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 4](#_Toc477409716)

[Overview 4](#_Toc477409717)

[Hackathon Prerequisites 4](#_Toc477409718)

[Install Visual Studio 2017 and .NET Core SDK 4](#_Toc477409719)

[Create the Contoso Claim Service Solution 5](#_Toc477409720)

[Complete the ContosoClaimService.Model project 5](#_Toc477409721)

[Complete the ContosoClaimService.Data project 6](#_Toc477409722)

[Complete the ContosoClaimService.API project 7](#_Toc477409723)

[OLD CONTENT - Create the Contoso Patient Portal Project 13](#_Toc477409724)

# 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)
* Optional: Create a [Visual Studio Team Services](https://www.visualstudio.com/team-services/) project to facilitate revision control and deployment.

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

1. Create an ASP.NET Core Web Application named **ContosoClaimService.API**
2. Add a .NET Standard Class Library project named **ContosoClaimService.Model**
3. Add a .NET Standard Class Library project named **ContosoClaimService.Data**
4. Add a .NET Core Unit Test project named **ContosoClaimService.Tests**

**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.

## **Complete the ContosoClaimService.Model project**

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

string ClaimId

string PolicyId

DateTime DateOfClaim

DateTime DateOfSettlement

Payment Payment

* 1. Payment

long PaymentId

decimal AmountPaid

## **Complete the ContosoClaimService.Data project**

1. Install the **Microsoft.EntityFrameworkCore** (version 1.1.1) NuGet package
2. Install the **Microsoft.EntityFrameworkCore.SqlServer** (version 1.1.1) NuGet package
3. Add a Project reference to the **ContosoClaimService.Model** project
4. Create a **ClaimServiceContext** class which inherits from **DbContext**
   1. Implement **DbSets** for the Claims and Payment entities.
   2. Implement an empty constructor with **DbContextOptions** as a parameter and inherit from the base object.
   3. Override **OnModelCreating** method to create 1..1 relationship between Claim and Payment entities (A claim has one payment)
   4. Specify **Payment.AmountPaid** as a column type of **Money**
5. Implement the repository pattern by creating an **IEntityBaseRepository** interface in a new folder called **Abstract**.
   1. Add operation for GetAll() returning IEnumerable<T>
   2. Add operation for Add(T entity) returning void
6. 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 GetAll() method returning all Claims from the context.
   3. Implement the Add(T entity) method using the context to create and insert a new entity.
7. Create a **IRepositories** class in the **Abstract** folder
   1. Within the class, create an interface called **IClaimRepository** inheriting from **IEntityBaseRepository**, where **T** is **Claim**
8. Implement the **IClaimRepository** 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. Override the GetAll(), replacing **T** with **Claim** method to eager load the **Payment** entity using the include extension.
9. 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 Payment seed data to populate the database.
      * Call context.SaveChanges();

## **Complete the ContosoClaimService.API project**

1. Install the **Microsoft.EntityFrameworkCore** (version 1.1.1) NuGet package
2. Install the **Microsoft.EntityFrameworkCore.SqlServer** (version 1.1.1) NuGet package
3. Install the **AutoMapper** (version 5.2.0) NuGet package
4. Add a Project reference to the **ContosoClaimService.Model** project
5. Add a Project reference to the **ContosoClaimService.Data** project
6. Create an **appsettings.json** file in the project with a property called **DefaultConnection**. Specify your local database connectionstring as the property value.
7. 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.
8. Register services to the .NET Core container in the **ConfigureServices** method of the **Startup** class:
   1. Register and add a DbContext of type **ClaimDBContext** 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.
9. 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)
   3. Within development environment conditional, call the ClaimDBInitializer.Initialize method passing app.ApplicationServices as a parameter.
10. Create a new **ViewModels** folder. Create classes (or DTOs) for mapping domain entities to the API object model.
    1. ClaimViewModel
    2. PaymentViewModel
11. Create Automapper mapping profile classes within a **Mappers** folder inside the **ViewModel** folder.
    1. ViewModelToDomainMappingProfile
    2. DomainToViewModelMappingProfile
    3. Create the **AutoMapperConfiguration** class to initialize the mapping profiles inside a static **Configure** method.
    4. Add AutoMapperConfiguration.Configure(); to **ConfigureServices** method in the **Startup** class.
12. 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 **GetAllClaims** and set the **HttpGet** attribute and name to **GetAllClaims.**
    4. Within the **GetAllClaims** method call the claim repository to return all claims.
    5. Reference the AutoMapper map method to convert the entity to a viewmodel.
    6. Return the resulting claims in Json format.
13. Set the **ContosoClaimService.API** as Startup project and run the application from a kestrel web server instance.
    1. Navigate to **/api/Claim/GetAllClaims** to validate database items are returned from the service.

## Answers

## **Complete the ContosoClaimService.Model project**

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

namespace ContosoClaimService.Model.Entities

{

public class Claim

{

public string ClaimId { get; set; }

public string PolicyId { get; set; }

public DateTime DateOfClaim { get; set; }

public DateTime DateOfSettlement { get; set; }

public Payment Payment { get; set; }

}

}

* 1. Payment

namespace ContosoClaimService.Model.Entities

{

public class Payment

{

public long PaymentId { get; set; }

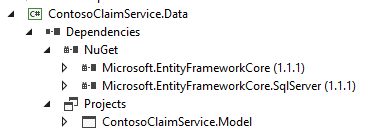
public decimal AmountPaid { get; set; }

}

}

## **Complete the ContosoClaimService.Data project**

1. Install the **Microsoft.EntityFrameworkCore** (version 1.1.1) NuGet package
2. Install the **Microsoft.EntityFrameworkCore.SqlServer** (version 1.1.1) NuGet package
3. Add a Project reference to the **ContosoClaimService.Model** project



1. Create a **ClaimServiceContext** class which inherits from **DbContext**
   1. Implement **DbSets** for the Claims and Payment entities.

public class ClaimServiceContext : DbContext

{

public DbSet<Claim> Claims { get; set; }

public DbSet<Payment> Payments { get; set; }

/\*…\*/

}

* 1. Implement an empty constructor with **DbContextOptions** as a parameter and inherit from the base object.

public ClaimServiceContext(DbContextOptions<ClaimServiceContext> options) : base(options) { }

* 1. Override **OnModelCreating** method to create 1..1 relationship between Claim and Payment entities (A claim has one payment)

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Claim>()

.ToTable("Claim");

modelBuilder.Entity<Claim>()

.HasOne(p => p.Payment);

modelBuilder.Entity<Payment>()

.ToTable("Payment");

* 1. Specify **Payment.AmountPaid** as a column type of **Money**

modelBuilder.Entity<Payment>()

.Property(i => i.AmountPaid) .HasColumnType("Money");

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

public interface IEntityBaseRepository<T> where T : class, new()

{

IEnumerable<T> GetAll();

void Add(T entity);

}

1. 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 GetAll() method returning all Claims from the context.
   3. Implement the Add(T entity) method using the context to create and insert a new entity.

namespace ContosoClaimService.Data.Repositories

{

public class EntityBaseRepository<T> : IEntityBaseRepository<T>

where T : class, new()

{

private ClaimServiceContext \_context;

public EntityBaseRepository(ClaimServiceContext context)

{

\_context = context;

}

public virtual IEnumerable<T> GetAll()

{

IQueryable<T> query = \_context.Set<T>();

return query.AsEnumerable();

}

public virtual void Add(T entity)

{

EntityEntry dbEntityEntry = \_context.Entry<T>(entity);

\_context.Set<T>().Add(entity);

}

}

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

namespace ContosoClaimService.Data.Abstract

{

public interface IClaimRepository : IEntityBaseRepository<Claim> { }

}

1. Implement the **IClaimRepository** 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. Override the GetAll(), replacing **T** with **Claim** method to eager load the **Payment** entity using the include extension.

namespace ConotosClaimService.Data.Repositories

{

public class ClaimRepository : EntityBaseRepository<Claim>, IClaimRepository

{

private ClaimServiceContext \_context;

public ClaimRepository(ClaimServiceContext context) : base(context)

{

\_context = context;

}

public override IEnumerable<Claim> GetAll()

{

IQueryable<Claim> query = \_context.Set<Claim>();

return query.Include("Payment").AsEnumerable();

}

}

}

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 Payment seed data to populate the database.
      * Call context.SaveChanges();

namespace ConotosClaimService.Data.Initializers

{

public static class ClaimDBInitializer

{

private static ClaimServiceContext context;

public static void Initialize(IServiceProvider serviceProvider)

{

context = (ClaimServiceContext)serviceProvider.GetService(typeof(ClaimServiceContext));

InitializeClaims();

}

private static void InitializeClaims()

{

context.Database.EnsureCreated();

if (context.Claims.Any())

{

return; // DB has already been seeded

}

IList<Claim> claims = new List<Claim>();

claims.Add(new Claim() {

PolicyId = "000001",

ClaimId = "12345",

Payment = new Payment() { AmountPaid = 1921.39m }});

claims.Add(new Claim() {

PolicyId = "000001",

ClaimId = "12346",

Payment = new Payment() { AmountPaid = 92.99m }});

claims.Add(new Claim() {

PolicyId = "000001",

ClaimId = "12347",

Payment = new Payment() { AmountPaid = 768.01m } });

foreach (Claim claim in claims)

{

context.Claims.Add(claim);

}

context.SaveChanges();

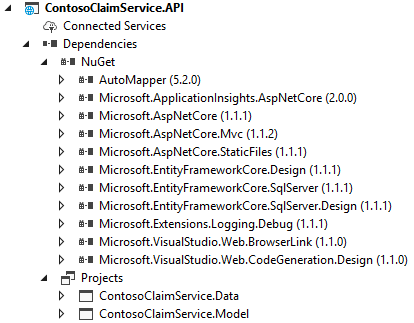
}

}

}

## **Complete the ContosoClaimService.API project**

1. Install the **Microsoft.EntityFrameworkCore** (version 1.1.1) NuGet package
2. Install the **Microsoft.EntityFrameworkCore.SqlServer** (version 1.1.1) NuGet package
3. Install the **AutoMapper** (version 5.2.0) NuGet package
4. Add a Project reference to the **ContosoClaimService.Model** project
5. Add a Project reference to the **ContosoClaimService.Data** project



1. Create an **appsettings.json** file in the project with a property called **DefaultConnection**. Specify your local database connectionstring as the property value.

{

"DefaultConnection": "Server=(local)\\SQLEXPRESS;Database=ContosoMedical;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

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.

public IConfigurationRoot Configuration { get; set; }

public Startup(IHostingEnvironment env)

{

var builder = new ConfigurationBuilder()

.SetBasePath(env.ContentRootPath)

.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)

.AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)

.AddEnvironmentVariables();

Configuration = builder.Build();

}

1. Register services to the .NET Core container in the **ConfigureServices** method of the **Startup** class:
   1. Register and add a DbContext of type **ClaimDBContext** 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.

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<ClaimServiceContext>(options =>

options.UseSqlServer(Configuration.GetValue<string>("DefaultConnection")));

// Register Repositories

services.AddScoped<IClaimRepository, ClaimRepository>();

// Automapper Configuration

AutoMapperConfiguration.Configure();

// Enable Cors

services.AddCors();

// Add MVC services to the services container.

services.AddMvc().AddJsonOptions(opts =>

{

// Force Camel Case to JSON

opts.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();

}); ;

}

1. 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)
   3. Within development environment conditional, call the ClaimDBInitializer.Initialize method passing app.ApplicationServices as a parameter.

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)

{

loggerFactory.AddConsole();

app.UseCors(builder =>

builder.AllowAnyOrigin()

.AllowAnyHeader()

.AllowAnyMethod());

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

if (env.IsDevelopment())

{

//Create and Seed Claim Database

ClaimDBInitializer.Initialize(app.ApplicationServices);

}

}

1. Create a new **ViewModels** folder. Create classes (or DTOs) for mapping domain entities to the API object model.
   1. ClaimViewModel
   2. PaymentViewModel
2. Create Automapper mapping profile classes within a **Mappers** folder inside the **ViewModel** folder.
   1. ViewModelToDomainMappingProfile

public class ViewModelToDomainMappingProfile : Profile

{

public ViewModelToDomainMappingProfile()

{

CreateMap<ClaimViewModel, Claim>().ForMember(c => c.Payment, map => map.UseValue(new Payment()));

CreateMap<PaymentViewModel, Payment>();

}

}

* 1. DomainToViewModelMappingProfile

namespace ContosoClaimService.API.ViewModels.Mappers

{

public class DomainToViewModelMappingProfile : Profile

{

public DomainToViewModelMappingProfile()

{

CreateMap<Claim, ClaimViewModel>()

.ForMember(vm => vm.ClaimId,

map => map.MapFrom(e => e.ClaimId))

.ForMember(vm => vm.PolicyId,

map => map.MapFrom(e => e.PolicyId))

.ForMember(vm => vm.DateOfClaim,

map => map.MapFrom(e => e.DateOfClaim))

.ForMember(vm => vm.DateOfSettlement,

map => map.MapFrom(e => e.DateOfSettlement));

CreateMap<Payment, PaymentViewModel>()

.ForMember(vm => vm.PaymentId,

map => map.MapFrom(e => e.PaymentId))

.ForMember(vm => vm.AmountPaid,

map => map.MapFrom(e => e.AmountPaid));

}

}

}

* 1. Create the **AutoMapperConfiguration** class to initialize the mapping profiles inside a static **Configure** method.

public class AutoMapperConfiguration

{

public static void Configure()

{

Mapper.Initialize(x =>

{

x.AddProfile<DomainToViewModelMappingProfile>();

x.AddProfile<ViewModelToDomainMappingProfile>();

});

}

}

* 1. Add AutoMapperConfiguration.Configure(); to **ConfigureServices** method in the **Startup** class.

// Automapper Configuration

AutoMapperConfiguration.Configure();

1. Create a **ClaimController** class inside a new folder called **Controllers**.
   1. Set the route of the controller to “api/[controller]”

[Route("api/[controller]")]

* 1. Inject the **ClaimRepository** container service into the controller constructor.

private readonly IClaimRepository \_claimRepository;

public ClaimController(IClaimRepository claimRepository)

{

\_claimRepository = claimRepository;

}

* 1. Create a controller action named **GetAllClaims** and set the **HttpGet** attribute and name to **GetAllClaims.**
  2. Within the **GetAllClaims** method call the claim repository to return all claims.
  3. Reference the AutoMapper map method to convert the entity to a viewmodel.
  4. Return the resulting claims in Json format.

[HttpGet("GetAllClaims", Name = "GetAllClaims")]

public IActionResult GetAllClaims()

{

var \_claims = \_claimRepository.GetAll();

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

return Json(claimVMs);

}

1. Set the **ContosoClaimService.API** as Startup project and run the application from a kestrel web server instance.
   1. Navigate to **/api/Claim/GetAllClaims** to validate database items are returned from the service.

**Integrate Patient Portal and Claim Service**

1. Create Services Folder in Portal project,
2. Create ClaimServiceClient class in Services Folder
3. Register Service in ConfigureServices method in Startup method
4. Set state configuration to Transient (explore Scoped/Singleton) options
5. Inject ClaimService into ClaimController constructor.
6. Use service operations to invoke service and map to viewmodel.
7. Surface to Dashboard and Appointment views interface display.

**Deployment Scenarios and IIS Configuration**

1. Create publish profile to local IIS. (Self Hosted Model)
2. Configure site in IIS, remove runtime by setting app pool to **No Managed Code**
3. Demonstrate targeted runtime deployment with typical IIS settings.
4. Deploy side-by-side

**.NET Core Command Line Interface (CLI)**

1. Create project from command line using dotnet new command
   1. dotnet new -t web and other options
   2. restore packages from CLI
   3. build project from CLI
   4. run project in Kestrel from CLI.

**.NET Standard Library**

1. Defines uniform set of BCL APIs for all .NET platforms to implement, independent of workload.
2. Enables developers to produce portable libraries that are usable across .NET runtimes, using this same set of APIs.
3. Reduces and hopefully eliminates conditional compilation of shared source due to .NET APIs, only for OS APIs.