

Dist. Sys.

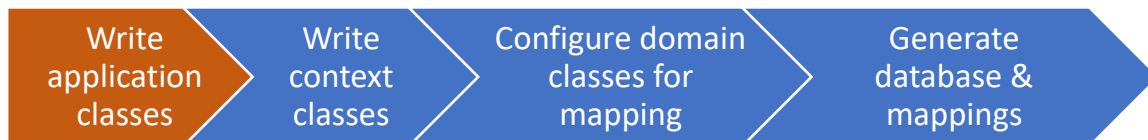
Entity Framework Lab

In this lab you'll be looking at Entity Framework Core.

We'll be following a code-first approach. If you have prior experience in Entity Framework, use this lab time to refresh your knowledge and experiment with model-first/database-first.

The code-first approach allows us to create our database and mappings, based on code we write.

The workflow we will be following is:



Open Visual Studio and create a new .NET Core Console app called **EntityFrameworkLab**

You can save this anywhere but if you get unexpected errors you may want to build and run to a local drive (e.g. C:/Users/<yourUID>/MyProjects)

Right-click on the auto-generated project and select 'Manage NuGet Packages'.

Click 'Browse' in the top left of the new window that opens, type "EntityFrameworkCore" into the search bar and hit 'Enter' to search.

You should see 'Microsoft.EntityFrameworkCore' as the first search result. Select this package and click the 'Install' button in the frame on the right-hand side of the window. Click OK in the dialog the opens then accept the license (although you should read this...).

The package manager will now automatically download and install Entity Framework into your project. The 'Output' window will update with progress.

Wait until it reads "===== Finished ====="

We also want to use SQL Server as our database so do the same NuGet package install for 'Microsoft.EntityFrameworkCore.SqlServer'.

Entity Framework is now installed. You can check this by ensuring that the Dependencies -> NuGet section of your project (in the Solution Explorer Window) includes 'EntityFrameworkCore'.

Now we're ready to start producing our application!



Click your project (in the Solution Explorer Window), hit Ctrl+Shift+A and create a new class. Give your class the filename 'Person.cs'. Repeat and add a class called 'Address.cs'.

Modify the 'Person' class to read:

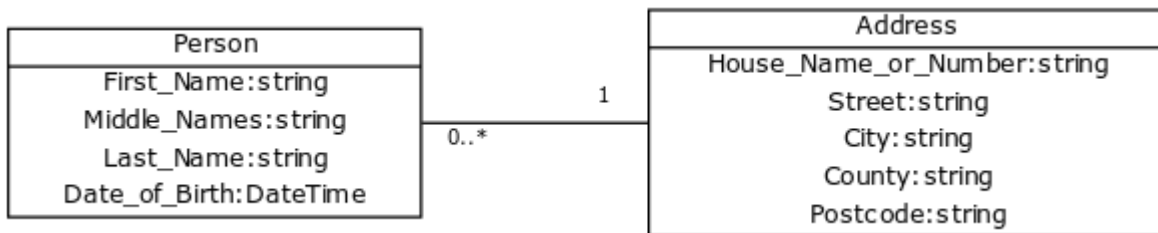
```
public class Person
{
    public string First_Name { get; set; }
    public string Middle_Name { get; set; }
    public string Last_Name { get; set; }
    public DateTime Date_of_Birth { get; set; }
    public Address Address { get; set; }
    public Person() { }
}
```



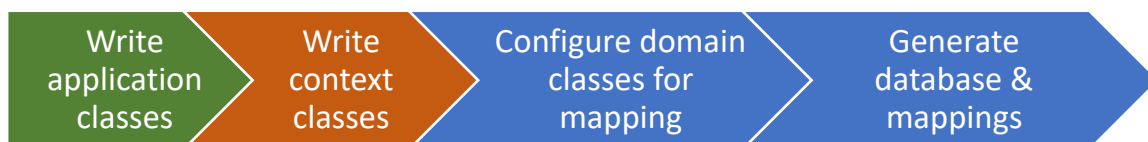
Modify the 'Address' class to read:

```
public class Address
{
    public string House_Name_or_Number { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public string County { get; set; }
    public string Postcode { get; set; }
    public ICollection<Person> People { get; set; }
    public Address() { }
}
```

Consider what structure this gives us:



We've created our application classes, now it's time to write our context classes.



The Context class (DbContext) is responsible for mapping entities to database tables, allowing and performing queries on your database, inserting, updating and deleting data in the database (based on the state of your entities in code) and managing data relationships.

You need to create the context class so you can tell Entity Framework which entities to map to database tables. This is very easy and simply means declaring one new class and a DbSet for each entity you want to map.



Create a new class called 'MyContext.cs'

MyContext needs to inherit from DbContext so add: `using Microsoft.EntityFrameworkCore;` to the top of your file. Now change your class signature to: `public class MyContext : DbContext`

Add the constructor to your class: `public MyContext() : base() { }`

Now also add the DbSets you want to map:

```
public DbSet<Person> People { get; set; }
public DbSet<Address> Addresses { get; set; }
```


Finally, we want to override the OnConfiguring method to set a connection string. Add the method:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer(
        "Server = (localdb)\\mssqllocaldb; Database = EFLab;");
    base.OnConfiguring(optionsBuilder);
}
```

Your context is now complete. I told you it was easy! However, in order to see it work, you'll first need to make a change to the data. This means using your new context class to make some changes to your entities. The best practice for use of your context class is by employing a 'using' statement.

A **using statement** should be used where the type you are using implements the `IDisposable` interface. That is to say that it can and should be disposed of once you have finished using it. The 'using' statement ensures that the 'Dispose' method is called without you having to remember to call it EVEN IF an exception occurs!

The other option is using Try/Finally blocks and calling Dispose inside the Finally block (this is actually what a 'using' statement does under the hood) but a using statement is considered better for readability and maintainability, and you're not going to forget to call the Dispose method.



Open 'Program.cs' from your solution folder.

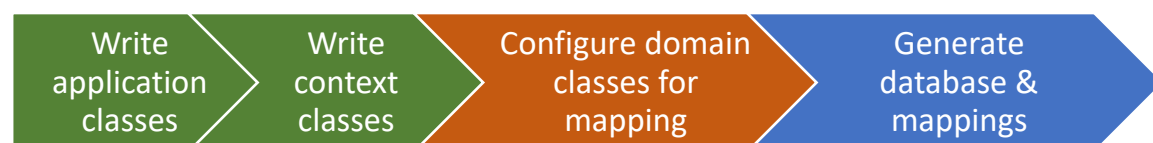
Add this code to the Main method and think about what you expect it will do...

```
using (var ctx = new MyContext())
{
    Address addr = new Address() { House_Name_or_Number = "1076", Street = "Some Street", City = "Some City", County = "Some County", Postcode = "Some Postcode", People = new List<Person>() };
    Person prsn = new Person() { First_Name = "Jane", Middle_Name = "Janet", Last_Name = "Doe", Date_of_Birth = new DateTime(2010,10,1), Address = addr};

    ctx.Addresses.Add(addr);
    ctx.People.Add(prsn);
    ctx.SaveChanges();
}
```

Now hit F5 to compile and run your program! Using code-first, Entity Framework is designed to make the database for you based on your code. It may take some time to run the first time.


Did you get an error? Have a read of the exception that was thrown. What do you understand has happened?



If you read through the exception details then you should be able to decipher that Entity Framework is upset that you haven't included any keys. We need to configure our domain classes for mapping.

Keys are used by databases to identify each record as a unique data item. Every data table **MUST** have a key so we've got to add keys to our classes in order to allow Entity Framework to create our database.

There are two ways to add a key so we'll use a different approach in each of our classes.



Open 'Person.cs' and add the property:

```
public int PersonID { get; set; }
```

This property is a **convention** so Entity Framework natively understands to make this the key for the People table.

Open a browser and read about Entity Framework Code First Conventions at this URL: [https://msdn.microsoft.com/en-us/library/jj679962\(v=vs.113\).aspx](https://msdn.microsoft.com/en-us/library/jj679962(v=vs.113).aspx)



Open 'Address.cs' and add the property:

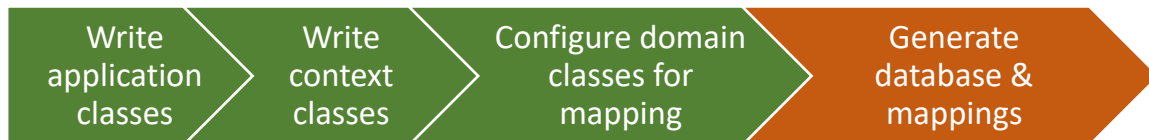
```
public int AddressIdentifier { get; set; }
```

This property does not conform to convention so Entity Framework will have no idea that you want this to be the primary key. However, we can tell it explicitly to use this as the key by using the [Key] attribute.

First, add Conventions namespace to the top of your file:

```
using System.ComponentModel.DataAnnotations;
```

Then add the attribute [Key] directly above the AddressIdentifier property.



Now hit F5 to compile and run your program!

This time, a new error. Entity Framework used to create the database for us... is no longer does. So we'll have to do it manually.



Open the NuGet package manager again and search for the package called: Microsoft.EntityFrameworkCore.Tools

This package contains tools we, as a developer, need to manage our databases using Entity Framework. However, these tools are not required for deployment (which is why they are in a different package). Install the package to your project.

Open the Package Manager Console window. If this isn't already listed along the bottom of your Visual Studio window you can access this through:

View -> Other Windows -> Package Manager Console

Type: **Add-Migration InitialCreate** and press Enter. The package manager should now detect your database context in your code, scaffold a migration and create a new Migrations folder (with three files) in your project.

Open the file named <number>_InitialCreate.cs where <number> is a long string of digits which relate to the date and time.

Look at the methods in this class. One is Up() and one is Down(). Look at the method content. What are the methods doing?

The Up() and Down() methods allow Entity Framework to revert and reinstate changes that were made to the underlying database.

In the case of the Initial Create, the original state of the database was no tables so the Down method removes all of the keys and tables. The Up method, on the other hand, generates the tables, columns and assigns keys.



In the Package Manager Console window Type: **Update-Database** and press Enter. The package manager should now apply the migration and make the database for you.

Let's investigate what happened under the hood...

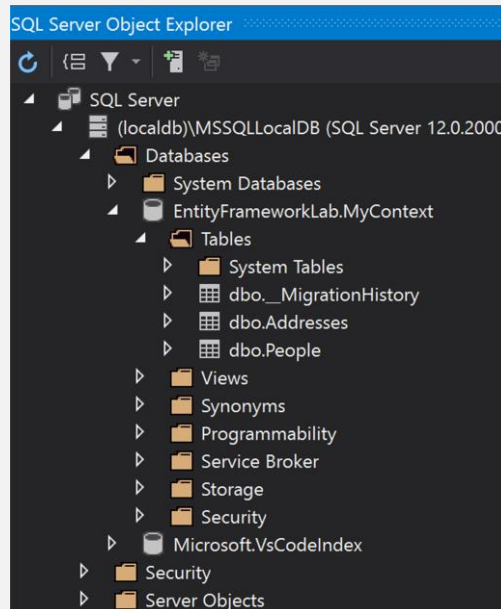


Click 'View' on the toolbar and click 'SQL Server Object Explorer'.

Find the database containing the name: MSSQLLocalDB

Expand the Database with your Solution name, then expand the 'Tables' folder.

This should look something like this:



Right-click `dbo.People` and click 'View Data'. You should now see your database table as a series of columns with no data.

Now view the Data for Addresses.

Compare the Addresses database table to your Address class. What do you notice seems to be missing in the table?

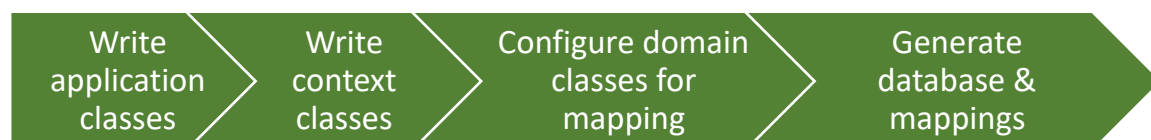
You might have noticed that there is no field for 'People'. If you didn't notice this, go back and look at the class/table columns to see what I'm talking about.

Why do you think this is?

Go back to the data for the People table and take another look. See how there is a column for `AddressIdentifier` – this is a foreign key relationship with the new Address you added. This works because of the mapping. As each Address may be related to zero or more People, it makes sense to store the key to the linked address in People, rather than store a list of keys to People in each Address. Entity Framework is clever enough to convert this type of conventional database relationship to and from the C# object references for you (although there is some overhead).

Now hit F5 to compile and run your program! It may take some time to calculate.

Finally, go back and inspect the shiny new data in your database. If you already have the View Data window open you'll need to hit the refresh arrow.



So that's done then! Well... not quite...



Open 'Address.cs' and add the property:

```
public string Country { get; set; }
```

and add `Country = "UK"` where you define the new Address in Program.cs.

Now hit F5 to compile and run your program! It may take some time to calculate.

You've probably hit an exception. Read the exception detail. What do you think happened?

The problem is that your Model and Database are now no longer aligned. You need to tell Entity Framework to update the database in order to allow it to work. Again, this is done through **Migrations** and these are performed through the **Package Manager Console**.



Remember that our Up() and Down() methods inside our migration allow Entity Framework to revert and reinstate changes that were made to the underlying database. In the case of the Initial Create, there was no Country field so this hasn't been pushed to our database yet.

You'll notice that the `dbo.Addresses` table doesn't specify Country. In order to update the database to match our model we'll have to make a new migration that inserts the Country column.



Open the Package Manager Console window again.

Type: **Add-Migration** and press Enter. The package manager should now ask you for a name. Give it the name **Migration1**

When it's finished it should automatically open the file named `<number>_Migration1.cs` where `<number>` is a long string of digits which relate to the date and time.

Look at the methods in this class.

Note how the new migration again has Up and Down methods. Consider what the methods do.



Open the Package Manager Console window again.

Type: **Update-Database** and press Enter. The package manager should now apply the migration.

Click 'View' on the toolbar and click 'SQL Server Object Explorer'.

Find the database containing the name: MSSQLLocalDB

Expand the Database with your Solution name, then expand the 'Tables' folder.

Find table named dbo.EFMigrationHistory and view its data. You should see that there are two rows of data which correspond with the two migrations you have created. This database table remembers which migrations have been applied allowing you to apply multiple migrations or roll back to a predefined point in time.

Now open dbo.Addresses and view its data. Note how the Country field on the row you had already added is NULL. This makes sense because you didn't originally have a Country column when this data was inserted so there can be no data here.

Now hit F5 to compile and run your program! It may take some time to calculate.

Your program should run without a hitch. Take a look at the new data in the database tables. Your new Address should now have the Country field set as 'UK'.

Easy! But what about more complex tasks?



Open 'Person.cs' and change the property: `public string Middle_Name { get; set; }` to `public string Middle_Names { get; set; }`

and change `Middle_Name = "Janet"` to `Middle_Names = "Janet"` where you define the new Person in Program.cs.

We know that this will require a migration, so let's do that too...



Open the Package Manager Console window again.

Type: **Add-Migration** and press Enter. The package manager should now ask you for a name. Give it the name **Migration2**. Press Enter. Now note the warning in yellow...

Look at the code that the migration has created for us. The up and down methods seem to be indicating that we to drop the Middle_Name column altogether. We don't want this as it will delete our existing data! We only wanted to rename it. Entity Framework isn't clever enough to identify this so we need to give it a hand.

Remove the content of the Up method and write in:

```
migrationBuilder.RenameColumn(  
    name: "Middle_Name", newName: "Middle_Names", table: "People");
```

Remove the content of the Down method and write in:

```
migrationBuilder.RenameColumn(  
    name: "Middle_Names", newName: "Middle_Name", table: "People");
```



Type: **Update-Database** into the Package Manager Console and press Enter.

Entity Framework updates the database. Your database column has now been renamed rather than replaced, and you've kept all the data too!



Entity Framework is cleverer still... but you'll need to get used to using attributes or fluent API if you want to make the most of it.

Read through this post on modeling:

<https://docs.microsoft.com/en-us/ef/core/modeling/>

and experiment with some of this functionality in your code (e.g. add an index and a database generated timestamp)

There is also lots more to learn here:

<https://docs.microsoft.com/en-us/ef/core/>

Further Work:

- We'll be using Entity Framework in the Coursework so it makes sense to get your head around it.
- Research the difference between Code-First, Model-First and Database-First implementations of Entity Framework. These options are provided for different people with different skills or projects starting points.
- Add a custom Entity Framework database to the server side of one of your previous (SignalR?) distributed applications. Add functionality to have the server send back data to the client which has been retrieved from the database and store data which has been sent from the client.
- As an example, you could have the client send a string of text and have it stored in the database with a timestamp; then when a new client requests previous messages to a certain date/time your server could get these all and return them. You could make this more complex by adding client IDs so your client is differentiated from other clients. Have the client send it's ID with each request and store this alongside the message so you know who sent what.