A white letter on a black background

Description automatically generatedA black and green line

Description automatically generated with medium confidence

Lab Workbook

**Advanced C#**

QA.COM

**The C# Programming Language**

# Lab 00: Introduction to Git and GitHub (OPTIONAL)

**Use these instructions to familiarize yourself with GitHub (if you have not used GitHub previously)**

**Objective**

Gain an understanding of how GitHub can be used to work on projects in a collaborative environment. You are encouraged to use GitHub as a repository for **ALL** the code you work on as part of this course. Using it means you are less likely to be impacted on a virtual machine timing out at some point. This is especially true of LOD machines and less so if you are using GoToMyPC. However, **ALL** VM's will be reset at the course's conclusion so **backing things up is a really good idea**.

**Requirements**

Create a GitHub account (if necessary) and then follow a the “hello world intro to GitHub” on the GitHub portal. Then use Visual Studio in conjunction with Github to manage versioning.

**Steps to Complete**

|  |  |
| --- | --- |
| **1** | **Create your GitHub account**  <https://docs.github.com/en/get-started/start-your-journey/creating-an-account-on-github> |
| **2** | **Create a repository and use it to merge changes into the main branch from a another**  <https://docs.github.com/en/get-started/start-your-journey/hello-world> |
| **3** | **Using GitHub with Visual Studio**   1. Using Visual Studio Open the Solution called StockValueCalculator.sln located in Labs\00 Introduction to Git and GitHub\Starter folder 2. Under the Git item on the main menu select Create Git Repository 3. Sign in to GitHub using the account you used earlier. |
| **4** | A screenshot of a computer  Description automatically generated  A screenshot of a computer  Description automatically generated  A cat logo with text  Description automatically generated |

|  |  |
| --- | --- |
| 5 | Add your initials to the end of the auto-generated repository name (“PC” in the example below), deselect the Private repository option and select the Add a README.md |
| 6 | A screenshot of a computer  Description automatically generated |
| 7 | Then click on the Create and Push button |
| 8 | Your project has now been uploaded to GitHub and you also have a local copy(clone) on the local file system. |
| 9 | A screenshot of a computer  Description automatically generated |
| * + 9   10 | Now make a branch to support updates to the current project. Click on the Git item in the main menu of Visual Studio and select the **New Branch** Item. Name the branch **Minor-Changes** and confirm that the **Checkout branch** option is selected |
| 11 | A screenshot of a computer  Description automatically generated |
| 12 | Click on create, and if prompted for further input, accept the defaults and continue. |
| 13 | The **Git Changes** window should show that there are currently no changes deployed to the branch named **Minor-Changes** |
| 14 | A screenshot of a computer screen  Description automatically generated |
| 15 | Open the Inventory.cs file and declare a private field of type string named S1 and initialise this to have a default value of “test” |
| 16 | A screen shot of a computer  Description automatically generated |
| 17 | Notice that the modified file has a red tick associated with it in the Solution Explorer window |
| 18 | A screenshot of a computer  Description automatically generated |
| 19 | Open the Git Changes tab and **double click** on the Inventory.cs file. The diff view appears highlighting the change (Additional line) added to the source code |
| 20 | Additionally, the status bar at displayed at the bottom of the Visual Studio window displays 1 change to the file associated with the branch named Minor-Changes |
| 21 | A screenshot of a computer  Description automatically generated |
| 22 | Now we will commit the changes to the GitHub Repo. In solution explorer right click on the Inventory.cs file select the Git menu item and then select the Commit or Stash item. Enter a description in the input text field, such as “string field added”. and click on the + Stage All button  A screenshot of a computer  Description automatically generated |
| 23 | The UI should now display as shown below. Click on the **Commit Staged** button. Then click on the **up arrow** to upload the changes to the Minor-Changes Branch  A screenshot of a computer  Description automatically generated |
| 24 | Check that the changes have been pushed to the repo by using a browser to navigate to <https://github.com/> and select the StockValueCalculatorXX repo. |
| 25 | Navigate to the Minor-Changes branch using the drop down menu  A screenshot of a computer  Description automatically generated |
| 26 | Open the StockValue calculator folder that contains the Inventory.cs file. You should see the modifications have been pushed to the branch named Minor-Changes. The master branch should still have the original version. We will now merge the changes recorded in the Minor-Changes branch with the master branch |
| 27 | In the GitHub Portal ensure the Minor-Changes branch is selected in the dropdown. Then from the Contribute menu click **Open Pull Request**  A screenshot of a computer  Description automatically generated |
| 28 | In the form that appears add a description e.g. Added String Field to Inventory class and then Click on the **Create Pull Request** button.  A screenshot of a computer program  Description automatically generated |
| 29 | On the next page click on the **Merge pull request** and then **Confirm merge** buttons in order to commit the changes with the master branch |
| 30 | Go back to Visual Studio select the **master** branch from the dropdown in the **Git Changes** tab and click on the **Sync (Push and Pull**) button  A screenshot of a computer  Description automatically generated |
| 31 | The graph displaying the updates to the master branch will now include the change you recorded the Minor-Changes branch.  A screenshot of a computer  Description automatically generated |
| 32 | Feel free to use GitHub when working on other exercises in the labs that follow this one. |

# Lab 01: SOLID Principles

## Objective

Your goal is to refactor code so that adheres to SOLID principles by implementing an inventory system for a bookstore.

## Overview

You are provided with a "Starter" program that manages an inventory system for a bookstore. The system allows adding books and CDs to the inventory and calculating the total stock value. The original code violates multiple SOLID principles. Your task is to refactor this code to make it more modular, maintainable, and extensible.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## Steps

Open the StockValueCalculator solution in Visual Studio. It contains two projects one called StockValueCalculator that contains functioning code that keeps track of a collection of products and their collective value and the other called StockValueCalculatorTests that hosts a single unit test that validates the functionality of the StockValueCalculator project.

Run the test to ensure the code functions correctly.

**Problems in the Starter Code:**

**Single Responsibility Principle**: The Product class is managing products without distinction between types.

**Open/Closed Principle**: Adding a new product type (like magazines or DVDs) would require modifying existing methods and possibly the Product class.

**Dependency Inversion Principle**: There is a direct dependency on low-level module details (like product type checks) in the inventory management.

**Requirements**

Refactor the Starter code so that it adheres to SOLID principles such that:

* Each product type (Book, CD) has its class that handles its specific attributes. (Single Responsibility Principle)
* The addition of new product types is straightforward and will not require changes to the existing code. (You can achieve this by creating a new class that implements an IProduct interface). (Open/Closed Principle)
* The Inventory class works with the IProduct interface, not concrete classes, which will decouple the code and make it more flexible. (Dependency Inversion Principle).

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 02: Coding Patterns

**Exercise: Implement a Vehicle Management System using Design Patterns**

## Objective

Your goal is to implement the functionality described below using the specified design patterns

## Overview

You are tasked with designing and implementing a vehicle management system in C#. This system will allow users to create, manage, and monitor different types of vehicles such as cars, lorries, and motorcycles. To ensure the application is scalable, maintainable, and well-organized, you should employ the following three design patterns: **Factory**, **Composite**, and **Observer**.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## Steps

In this lab you are provided with a starter project that contains a set of relevant code files each of which contain sets of comments that suggest what functionality the classes/interfaces should provide.

Open the starter project and implement the following requirements:

**Requirements**

1. **Vehicle Creation (Factory Pattern)**
   * Implement a **VehicleFactory** class that creates vehicles like cars, lorries, and motorcycles. This factory will facilitate object creation and can be extended in the future to include more vehicle types without modifying the client code.
   * Each vehicle should be derived from a common interface or abstract class, e.g., **IVehicle**.
2. **Managing Fleets of Vehicles (Composite Pattern)**
   * Use the Composite pattern to treat individual vehicles and groups of vehicles uniformly.
   * Implement a **VehicleGroup** class that can contain individual vehicles or other groups of vehicles. This class should also implement the **IVehicle** interface.
   * Include methods for adding and removing vehicles from groups.
3. **Monitoring Vehicle Status (Observer Pattern)**
   * Implement an Observer pattern where a **VehicleMonitor** class (which displays vehicle statuses) observes changes in the vehicles' properties or compositions (like adding or removing a vehicle in a **VehicleGroup** or starting or stopping a vehicle's engine).
   * Whenever a vehicle's status is updated, the monitor should automatically update to reflect changes.

**Steps to Complete**

1. **Define IVehicle Interface**
   * Define common operations like **DisplayStatus**, **StartEngine**, and **StopEngine**.
   * The interface should also define a property named Owner of type string
2. **Implement Concrete Vehicles**
   * Create classes like **Car**, **Lorry**, and **Motorcycle** that implement the **IVehicle** interface.
   * For each class add a constructor that has a single string parameter. Use the parameter value passed to initialise the Owner property
3. **Create Vehicle Factory**
   * Implement the **VehicleFactory** with methods to create different vehicles based on input parameters, such as **CreateVehicle("Car")**.
4. **Implement VehicleGroup**
   * This class should implement **IVehicle** and contain a list of **IVehicle** objects. It should delegate calls to its contained vehicles (e.g., it displays its status by asking each contained vehicle to display its own status).
   * In the VehicleGroup class add a constructor that accepts a string parameter, which is then used to initialise the Owner property.
5. **Implement VehicleMonitor and Observer Logic**
   * Create a **VehicleMonitor** class that is notified when any of its vehicles change (e.g. "Car engine started" or "Lorry engine stopped") or when the status of a vehicle within a group changes.
   * Implement an interface like **IVehicleChangedObserver** with a method **Update** that **VehicleMonitor** will implement. Vehicles will notify the **VehicleMonitor** through this interface when they change.
6. **Test Your Application**
   * Write a simple main program to demonstrate creating vehicles via the factory, adding them to the monitor, grouping vehicles using **VehicleGroup**, and starting and stopping vehicle engines to see the monitor update.
   * Alternatively, or if you have time, create a set of unit tests for your application.

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Enhancement (If you have time)

Try to enhance the Vehicle Management System by incorporating the Command Pattern. This pattern will provide a flexible and extendable way to encapsulate all details of operations performed on vehicles, such as starting or stopping engines, into command objects. This will also allow for easier tracking of operations (useful for undo/redo functionalities in more complex applications) and can organize the commands into a queue or a history log.

Steps you will need to take:

1. **Define a Command Interface.**
   * Give the interface a name of ICommand.
   * Get the interface to support Execute and Undo methods. Both methods should be void and take no parameters.
2. **Implement StartEngineCommand and StopEngineCommand classes**.
   * Create concrete command classes for starting and stopping the vehicle engines.
   * Make the classes implement ICommand.
   * Each class's constructor should be passed an IVehicle object that the Execute and Undo methods should use to invoke the StartEngine and StopEngine methods.
3. **Create a CommandInvoker class**.Implement an invoker class that can execute commands and optionally manage a history of commands for undo operations. It is suggested you do this by defining and instantiating a Stack<ICommand> collection within the class. Then implement the following methods:
   * **ExecuteCommand** thattakes an **ICommand** object as a parameter, invokes its **Execute** method and then pushes the command onto the Stack.
   * **UndoLastCommand** that takes an **ICommand** object as a parameter, checks to ensure the stack isn't empty and if not pops the last command off the stack and invokes its **Undo** method.
4. **Integrate the Commands into the Main Program.** Modify the main program to use commands for vehicle operations.
   * Declare and instantiate a **CommandInvoker** object.
   * Create some **StartEngineCommand** and **StopEngineCommand** objects passing appropriate parameters to the constructor.
   * Call the invoker object's **ExecuteCommand** method a number of times passing in the **ICommand** objects you've just created.
   * Call the invoker object's **UndoLastCommand** method.
   * Check the programs output to ensure the code is working as expected.

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

Lab 03: Asynchronous Programming and Concurrency

## Exercise: Word Prefixes

## Objective

The purpose of this exercise is to experiment with different scenarios mentioned in the asynchronous programming module.

## Overview

Word prefixes are also called stems. We have written a starter program, StemsLab, that contains a file (StemsOrig.cs) that reads the contents of a file that contains a large number of words and generates the most popular stems of 2 to n characters long. For example, the most common 2 letter stem is "co" (meaning that most words in the file start with these letters – there are 1793 of them!). The most common 3 letter stem is "con" (occurs 737 times) and 4 letter stem is "inte" (254 times).

The code uses a Timer class that calculates and prints how long a piece of code takes to run.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## Initial Steps

Open the StemsLab **starter** solution in Visual Studio. Build and run the program and note the time it takes to execute. You will note that no word exceeds 28 characters, so n could be 28. However, we can increase the value of n to obtain a longer runtime and demonstrate multiprocessing.

This program could complete more quickly by splitting the searches into separate tasks. Where each task works on a separate stem size (2 chars, 3 chars, up to 28 chars).

**Scenarios**:

1. ***n*** worker processes.

This is where we split the task such that each stem length search runs in its own child process.

1. 2 worker processes ***n***/2 stem sizes each.

This assumes 2 CPU cores. It will require two processes to be launched explicitly, and each to be given a range of stem lengths to handle.

**If you have time:**

1. 2 worker processes using a queue.

This assumes 2 CPU cores. As in b), but instead of passing a range, pass the stem lengths through a queue. Make sure you have a protocol for the worker processes to detect that the queue has finished. Note, you can't just use any old queue, you need one that is threadsafe such as a ConcurrentQueue located in System.Collections.Concurrent.

## Part A: Split the searching up into individual tasks one per stem size:

|  |  |
| --- | --- |
| 1 | Add a new class to the project called StemsA. |
| 2 | Copy the code in StemsOrig to the new class. |
| 3 | Add a new static function to the StemsA class called StemSearch. The function should take the stems dictionary and an integer that will specify the stem size being searched for (2 chars, 3 chars, etc.). The function should return a Tuple<int, string, int> where the first int will be the stem size, the second value (string) will be the bestStem and the final value (int) will be the bestCount (the number of stem occurrences in the data). Declare a variable of this type called *val* at the top of the function and set its value to null. |
| 4 | Cut the code that lies inside the StemsOrig's for loop and paste it into the StemSearch function (in StemsA) you just created. |
| 5 | Delete or comment out the Console.WriteLine statement that lies within the if expression (that tests to see if bestStem isn't empty) and add a line of code that sets the val variable to a new Tuple<int, string, int> populating it with the relevant values (stemSize, bestStem and bestCount). |
| 6 | Make the function return val. |

We next need to add code to the StemsA class's FindStems function that creates a set of Tasks that each point at the StemSearch function and coordinate their behaviours.

|  |  |
| --- | --- |
| 7 | Declare and instantiate a variable called tasks just before the for loop. Specify its type as List<Task<Tuple<int, string, int>>>. This Collection will hold all of the Tasks that will be generated in the loop (each Task will tackle a different stem length). |
| 8 | Declare and instantiate a variable called popularStems also just before the for loop. Specify its type as List<Tuple<int, string, int>>. This Collection will eventually hold all of the Tuples returned from the calls to the FindStems function. |
| 9 | Within the for loop declare a integer variable called size making it equal to the current value of stemSize. We're going to pass this (rather than stemSize) to the StemSearch function because by the time stemSearch functions get up and running there's a strong likelihood the stemSize variable (which is driving the For loop in the FindStems function) will have changed. |
| 10 | As the next line of code in the loop declare a Task<Tuple<int, string, int>> variable called task making it equal to Task.Run(() => StemSearch(stems, size)). This will grab a Thread from the ThreadPool and get it running the StemSearch function. |
| 11 | Add task to the tasks collection. |
| 12 | Beneath the for loop, add code that Waits until all the tasks have finished by writing:  Task.WhenAll(tasks).Wait(); |
| 14 | Next, add code that iterates around the tasks collection (tasks.ForEach(t => ...)) adding each task's Result to the popularStems collection. | |
| 15 | Finally, create a loop that iterates around the popularStems collection checking for non-null values before printing each Tuple's information (stem size, stem and number of occurrences. | |
| 16 | Edit the code in Program Main to call StemsA.FindStems(). | |
| 17 | Run the program and confirm it produces the same output as the original code. You should find the code runs quicker than before. This is because as they say "many hands make light work". | |

## Part B: Rework the logic so there are only 2 tasks which work through a range of stem sizes:

|  |  |
| --- | --- |
| 1 | Copy your solution to part A into a new class file called StemsB. |
| 2 | Edit the StemSearch method so it takes the stems dictionary and two integers one called start and the other called end. StemSearch should now have three parameters. |
| 3 | **Copy** the declaration of the popularStems variable from the FindStems method and add it to the top of the StemSearch method. |
| 4 | **Cut** the **declaration** of the for loop from the FindStems method and paste it immediately below the declaration of the popularStems variable but before the declaration of bestStem. Edit the declaration so the loop starts with a stemSize set to the start parameter and change the condition, so the loop runs while stemSize is less than the value of the end parameter. |
| 5 | Change the return type of the StemSearch method so it returns a **List**<Tuple<int, string, int>>. |
| 6 | At the foot of the method make it return popularStems (rather than var). |
| 7 | Delete the declaration of val (located towards the top of the method). |
| 8 | Move the variables bestStem and bestCount into the for loop (at the top of the loop) |
| 9 | Now move the **foreach loop** and the following **if** statement (inside the StemSearch function) into the same for loop so that they appear below the line that initialises the bestCount variable. |
| 10 | Locate the code inside the if (!string.IsNullOrEmpty(bestStem)) expression. Change it so it adds the new Tuple to the popularStems collection. |
| 11 | Delete the two lines in the StemSearch function that display compiler errors:  Task<Tuple<int, string, int>> task = Task.Run(() => StemSearch(stems, size));  tasks.Add(task); |
| 12 | Delete the variable named size. |

The StemsSearch function will now hunt for a range of stems specified by the values passed to the function's start and end parameters. The function returns a collection of popular stems.

|  |  |
| --- | --- |
| 13 | In the FindStems method just beneath the declaration of the integer variable named n, declare, and instantiate a variable of type Task<List<Tuple<int, string, int>>> naming the variable task1 and passing stems, 2, n/2 + 1 as parameters to the StemSearch method.  Task<List<Tuple<int, string, int>>> task1 = Task.Run(() =>  StemSearch(stems, 2,n/2 + 1)); |
| 14 | Copy the newly changed line that declares task1 and paste it immediately beneath it. Rename it as task2 and pass stems, **n / 2 + 1**, **n + 1** as the parameters to the StemSearch method. |
| 15 | Change the Task.WhenAll() to wait for both task1 and task2 to complete. |
| 16 | Add 2 lines that add the Results of each completed Task to the popularStems collection (by using its AddRange() method). |
| 17 | Edit the code in Program Main to call StemsB.FindStems(). |
| 18 | Build and run the program. The output should be the same as before, but the code will run more quickly than the original but more slowly than Part A. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

## If You Have Time:

## Part C– Rework the logic so there are 2 tasks that share a queue to work through a range of stem sizes:

You're on your own with this one. You need to use a Thread safe queue such as System.Collections.Concurrent.ConcurrentQueue. The queue will need to be filled with a set of stem sizes ranging from 1 to 30. You can use its Enqueue method to do this. ConcurrentQueues don't support a Dequeue method, but they do have a TryDequeue method that returns a boolean to indicate success or failure the actual queued value should be passed as an out parameter.

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 04: A Quick Tour Around ASP.NET Core Web API MVC

## Objective

In this exercise we give you a quick tour around the fundamentals of ASP.NET Core Web API MVC.

## Overview

You start out be creating a basic ASP.NET MVC API Core project and then explore how to go about adding a controller and giving it a set of Actions. You will explore how C# method overloading causes issues that can be overcome by adding routing via HttpGet attributes. You will test the applications by using the built in Swagger capabilities and also take a look at using Postman as an alternative.

This exercise will take around minutes.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

|  |  |
| --- | --- |
| 1 | There is no starter for this project. Instead, create a new Web Application:  We will create one that is very similar to the one which we will use in most of the labs.  In Visual Studio, Select Create a new project A screenshot of a computer program  Description automatically generated  Search for “Web Core” and select ASP.NET Core Web API Application  A screenshot of a computer  Description automatically generated  Name the project **QuickTour.** Leave other settings as is:A screenshot of a computer  Description automatically generated  Select Next  Ensure the setting are as specified in the next screenshot and press Create  A screenshot of a computer  Description automatically generated  Visual Studio creates a new MVC API project, based on the default project template,.  A screenshot of a computer  Description automatically generated |
| 2 | To make sure it’s a runnable website, press F5.  You may be asked to accept a certificate the very first time you run an MVC application in development mode. If so, you should accept the certificate. Then, you will see a Swagger page in a web browser:  A screenshot of a computer  Description automatically generated  Feel free to follow the prompts and invoke the WeatherForecast functionality. You should see something like the following.  A screenshot of a computer  Description automatically generated  The data highlighted in the red box (above) shows some randomly generated data that is supposed to forecast what the weather will be like over the next 5 days.  Close the browser. |
| 3 | Expand the Dependencies > Packages folder. Also, right-click the project > Edit Project File. Note that the (meta) packages listed here match those in the Packages folder  A computer screen with text  Description automatically generated |
| 4 | Rather than working with the existing "Weather" logic we will learn more about ASP.NET MVC API by adding additional code to the site. To get started we are going to need some data.  To flesh it out a bit we will imagine we’re building an on-line shop, so let’s use that as the topic.  Right-click on the QuickTour project in the Solution Explorer window and select Add | New Folder giving it the name Models. Add a C# class called Product.cs.  Populate it like this:   |  | | --- | | public class Product  {  public int ProductId { get; set; }  public string Name { get; set; }  } |   Note we have added an Id because we would intend to store this in a database eventually.  The recommendation is to use <className>Id as it fits in with EntityFramework (discussed later) conventions somewhat better than just ‘Id’ |
| 5 | Right-click on the Controllers folder and select Add | Controller… Then, in the Add New Scaffold Item dialog box select MVC Controller – Empty and click Add.    In the Add New Item dialog select the **API Controller – Empty** option.  Name the controller ‘ProductsController’ and press "Add". |
| 6 | You should now see an empty class that is decorated with two attributes [Route("api/[controller]")] and [ApiController].  The Route attribute is specifying a template that dictates the part of the URL that directs the request to the controller. The [controller] section tells the run-time to replace it with the name of the controller (in this case Products) and would be analogous to [Route("api/Products")]. The benefit coming should the developer ever change the controller class name. If this attribute is omitted, then routing is based on method level routing.  The [ApiController] attribute can be applied to controller classes to enable some API-specific behaviours such as making attribute routing a mandatory requirement (i.e. use of the Route attribute (see above).  Add a new method to the class called Products that returns an IEnumerable<Product>.  Decorate the method with an HttpGet attribute. Note this attribute isn't strictly necessary but Swagger uses it to determine how the method is to be used (Get, Post, Put, etc..) and will display an error if the attribute is missing.  Public methods in a controller are called Actions – this is the Products() Action.  Edit the method so it generates a number of Product objects adding them to a List. Then get the method to return the list:   |  | | --- | | [ApiController]  [Route("api/[controller]")]  public class ProductsController : ControllerBase  {  [HttpGet]  public IEnumerable<Product> Products()  {  List<Product> products = new List<Product>();  products.Add(new Product { ProductId = 1, Name = "Rolos" });  products.Add(new Product { ProductId = 2, Name = "Bag of Crisps" });  products.Add(new Product { ProductId = 3, Name = "Apple" });  products.Add(new Product { ProductId = 4, Name = "Cheese Sandwich" });  return products;  }  } | |
| 7 | Press F5 to launch the service. Ensure the Swagger page appears  A screenshot of a computer  Description automatically generated  Test drive the Get /api/Products option by pressing the drop-down arrow and clicking the Try it out button and then press execute. You should see the following:  A screenshot of a computer  Description automatically generated |
| 8 | Let’s add a second end-point (Action) to the controller. To keep things brief we'll get it to do the same thing as the Products method but return the list in alphabetical order of product name.  Add the following code to the Products controller:   |  | | --- | | [HttpGet]  public IEnumerable<Product> OrderedProducts()  {  List<Product> products = new List<Product>();  products.Add(new Product { ProductId = 1, Name = "Rolos" });  products.Add(new Product { ProductId = 2, Name = "Bag of Crisps" });  products.Add(new Product { ProductId = 3, Name = "Apple" });  products.Add(new Product { ProductId = 4, Name = "Cheese Sandwich" });  return products.OrderBy(p => p.Name).ToList();  } | |
| 9 | F5 and prepare to be disappointed. Swagger will be unhappy:  A screenshot of a computer error  Description automatically generated |
| 10 | The reason for the error is a little strange. If you dig into Visual Studio's Output window you will see the following error:  A screenshot of a computer  Description automatically generated  It's complaining about a conflicting method/path for QuickTour.Controllers.ProductsController.Products and QuickTour.Controllers.ProductsController.OrderedProducts  And further states Actions require a unique method/path combination for Swagger. You'd be forgiven for thinking that the two methods do have unique method/path combinations given their different names. However, it's not just Swagger that's complaining. If you were to call the method from an external client as a real API call, you'd get a similar error. Weirdly, in spite of the different method names, the runtime is upset because the signatures of the two methods are identical (i.e. neither take any parameters) and are therefore deemed to be the same! |
| 11 | The workaround is to extend the HttpGet attributes to specify an extension to the controller's template ("api/[Controller]"):  Note, it is perfectly OK to give the template the same value as the Action name. In the above example the two URL's needed to invoke the methods are:  [HttpGet("ProductsDetail")]  public IEnumerable<Product> ProductsDetail()  {  List<Product> products = new List<Product>();  products.Add(new Product { ProductId = 1, Name = "Rolos" });  products.Add(new Product { ProductId = 2, Name = "Bag of Crisps" });  products.Add(new Product { ProductId = 3, Name = "Apple" });  products.Add(new Product { ProductId = 4, Name = "Cheese Sandwich" });  return products;  }  [HttpGet("OrderedProducts")]  public IEnumerable<Product> OrderedProducts()  {  List<Product> products = new List<Product>();  products.Add(new Product { ProductId = 1, Name = "Rolos" });  products.Add(new Product { ProductId = 2, Name = "Bag of Crisps" });  products.Add(new Product { ProductId = 3, Name = "Apple" });  products.Add(new Product { ProductId = 4, Name = "Cheese Sandwich" });  return products.OrderBy(p => p.Name).ToList();  }  <https://localhost:7147/api/Products/ProductsDetail>  [https://localhost:7147/api/Products/OrderedProducts](https://localhost:7147/api/Products/OrderedProductsDetail)  It would be perfectly OK to alter the attributes to the following:  [HttpGet("Products")]...  [HttpGet("Ordered")]...  And then the respective URL's would be:  [https://localhost:7147/api/Products/Products](https://localhost:7147/api/Products/ProductsDetail)  [https://localhost:7147/api/Products/Ordered](https://localhost:7147/api/Products/OrderedProductsDetail)  Test drive the app with both of the suggested changes and ensure everything works as expected. |
| 12 | Let's now add an additional Action that takes a parameter.  [HttpGet("Products/{id}")]  public Product ProductsDetail(int id)  {  Product p = new Product { ProductId = id, Name = "Rolos" };  return p;  } |
| 13 | Notice we now have two overloaded methods (same name different signature which means the C# compiler will be happy and the new Action's HttpGet attribute starts off the same as its sister but is extended to include "/{id}" . The Squiggly braces indicate the URL will have a piece of data at its end (e.g. api/Products/Products/**12**) which will be passed on to the int id parameter specified in the method signature. |
| 14 | Run the code.  When testing in Swagger you will be given the opportunity to enter a value for the id into a Parameters text box:  A screenshot of a computer  Description automatically generated  Obviously, the code, as it stands, will ignore the value and always return Rolos. |
| 15 | It is absolutely fine to decorate an Action with more than one route (HttpGet attribute]. Try decorating the appropriate Actions with the following additional routes. Think carefully which method should get which attribute and what the corresponding URLs would look like:  [HttpGet("")]  [HttpGet("{id}")]  [HttpGet("ProductDetail/{id}")] |
| 16 | What do you think would happen if we removed the controller level [Route("api/[controller]")] attribute?  Think about it, make a prediction and then launch the app to see if you were right. |
| 17 | Add another Action with the same ProductsDetail name that takes the name of a product as a string parameter and returns an associated Product (again fake the search in the same way we did when passing the id).  Decorate the new Action with the same HttpGet attributes as with the id approach but replace "id" with "name" wherever it occurs. |
| 18 | The C# compiler should be happy because whilst there are now three methods that each have the same name it can distinguish between them because of the parameter types (no parameters, int and string).  Launch the app and use Swagger to test the new Actions…  Unfortunately, the method calls (both the ones that use the new string parameter and also the ones that used to work that used an int parameter) fail with the following message:  Microsoft.AspNetCore.Routing.Matching.AmbiguousMatchException: The request matched multiple endpoints.  Even though we've satisfied the C# compiler the ASP.NET Route selector logic can't distinguish between the two types of parameter, because they both appear to be strings! |
| 19 | The solution to the problem is to spell out the parameter type inside the HttpGet routing template as follows:  [HttpGet("ProductDetail/{**id:int**}")]  [HttpGet("Products/{**id:int**}")]  [HttpGet("{**id:int**}")]  public Product ProductsDetail(int id)  {  Product p = new Product { ProductId = id, Name = "Rolos" };  return p;  }  There's no need to do the same for the "name" parameters because the default is to treat them as strings. |
| 20 | An alternative to Swagger.  Swagger is OK as an an-hoc way of testing your Actions but if you want a set of slightly more permanent tests that save you from having to continually type the same things into the various Swagger text boxes you should consider using a tool like Postman. |
| 21 | Launch the app in Visual Studio.  Start Postman form the Windows Start menu.  You should see a window for an untitled GET request that is prompting for a URL.  A screenshot of a computer  Description automatically generated  Enter the following as the URL:  <https://localhost:7147/api/Products/Products/4>  Press the blue Send button  Note you may receive an "SSL error: unable to verify the first certificate" warning message. It's OK to take the suggested option of disabling SSL verification.  Look at the response in the bottom half of the screen and confirm it is what you expect.  A screenshot of a computer  Description automatically generated  To create a new request, you can press the plus "+" button towards the top centre of the screen. You will also be able to create Post, Put and Delete requests by dropping the down arrow that is located to the right of the word GET (and to the left of the test URL.  Note, if you are prepared to register a set of credentials with Postman (it's free to do this) then you will be able to permanently save your requests (by pressing the Save button to the upper right of the screen).  It is definitely worth getting familiar with a tool like Postman it could transform your life! |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# If You Have Time

|  |  |
| --- | --- |
| 1 | Use Postman to create a set of test calls that exercise all the possible URLs that your Quick Tour project supports. |
| 2 | Add additional Actions that take multiple parameters |
| 3 | Rework the code in Actions that return a single Product so that it picks a matching value from a collection of Products |
| 4 | Refactor the code so it contains no duplicate logic |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 05: Dependency Injection, Scope and Configuration

## Objective

The goal of this lab is to investigate how Dependency Injection and its configuration is done in ASP.NET applications

## Overview

In this exercise you get to do Dependency Injection (DI) and learn how it is configured. You will also see the effect of the different DI scopes.

This exercise will take around minutes.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## Steps:

## Dependency Injection and Scope

|  |  |
| --- | --- |
| 1 | The starter application is the “Quick Tour” that we did previously, with just a bit of extra stuff added (commented out for now, but we will un-comment it later).  Open the ‘Begin’ folder and compile the application. Check that it runs. |
| 2 | We left the ProductsController in the Quick Tour app instantiating its own data, which, to be frank, isn't a very clever thing to be doing. We will fix this by creating a new class in the Models folder which represents a fake database. (Of course, we will explore real database connections a little later in the course.) Call the class MockProductsContext. Add this method to the new class:   |  | | --- | | public class MockProductsContext  {  public IEnumerable<Product> GetProducts()  {  return new List<Product>() {  new Product { ProductId = 1, Name = "Rolos" },  new Product { ProductId = 2, Name = "Bag of Crisps" },  new Product { ProductId = 3, Name = "Apple" },  new Product { ProductId = 4, Name = "Cheese Sandwich" },  new Product { ProductId = 5, Name = "Can of Coke" }  };  }  } |   Modify the ProductsController class to use this new method:   |  | | --- | | public class ProductsController : ControllerBase  {  IEnumerable<Product> \_products = null;  public ProductsController()  {  MockProductsContext mockContext = new MockProductsContext();  \_products = mockContext.GetProducts();  }  ... | |
| 3 | At the moment, the ProductsController is making its own MockProductsContext object.  We should always be wary of a class creating service classes itself. Let’s change it to use constructor injection, using the built-in dependency injection framework.  Dependency injection works better if it’s based on interfaces, so add the following interface to the Models folder:   |  | | --- | | public interface IProductsContext  {  public IEnumerable<Product> GetProducts();  } |   And modify the mock Product context to implement this interface:   |  | | --- | | public class MockProductsContext : IProductsContext  {  ... | |
| 4 | In the ProductsController, rework the code at the top of the file to read as follows:   |  | | --- | | public class ProductsController : ControllerBase  {  IProductsContext \_context;  public ProductsController(IProductsContext context)  {  \_context = context;  }  ... |   You can see the constructor is expecting an object that implements IProductsContext to be passed (injected) to it as a parameter. |
| 5 | Modify **all­** of the ProductsDetail and OrderedProducts actions to use the injected database context:   |  | | --- | | ...  public IEnumerable<Product> ProductsDetail()  {  return \_context.GetProducts();  }  ... | |
| 6 | In the Program.cs file, add the following line that declares the builder object just before the line that declares the app object. The aim is to identify all the services which can be provided by dependency injection and map the interface that the class requires to the concrete class that we are going to supply.   |  | | --- | | builder.Services.AddScoped<IProductsContext, MockProductsContext>(); |   Run the application and check that it still works. Using dependency injection is as easy as that! |
| 7 | In the next part of the lab, we are going to investigate the different lifetimes that are available to us when we use dependency injection.  Drag the file Dependencies.cs from the Assets folder (in Explorer) onto the Models folder in your project. The contents are 3 classes following this pattern:  A close-up of a white background  Description automatically generated  Which we will inject with the corresponding scope.  Have a look at the TransientDependency – you will see it writes out to the logger each time a new instance is instantiated.  Whenever the method WriteGuidToConsole() is called, it will show the unique Guid and the thread on which it is running. |
| 8 | In the program.cs file just beneath the Service addition of the IProductsContext, register the class TransientDependency as the desired implementation of the interface ITransient and give it Transient scope.  Do the same for Scoped and Singleton – like this:  builder.Services.AddTransient<ITransient, TransientDependency>();  builder.Services.AddScoped<IScoped, ScopedDependency>();  builder.Services.AddSingleton<ISingleton, SingletonDependency>(); |
| 9 | Modify the ProductsController to require these dependencies, and also to add a bit more debug information:   |  | | --- | | private readonly ITransient \_tran;  private readonly IScoped \_scoped;  private readonly ISingleton \_single;  private readonly ILogger<ProductsController> \_logger;  private IProductsContext \_context;  public ProductsController(ILogger<ProductsController> logger,  IProductsContext context,  ITransient tran,  IScoped scoped,  ISingleton single)  {  \_logger = logger;  \_context = context;  \_tran = tran;  \_scoped = scoped;  \_single = single;  }    [HttpGet("")]  [HttpGet("Products")]  [HttpGet("ProductDetails")]  public IEnumerable<Product> ProductsDetail()  {  \_logger.LogInformation("In the HttpGet Products Index() method <=======");  \_tran.WriteGuidToConsole();  \_scoped.WriteGuidToConsole();  \_single.WriteGuidToConsole();  \_logger.LogDebug("About to get the data");  IEnumerable<Product> products = \_context.GetProducts();  \_logger.LogDebug($"Number of Products: {products.Count()}");  return products();  } | |
| 10 | Run the application from the console.  (A reminder of how to do this: right-click on the project, select “Open folder in file explorer”. Then, in file explorer, click into the address bar and type “cmd”. From the command window, type “dotnet run”.)  Open a web browser, go to <https://localhost:xxxx> (replacing xxxx with the port the app runs on as specified in the command window) . Once the page has been displayed, press the refresh button to display it a second time.  The console output is shown below, with annotations which explain what has happened:  A screenshot of a computer program  Description automatically generated  Singleton keeps same GUID, despite different thread  New transient & scoped – but not singleton  Each has their own unique GUID  Create the three dependencies  Refresh page |
| 11 | Take a closer look at the three dependency classes. Notice that each of them requires a constructor parameter – a logger. Where did the logger come from?  When dependency injection is used to create the dependency object (or, indeed, any object), it will check whether that new object has any of its own dependencies and will create those too! Dependency injection is used to create (for example) a ScopedDependency object, and as part of that process it also creates an ILogger<IScoped> that the ScopedDependency needs!  This enables a complex series of dependencies to be built up very simply. As you write each class, you need to know what that class depends on, but you don’t need to worry about any dependencies any deeper into the chain, because the dependency injection framework takes care of that for you. |
| 12 | In the screenshot above, the scoped dependency and the transient dependency appear to behave the same way – we get a new instance of the dependency each time we refresh the page.  Let’s modify our demonstration to show where these two types of lifecycle differ from each other.  Add an instance of each dependency to the MockProductContext class and use constructor injection to get instances of those dependencies. Then, call the WriteGuidToConsole() method on each dependency when creating the data:   |  | | --- | | public class MockProductsContext : IProductsContext  {  private readonly ITransient \_tran;  private readonly IScoped \_scoped;  private readonly ISingleton \_single;  public MockProductsContext(ITransient tran,  IScoped scoped,  ISingleton single)  {  \_tran = tran;  \_scoped = scoped;  \_single = single;  }  public IEnumerable<Product> GetProducts()  {  \_tran.WriteGuidToConsole();  \_scoped.WriteGuidToConsole();  \_single.WriteGuidToConsole();  ... |   Since we already use dependency injection to create the MockProductContext, and the dependencies we need are already registered, we don’t need to do anything else. |
| 13 | Run the application from the console, and once it’s running, visit the web site *only once*. The console output now looks like this:  A screenshot of a computer program  Description automatically generated  Scope dependency lasts for whole request, keeps same GUID.  Transient constructor called twice – once for controller, once for context  Transient constructor called twice – once for controller, once for context  Here, you can see that the scoped dependency is shared between the two classes that use it, whereas the transient dependency is not (and therefore the dependency injection framework needs to create two separate instances of the transient dependency.) |
| 14 | Add the following code to the ProductsController. It defines a method called Rare.  [HttpGet("Rare")]  public string Rare([FromServices]IActionInjection ai)  {  ai.WriteGuidToConsole();  return "That's all from rare this time!";  }  The idea is that this is a rarely used method, the resource it uses is expensive, so we don’t want to create it every time – just when this rare method is called.  You will find the interface and implementing class in Models/Depdendencies.cs |
| 15 | Register in Program.cs:  builder.Services.AddTransient<IActionInjection, ActionInjectionDependency>(); |
| 16 | Run again from the command line.    Do a few browser refreshes and note that the ActionInjection object is not created.    Now append ‘/Rare’ to the Url and note that the action dependency is now created.    (We’ve used the LogWarning method here, in contrast to LogInformation in other places, so you can clearly see the difference by the different colour.) |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# If you have time

## Configuring the Pipeline

|  |  |
| --- | --- |
| 17 | Add a trivial module into the pipeline – copy the folder called Middleware from the Assets folder and add it to the project. Have a look at the code the classes contain.  Inject this module into the pipeline by adding it in Program.cs as shown:  app.UseAuthorization();  app.UseMiddleware<CustomMiddleware1>();  app.UseMiddleware<CustomMiddleware2>(); |
| 18 | Suppress most of the trace information – in appsettings.Development.json   |  | | --- | | "Logging": {  "LogLevel": {  "Default": "None",  "Microsoft": "None",  "QuickTour": "None",  "QuickTour.Controllers": "Information",  "QuickTour.Middleware": "Debug"  } | |
| 19 | Start using dotnet run and open a browser at port whatever port the app is running on.  Refresh the browser a couple of times. Again, we’ve used LogWarning() to make the middleware messages stand out. Note that the middleware objects are created once at application startup, and then invoked for every web request.  Adding middleware is the process by which a standard MVC application is configured (for example, adding authentication and authorisation modules to the pipeline),so it’s definitely worth understanding what we mean when we talk about middleware, although writing your own middleware is not something you will need to do too often.  Each middleware component can check details of the request, and either return a response to the web browser, or pass the request on to the next middleware component, modifying either the request or the response as appropriate. Our very basic example simply passes the request along to the next component without altering it in any way. |

|  |  |
| --- | --- |
| Configuration – The Options Pattern | |
| 20 | Drag the folder Configuration from the Assets folder onto your project. This contains a class with 2 configuration options  A screenshot of a computer  Description automatically generated |
| 20 | Add this section to the end of appsettings.json, just above the final }   |  | | --- | | "Features": {  "EnableMyOption1": true,  "EnableMyOption2": false  } |   Visual Studio will automatically add the trailing comma to the preceding entry |
| 21 | Now bind the json section to the strongly-typed FeaturesConfiguration in Program.cs   |  | | --- | | builder.Services.Configure<FeaturesConfiguration>(builder.Configuration.GetSection("Features")); | |
| 22 | Go to ProductsController.  Modify the constructor:   |  | | --- | | private readonly FeaturesConfiguration \_features;  private reaconly ILogger<ProductsController> \_ logger;  public ProductsController(..., IOptions<FeaturesConfiguration> features, ILogger<ProductsController> logger)  {  ...  \_features = features.Value;  \_logger = logger;  } |   Note: we are giving the controller logging capabilities via dependency injection. In ASP.NET Core, the logging functionality is built-in and available by default because it is part of the framework’s dependency injection (DI) system. The logging services are automatically registered and configured when you create a new ASP.NET Core application so there is no need to add any special instructions to the Program.cs file. |
| 23 | Add this line to beginning of the the basic ProductsDetail() method that returns the original list of Products.   |  | | --- | | \_logger.LogInformation($"MyOption1 = {\_features.EnableMyOption1}, MyOption2 = {\_features.EnableMyOption2}"); | |
| 24 | Dotnet run and launch a browser. Note that the appsettings.json settings have been set into a FeaturesConfiguration object:  A black background with white text  Description automatically generated  In appsettings.json, set MyOption2 to be true and save the file (without re-compiling). Refresh the browser and note that the new value has not been read in.  Close and restart the app: it is only read in on app startup. |
| 25 | Make these changes to ProductsController   |  | | --- | | private readonly IConfiguration \_config;  public ProductsController(... IConfiguration config)  { .  .  .  \_config = config;  } |   And add this line to the ProductsDetail method just after your current features output :   |  | | --- | | \_logger.LogInformation($"MyOption1 = {\_config["Features:EnableMyOption1"]}, MyOption2 = {\_config["Features:EnableMyOption2"]}"); |   So now we have a type-safe way of reading configuration data (IOptions) and a type-unsafe way (config[“key”]). |
| 26 | Refresh the page and check that the type-unsafe options are the same as the type-safe options. |
| 27 | In appsettings.json, set MyOption2 to be false and save the file (without re-compiling). Refresh the browser and note that the new value has been picked up by the new config["key"] approach. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

|  |  |
| --- | --- |
| If you still have timeReading environment-specific variants of appsettings.json | |
| 28 | Add this to appsettings.json, just before the final }   |  | | --- | | "Message": "Hello from appsettings.json" | |
| 29 | Add this to appsettings.Development.json, just before the final }  (Note that you may need to click the arrow next to appsettings.json to see the Development file)   |  | | --- | | "Message": "Hello from appsettings.Development.json" | |
| 30 | By right clicking on the project in Solution explorer, add a new app settings file called appsetting.Staging.json to the project. Copy the contents of appsettings.Development.json to it and alter the "Message" entry to   |  | | --- | | "Message": "Hello from appsettings.Staging.json" | |
| 30 | Add to Program.cs a line that declares a Microsoft.Extensions.Configuration.ConfigurationManager callled config and make it equal to builder.Configuration:   |  | | --- | | Microsoft.Extensions.Configuration.ConfigurationManager config = builder.Configuration; |   In Program.cs insert the following code just after the app.UseMiddleware lines you added earlier   |  | | --- | | Console.ForegroundColor = ConsoleColor.Magenta;  Console.WriteLine(config["Message"]);  Console.ForegroundColor = ConsoleColor.White; | |
| 29 | Open the Project > Properties and go to the Debug tab and select the "Open debug launch profiles UI" link.  Ensure the Environment variables for http, https and IIS Express each have the following entry: ASPNETCORE\_ENVIRONMENT=Development:  A screenshot of a computer  Description automatically generated  A screenshot of a computer  Description automatically generated  A screenshot of a computer  Description automatically generated |
| 30 | Note you can select a number of different ways of communicating with the browser from Visual Studio's drop down start debug menu:  A screenshot of a computer  Description automatically generated  Launch the app in debug mode using any of the three options and note the cyan messagewritten to the console is from appsettings.Development.json i.e. "Hello from appsettings.Development.json" |
| 31 | Close the app, reopen the "Open debug launch profile UI" window and delete ‘Development’ for each of the three protocols:  A screenshot of a computer  Description automatically generated  A screenshot of a computer  Description automatically generated  A screenshot of a computer  Description automatically generated |
| 32 | Run the app in debug mode again. You may need to tweak the URL of the browser to make things work. Note, the console is now showing the message defined in appsetting.json i.e. " Hello from appsettings.json" |
| 33 | Edit in ‘Staging’ into the ASPNETCORE\_ENVIRONMENT for all three launch profiles:  A screenshot of a computer  Description automatically generated |
| 34 | Ctrl+F5 and you will see it now reads the Staging file |
| 35 | Lastly, just to confirm what has happened.  When you set ‘Staging’ in the environment, it modified the file ‘launchsettings.json’ (expand under Properties and you’ll see it).  "profiles": {  "http": {  "commandName": "Project",  "launchBrowser": true,  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Staging"  },  "dotnetRunMessages": true,  "applicationUrl": "http://localhost:5126"  },  "https": {  "commandName": "Project",  "launchBrowser": true,  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Staging"  },  "dotnetRunMessages": true,  "applicationUrl": "https://localhost:7236;http://localhost:5126"  },  "IIS Express": {  "commandName": "IISExpress",  "launchBrowser": true,  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Staging"  }  }  }…  Depending on the chosen launch protocol, the project that gets run is the first one in the list with |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 06a: Using the Entity Framework

## Part A – Code First (OPTIONAL)

## Objective

To become acquainted with the Microsoft Entity Framework and make use of its Code First capabilities.

## Overview

Entity Framework is an essential part of most .NET /CORE applications and is certainly used extensively in this course. We have this (optional) exercise to refresh/familiarise you with Entity Framework.

This exercise will take around 30 minutes.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## Steps

## The ‘Begin’ Solution

|  |  |
| --- | --- |
| 1 | Open the 'Begin' solution in Visual Studio and compile (Shift Ctrl+B).  Have a look at the code and note:   * Program/Main(): Currently the program does not touch a database * DataClasses: We have placed the data classes all in the 1 file. This is purely so you can see all the components. Operationally stick to 1 class in 1 file   In order to focus on only the Entity Framework elements, the exercise is presented in a Console Application, but we will use Dependency Injection.  There is nothing in the ‘Begin’ that wouldn’t have been there even if you had no intention of storing this in a database.  There are quite a few commented-out items  Run it (Ctrl-F5) – you should get this:  A black screen with white text  Description automatically generated  We now proceed to add what you need to implement Entity Framework, starting with the code (ie Code First). Later, we will do this from an existing database (aka CodeFirstFromDatabase) |
| 2 | In Visual Studio, select View/Task List. This will show a window where you can see all the To Do comments that we’ve added to the code.  Uncomment TODO 1, launch the app and confirm that the backpointer (animal to zoo) is null so you get a null reference exception.  Re-instate the comment-out |

|  |  |  |
| --- | --- | --- |
| NuGet | | |
| 3 | Read TODO 2 and Nuget the packages listed. | |
| 4 | We will need a context to describe which tables we wish to have in the database.  Go to TODO 3 and uncomment the zoocontext class (but OnConfiguring still commented out. You will need to add a using clause for Microsoft.EntityFrameworkCore to the top of the file listing. | |
| 5 | *Everything* in EF is a property – we’ve done this already.  However, because we are going to store Zoo and Animal objects in a relational database, they must have an Id. Go to TODO 4 and uncomment these. | |
| 6 | Comment out the foreach loop in Program/Main() and store it temporarily in Notepad | |
| 7 | Delete the entire contents of Program/Main() and replace with   |  | | --- | | static void Main(string[] args)  {  ConfigureServices();  Configure();  } |   Hopefully these names should look familiar to you... Dependency Injection! | |
| 8 | Right-click ConfigureServices() > Quick Actions and refactoring > Generate Method.  Repeat for Configure() | |
| 9 | Paste your commented-out foreach loop (in Notepad) into Configure(), in place of the “throw new NotImplementedException” line. For now leave it commented out | |
| 10 | Add in the following code – this is the Console equivalent of what ASPNET Core largely does for you :   |  | | --- | | class Program  {  public static ServiceCollection services;  public static ServiceProvider serviceProvider;  static void Main(string[] args)  {  ConfigureServices();  Configure();  }  private static void ConfigureServices()  {  services = new ServiceCollection();  string conn = @"Server=.\SqlExpress;Encrypt=False;Database=EFRefresh;Trusted\_Connection=true;  MultipleActiveResultSets=True";  services.AddDbContext<ZooContext>(options => options.UseSqlServer(conn));  serviceProvider = services.BuildServiceProvider();  } |   NB – in the above we have had to split the connection string across 2 lines to fit into the Word document. The Connection string **must** be all on 1 line, not just in this lab but in any other labs that follow.  **ALSO: If you are using the developer version of SQL Server then replace the .\SQLExpress in the connection string with (local)**. | |
| 11 | In Configure() we use the injected ZooContext. Add this line:   |  | | --- | | private static void Configure()  {  ZooContext ctx = serviceProvider.GetService<ZooContext>();  //foreach (Zoo zoo in zoos)  //{  // Console.WriteLine($"\nName = {zoo.Name}, number of animals = {zoo.Animals.Count()}");  // foreach (Animal animal in zoo.Animals)  // {  // Console.WriteLine($"...{animal.Type,-10}{animal.Name}");  // // TODO 1 Console.WriteLine($"......in zoo {animal.Zoo.Name}");  // }  //}  } | | |
| 12 | Uncomment the foreach loop and make this 1 change:   |  | | --- | | foreach (Zoo zoo in ctx.Zoos) | | |
| 13 | If we ran it now, it would fail because there is no database (if you’re familiar with the older Entity Framework 6, this behaves differently – it would create the database automatically). In a Console App, the way that our database server is configured means that we have the permissions to be able to drop and create databases and hence seed some data.  Note a web app does *not* typically have these permissions  Add this code   |  | | --- | | ZooContext ctx = serviceProvider.GetService<ZooContext>();  ctx?.Database.EnsureDeleted();  ctx?.Database.EnsureCreated();  AddSampleData(ctx);  foreach (Zoo zoo in ctx.Zoos) |   Uncomment the provided code for AddSampleData() (TO DO 5) | |
| 11 | Ctrl+F5 to run it and you should get the same output as before.  Now comment out the three highlighted lines that you have just added in (highlighted in yellow above) – the database is present and populated so we should get the same result when we run it?  Try it and see.  Can you explain why it worked with a new database but not an existing one? (Hint – Fix Up) | |
| 12 | Let see what SQL commands EntityFramework is invoking.  In the Assets folder is the Express Profiler. There is a .msi file but you can just run the .exe  Make sure the Server is set to the appropriate server (.\SQLExpress or (local). Press the green arrow to start a new trace and run the Console App (with the 3 lines commented out)  The trace should show  A screenshot of a computer  Description automatically generated  I.e. only the Zoos have been requested. | |
| 13 | Solve the problem by using Eager Loading   |  | | --- | | foreach (Zoo zoo in ctx.Zoos.Include(z=>z.Animals)) | | |
| 14 | Note we still have the ‘backpointer’ line  Console.WriteLine($"......in zoo {animal.Zoo.Name}");  Which previously caused a NullReferenceException.  Confirm this now works – ie EntityFramework populates it for free. | |
| 15 | Undo your Eager Loading solution.  Solve the problem by Lazy Loading :  Uncomment TODO 6  And use NuGet to import Microsoft.EntityFrameworkCore.Proxies  Finally, make this change in the Zoo class   |  | | --- | | public virtual ICollection<Animal> Animals { get; set; } = new List<Animal>(); |   Run it - it should work.  Use the profiler to see the 3 SQL statements | |
| 16 | In OnConfiguration(), comment out  optionsBuilder.UseLazyLoadingProxies(); | |
| 17 | Explicit Loading is not so easy to remember. Implement it:   |  | | --- | | ctx.Entry(zoo).Collection(z=>z.Animals).Load();  Console.WriteLine($"\nName...  foreach (Animal animal in zoo.Animals) |   Check it works | |
| Migration | | |
| 18 | | Make this change to the Animal class   |  | | --- | | public string Name { get; set; }  public double Weight { get;set;}  public virtual Zoo Zoo { get;set;} |   Ctrl+F5 and confirm you get an exception:  A black background with white text  Description automatically generated  Comment out this new line. |
| 19 | | We will use ‘Migrations” to resolve this.  Since Migrations does not use dependency injection, we need to tell it how to configure our connection. Uncomment the ZooContextFactory at the bottom of the ZooContext.cs file. Since this class implements the interface IDesignTimeDbContextFactory<ZooContext>, Migrations will automatically use it to create the context. Watch out for the server name in the connection string (.\SQLExpress or (local). |
| 20 | | Now, select Tools/Nuget Package Manager/Package Manager Console  Into the new console window that appears, type the following and press enter:  add-migration Initial  A class gets created (and opened) in the new Migrations folder, called Initial. It has a method called “Up”, which creates the database (without the Weight property), but we don’t need that to happen – our database already exists.  Modify the method so that it doesn’t do anything:   |  | | --- | | protected override void Up(MigrationBuilder migrationBuilder)  {  return;  migrationBuilder.CreateTable(...... |   Now, back in the package manager console, type and run:  update-database  And then, remove the “return” from the “Up” method. (This method won’t run again, since entity framework remembers that this migration has already been run – unless we drop and create the database.) |
| 21 | | Un-comment the Weight property. Then, type the following two commands in the package manager console to add it to the database:  add-migration AddWeight  update-database  The new column has now been added to the Animals table. Check this is SSMS, and also press F5 and check your program now runs again. |
| 22 | | All the animals have weights which have defaulted to 0. We could now easily modify our program to include weights for each animal and display those weights. If you have time, try to do this! |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 06b: Using the Entity Framework

## Part B – Code First from Database

## Objective

To learn how to create projects that use the Entity Framework to operate in a Code First from Database manner.

## Overview

In the previous (optional) lab you did Code First – i.e. start with the code and construct the database from it. In this lab, we do it the other way around – start with the Database and build the code from the database schema.

This exercise will take around minutes.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## Steps:

|  |  |
| --- | --- |
| 1 | Start a new Core Console App called EFFromDatabase and add the following NuGet packages:  Microsoft.EntityFrameworkCore.SqlServer  Microsoft.EntityFrameworkCore.Tools  Microsoft.EntityFrameworkCore.Design  Bricelam.EntityFrameworkCore.Pluralizer  (without the last one, the scaffolder will produce plural-named classes eg Customer**s**) |
| 2 | Ensure you have the Northwind database installed in .\SqlExpress or (local).  If not, there is a SQL script for installing is in the Assets folder |
| 3 | Open a command window at the project folder (as we’ve done in previous labs) and enter this (it must all be on 1 line so drop it into Notepad first and check no new lines and regular quotes – not “xx” type of quotes).  dotnet ef dbcontext scaffold "Server=.\SqlExpress;Encrypt=False;Database=Northwind;Trusted\_Connection=True" Microsoft.EntityFrameworkCore.SqlServer -o Models --context NorthwindContext  **NOTE: Depending on the version of SQL Server you are using you may need to replace** .\SQLExpress with (local).  This should produce all classes into a folder named ‘Models’ |
| 4 | Have a look at the Customer class and note:  It’s a partial class  There are no fields – it is all properties  Collections are virtual |
| 5 | Enter this into Main() and run it  using (NorthwindContext ctx = new NorthwindContext())  {  foreach (Customer c in ctx.Customers  .Include(c => c.Orders)  .ThenInclude(o => o.OrderDetails)  .ThenInclude(od => od.Product))  {  Console.WriteLine(c.ContactName);  foreach (Order o in c.Orders)  {  Console.WriteLine("..." + o.OrderId);  foreach (OrderDetail od in o.OrderDetails)  {  Console.WriteLine("......" + od.Product.ProductName);  }  }  }  }  You should get lots of Order and Order Details |

|  |  |
| --- | --- |
| Query Filters | |
| 6 | Comment-out all the above code in Main() and add in this code  using (NorthwindContext ctx = new NorthwindContext())  {  var discontinued = ctx.Products.Where(p => p.Discontinued).ToList();  discontinued.ForEach(d => Console.WriteLine(d.ProductName));  }  Run it – it will output a list of all discontinued products:  A screenshot of a computer  Description automatically generated |
| 7 | But suppose you *never* wanted to see discontinued products in any of your queries. Right now, you’d have to put in a Where statement into every query. EFCore has a QueryFilter feature where you can do this globally.  Go to the Northwind context class and search for <Product> (include the angle brackets in the search)  When you reach this line  A close up of a text  Description automatically generated  Somewhere in this section add a global query filter:  entity.HasQueryFilter(e=>!e.Discontinued); |
| 8 | Run again and this time your query won’t even see discontinued products. |
| 9 | If, in particular queries, you really did want to see the discontinued products, you can override this (in Program/Main()) :  ctx.Products.Where(p => p.Discontinued).**IgnoreQueryFilters().**ToList();  Do this and confirm you again see the discontinued products. |
| 10 | Remove the Query Filter you just added to NorthwindContext and comment-out all code in Main() |

|  |  |
| --- | --- |
| Adding own functions to database classes | |
| 11 | Suppose that the company wanted to sell off discontinued lines at half price.  Comment out existing code in Main() and add this:  using (NorthwindContext ctx = new NorthwindContext())  {  Product chang = ctx.Products  .Where(p => p.ProductName == "Chang").Single();  Console.WriteLine(  $"Unitprice={chang.UnitPrice } unitInStock={chang.UnitsInStock}");  decimal? offerPrice = chang.UnitPrice;  if (chang.UnitsInStock < 5 && chang.Discontinued)  {  offerPrice /= 2;  }  Console.WriteLine($"Offer price = {offerPrice}");  }  Run this  The offer price is 19 because this produce does not meet the UnitsInStock<5 and Discontinued conditions.  A black and white screen with white text  Description automatically generated |
| 12 | Open up the Products table in SSMS by right-clicking and “Edit top 200 rows”. Find the product    Change the Units in stock to 4 and set the Discontinued status to True to show the above code is working  A screenshot of a computer  Description automatically generated |
| 13 | It would be good to move this code into the Product class – that’s where it really belongs.  Open the Product class and add  public decimal? OfferPrice =>   (UnitsInStock < 5 && Discontinued) ? UnitPrice /= 2 : UnitPrice;  Have this as your code in Main()  using (NorthwindContext ctx = new NorthwindContext())  {  Product chang = ctx.Products  .Where(p => p.ProductName == "Chang").Single();  Console.WriteLine(  $"Unitprice={chang.UnitPrice } unitInStock={chang.UnitsInStock}");  Console.WriteLine($"Offer price = {chang.OfferPrice}");  }  Confirm it still works. |
| 14 | Now imagine that we want to delete the Discontinued column completely from the database. If we did this, we would need to re-generate our classes (imagine there might be other subtle changes we didn’t necessarily know about).  Doing this would lose the OfferPrice code we’ve just added to the Product class as this is going to get blown away. |
| 15 | To solve the first one, add a new folder called Logic alongside Models.  Hold down the Ctrl key and drag Product into Logic (ie copy it) |
| 16 | Delete OfferPrice from Models/Product and delete everything except OfferPrice from Logic/Product |
| 17 | Suppose we had an additional constraint that ProductName must be shorter than 50 characters. We can accommodate this too.  Make your Logic/Product file look like this:  using Microsoft.AspNetCore.Mvc;  using System;  using System.Collections.Generic;  using System.ComponentModel.DataAnnotations;  namespace EFFromDatabase.Models  {  [ModelMetadataType(typeof(Product\_Buddy))]  public partial class Product  {  public decimal? OfferPrice =>   (UnitsInStock < 5 && Discontinued) ? UnitPrice /= 2 : UnitPrice;  }  public class Product\_Buddy  {  [MaxLength(50)]  public string ProductName { get; set; }  }  }  If you Ctrl+dot ModelMetadataType it will suggest you install Microsoft.AspNetCore.Mvc.Core. Install this then resolve namespaces.  Ie this extra constraint is available to the MVC validation system.  Run and make sure all is still working |
| 18 | Now, in SSMS, right-click the Products table > Design and delete the Discontinued column.  Save the table |
| 19 | Run the app again and you will get  A black background with white text  Description automatically generated  Imagine it wasn’t just the one change and that now we need to do a complete re-scaffolding |
| 20 | Open a command window at the project directory and enter  dotnet ef dbcontext scaffold "Server=.\SqlExpress;Database=Northwind;Trusted\_Connection=True;Encrypt=False" Microsoft.EntityFrameworkCore.SqlServer -o Models --context NorthwindContext --force  Watch out for .\SQLExpress vs (local)  Check that the ‘Discontinued’ property has gone from the Product class |
| 21 | If you now compile, you can fix the 2 ‘Discontinued’ issues:  public decimal? OfferPrice => (UnitsInStock < 5) ? UnitPrice /= 2 : UnitPrice;  Run and you’re back in business! |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 07a: Introduction to the Estate Agent Microservices Project

The next set of labs are based around the creation of a set of microservices for a chain of estate agencies. We don't have the time to create the entire front and back ends but to set the scene here's an overview of the kind of things the finished site would be capable of.

The development revolves around an Estate Agent Management System with the following Features:

## Feature: Manage Buyer

**Scenario: Register Buyer**

Given the new buyer with the given first name and surname does not exist

When a create buyer request is received with the given first name and surname

Then a new buyer record is created with a buyer ID

## Feature: Manage Seller

**Scenario: Register Seller**

Given a seller with the given first name and surname does not exist

When a create seller request is received with the given first name and surname

Then a new seller record is created with a seller ID

## Feature: Manage Property

**Scenario: Add Property**

Given a seller exists for the new property

When a create property request for the given seller is received

Then the property is added to the catalogue

Then the property status is set to FORSALE

**NOTE**: A property can have the following status: FORSALE, SOLD, WITHDRAWN

**Scenario: Find properties**

When a Find properties request is received

Then a list of properties with the corresponding criteria is shown

**Scenario: Withdraw Property that is FORSALE**

Given The required Property exists

Given The required Property is FORSALE

When a Withdrawn property request is received

Then property status is changed to WITHDRAWN

**Scenario: Resubmit Property that has been WITHDRAWN**

Given The required Property exists

Given The required Property has been WITHDRAWN

When a Resubmit property request is received

Then property status is changed to FORSALE

**Scenario: Amend property details**

Given The required Property exists

Given The required Property is FORSALE

When an Amend property request is received

Then property details are updated

## Feature: Manage Bookings

**Scenario: Make booking with Slot available**

Given no active booking exists for the desired time slot for the property

Given the property status is FORSALE

Given the buyer is registered

When a viewing is requested

Then a booking is created for the buyer for the property at the given time slot

**Note**: Viewing slot is every hour on the hour between 8am to 5pm every day including weekends and holidays

**Scenario: Make Booking - Time Slot not available**

Given a booking already exists for the required timeslot for the given property

When a viewing is requested is made for that time slot

Then an error is shown to the user

**Scenario: Cancel Booking**

Given a booking exists

When a cancel booking request is made

Then the booking is removed

## Minimal Viable Product

**Manage Seller**

* Register a new seller
* Display all sellers

**Manage Properties**

* Add properties
* Display all properties
* Find and display properties with given search criteria on price, bedrooms, bathroom and garden
* Withdraw a property
  + Cascade delete any associated bookings
* Resubmit a property

**Manage Buyer**

* Register new buyer
* Display all buyers

**Manage Bookings**

* Add bookings
  + Ensure the proposed date and time are available
  + Don't allow bookings for houses that are SOLD
* Display all bookings for a property

# Database Schema

A screenshot of a computer

Description automatically generated

# Lab 07b: Creating an ASP.NET **MVC** API Microservice

## Objective

Your goal is to create a microservice for "buyer" information. The microservice should be support full CRUD capabilities.

## Overview

The Estate Agent application needs to keep track of potential property buyers. In this lab you will create a Visual Studio solution that hosts a Buyers microservice that is built using Visual Studio’s ASP.NET API template. The microservice should allow a consumer of the service to:

* Retrieve a list of all buyers.
* Retrieve a buyer by their id.
* Retrieve a buyer by their name.
* Add new buyers.
* Delete existing buyers.
* Update existing buyers.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## DATABASE SETUP

**Before you can start on the Web application development you will need to create the EstateAgent database in SQL Server. You will find a SQL script called " EstateAgentScriptWithDeleteConstraints.sql" in the Assets folder which you will need to open and run from inside SQL Server Management Studio.**

## STEPS

|  |  |
| --- | --- |
| 1 | Use Visual Studio to create a new ASP.NET Core Web API project called "Buyer Service". Call the solution that hosts the project "EstateAgentBackEnd". Ensure the project uses an up-to-date version of .NET (e.g. 8.0). Don't worry about authentication or enabling Docker but do ensure the "Use controllers" box is checked. A screenshot of a computer  Description automatically generated |
| 2 | Delete any preexisting controllers and/or classes based around the weather. |
| 3 | Use NuGet package manager to add references to:   1. Microsoft.EntityFrameworkCore 2. Microsoft.EntityFrameworkCore.SqlServer 3. Newtonsoft.Json |
| Sorting out the database access logic: | |
| 4 | Add a folder called Models to the project. |
| 5 | Add a class called Buyer to the Models folder. |
| 6 | Replace the code in the Buyer.cs file with the following:  using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;  namespace BuyerService.Models  {  [Table("buyer")]  public class Buyer  {  [Column("BUYER\_ID")]  [Key]  public int Id { get; set; }  [Column("FIRST\_NAME")]  public string? FirstName { get; set; }  [Column("SURNAME")]  public string? Surname { get; set; }  [Column("ADDRESS")]  public string? Address { get; set; }  [Column("POSTCODE")]  public string? Postcode { get; set; }  [Column("PHONE")]  public string? Phone { get; set; }  }  } |
| 7 | Add folder called Infrastructure to the project. |
| 8 | Add a class called BuyerContext to the Infrastructure folder. |
| 9 | Replace the code in the BuyerContext.cs file with the following:  using BuyerService.Models;  using Microsoft.EntityFrameworkCore;  namespace BuyerService.Infrastructure  {  public class BuyerContext : DbContext  {  public BuyerContext(DbContextOptions<BuyerContext> options) :  base(options)  {  }  public DbSet<Buyer> Buyers { get; set; }  }  } |
| 10 | Add an empty Controller called BuyerController to the Controllers folder. |
| 11 | Add using System.Net to the list of existing using statements. |
| 12 | Add a Route attribute to the BuyerController class with a value of "api/[controller]". |
| 13 | Add a private readonly variable of type BuyerContext called \_buyerContext to the top of the class. |
| 14 | Add a constructor to the class that takes a BuyerContext parameter BuyerContext called context. |
| 15 | Add a line of code to the constructor that sets \_buyerContext to the context parameter but only if the context is not null. Throw an ArgumentNullException if it is. |
| 16 | Delete the Index method. |
| 17 | Add a new public method to the BuyerController class called GetBuyers giving it a return type of async Task<IActionResult>. |
| 18 | Decorate the method with the following attributes:   1. HttpGet 2. Route with a value of "buyers" 3. ProducesResponseType with a type of IEnumerable<Buyer> and a statusCode of HttpStatusCode.OK. |
| 19 | Add a line of code to the method that awaits a call to \_buyerContext.Buyers.ToListAsync() placing the returned value into a nullable List<Buyer> variable called buyers. |
| 20 | Return the buyers collection from the method wrapped in an OKObjectResult. |
| 21 | Your code should look something like the following:  [Route("api/[controller]")]  public class BuyerController : Controller  {  private readonly BuyerContext \_buyerContext;  public BuyerController(BuyerContext context)  {  \_buyerContext = context ?? throw new  ArgumentNullException(nameof(context));  }  [HttpGet]  [Route("buyers")]  [ProducesResponseType(typeof(IEnumerable<Buyer>),  (int)HttpStatusCode.OK)]  {  List<Buyer>? buyers = await \_buyerContext.Buyers.ToListAsync();  return Ok(buyers);  }  } |
| 22 | Open up the appsettings.json file. |
| 23 | Add the following connection string details to the top of the file just below the first opening curly brace. **NOTE: The connection string assumes you have a local version of SQL Server installed that is up and running and, depending on the type of SQL Server engine installed you may need to use ".\\SQLExpress" as the Server name rather than "(local)"**:  "ConnectionStrings": {  "sqlestateagentdata":  "Server=(local);Database=estateagent;Trusted\_Connection=Yes;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True"  }, |
| 24 | Open up the Program.cs file. |
| 25 | Add the following code just beneath the builder.Services.AddControllers line:  builder.Services.AddDbContext<BuyerContext>(options =>  options.UseSqlServer(  builder.Configuration.GetConnectionString("sqlestateagentdata"))); |
| 26 | That's all the database access logic in place along with a method that should return the content of the buyers table. Unfortunately, neither the estateagent database nor the buyers table exist so we will need to create some code that checks for this and creates and seeds them if necessary. |
| 27 | Add a NuGet reference to AutoFixture. This library is usually used in the generation of test data inside unit test projects but we're going to use it to generate some random data to populate the buyers table. |
| 28 | Add a **static** class called BuyerSeeder to the Infrastructure folder |
| 29 | Add the following code to the class:  public static void Seed(this BuyerContext buyerContext)  {  if (!buyerContext.Buyers.Any())  {  Fixture fixture = new Fixture();  fixture.Customize<Buyer>(buyer => buyer.Without(p => p.Id));  //--- The next two lines add 100 rows to your database  List<Buyer> products = fixture.CreateMany<Buyer>(100).ToList();  buyerContext.AddRange(products);  buyerContext.SaveChanges();  }  } |
| 30 | Return to the Program.cs file and add the following inside the if (App.Environment.IsDevelopment()) test.  if (app.Environment.IsDevelopment())  {  using (var scope = app.Services.CreateScope())  {  var buyerContext =  scope.ServiceProvider.GetRequiredService<BuyerContext>();  buyerContext.Database.EnsureCreated();  buyerContext.Seed();  }  app.UseSwagger();  app.UseSwaggerUI();  } |
| 31 | Launch the app and wait for the Swagger page to open in a browser. Then Test drive the call to the "/api/Buyer/buyers" endpoint and ensure it returns some data that looks something like the following: A screenshot of a computer  Description automatically generated |

## Adding New Buyers

|  |  |
| --- | --- |
| 32 | Return to the BuyerController and create a new method with a return type of async Task<IActionResult> called InsertBuyer that takes a Buyer called buyer as a parameter. |
| 33 | Decorate the method with the following attributes (the Route and ProducesResponseType are exactly the same as those used on the GetBuyers method):   1. HttpPost 2. Route with a value of "buyers" 3. ProducesResponseType with a type of IEnumerable<Buyer> and a statusCode of HttpStatusCode.OK. |
| 34 | **NOTE: Whilst the methods have different names (GetBuyers and InsertBuyer) their Web API endpoints are identical (/api/Buyer/buyers). The difference is in the HTTP request types (Get and Post).** |
| 35 | We really ought to validate the properties of the buyer parameter to make sure they meet any business constraints. However, given you should already have a good idea as to how to go about doing this we'll give it a miss and focus on the "microservice" elements of the tasks in hand. |
| 36 | Add code to the method that calls the Add method of the Buyers collection associated with the \_buyerContext passing it the buyer object. |
| 37 | Invoke the \_buyerContext object's SaveChanges method. |
|  | If the insert is successful, the entity framework should have updated the buyer object's Id property with the value automatically generated by the database. Consequently, we will return the updated buyer object wrapped in an OKObjectResult. |
| 38 | Launch the app and test drive the new insert method by using the Swagger interface. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

## If you have time:

|  |  |
| --- | --- |
| 39 | Try to create methods that allow Buyers to be removed from the database and have their data updated making use of the HttpDelete and HttpPut attributes. Make sure to keep the Route signatures the same as those used for GetUsers and InsertUser. Note:   1. To delete a Buyer, you will need to ensure they exist in the database by making use of the \_buyerContext.Buyers.SingleOrDefaultAsync method. If the lookup is successful you need to pass the object reference to the \_buyerContext.Buyers.Remove method. 2. To update a Buyer, note there is no Update method. Instead, you will have to make use of the use of the \_buyerContext.Buyers.SingleOrDefaultAsync method to retrieve the appropriate Buyer object (let's call it "b") from the database. Then you need to set this object's properties to those of the passed in Buyer parameter. Finally, you need to invoke SaveChanges. |
| 40 | If you manage to do all of the above, then add two final methods to the Controller class that retrieve a single Buyer object based on a passed in Id or buyer name. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 08: Creating an ASP.NET **Minimal** API Microservice

## Objective

Your goal is to return to the Estate Agent application and create a microservice for "seller" information.

## Overview

The Estate Agent application you worked on in the previous lab needs to keep track of property sellers as well as potential buyers. In this lab you will add a Sellers microservice that is built using Visual Studio’s ASP.NET **Minimal** API template. The microservice should allow a consumer of the service to:

* Retrieve a list of all sellers.
* Retrieve a seller by their id.
* Retrieve a seller by their name.
* Add new sellers.
* Delete existing sellers.
* Update existing sellers.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## STEPS

|  |  |
| --- | --- |
| 1 | Use Visual Studio to add a new ASP.NET Core Web API project called "Seller Service" to the "EstateAgentBackEnd" solution. Ensure the project uses an up-to-date version of .NET (e.g. 8.0). Don't worry about authentication or enabling Docker but do ensure the "Use controllers" box is **unchecked**. A screenshot of a computer  Description automatically generated |
| 2 | Delete any preexisting code and/or classes based around the weather including the summaries array and app.MapGet function in Program.cs. |
| 3 | Use NuGet package manager to add references to:   1. Microsoft.EntityFrameworkCore 2. Microsoft.EntityFrameworkCore.SqlServer 3. Newtonsoft.Json |

## Sorting out the database access logic:

|  |  |
| --- | --- |
| 4 | Add a folder called Models to the project. |
| 5 | Add a class called Seller to the Models folder. |
| 6 | Replace the code in the Seller.cs file with the following:  using System.ComponentModel.DataAnnotations.Schema;  using System.ComponentModel.DataAnnotations;  namespace SellerService.Models  {  [Table("seller")]  public class Seller  {  public Seller()  {  //Properties = null;  }  [Column("SELLER\_ID")]  [Key]  public int Id { get; set; }  [Required]  [StringLength(255)]  [Column("FIRST\_NAME")]  public string FirstName { get; set; }  [Required]  [StringLength(255)]  [Column("SURNAME")]  public string Surname { get; set; }  [Required]  [StringLength(255)]  [Column("ADDRESS")]  public string Address { get; set; }  [Required]  [StringLength(255)]  [Column("POSTCODE")]  public string Postcode { get; set; }  [Required]  [StringLength(20)]  [Column("PHONE")]  public string Phone { get; set; }  public object Clone()  {  return new Seller  {  Id = this.Id,  FirstName = this.FirstName,  Surname = this.Surname,  Address = this.Address,  Postcode = this.Postcode,  Phone = this.Phone  };  }  public bool Equals(Seller? other)  {  return Id == other.Id;  }  }  } |
| 7 | Add a folder called Infrastructure to the project. |
| 8 | Add a class called SellerContext to the Infrastructure folder. |
| 9 | Replace the code in the SellerContext.cs file with the following:  using Microsoft.EntityFrameworkCore;  using SellerService.Models;  namespace SellerService.Infrastructure  {  public class SellerContext : DbContext  {  public SellerContext(DbContextOptions<SellerContext> options) :  base(options)  {  }  public DbSet<Seller> Sellers { get; set; }  }  } |
| 10 | Open up the appsettings.json file. |
| 11 | Add the following connection string details to the top of the file just below the first opening curly brace. **NOTE: The connection string assumes you have a local version of SQL Server installed that is up and running and, depending on the type of SQL Server engine installed you may need to use ".\SQLExpress" as the Server name rather than "(local)"**:  "ConnectionStrings": {  "sqlestateagentdata": "Server=(local);Database=estateagent;Trusted\_Connection=Yes;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True"  }, |
| 12 | Open up the Program.cs file. |
| 13 | Add the following code just beneath the "//Add services to the container" comment:  builder.Services.AddDbContext<SellerContext>(options =>  options.UseSqlServer(  builder.Configuration.GetConnectionString("sqlestateagentdata"))); |
| 14 | Add a NuGet reference to AutoFixture. This library is usually used in the generation of test data inside unit test projects but we're going to use it to generate some random data to populate the sellers table. |
| 15 | Add a **static** class called SellerSeeder to the Infrastructure folder |
| 16 | Add the following code to the class:  public static void Seed(this SellerContext sellerContext)  {  if (!sellerContext.Sellers.Any())  {  Fixture fixture = new Fixture();  fixture.Customize<Seller>(seller => seller.Without(p => p.Id));  //--- The next two lines add 100 rows to your database  List<Seller> sellers = fixture.CreateMany<Seller>(100).ToList();  sellerContext.AddRange(sellers);  sellerContext.SaveChanges();  }  } |
| 17 | Return to the Program.cs file and add the following inside the if (App.Environment.IsDevelopment()) test.  if (app.Environment.IsDevelopment())  {  using (var scope = app.Services.CreateScope())  {  var sellerContext =  scope.ServiceProvider.GetRequiredService<SellerContext>();  sellerContext.Database.EnsureCreated();  sellerContext.Seed();  }  app.UseSwagger();  app.UseSwaggerUI();  } |
| 18 | That's all the database access logic in place. All we need now are some Http endpoints. |
| 19 | Given that this time we are creating a minimal microservice there is no need to add any Controllers. Instead, we are going to add our endpoints directly to the Program.cs file in the form of lambda expressions. |
| 20 | At the foot of the Program.cs file just before the "app.Run()" line add the following code:  app.MapGet("/sellers", async (SellerContext db) =>  await db.Sellers.ToListAsync()); |
| 21 | The code creates an anonymous function that specifies an Http endpoint ("/sellers") that uses the SellerContext to asynchronously implicitly return all the sellers in the database's sellers table. |
| 22 | Before running the program you will need to delete the database from SQL Server otherwise the code in SellerSeeder will trigger an SQLException. You can do this inside SQL Server Management Studio (SSMS) byt right-clicking on the estateagent database and selecting Delete. Make sure the Close existing connections box is checked and press OK. |
| 23 | Make sure the SellerService project has been configured to be the Start-up project and launch the app. Wait for the Swagger page to open in a browser. Then Test drive the call to the "/sellers" end-point and ensure it returns some data that looks something like the following: A screenshot of a computer  Description automatically generated |

## Adding New Sellers

|  |  |
| --- | --- |
| 24 | Return to the Program.cs and create a new anonymous method that calls the app object's MapPost method passing it "/sellers" as the pattern parameter and async (Seller seller, SellerContext db) => as the signature of the lambda delegate. |
| 25 | Add the following code as the delegate logic:  db.Sellers.Add(seller);  await db.SaveChangesAsync();  return Results.Created($"/sellers/{seller.Id}", seller); |
| 26 | **NOTE: Whilst the two methods have the same endpoints (sellers). Like before, with Buyers, the difference is in the HTTP request types (MapGet and MapPost).** |
| 27 | We really ought to validate the properties of the seller object to make sure they meet any business constraints. However, we didn't do it for the Buyer functionality so we're not going to do it here! The rest of the functionality is pretty much the same as it was for the BuyerService's InsertBuyer method and so, needs no explanation |
| 28 | Launch the app and test drive the new insert method by using the Swagger interface. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

## If you have time:

|  |  |
| --- | --- |
| 29 | Try to create additional methods that allow Sellers to be removed from the database and have their data updated making use of the app.MapDelete and app.MapPut methods. Make sure to keep the Route signatures the same as those used for MapGet and MapPost. Note:   1. To delete a Seller, you will need to ensure they exist in the database by making use of the SellerContext's .Sellers.FindAsync method. If the lookup is successful you need to pass the object reference to the SellerContext's Sellers.Remove method. 2. To update a Seller, note there is no Update method. Instead, you will have to make use of the use of the SellerContext's Sellers. FindAsync method to retrieve the appropriate Seller object (let's call it "s") from the database. Then you need to set this object's properties to those of the passed in Seller parameter. Finally, you need to invoke SaveChanges. |
| 30 | If you manage to do all of the above, then add two final methods to the Controller class that retrieve a single Single object based on a passed in Id or seller name. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 09: Estate Agent Microservice – One Database per Microservice

## Objective

Your goal is to edit the microservices so that they each use their own single table database. You will also need to worry about referential integrity between the databases.

## Overview

In the current setup all the microservices use the same database which is configured to "cascade delete" any dependencies. In our case this means when a property is removed from the properties table the database ensures any associated bookings for that property are automatically deleted from the "bookings" table. In the "new world" of microservices we are encouraged to develop services that each use their own database such that the databases only host a minimal number of tables (typically just one). This means we, as developers, need to worry about the referential integrity of our data rather than letting the databases do it for us.

Note: There are many other dependencies which the original database could (and would) have managed automatically. For the purposes of brevity, we won't be managing all of these issues in this lab (that's something you could consider doing later if and when you have time). However, we will deal with the "bookings for deleted properties" issue mentioned above and we will also create code that ensures a booking can't be made for a non-existent property or buyer.

**Important note: Splitting a relational database up in the way the lab does is often not a good idea. What if the deletion of a property is successful but, for some reason, the deletion of associated bookings fails? For the purpose of this and other following labs we are going to ignore these concerns. The point of the labs is to understand how to create and containerize microservices and to deal “conceptually” with some of the issues that raises.**

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## STEPS

## Reconfigure code so each service uses its own single table database.

|  |  |
| --- | --- |
| 1 | Locate the SQL script called "EstateAgentFourSeparateDatabasesScript.sql" in the Assets folder. Open and run it inside SQL Server Management Studio (SSMS). The script generates four individual databases called EABuyer, EASeller, EAProperty and EABooking. Each database contains a single table whose name is directly related to the name of the database it resides in. Each table contains a small amount of test data. |
| 2 | Open the starter project. You will notice we've added two more projects to the solution (BookingService and PropertyService). Both are fully functioning and currently make use of the same single EstateAgent database that the (unchanged) Buyer and Seller services are using.  Also note all four services offer only basic CRUD capabilities. No validation is done on any of the submitted data. We've done this for the sake of simplicity. There would be a world of pain to go through to get the code up to scratch for a real-world deployment 😉. |
| 3 | Open the BookingService project's appsettings.json file and change the name of the database (located in the connection string) to EABooking:  "ConnectionStrings": {  "sqlestateagentdata": "Server=(local);Database=**EABooking**;Trusted\_Connection=Yes;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True"  } |
| 4 | Repeat step 3 for each of the other projects changing the database names to EABuyer, EAProperty or EASeller accordingly. |

## Manage the deletion of properties that have associated bookings.

|  |  |
| --- | --- |
| 5 | If a property has been successfully removed from the EAProperty database, we need to get the BookingService to remove any associated bookings that have been made by any potential buyers. We'll do this by making a call to a BookingService endpoint (that we've yet to write) that takes a propertyId as a parameter. But first we will tackle the code that needs to be added to the PropertyService: |
| 6 | Locate the MapDelete function in the PropertyService project's program.cs file. |
| 7 | Add the following code between the line that calls db.SaveChanges() and the return statement (**Note: the highlighted port number will probably be different for your app. You can find the number by looking at the BookingService project's launchSettings.json file located in the Properties folder.**):  var http = new HttpClient();  string url = $"http://localhost:**5225**/bookingsByPropertyId/{id}";  HttpResponseMessage response = await http.DeleteAsync(url); |
| 8 | Change the return statement to test the result of the API call to see if it was successful (response.IsSuccessStatusCode). If it is then return Results.NoContent() but if not return Results.NotFound(). |
| 9 | Locate the MapDelete function in the BookingService project's program.cs file and add a new MapDelete function beneath it with a pattern parameter of bookingsByPropertyId. Use the same lambda expression for the second (delegate) parameter as the other DeleteMap function. |
| 10 | Add code to the new function that returns all the bookings (as a List) where the PropertyId of each booking matches the passed in id parameter. Make this call an awaited task.  Your code may look like the following:  app.MapDelete("/bookingsByPropertyId/{id}",  async (int id, BookingContext db) =>  {  List<Booking> bookings =  await db.Bookings.Where(b => b.PropertyId == id).ToListAsync(); |
| 11 | Add code that tests to ensure a collection was returned and the collection is not empty. If not return Results.NotFound().  If some bookings have been found iterate around the collection and call the db.Bookings.Remove() method for each of them. |
| 12 | After the loop completes call db.SaveChangesAsync() to force the changes to the database and return Results.NoContent(). |
| 13 | Test the application by starting the app (F5). Notice the solution has been configured to start all four projects. |
| 14 | Use Swagger (or Postman or SSMS) to add a new property to the EAProperty database. Look at the response to ensure this has been successful and make a note of the newly generated propertyId. Then (again using Swagger, Postman or SSMS) add 3 new bookings for the property. |
| 15 | Finally Use Swagger (or Postman) to trigger the PropertyService's Delete function to delete the property. Ensure that both the property and its corresponding bookings have been removed from both databases. You may wish to set some breakpoints within the relevant functions and use the debugger to step through the code to follow the action. |

## Ensuring a booking can't be made for a non-existent property or buyer.

|  |  |
| --- | --- |
| 16 | Return to the program.cs file in the BookingService project. |
| 17 | Locate the MapPost function. |
| 18 | Before we can add a new booking, we need to ensure that passed in booking's propertyId and buyerId are set to values that exist in their corresponding databases. We'll achieve this by making API calls to the "Get by Id" functions supported by both of the services and ensuring we get valid Property and Buyer objects back.  Add the following to the top of the function just above the db.Bookings.Add(booking) line (make sure you replace the highlighted port number with the one associated with your PropertyService API (You can find the number by looking at the PropertyService project's launchSettings.json file located in the Properties folder.).:  var http = new HttpClient();  //Check to see if PropertyId is valid  string url = $"https://localhost:**7139**/properties/{booking.PropertyId}";  HttpResponseMessage response = await http.GetAsync(url);  string responseJson = response.Content.ReadAsStringAsync().Result;  dynamic responseData = JsonConvert.DeserializeObject(responseJson);  if (responseData == null || responseData["id"] != booking.PropertyId)  return Results.NotFound(); |
| 19 | Add similar code that tests to see if the passed in booking object's buyerId is also valid. **Note, that because we created the BuyerService using a Controller class the endpoint URL will be a little bit different**. |
| 20 | Run and test the functionality to ensure bookings are only added to the Bookingservice's database when both the PropertyId and BuyerId exist as entries in their respective datbases. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

## If you have time:

|  |  |
| --- | --- |
| 21 | Try and add further code to the projects so as to further maintain relational integrity. Scenarios to consider include:   * Deleting any associated properties when a seller is removed from the EASeller database. * Deleting any associated bookings when a buyer is removed from the EABuyer database. You may also want to consider setting any related BuyerId's to null in the EAProperty database. * Not allowing a property to be added if the sellerId is not in the EASeller database and. If specified, the BuyerId is not in the EABuyer database.   **Note, the model solution does not implement any of these potential enhancements.** |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 10a: Docker MINI Lab 1 - Register for Docker

In this mini lab, we will register for Docker and then exercise the commands outlined previously.

First, register an account with Docker at https://hub.docker.com

Once created, open Windows PowerShell, authenticate the Docker CLI to Dockerhub with the login command.

docker login

Next, let's download a hello world image.

docker search hello-world

The search command returns a table of images relevant to the search term with their description and whether they are an official image.

|  |  |  |  |
| --- | --- | --- | --- |
| NAME | DESCRIPTION | STARS | OFFICIAL |
| hello-world | Hello World! (an example of minimal Dockeriz… | 2298 | [OK] |
| atlassian/hello-world | 0 |  |  |
| rancher/hello-world | This container image is no longer maintained… | 6 |  |
| hellobello/hello-world | 0 |  |  |
| hellojinl/hello-world | 0 |  |  |
| tutum/hello-world | Image to test docker deployments. Has Apache… | 90 |  |
| infrastructureascode/hello-world | A tiny "Hello World" web server with a healt… | 1 |  |
| uniplaces/hello-world | 0 |  |  |
| prajwalendra/hello-world | 0 |  |  |
| arm64v8/hello-world | Hello World! (an example of minimal Dockeriz… | 3 |  |
| silver8642/hello-world | 0 |  |  |
| satoune007/hello-world | 0 |  |  |
| dhaneshande/hello-world | 1 |  |  |
| nirmata/hello-world | 0 |  |  |
| ewocker/hello-world | 0 |  |  |
| i386/hello-world | Hello World! (an example of minimal Dockeriz… | 0 |  |
| arm32v5/hello-world | Hello World! (an example of minimal Dockeriz… | 1 |  |
| dotnetcoreservices/hello-world | 1 |  |  |
| garystafford/hello-world | Simple hello-world Spring Boot service for t… | 0 |  |
| topmonkscom/hello-world | to be removed | 0 |  |
| aashaymz/hello-world | 0 |  |  |
| songleo/hello-world | 0 |  |  |
| venkyboyz/hello-world | 0 |  |  |
| lightsail/hello-world | 0 |  |  |
| ashyadav002/hello-world | 0 |  |  |

We see a hello-world image on line 1. Use the docker pull command to download the image.

docker pull hello-world

See what docker images exist by using the docker images command:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
| hello-world | latest | d2c94e258dcb | 15 months ago | 13.3kB |

**Rename the hello-world image so that it is prefixed with your *Docker hub* name**

Replace <your\_docker\_username> with your Docker username and run the following command:

docker tag hello-world <your\_docker\_username>/hello-world

Now, when we run docker images, we can see two images. We have created a new image with a different name.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
| hello-world | latest | d2c94e258dcb | 15 months ago | 13.3kB |
| <your\_docker\_username>/hello-world | latest | d2c94e258dcb | 15 months ago | 13.3kB |

Now we can push the newly tagged image to our repository by tagging it in the format [USERNAME]/[IMAGE]:[TAG].

docker push <your\_docker\_username>/hello-world

Navigate to [Dockerhub](https://hub.docker.com/), and you should see a new repository.

Now that the image has been uploaded, we can delete the images we have locally and free up space.

**Delete the *hello-world* image**

docker rmi hello-world

The output displays the different layers being deleted. The last two lines:

Deleted: sha256:dbf7b16cf5d32dfec3058391a92361a09745421deb2491545964f8ba99b37fc2

Deleted: sha256:a2ae92ffcd29f7ededa0320f4a4fd709a723beae9a4e681696874932db7aee2c

Repeat for the renamed image

docker rmi <your\_docker\_username>/hello-world

Now, when we run docker images we should see an empty table.

# Lab 10b: Docker MINI LAB 2 – Run and Test a Container

In this mini lab we will go through the process of setting up a SQL Server instance inside a container.

Checklist:

* Spin up a container using the official Microsoft SQL-Server image.
* Map port the machine's port 1433 to the container port 1433
* Retrieve the initial administrator password from the container

**DownlStart the container**

It is important to remember to map the container port to the machine's port. In this case, SQL-Server is configured to port 1433. So, we use the -p flag map the ports to each other. And the -e flag to set a default password.

# -d **for** detached mode so we can still use the terminal

# -p maps the machine's port 1433 to the container's port 1533

# -e sets up environment variables

# *--name names the container so that we are not using a random name*

docker run -e "ACCEPT\_EULA=Y" -e "MSSQL\_SA\_PASSWORD=**<Your\_Strong\_Password>**" -d -p 1533:1433 *--name sql\_server\_container mcr.microsoft.com/mssql/server:2022-latest*

Replace <Your\_Strong\_Password> with a strong password of your own. **For the purpose of the course it is suggested you use "PaSSw0rdPaSSw0rd"**

Now we can check if the container has started properly:

docker ps

You should see a similar output to the following:

CONTAINER ID: 4017a9ca9621

IMAGE: mcr.microsoft.com/mssql/server:2022-latest

COMMAND: "/opt/mssql/bin/perm…"

CREATED: 3 minutes ago

STATUS: Up 3 minutes

PORTS: 0.0.0.0:1533->1433/tcp

NAMES: sql\_server\_container

**connect to SQL Server Linux containers**

Next, we need to connect and log in to the SQL Server instance

Open Azure Data Studio and click the connect button:

A screenshot of a computer

Description automatically generated

Fill in the connection window with the following (use the secure password you specified when setting up the container):

A screenshot of a computer

Description automatically generated

Click the Advanced… button.

Scroll down and enter 1533 for the Port.

Press OK and then press Connect.

If all is well, you will have successfully made a connection to the server.

Close the Azure Data Studio window.

**Tear down**

To stop and delete the container:

*# Stop the container*

docker stop sql\_server\_container

*# Deletes the container*

docker rm sql\_server\_container

# Lab 10c: Docker MINI LAB 3 – Creating and using a Docker File

**1. Create a Minimal C# Application**

First, create a new C# Console Application. There’s no need to fire up Visual Studio, instead, run the following instructions in PowerShell:

dotnet new web -n HelloWorldAspNet

cd HelloWorldAspNet

generates a simple ASP.NET Core web application with a Program.cs file that hosts a minimal web server.

Edit the Program.cs file by typing:

Notepad Program.cs

Replace the code with the following which is essentially the same but ensures the Kestrel server is configured to listen on port 80:

var builder = WebApplication.CreateBuilder(args);

// Explicitly set Kestrel to listen on port 80

builder.WebHost.ConfigureKestrel(serverOptions =>

{

serverOptions.ListenAnyIP(80); // Listen on port 80 for any IP address

});

var app = builder.Build();

app.MapGet("/", () => "Hello, World from ASP.NET Core 8.0!");

app.Run();

**2. Create a Dockerfile**

Next, create a Dockerfile in the same directory as your Program.cs file by typing:

New-Item Dockerfile

Edit the Dockerfile file by typing:

Notepad Dockerfile

Copy code the following code into the file

# Use the official .NET SDK image for building the application

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

WORKDIR /src

# Copy the project files into the container

COPY . .

# Restore dependencies

RUN dotnet restore

# Publish the application

RUN dotnet publish -c Release -o /app

# Use the official .NET runtime image for running the application

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime

WORKDIR /app

# Copy the published output from the build stage

COPY --from=build /app .

# Set the entry point to run the application

ENTRYPOINT ["dotnet", "HelloWorldAspNet.dll"]

# Expose port 80

EXPOSE 80

**3. Build the Docker Image**

Build the Docker image using the following command:

docker build -t helloworldaspnet .

**4. Run the Docker Container**

Run the Docker container using the following command:

docker run --rm -p 8080:80 helloworldaspnet

Browse to localhost:8080. This will output something like:

Hello, World from ASP.NET Core 8.0!

**Summary**

With this small amount of code, you have demonstrated how to deploy a basic C# application to a Docker container. The essential components are:

1. A basic C# console application (Program.cs).
2. A Dockerfile to define how the application is built and run in a Docker container.

# Lab 10d: Estate Agent Microservice – Containerising the Services Using Docker

## Objective

Your goal is to reengineer the Estate Agent services so that they each run in their own Docker container and in addition, each service should make use of a dedicated SQL server service that hosted in their own Docker containers.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## STEPS

## Launch and configure Docker Desktop.

|  |  |
| --- | --- |
| 1 | If you’ve not already used Docker before and/or don’t have a login then register with Docker at Docker.com. If you already have an account then sign in. |
| 2 | Launch Docker Desktop answering any questions and taking defaults as appropriate. |

## Add Docker Support to the Four projects.

|  |  |
| --- | --- |
| 5 | Open the Visual Studio starter solution located in the …" Labs\10 Docker Containerisation\Starter" folder. |
| 6 | Right click on the BuyerService in Visual Studio’s Solution Explorer window and select Add | Docker Support... |
| 7 | Select Linux as the Target OS and Dockerfile as the Container build type and press OK.  **Note: If you were creating a new Visual Studio project from scratch with the “ASP.NET Core Web Application” project templates you could select the Enable Docker Support” check box as part of the process.** |
| 8 | A Docker file should have been added to the project: |
| 9 | Inspect the file’s content: |
| 10 | Everything you need to do to containerise the app has been configured for you! The only thing you may want to do is change the port numbers you want the app to operate and be exposed on. However, there is a gotcha. The port numbers are for internal use only. I.E. they are exposing the app to other services that would be running in the same container. Given we are creating microservices this isn’t going to be terribly helpful to us. |
| 11 | Look at the launchSettings.json file in the BuyerService’s Properties folder. Note that a new section called “Container (Dockerfile)has been added (this was done at the same time the DockerFile was created). To be honest this new entry isn’t terribly useful to us, but it does highlight a point of potential conflict because the ports are being mentioned again that are the same “internal” ports the app will be exposing to other services running in the same container. Changing the port numbers here should be kept the same as in the DockerFile otherwise things will get confusing and we’ll be entering a game of last in wins. |
| 12 | Remember the Solution is configured (from the last lab) to launch all 4 projects at the same time. We don’t want this to happen so, right click on the BuyerService project in Visual Studio’s Solution Explorer window and select “Debug | Start New Instance. |
| 13 | Look in the Docker Desktop window and notice the buyersservice image will appear (possibly after some time) in the Images tab: |
| 14 | A short while after, if you click on the Containers tab you will see a buyerService container get up and running. If you are lucky, you may even see some port information:    However, don’t worry if the port information is missing, Docker Desktop sometimes chooses not to show it! You can always see what ports are being used by firing up either a Command (cmd) or PowerShell window and entering:  Docker ps |
| 15 | The port that is exposed to the external network is always specified to the left of the arrow (->) symbol and the first pair of ports will typically be exposed on HTTP whilst the second pair will be using HTTPS. So, in the above example you can see a client running on the same local machine could access the code running in the container by using a URL that starts with either:  <HTTP://localhost:32779/>  Or:  HTTPS://localhost:32778/ |
| 16 | If everything is running smoothly Visual Studio should have opened up a browser window exposing the BuyerService via Swagger and you will notice it is using the HTTPS port number: |
| 17 | Try testing the service by invoking the Get Buyers API function. Unfortunately, the application will fail with a 500 response. Digging into the response body you will see the issue lies around establishing a connection to the SQL Server database. |
| 18 | There are actually two issues that we need to solve. The first is we are trying to access the database using a Trusted Connection which requires a token associated with the ID of the calling process. In previous exercises that ID was the one given to you when you logged onto Windows. Unfortunately (or perhaps fortunately given the real world need for security) the ID associated with the call coming from the Docker container isn’t yours and is one that isn’t associated with the database. Fortunately, the SQL script you ran that created the database also added a user called eauser.  Locate the appsettings.json file in the BuyerService and amend the sqlestateagentdata connection string so that rather than using a Trusted\_Connection setting it passes in a user id and password:  "Server=(local);Database=EABuyer;**User Id=eauser;Password=Pa$$w0rd**;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True" |
| 19 | There’s another thing that needs to be done to resolve this first issue. In order to allow user ids and passwords SQL Server needs to be configured to allow both Windows and SQL Server authentication modes (when SQL Server is first installed it defaults to only allowing Windows authentication).  In SSMS right click on the server name at the top of the Object Explorer window and select Properties.    Click on the Security tab and select the “SQL Server and Windows Authentication mode” radio button. Then press the OK button. |
| 20 | The second issue that needs to be resolved is the “Server” setting in the connection string. When it runs the code will be trying to access the database server that is running on the host machine from within the Docker container and the current setting of (local) isn’t going to work. |
| 21 | Search Windows start for the “SQL Server Configuration Manager” and launch it. |
| 22 | Open the SQL Server Configuration Manager (Local) | SQL Server Network Configuration menu and click the Protocols for MSSQLSERVER. |
| 23 | If `TCP/IP` protocol is `Disabled` as shown below, double click the TCP/IP protocol and toggle the Enabled setting to Yes.    Press OK |
| 24 | We next need to restart SQL Server so the changes will take effect.  Go to "SQL Server Configuration Manager (Local)" | "SQL Server Services", right-click the "SQL Server (MSSQLSERVER) service" and press the `Restart` button to apply changes. |
| 25 | Now we are ready to access SQL Server from within the Docker container. The final thing we need to do is change the Server setting in the connection string (in appsettings.json) from "(local)" to "host.docker.internal". This special DNS name is helpful because it resolves to the internal IP address used by the host machine thus future proofing if the IP address of the host if (and when) it ever changes during the development process.  So, change the connection string to:  "Server=**host.docker.internal**;Database=EABuyer;User Id=eauser;Password=Pa$$w0rd;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True" |
| 26 | Right click on the BuyerService project in Visual Studio’s Solution Explorer window and select “Debug | Start New Instance. |
| 27 | If everything is running smoothly Visual Studio should have opened up a browser window exposing the BuyerService via Swagger and you will notice it is using the HTTPS port number:  A computer screen shot of a computer  Description automatically generated |
| 28 | Try testing the service by invoking the Get Buyers API function. Hopefully, all will be fine, and the Buyer data will be returned from the database.  You can try out the other API function calls to guarantee all is working. |

## Add Container Orchestrator Support

|  |  |
| --- | --- |
| 29 | Having to worry about changing container and host machine IP addresses is rapidly going to become tedious if we don't do anything about it. Fortunately, it is possible to add additional configuration files to the solution which can be used to orchestrate settings such as these. This is achieved through the use of Docker Compose. |
| 30 | Right click on the BuyerService project in Visual Studio's Solution Explorer Window and select Add | Container Orchestrator Support…  If you have a choice, select Docker Compose as the Container Orchestrator and press OK.  Then select a Target OS of Linux and click OK.  Visual Studio will add a docker-compose project to the solution. |
| 31 | The docker-compose project will contain a yaml file called docker-compose.yml. Open this now.  The file contains information that is used to define the collection of images that will be built and run whenever the docker-compose build and docker-compose run commands are invoked.  version: '3.4'  services:  buyerservice:  image: ${DOCKER\_REGISTRY-}buyerservice  build:  context: .  dockerfile: BuyerService/Dockerfile  As it stands the code specifies the name that will be given to the Docker image created for the BuyerService and also specifies the location of the service's Dockerfile. |
| 32 | If you click on the black arrow in Solution Explorer that sits to the left of the docker-compose.yml file you will reveal another file called docker-compose.override.yml.    This file is optional but is read and used by Docker and contains configuration overrides for services. In this case you can see it is specifying the internal ports of 8080 and 8081 that will be used by services running inside the container.  Using the configuration-specific override files, you can specify different configuration settings (such as environment variables or entry points) for Debug and Release build configurations.  Change the file so it specifies the use of port 3011 rather than 8080 for ASPNETCORE\_HTTP\_PORTS and **delete** the - ASPNETCORE\_HTTPS\_PORTS = 8081 and – 8081 lines:  version: '3.4'  services:  buyerservice:  environment:  - ASPNETCORE\_ENVIRONMENT=Development  - ASPNETCORE\_HTTP\_PORTS=**3011**  ports:  - "**3011**"  volumes:  - ${APPDATA}/Microsoft/UserSecrets:/home/app/.microsoft/usersecrets:ro  - ${APPDATA}/ASP.NET/Https:/home/app/.aspnet/https:ro |

Adding a SQL Server Container

|  |  |
| --- | --- |
| 33 | In the world of microservices we would consider each of our services be run in a separate container and this goes for our SQL Server databases as well. In this section we are going to edit the docker-compose.yml file to spin up a Docker container that hosts an instance of a SQL Server EABuyer database and get the BuyerService to use it.  We will also configure a network so we can specify the IP addresses that will be used by our services. |
| 34 | Add the following to the foot of the docker-compose.yml file:  networks:  EstateAgentMicroservicenetwork:  driver: bridge  ipam:  driver: default  config:  - subnet: 172.19.0.0/16  Take care with the indentations, Yaml is fussy about them. The "networks:" label should sit at the same level of indentation as the "services" label (i.e. **not** indented) |
| 35 | Edit the buyerservice entry to look like the following:  buyerservice:  image: ${DOCKER\_REGISTRY-}buyerservice  **environment:**  **- ConnectionString=sqlestateagentdata**  build:  context: .  dockerfile: BuyerService/Dockerfile  **container\_name: buyerservice**  **ports:**  **- "3011:3011"**  **depends\_on:**  **- sqlestateagentbuyerdata**  **networks:**  **EstateAgentMicroservicenetwork:**  **ipv4\_address: 172.19.10.211**  Note, the "buyerservice:" label is indented to lie within the "services" section.  Note too, we've specified an IP address that belongs to the subnet specified in the networks section.  The buyerservice also makes mention of a dependency called sqlestateagentbuyerdata which we will need to set up next. |
| 36 | Add the following to sit within the services section with the same level of indentation as the buyerservice:  sqlestateagentbuyerdata:  image: mcr.microsoft.com/mssql/server:2019-latest  environment:  - SA\_PASSWORD=PaSSw0rdPaSSw0rd  - ACCEPT\_EULA=Y  container\_name: sqlestateagentbuyerdata  ports:  - "1402:1433"  networks:  EstateAgentMicroservicenetwork:  ipv4\_address: 172.19.10.202  Note, we are specifying what the password of the new SQL Server installation within the sqlestateagentbuyerdata service will be. The SQL Server image will be pulled from Microsoft's web site. We have also specified an IP address that belongs to the subnet specified in the networks section. |
| 37 | Open the BuyerService project's appsetting.json file and amend the ConnectionStrings entry to be:  "sqlestateagentdata": "Server=**sqlestateagentbuyerdata**;Database=EABuyer;User Id=**sa**;Password=**PaSSw0rdPaSSw0rd**;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True"  Notice the server is specified as sqlestateagentbuyerdata which happens to be the name given to the name of the code we have just added to the docker-compose.yaml file and this is what links the two (we don't need to know the IP address of the database server, docker-compose will sort that out for us. |
| 38 | Given the database and its table need be created and populated within the running SQL Server container. It would be sensible to do this (when the apps are running in Development mode rather than Release mode) when they are first run and this can be done by making use of the AutoFixture external assembly.  Use NuGet to add the AutoFixture package to the BuyerService project.  Add a new class to the BuyerService's Infrastructure folder called BuyerSeeder. Add the following code to it:  using AutoFixture;  using BuyerService.Models;  namespace BuyerService.Infrastructure  {  public static class BuyerSeeder  {  public static void Seed(this BuyerContext buyerContext)  {  if (!buyerContext.Buyers.Any())  {  Fixture fixture = new Fixture();  fixture.Customize<Buyer>(buyer =>  buyer.Without(p => p.Id));  //--- The next two lines add 100 rows to your database  List<Buyer> buyers =  fixture.CreateMany<Buyer>(100).ToList();  buyerContext.AddRange(buyers);  buyerContext.SaveChanges();  }  }  }  }  The fixture.CreateMany method generates 100 Buyer objects and populates each property with Guid's. Not very real world but good enough for our requirements. |
| 39 | Next we will need to add code that runs when the web app starts that checks to see if the database exists and if not invokes the Seed() method we've just written. Remember we only want this code to run if the code is operating in Development mode so add the following lines to the if expression in Program.cs that tests to see if the app.Environment.IsDevelopment is true.  using (var scope = app.Services.CreateScope())  {  var buyerContext =  scope.ServiceProvider.GetRequiredService<BuyerContext>();  buyerContext.Database.EnsureCreated();  buyerContext.Seed();  } |
| 40 | It is time to test our code. You will have noticed when we added docker orchestration the docker-compose project was made to be the startup project. You may have also noticed the debug option in Visual Studio's toolbar has been labelled as "Docker Compose".    If you expand the dropdown options we could simply launch the app in a Web Browser but we don't want to do this, we want the Docker-Compose code to do its thing. So, click on the green triangle (or press F5).  Docker will create a dockercompose section in DockerDesktop's Containers tab. Launch the BuyerService and pull the SQL Server image from Microsoft and run it in a container (this will take some time to complete). Eventually, you should see a browser window open with the traditional Swagger options that will allow you to test the BuyerService API.  If you get any build errors then try stopping and deleting all the containers and all of the images in Docker Desktop.  Or, if you still have no joy, click on the bug symbol in DockerDesktop's menu and select the option to clean and purge data. |
| 41 | If you look at the Containers tab in Docker Desktop you will see two containers that are running on the ports that we specified in the docker-compose file (plus one that has been randomly allocated): |
| 42 | When the browser finally opens it may be using a random port on HTTPS but if you amend the url to use port 3011 on HTTP the swagger API options should still be made available to you and should still work:  <http://localhost:3011/swagger/index.html> |
| 43 | Note, even though Visual Studio is deploying and running code in Docker containers it is fully debuggable. Any break points you add to your code will be hit in the conventional manner. The only potential issue is the process will probably be a lot slower than if you were not using Docker. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

## If you have time:

|  |  |
| --- | --- |
| 45 | Repeat steps 29 to 44 (ignoring step 34) for the SellerService giving it an HTTP port number of 3013 and an IP address of 172.19.10.213.  When specifying the new database service use the following:  sqlestateagent**seller**data:  image: mcr.microsoft.com/mssql/server:2019-latest  environment:  - SA\_PASSWORD=PaSSw0rdPaSSw0rd  - ACCEPT\_EULA=Y  container\_name: sqlestateagent**seller**data  ports:  - "**1404**:1433"  networks:  EstateAgentMicroservicenetwork:  ipv4\_address: 172.19.10.**204** |
| 46 | Repeat steps 29 to 44 (ignoring step 34) for the PropertyService giving it an HTTP port number of 3012 and an IP address of 172.19.10.212.  When configuring the propertyservice entries in the docker-compose.yml file add another dependency for rabbitmq:  **depends\_on:**  **- rabbitmq**  **- sqlestateagentbuyerdata**  When specifying the new database service use the following:  sqlestateagent**property**data:  image: mcr.microsoft.com/mssql/server:2019-latest  environment:  - SA\_PASSWORD=PaSSw0rdPaSSw0rd  - ACCEPT\_EULA=Y  container\_name: sqlestateagent**property**data  ports:  - "**1403**:1433"  networks:  EstateAgentMicroservicenetwork:  ipv4\_address: 172.19.10.**203** |
| 47 | Repeat steps 29 to 44 (ignoring step 34) for the BookingService giving it an HTTP port number of 3010 and an IP address of 172.19.10.210.  When configuring the bookingservice entries in the docker-compose.yml file add another dependency for rabbitmq:  **depends\_on:**  **- rabbitmq**  **- sqlestateagentbuyerdata**  When specifying the new database service use the following:  sqlestateagent**booking**data:  image: mcr.microsoft.com/mssql/server:2019-latest  environment:  - SA\_PASSWORD=PaSSw0rdPaSSw0rd  - ACCEPT\_EULA=Y  container\_name: sqlestateagent**booking**data  ports:  - "**1401**:1433"  networks:  EstateAgentMicroservicenetwork:  ipv4\_address: 172.19.10.**201** |
| 48 | Next, we need to think about managing the delete property and add booking functionality. Remember both methods make calls to other services and we need to ensure these continue to work but somewhat strangely, in their current form they won't.  Test this out by making (via Swagger or PostMan) a call to the BookingService's Post method and notice the app crashes with a Connection refused error.  The fix is surprisingly simple rather than mentioning "localhost" in the embedded calls to the property and buyer services (that check to ensure the passed in ids are genuine) we simply need to specify the docker container\_name we specified in the docker-compose file. |
| 49 | Locate the app.MapPost("/bookings"… lambda function declared in the BookingService's Program.cs file. Then find the line that sets up the url for the call to the Property Service and replace "localhost" with "propertyservice":  string url = $"https://**propertyservice**:3012/properties/{booking.PropertyId}"; |
| 50 | Further down in the same method edit the line that sets up the url for the call to the buyer service again replacing "localhost" but this time with "buyerservice" so the line ends up as follows:  url = $"https://**buyerservice**:3011/api/buyer/buyers/{booking.BuyerId}"; |
| 51 | Relaunch the services and test the altered functionality confirming that you can now add new bookings but only if the two tests pass. |
| 52 | Now try to do something similar for the app.MapDelete("/Properties"… lambda function located in the PropertyService's Program.cs file. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 11: Estate Agent Microservice – Creating an Event Bus

## Objective

Your goal is to reengineer the Estate Agent Property and Booking services, so they don’t directly rely on each other when a property and any related bookings are deleted. Instead, we are going to create a loosely coupled relationship by implementing an event bus.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## STEPS

## Reengineer the PropertyService.

|  |  |  |
| --- | --- | --- |
| 1 | In this lab we will be working with and adapting code that focuses on the deletion of properties and any related bookings which was the very last element of the “if you have time” section of the previous lab. Consequently, unless you managed to complete the entire lab you would be best advised to open up the Visual Studio solution located in the Starter folder. | |
| 2 | Add a new class library project to the solution called Events. | |
| 3 | Rename the Class.cs file to be PropertyDeletedEvent.cs taking the Yes option to rename all references. | |
| 4 | Add a single integer property to the class called PropertyId. An instance of this class will be sent from the property service to the booking service as part of the Event Bus mechanism. | |
| 5 | Locate the PropertyService project in Solution Explorer, right click on the Dependencies tab and select “Manage NuGet Packages…” and install the following packages:   * DotNetCore.CAP * DotNetCore.CAP.Dashboard * DotNetCore.CAP.SqlServer * DotNetCore.CAP.RabbitMQ | |
| 6 | Add a Project reference from the PropertyService project to the Events project. | |
| 7 | We need to configure the service to support the use of a CAP service and RabbitMQ Message Queue. Open the project’s Program.cs file and add the following just beneath the code that configures the DbContext service.  builder.Services.AddCap(options => {  }); | |
| 8 | The Cap service needs to use a database to ensure its messages aren’t lost. We will hook it up to use the EAProducts database rather than create a whole new one.  Add the following code to the lambda:  options.UseEntityFramework<PropertyContext>();  options.UseSqlServer(  builder.Configuration.GetConnectionString(“sqlestateagentdata”)); | |
| 9 | It will be helpful to view the Cap dashboard to monitor how things are going when the app is executed so add the following lines to the lambda:  options.UseDashboard(d => {  d.AllowAnonymousExplicit = true;  }); | |
| 10 | Next, we need to get the Cap service to use RabbitMQ so add the following code to the lambda:  options.UseRabbitMQ(options =>  {  options.ConnectionFactoryOptions = options => {  options.Ssl.Enabled = false;  options.HostName = “rabbitmq”;  options.UserName = “guest”;  options.Password = “guest”;  options.Port = 5672;  };  }); | |
| 11 | Now we need to edit the code in the app.MapDelete(“/Properties”{id}… function located towards the end of the Program.cs file.  Add an ICapPublisher parameter called capPublisher to the function declaration (such that it takes three parameters). This parameter will be injected by the runtime from the service we set up at the higher up the page.  Add a using clause for DotNetCore.CAP to the list of using statements at the top of the file. | |
|  | **Delete** the four lines of code in the if expression that set up the:   * HttpClient * url * HttpResponseMessage variable * line that returns a result | |
| 12 | Replace them with the following:  PropertyDeletedEvent propertyDeletedEvent =  new PropertyDeletedEvent { PropertyId = property.Id };  var content = JsonConvert.SerializeObject(propertyDeletedEvent);  await capPublisher.PublishAsync(“PropertyDeleted”, content);  return Results.NoContent();  You will need to add a using clause for the Events namespace. | |
| 13 | Next, we need to sort out the BookingService so it responds to messages that have been added to the Event Bus’s queue.  Locate the BookingService project in Solution Explorer, right click on the Dependencies tab and select “Manage NuGet Packages…” and install the following packages:   * DotNetCore.CAP * DotNetCore.CAP.Dashboard * DotNetCore.CAP.SqlServer * DotNetCore.CAP.RabbitMQ   Add a Project reference from the BookingService project to the Events project and add “using Events;” to the list of existing using statements at the top of the file. | |
| 14 | Right click on the BookingService project in Solution Explorer and select “Add | New Folder” calling it DomainEventHandler. | |
| 15 | Add a new class to the folder calling it PropertyDeletedEventSubscriber and make it inherit ICapSubscribe. | |
| 16 | Add the following code to the class that deals with the injection of a BookingContext:  private readonly BookingContext \_bookingContext;  public PropertyDeletedEventSubscriber(BookingContext bookingContext) {  \_bookingContext = bookingContext;  } | |
| 17 | Add a new public async Task<IResult> function called Consumer that takes a string parameter called content. |
| 18 | Decorate the function with a CapSubscribe attribute passing it “PropertyDeleted” as a string:  [CapSubscribe(“PropertyDeleted”)] |
| 19 | Add the following line of code that uses JsonConvert to deserialize the content parameter into a PropertyDeletedEvent object:  var property =  JsonConvert.DeserializeObject<PropertyDeletedEvent>(content); |
| 20 | Add the following code to the method that uses the property object’s propertyId property to retrieve any relevant bookings from the database and deletes them if any are found:  var bookings = \_bookingContext.Bookings  .Where(b => b.PropertyId == property.PropertyId).ToList();  if (bookings is null || bookings.Count() == 0)  return Results.NotFound();  \_bookingContext.Bookings.RemoveRange(bookings);  \_bookingContext.SaveChanges();  return Results.NoContent(); |
| 21 | We need to configure the BookingService to use CAP so return to the PropertyService’s program.cs file and copy all the code that creates and configures the CAP service and paste it into the equivalent position in the BookingService’s Program.cs file.  Change the options.UseEntityFramework <PropertyContext>() to use BookingContext. |
| 22 | The last thing we need to in the BookingService is to add the event handler service. Add the following to the service’s Program.cs file just beneath the code you added in the previous step:  builder.Services.AddScoped<PropertyDeletedEventSubscriber>(); |
| 23 | Finally, we need to edit the docker-compose.yml and docker-compose.override.yml files in the docker-compose project to spin up a rabbitmq container and add it as a dependency to the property service and bookingservice containers. |
| 24 | Open the docker-compose.yml file and add the following service details:  rabbitmq:  image: rabbitmq:latest  restart: always  container\_name: rabbitmq  ports:  - “5672:5672”  - “15672:15672”  volumes:  - Rabbitmq\_data:/var/lib/rabbitmq  networks:  EstateAgentMicroservicenetwork:  ipv4\_address: 172.19.10.205 |
| 25 | Edit the bookingservice entry to look like the following:  bookingservice:  image: ${DOCKER\_REGISTRY-}bookingservice  **restart: on-failure**  environment:  - ConnectionString=sqlestateagentdata  **- "EventBusSettings:HostAddress=amqp://guest:guest@rabbitmq:5672"**  **depends\_on:**  **- rabbitmq**  **- sqlestateagentbookingdata**  build:  context: .  dockerfile: BookingService/Dockerfile  container\_name: bookingservice  ports:  - "3011:3011"  networks:  EstateAgentMicroservicenetwork:  ipv4\_address: 172.19.10.210 |
| 26 | Edit the propertyservice entry to look like the following:  propertyservice:  image: ${DOCKER\_REGISTRY-}propertyservice  **restart: on-failure**  environment:  - ConnectionString=sqlestateagentdata  **- "EventBusSettings:HostAddress=amqp://guest:guest@rabbitmq:5672"**  **depends\_on:**  **- rabbitmq**  **- sqlestateagentpropertydata**  build:  context: .  dockerfile: PropertyService/Dockerfile  container\_name: propertyservice  ports:  - "3012:3012"  networks:  EstateAgentMicroservicenetwork:  ipv4\_address: 172.19.10.210 |
| 27 | Add the following at the bottom of the file just before the networks section:  volumes:  rabbitmq\_data: |
| 28 | You should now be in the position where you can test your amendments and ensure the Event Bus technology is working.  Set the following breakpoints in your code:   * On the line in the Property Service’s app.MapDelete(“/Properties/{id}” function that awaits the call of the capPublisher object’s PublishAsync method call. * On the first line of code inside the Booking Service’s PropertyDeletedEventSubscriber class’s Consumer method |
| 29 | Make sure the docker-compose project is set as the startup project. Press F5 (or press the green triangle) to launch the docker-compose project in debug mode. |
| 30 | When the services are up and running use Postman or swagger pages (http://localhost:3012/swagger/index.html) to insert a new Property to the database making a note of the (property) id that gets generated.  Then insert a number of new bookings (http://localhost:3010/swagger/index.html) that use the newly generated property id and a number of genuine buyer id’s (anything between 1 and 100). |
| 31 | Finally try to delete the newly generated property (<http://localhost:3012/swagger/index.html>) and wait for the first breakpoint to be hit.  Ensure the “content” variable contains a PropertyId that has been set to the right value.  Press the green triangle to continue running the code and wait for the second breakpoint to be hit. Check to make sure the content parameter has the same Id value and step through the code to ensure the corresponding bookings have been deleted.  Press F5 to continue running the code. |
| 32 | Use swagger (or Postman) to retrieve all the properties and ensure the recently added property and all of its related bookings have been removed. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

If you have time:

Reconfigure the code in the BookingService so that when trying to add a new booking and the code checks to see if the Buyer Id and Property Id are valid (i.e. they already exist in the relevant databases) we will make use of environment variables in the docker-compose file (rather than hardcoding the service names).

|  |  |
| --- | --- |
| 33 | Open the Docker-compose file and locate the bookingservice section. |
| 34 | Add the following lines to the environment section (just beneath the ConnectionString variable):  PROPERTYSERVICE=http://propertyservice:3012  BUYERSERVICE=http://buyerservice:3011 |
| 35 | Add the following lines to the depends\_on section (beneath the two entries that are currently there):  - propertyservice  - buyerservice |
| 36 | Open the BookingService’s Program.cs file and locate the app.MapPost(“/bookings”,.. lambda function. |
| 37 | Replace the line that declares and sets the url variable with the following:  string PROPERTYSERVICE =  Environment.GetEnvironmentVariable("PROPERTYSERVICE");  //Check to see if PropertyId is valid  string url = "";  if (PROPERTYSERVICE == null)  url = $"http://propertyservice:3012/properties/{booking.PropertyId}";  else  url = $"{PROPERTYSERVICE}/properties/{booking.PropertyId}"; |
| 38 | Do something similar for the code that sets up the url to the BuyerService. |
| 39 | Test your program. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 12: Estate Agent Microservice – Kubernetes

## Objective

Your goal is to use Kubernetes to configure the Estate Agent Microservices ready for deployment.

## Overview

In this lab you will configure the image names in docker-compose.yml file to prepare them for deployment to Docker Hub. Then you will create a set of YAML files, one for each service, with configuration settings that will be used in a Kubernetes deployment. You will then invoke the relevant kubectl commands to carry out the deployment. Finally you will test the endpoints to ensure everything works as expected.

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## STEPS

## Upload Docker Images to Docker Hub

|  |  |
| --- | --- |
| 1 | Make sure you are signed in to Docker Hub. You can do this from within Docker Desktop by clicking the Sign in button on the top right of the window. |
| 2 | Either open your EstateAgentBackEnd Visual Studio solution to the previous lab or use the solution found in this lab’s Starter folder. |
| 3 | Open the docker-compose.yml file and locate the image name setting for the bookingservice. Note it’s currently set to the following:  image: ${DOCKER\_REGISTRY-}bookingservice  Replace the ${DOCKER\_REGISTRY-} text with your docker username followed by a forward slash (/). **Note, docker is case sensitive to make sure you use the correct casing**.  For example, if your docker username is amynonymous then the line should look like the following:  image: amynonymous/bookingservice |
| 4 | Repeat step 3 for each of the other services (buyer, property and seller). |
| 5 | Make sure the solution has been built, the docker images and containers exist (use Docker Desktop or run “docker ps” from a command window or PowerShell. |
| 6 | Open a command or PowerShell window.  Run the following instruction:  Docker images  You should see something like the following: |
| 7 | Before pushing your images to Docker Hub you first have to change their tags from “dev” to “latest”.  For **each image in turn** enter and run the following. Remember docker is case sensitive so make sure you use the correct casing for your username:  docker tag [YOUR DOCKER USER NAME]/[ [IMAGE NAME]:[TAG] [YOUR DOCKER USER NAME]/[IMAGE NAME]  For example, if your docker username is amynonymous, an image name of bookingservice and a tag of dev, then type the following:  docker tag amynonymous/buyerservice:dev amynonymous/buyerservice |
| 8 | Upload the images to Docker Hub. You can do this in one of two ways:   * From the Images tab within Docker Desktop in the Actions column select the 3 vertical dots for each of your images (you **don’t** need to do this for the rabbitmq or sql server images) and select “Push to Docker Hub” * Within a cmd or PowerShell window and type the following:   docker push [YOUR DOCKER USER NAME]/[IMAGE NAME]  Repeat for each image |
| 9 | Browse to hub.docker.com to ensure all the images have arrived safely. |

## Some Housekeeping

|  |  |
| --- | --- |
| 10 | Before we get stuck into Kubernetes there’s a little bit of housekeeping that needs to be done.  Each of the services has some code that tests to see if the code is running in a “Development” mode and sets up the dummy database data only if this is true. When the code gets deployed to a Kubernetes environment it will be considered to be operating in “Release” mode. Unfortunately, this means the databases will be empty. Rather than waste time trying to sort out sets of more genuine looking data we are going to be lazy.  For each of the four service's Program.cs files, locate the and cut the following lines of code from within the if (app.Environment.IsDevelopment()) expression and paste it just beneath the if’s closing curly brace so that it will always be invoked:  if (app.Environment.IsDevelopment())  {  app.UseSwagger();  app.UseSwaggerUI();  }  **using (var scope = app.Services.CreateScope())**  **{**  **var bookingContext =**  **scope.ServiceProvider.GetRequiredService<BookingContext>();**  **bookingContext.Database.EnsureCreated();**  **bookingContext.Seed();**  **}** |

## Deploy the microservice containers to Kubernetes

|  |  |
| --- | --- |
| 11 | In order to use Kubernetes within Docker Desktop you will need to do a bit of configuration. We have already done the first step for you by adding the relevant Kubernetes extensions to Docker Desktop, **but we haven’t enabled it**. To do this open Docker Desktop and click on the Settings button (the cog symbol top right).  Select the Kubernetes tab and check the Enable Kubernetes check box.  Press Apply & restart. The installation will take a few minutes, so while you are waiting you can get on with the following steps: |
| 12 | When deploying to Kubernetes, you will need to create a new set of YAML files that are different from the Docker-Compose files. This is necessary because Kubernetes and Docker Compose are fundamentally different orchestration platforms with distinct features, paradigms, and configurations. Both sets of files do similar things (specify container names, port numbers, environment variables…) However, the Kubernetes YAML can also be used to handle not only deployment but also scaling, self-healing, service discovery, and rolling updates. Note, we won’t be doing much of this because it’s beyond the scope of this course. |
| 13 | Add a new folder to the docker-compose project called K8s. |
| 14 | Add a new item to the new folder called buyerservice-deploy.yml |
| 15 | Add the following code to the file replacing the [your-name-here] section with you docker username name:  ---  apiVersion: apps/v1  kind: Deployment  metadata:  name: buyerservice  spec:  replicas: 1  template:  metadata:  labels:  app: buyerservice  spec:  containers:  - name: buyerservice  image: **[your-name-here]**/buyerservice:latest  imagePullPolicy: Always  ports:  - containerPort: 3011  env:  - name: ASPNETCORE\_ENVIRONMENT  value: "Production"  - name: ASPNETCORE\_URLS  value: http://\*:3011  - name: ConnectionStrings\_\_sqlestateagentdata  value: "Server=sqlestateagentbuyerdata;Database=EABuyer;User Id=sa;Password=PaSSw0rdPaSSw0rd;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True"  selector:  matchLabels:  app: buyerservice  ---  apiVersion: v1  kind: Service  metadata:  name: buyerservice  spec:  type: NodePort  ports:  - protocol: TCP  port: 3011  targetPort: 3011  nodePort: 32011  selector:  app: buyerservice  The code is split into two parts which define two different Kubernetes resources:   * Deployment: This section is responsible for managing the application pods, including how many replicas should run, which Docker image to use, and the environment variables for the application. The key elements of a Deployment are:   + Replicas which specifies how many instances of the pod should run.   + Template describes the pods, including their labels, the container image to use, and the ports to expose inside the container.   + Env defines environment variables for the application. * Service: This section exposes the application to the network, allowing other services or external users to access it via a specific port. The key elements of a Service are:   + type: NodePort: Exposes the service on a static port on each node in the cluster.   + ports: Defines the ports on which the service is exposed.   + port: The port the service exposes inside the cluster.   + targetPort: The port on the pod that the service will forward traffic to.   + nodePort: The port on the node that will be used to access the service from outside the cluster.   + selector: Specifies which pods the service should forward traffic to by matching the labels defined in the Deployment.   You will notice how the database’s connection string has been specified as an environmental variable. |
| 16 | Add a new item to the K8s folder called sellerservice-deploy.yml |
| 1617 | Add the following code to the file (again replacing the [your-name-here] section with your Docker username:  ---  apiVersion: apps/v1  kind: Deployment  metadata:  name: sellerservice  spec:  replicas: 1  template:  metadata:  labels:  app: sellerservice  spec:  containers:  - name: sellerservice  image: **[your-name-here]**/sellerservice:latest  imagePullPolicy: Always  ports:  - containerPort: 3013  env:  - name: ASPNETCORE\_ENVIRONMENT  value: "Production"  - name: ASPNETCORE\_URLS  value: http://\*:3013  - name: ConnectionStrings\_\_sqlestateagentdata  value: "Server=sqlestateagentsellerdata;Database=EASeller;User Id=sa;Password=PaSSw0rdPaSSw0rd;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True"  selector:  matchLabels:  app: sellerservice  ---  apiVersion: v1  kind: Service  metadata:  name: sellerservice  spec:  type: NodePort  ports:  - protocol: TCP  port: 3013  targetPort: 3013  nodePort: 32013  selector:  app: sellerservice  There’s no surprise that it is very similar to the buyerservice-deploy content. |
| 18 | Add a new item to the K8s folder called bookingservice-deploy.yml |
| 19 | Add the following code to the file (don't forget to change the [your-name-here] section to your Docker username):  ---  apiVersion: apps/v1  kind: Deployment  metadata:  name: bookingservice  spec:  replicas: 1  template:  metadata:  labels:  app: bookingservice  spec:  containers:  - name: bookingservice  image: **[your-name-here]**/bookingservice:latest  imagePullPolicy: Always  ports:  - containerPort: 3010  env:  - name: ASPNETCORE\_ENVIRONMENT  value: "Production"  - name: ASPNETCORE\_URLS  value: http://\*:3010  - name: ConnectionStrings\_\_sqlestateagentdata  value: "Server=sqlestateagentbookingdata;Database=EABooking;User Id=sa;Password=PaSSw0rdPaSSw0rd;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True"  - name: PROPERTYSERVICE  value: http://propertyservice:3012  - name: BUYERSERVICE  value: http://buyerservice:3011  selector:  matchLabels:  app: bookingservice  ---  apiVersion: v1  kind: Service  metadata:  name: bookingservice  spec:  type: NodePort  ports:  - protocol: TCP  port: 3010  targetPort: 3010  nodePort: 32010  selector:  app: bookingservice |
| 20 | Add a new item to the K8s folder called propertyservice-deploy.yml |
| 21 | Add the following code to the file (changing the [your-name-here] section to your Docker username:  ---  apiVersion: apps/v1  kind: Deployment  metadata:  name: propertyservice  spec:  replicas: 1  template:  metadata:  labels:  app: propertyservice  spec:  containers:  - name: propertyservice  image: **[your-name-here]**/propertyservice:latest  imagePullPolicy: Always  ports:  - containerPort: 3012  env:  - name: ASPNETCORE\_ENVIRONMENT  value: "Production"  - name: ASPNETCORE\_URLS  value: http://\*:3012  - name: ConnectionStrings\_\_sqlestateagentdata  value: "Server=sqlestateagentpropertydata;Database=EAProperty;User Id=sa;Password=PaSSw0rdPaSSw0rd;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True"  selector:  matchLabels:  app: propertyservice  ---  apiVersion: v1  kind: Service  metadata:  name: propertyservice  spec:  type: NodePort  ports:  - port: 3012  targetPort: 3012  nodePort: 32012  selector:  app: propertyservice |
| 22 | Add a new item to the K8s folder called  sqlestateagentbuyerdata-deploy.yml |
| 23 | Add the following code to the file:  apiVersion: apps/v1  kind: Deployment  metadata:  name: sqlestateagentbuyerdata  spec:  replicas: 1  selector:  matchLabels:  app: sqlestateagentbuyerdata  template:  metadata:  labels:  app: sqlestateagentbuyerdata  spec:  containers:  - name: sqlestateagentbuyerdata  image: mcr.microsoft.com/mssql/server:2019-latest  ports:  - containerPort: 1433  env:  - name: ACCEPT\_EULA  value: "Y"  - name: SA\_PASSWORD  value: "PaSSw0rdPaSSw0rd"  ---  apiVersion: v1  kind: Service  metadata:  name: sqlestateagentbuyerdata  spec:  type: NodePort  selector:  app: sqlestateagentbuyerdata  ports:  - protocol: TCP  port: 1433  targetPort: 1433  name: tcpsql  Standard configuration for a SQL server database pod. |
| 24 | Add a new item to the K8s folder called  sqlestateagentsellerdata-deploy.yml |
| 25 | Add the following code to the file:  apiVersion: apps/v1  kind: Deployment  metadata:  name: sqlestateagentsellerdata  spec:  replicas: 1  selector:  matchLabels:  app: sqlestateagentsellerdata  template:  metadata:  labels:  app: sqlestateagentsellerdata  spec:  containers:  - name: sqlestateagentsellerdata  image: mcr.microsoft.com/mssql/server:2019-latest  ports:  - containerPort: 1433  env:  - name: ACCEPT\_EULA  value: "Y"  - name: SA\_PASSWORD  value: "PaSSw0rdPaSSw0rd"  ---  apiVersion: v1  kind: Service  metadata:  name: sqlestateagentsellerdata  spec:  type: NodePort  selector:  app: sqlestateagentsellerdata  ports:  - protocol: TCP  port: 1433  targetPort: 1433  name: tcpsql |
| 26 | Add a new item to the K8s folder called  sqlestateagentbookingdata-deploy.yml |
| 27 | Add the following code to the file:  apiVersion: apps/v1  kind: Deployment  metadata:  name: sqlestateagentbookingdata  spec:  replicas: 1  selector:  matchLabels:  app: sqlestateagentbookingdata  template:  metadata:  labels:  app: sqlestateagentbookingdata  spec:  containers:  - name: sqlestateagentbookingdata  image: mcr.microsoft.com/mssql/server:2019-latest  ports:  - containerPort: 1433  env:  - name: ACCEPT\_EULA  value: "Y"  - name: SA\_PASSWORD  value: "PaSSw0rdPaSSw0rd"  ---  apiVersion: v1  kind: Service  metadata:  name: sqlestateagentbookingdata  spec:  type: NodePort  selector:  app: sqlestateagentbookingdata  ports:  - protocol: TCP  port: 1433  targetPort: 1433  name: tcpsql |
| 28 | Add a new item to the K8s folder called  sqlestateagentpropertydata-deploy.yml |
| 29 | Add the following code to the file:  apiVersion: apps/v1  kind: Deployment  metadata:  name: sqlestateagentpropertydata  spec:  replicas: 1  selector:  matchLabels:  app: sqlestateagentpropertydata  template:  metadata:  labels:  app: sqlestateagentpropertydata  spec:  containers:  - name: sqlestateagentpropertydata  image: mcr.microsoft.com/mssql/server:2019-latest  ports:  - containerPort: 1433  env:  - name: ACCEPT\_EULA  value: "Y"  - name: SA\_PASSWORD  value: "PaSSw0rdPaSSw0rd"  ---  apiVersion: v1  kind: Service  metadata:  name: sqlestateagentpropertydata  spec:  type: NodePort  selector:  app: sqlestateagentpropertydata  ports:  - protocol: TCP  port: 1433  targetPort: 1433  name: tcpsql |
| 30 | Add a new item to the K8s folder called  rabbitmq-deploy.yml |
| 31 | Add the following code to the file:  apiVersion: apps/v1  kind: Deployment  metadata:  name: rabbitmq  spec:  replicas: 1  selector:  matchLabels:  app: rabbitmq  template:  metadata:  labels:  app: rabbitmq  spec:  containers:  - name: rabbitmq  image: rabbitmq:latest  ports:  - containerPort: 5672  - containerPort: 15672  ---  apiVersion: v1  kind: Service  metadata:  name: rabbitmq  spec:  ports:  - name: amqp  port: 5672  protocol: TCP  targetPort: 5672  - name: http  port: 15672  protocol: TCP  targetPort: 15672  selector:  app: rabbitmq |
| 31 | Now we are ready to start deploying our services to Kubernetes. Here is an overview of the steps we will need to follow:   * For clarity of demo, it will be helpful to stop and delete any containers that are currently running in Docker. You can do this from the Containers tab in Docker Desktop. * Make sure you are signed in to both:   + Docker Desktop   + <Https://Hub.Docker.Com> via a browser * Carry out a docker compose build to ensure we have a current set of Docker images. * Carry out a docker compose push to push the built Docker images to a remote container registry. In this case Docker Hub. * Create resources in a Kubernetes cluster based on the configuration specified in the (Kubernetes) YAML files * Check to see if all the pods are up and running * Check to see the details of the running services * Expose the service ports to the outside world * Browse to the service endpoints and ensure everything is working.   We will action the steps now: |
| 32 | Stop and delete any containers that are currently running in Docker Desktop by clicking on the Containers tab, selecting all the containers and clicking the Delete button. |
| 33 | Make sure you are signed into Docker Desktop by clicking the Sign In option in the Docker Desktop menu in the top right of the window (Note, this signs you in to app.Docker.com) |
| 34 | Browse to <Https://Hub.Docker.com> and sign in. Note, whilst your credentials will be the same as the ones you used in the previous step, you are signing into a different site. |
| 35 | Open a PowerShell (or command) window and browse to the folder where your Visual Studio solution is located by entering something like:  cd “C:\A Folder\Another Folder\...\12 Kubernetes\Starter” |
| 36 | Enter the following line of code:  docker compose build  This command processes each service defined in the docker-compose.yml file that has a build section. For each service, Docker Compose reads the corresponding Dockerfile and uses it to build a Docker image. Each image will be tagged and stored in the local Docker environment.  This process is carried out automatically by Visual Studio whenever you run your service applications when the start up project configured to be docker-compose meaning that, in your case, it is probably unnecessary given you have already got a complete set of Docker images stored in the Docker environment. |
| 37 | Once the previous step has completed. Enter the following line of code:  docker compose push  This command is used to push built Docker images to a remote registry (in this case Docker Hub). It’s useful to do this when you want to share your images with others or deploy them to a remote environment.  The command pushes the Docker images for each service defined in the docker-compose.yml file that specifies an image tag. The images are uploaded to the remote registry specified in the image name, such as myregistry.com/myproject/myimage. If no registry is mentioned it defaults to Docker Hub. |
| 38 | Via a browser, have a look at <https://hub.docker.com> and the Repositories tab. You should see copies of all the relevant images (yourname/bookingservice, yourname/propertyservice, yourname/buyerservice, yourname/sellerservice). There will be no mention of the SQL Server or rabbitmq services because they already exist in other remote container registries. |
| 39 | The next step is to create resources in a Kubernetes cluster based on the configuration specified in the YAML files we’ve created in the K8s folder.  Before we do this, we will create a namespace for our clusters to go into. Enter the following line and press enter:  kubectl create namespace kd  Note, the name kd has no significance and was chosen as a shorthand for **K**ubernetes and **D**ocker. |
| 40 | We are now in a position to create the Kubernetes cluster resources by using the Kubernetes kubectl apply command. You can do this on a file-by-file basis or make use of the command’s -f parameter to create cluster resources from multiple YAML files. Enter the following:  kubectl apply -n kd -f ./K8s/  (Note, the -n kd ensures the clusters are created in the kd namespace  Hopefully everything will run cleanly and if you look at the Containers tab in Docker Desktop you will see a set of containers that are in the process of getting up and running: |
| 41 | Somewhat confusingly you may see round about double the number of containers you were expecting some with the letters “POD” towards the start of their names. This is likely to be happening because when you run Kubernetes on Docker Desktop, each Pod typically consists of at least one container, but the underlying infrastructure in Docker Desktop might show additional containers that support the Pod's operation. The actual application containers will be the ones that don’t contain “POD” in their names and these correspond directly to the containers defined in the Kubernetes YAML files. The containers that have “POD” in their names are probably “pause” containers. Kubernetes uses these containers as a “parent” container for the Pod. The pause container is responsible for holding the network namespace and other shared resources for the actual application containers in the Pod. Even though they don't run your application code, they are essential for maintaining the Pod's state, networking, and other infrastructure-level tasks. In Kubernetes, each Pod shares the same network namespace. The pause container helps establish this shared namespace, allowing all containers in the Pod to communicate over localhost. |
| 41 | The kubectl get command allows us to retrieve information about a number of different Kubernetes resources, including: pods, deployments, services, nodes, namespaces, endpoints and many more. |
| 42 | Invoke kubectl -n kd get pods to see information about the pods. You should see something like the following:    Pay particular attention to the Ready and Status columns. If the Ready column contains “0/1” it means the pod isn’t yet properly up and running (a “1/1” indicates the pod is ready to be used. The Status column should contain “Running“, if all is well but may contain other values that provide insights into a pods health and readiness e.g. Pending, Failed, Succeeded.  The Restarts column often looks slightly concerning because if a value is greater than zero it suggests something went wrong when a service was starting up. In the screenshot above you can see every service restarted at least twice. There is no need to worry about this if everything stabilises there are often dependencies where one service needs another to be running. If a dependency is not available, the service may end and then be restarted. This process can occur multiple times until things sort themselves out. |
| 43 | Invoke kubectl -n kd get services to see information about the services. You should see something like the following:    The interesting thing here is the IP addresses that have been assigned to the services and the ports on which they are exposed. The lefthand port number is the one that is exposed to the outside world whilst the one on the right is the NodePort which is typically exposed to the other pods. So, in the example above the exposed port for the bookingservice is 3010 and its NodePort is 32010.  You will notice that none of the services have been given an external IP address. This would make sense if they were meant to be exposed to a local web UI application that will be the exposed to the world. However, if we wanted to expose them as an API to external clients all we would have to do is change the service type in the YAML files from “NodePort” to “LoadBalancer”. Unfortunately, in the current set up we will not be able to do this because we are using local Kubernetes where external load balancers are not available to us.  There is a simple workaround which is to browse to localhost with the NodePort being specified as the port.  If you really want to test the services using the “external” port number then we will tackle this in the “If you have time” section. |
| 44 | Open a browser window and enter:  [http://localhost:<Node](http://localhost:%3cNode) Port for the bookingservice>/bookings  e.g.  http://localhost:32010/bookings  Press enter and verify appropriate data is returned from the service. |
| 45 | TROUBLE SHOOTING  If you find the services are unavailable, you may find the following instructions of use:  Perhaps the best tactic is to take the nuclear option and go back to first principles by:   * Deleting the entire content of the Kubernetes' namespace (see below). * Deleting all the Docker images you uploaded to Hub.docker.com by clicking on each service, selecting Settings and scrolling down to the Delete repository option. * Stop and Delete any running containers in Docker Desktop. * Make sure everything in Visual Studio has been properly saved. * Running all the steps from step 36 onwards.   We are trying to run a load of services and pods hosted in Docker containers. These can eat up considerable amounts of computer memory and processor power. You could consider only deploying a single service (e.g. the buyerservice along with the associated sqlestateagentbuyerservice). You can do this by commenting out all the code in the respective yml files located in the K8s folder.  To delete a service:  kubectl delete service bookingservice  To delete a pod (note deleting a pod will NOT delete any services associated with it)  kubectl delete pod bookingservice-84bcbb7648-db2bc  Note your pod names  To delete the entire content of a Kubernetes namespace:  kubectl delete all -n kd –all  To delete a namespace and everything in it:  kubectl delete namespace kd |
| 46 | Open up Postman and run a few tests against the deployed services to ensure they support full CRUD capability.  Pay particular attention to the deletion of properties that have associated Bookings (you’ll need to set this test up in a similar way to how you did it in a previous lab (adda new property making a note of the generated ID and then add some new bookings for using the id as the propertyid).  Also ensure Bookings can only be added for existing properties and buyers. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

If you have time:

Configure the services to be exposed on the required port number.

|  |  |
| --- | --- |
|  | If you want the services to be exposed on the port numbers specified in the YAML files but are unable to use specify a service type of LoadBalancer then you can use a technique known as port forwarding. You will need to start up four separate instances of PowerShell if you want to apply this to all of the ASP.NET API services.  For each service (in a separate PowerShell instance) run the following:  kubectl port-forward -n kd pods/<pod name> <external port>:<internal port>      So, for the bookingservice above the command would look like the following:  kubectl port-forward -n kd pods/bookingservice-84bcbb7648-6j972 3010:3010  The port that will be exposed via localhost is the one highlighted in green.  Browsing to <http://localhost:3010/bookings> should bring back the appropriate data. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

# Lab 13: Estate Agent Microservice – Adding Jason Web Token (JWT) Security

## Objective

Your goal is to alter the logic of the Estate Agent Microservices SellerService project so that none of its methods can be called unless the user’s credentials have been authenticated. Proof of successful authentication will be demonstrated in the passing and receival of a Jason Web Token (JWT).

**GITHUB**

Before starting on the lab please think seriously about using GitHub as a repository for the code.

## STEPS

## Install the necessary NuGet Package

|  |  |
| --- | --- |
| 1 | Open the SellerService project in Visual Studio. You can use your solution to the previous lab or use the provided starter project. Note, given we are only worrying about providing JWT security to the Seller Service we have commented out references to all of the services in the docker-compose.yml and docker-compose.override.yml files except for the sellerservice and sqlestateagentsellerdata services. We have done this to make things run a little quicker but there is no need to do this if you don’t want to.  Note, if you use our starter code you will need to update the [your-name-here] sections replacing them with your Docker username. |
| 2 | Use NuGet to install the Install the Microsoft.AspNetCore.Authentication.JwtBearer package, which will allow the service to authenticate requests using JWT |

## Configure JWT Authentication

|  |  |
| --- | --- |
| 3 | Locate Program.cs file and add the following using directives to the top of the file:  using Microsoft.AspNetCore.Authentication.JwtBearer;  using Microsoft.IdentityModel.Tokens;  using Microsoft.IdentityModel.Tokens.Jwt;  using System.Text; |
| 4 | Add the JWT authentication configuration to the builders Services collection. Place this just beneath the AddDBContext code:  builder.Services.AddAuthentication(options =>  {  options.DefaultAuthenticateScheme =  JwtBearerDefaults.AuthenticationScheme;  options.DefaultChallengeScheme =  JwtBearerDefaults.AuthenticationScheme;  }).AddJwtBearer(options =>  {  options.TokenValidationParameters = new  Microsoft.IdentityModel.Tokens.TokenValidationParameters  {  ValidateIssuer = true,  ValidateAudience = true,  ValidateLifetime = true,  ValidateIssuerSigningKey = true,  ValidIssuer = builder.Configuration["Jwt:Issuer"],  ValidAudience = builder.Configuration["Jwt:Audience"],  IssuerSigningKey = new  SymmetricSecurityKey(Encoding.UTF8.GetBytes(  builder.Configuration["Jwt:Key"]))  };  });  builder.Services.AddAuthorization();  The first section of code builder.Services.AddAuthentication is registering the authentication services with the Dependency Injection container. The two lines of code within this section specify the default scheme the application will use for authentication (JWT Bearer tokens in this case) and the default scheme to use when a challenge is issued. Challenges occur when an unauthenticated user tries to access a resource that requires authentication. In this case the code will again be using JWT Bearer tokens)  The second section which starts with a call to AddJwtBearer()configures the JWT Bearer authentication scheme, providing options for how tokens should be validated. The TokenValidationParameters object contains settings that dictate how the incoming JWT tokens should be validated.   * ValidateIssuer = true: Ensures that the token’s issuer claim matches the expected issuer, as defined in the ValidIssuer parameter. The issuer is the entity that issued the token, usually your application or organization. * ValidateAudience = true: Ensures that the token’s audience claim matches the expected audience, as defined in the ValidAudience parameter. The audience will be the recipients of the token (e.g., your API). * ValidateLifetime = true: Ensures that the token has not expired. It checks the expiration claims of the token to ensure that it’s still valid in terms of time. * ValidateIssuerSigningKey = true: Ensures that the token’s signature is valid and that it was signed using the correct key. The signing key is defined by the IssuerSigningKey parameter. * ValidIssuer = builder.Configuration["Jwt"]: Specifies the expected issuer of the token, retrieved from your configuration (appsettings.json). The token’s issuer claim must match this value. * ValidAudience = builder.Configuration["Jwt"]: Specifies the expected audience of the token, also retrieved from your configuration. The token’s audience claim must match this value. * IssuerSigningKey = new SymmetricSecurityKey(   Encoding.UTF8.GetBytes(builder.Configuration["Jwt"]))  This sets the key that will be used to validate the token's signature. The key is created from a string stored in your configuration and is converted to a byte array using UTF-8 encoding. |

## Create a User Model and Service

|  |  |
| --- | --- |
| 5 | Add a User class to the Models folder. This will be used to represent the credentials of individual users of the system.  namespace SellerService.Models  {  public class User  {  public string? UserName { get; set; }  public string? Password { get; set; }  public string? Role { get; set; }  }  } |
| 6 | For simplicity we will not use a genuine database to host our users. Instead, we will create a simple “in-memory” user store. Converting it to be a genuine database can be an “If you have time” task.  Add a new static class called UserStore to the Infrastructure folder.  Namespace SellerService.Infrastructure{  public static class UserStore  {  public static List<User> Users = new()  {  new User {  Username = "Ady Admin",  Password = "PaSSw0rd",  Role = "Admin"  },  new User {  Username = "Kamran Senhadi",  Password = "PaSSw0rd",  Role = "Clerk"  }  };  }  } |

## Create an Endpoint for Login and Generation of JWT’s

|  |  |
| --- | --- |
| 7 | Add an app.MapPost function to Program.cs siting it at the bottom of the listing just before the app.Run() line. Make it take a User object called loginUser as a parameter and use a lambda notation into which we will add further code.  app.MapPost("/login", (User loginUser) =>  {  }); |
| 8 | Add a line to the function that uses a FirstOrDefault method call to see if the passed in loginUser object’s credentials can be found in the “database”.  User user = UserStore.Users.FirstOrDefault(  u => u.Username == loginUser.Username  && u.Password == loginUser.Password); |
| 9 | Follow this with a test to see if the user variable is null. If it is return Results.Unauthorized() from the function. |
| 10 | Create a new List of Claim objects called claims. Populate it with two new Claim objects passing the following to their constructors:   * ClaimTypes.Name, user.UserName * ClaimTypes.Role, user.Role   Claims are key-value pairs that represent data about the user. They are embedded within the JWT and can be used by the server to authorize the user. These claims will be part of the JWT's payload and can be used by the server to identify and authorize the user. |
| 11 | Create a new SymmetricSecurityKey object called key passing the following to its constructor.  SymmetricSecurityKey key = new SymmetricSecurityKey(  Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]));  A SymmetricSecurityKey represents the secret key used to sign the JWT. It ensures that the token can be validated by the server. The key (a string) gets converted to a byte array via the call to Encoding.UTF8.GetBytes. Its value is stored in the project’s appsettings.json file and should be kept secret and should be a strong, random key to ensure security. We will set this up in a later step. |
| 12 | Create a new SigningCredentials object called creds passing key and SecurityAlgorithms.HmacSha256 to its constructor.  SigningCredentials creds =  new SigningCredentials(key, SecurityAlgorithms.HmacSha256);  A SigningCredentials object represents the credentials (i.e., the security key and algorithm) used to create the digital signature of the JWT. The signature ensures the token hasn't been tampered with and verifies the authenticity of the token. In this example we are using the HmacSha256 hashing algorithm that uses the key to create the signature. |
| 13 | We have now created or have access to all of the values needed to create our JWT security token. Its constructor needs to be given the following information:   * issuer: Typically, the URL of your application or organization. This can come from an entry in appsettings.json. * audience: The intended recipient of the token. Again, this is typically the URL of your API or service which can be stored alongside the listener information in appsettings.json. * claims: This includes the claims collection that was created earlier. These claims are embedded in the payload of the JWT and provide information about the user. * expires: sets the expiration time of the token. In this example we are setting it to expire 30 minutes after it has been issued. After this time, the token will be invalid, and the user will need to obtain a new one. * signingCredentials: This includes the credentials used to sign the token, created earlier. It will ensure the token is securely signed and can be verified by the server.   JwtSecurityToken token = new JwtSecurityToken(  issuer: builder.Configuration["Jwt:Issuer"],  audience: builder.Configuration["Jwt:Audience"],  claims: claims,  expires: DateTime.Now.AddMinutes(30),  signingCredentials: creds); |
| 14 | Add the following code to the foot of the function:  return Results.Ok(new  {  token = new JwtSecurityTokenHandler().WriteToken(token)  });  The code is returning a response with a status code of “200 OK” that contains a JSON object with a JWT token. The new { token = …} section creates an anonymous object with a single property called token. This object is serialized to JSON and will end up looking something like the following:  {  "token": "your\_jwt\_token\_here"  }  new JwtSecurityTokenHandler().WriteToken(token) creates a JWT security token handler object whose WriteToken() method takes a JwtSecurityToken (created earlier) and serializes it into a compact, URL-safe string format. |

## Protect the relevant API Endpoints with JWT Authentication

|  |  |
| --- | --- |
| 15 | Add protection to **ALL** the other Seller Service endpoints so that they require authentication. For example:  app.MapGet("/sellers", async (SellerContext db) =>  await db.Sellers.ToListAsync())**.RequireAuthorization()**; |

## Configure JWT Settings in “appsettings.json”

|  |  |
| --- | --- |
| 16 | Open the appsettings.json file and add the following JWT settings:  {  "ConnectionStrings": {  "sqlestateagentdata": "Server=sqlestateagentsellerdata;Database=EASeller;User Id=sa;Password=PaSSw0rdPaSSw0rd;MultipleActiveResultSets=true;Encrypt=False;TrustServerCertificate=True"  },  "Logging": {  "LogLevel": {  "Default": "Information",  "Microsoft.AspNetCore": "Warning"  }  }**,**  **"Jwt": {**  **// The key should be a long, random string such as a GUID**  **"Key":**  **"Yh2k7QSu4l8CZg5p6X3Pna9L0Miy4D3Bvt0JVr87UcOj69Kqw5R2Nmf4FWs03Hdx",**  **"Issuer": "QA.com",**  **"Audience": "QA.com"**  **}**,  "AllowedHosts": "\*"  } |

## Test the program

|  |  |
| --- | --- |
| 17 | Launch the app using docker-compose as the startup project. Use Postman (or equivalent tool) to test the logic. |
| 18 | First, perform a GET request to /sellers and ensure your request is rejected with a 401 Unauthorized message. |
| 19 | Second, perform a POST request to /login with the following JSON body:  {  "username": "Ady Admin",  "password": "PaSSw0rd"  }  Note, your port number **may not** be the same as the one used in the screenshot below:    If the credentials are valid, the API will return a JWT token. |
| 20 | Use the returned token to access the secure endpoints:   * Copy the returned JWT token. Everything inside the quotes but **not** the word “token” or the colon * Go to the "Authorization" tab.      * Choose "Bearer Token" from the dropdown. * Paste the JWT token into the field provided. * Enter the appropriate URL into the “URL” box. Note, your port number **may not** be the same as the one used in the screenshot      * Press send     Observe the function works and returns appropriate data. |

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

If you have time:

Add JWT security to the other services and/or test the security works for a Kubernetes deployment.

|  |  |
| --- | --- |
| 21 | Rework the Seller Service JWT security to use a SQL server database that gets deployed with the estateagent seller database. |
| 22 | Add JWT security to some (or all) of the other services. |
| 23 | Try doing a Kubernetes deployment and confirm the JWT security features work as expected. |

Note, the model solutions do not cover any of the above “If you have time” ideas.

**GITHUB**

Before moving on don't forget to commit and push your work to GitHub.

A blue and green arrow

Description automatically generated

V1.0