# SOLID Principles

**Exercise: Implement an inventory system for a bookstore**

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.

**Objective**

Your goal is to refactor code so that adheres to SOLID principles.

**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).

# Coding Patterns

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

**Background**

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**.

**Objective**

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

**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.
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**.
   * Include methods for adding and removing vehicles from groups.
   * 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.

# 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.

Asynchronous Programming and Concurrency

**Exercise: Word Prefixes**

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

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.

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 stems size (2 chars, 3 chars, up to 28 chars).

**Objective**

Your goal is to refactor code so that adheres to SOLID principles.

**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 Initial Steps – 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 occurences in the data). Declare a variable of this type called *val* at the top of the function.
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.

1. 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 genereated in the loop (each Task will tackle a different stem length).
2. 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.
3. 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 liklihood the stemSize variable (which is driving the For loop in the FindStems function) will have changed.
4. As the next line of code in the loop declare a Task<Tuple<int, string, int>> varaible 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.
5. Add task to the tasks collection.
6. Beneath the for loop, add code that Waits until all the tasks have finished by writing:

Task.WhenAll(tasks).Wait();

1. Next, add code that iterates around the tasks collection (tasks.ForEach(t => ...)) adding each task's Result to the popularStems collection.
2. Finally, create a loop that iterates around the popularStems collection and prints each Tuple's information (stem size, stem and number of occurences.
3. Edit the code in Program Main to call StemA.FindStems().
4. 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 Initial Steps – 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.
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. Locate the code inside the if (!string.IsNullOrEmpty(bestStem)) expression. Change it so it adds the new Tuple to the popularStems collection.

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.

1. Return to the FindStems method and locate the code that was inside the for loop. Delete the int size = stemSize; and tasks.Add(task); lines. Change the line that declares a Task<List<Tuple<int, string, int>>> by renaming the variable to task1 and passing stems, 2, n/2 + 1 as parameters to the StemSearch method.
2. 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.
3. Change the Task.WhenAll() to wait for both task1 and task2 to complete.
4. Add 2 lines that add the Results of each completed Task to the popularStems collection (by using its AddRange() method).
5. Edit the code in Program Main to call StemA.FindStems().
6. 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.

**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.

# ASP.NET API Exercise: Estate Agent Microservice

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

# Estate Agent Microservice 1 – ASP.NET MVC API

**Objective**

Your goal is to create a microservice for "buyer" information. The microservice should be created from an ASP.NET **MVC** API template and 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.

## 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:

1. Add a folder called Models to the project.
2. Add a class called Buyer to the Models folder.
3. 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; }

}

}

1. Add folder called Infrastructure to the project.
2. Add a class called BuyerContext to the Infrastructure folder.
3. 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; }

}

}

1. Add an empty Controller called BuyerController to the Controllers folder.
2. Add a Route attribute to the BuyerController class with a value of "api/[controller]".
3. Add a private readonly variable of type BuyerContext called \_buyerContext to the top of the class.
4. Add a constructor to the class that takes a BuyerContext parameter BuyerContext called context.
5. 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.
6. Delete the Index method.
7. Add a new public method to the BuyerController class called GetBuyers giving it a return type of async Task<IActionResult>.
8. 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.
9. 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.
10. Return the buyers collection from the method wrapped in an OKObjectResult.
11. 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);

}

}

1. Open up the appsettings.json file.
2. 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"

},

1. Open up the Program.cs file.
2. Add the following code just beneath the builder.Services.AddControllers line:

builder.Services.AddDbContext<BuyerContext>(options =>

options.UseSqlServer(

builder.Configuration.GetConnectionString("sqlestateagentdata")));

1. 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.
2. 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.
3. Add a **static** class called BuyerSeeder to the Infrastructure folder
4. 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();

}

}

1. 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();

}

1. Launch the app and wait for the Swagger page to open in a browser. Then Test drive the call to the "/api/Buyer/buyers" end-point and ensure it returns some data that looks something like the following:  
   A screenshot of a computer

   Description automatically generated

## Adding New Buyers

1. 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.
2. 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.
3. **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).**
4. 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.
5. Add code to the method that calls the Add method of the Buyers collection associated with the \_buyerContext passing it the buyer object.
6. Invoke the \_buyerContext object's SaveChanges method.
7. 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.
8. Launch the app and test drive the new insert method by using the Swagger interface.

## If you have time:

1. 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.
2. 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.

# Estate Agent Microservice 2 – ASP.NET **Minimal** API

**Objective**

Your goal is to create a microservice for "seller" information. The microservice should be created from an ASP.NET **Minimal** API template and 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.

## STEPS

1. Use Visual Studio to create a 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:

1. Add a folder called Models to the project.
2. Add a class called Seller to the Models folder.
3. 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;

}

}

}

1. Add folder called Infrastructure to the project.
2. Add a class called SellerContext to the Infrastructure folder.
3. 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; }

}

}

1. Open up the appsettings.json file.
2. 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"

},

1. Open up the Program.cs file.
2. Add the following code just beneath the "//Add services to the container" comment:

builder.Services.AddDbContext<SellerContext>(options =>

options.UseSqlServer(

builder.Configuration.GetConnectionString("sqlestateagentdata")));

1. 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.
2. Add a **static** class called SellerSeeder to the Infrastructure folder
3. 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();

}

}

1. 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();

}

1. That's all the database access logic in place. All we need now are some Http endpoints.
2. 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.
3. 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());

1. 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.
2. 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.
3. 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

1. 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.
2. Add the following code as the delegate logic:

db.Sellers.Add(seller);

await db.SaveChangesAsync();

return Results.Created($"/sellers/{seller.Id}", seller);

1. **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).**
2. 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
3. Launch the app and test drive the new insert method by using the Swagger interface.

## If you have time:

1. 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.
2. 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.

# Estate Agent Microservice 3 – One Database per Microservice

**Objective**

Your goal is to create 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 and implement an Event Bus that tracks the deletion of a Property so that it also removes any associated bookings:

## STEPS

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

## Add an Event Bus to track Property deletions and trigger the removal of any associated Bookings.