

# PizzaPlace Guide

## **Prerequisite:**

Computer:

This guide assumes you are running on Windows and have Visual Studio with Git installed.

You:

This code is written in C#. It does not require extensive knowledge of this coding language. It requires you to understand common coding practices and to be inquisitive about stuff that you might not (yet) be familiar with. Getting the code explained might be easier if you have access to Github Copilot that should be able to explain code segments.

# 1 Starting

**Goal:** After this you should have an understanding of endpoints, debugging, dependency injection and Git commits.

## 1.1 Basics

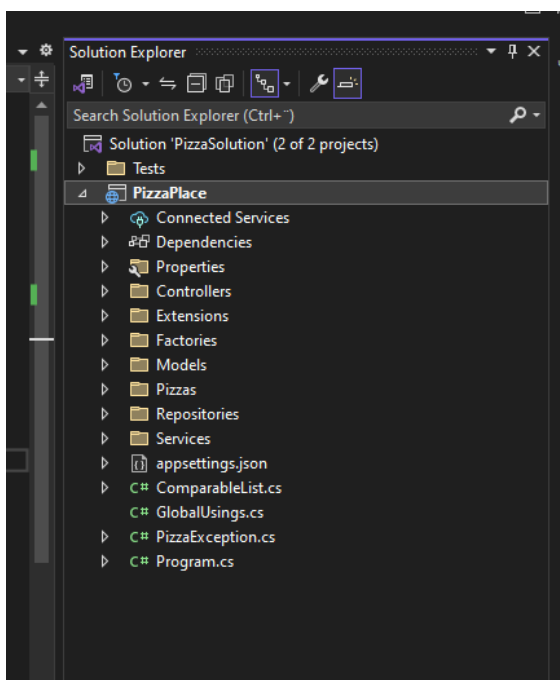
 PizzaSolution.sln	28-12-2023 08:00	Visual Studio Solu...	2 KB
---	------------------	-----------------------	------

Have your solution folder unzipped and open the PizzaSolution in Visual Studio - the latter can be accomplished by e.g. opening the PizzaSolution.sln.

The sample code is a very simple ASP.NET Core web service. This means that it has endpoints that are reachable through http protocol.

A web service can transfer and receive data across the internet or in a container setup in Kubernetes between deployed services.

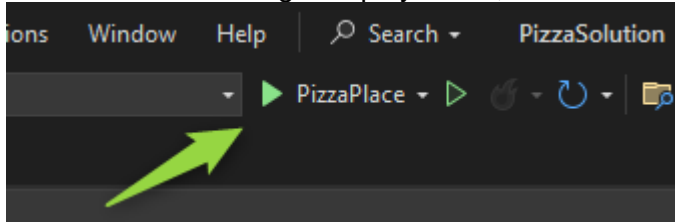
On the right side you have the Solution Explorer, that mostly serves as a simple file explorer for the solution



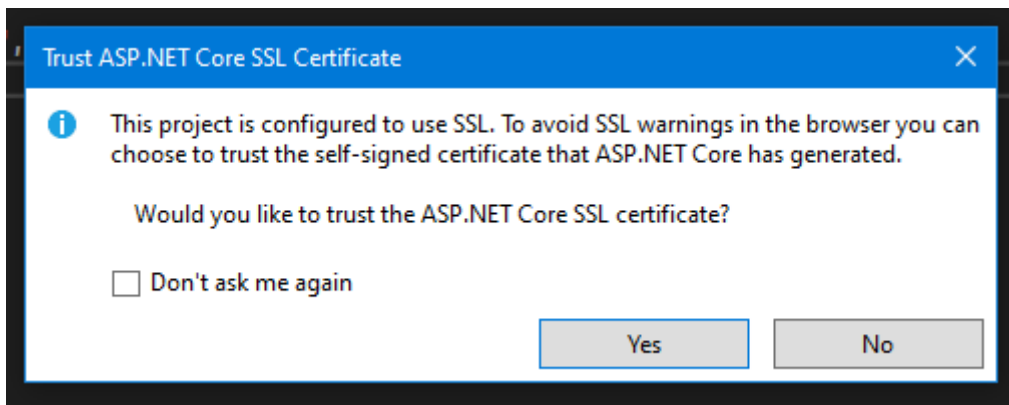
A solution can contain many projects: In this solution there is only one web service project: "PizzaPlace" but there is also a project inside the "Tests" folder called "PizzaPlace.Test". The latter will be referred to as the test project.

In the middle of the screen should be some "play" arrows. It should say "PizzaPlace" next to

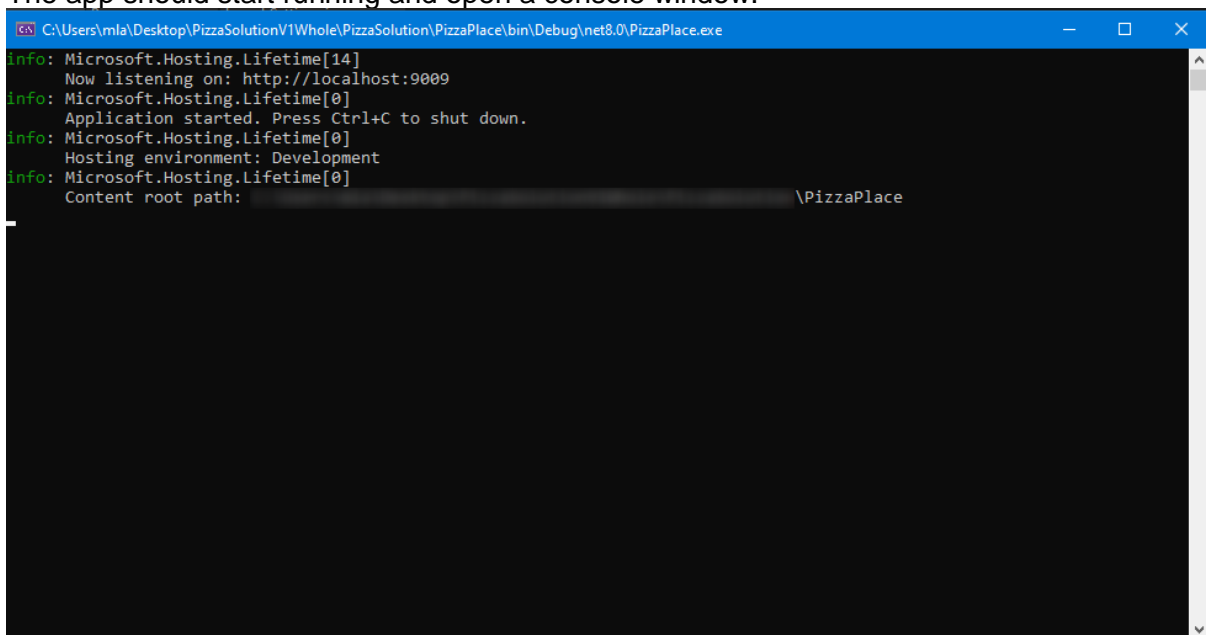
them. Click the solid green play arrow, to start the PizzaPlace web service in debug mode:



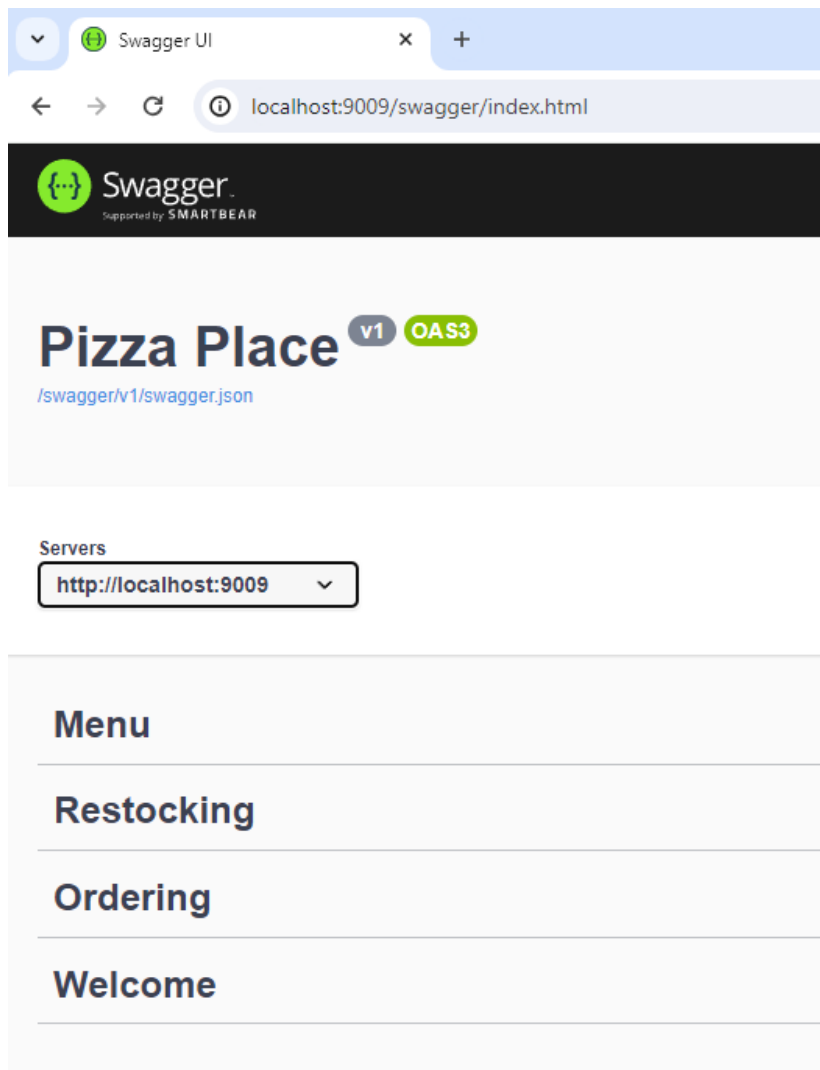
You might be asked about certificates. Check “Don’t ask me again” and click yes, then install when prompted.



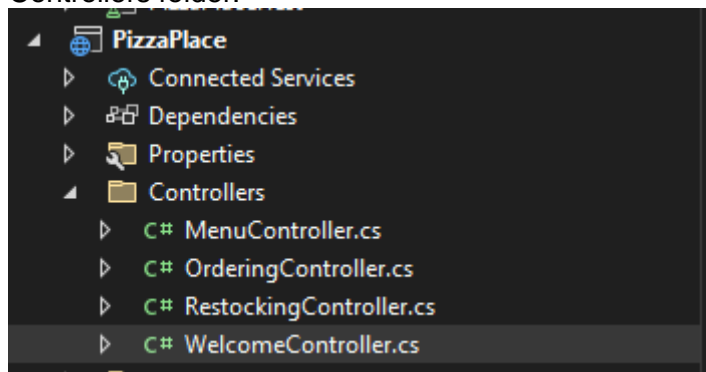
The app should start running and open a console window:



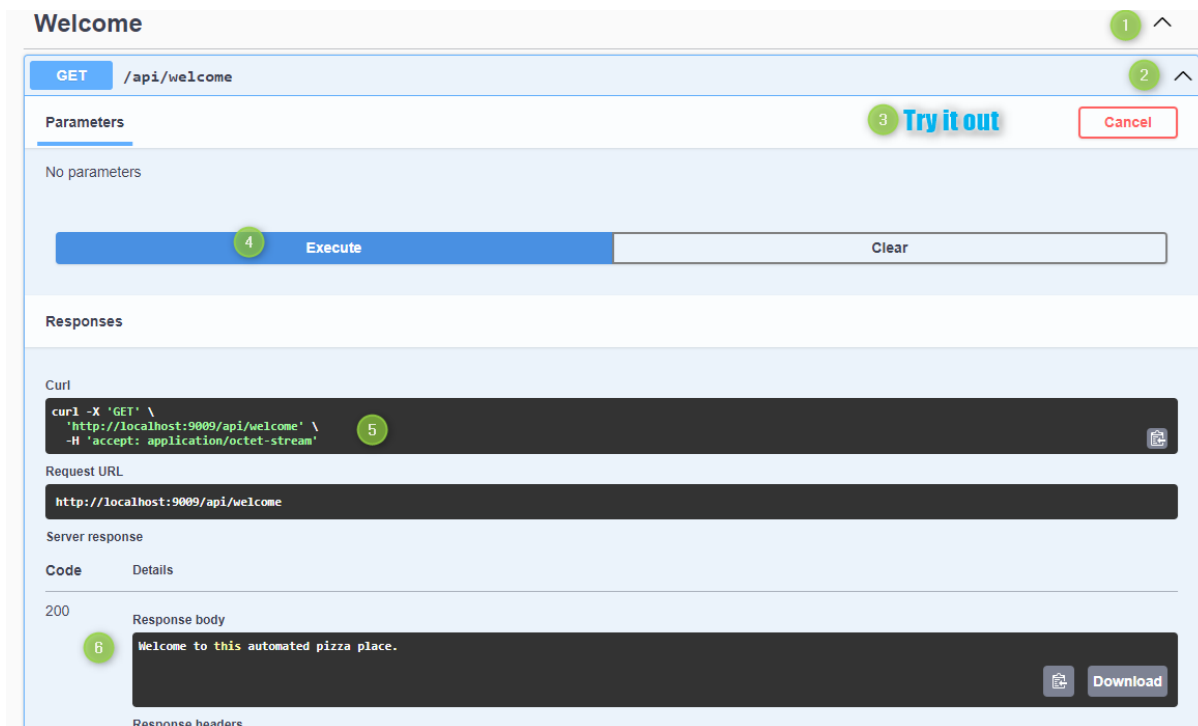
Open a browser and go to “localhost:9009/swagger” - this will be referred to as the Swagger page:



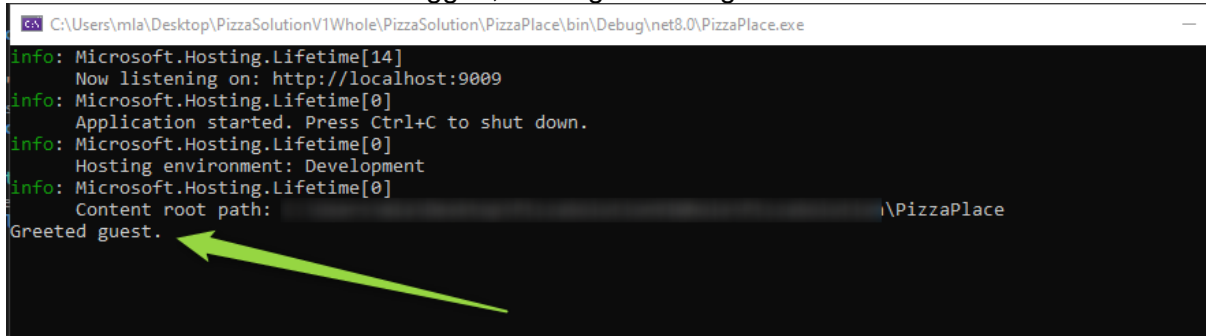
This shows four controllers corresponding to the controllers defined in the code in the Controllers folder:



On the Swagger page expand the Welcome endpoints (1), expand the Get endpoint /api/welcome (2), click "Try it out" (3) (will change to a cancel button afterwards). This endpoint doesn't require any input, so click "Execute" (4). The curl command is then written for us (5), and more important we should see that "<http://localhost:9009/api/welcome>" returns a 200 with a string response body (6) giving a welcome.

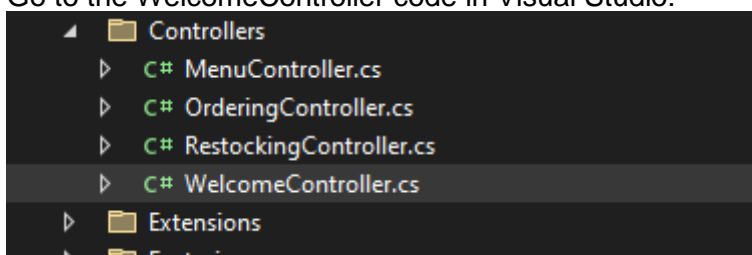


In the console it will also have logged, that it greeted a guest:



## 1.2 Debugging and dependency injection

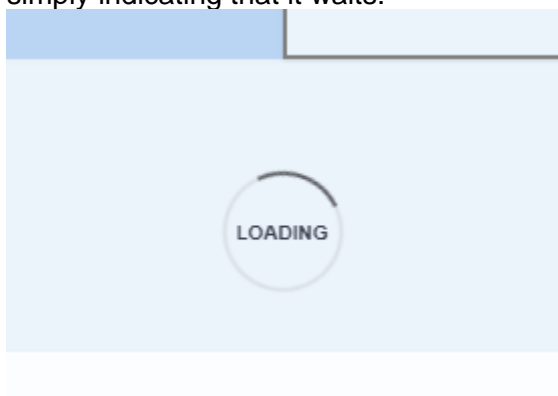
Go to the WelcomeController code in Visual Studio:



This is where the endpoints are defined. This controller only has a single Get endpoint. Click the left margin to place a breakpoint - this is noted with a red dot and can be removed by clicking the dot again.

```
1 using Microsoft.AspNetCore.Mvc;
2
3 namespace PizzaPlace.Controllers;
4
5 [Route("api/welcome")]
6 public class WelcomeController : ControllerBase
7 {
8     [HttpGet]
9     public IActionResult Greet()
10     {
11         Console.WriteLine("Greeted guest.");
12         return Ok("Welcome to this automated pizza place.");
13     }
14 }
15
16
```

Go back to the Swagger page and execute the welcome endpoint again. You should now experience that a response is not delivered back to the page. Instead a loading symbol is simply indicating that it waits:



In the code in Visual Studio the line next to the breakpoint should be marked:



```
1 using Microsoft.AspNetCore.Mvc;
2
3 namespace PizzaPlace.Controllers;
4
5 [Route("api/welcome")]
6 public class WelcomeController : ControllerBase
7 {
8     [HttpGet]
9     public IActionResult Greet()
10     {
11         Console.WriteLine("Greeted guest.");
12
13         return Ok("Welcome to this automated pizza place.");
14     }
15 }
```

This signifies that the program is currently stopped at this line of code for you to debug. There isn't much information to get here. It is possible to let the program continue by clicking the "Continue" or clicking F5.

However if instead clicking F10, then the program will only continue to the next line of code. This can be useful to follow the run logic more closely. Try clicking F10 (three times in total) to let the program proceed. In the Swagger page you should now see the response.

Go to the MenuController and set a breakpoint at the return statement:

```
1 using Microsoft.AspNetCore.Mvc;
2 using PizzaPlace.Services;
3
4 namespace PizzaPlace.Controllers;
5
6 [Route("api/menu")]
7 public class MenuController(TimeProvider timeProvider, IMenuService menuService) : ControllerBase
8 {
9     [HttpGet]
10     public IActionResult GetMenu()
11     {
12         return Ok(menuService.GetMenu(timeProvider.GetUtcNow()));
13     }
14 }
```

Go to the Swagger page and run the Get endpoint in the Menu. Similar to before the program should stop at the breakpoint you inserted.

What we want to do now is to "Step into" - basically that means that we want to examine method calls on that specific line. For the current situation it means that we would go to the code for the "GetMenu" method on the menuService.

To "Step into" click F11. Then you should be shown the code, that is actually in the MenuService class:



```
1 using PizzaPlace.Models;
2
3 namespace PizzaPlace.Services;
4
5 public class MenuService : IMenuService
6 {
7     public Menu GetMenu(DateTimeOffset menuDate)
8     {
9         throw new NotImplementedException("No menu has been implemented yet.");
10     }
11 }
12
```

Notice the arrow in the left margin indicating where in the code we have currently stopped. This service doesn't have any interesting code, since it hasn't been implemented yet. Continuing will throw the exception and then return a 500 response indicating internal server error.

**Note:** In real applications one should catch these exceptions so that they are not returned to the caller. Then give the caller some more generic message but keep the internal description of the error in a logging system. We are not going to do that however.

We are now going to examine why the MenuService was called at all.

You might have noticed that the MenuService implements the interface IMenuService. Going to the controller MenuController look at the (primary) constructor:

```
6 [Route("api/menu")]
7 public class MenuController(TimeProvider timeProvider, IMenuService menuService) : ControllerBase
8 {
9     [HttpGet]
10     public IActionResult GetMenu()
11     {
12         return Ok(menuService.GetMenu(timeProvider.GetUtcNow()));
13     }
14 }
15
```

We can see that the MenuController in its constructor has both a TimeProvider as well as the interface IMenuService.

Click with the mouse on the GetMenu on line 12, then press F12. This will transfer you to the IMenuService interface:

```
5 public interface IMenuService
6 {
7     Menu GetMenu(DateTimeOffset menuDate);
8 }
9
```

You can go to implementation from here - or you could have done so directly from the



MenuController - by pressing ctrl+F12 while GetMenu is selected. Doing so will bring you to MenuService - however this only works since in the solution there is only one implementation of this interface. Otherwise it would give you the option to select which implementation you would want to go to.

So what we know is that a MenuController requires a class implementing IMenuService in its constructor, and that MenuService implements the IMenuService interface. There is one last thing needed for the injection of the IMenuService dependency in the MenuController to work. That is to register the MenuService on the interface when setting up the application - then ASP.NET Core knows which class to create when an IMenuService is required.

This is done in the Program.cs on line 40, and that is the reason it works:

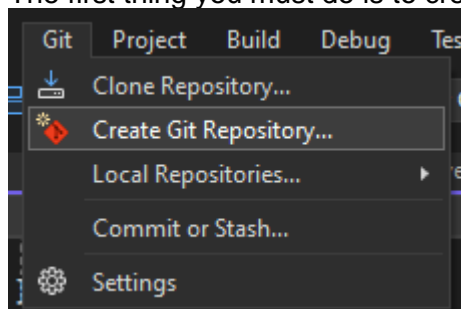
```
28 // Register services:
29 var services = builder.Services;
30 services.AddSingleton(TimeProvider.System);
31
32 services.AddTransient<IStockRepository, FakeStockRepository>();
33 services.AddTransient<IRecipeRepository, FakeRecipeRepository>();
34
35 services.AddTransient<IPizzaOven, NormalPizzaOven>();
36
37 services.AddTransient<IStockService, StockService>();
38 services.AddTransient<IRecipeService, RecipeService>();
39 services.AddTransient<IOrderingService, OrderingService>();
40 services.AddTransient<IMenuService, MenuService>();
41
42 var app = builder.Build();
```

**Note:** It is possible to register under different scopes. Here we will just use transient.

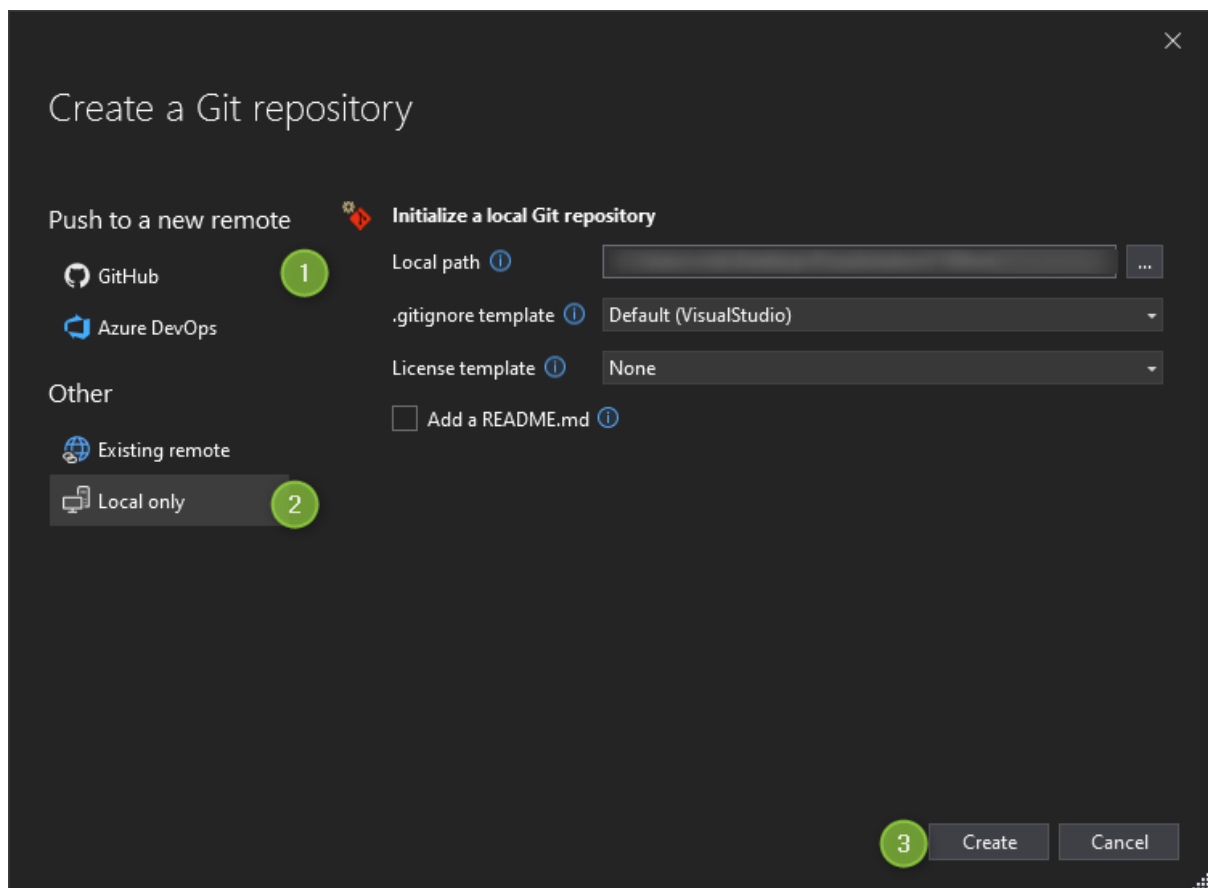
### 1.3 Git commit

The code will not be under version control, when you receive it. Even if this code is only for a training purpose and it will only be you doing modifications to it, using git and doing commits will be a good idea. This will also prepare you for building code in teams.

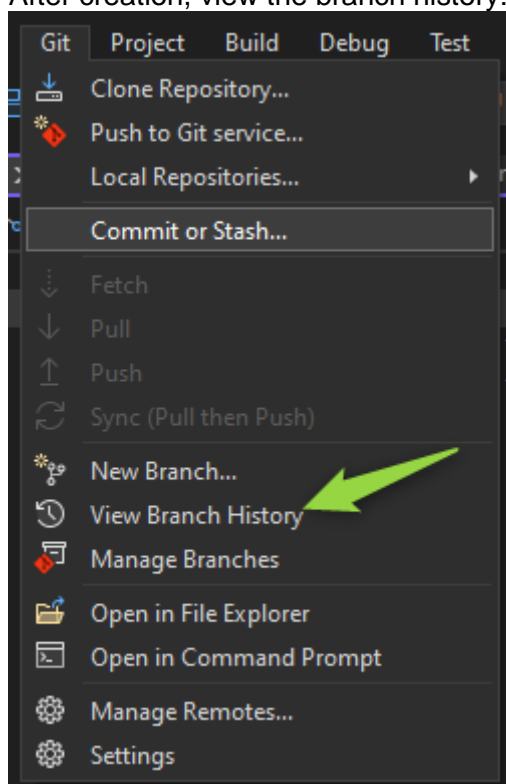
The first thing you must do is to create a Git Repository. Go to Git and Create Git Repository:






If you have a GitHub account feel free to add the repository as a private GitHub repository (1) otherwise a local (2) will be fine:



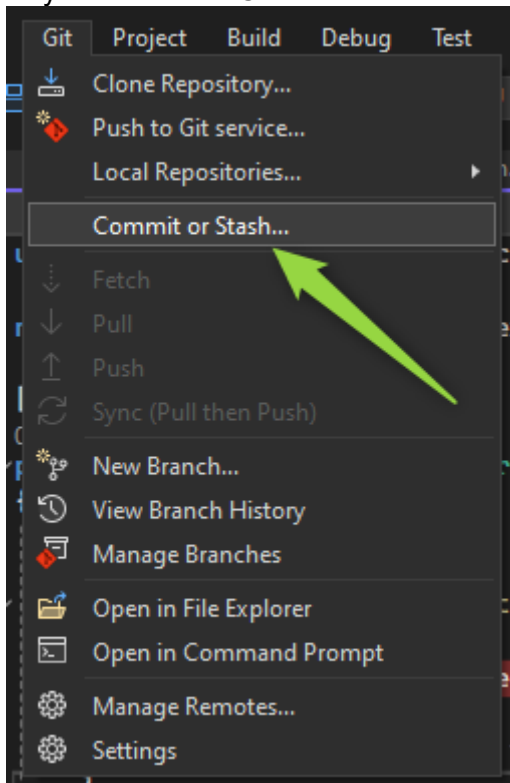
After creation, view the branch history:



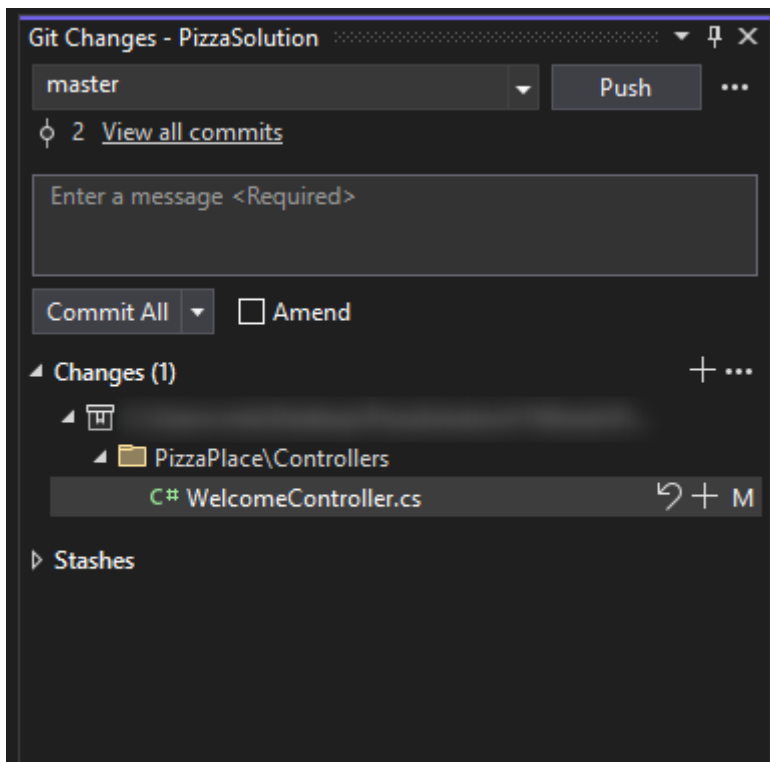
This should show you two initial commits:

Graph	ID	Author	Date	Message
Local History				
	84941a5e		17-03-2024 11:15:53	Add project files.
	2d80768e		17-03-2024 11:15:49	Add .gitattributes and .gitignore.

Say the WelcomeController has been changed, then open “Commit or Stash” in Git:



Then you get an overview of git changes since the last commit. You can see which files were changed. Double clicking on a file will show a comparison since the last commit. Enter a commit message and click “Commit All” when ready.



Do regular commits when changing code for specific tasks. This will make it apparent which code was changed when and for what reason. This also applies to **doing tasks set in this guide**.

When working on projects this helps narrow down possible sources, when bugs are found.

#### Task 1A

Modify the returned welcome message in the WelcomeController and commit the changes.

## 2. Creating a pizza menu

**Goal:** After this the application should have two menus returned depending on the time of the day, and you should have an overall understanding of JSON.

### 2.1 The menu model

The method return type of MenuService's GetMenu has already been given:

```

5  public class MenuService : IMenuService
6  {
7      public Menu GetMenu(DateTimeOffset menuDate)
8      {
9          throw new NotImplementedException("No menu has been implemented yet.");
10     }
11 }

```

The class (or more precise **record**) Menu, that is returned looks like:

```

1  namespace PizzaPlace.Models;
2
3  public record Menu(string Title, ComparableList<MenuItem> Items);
4

```

This actually means that Menu is an immutable object with two properties. One called "Title" and one (list) called "Items".

#### Task 2A

You have been tasked to implement GetMenu in MenuService and create some test menus. One standard menu and one menu for lunch. For testing purposes these will be hardcoded. They should be returned according to the input time.

You have full creative freedom to create these menus and add the necessary code or enum items to make it work. The Menu and MenuItem models are given and should not be changed. The only requirement is as follows:

The two menus must differ - if only in pricing and title. Both menus should have exactly 12 items. The lunch menu should be returned between 11AM and 2PM (UTC) - otherwise the standard menu should be returned.

### 2.2 JSON, serializing and deserializing

Run the application and open the Menu on the Swagger page. Try out the endpoint to get the menu you just created.

The endpoint should now give a 200 with a JSON response like this simplified response:



Request URL

`http://localhost:9009/api/menu`

Server response

Code	Details
200	<p>Response body</p> <pre>{   "title": "Test menu",   "items": [     {       "description": "Not a good item.",       "pizzaType": "StandardPizza",       "price": 10,       "soldOut": false     }   ] }</pre> <p>Response headers</p> <pre>content-type: application/json; charset=utf-8 date: Mon, 18 Mar 2024 14:18:43 GMT server: Kestrel transfer-encoding: chunked</pre>

The returned response body is the Menu object automatically serialized as JSON. If JSON notation is new to you, spend some time looking into the JSON notation.

Reversely it is possible to get a C# object from JSON by deserializing it to a known object. The ASP.NET Core controllers actually do this by default for us.

Go to the OrderingController and set a breakpoint at the return line:

```
PizzaPlace
1  using Microsoft.AspNetCore.Mvc;
2  using PizzaPlace.Models;
3  using PizzaPlace.Services;
4
5  namespace PizzaPlace.Controllers;
6
7  [Route("api/order")]
8  public class OrderingController(
9      IOrderingService orderingService) : ControllerBase
10  {
11      [HttpPost]
12      public async Task<ActionResult> PlacePizzaOrder([FromBody] PizzaOrder pizzaOrder)
13      {
14          return Ok(new
15          {
16              pizzas = await orderingService.HandlePizzaOrder(pizzaOrder),
17          });
18      }
19  }
20
```

The endpoint here is a Post endpoint, so it can have a body. As we can see in the method signature it expects a PizzaOrder from the body of the http request.

Go to the Swagger page for the OrderingController and click try it out. It will give a suggested JSON request body for the endpoint:



## Ordering

POST

/api/order

Cancel

Parameters

No parameters

Request body required

application/json

```
{  "requestedOrder": [    {      "pizzaType": "StandardPizza",      "amount": 0    }  ]}
```

### Task 2B

Write a request body with a minimum of three different pizza types. Send the request and look at the input object in the controller. Try with an invalid `PizzaRecipeType` and see how the input object then looks like.

**Note:** When the program has stopped at some line in debug mode it is possible to inspect the objects currently available. This can be done e.g. by hovering over them or finding the variable in Locals (or autos or watch):

The screenshot shows a C# controller method `PlacePizzaOrder` in `PizzaPlace.Controllers.OrderingController`. The method takes a `PizzaOrder` object as a parameter. The `Locals` window at the bottom displays the current state of the program:

Name	Value	Type
<code>this</code>	<code>{PizzaPlace.Controllers.OrderingController}</code>	<code>PizzaPlace.Controllers....</code>
<code>pizzaOrder</code>	<code>{PizzaOrder { RequestedOrder = [{"PizzaType": "StandardPizza", "Amount": 0}] }}</code>	<code>PizzaPlace.Models.Pizza...</code>
<code>orderingService</code>	<code>{PizzaPlace.Services.OrderingService}</code>	<code>PizzaPlace.Services.IOrd...</code>

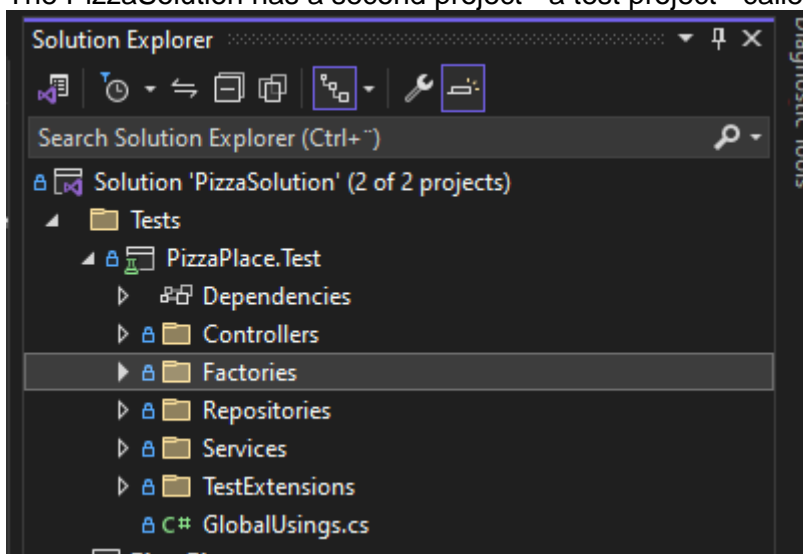
Two green arrows point from the `pizzaOrder` variable in the `Locals` window to the `pizzaOrder` parameter in the method signature and to the `HandlePizzaOrder` method call.

## 3. Creating and running tests

**Goal:** That you know how to write tests using the Arrange-Act-Assert pattern, and can run tests to check if the code does as intended.

### 3.1 Arrange-Act-Assert

The PizzaSolution has a second project - a test project - called PizzaPlace.Test:



The tests here use MSTest, though any testing framework could be used. Let's look at the RecipeServiceTests in Services:

```

8 [TestClass] 1
9 public class RecipeServiceTests
10 {
11     1 reference | 0/1 passing
12     private static RecipeService GetService(Mock<IRecipeRepository> recipeRepository) =>
13         new(recipeRepository.Object);
14
15     [TestMethod] 2
16     public async Task GetPizzaRecipes()
17     {
18         3 // Arrange
19         var order = new PizzaOrder([
20             new PizzaAmount(PizzaRecipeType.RarePizza, 1),
21             new PizzaAmount(PizzaRecipeType.OddPizza, 2),
22             new PizzaAmount(PizzaRecipeType.RarePizza, 20),
23         ]);
24         var rareRecipe = new PizzaRecipeDto(PizzaRecipeType.RarePizza, [new StockDto(StockType.UnicornDust, 1)], 1);
25         var oddRecipe = new PizzaRecipeDto(PizzaRecipeType.OddPizza, [new StockDto(StockType.Sulphur, 10)], 100);
26         ComparableList<PizzaRecipeDto> expected = [rareRecipe, oddRecipe];
27
28         var recipeRepository = new Mock<IRecipeRepository>(MockBehavior.Strict);
29         recipeRepository.Setup(x => x.GetRecipe(PizzaRecipeType.RarePizza))
30             .ReturnsAsync(rareRecipe);
31         recipeRepository.Setup(x => x.GetRecipe(PizzaRecipeType.OddPizza))
32             .ReturnsAsync(oddRecipe);
33
34         var service = GetService(recipeRepository);
35
36         4 // Act
37         var actual = await service.GetPizzaRecipes(order);
38
39         5 // Assert
40         Assert.AreEqual(expected, actual);
41         recipeRepository.VerifyAll();
42     }
43 }

```

The (public) class itself is given the [TestClass] attribute (1). Each test method (here only



one) is then given the [TestMethod] attribute (2).

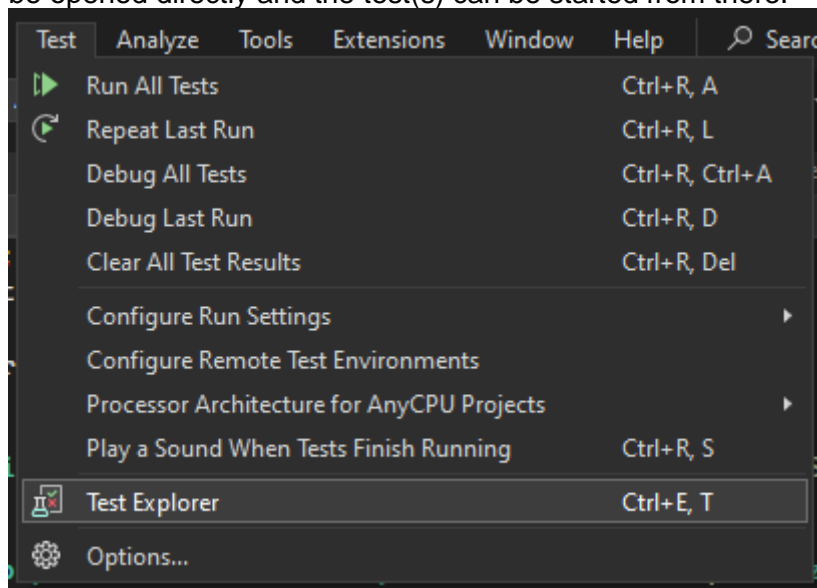
Each test should be set up with the same structure with Arrange (3), Act (4) and then Assert (5). Even when it seems superfluous, always **add the comment headline** indicating which phase of the test it is currently in. This will make it easier for you and others when first reading the test.

This specific test starts by arranging as follows: It sets up a PizzaOrder to be used as input. Then it defines two dummy recipes, and declares what the expected result should be. A mock is created for the only injected interface, an IRecipeRepository. It sets the dummy recipes to be returned when requesting for the respective PizzaRecipeTypes. The mock is strict, so if any calls not set up were called, it would throw an exception and the test would fail. Finally we create the service (sometimes called SUT - system under test) with the repository.

In the Act phase, an actual is declared as the result of the method we are testing. The act phase should always be as simple as possible - mostly with only a single call.

Finally we assert, that actual is indeed what we expected. We do a VerifyAll on the repository mock, so that we know that indeed all the mock setups were matched during the test.

To run the test, you can right click and select "Run Tests". Otherwise the Test Explorer can be opened directly and the test(s) can be started from there.



#### Task 3A

Run all tests in the test project. Locate any failing tests and evaluate whether to fix or remove them.

#### Task 3B

Write tests for your implementation of the MenuService. Have tests that get the lunch menu and the standard menu - hopefully there is an easy way to see the difference.

#### Task 3C

The StockService and the RestockingController have not been implemented. Add code that makes sense and write (passing) tests for them. (Or write the tests first.)

### Task 3D

Currently in the code, there is no way to add recipes - not even “standard” recipes. Create a relevant controller and whatever else you feel necessary to add and update recipes. Including tests.

## 4. Test Driven Development

**Goal:** Awareness about the concept of Test Driven Development.

### *4.1 Tests first, then implementation of the code*

In short, Test Driven Development is about writing tests first and then writing the code, so the tests will pass. For a more correct presentation of the subject please look online.

Task 4A

In `PizzaPlace.Test.Factories` are some (commented) tests for `AssemblyLinePizzaOven` and for `GiantRevolvingPizzaOven` - both of these factories inherit from `PizzaOven` but the necessary override method has not been implemented.

For each of these, uncomment the corresponding test class and then fix the implementation - hopefully correctly or at least so that the tests pass.

The requirements are:

**AssemblyLinePizzaOven** has a capacity of 1, when starting a new type of pizza it has an extra setup time of 7 minutes. Each subsequent pizza of the same type will however be produced 5 minutes faster than the previous - to a minimum of 4 minutes.

**GiantRevolvingPizzaOven** has a capacity of 120, however each pizza on the revolving oven surface must have the exact same cooking time - otherwise the spot should be left empty.

**Note:** to uncomment an entire file you can mark all (ctrl+A), then hold down ctrl and click 'K' then 'U'. To comment all the selected hold down ctrl and click 'K' then 'C'.

## 5. Add database

**Goal:** Get to know how to set up a database locally and connect to it.

### ***5.1 Adding persistent state in between program restarts***

Currently the program has only had the fake repositories. So everything changed would be wiped if the application was shut down and started again.

To let the application have a state you will need to connect it to a database.

Task 5A

Install a database locally and connect to it - this could e.g. be PostgreSQL. Look to the internet for guidance.

Implement the RecipeRepository and the StockRepository to use the database - remember to change the registered IRecipeRepository and IStockRepository.

## 6. Improve the setup

**Goal:** Learn to be critical about other people's code and see where to improve.

### ***6.1 Improve the placing a pizza order***

The implementation you got returned the pizzas, when receiving a pizza order in the ordering controller.

This might not be the best thing to do, because it could take a long time before the pizzas were ready.

Instead what might be reasonable to do is to return the cost of the order as well as how long it will take for the order to be produced - while still starting the order.

When the order has been completed, what we might want to do is to “call back” whoever ordered. Since we only have this test application simply logging to the console, that the pizzas are done would be okay.

Task 6A

Improve the program functionality to not let the person/system ordering be locked in the http call for several minutes. Feel free to add other improvements.

# 7. Develop

**Goal:** Be imaginative and develop.

## ***7.1 Requirements for the application***

The specific requirements given to you when working on a project might at the very best be scarce and vague. You should be able to come with suggestions of functionality and adapt when getting feedback.

### Task 7A

The “universe” of this application might be clear to you now: It is running an automated pizza place, taking orders and creating pizzas. What more would be nice? Maybe some nice nifty ovens with cool functionality? Maybe an overview of used stock and suggestion of what to restock to meet demand based on previous consumption? Maybe have the current stock be part of the menu to see whether some pizzas might be sold out?

Maybe you could figure out something better?

Go develop!