

---

# DevOps Fundamentals

---

Lab 03: DevTest – Shift Left Testing

Lab Manual

---

## 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.

---

## Copyright and Trademarks

© 2017 Microsoft Corporation. All rights reserved.

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

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

For more information, see Use of Microsoft Copyrighted Content at

<http://www.microsoft.com/en-us/legal/intellectualproperty/Permissions/default.aspx>

DirectX, Hyper-V, Internet Explorer, Microsoft, Outlook, OneDrive, SQL Server, Windows, Microsoft Azure, Windows PowerShell, Windows Server, Windows Vista, and Zune 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

<b>LAB 3: SHIFT LEFT TESTING .....</b>	<b>6</b>
EXERCISE 1: ADDING A TEST PROJECT .....	7
EXERCISE 2: TEST DRIVEN DEVELOPMENT .....	8
EXERCISE 3: DEPENDENCY INJECTION .....	13
EXERCISE 4: DESIGNING FOR TESTABILITY .....	21

## Lab 3: Shift Left Testing

### Introduction

In this lab, you will learn about shift left testing and how to incorporate test-driven development through test projects and integration tests.

### Objectives

After completing this lab, you will be able to:

- Add a test project
- Add unit tests
- Understand dependency injection
- Add integration tests

### Prerequisites

None

### Estimated Time to Complete This Lab

90 minutes

### For More Information

Testing in Visual Studio: <https://www.visualstudio.com/vs/testing-tools/>

## Exercise 1: Adding a test project

### Introduction

In this exercise, you will add a unit testing project to eventually add tests to.

### Objectives

After completing this lab, you will be able to:

- Create a test project

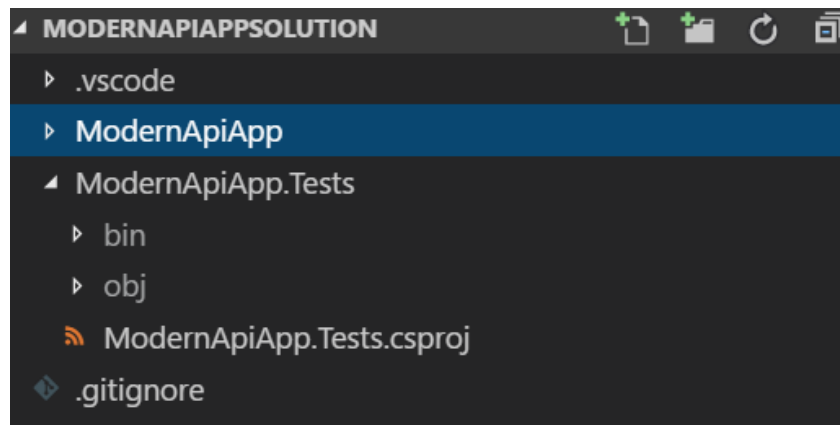
### Prerequisites

None

### Tasks

1. Open a PowerShell window, change the directory to your “[ModernApiAppSolution](#)” directory then switch to your master branch:  
“[git checkout master](#)”
2. Pull all the changes you’ve made to the master branch on Azure DevOps Services:  
“[git pull](#)”
3. Take the content from the “[Labs\Lab 03\Start\ModernApiAppSolution](#)” folder that was provided to you with the workshop materials and overwrite the content of the folder you created for the solution.

This will add a test project to your solution:



4. Make sure all the code you copied builds. Open a PowerShell window and use the following commands:  
[cd c:\hol\ModernApiAppSolution](#)  
[dotnet build ModernApiApp/ModernApiApp.csproj](#)  
[dotnet build ModernApiApp.Tests/ModernApiApp.Tests.csproj](#)
5. Run all the tests in the new test project:  
[dotnet test ModernApiApp.Tests/ModernApiApp.Tests.csproj](#)  
You will get a warning saying that there are no tests available in your test project. We will add tests in the next exercise.

## Exercise 2: Test driven development

### Introduction

In this exercise, you will add a unit test to your testing project.

### Objectives

After completing this lab, you will be able to:

- Create and add a unit test

### Prerequisites

None

### Tasks

1. From Azure DevOps Services create a story titled: “[Add a first unit test](#)”.
2. Create a topic branch for it using the same pattern used in the previous exercise (“topics/tb-123”).
3. Fetch the changes to your local repository and checkout the topic branch locally.
4. From the “[ModernApiApp.Tests](#)” project:
  - Add a new file “[CustomersControllerTests.cs](#)”, which will host all the unit tests for the customers controller:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using ModernApiApp.Controllers;

namespace ModernApiApp.Tests
{
    [TestClass]
    public class CustomersControllerTests
    {
    }
}
```

We will add a test method which **will initially fail because it will test a method that doesn't exist yet**. This allows us to use tests to **describe the requirements for the method**.

5. Add a method named file

“TestGetCustomerById\_ShouldReturnAValidCustomerWhenGivenAValidId”:

```
[TestMethod]
    public void
TestGetCustomerById_ShouldReturnAValidCustomerWhenGivenAValidId()
    {
        //Arrange
        var customersControllerUnderTest = new CustomersController();
        const int idUnderTest = 1;
        //Act
        var returnedResult =
customersControllerUnderTest.GetById(idUnderTest);

        //Assert
        Assert.IsNotNull(returnedResult, $"The GetById method didn't
return any customer for ID= {idUnderTest}");
        Assert.AreEqual(typeof(ModernApiApp.Models.Customer),
            returnedResult.GetType(),
            $"The GetById method returned the wrong type,
{returnedResult.GetType().ToString()} instead of
{typeof(ModernApiApp.Models.Customer)}");
    }
```

6. Save your changes to the local topic branch.
7. From the same PowerShell window, you’ve been using, run the following command at the solution root folder:

`dotnet build ModernApiApp/ModernApiApp.csproj`

`dotnet build ModernApiApp.Tests/ModernApiApp.Tests.csproj`

You will get compile errors, resulting from missing nuget packages, you will fix those in the next step.

8. From the PowerShell command line run the following commands:

`cd ModernApiApp.Tests`

`dotnet test`

You will get the following error:

```
Build started, please wait...CustomersControllerTests.cs(13,63): error CS1061:
'CustomersController' does not contain a definition for 'GetById' and no extension
method 'GetById' accepting a first argument of type 'CustomersController' could be
found (are you missing a using directive or an assembly reference?)
```

The test has failed because the initial class did not have the method we are testing “GetById”. We are using the test in this case to express how we expect the method to work. We need to have enough tests to express all our requirements from



such a method. We also need to have tests that make sure **the method doesn't do actions we don't want it to do**. For this exercise we will only use one test.

9. Go back to the CustomersController class (in the ModernApiApp project) and add the following method:

```
[HttpGet]
public Object GetById(int id)
{
    throw new NotImplementedException();
}
```

10. Save your changes to the topics branch.

11. From the PowerShell window, run the following commands:

```
cd .. ; dotnet build ModernApiApp/ModernApiApp.csproj
```

```
dotnet build ModernApiApp.Tests/ModernApiApp.Tests.csproj
```

```
dotnet test ModernApiApp.Tests/ModernApiApp.Tests.csproj
```

Starting test execution, please wait...

**Failed** TestGetCustomerById\_ShouldReturnAValidCustomerWhenGivenAValidId

Error Message:

Test method

ModernApiApp.Tests.CustomersControllerTests.TestGetCustomerById\_ShouldReturnAValidCustomerWhenGivenAValidId threw exception:

System.NotImplementedException: **The method or operation is not implemented.**

**This test is now failing because the code is returning an exception.** This is the exception that we have put in place to remind us that we haven't yet written the necessary code for the method.

12. Replace the controller GetById method with the following code:

```
[HttpGet]
public IActionResult GetById(int id)
{
    Customer customer = null;
    if (id > 0)
    {
        customer = new Customer(){Id= id, Name = $"Customer {id}" ,
Address = $"{{(id) * 100} Main St"  };
    }
    return new ObjectResult(customer);
}
```

13. Replace the unit test code with the following code:

```
[TestMethod]
    public void
TestGetCustomerById_ShouldReturnAValidCustomerWhenGivenAValidId()
    {
        //Arrange
        var customersControllerUnderTest = new CustomersController();
        const int idUnderTest = 1;
        //Act
        var returnedResult =
customersControllerUnderTest.GetById(idUnderTest);
        //Assert
        Assert.IsNotNull(returnedResult, $"The GetById method didn.M't
return any customer for ID= {idUnderTest}");
        Assert.AreEqual(returnedResult.GetType(), typeof(ObjectResult),
            $"The GetById returned type: {returnedResult.GetType()} instead of
{typeof(ObjectResult)}" );
        Assert.IsInstanceOfType((returnedResult as ObjectResult).Value,
typeof(Customer),
            $"The object result returned GetById contains type:
{(returnedResult as ObjectResult).Value.GetType()} instead of
{typeof(Customer)}");
        Customer returnedCustomer = (Customer)(returnedResult as
ObjectResult).Value;
        Assert.AreEqual(idUnderTest, returnedCustomer.Id,
            $"The customer result returned GetById has Id:
{returnedCustomer.Id} instead of {idUnderTest}");
    }
```

(try to write the code yourself, so you can understand the logic)

14. Add the following using statements to the test class:

```
using Microsoft.AspNetCore.Mvc;
using ModernApiApp.Models;
```

15. Save all your changed and re-run the test:

`dotnet build ModernApiApp/ModernApiApp.csproj`

`dotnet build ModernApiApp.Tests/ModernApiApp.Tests.csproj`

`dotnet test ModernApiApp.Tests/ModernApiApp.Tests.csproj`

You will get the following results:

Total tests: 1. **Passed: 1.** Failed: 0. Skipped: 0.

Test Run **Successful.**

Test execution time: 1.3140 Seconds

16. Commit the change to the local topic branch.
17. Push them to the remote topic branch.
18. Create, Approve and complete a pull request to merge the changes to the master branch.
19. Pull the changes to the local master branch.
20. Cleanup the local branches.

## Exercise 3: Dependency Injection

### Introduction

The project you are currently uses simple code but it is hard to test because of dependencies:

Consider the following line of code, which is how the tests starts:

```
var customersControllerUnderTest = new CustomersController();
```

Every time you instantiate a new customers controller, all its dependencies have to be instantiated. This means that the unit test might have a **dependency on a real database connection, a web service connection or another system dependency**. It also means that there might be a dependency on specific data in the database. There are few disadvantages to that approach:

- **Unit tests take longer to run.**
- **Unit tests running on a build server will need access to a database server.**
- **Data can get corrupted if multiple tests are modifying it.**

In addition, this is an example with one dependency only. If your application had more dependencies, then setting up tests becomes even harder.

Let's introduce the concept of dependency injection into our project and our tests. This concept exists in all development technologies. The goal from covering it in this hands-on lab is to realize the value it adds to make your code more testable.

We will add few things to the solution:

- A **data access project** containing an entity framework core data context, a Repository implementation and a Unit of work implementation.
- An **entities project** containing the models that will be served by our Api.

Instead of having you do a lot of typing, we have provided you with all the required code. You will then extend the provided code by adding more tests.

### Objectives

After completing this lab, you will be able to:

- Understand and use dependency injection

### Prerequisites

None

## Tasks

1. From Azure DevOps Services, create a new STORY and call it “[Add a database and data access project to the repo](#)”
2. Create a remote topic branch for it.
3. Duplicate the topic branch locally (fetch and checkout).
4. Switch to the topic branch.
5. Replace the content of your “C:\hol\ModernApiAppSolution” with the files provided to you “[..\ Labs\Lab 03\Tasks\exercice 3\ ModernApiAppSolution](#)”.
6. Save your changes.
7. Open a PowerShell window and go to your topic repository folder. You can use the integrated command line from within VS Code.
8. Go to the root folder of your topic branch.
9. Run the following script to restore all the projects you have copied:

[Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass](#)

(select Y)

[.\restoreApiSolutionLocally.ps1](#)

```
C:\hol\ModernApiAppSolution [master = +10 ~7 -8 !]> .\restoreApiSolutionLocally.ps1
Restoring packages for C:\hol\ModernApiAppSolution\ModernApiApp.Entities\ModernApiApp.Entities.csproj...
Restoring packages for C:\hol\ModernApiAppSolution\ModernApiApp.DataAccess\ModernApiApp.DataAccess.csproj...
Restoring packages for C:\hol\ModernApiAppSolution\ModernApiApp\ModernApiApp.csproj...
Generating MSBuild file C:\hol\ModernApiAppSolution\ModernApiApp.Entities\obj\ModernApiApp.Entities.csproj.nuget.g.props.
Restore completed in 263.47 ms for C:\hol\ModernApiAppSolution\ModernApiApp.Entities\ModernApiApp.Entities.csproj.
Generating MSBuild file C:\hol\ModernApiAppSolution\ModernApiApp.DataAccess\obj\ModernApiApp.DataAccess.csproj.nuget.g.props.
Restore completed in 393.24 ms for C:\hol\ModernApiAppSolution\ModernApiApp.DataAccess\ModernApiApp.DataAccess.csproj.
Generating MSBuild file C:\hol\ModernApiAppSolution\ModernApiApp\obj\ModernApiApp.csproj.nuget.g.props.
Restore completed in 960.45 ms for C:\hol\ModernApiAppSolution\ModernApiApp\ModernApiApp.csproj.
Restoring packages for C:\hol\ModernApiAppSolution\ModernApiApp.Tests\ModernApiApp.Tests.csproj...
Restore completed in 93.71 ms for C:\hol\ModernApiAppSolution\ModernApiApp.Entities\ModernApiApp.Entities.csproj.
Restore completed in 104.75 ms for C:\hol\ModernApiAppSolution\ModernApiApp.DataAccess\ModernApiApp.DataAccess.csproj.
Restore completed in 133.93 ms for C:\hol\ModernApiAppSolution\ModernApiApp\ModernApiApp.csproj.
Generating MSBuild file C:\hol\ModernApiAppSolution\ModernApiApp.Tests\obj\ModernApiApp.Tests.csproj.nuget.g.props.
Restore completed in 1.31 sec for C:\hol\ModernApiAppSolution\ModernApiApp.Tests\ModernApiApp.Tests.csproj.
```

10. Stop and remove all running containers:
11. Use the following command to build the docker images required for the application and run them as containers:

[docker stop \\$\(docker ps -a -q\)](#)

[docker rm \\$\(docker ps -a -q\)](#)

[docker-compose -f "dc.ApiApp.yml" up -d --build](#)

This command builds two containers, one for API application and one for the SQL database and opens all the required ports so that the two containers can exchange data.

[docker ps](#)

This command should show your containers:

```
C:\hol\ModernApiAppSolution [master ≡ +8 ~7 -0 !]> docker ps
CONTAINER ID        IMAGE               COMMAND
bf9ec4fe5b60       modernapiapp       "dotnet ModernApiApp..."
c8e2e8f19214       modernapiapp.database "powershell -Command..."
```

12. Open your browser and go to the following URL, you should get a list of 4 customers:

<http://localhost:8080/api/customers>

13. Stop the composed infrastructure (containers and network):

```
docker-compose -f "dc.ApiApp.yml" up -d --build
```

14. From the command line:

```
git add --all
```

```
git commit -m 'Added a data access project, and a database to the solution'
```

```
git push
```

We will revisit the concept of containers later in a hands-on lab.

15. Create, approve and complete a pull request to merge the changes to the master branch.
16. Pull the changes to the local master branch.
17. Cleanup the local branches.
18. From the command line, use the following command to open the master branch using Visual Studio Code:

[Code](#) .

Examine the following projects and files to understand the implementation and the value of dependency injection:

19. Expand the [ModernApiApp.DataAccess](#) project files
20. Open and examine the [ApiContext.cs](#) file:

```
public partial class ApiContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public ApiContext()
    {
    }
    public ApiContext(DbContextOptions<ApiContext> options) : base(options) { }
}
```

This class defines the entity framework [datacontext](#) which is the class that manages the communication with the database. The class however doesn't know where the database is and will get that information at runtime. This allows us to swap a database for each environment and allows us to use in memory databases as well, while keeping the same data schema.

21. Open and examine the [IRepository.cs](#) file:

```
public interface IRepository<TEntity> where TEntity : class
{
    void Delete(TEntity entity);
    void Edit(TEntity entity);
    IQueryable<TEntity> GetAll();
    Task<TEntity> GetByIdAsync(int id);
    void Add(TEntity entity);
    IQueryable<TEntity>
    SearchFor(System.Linq.Expressions.Expression<Func<TEntity, bool>> predicate);
}
```

This class defines the database operations that will be exposed to the other application layers. It is also generic, allowing it to be used with any entity (data table). This approach allows us to unify the way we communicate with any database entity. Hence minimizing the code we have to test for data access, and making sure enhancements or bug fixes to this class positively impact the overall project.

22. Open the file [IUnitOfWork.cs](#)

```
public interface IUnitOfWork<TDbContext> where TDbContext : DbContext, new()
{
    TDbContext DbContext { get; }
    System.Threading.Tasks.Task<int> SaveAsync();
}
```

This interface defines the unit of work operations which include persisting any operations performed by one or many repositories. It allows our database operations to be atomic.

For more information on the Repository and Unit of Work design patterns, refer to the following: <https://www.martinfowler.com/eaCatalog/unitOfWork.html>

<https://www.martinfowler.com/eaCatalog/repository.html>

23. Expand the `ModernApiApp` project files:

24. Open the `Startup.cs` file:

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }

// This method gets called by the runtime. Use this method to add services to the
// container.
public void ConfigureServices(IServiceCollection services)
{
    var connectionString = buildConnectionString();
    services.AddDbContext<ApiContext>(options => options.UseSqlServer(
        connectionString,
        b => b.MigrationsAssembly("ModernApiApp")
    ));

    services.AddTransient<IUnitOfWork<ApiContext>,
    UnitOfWork<ApiContext>>();
    services.AddMvc();
}
```

Notice how in the `ConfigureServices` method, we register an `ApiContext`, we ask it to use SQL server and we give it a connection string. **Any other class in the project that needs an `ApiContext` will simply ask for one, without needing to know of the details about what kind of database it is or where it is.** This is an example of the concept of dependency injection.

Notice also how the `buildConnectionString` method builds a `connectionString` to the database, by **reading environment variables**. This allows secrets to be **stored within the environment** itself and allow us to easily **swap those secrets** during our release process.

25. Open the `CustomersController.cs` under the Controllers folder:

```
[Produces("application/json")]
[Route("api/Customers")]
public class CustomersController : Controller
{
    IUnitOfWork<ApiContext> _uow;
    IRepository<Customer> _customerRepository;
```



```

public CustomersController(IUnitOfWork<ApiContext> uow)
{
    _uow = uow;
    _customerRepository = new Repository<Customer>(_uow.DbContext);
}

// GET api/customers
[HttpGet]
public IEnumerable<Customer> Get()
{
    var customers = _customerRepository.GetAll().Take(10);
    return customers.ToArray();
}

public IActionResult GetById(int id)
{
    Customer customer = null;
    if (id > 0)
    {
        customer = new Customer() { Id = id, Name = $"Customer {id }",
Address = $"{(id) * 100} Main St" };
    }
    return new ObjectResult(customer);
}
}

```

Notice how the controller simply expresses that it needs a [UnitOfWork](#) class containing an [ApiContext](#), which will handle data persistence. It relies on [dependency injection](#), to find the implementation in the services we registered in the “[Startup.cs](#)” class. The rest of the data access operations aren’t aware of any of this injection.

The [advantage](#) of using such a pattern is that it [allows to inject or swap the database context at runtime](#), which means that in our tests we will not need to use a real database.

26. Expand the [ModernApiApp.Tests](#) project files:

27. Open the [Stubs.cs](#) file:

```

public static IUnitOfWork<ApiContext> setUpUnitOfWorkMock()
{
    var options = new DbContextOptionsBuilder<ApiContext>()
        .UseInMemoryDatabase(databaseName: "InMemoryApiAppDatabase")
        .Options;

    _apiContext = new ApiContext(options);
    for (int i = 0; i < 100; i++)
    {
        _apiContext.Customers.Add(new Customer { Id = i + 1, Name = $"Name
{i}", Address = $"{(i + 1) * 10} main street" });
    }
}

```

```

    }
    _apiContext.SaveChanges();
    var uow = new UnitOfWork<ApiContext>(_apiContext);
    return uow;
}

```

This method generates an `IUnitOfWork` for the `ApiContext` class that uses an entity framework in memory database.

28. Open the `CustomersControllerTests.cs` file:

```

public CustomersControllerTests()
{
    _uow = Stubs.setUpUnitOfWorkMock();
    _customerRepository = new Repository<Customer>(_uow.DbContext);
}

[TestMethod]
public void
TestGetCustomerById_ShouldReturnAValidCustomerWhenGivenAValidId()
{
    //Arrange
    var customersControllerUnderTest = new CustomersController(_uow);
    int idUnderTest = _customerRepository.GetAll().FirstOrDefault().Id;
    //Act
    var returnedResult = customersControllerUnderTest.GetById(idUnderTest);
    //Assert
    Assert.IsNotNull(returnedResult, $"The GetById method didn't return any
customer for ID= {idUnderTest}");
    Assert.AreEqual(returnedResult.GetType(), typeof(ObjectResult),
        $"The GetById returned type: {returnedResult.GetType()} instead of
{typeof(ObjectResult)}");
    Assert.IsInstanceOfType((returnedResult as ObjectResult).Value,
typeof(Customer),
        $"The object result returned GetById contains type: {(returnedResult
as ObjectResult).Value.GetType()} instead of {typeof(Customer)}");
    Customer returnedCustomer = (Customer)(returnedResult as
ObjectResult).Value;
    Assert.AreEqual(idUnderTest, returnedCustomer.Id,
        $"The customer result returned GetById has Id: {returnedCustomer.Id}
instead of {idUnderTest}");
}

```

Notice how we **inject** the stubbed unit of work to the customer repository. The rest of the test and the Api itself don't see any difference. **We have basically made the test simpler by not having to worry about having a real database with real data at test time.**

If this concept is still unclear, please ask your instructor to walk you through the code.

29. From your shell windows, execute the following commands:

```
cd .\ModernApiApp.Tests\  
dotnet test
```

Notice how the test is successful. We can invoke the Api controller **without using a real database**.

## Exercise 4: Designing for testability

### Introduction

The application we have created so far, exposes data as a REST API. We are going now to add another application that uses the data from the REST API and evaluate how we would test such interactions.

### Objectives

After completing this lab, you will be able to:

- Add integration tests to a project

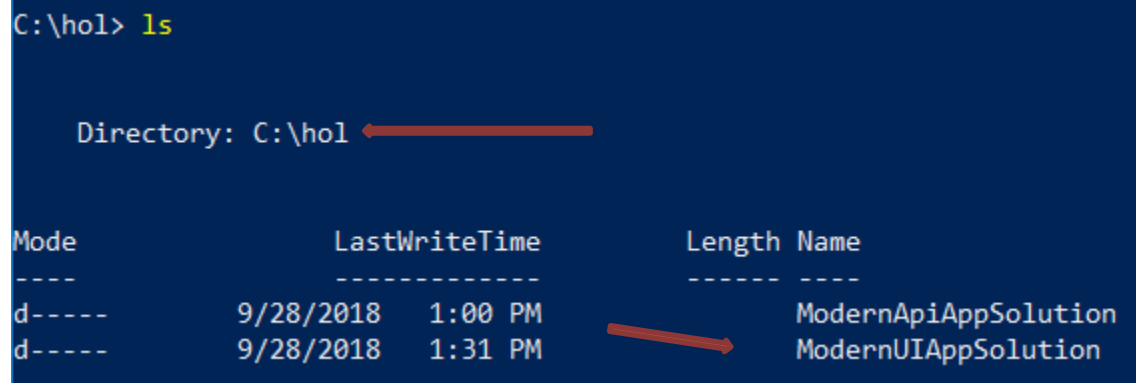
### Prerequisites

None

### Tasks

Execute these next steps **with minimal visual assistance**:

- Copy the folder “[Labs\Lab 03\Tasks\exercise 4\ModernUIAppSolution](#)” to your local “[hol](#)” folder.

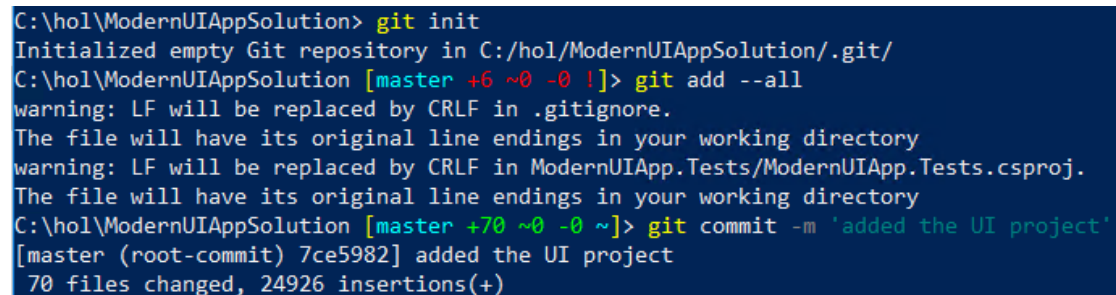


```
C:\hol> ls

Directory: C:\hol

Mode                LastWriteTime         Length Name
----                -
d-----          9/28/2018   1:00 PM           ModernApiAppSolution
d-----          9/28/2018   1:31 PM           ModernUIAppSolution
```

- Initialize a git repository in that folder, stage all files and commit them locally.



```
C:\hol\ModernUIAppSolution> git init
Initialized empty Git repository in C:/hol/ModernUIAppSolution/.git/
C:\hol\ModernUIAppSolution [master +6 ~0 -0 !]> git add --all
warning: LF will be replaced by CRLF in .gitignore.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in ModernUIApp.Tests\ModernUIApp.Tests.csproj.
The file will have its original line endings in your working directory
C:\hol\ModernUIAppSolution [master +70 ~0 -0 ~]> git commit -m 'added the UI project'
[master (root-commit) 7ce5982] added the UI project
70 files changed, 24926 insertions(+)
```

- From Azure DevOps Services, go to the repositories management in project settings, and add another repository named “[ModernUIAppRepository](#)”.
- Push the code from your local “[ModernUIAppSolution](#)” folder to the “[ModernUIAppRepository](#)” on Azure DevOps Services.

```

C:\hol\ModernUIAppSolution [master]> git remote add origin https://iliasj@dev.azure.com/iliasj/hol/_git/ModernUIAppRepository
C:\hol\ModernUIAppSolution [master]> git push -u origin --all
Enumerating objects: 96, done.
Counting objects: 100% (96/96), done.
Delta compression using up to 4 threads
Compressing objects: 100% (92/92), done.
Writing objects: 100% (96/96), 529.79 KiB | 3.44 MiB/s, done.
Total 96 (delta 7), reused 0 (delta 0)
remote: Analyzing objects... (96/96) (38 ms)
remote: Storing packfile... done (107 ms)
remote: Storing index... done (33 ms)
To https://dev.azure.com/iliasj/hol/_git/ModernUIAppRepository
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

```

By having the two repositories in a parent folder containing all the code you may need, you can easily switch between working on the Database, the API or the UI yet preserve modularity by having distinct repositories:

```

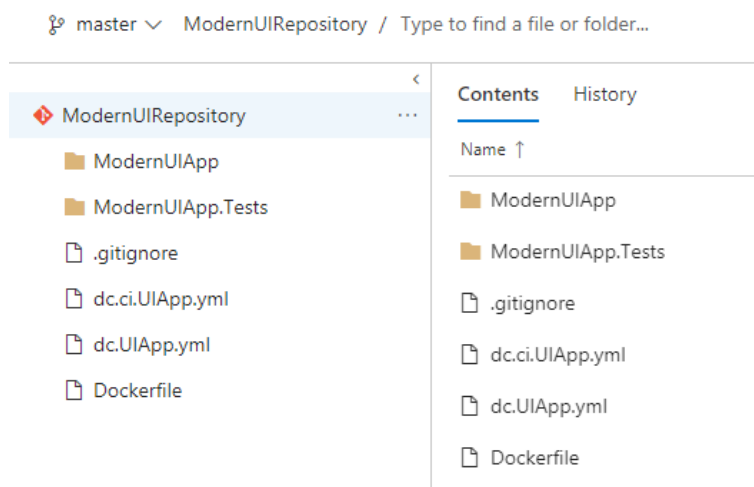
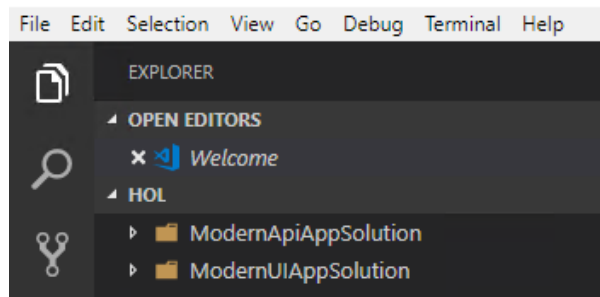
C:\hol\ModernUIAppSolution [master =>]> cd ..
C:\hol> ls

Directory: C:\hol

Mode                LastWriteTime         Length Name
----                -
d-----          9/28/2018   1:00 PM             ModernApiAppSolution
d-----          9/28/2018   2:18 PM             ModernUIAppSolution

C:\hol> Code .



```




The web application will retrieve customer's data from the REST API you've built in the previous exercises and display it in web view. Our first step will be to capture this requirement:

- Go the product backlog and add a new User Story: “[Add customers list UI](#)”
- In the user story acceptance criteria add the following:
  - Customer data is displayed in a list format.
  - Customer data is ordered by customer name.

### 823 Add customers list UI

 Ilias Jennane  0

State	<span style="color: blue;">●</span> Committed	Area	
Reason	 Commitment made by the t	Iteration	

**Description**

**Acceptance Criteria**

- Customer data is **displayed** in a list format.
- Customer data is **ordered** by customer name.

Create a topic branch for this STORY, from the [ModernUIAppRepository](#) repository.

Create a branch

Name

topics/tb-823

Based on

ModernUIAppRepository master

Work items to link

Search work items by ID or title

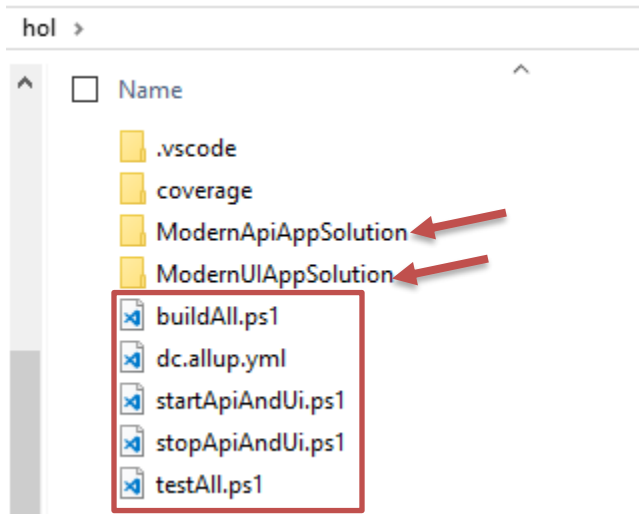
823 Add customers list UI  
Updated 3 minutes ago, Committed

Create branch Cancel

- Checkout the topic branch locally.
- Copy the “[Customer.cs](#)” file provided to you in the “[..\Labs\Lab 03\Tasks\exercise 4\Helper Files\Models](#)” folder to your “[ModernUIAppSolutionFolder\Models](#)” folder:
- Examine the content of the “[Customer.cs](#)” class from VS Code.
- Commit the changes to your local topic branch.
- Push the changes to the remote topic branch.
- Create a pull request to merge your changes into the master branch
- Clean up your local branches.

For your convenience we have added few files to help you build, start, test and stop the application layers location.

- Copy the following files from the “[..\Labs\Lab 03\Tasks\exercice 4\ Helper Files\Scripts](#)” folder to your upmost root folder “[C:\hol](#)”. The one containing both the UI and the API:



- Open a PowerShell window in the same folder and run the following commands:
  - [.\stopApiAndUi.ps1](#)
  - [.\startApiAndUi.ps1](#) (Make sure all the docker containers are built)
  - [.\testAll.ps1](#) (Make sure all the tests are passing)
- Open any internet browser and go to the following URL:  
<http://localhost:9090/customers> you should get the following web page:

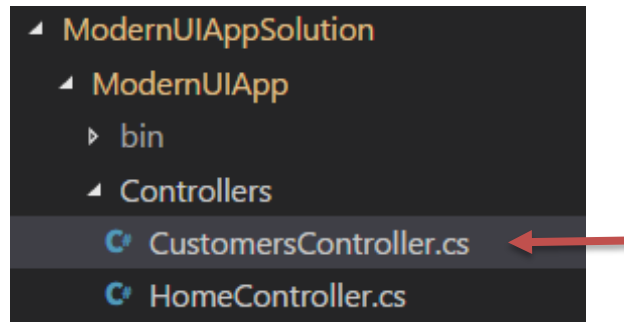
ModernUIApp   Home   About   Contact				
Customers				
<a href="#">Create New</a>				
Id	Name	Address		
1	Joe Doe	100 main street	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>	
2	Jane Doe	200 main street	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>	
3	Jimmy Doe	300 main street	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>	
4	Jesse Doe	400 main street	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>	

- Stop all the running containers and clean up any unused docker images by using the following command  
[.\stopApiAndUi.ps1](#)

**This will be the process you will, build and run the application layers locally going forward.**

**Note:** We can test the API Layer separately. However, for us to test the user interface layer, we are forced to have the whole system running. Two additional layers. Let's try to understand why this is the case.

- Open the “CustomersController.cs” class in the “ModernUIApp:



- Note how the class constructor has a dependency on the API layer:

```
public class CustomersController : Controller
{
    private HttpClient _customersApiClient;
    private string _customerApiUrl;
    public CustomersController(IConfiguration
configuration)
    {
        _customersApiClient = new HttpClient();
        customerApiUrl =
configuration.GetValue<string>("ApiURL");
    }
    public IActionResult Index()
    {
        var customers = getCustomersFromApi();
        return View(customers);
    }
}
```

The constructor also has a dependency on an HTTP client. Which means it will need to make an I/O operation during the test.

There are few options on how to work around this, to **make our code more testable**.

- Mock the HTTP client, then mock the API call return.
- Or: Refactor the code and abstract completely where the data is coming from, and have the class depend on an Interface rather than a concrete implementation.

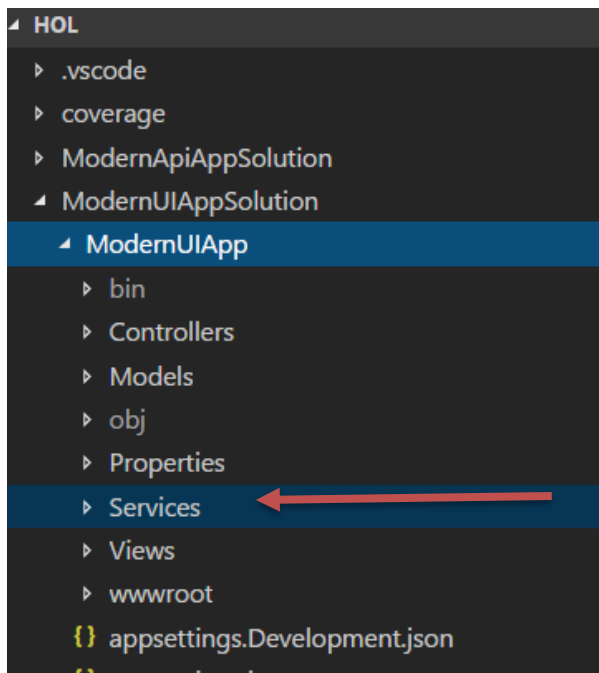
We would be using the following design pattern:

[https://en.wikipedia.org/wiki/Bridge\\_pattern](https://en.wikipedia.org/wiki/Bridge_pattern)

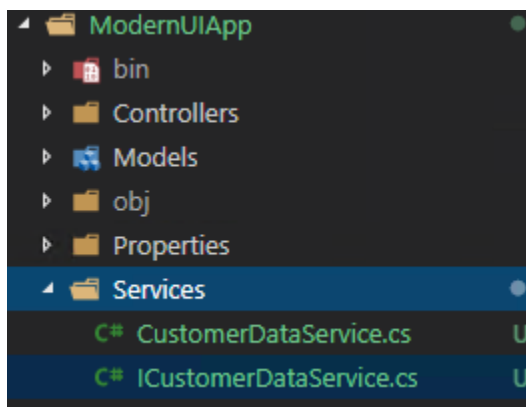
We will use this approach to make our code more testable.



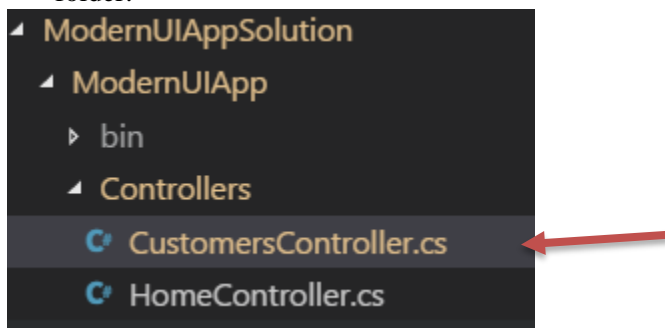
- Add a folder at the root of the “[ModernUIApp](#)” and name it “[Services](#)”:



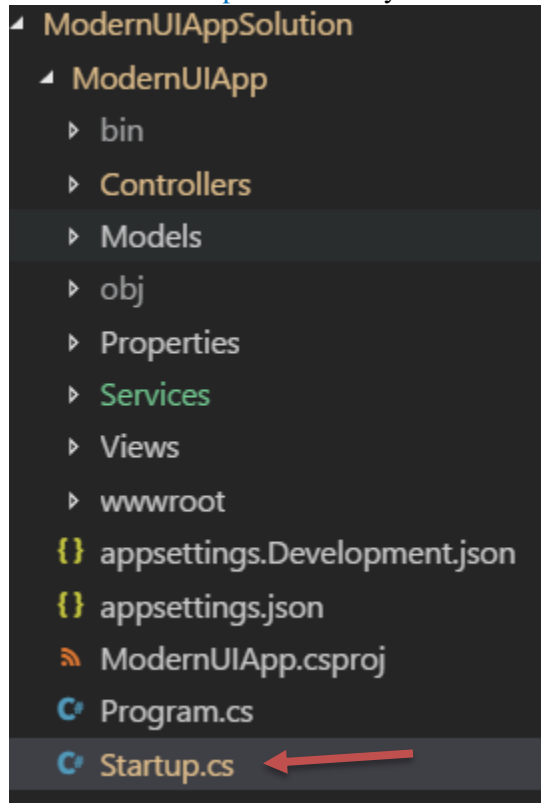
- Copy the two files from the “[..\Labs\Lab 03\Tasks\exercice 4\Helper Files\Services](#)” folder to the services folder you just created.



- Replace the “[CustomersController.cs](#)” file from your “[ModernUIApp\Controllers](#)” folder with the one provided in the “[..\Labs\Lab 03\Tasks\exercice 4\ Helper Files\Controllers](#)” folder:



- Locate the “Startup.cs” class in your “ModernUIApp” project:



add the following code to the “ConfigureServices” method

```
services.AddScoped<ICustomerDataService, CustomerDataService>();
```

You will also need to add the following code at the top of the Startup class:

```
using ModernUIApp.Services;
```

```
public class Startup
{
    0 references
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

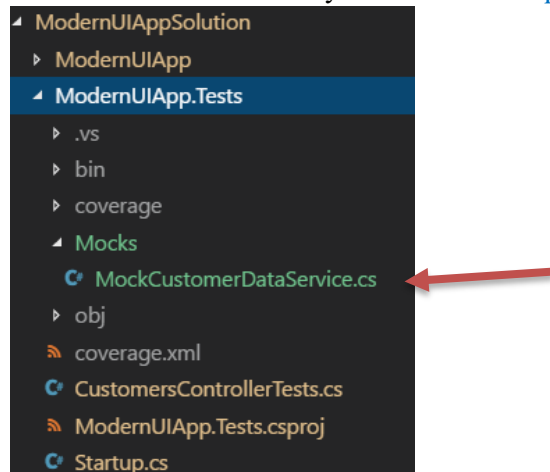
    1 reference
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
        services.AddScoped<ICustomerDataService, CustomerDataService>();
    }
}
```

This line of code declares which implementation should be used in the UI project, for the “[ICustomerDataService](#)” interface. Obviously, we will want the real implementation in this case “[CustomerDataService](#)”. This means that any controller or class that has such a dependency can simply declare that it needs an “[ICustomerDataService](#)” and the application will figure out which implementation of that interface needs to be used.

This also allows us to swap the real implementation with a “mocked” one during our tests, removing the hard dependency.

- Add a folder to the root of your “[ModernUIApp.Tests](#)” project and name it “[Mocks](#)”



- Copy the “[MockCustomerDataService.cs](#)” from the “[..\Labs\Lab 03\Tasks\exercise 4\Helper Files\Mocks](#)” folder to the “[Mocks](#)” folder you just created.
- Replace the “[CustomersControllerTests.cs](#)” in the root folder of your “[ModernUIApp.Tests](#)” project with the “[CustomersControllerTests.cs](#)” file from the “[..\Labs\Lab 03\Tasks\exercise 4\Helper Files\Tests](#)” folder.
- Save all your changes and run the test utility we provided you with earlier:  
`.\buildAll.ps1`  
`.\stopApiAndUi.ps1`  
`.\testAll.ps1`

```
C:\hol> .\testAll.ps1
Test run for C:\hol\ModernUIAppSolution\ModernApiApp.Tests\bin\Debug\netcoreapp2.2\ModernApiApp.Tests.dll(.NETCoreApp,Version=v2.2)
Microsoft (R) Test Execution Command Line Tool Version 16.0.1
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...

Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 1.5445 Seconds
Test run for C:\hol\ModernUIAppSolution\ModernUIApp.Tests\bin\Debug\netcoreapp2.2\ModernUIApp.Tests.dll(.NETCoreApp,Version=v2.2)
Microsoft (R) Test Execution Command Line Tool Version 16.0.1
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...

Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 1.7869 Seconds
```

Note that **we didn't have to launch any of the applications to tests them**. This was possible by removing the hard dependencies:

```
1 reference | You, 7 days ago | 1 author (You)
public class CustomersController : Controller
{
    2 references
    private ICustomerDataService _customerDataService;
    1 reference
    public CustomersController(ICustomerDataService customerDataService)
    {
        _customerDataService = customerDataService;
    }
    1 reference
    public IActionResult Index()
    {
        var customers = _customerDataService.GetAll();
        return View(customers);
    }
}
```

The Customers controller **doesn't need to know where the data is coming from**, it just declares that it need a data service.

```
2 references
private ICustomerDataService _customerDataService;
0 references
public CustomersControllerTests () {
    // Arrange
    var services = new ServiceCollection();
    services.AddTransient<ICustomerDataService, MockCustomerDataService>();
    var serviceProvider = services.BuildServiceProvider();

    _customerDataService = serviceProvider.GetService<ICustomerDataService>();
}
```

The test class can then pass a “mocked” or “fake” data service, but still test the controller functionality:

```
[TestCategory ("L0")]
[TestMethod]
0 references | Run Test | Debug Test
public void Index__ShouldReturnAViewContainingAListOfCustomersSortedAlphabetically () {
    //Arrange
    var customersControllerUnderTest = new CustomersController (_customerDataService);
    //Act
    var returnedResult = customersControllerUnderTest.Index ();
    // Assert
    Assert.AreEqual (returnedResult.GetType (), typeof (ViewResult));
    var returnedView = returnedResult as ViewResult;
    Assert.AreEqual (returnedView.Model.GetType (), typeof (List<Customer>), "Customers co
}
```