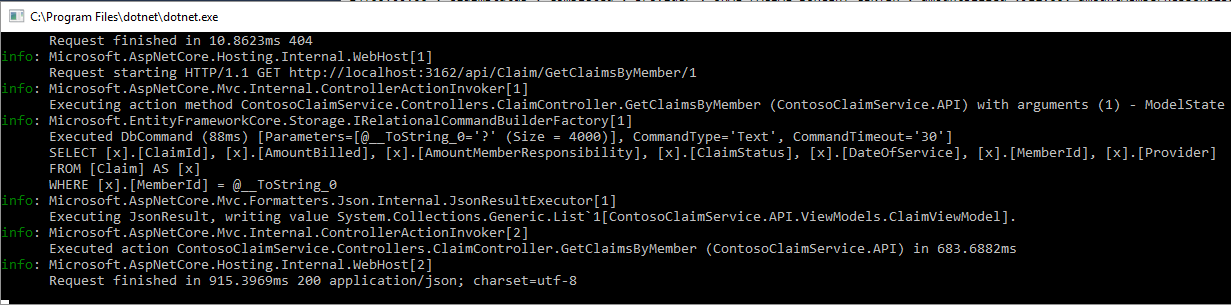
<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel>

1. Set the **ContosoClaimService.Api** as Startup project and run the application from a kestrel web server instance by ensuring IIS Express is not selected in the debug configuration and by pressing **Ctrl+F5** or clicking the “play” button seen below:



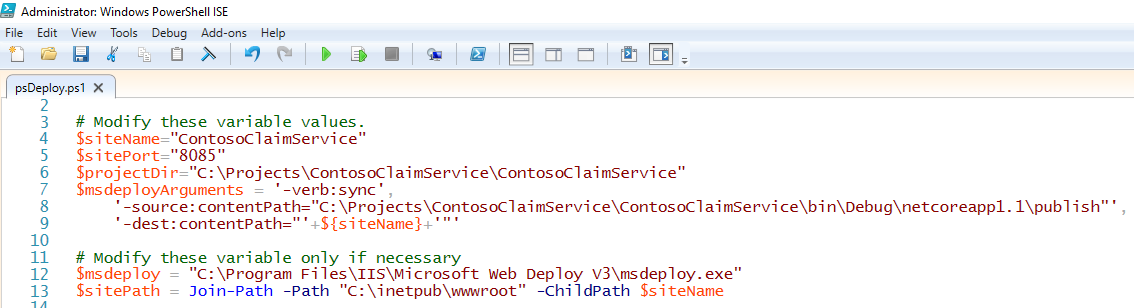
1. Navigate to **/api/Claim/GetClaimsByMember/1** to validate database items are returned from the service. Note: In the command prompt launch by Visual Studio, you can view corresponding log information from the Kestrel server.





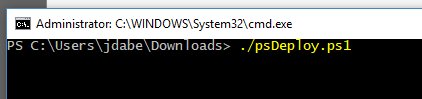
### Task 6: Publish the ContosoClaimService to Local IIS.

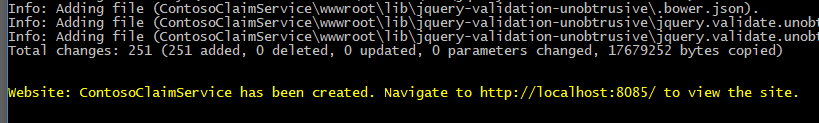
1. Open the psDeploy.ps1 script included in your hackathon files. Inside the script modify the PowerShell values with the following:
   1. $siteName: **ContosoClaimService**
   2. $sitePort: **8085**
   3. $projectUrl: Project Directory.
   4. $msdeploymentArguments: Update with output directory for published binaries.



**Note**: Ensure you have [msdeploy v3](https://www.microsoft.com/en-us/download/details.aspx?id=43717) installed on your computer.

1. Once the psDeploy.ps1 script has been configured, run command prompt as Administrator, navigate to the directory with the script and execute with the following command: **./psDeploy.ps1**





1. Navigate to the site url and verify application is hosted and running.

**Twelve-Factor App: V. Build, release, run**

A [codebase](https://12factor.net/codebase) is transformed into a (non-development) deploy through three stages: Build, release and run. **The twelve-factor app uses strict separation between stages. Microsoft’s build and release management facilities such as MSBuild, WebDeploy, etc implement this principle**

<https://12factor.net/build-release-run>

## 

## **Create the Contoso Member Service Solution**

**Introducing the .NET Core CLI**

The .NET Core command-line interface (CLI) is a new cross-platform toolchain for developing .NET applications. The CLI is a foundation upon which higher-level tools, such as Integrated Development Environments (IDEs), editors, and build orchestrators, can rest.

dotnet is a generic driver for the Command Line Interface (CLI) toolchain. Invoked on its own, it provides brief usage instructions. Each specific feature is implemented as a command. In order to use the feature, the command is specified after dotnet, such as dotnet build. All the arguments following the command are its own arguments.

## Exercise 1: Create the Solution using the dotnet CLI.

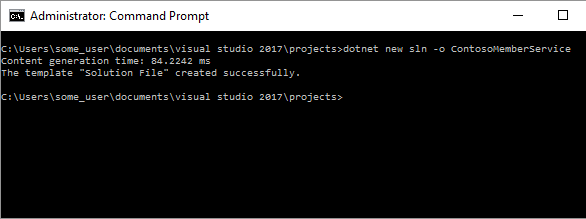
Duration: 10 Minutes

### Tasks to complete

* Create the solution using dotnet sln command
* Create the API and Data projects using the dotnet new command.
* Add the projects to the solution using cli commands.
* Restore packages, build and run the solution using cli commands.

### Task 1: Create the solution using dotnet sln command.

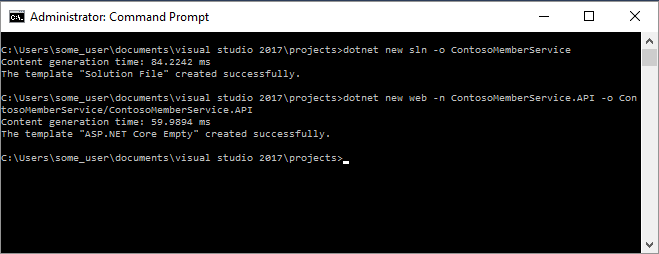
1. Open command prompt and navigate to your desired solution location.
2. Create the solution with the following command:

**dotnet new sln -o ContosoMemberService**

### Task 2: Create the API and Data projects using the dotnet new command

1. Create the API web project with the following command:

**dotnet new web -n ContosoMemberService.Api -o ContosoMemberService.Api**



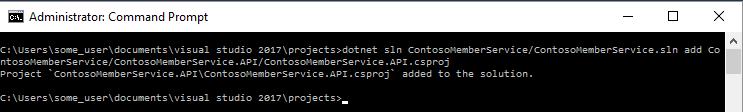
1. Repeat for Data project class library

**dotnet new classlib -n ContosoMemberService.Data -o ContosoMemberService.Data**

### Task 3: Add the projects to the solution using cli commands

1. Associate the API project to the solution with the following command

**dotnet sln ContosoMemberService.sln add ContosoMemberService.Api/ContosoMemberService.Api.csproj**

****

1. Repeat for Data project.

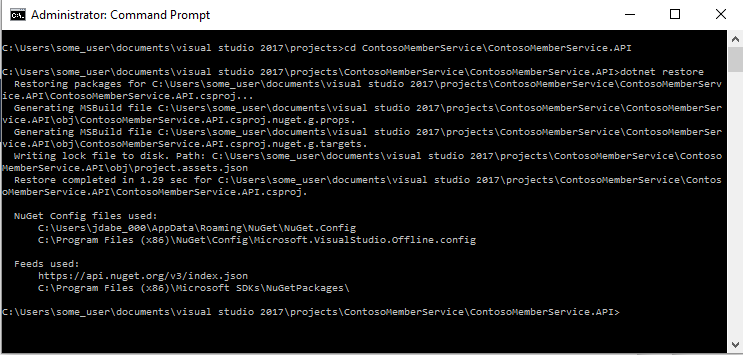
**dotnet sln ContosoMemberService.sln add ContosoMemberSerivce.Data/ContosoMemberService.Data.csproj**

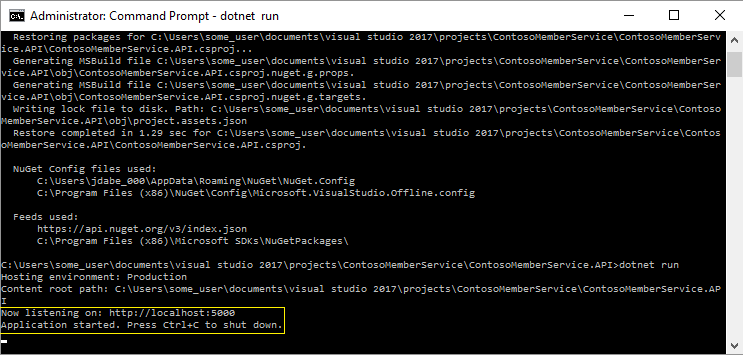
### Task 4: Restore packages, build and run the solution using cli commands

1. Navigate to the API project folder and execute the following commands:

**dotnet restore**

**dotnet run**

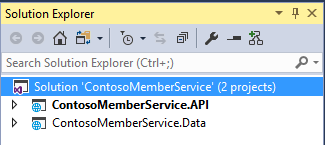
****

****

1. Navigate to localhost:5000/ and verify “Hello World” response.



1. Open the solution in Visual Studio and verify related projects are attached.



## Exercise 2: Complete the ContosoMemberService.Data Project.

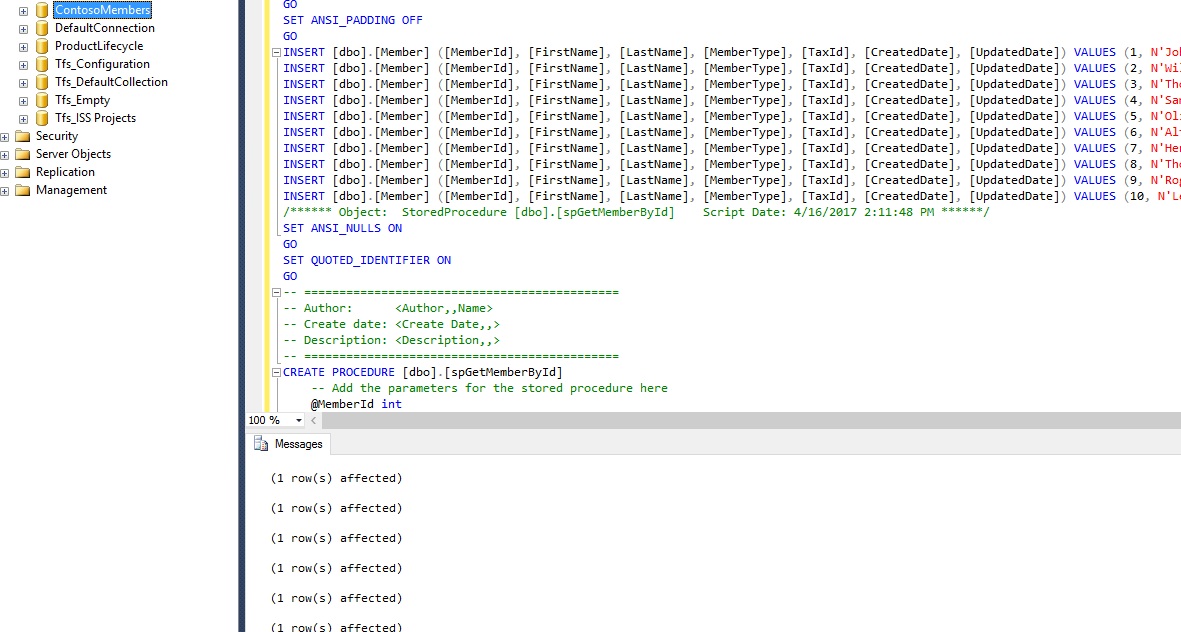
Duration: 20 Minutes

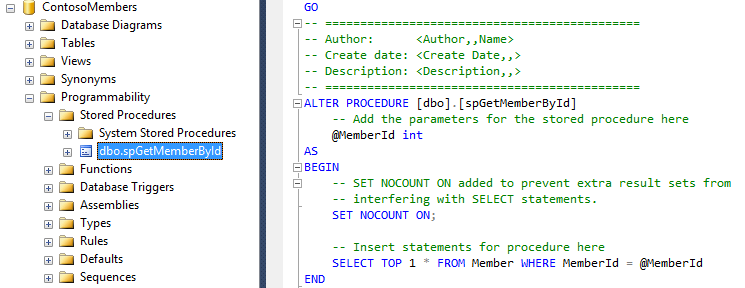
### Task 1: Configure Project Dependencies

1. Install the **Microsoft.EntityFrameworkCore** (version 1.1.2) NuGet package
2. Install the **Microsoft.EntityFrameworkCore.SqlServer** (version 1.1.2) NuGet package

### Task 2: Create Members Database from Script

1. Execute **ContosoMembers.sql** script against your SQL Server database instance.
2. Verify the **ContosoMembers** Database has been created.
3. Verify the **Members** table has been created with data.
4. Verify **spGetMemberById** stored procedure has been created.





### Task 3: Create A Model Class for the Member Object

1. Create a folder in the project named Model.
2. Create the following class and fields inside the Model folder.
3. Member

long MemberId

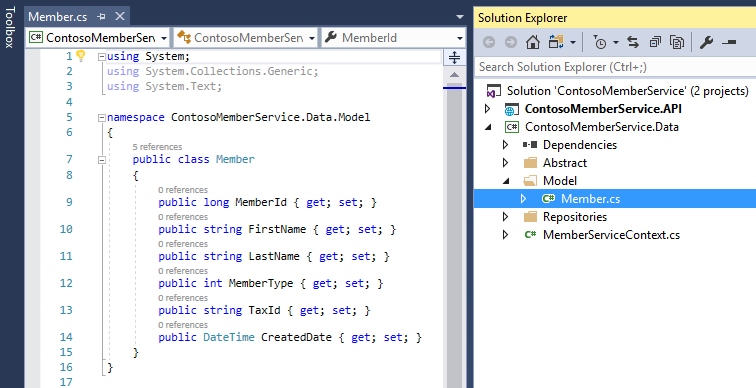
string FirstName

string LastName

int MemberType

string TaxId

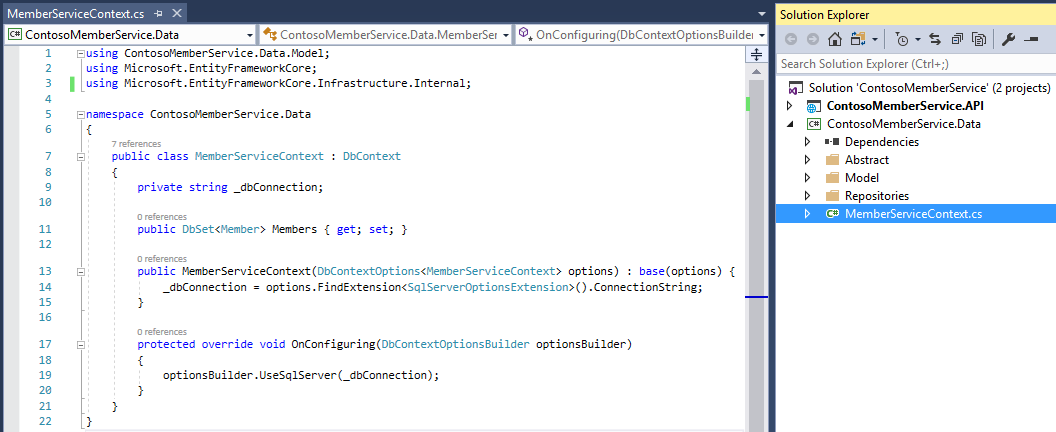
DateTime CreatedDate



### Task 4: Create A DBContext Object for The Project

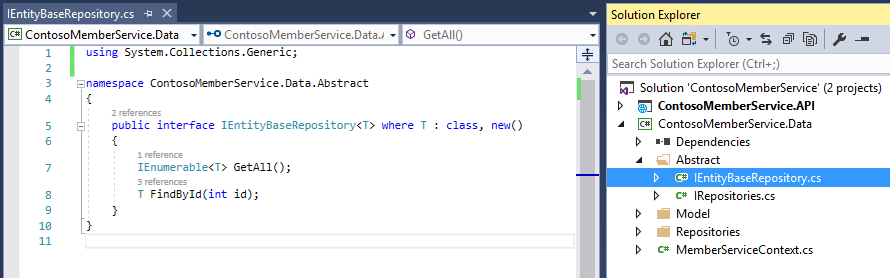
1. Create a **MemberServiceContext** class which inherits from DbContext
   1. Create a private class level property of type string called **\_dbConnection**.
   2. Implement a single DbSet using the Member class as TEntity.
   3. Implement a constructor with DbContextOptions as a parameter and inherit from the base object.
   4. Within the MemberServiceContext constructor, set \_dbConnection equal to the incoming SQL connection string using the **FindExtension<>** method on the DbContextOptions object**.**
   5. Override **OnConfiguring** method to configure the context to connect to Microsoft SQL Server database defined by **\_dbConnection.**

optionsBuilder.UseSqlServer(\_dbConnection);



### Task 5: Implement A Repository Pattern

1. Implement the repository pattern by creating an IEntityBaseRepository interface in a new folder called **Abstract**.
   1. Add operation for IEnumerable<T> GetAll()
   2. Add operation for T FindById(int id)



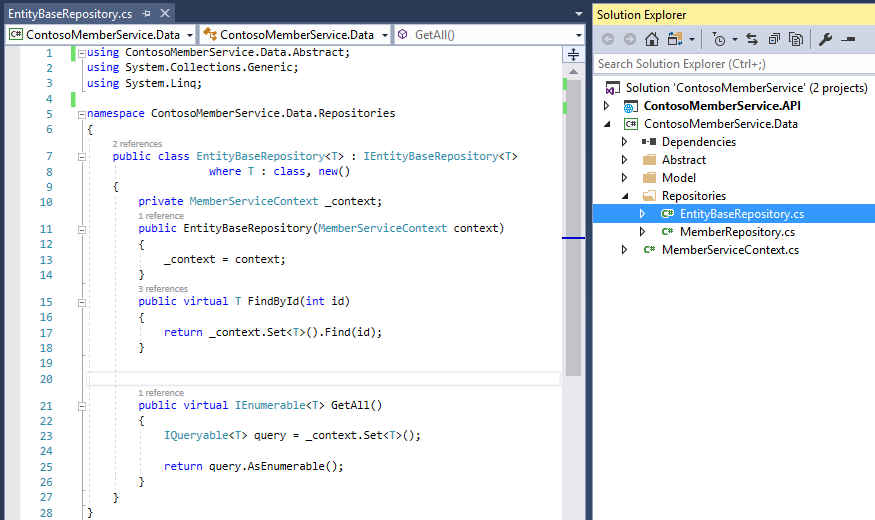
1. Implement the IEntityBaseRepository interfaceby creating the EntityBaseRepository class in a new folder called **Repositories**

**Introduction to Dependency Injection in ASP.NET Core**

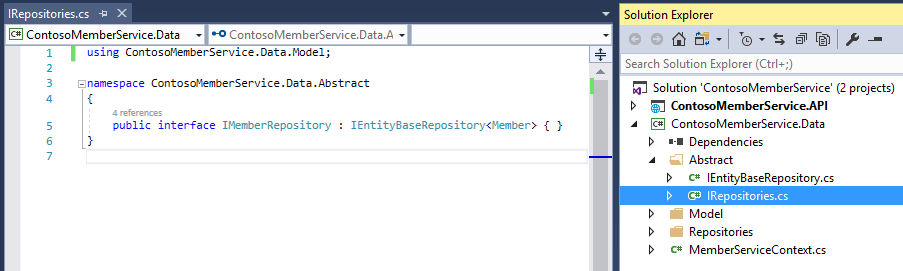
ASP.NET Core is designed from the ground up to support and leverage dependency injection. ASP.NET Core applications can leverage built-in framework services by having them injected into methods in the Startup class, and application services can be configured for injection as well. The default services container provided by ASP.NET Core provides a minimal feature set and is not intended to replace other containers.

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection>

* 1. Create a private **MemberServiceContext** field, and Inject the **MemberServiceContext** into the EntityBaseRepository constructor, setting the private field equal to the injected parameter.
  2. Implement the GetAll() method returning all Members from the context.
  3. Implement the T FindById(int id) method using the context to return a single result by id.

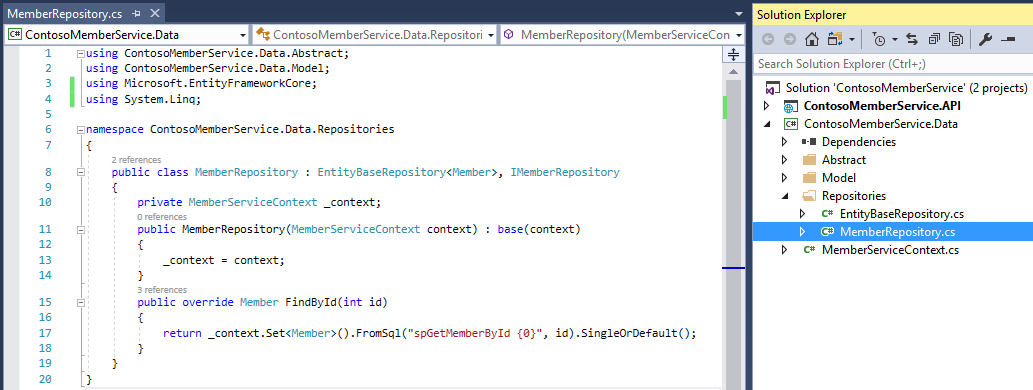


1. Create a **IRepositories** class in the **Abstract** folder
   1. Within the class, create an interface called IMemberRepository inheriting from IEntityBaseRepository, where T is Member



1. Implement theIMemberRepository interfaceby creating the MemberRepository class in the **Repositories** folder.
   1. In addition to implementing the IMemberRepository interface, inherit from the EntityBaseRepository class, where T is Member
   2. Inject the **MemberServiceContext** into the MemberRepository constructor
   3. Override the FindById(int id),replacing T with Member method and use the context to invoke the spGetMemberById Stored procedure.

return \_context.Set<Member>().FromSql("spGetMemberById {0}", id).SingleOrDefault();



## Exercise 3: Complete the ContosoMemberService.Api Project.

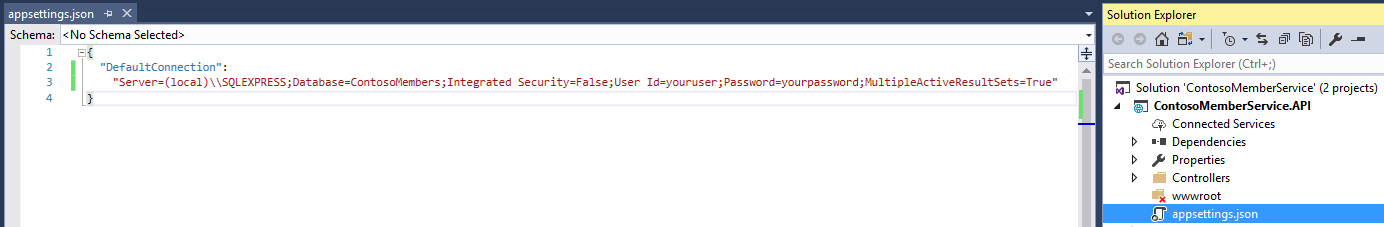
Duration: 20 Minutes

### Tasks to complete

* Configure Project Dependencies
* Configure Services and Middlewares in the .NET Core Startup Class
* Create API Controller and Configure Routing
* Run and Verify Solutions

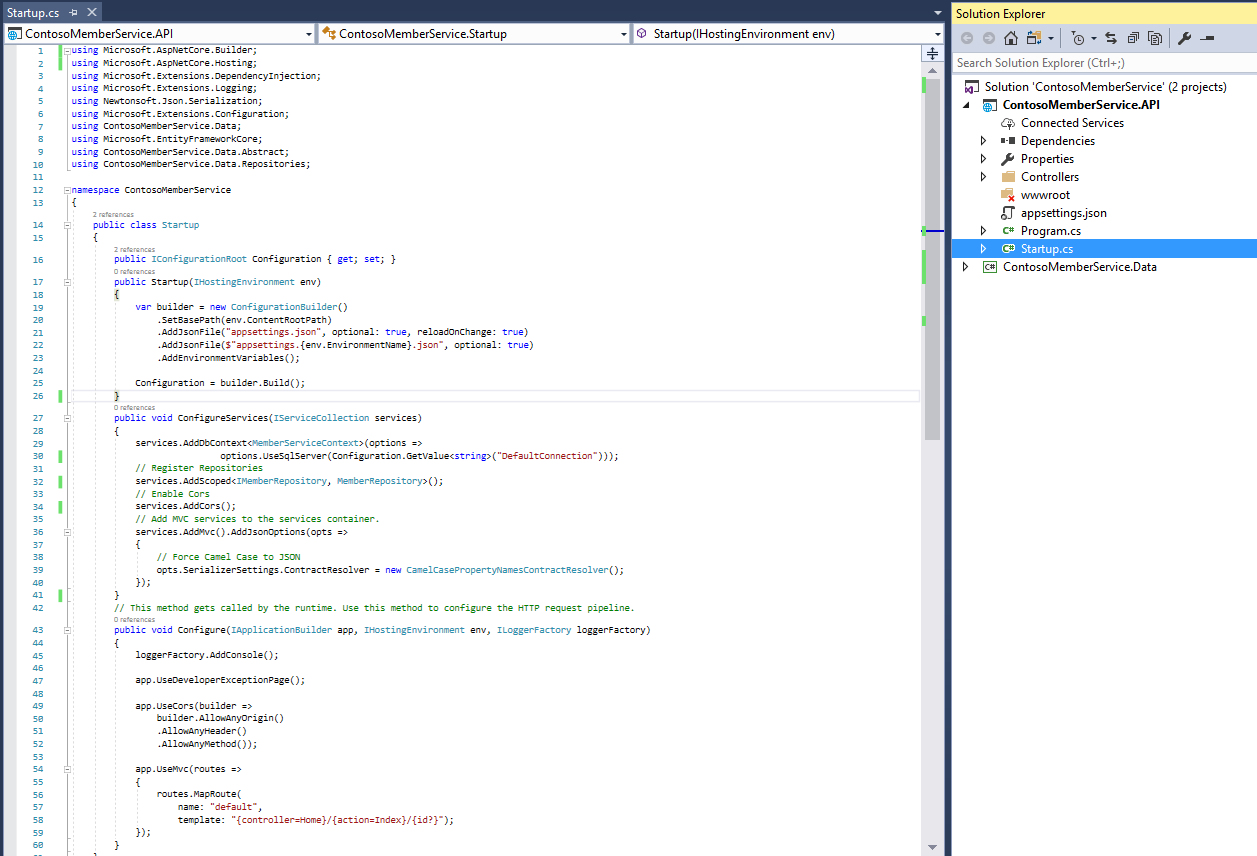
### Task 1: Configure Project Dependencies

1. Install the **Microsoft.EntityFrameworkCore** (version 1.1.2) NuGet package
2. Install the **Microsoft.EntityFrameworkCore.SqlServer** (version 1.1.2) NuGet package
3. Install the **Microsoft.AspNetCore.Mvc** (version 1.1.3) NuGet package.
4. Add a Project reference to the **ContosoMemberService.Data** project
5. Create an **appsettings.json** file in the project and specify your local database connectionstring: "DefaultConnection": "Data Source=(localdb)\\MSSQLLocalDB; Initial Catalog=ContosoMembers; Integrated Security=True;"



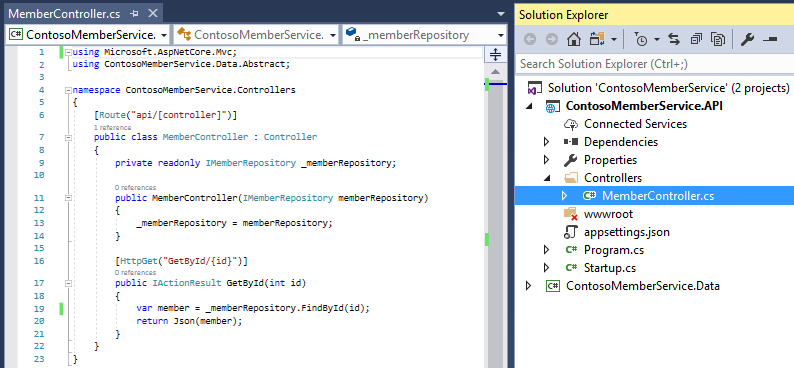
### Task 2: Configure Services and Middlewares in the .NET Core Startup Class

1. Configure the **Startup** class:
   1. Create a Startup class constructor with a parameter of IHostingEnvironment
   2. Create a class level IConfigurationRoot property call **Configuration.**
   3. Within the **Startup** constructor, implement a ConfigurationBuilder object which references the **appsettings.json** configuration file.
   4. Build the configuration and set it the **Configuration** class level property.
2. Register services to the .NET Core container in the **ConfigureServices** method of the **Startup** class:
   1. Register and add a DbContext of type MemberServiceContext using SqlServer options and configuration value specified in the DefaultConnection property.
   2. Register the MemberRepository service with a scoped lifetime.
   3. Register CORS support.
   4. Register the MVC service and configure camel case Json serialization.
3. Configure application middlewares in the **Configure** method of the Startup class.
   1. Configure CORS to allow any origin, header and method.
   2. Configure MVC routing (use default maproute)



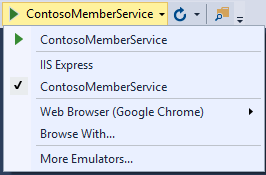
### Task 3: Create API Controller and Configure Routing.

1. Create a **MemberController** class inside a new folder called **Controllers**.
   1. Set the route of the controller to "api/[controller]"
   2. Inject the MemberRepository container service into the controller constructor.
   3. Create a controller action named **GetById** and set the HttpGet attribute route to "GetById/{id}"
   4. Within the **GetById** method call the member repository to return a member based on the incoming id parameter.
   5. Return the resulting member in Json format.

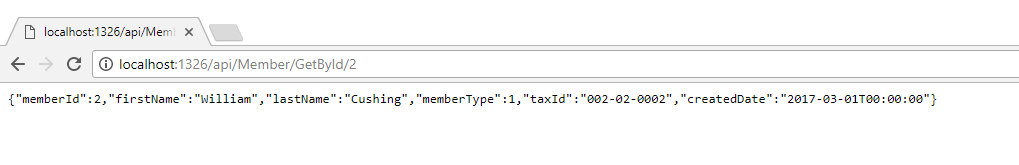


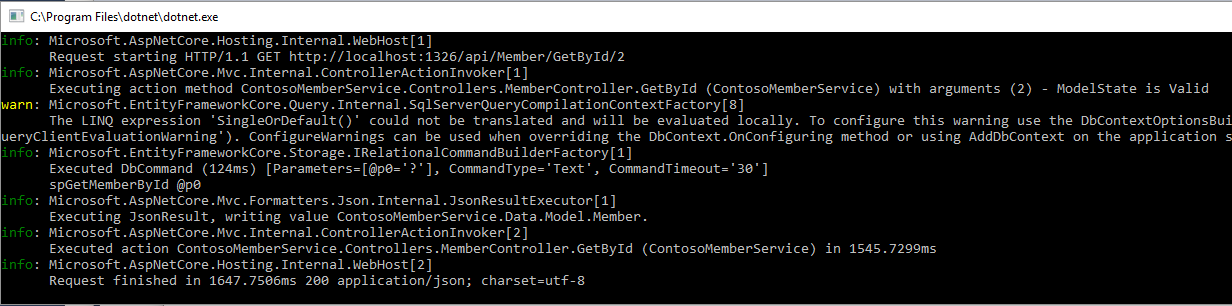
### Task 4: Run and Verify Solution

1. Set the **ContosoMemberService.Api** as Startup project and run the application from a kestrel web server instance by ensuring IIS Express is not selected in the debug configuration and by pressing **Ctrl+F5** or clicking the “play” button seen below:



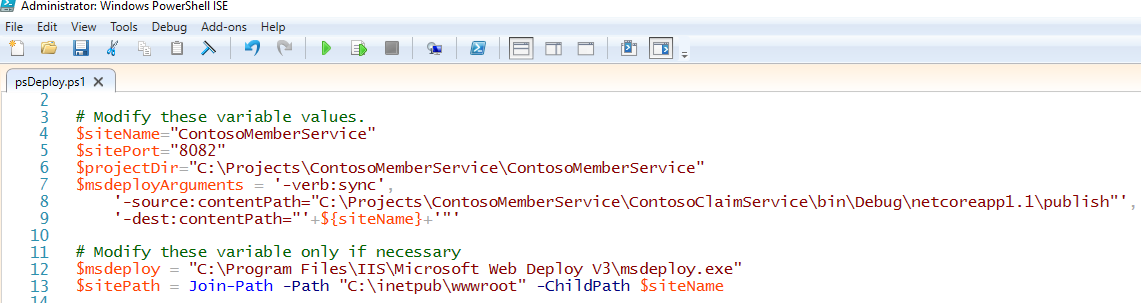
1. Navigate to **/api/Member/GetById/2** to validate database item are returned from the service and database stored procedure. Note: In the command prompt launch by Visual Studio, you can view corresponding log information from the Kestrel server.





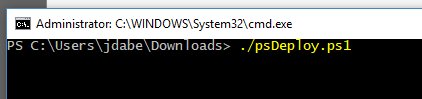
### Task 5: Publish the ContosoMemberService to Local IIS.

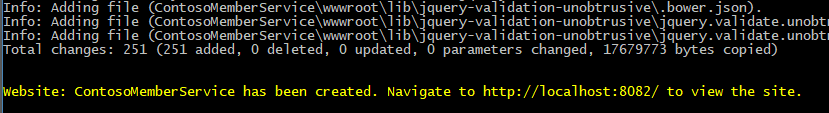
1. Open the psDeploy.ps1 script included in your hackathon files. Inside the script modify the PowerShell values with the following:
   1. $siteName: **ContosoMemberService**
   2. $sitePort: **8082**
   3. $projectUrl: Project Directory.
   4. $msdeploymentArguments: Update with output directory for published binaries.



**Note**: Ensure you have [msdeploy v3](https://www.microsoft.com/en-us/download/details.aspx?id=43717) installed on your computer.

1. Once the psDeploy.ps1 script has been configured, run command prompt as Administrator, navigate to the directory with the script and execute with the following command: **./psDeploy.ps1**





1. Navigate to the site url and verify application is hosted and running.

## **Create the Contoso Member Dashboard Solution**

## Exercise 1: Create and configure the Dashboard Solution

Duration: 30 Minutes

### Tasks to complete

* Create the solution using VS or CLI and Configure Project Dependencies
* Create services and register with the asp.net core services container.
* Create views for dashboard and claim detail.
* Implement cookie middleware authentication.
* Publish and verify solution and provider services.

### Task 1: Configure Project Dependencies

1. Create the **ContosoDashboard** solution and **ContosoDashboard.Web** project using either the VS 2017 IDE or the .net core command line (CLI).
2. Install the **Microsoft.AspNetCore.Mvc** (version 1.1.3) NuGet package.
   1. Register the MVC service and configure default routes inside the Startup class.
3. Install the **Microsoft.AspNetCore.StaticFiles** (version 1.1.2) NuGet package
   1. Configure UseStaticFiles in the Configure method of the Startup class.
4. Install the **Microsoft.AspNetCore.Authentication.Cookies** (version 1.1.2) NuGet package
5. Install the **Newtonsoft.Json** (version 10.0.3) NuGet package
6. Create an **appsettings.json** file in the project with properties for both the claims and member services. Replace “XXXX” in baseUrl with the appropriate port number for each service.

{

"ClaimService": {

"username": "",

"password": "",

"baseUrl": "http://localhost:XXXX",

"getClaimByIdEndpoint": "/api/Claim/GetClaimById/",

"getClaimsByMemberEndpoint": "/api/Claim/GetClaimsByMember/"

},

"MemberService": {

"username": "",

"password": "",

"baseUrl": "http://localhost:XXXX",

"getMemberByIdEndpoint": "/api/Member/GetById/"

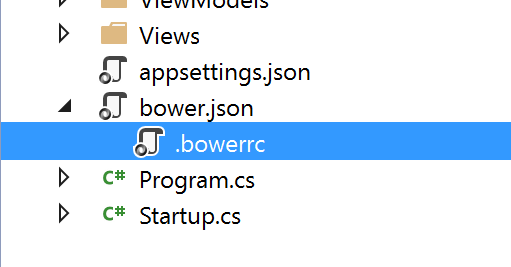
}

}

1. Configure the Startup class:
   1. Create a Startup class constructor with a parameter of IHostingEnvironment
   2. Create a class level IConfigurationRoot property call **Configuration.**
   3. Within the **Startup** constructor, implement a ConfigurationBuilder object which references the **appsettings.json** configuration file.
   4. Build the configuration and set it the **Configuration** class level property.
   5. Register the **Configuration** object with the service container to use throughout the project:

services.AddSingleton<IConfiguration>(Configuration);

1. Install bootstrap-css v3.3.7 using the Bower Package Manager.
   1. Right-click on the solution and choose Add->New…
   2. Select Visual C# -> ASP.NET Core -> Web -> General and choose **Bower Configuration file.**
   3. Verify a **bower.json** file has been added to the project.
   4. If your corporate proxy blocks bower, locate the **.bowerrc** file, underneath the **bower.json** in your project.



* 1. Modify the contents of the **.bowerrc** file with the following configuration and save:

{

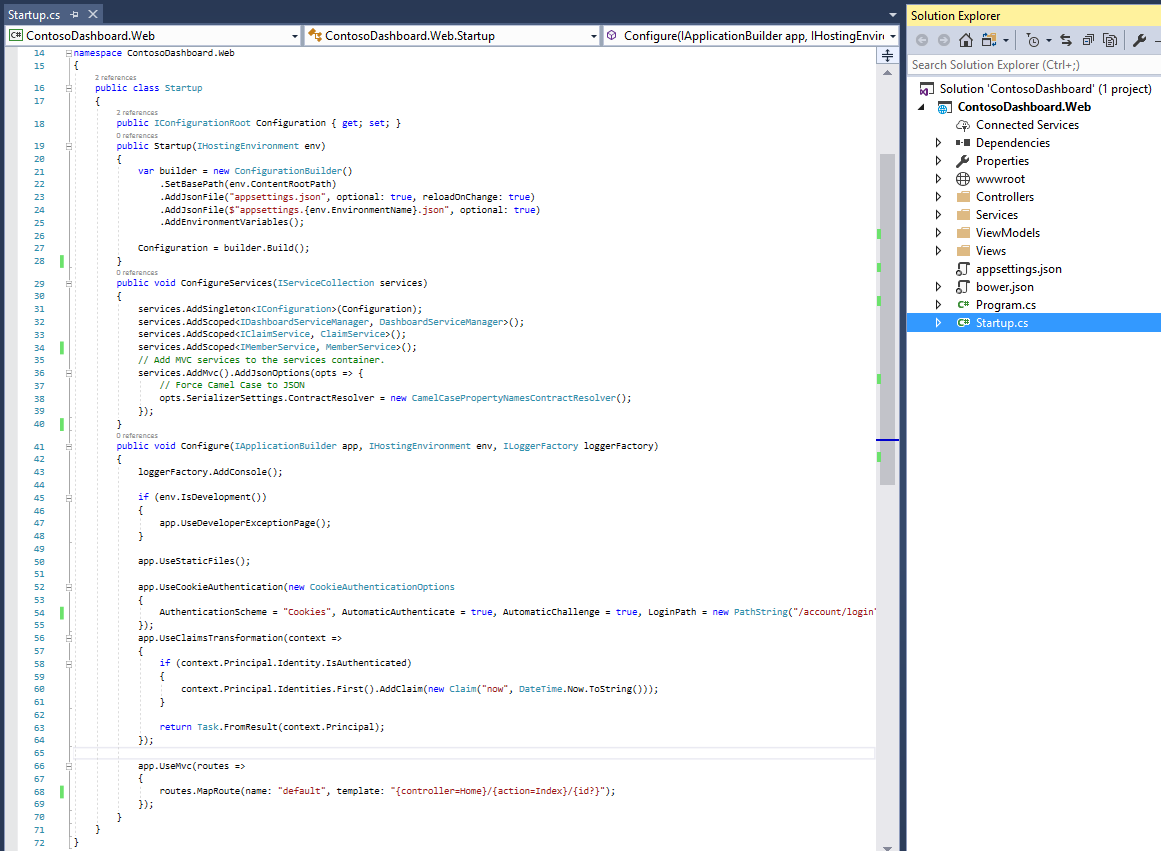
"directory": "wwwroot/lib",

"proxy": "http://gateway.zscaler.net:443",

"https-proxy": <http://gateway.zscaler.net:443>

}

* 1. Right-click on the **ContosoDashboard.Web** project and choose **Manage Bower Packages**. Note: If the option doesn’t exist, you may need to install it from NuGet with the following command in the package manager console: Install-Package Bower -Version 1.3.11 or from https://www.nuget.org/packages/Bower/
  2. Search for and download **bootstrap-css** (version 3.3.7)



### Task 2: Create ViewModels, services and register with the asp.net core services container.

**Twelve-Factor App: IV: Backing Services**

A backing service is any service the app consumes over the network as part of its normal operation. **The code for a twelve-factor app makes no distinction between local and third party services.** To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the [config](https://12factor.net/config).

[https://12factor.net/backing-services](https://12factor.net/build-release-run)

1. Create a ViewModels folder in the **ContosoDashboard.Web** project
   1. Create the ClaimViewModel:

string ClaimId

string MemberId

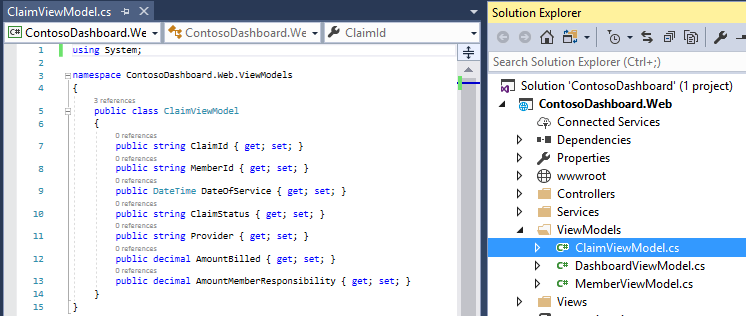
DateTime DateOfService

string ClaimStatus

string Provider

decimal AmountBilled

decimal AmountMemberResponsibility



* 1. Create the MemberViewModel:

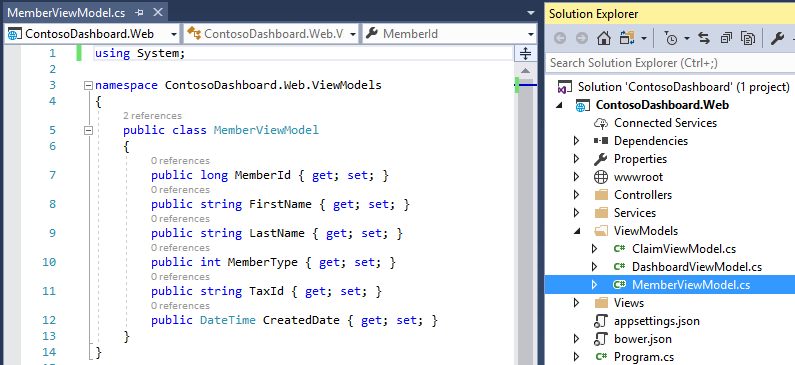
long MemberId

string FirstName

string LastName

int MemberType

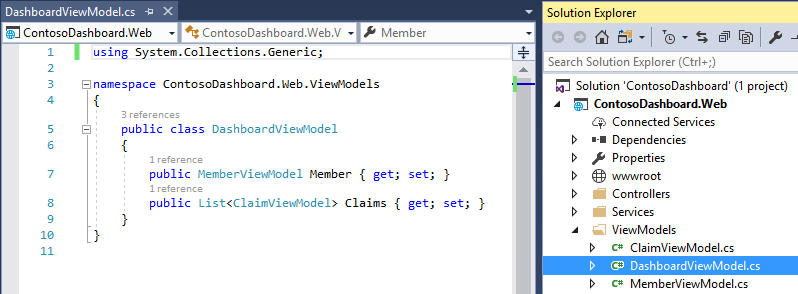
DateTime CreatedDate



* 1. Create the DashboardViewModel:

public MemberViewModel Member

public List<ClaimViewModel> Claims



1. Create a **Services** folder in the **ContosoDashboard.Web** project.

**Asynchronous Programming in .NET**

If you have any I/O-bound needs (such as requesting data from a network or accessing a database), you'll want to utilize asynchronous programming. You could also have CPU-bound code, such as performing an expensive calculation, which is also a good scenario for writing async code.

C# has a language-level asynchronous programming model which allows for easily writing asynchronous code without having to juggle callbacks or conform to a library which supports asynchrony. It follows what is known as the [Task-based Asynchronous Pattern (TAP)](https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap).

The core of async programming are the Task and Task<T> objects, which model asynchronous operations. They are supported by the async and await keywords. The model is simple in most cases:

For I/O-bound code, you await an operation which returns a Task or Task<T> inside of an async method.

For CPU-bound code, you await an operation which is started on a background thread with the Task.Run method.

<https://docs.microsoft.com/en-us/dotnet/csharp/async>

1. Create the ClaimService:
   1. Within the Services folder, create an IClaimService interface and a ClaimService class which implements the following:

Task<JArray> GetClaimsByMemberId(int id);

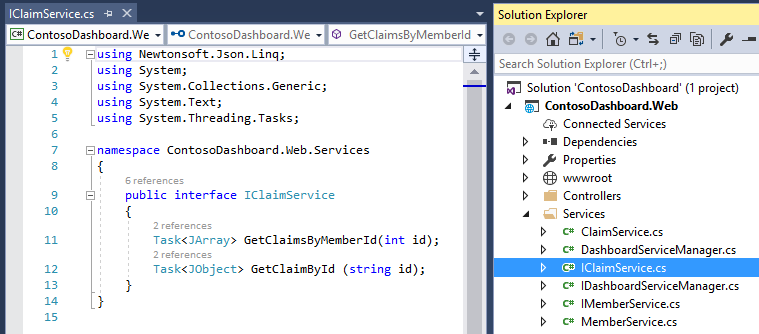
Task<JObject> GetClaimById (string id);

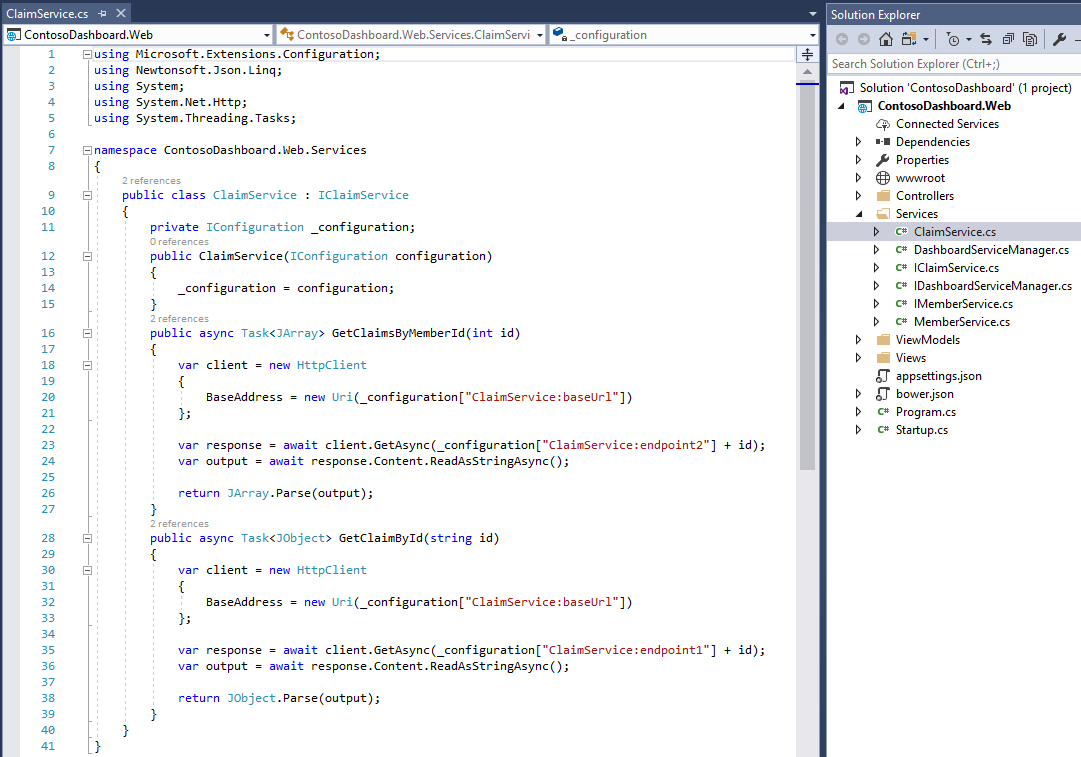
* 1. Inject the IConfiguration service into the ClaimService constructor and use the configuration object to access baseUrl and endpoint properties from the **appsettings.json** configuration file.
  2. Implement an HttpClient leveraging the configuration object and incoming parameters to invoke the ContosoClaimService:

var response = await client.GetAsync(\_configuration["ClaimService:endpoint1"] + id);

var output = await response.Content.ReadAsStringAsync();

* 1. Parse the resulting await object into JArray return type.





1. Create the MemberService:
   1. Within the Services folder, create an IMemberService interface and a MemberService class which implements the following:

Task<JObject> GetMember(int id);

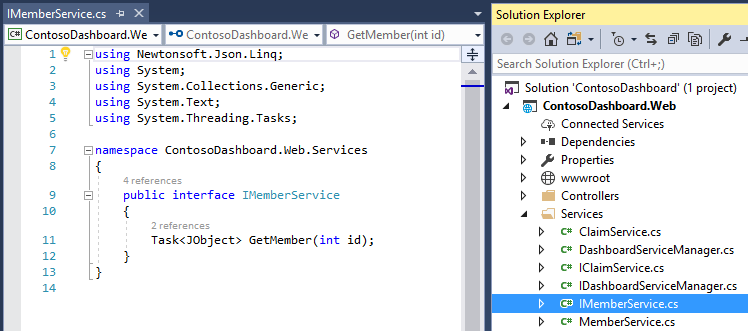
* 1. Inject the IConfiguration service into the MemberService constructor and use the configuration object to access baseUrl and endpoint properties from the **appsettings.json** configuration file.
  2. Implement an HttpClient leveraging the configuration object and incoming parameters to invoke the ContosoClaimService:

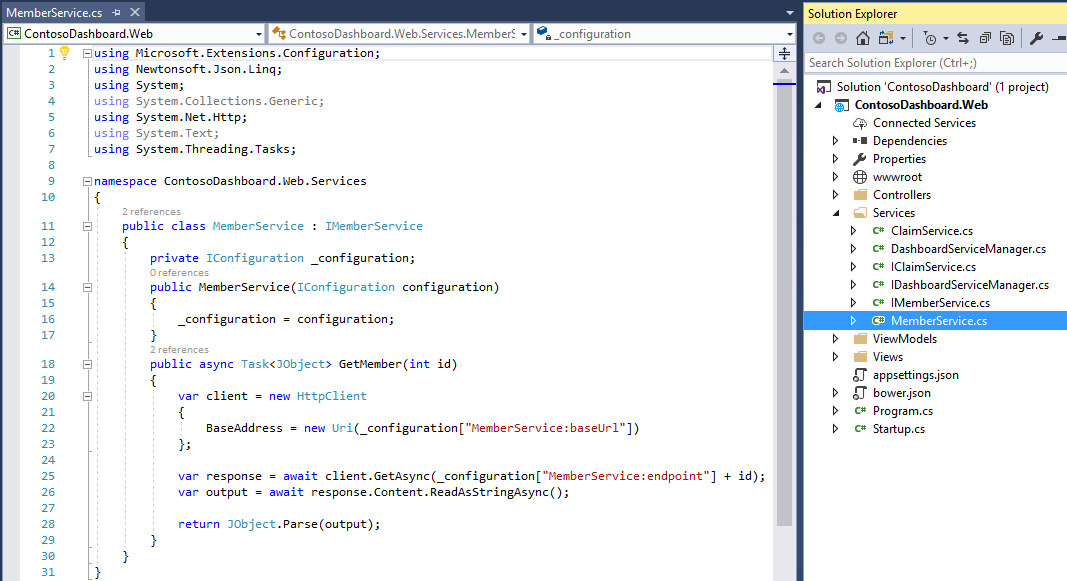
var response = await

client.GetAsync(\_configuration["MemberService:endpoint"] + id);

var output = await response.Content.ReadAsStringAsync();

* 1. Parse the resulting await object into JObject return type.





1. Create the DashboardServiceManager:
   1. Within the Services folder, create an IDashboardServiceManager interface and a DashboardServiceManager class which implements the following:

DashboardViewModel GetMemberData(int id);

* 1. Inject the IMemberService and IClaimService service into the DashboardServiceManager constructor and set equal to private members of the class.
  2. Implement the GetMemberData method calling GetMember and GetClaimsByMember operations to build the DashboardViewModel:

var memberObj = \_memberService.GetMember(id);

var claimsObj = \_claimService.GetClaimsByMemberId(id);

var dashboardVM = new DashboardViewModel()

{

Member = memberObj.Result.ToObject<MemberViewModel>(),

Claims = claimsObj.Result.ToObject<List<ClaimViewModel>>()

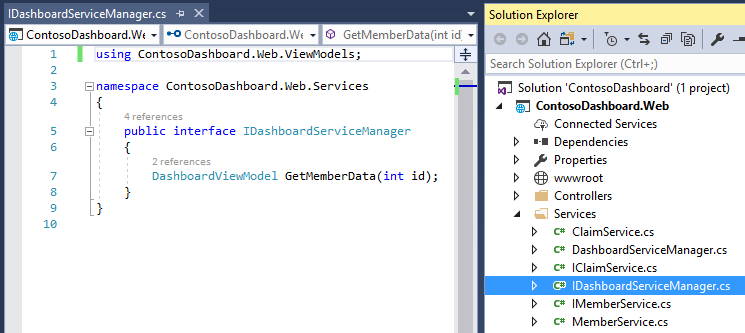
};

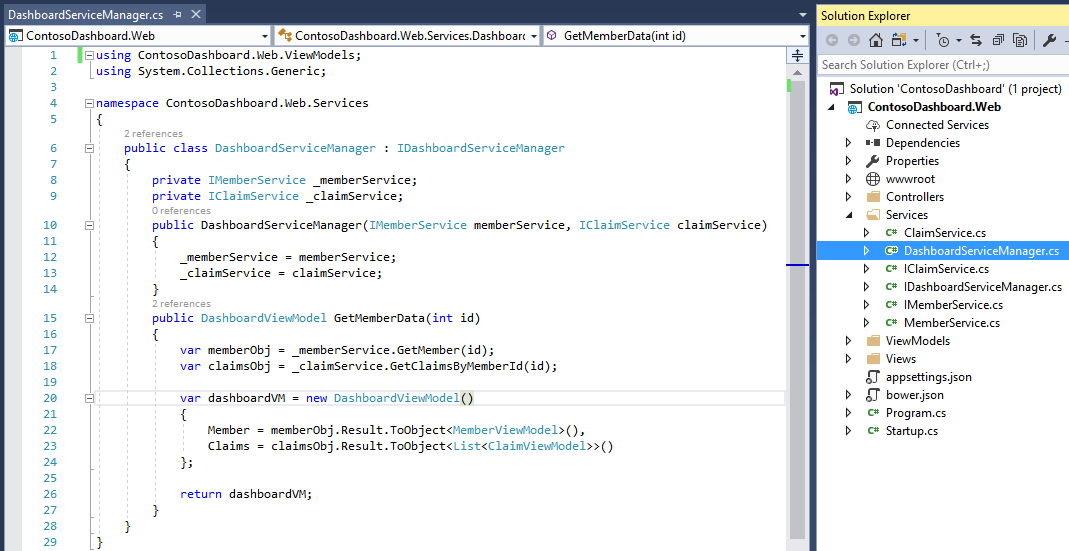
1. Register the services with the services container in the Startup class:

services.AddScoped<IDashboardServiceManager, DashboardServiceManager>();

services.AddScoped<IClaimService, ClaimService>();

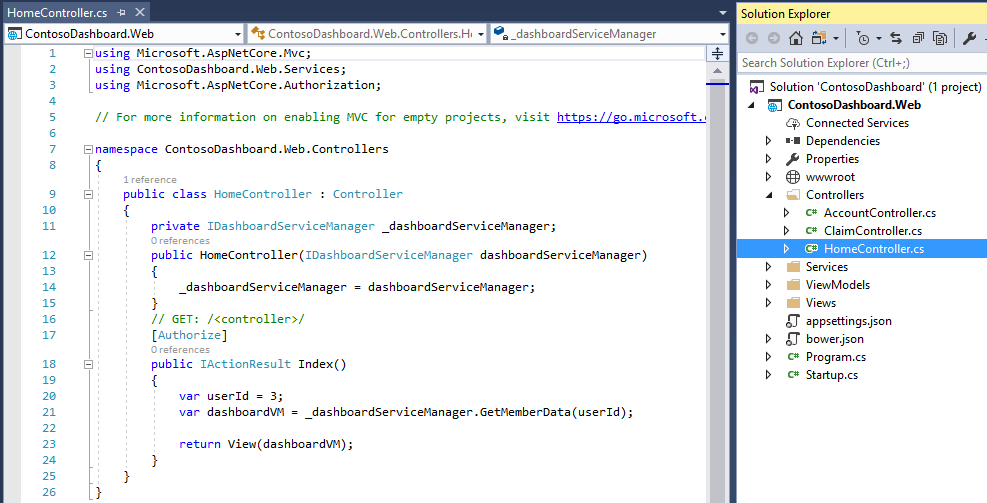
services.AddScoped<IMemberService, MemberService>();





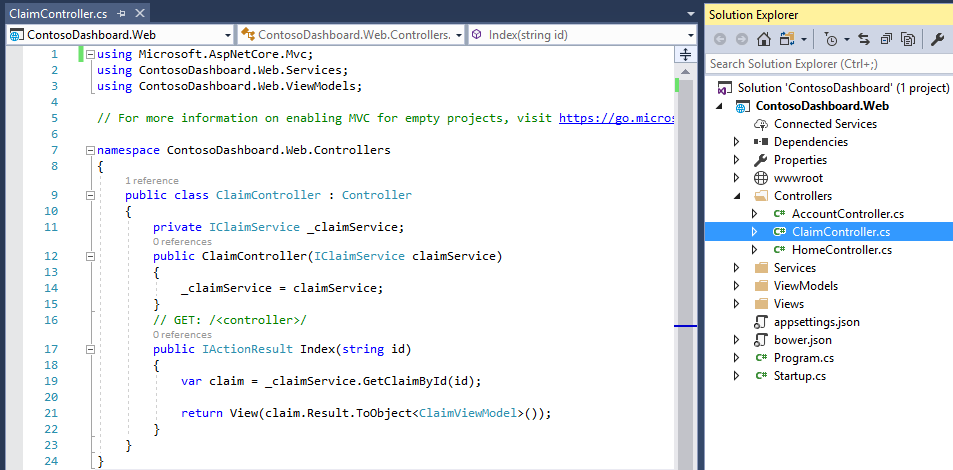
### Task 3: Create controllers and views for dashboard and claim detail

1. Create a **Controllers** folder in the **ContosoDashboard.Web** project
2. Create a HomeController:
   1. inject the IDashboardServiceManager service into the constructor.
   2. Within the Index action, invoke the GetMemberData operation from the service manager and return the DashboardViewModel to the view. Pass the GetMemberData operation a hardcoded member Id of **3**
   3. Annotate the Index action with [Authorize] to enforce authentication.



1. Create a ClaimController:
   1. Inject the IClaimService service into the constructor
   2. Add an id parameter of type string to the Index controller action.
   3. Invoke the GetClaimById operation passing the incoming parameter.
   4. Map the result to a ClaimViewModel object:

return View(claim.Result.ToObject<ClaimViewModel>());



1. Create a **Views** folder in the **ContosoDashboard.Web** project
2. Create a **Shared** folder in the **Views** folder and create the \_Layout.cshtml view:

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>@ViewBag.Title</title>

<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />

</head>

<body>

<nav class="navbar navbar-default">

<div class="container-fluid">

<div class="navbar-header">

<a class="navbar-brand" href="#">

Contoso Dashboard

</a>

</div>

</div>

</nav>

<div class="container-fluid">

<div class="row">

<div class="col-md-2 sidebar">

<ul class="nav nav-sidebar">

<li class="active"><a href="#">Manage My Account <span class="sr-only">(current)</span></a></li>

<li><a href="#">My Contoso Plan</a></li>

<li><a href="#">Stay Healthy</a></li>

</ul>

</div>

<div class="col-md-9 main">

@RenderBody()

</div>

</div>

</div>

</body>

</html>

1. Create a **Home** folder in the **Views** folder and create an **Index.cshtml** view:

@model ContosoDashboard.Web.ViewModels.DashboardViewModel

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h1 class="page-header">Welcome, @Model.Member.FirstName @Model.Member.LastName </h1>

<div class="col-md-6">

<div class="panel panel-primary" style="height:110px;">

<div class="panel-heading">

<h3 class="panel-title">Member Information</h3>

</div>

<div class="panel-body">

Member Since: @Model.Member.CreatedDate.ToString("MMMM dd, yyyy")<br />

SSN: @Model.Member.TaxId

</div>

</div>

</div>

<div class="col-md-6">

<div class="panel panel-success" style="height:110px;">

<div class="panel-heading">

<h3 class="panel-title">Find a Doctor</h3>

</div>

<div class="panel-body">

<div class="input-group">

<input type="text" class="form-control" placeholder="Search for a Healthcare provider">

<span class="input-group-btn">

<button class="btn btn-primary" type="button">Submit</button>

</span>

</div>

</div>

</div>

</div>

<div class="clearfix"></div>

<h2 class="sub-header">Claims</h2>

<div class="table-responsive">

<table class="table table-striped">

<thead>

<tr>

<th>Claim Id</th>

<th>Member Id</th>

<th>Date Of Service</th>

<th>Claim Status</th>

<th>Provider</th>

<th>Amount Billed</th>

<th>Your Responsibility</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model.Claims.OrderByDescending(x => x.DateOfService))

{

<tr>

<td>@item.ClaimId</td>

<td>@item.MemberId</td>

<td>@item.DateOfService.ToString("MMMM dd, yyyy")</td>

<td>@item.ClaimStatus</td>

<td>@item.Provider</td>

<td>$@String.Format("{0:0.00}", item.AmountBilled)</td>

<td>$@String.Format("{0:0.00}", item.AmountMemberResponsibility)</td>

<td><a class="btn btn-primary" href="/Claim/Index/@item.ClaimId">View</a></td>

</tr>

}

</tbody>

</table>

</div>

1. Create a **Claim** folder in the **Views** folder and create an **Index.cshtml** view:

@model ContosoDashboard.Web.ViewModels.ClaimViewModel

@\*

For more information on enabling MVC for empty projects, visit https://go.microsoft.com/fwlink/?LinkID=397860

\*@

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<div class="col-md-12">

<p>

<a class="btn btn-primary" href="/Home">Back</a>

</p>

<div class="clearfix"></div>

<div class="panel panel-success">

<div class="panel-heading">

<h3 class="panel-title">Claim Information</h3>

</div>

<div class="panel-body">

<div>Claim Id: @Model.ClaimId</div>

<div>Member Id: @Model.MemberId</div>

<div>Date Of Service: @Model.DateOfService.ToString("MMMM dd, yyyy")</div>

<div>Claim Status: @Model.ClaimStatus</div>

<div>Provider: @Model.Provider</div>

<div>Amount Billed: $@String.Format("{0:0.00}", Model.AmountBilled)</div>

<div>Amount Member Responsibility: $@String.Format("{0:0.00}", Model.AmountMemberResponsibility)</div>

</div>

</div>

</div>

### Task 4: Implement cookie middleware authentication

1. Create an **Account** folder in the **Views** folder and create an **Login.cshtml** view:

<h2>Login</h2>

<div class="row">

<div class="col-md-8">

<section>

<**form** **asp-controller**="Account" **asp-action**="Login" **asp-route-returnurl**="@ViewData["ReturnUrl"]" method="post" class="form-horizontal" role="form">

<div class="form-group">

<label class="col-md-2 control-label">UserName</label>

<div class="col-md-10">

<input type="text" name="username" />

</div>

</div>

<div class="form-group">

<label class="col-md-2 control-label">Password</label>

<div class="col-md-10">

<input type="password" name="password" />

</div>

</div>

<div class="form-group">

<div class="col-md-offset-2 col-md-10">

<button type="submit" class="btn btn-default">Log in</button>

</div>

</div>

</**form**>

</section>

</div>

</div>

1. Configure ASP.NET Core cookie authentication in the Startup.cs, **Configure** method

**Using Cookie Authentication without ASP.NET Core Identity**

ASP.NET Core provides cookie middleware which serializes a user principal into an encrypted cookie and then, on subsequent requests, validates the cookie, recreates the principal, and assigns it to the HttpContext.User property. If you want to provide your own login screens and user databases, you can use the cookie middleware as a standalone feature.

<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/cookie?tabs=aspnetcore1x>

app.UseCookieAuthentication(new CookieAuthenticationOptions

{

AuthenticationScheme = "Cookies",

AutomaticAuthenticate = true,

AutomaticChallenge = true,

LoginPath = new PathString("/account/login")

});

app.UseClaimsTransformation(context =>

{

if (context.Principal.Identity.IsAuthenticated)

{

context.Principal.Identities.First().AddClaim(new Claim("now", DateTime.Now.ToString()));

}

return Task.FromResult(context.Principal);

});

1. Create a AccountController:
   1. Implement HttpGet and HttpPost for the Login controller action:

[HttpGet]

public IActionResult Login(string returnUrl = null)

{

ViewData["ReturnUrl"] = returnUrl;

return View();

}

[HttpPost]

public async Task<IActionResult> Login(string userName, string password, string returnUrl = null)

{

ViewData["ReturnUrl"] = returnUrl;

if (!string.IsNullOrWhiteSpace(userName) &&

userName == password)

{

var claims = new List<Claim>

{

new Claim(ClaimTypes.PostalCode, "37174"),

new Claim(ClaimTypes.Sid, "99999"),

new Claim("customType", "AspNetCore")

};

var id = new ClaimsIdentity(claims, "user");

var p = new ClaimsPrincipal(id);

await HttpContext.Authentication.SignInAsync("Cookies", p);

return LocalRedirect(returnUrl);

}

return View();

}

public async Task<IActionResult> Logout()

{

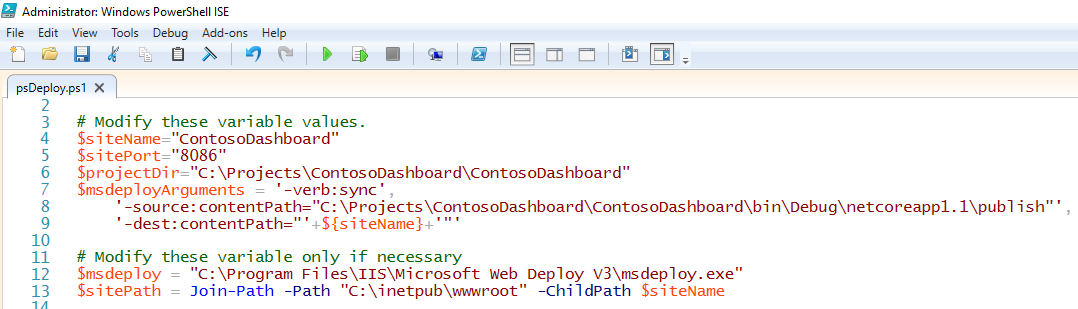
await HttpContext.Authentication.SignOutAsync("Cookies");

return Redirect("/");

}

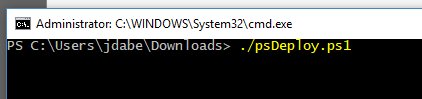
### Task 5: Publish the ContosoMemberService to Local IIS.

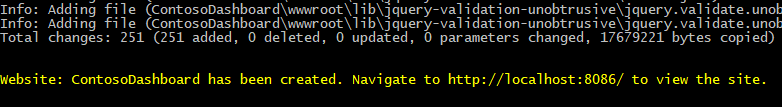
1. Open the psDeploy.ps1 script included in your hackathon files. Inside the script modify the PowerShell values with the following:
   1. $siteName: **ContosoDashboard**
   2. $sitePort: **8086**
   3. $projectUrl: Project Directory.
   4. $msdeploymentArguments: Update with output directory for published binaries.



**Note**: Ensure you have [msdeploy v3](https://www.microsoft.com/en-us/download/details.aspx?id=43717) installed on your computer.

1. Once the psDeploy.ps1 script has been configured, run command prompt as Administrator, navigate to the directory with the script and execute with the following command: **./psDeploy.ps1**





1. Navigate to the site url and verify application is hosted and running.
2. Login to the application using **user**/**user** as credentials.

