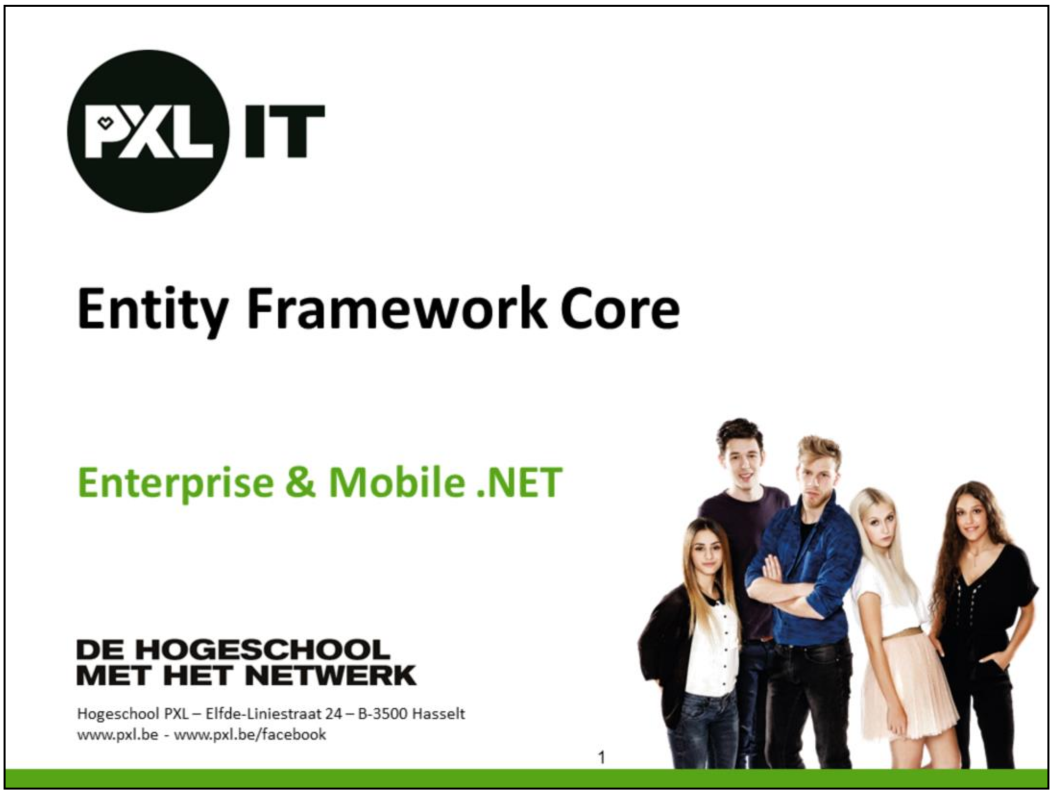


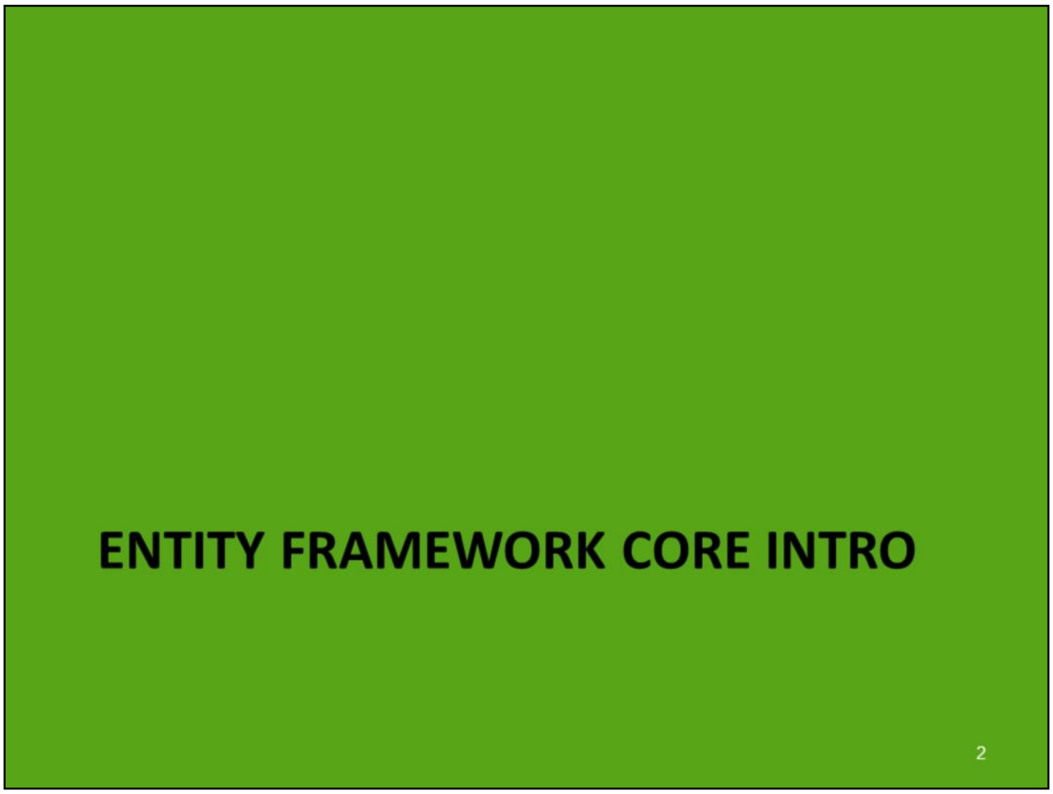
Entity Framework Core

Enterprise & Mobile .NET

DE HOGESCHOOL
MET HET NETWERK

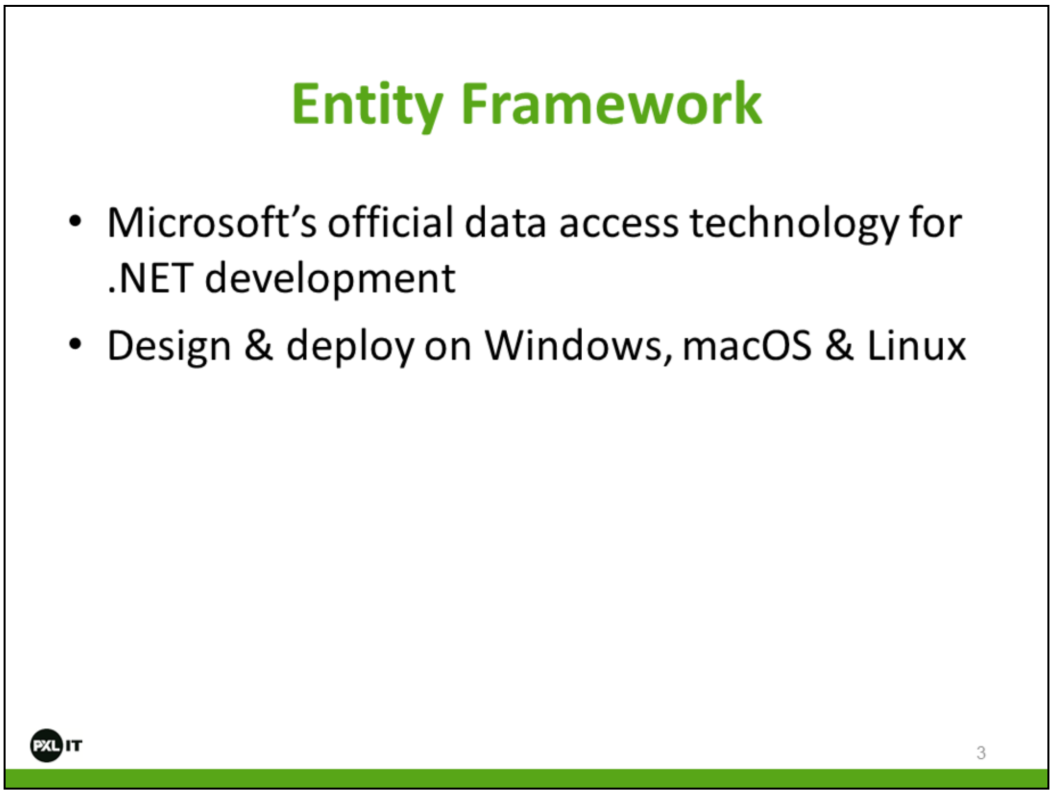
Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

1



ENTITY FRAMEWORK CORE INTRO

2



Entity Framework

- Microsoft's official data access technology for .NET development
- Design & deploy on Windows, macOS & Linux

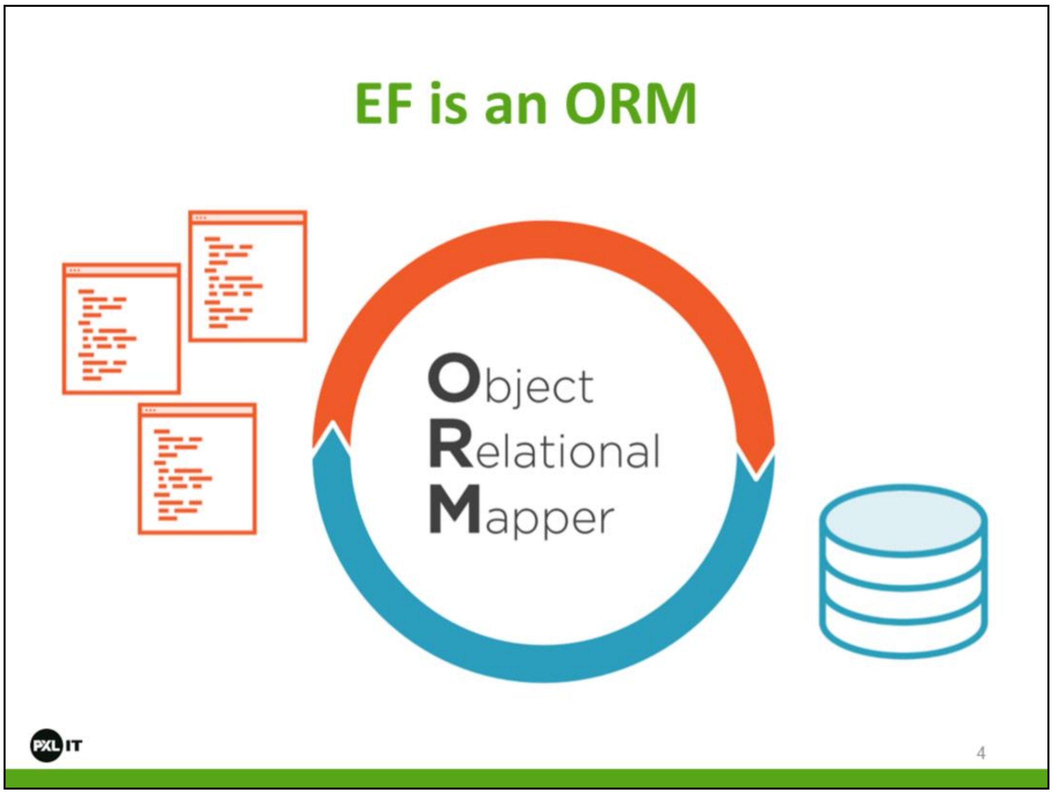
3

Entity Framework Core, or EF Core, is an evolution of Microsoft's **object relational mapper**, which has been around since 2008 and evolving and improving dramatically over the years.

EF Core encompasses a set of .NET APIs for performing data access in your software, and it's the **official data access platform from Microsoft**.

It's used in millions of applications, and its **cross-platform**.

You can build and run EF Core-based APIs or applications on Windows, Mac OS, Linux, and even in Docker containers, whether on-premises or in the cloud.



EF is an ORM

Object relational mappers, or ORMs, are designed to **reduce the friction between how data is structured in a relational database and how you define your classes**.

Without an ORM, we typically have to write a lot of code to transform database results into instances of the types in our software.

An ORM allows us to express our queries using our classes, and then the ORM itself builds and executes the relevant SQL for us, as well as materializing objects from the data that came back from the database.

It also allows you to store, and update, and even delete data in the database.

Using an ORM can really eliminate a slew of redundant data interaction coding tasks.

In doing so, EF Core can really enhance developer productivity.

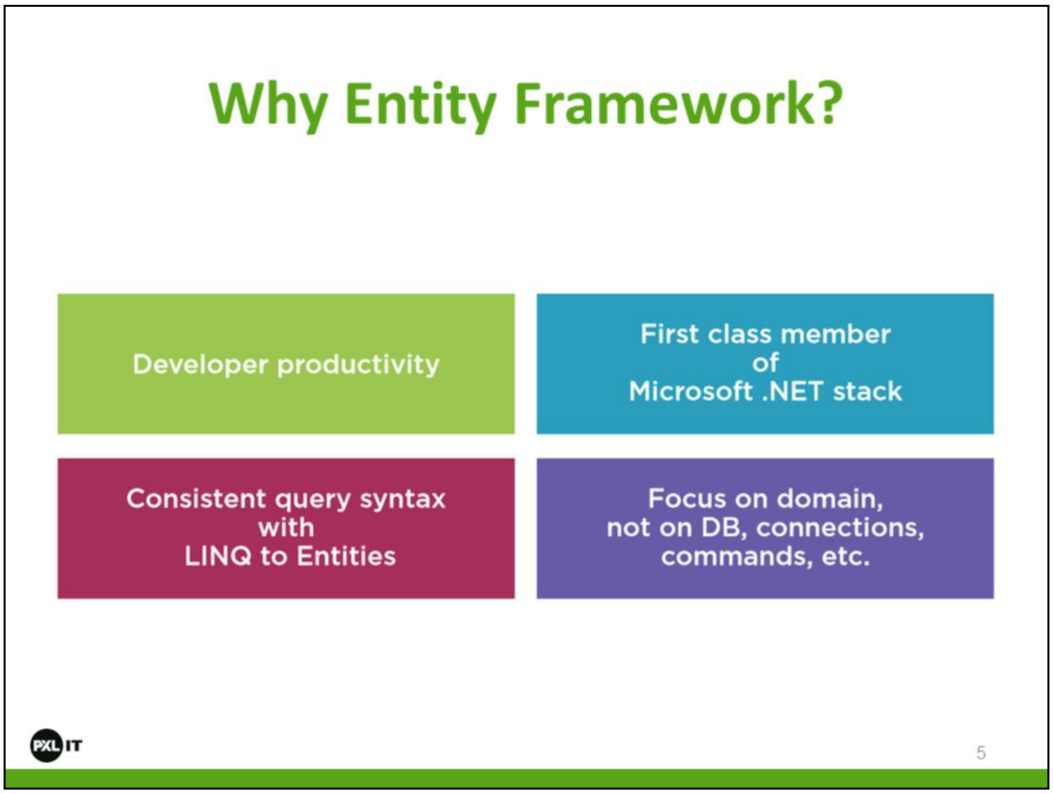
It also **provides consistency** in the tasks that it does, rather than having various members of your team invent their own means of designing their data access.

While a typical ORM infers that the classes and database tables are of a similar structure, EF Core has a **mapping layer** in between, and that gives us a lot more flexibility in how to get from objects to tables and from object properties to table columns.

EF Core allows you to **connect to a variety of data stores**, mostly relational databases.

While Microsoft maintains rich providers for SQL Server and SQLite, and even an in-memory provider that's incredible for automated testing, there are many other third-party providers as well, some commercial and some from the community. One of the new features that came with EF Core 3 is support for a non-relational database, Azure CosmosDB, Microsoft's document database that's in the cloud.

This allows you to leverage your existing knowledge of EF Core to interact with CosmosDB, instead of having to learn another client API.



Why Entity Framework?

Developer productivity	First class member of Microsoft .NET stack
Consistent query syntax with LINQ to Entities	Focus on domain, not on DB, connections, commands, etc.

PXL IT

5

So why an ORM over other ways of doing data access and why this ORM, **why Entity Framework Core?**

Using an ORM can really eliminate a slew of redundant data interaction coding tasks, and doing so, EF Core can really enhance **developer productivity**.

It also provides consistency in the tasks that it does, rather than having various members of your team invent their own means of designing their data access tasks.

Not to say that there isn't a learning curve for EF Core, especially if you want to leverage its advanced features, but in simpler scenarios, the learning curve can be quick.

EF Core has a dedicated team at Microsoft and Entity Framework has been around for over 10 years and has gone through a number of evolutionary steps as it gains functionality and has become more sophisticated.

The last version of what we might refer to as classic Entity Framework, and that's EF6, has over 14 million downloads.

It's used in myriad (large) applications and will continue to be supported and maintained, but the evolution to EF Core in order to continue to innovate on Entity Framework shows that this investment will continue to be of high importance to Microsoft.

Most ORMs allow you **to connect to a variety of databases** and EF Core is no exception.

While Microsoft maintains rich **providers for** a **SQL Server** and **SQLite** and even an **in-memory provider** that's incredible for automated testing, there are **many other third-party providers** as well, some commercial and some from the community.

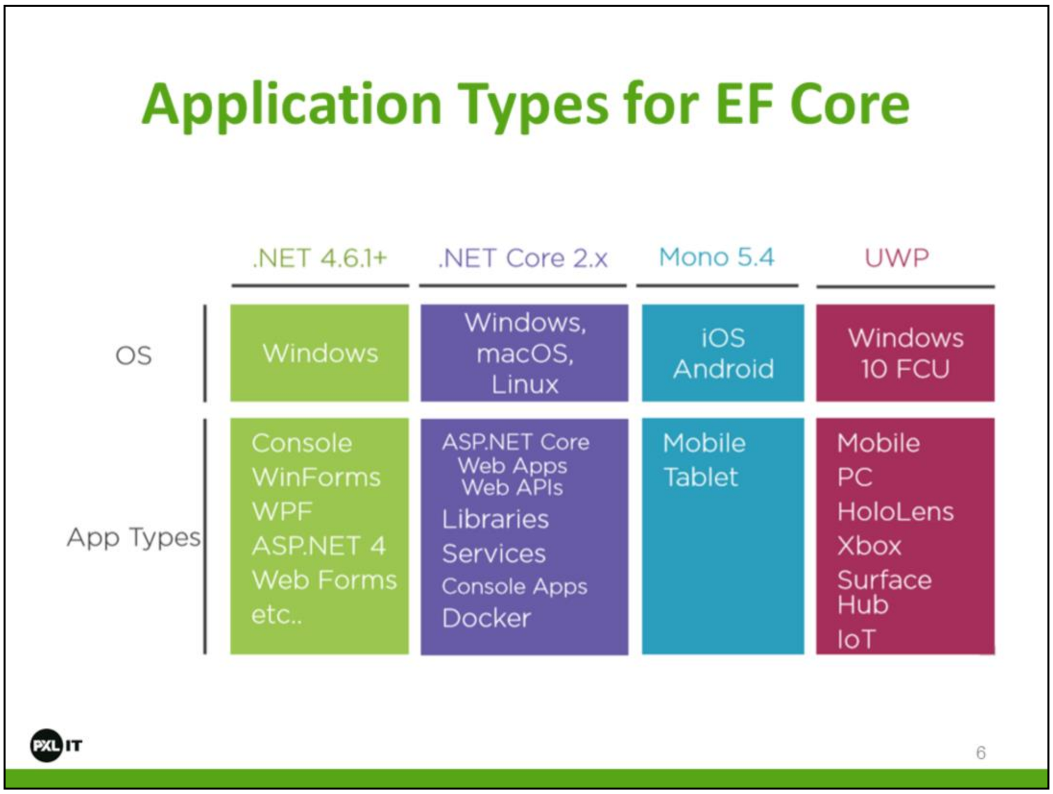
Rather than writing the relevant SQL to target whatever relational database you're working with, **EF Core uses the LINQ** syntax that's part of the .NET Framework.

LINQ to Entities allows developers to use a consistent and strongly-typed query language regardless of which database they're targeting.

Additionally, LINQ for objects is used for querying other elements in .NET, even in-memory objects.

So developers benefit from their knowledge of LINQ whether they're using LINQ to Entities, LINQ to Objects, or some other flavor of LINQ.

Using an ORM allows developers to **focus on their domain**, their business rules, and their business objects, they **don't have to worry about direct interaction with the database** or being intimately familiar with the database schema. Developers still need to understand how Entity Framework Core works and some of its nuances with regards, for example, to tracking changes that need to be persisted to the database or patterns for working disconnected applications, but that follows the importance of understanding how any of your tools work and not just blindly using them in your software.



Application Types for EF Core

	.NET 4.6.1+	.NET Core 2.x	Mono 5.4	UWP
OS	Windows	Windows, macOS, Linux	iOS Android	Windows 10 FCU
App Types	Console WinForms WPF ASP.NET 4 Web Forms etc..	ASP.NET Core Web Apps Web APIs Libraries Services Console Apps Docker	Mobile Tablet	Mobile PC HoloLens Xbox Surface Hub IoT

PXL IT

6

EF Core is incredibly flexible with respect to where you can deploy it, as well as where you can build apps that use it.

EF Core runs on top of the **Cross-platform .NET Standard 2.0**, which is supported by a variety of runtimes from **.NET Framework** to **Linux** to **UWP** to **iOS**, but this means you can only build and run EF Core in apps whose runtime does sit on top of .NET Standard 2.

There has been a lot of confusion around this and I believe that many are still surprised that .NET Framework is one of those options and this doesn't just mean where you can deploy your apps, but also where you can build them.

Let's first return to a slide you saw briefly already just to be sure you're really clear about these options.

Because **.NET 4.6.1 and above supports .NET Standard 2.0**, you can use EF Core in any of the **classic Windows (WPF)** or **web applications**.

That means you can benefit from features of EF Core without being forced to give up the application types your business and customers might still be relying on, and hey, no judgment here, I'm still supporting many of those application types, and if you follow me on Twitter, you know that these are modern compared to the old FoxPro and VB6 apps I wrote decades ago that will not die.

More obvious for EF Core is the fact that you can use it in **.NET Core apps** that run **cross platform** on **Windows**, **Mac 0S**, and **Linux**.

For Windows 10 Universal Windows Platform (UWP) apps, EF Core can run on any device that has **Windows 10 Fall Creators update** installed.

And **Xamarin** will let you use EF Core in native apps that run on **iOS**, **Android**, **Windows**, and **Mac OS**.

The requirements here are a little complicated because they differ from one environment to another.

In order to build anything with EF Core, you must have the .NET Core SDK installed on your machine.

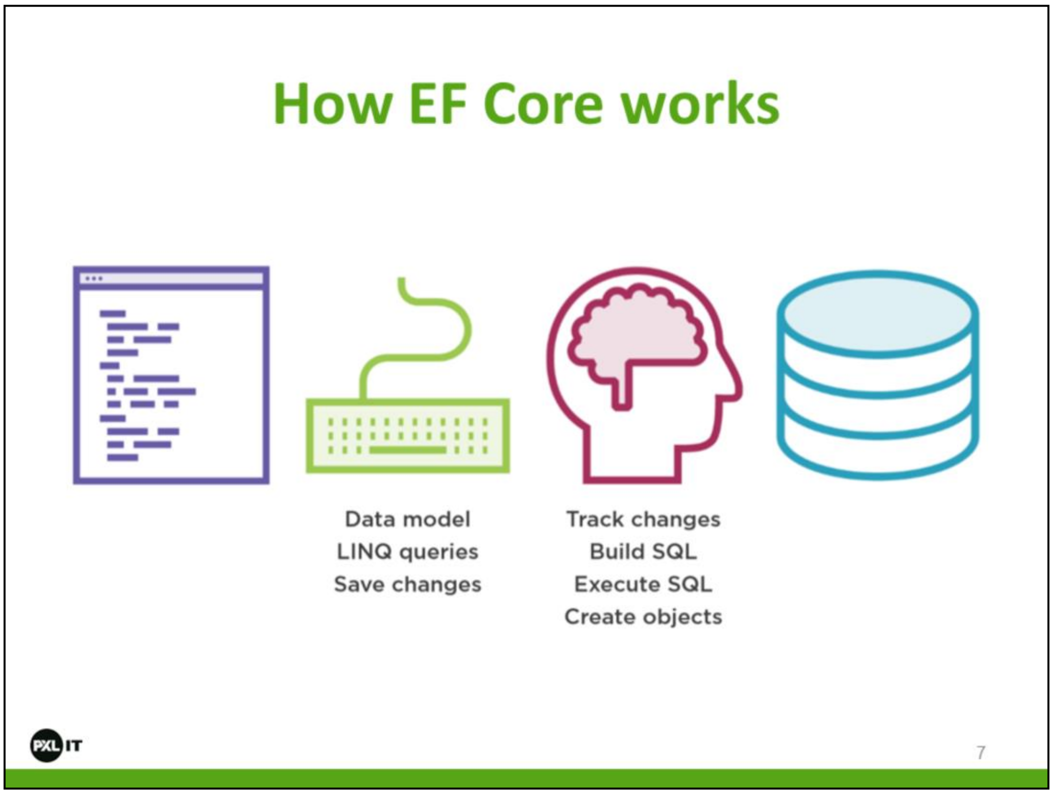
If you're planning to build software that will only ever run on .NET Framework, you'll need to be on a Windows machine to do your development and you can use Visual Studio.

If you plan to build Cross-platform apps using .NET Core, you've got a lot of options, Visual Studio on Windows, Visual Studio for Mac, or the lightweight Cross-platform Visual Studio code, which you can use on Windows, Mac OS, or Linux, but there is still more.

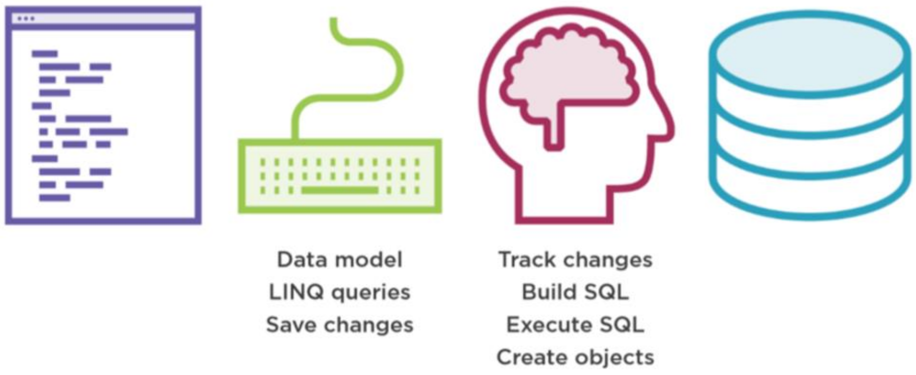
If you're building mobile apps targeting UWP, your development machine must have Windows 10 Fall Creators Update on it and the latest version of Visual Studio is always best, but it has to be at least version 15.4.

And finally, building Xamarin apps also has a number of options, but the targets will determine your requirements.

Regardless of where you're doing the coding or where your application gets deployed, there is only one set of APIs that you'll use when working with EF Core.



How EF Core works



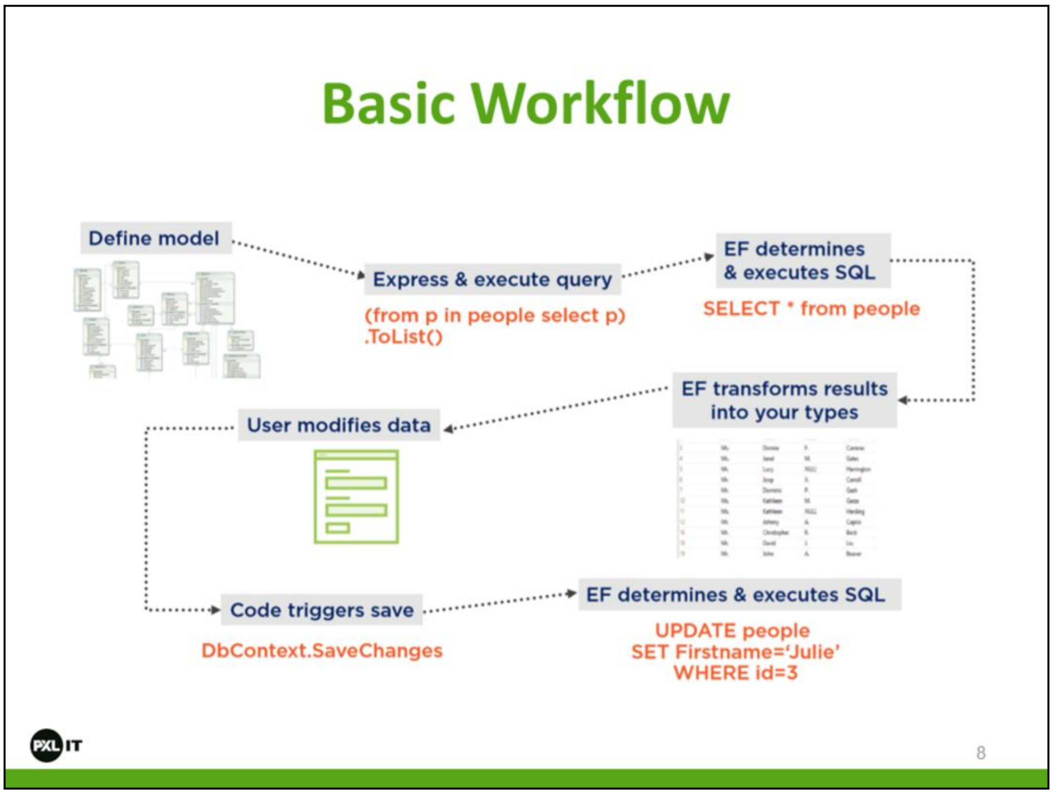
PXL IT

7

Here is a high-level look at EF Core's workflow which I feel is important to understand before jumping into any code.

The puzzle pieces involved fall into four different categories.

1. First, you need **the classes that describe your domain**, the business problem that your software is going to solve, and this part actually has nothing to do with Entity Framework.
2. Then you use Entity Framework APIs to **define a data model** based on those domain classes.
   You also use Entity Framework APIs to **write and execute LINQ to Entities queries** against those classes, and in your code, you'll need to call Entity Framework's **save changes** to push data back to the database.
3. What Entity Framework's APIs will do for you is **keep track of the state of objects that it's aware of**, it'll **determine the SQL it needs to save data back to the database**, and for queries, Entity Framework will **transform** your **LINQ** to Entities queries **into SQL**, **execute** that **SQL**, and then **create objects from the query results**.
4. The last major puzzle piece is that **Entity Framework manages all of the interaction with the data store**.


Basic Workflow

With this information in hand, let's now look at a **basic workflow of how Entity Framework works** in an application.

As I said earlier, **it all starts with your domain classes**, not with Entity Framework, but with those classes, you can then use Entity Framework's **DbContext** API **to wrap the classes into a model and instruct Entity Framework as to how those classes in the model map to the database schema**.

With this understanding and the help of the relevant database provider, Entity Framework can then **translate queries** that you write in **LINQ** to entities against your classes **into SQL** that's understood by your database.

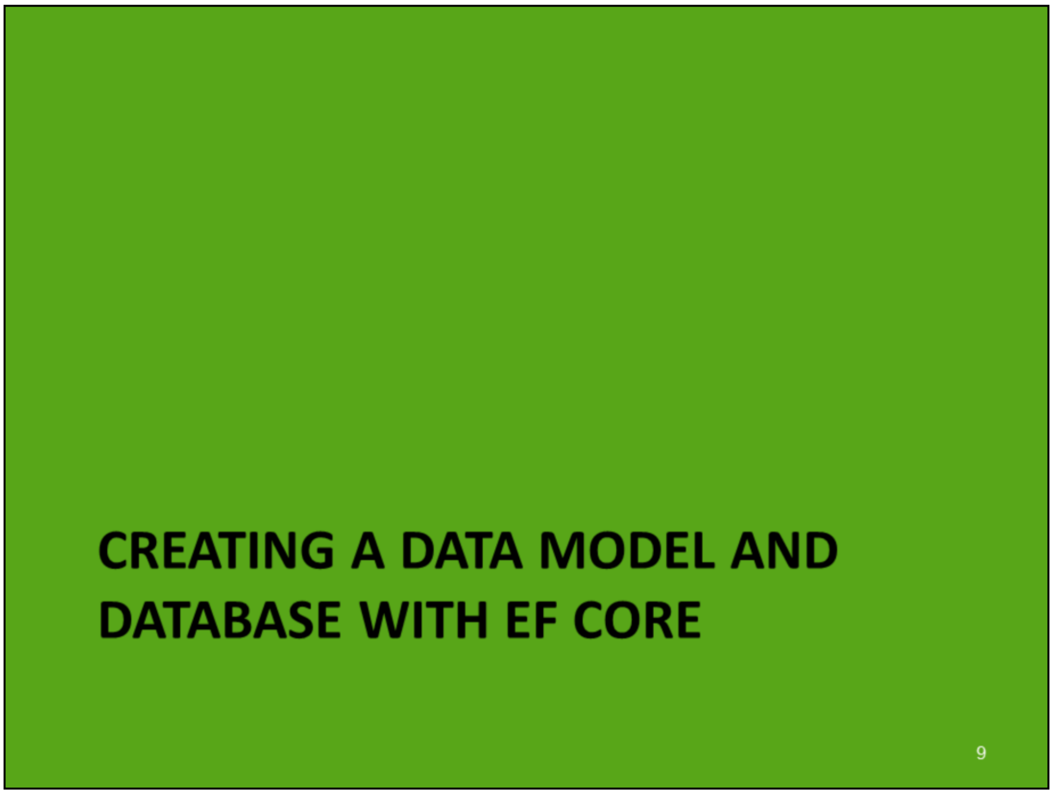
It then **executes the query** and **uses the results to return populated instances of your objects**, again, taking away all of that redundant work that we normally have to do.

You could also map to views, instead of tables, and if you need to, you can even execute stored procedures, maybe if Entity Framework just can't create efficiently performing SQL or if the query is just too hard to express with LINQ.

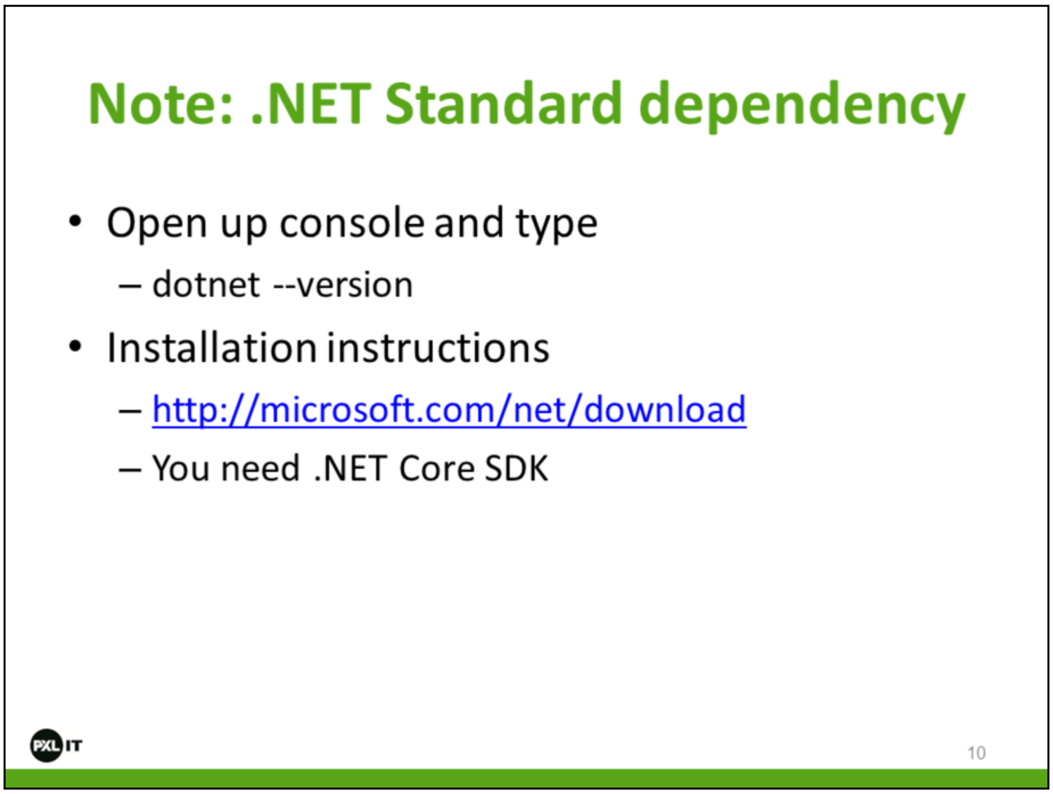
Entity Framework can **keep track of objects** as long as the *DbContext* is in scope, for example, this could happen in a desktop application. If you edit or add or delete an object, Entity Framework will be aware of it.

With disconnected apps though (e.g. a web application), we do have patterns to inform Entity Framework of this state of an object when it comes back from whatever was working with it.

Then with a single command, **save changes**, Entity Framework can use all that state information to **build and execute the relevant insert, update, or delete commands on the database**.



CREATING A DATA MODEL AND DATABASE WITH EF CORE



Note: .NET Standard dependency

- Open up console and type
  - dotnet --version
- Installation instructions
  - http://microsoft.com/net/download
  - You need .NET Core SDK

PXL IT 10

http://microsoft.com/net/download

Remember that EF Core has a dependency on .NET Standard 2.0, but that's not installed by Visual Studio 2019.

You can check to see if you've got the correct version using the command line.

First, type dotnet to see if dotnet is even installed. If it's there, then type dotnet --version and it will list the version of the dotnet command line interpreter or CLI.

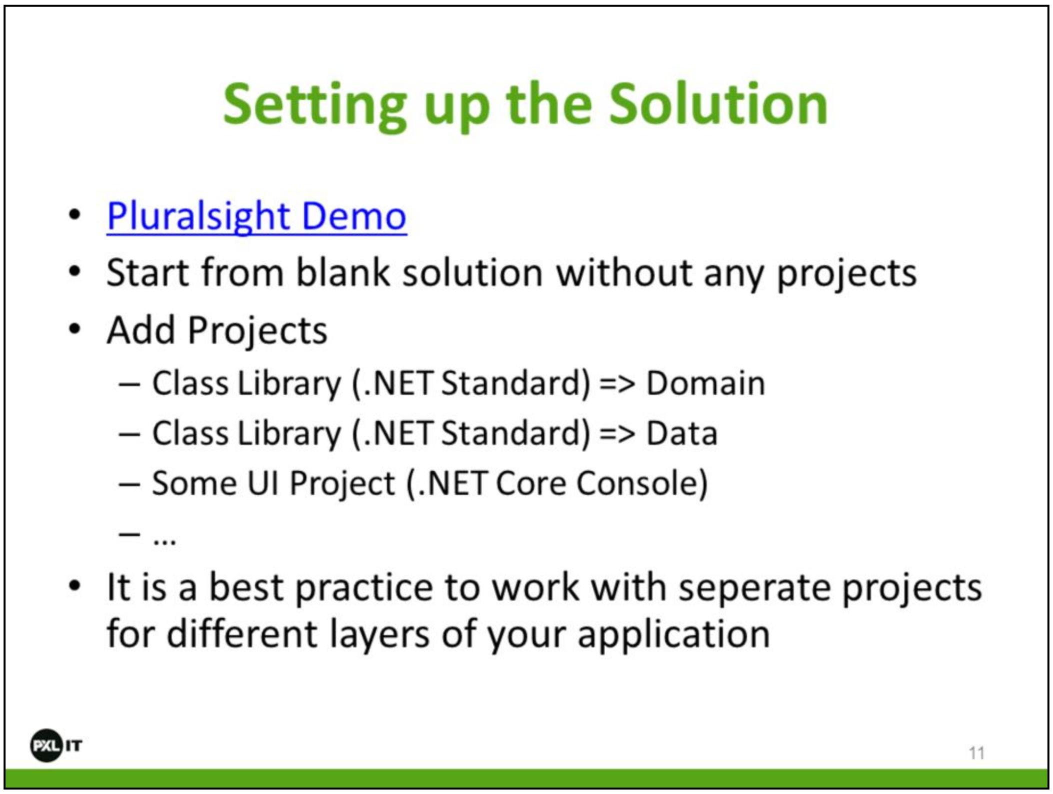
If you need to install the latest version of the SDK, you can go to microsoft.com/net/download to get it.

The URL redirected to Windows because I'm on a Windows machine.

You'll be downloading the .NET Core SDK, which will also install the runtime for you.

Once that's installed, you can get to work in Visual Studio 2019.

Note: at this moment the latest version is of the .NET Core SDK is 3.1

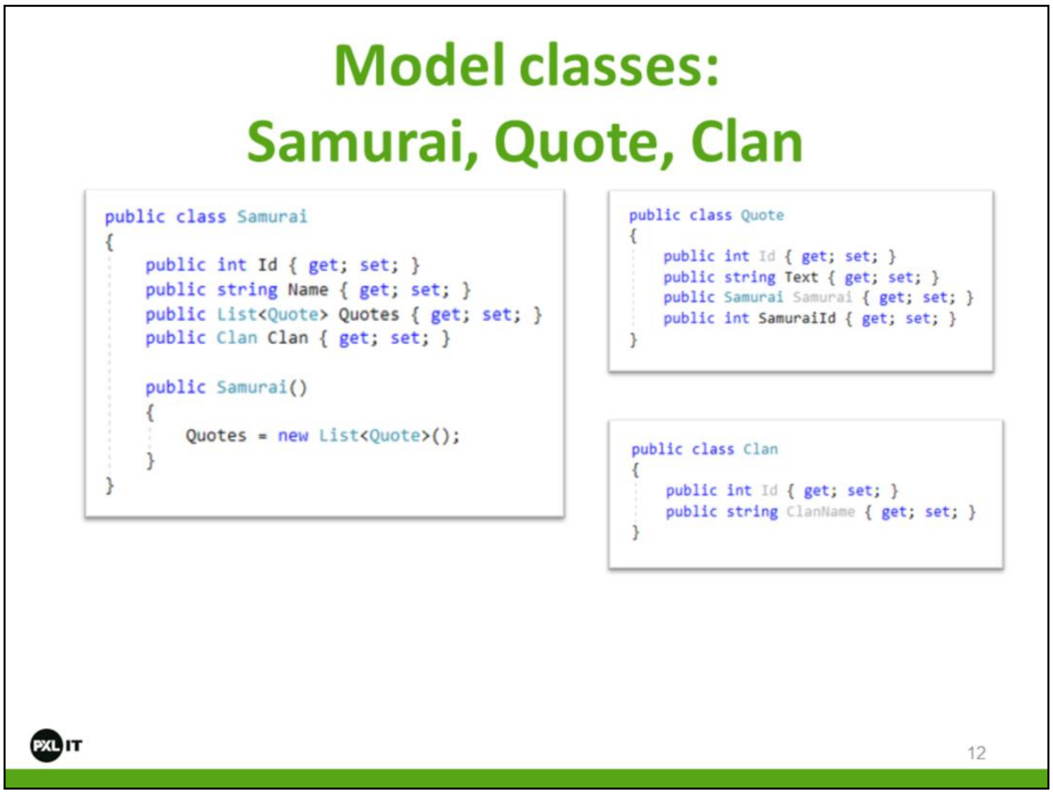


Setting up the Solution

- Pluralsight Demo
- Start from blank solution without any projects
- Add Projects
  - Class Library (.NET Standard) => Domain
  - Class Library (.NET Standard) => Data
  - Some UI Project (.NET Core Console)
  - …
- It is a best practice to work with seperate projects for different layers of your application

PXL IT

11



Model classes: Samurai, Quote, Clan

```
public class Samurai
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Quote> Quotes { get; set; }
    public Clan Clan { get; set; }

    public Samurai()
    {
        Quotes = new List<Quote>();
    }
}
```

```
public class Quote
{
    public int Id { get; set; }
    public string Text { get; set; }
    public Samurai Samurai { get; set; }
    public int SamuraiId { get; set; }
}
```

```
public class Clan
{
    public int Id { get; set; }
    public string ClanName { get; set; }
}
```

12

The ***Samurai*** class has an *Id* property that will be used for the identifier.

It also has a name, a list of quotes, that would be famous quotes that the Samurai spoke in the movie.

I also have a *Clan* property that points to whatever clan the Samurai fights for.

The constructor ensures that the list of quotes is already instantiated before I try to use it in code.

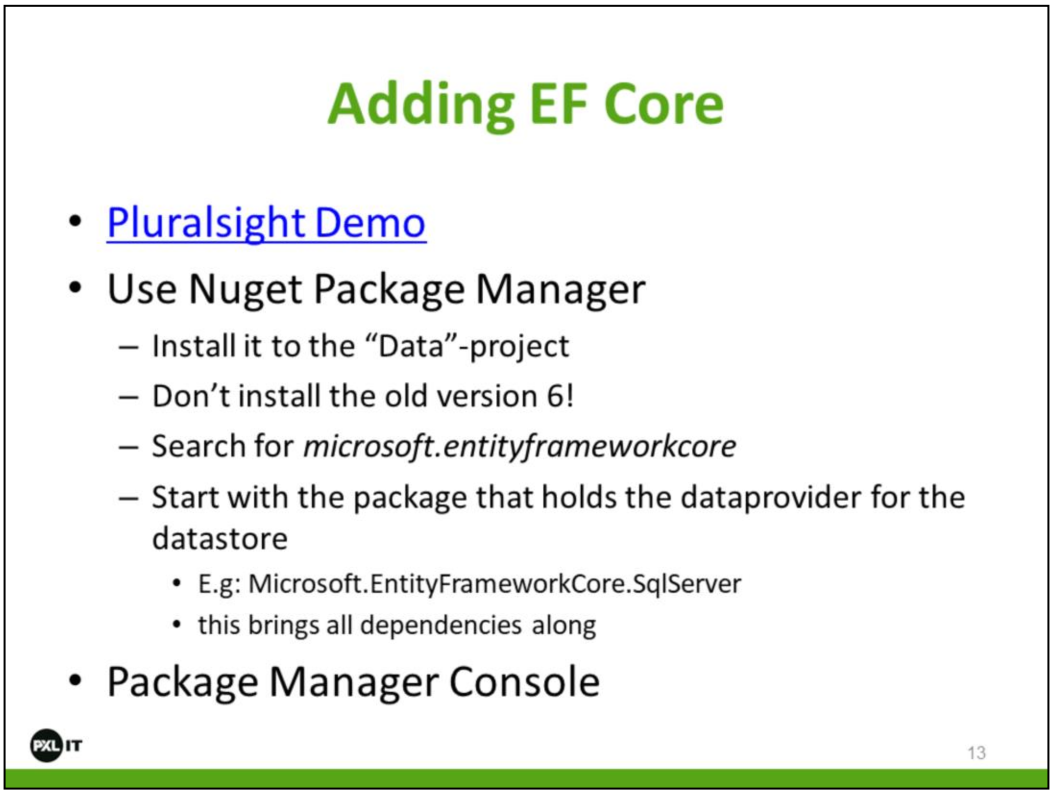
The ***Quote*** class has an *Id* property for tracking it and a *Text* property that will contain the spoken quote.

I've also got a reference property back to *Samurai* and an explicit integer property that will contain the foreign key value, which is the Id of the Samurai.

The ***Clan*** class has an *Id* property and the name of the Clan.

So that's the starting point for my domain classes.

It's pretty simple, but I do have some relationships in here that we'll be working with, and that should suffice for working with the EF Core for the first time.

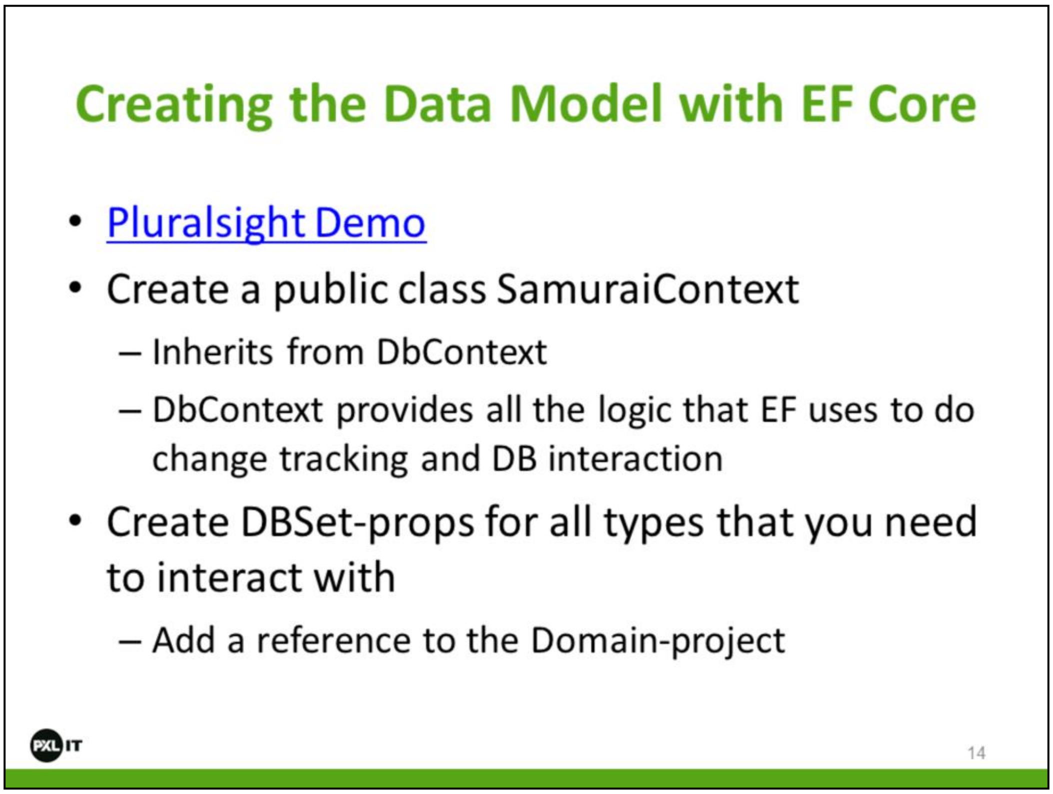

Adding EF Core

- Pluralsight Demo
- Use Nuget Package Manager
  - Install it to the “Data”-project
  - Don’t install the old version 6!
  - Search for *microsoft.entityframeworkcore*
  - Start with the package that holds the dataprovider for the datastore
    - E.g: Microsoft.EntityFrameworkCore.SqlServer
    - this brings all dependencies along
- Package Manager Console

13

In Package Manager Console, type:

install-package Microsoft.EntityFrameworkCore.SqlServer



Creating the Data Model with EF Core

- Pluralsight Demo
- Create a public class SamuraiContext
  - Inherits from DbContext
  - DbContext provides all the logic that EF uses to do change tracking and DB interaction
- Create DBSet-props for all types that you need to interact with
  - Add a reference to the Domain-project

14

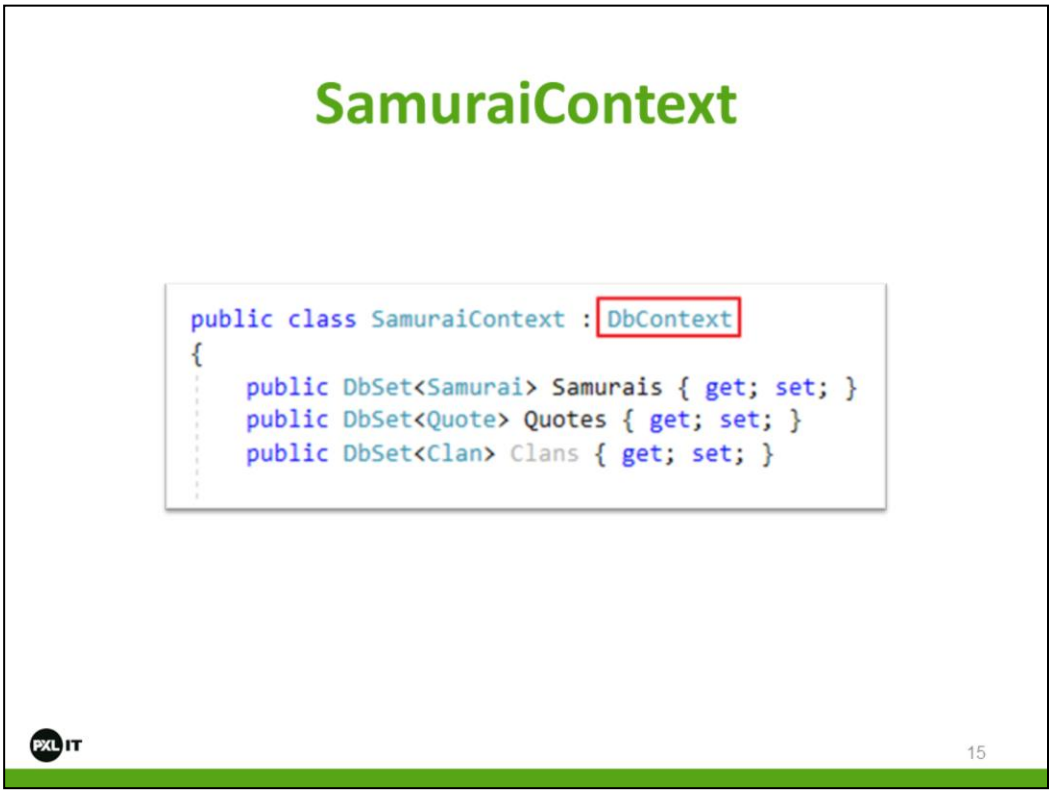
Now that Entity Framework Core is in the data project, I can go ahead and **build a data model to drive the persistence of my domain classes into the SQL Server database** that I'll be using.

I'll create a new class file and name it *SamuraiContext* because it will be using EF Core's ***DBContext*** class.

My class needs to be public and to inherit from EF Core's *DBContext*.

The ***DBContext*** **will provide all of the logic that EF Core is going to be using to do it's change tracking and data base interaction tasks**.

A *DBContext* needs to expose ***DbSets***, which become **wrappers to the different types that you'll interact with** while you're using the context.



SamuraiContext

```
public class SamuraiContext : DbContext
{
    public DbSet<Samurai> Samurais { get; set; }
    public DbSet<Quote> Quotes { get; set; }
    public DbSet<Clan> Clans { get; set; }
```

The first *DbSet* that I want to create is the *DbSet* of *Samurai* type, and I'll call that Samurais.

The data project does need a reference to the domain project, so it can find the Samurai and the other domain classes.

Now I'll add in *DbSets* for the quote and for the clan.

How do you define your *DBContext* is important to how EF Core treats your data at runtime, as well as how it's able to interact with your database, and it can define how you use the model in your coding.

Because I've exposed all three of these *DbSets*, I'll be able to directly query the *Samurais*, or the *Quotes*, or the *Clans*.

EF Core also comprehends the relationships that it discovers in the entity classes that we've just made it aware of.

And based on what the *DBContext* will discover from these classes, it will use its own **conventions** for how **to infer the relationships**.

It will see that there's a one-to-many relationship between *Samurai* and *Quote*, and that there's a relationship between *Samurai* and *Clan*, where any number of Samurais can belong to a single Clan.

The *DBContext* also affects how **EF Core infers the database schema**, whether you're working with an existing database or you're going to let EF Core create the database using this model.

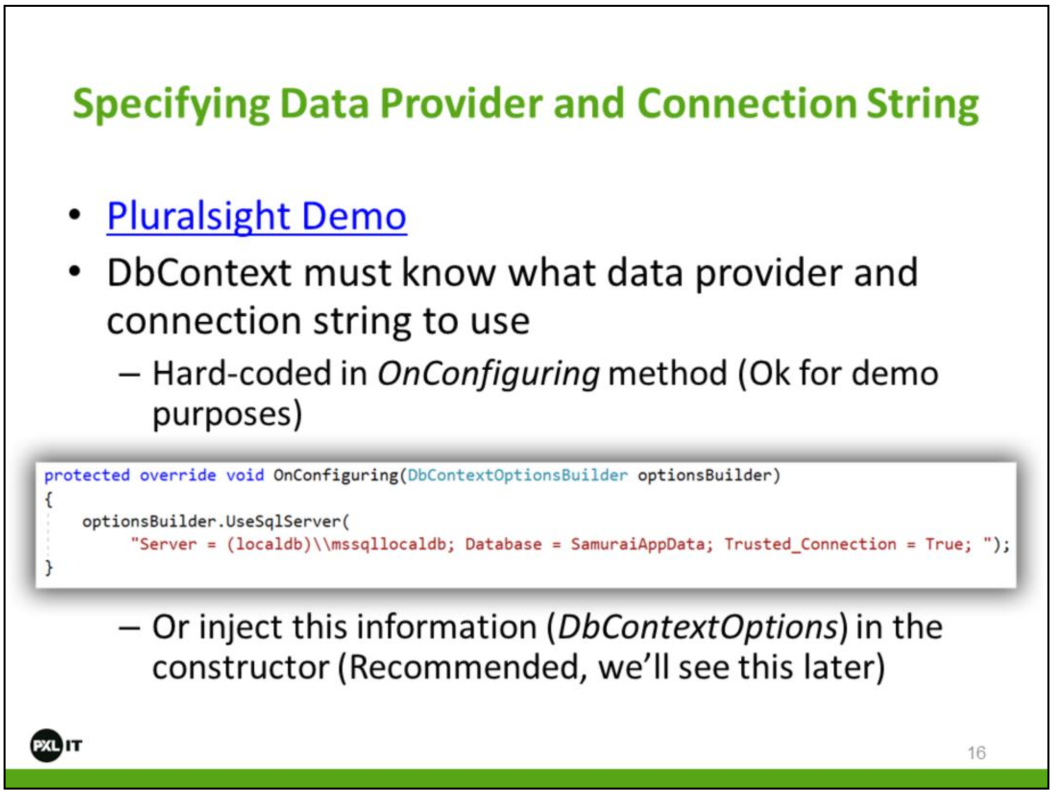
For example, EF Core will presume that the table names match these *DbSet* names.

There's a lot you can do to affect how EF Core infers that model at runtime as it reads the *DBContext* and any other information you might provide.

But for now, I'll just stick with EF Core's defaults, given that I've already made it aware of the *Samurai's*, the *Quotes*, and the *Clans*.

Further on in this course, I'll spend some time providing you with more information about defining the *DBContext.*

I'll give you some tips and tricks on taking more control over how EF Core interprets your data model by way of this context.



Specifying Data Provider and Connection String

- Pluralsight Demo
- DbContext must know what data provider and connection string to use
  - Hard-coded in *OnConfiguring* method (Ok for demo purposes)

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

  - Or inject this information (*DbContextOptions*) in the constructor (Recommended, we'll see this later)

PXL IT 16

Even though you've already seen me bring EF Core's SQL Server provider into this project, my *DbContext* is still completely unaware of that provider.

With EF Core, **you need to explicitly tell the context what data provider to use and explicitly give it a connection string**.

There are a few ways to do this.

One is directly in the DbContext class, and that's what I'll do here.
Now, hard coding the connection string into the context is great for demos and first looks, but not for your real software.

Later, you'll see me take advantage of some features of ASP.NET Core to inject this information into a *DbContext* at runtime.

In EF Core, the *DbContext* has a virtual method called ***OnConfiguring***.

I'll add that into my class using the override snippet.

OnConfiguring will get called internally by EF Core as it's working out what goes in the model, and it will also pass in an *optionsBuilder* object.

You can use that *optionsBuilder* to configure options for the *DbContext*.

Notice that the optionsBuilder has a method called *UseSqlServer*.

That's an extension method, and it's available because I've got a reference to the Microsoft Entity Framework Core's SQL Server API.

If I had pulled in SQLite, I would see a method called *UseSqLite* here.

So I'll choose the *UseSqlServer* method, which expects a parameter, that's the connection string.

And this is how, in a single statement, I can tell the *DbContext:* “*hey, you're going to be working with SQL Server and here's the connection string to the database that I want you to use*”.

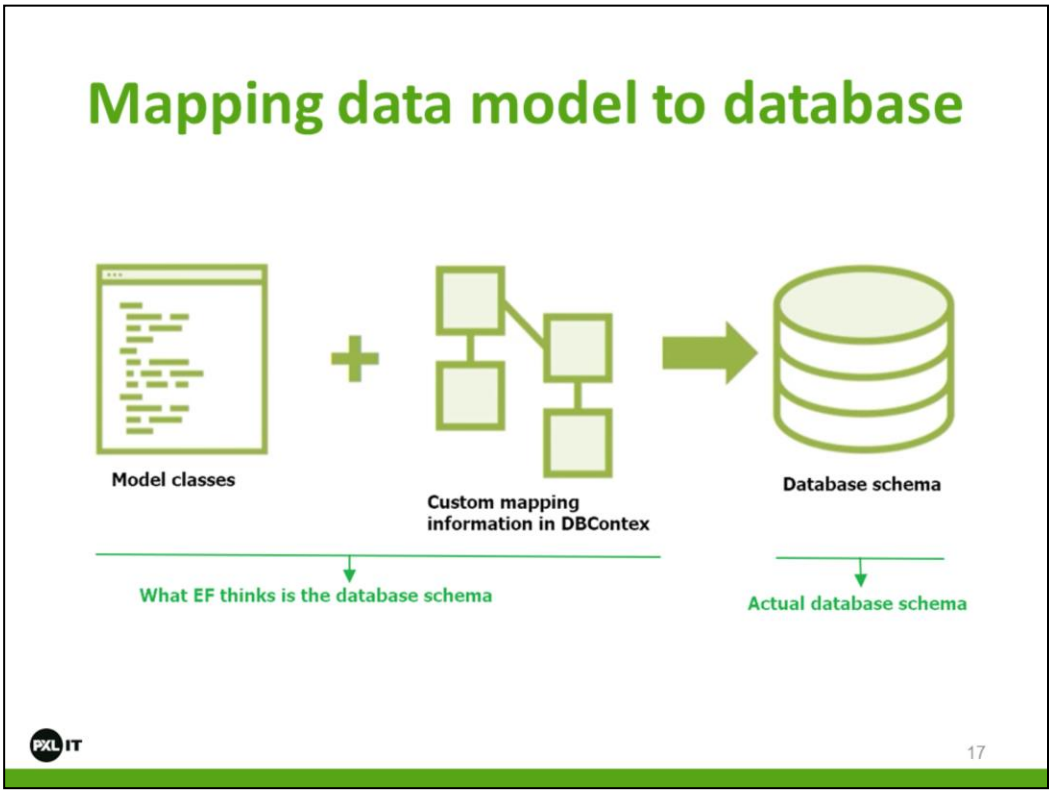
Now in my case, the database doesn't exist yet, but there are a number of ways to let EF Core help you create a database.

And that's possible because EF Core is able to read the model that this *DbContext* describes based on the classes that it's pointing to.

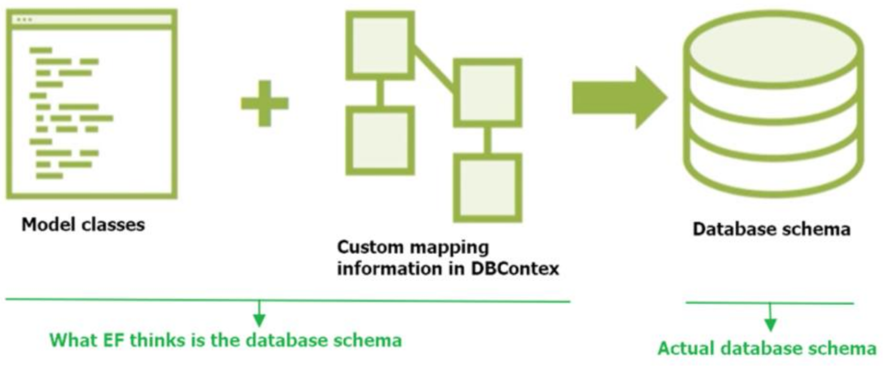
And it can do it on the fly, at runtime, or you can explicitly do this at design time, and you even have the option of letting it create the database directly or just having it create SQL scripts.

The first time EF Core instantiates the *SamuraiContext* at runtime, it will trigger the *OnConfiguring* method, learn that it should be using the SQL Server provider, and at the same time, be aware of the connection string.

So that way it will be able to find the database and do its work.



Mapping data model to database


Model classes
Custom mapping information in DBContex
Database schema
What EF thinks is the database schema
Actual database schema

17

If you've never used Entity Framework migrations before, it's important to understand the basics of what this feature does and it's workflow before interacting with the APIs.

Here, I'll provide a quick overview. Entity

Framework needs to be able to comprehend

- how to build queries that work with your database schema
- how to reshape data that's returned from the database in order to create your objects from them
- as you modify the data, how to get that data into the database.

=> See basic workflow slide

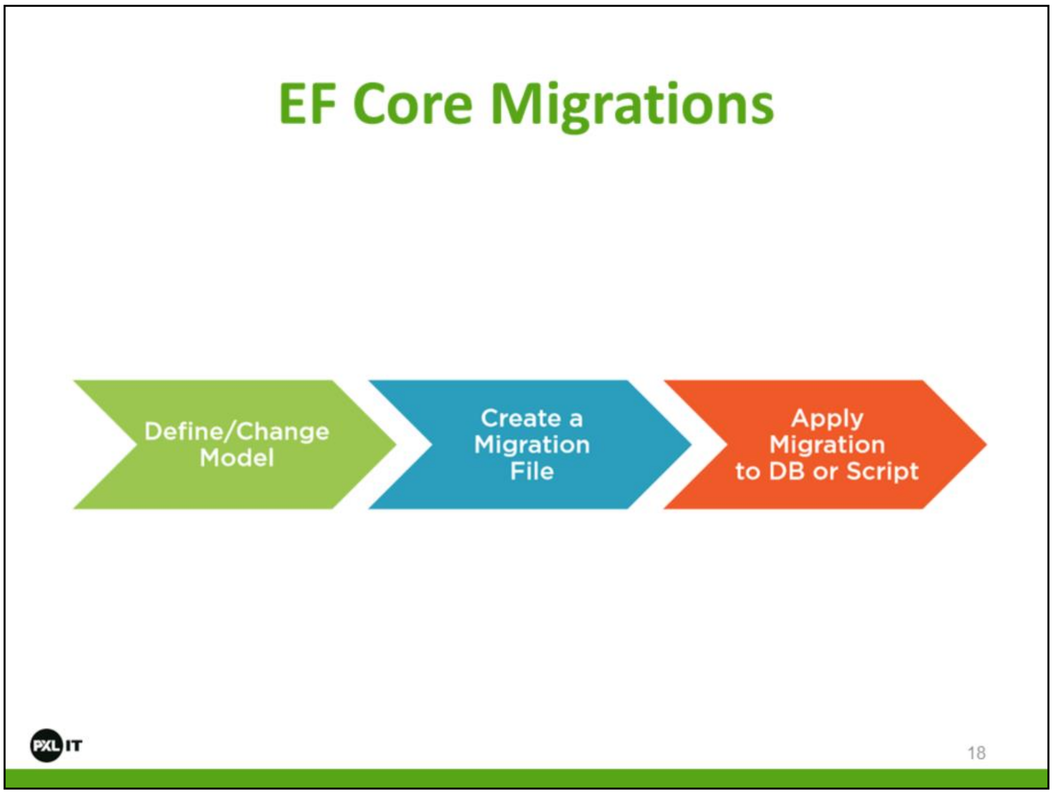
In order to do that, it has a comprehension about how the **data model** you've defined through *DBContext* **maps to the schema of the database.**

It performs that logic at runtime **by reading the *DBContext*** and using its **own conventions**, combined with any **additional custom mapping information** that you've provided to it to infer the schema of the database.

And with that information, it's able to figure out how to do those interactions, how to build queries, how to construct commands to push data to the database, and how to transform database results into your objects.

That also means if you evolve your data model, and that could be because you've made changes to the structure of your classes, or you've added more information to the *DBContext*, then Entity Framework's comprehension of the database schema will also change.

But that's just what Entity Framework thinks is in the database, so it's important to **make sure that what it thinks the database look like is actually what the database looks like**.



EF Core Migrations

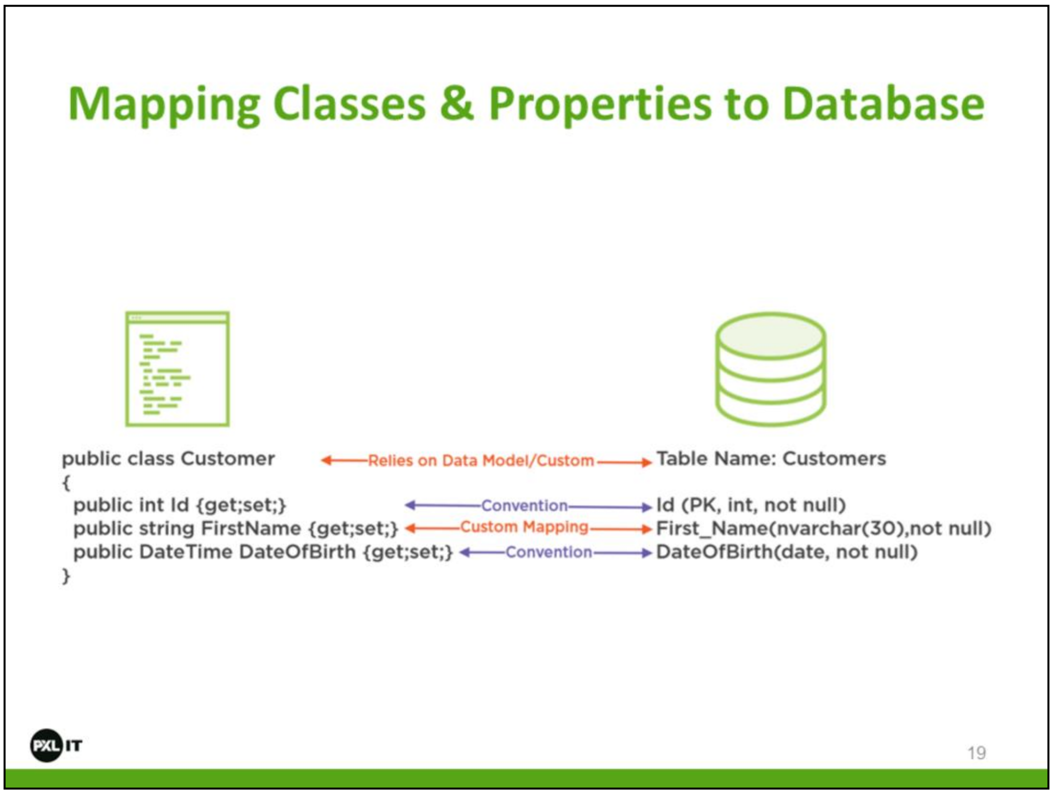
Along with a Code First paradigm, with the Entity Framework and Entity Framework Core, we also have a full set of APIs referred to as **migrations**.

With **each change to your model**, you can **create a new migration** that **describes the change**, and then let the migrations API create the proper SQL script.

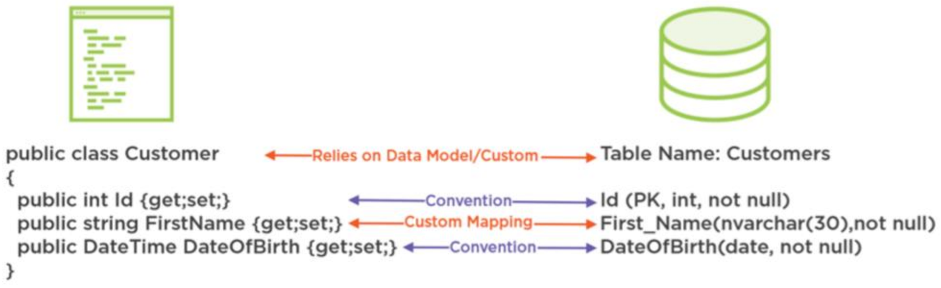
If you would like, the migrations API can also execute that script for you right on the database.

One last point, an important one that I want to make about migrations in the EF Core, is that they're designed to work easily with source control on your team.

And this is a big change that EF Core brought over earlier versions of Entity Framework. You'll see more about that as we move forward.



Mapping Classes & Properties to Database



```
public class Customer                                  Table Name: Customers
{
  public int Id {get;set;}                             Id (PK, int, not null)
  public string FirstName {get;set;}                   First_Name(nvarchar(30),not null)
  public DateTime DateOfBirth {get;set;}               DateOfBirth(date, not null)
}
```

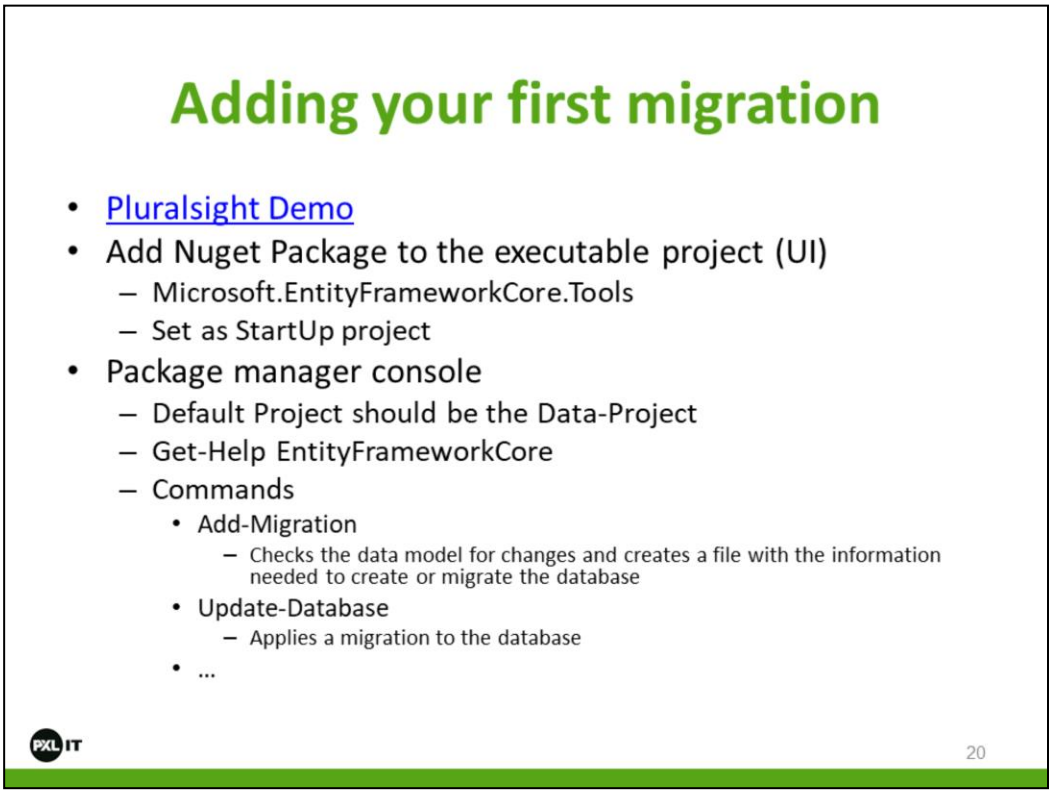
PXL IT 19

When defining the data model, your **classes aren't required to exactly match the schema of the database**.

Entity Framework does have a lot of **default rules for how it will infer what the database schema looks like** and that helps it determine what the SQL should be for commands and queries and also how to create objects from query results coming back from the database.

For example, it presumes that property names match the column names in the map table, it also makes presumptions about types in many other facets, but you can affect most of the mappings, for example, a custom mapping is helping Entity Framework understand that the *FirstName* property doesn't follow convention when it maps to the database, but instead, should map to a column named *First_Name* that's an *nvarchar* with a length of 30 and that it's not nullable.

I'll also take advantage of an EF core convention when it is on my context. That's one that ensures Entity Framework will understand that the name of the table I'm mapping to is plural. But keep in mind, these are facets of the database schema and they don't, by default, drive the business logic and validations in your code.



Adding your first migration

- Pluralsight Demo
- Add Nuget Package to the executable project (UI)
  - Microsoft.EntityFrameworkCore.Tools
  - Set as StartUp project
- Package manager console
  - Default Project should be the Data-Project
  - Get-Help EntityFrameworkCore
  - Commands
    - Add-Migration
      - Checks the data model for changes and creates a file with the information needed to create or migrate the database
    - Update-Database
      - Applies a migration to the database
    - …

20

It's time to create the first migration for the Samurai context, and here we're going to take advantage of the ConsoleApp (UI).

The **migrations tools need some type of executable** to help them run, so the data library project won't be able to do that on its own.

In order to create and execute migrations, we'll need access to the migrations commands and to the migrations logic.

Not every developer working with EF Core is going to create and execute migrations, so those are in separate packages.

The commands are in a package called **Microsoft.*EntityFrameworkCore.Tools***, which you can install using the NuGet Package Manager just like I did with the other packages.

Remember, you need to **add that to an executable project**.

So in my solution, that's the console project.

You'll run the commands in the **Package Manager console**, which is for executing PowerShell commands.

Because EF Core and the commands and the context are in the SamuraiApp.Data project, you have to be sure that the console's default project is pointed to that.

Let's take a look at what the **PowerShell commands** are for EF Core.

I can do that by typing ***get-help entityframework***.

You can see that the commands are

- Add-Migration
- Drop-Database
- Get-DbContext
- Remove-Migration
- Scaffold-DbContext
- Script-DbContext
- Script-Migration
- Update-Database.

Add-Migration will look at the *DbContext* and determine the data model.

Using that knowledge, it will **create a new migration file** with the information needed to create or migrate the database to match the model.

Update-Database **applies the migration** to the database.

Since the task right now is to add a new migration, I won't delve further into the other commands, and we'll just focus on adding migration and updating the database.

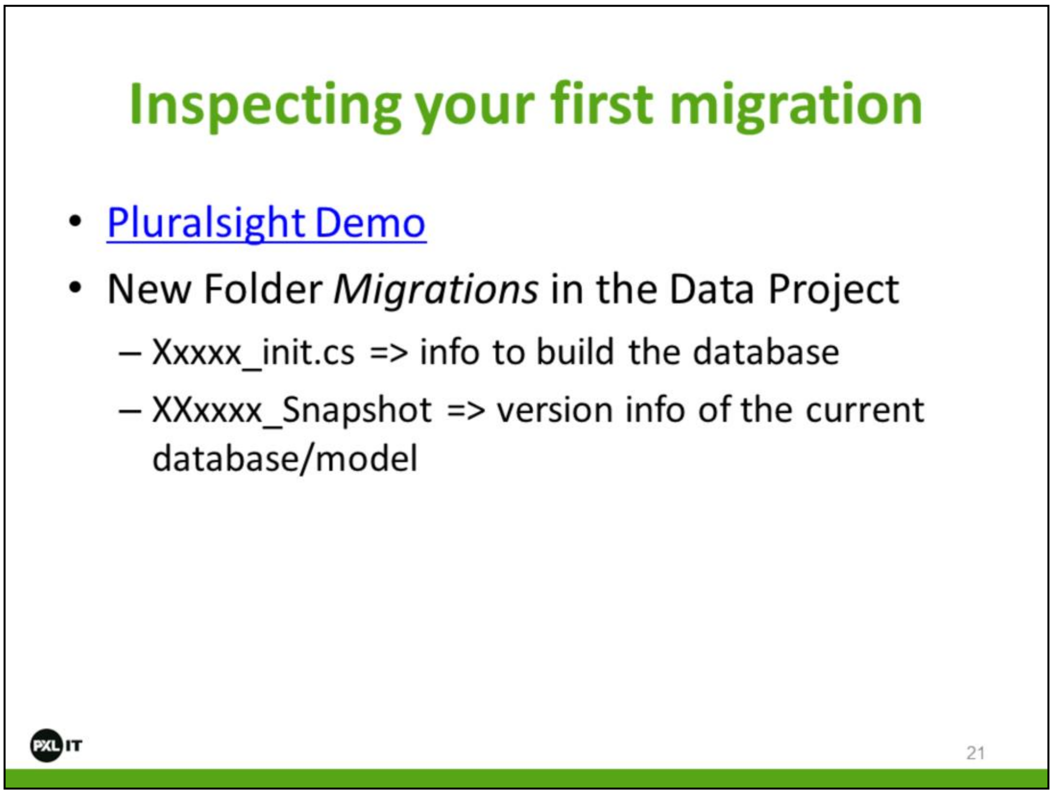
Add-Migration has a number of parameters, but I'm only going to use the required parameter, the name of the migration file I'm creating.

And you can see right away I get an error message.

The error says that the startup project, which currently happens to be *SamuraiApp.Data*, targets NETStandard but needs a runtime in order to execute the command.

I need to make sure the startup project and the solution are set to the ConsoleApp.

After making that change, I can go back to the console with a default project still pointed to data since that's where the migrations need to go, and I can run the *Add-Migration* command again, and this time it succeeds.



Inspecting your first migration

- Pluralsight Demo
- New Folder *Migrations* in the Data Project
  - Xxxxx_init.cs => info to build the database
  - XXxxxx_Snapshot => version info of the current database/model

21

Let's take a look at the migration I just created.

Notice that there is a new Migrations folder in the data project, and in there is the new migration file called *init* along with a timestamp.

There is another file in there called **model snapshot** and that's where Entity Framework Migrations **keeps track of the current state of the model** and that's really important because next time you add a migration, EF Core will read that snapshot, compare it to the new version of the model, and that's how it figures out what needs to be changed in the schema.

The snapshot is just a file, it's part of the project, and it can participate fully in source control when you're using EF Core and Migrations across members of your team.

If you're coming from a previous version of Entity Framework and you're on a team that's using migrations, this is probably cause for celebration. This is a really important improvement that we got with EF Core.

The **migration** uses EF Core's Migration API to **describe what needs to happen in the database**, and notice the annotation in the generated file that specifically says *SqlServer:ValueGenerationStrategy*.

That's because migrations read the configuration information that we supplied in the *SamuraiContext* and knew that we're targeting SQL Server and so it interacted with the SQL Server provider to determine some important metadata.

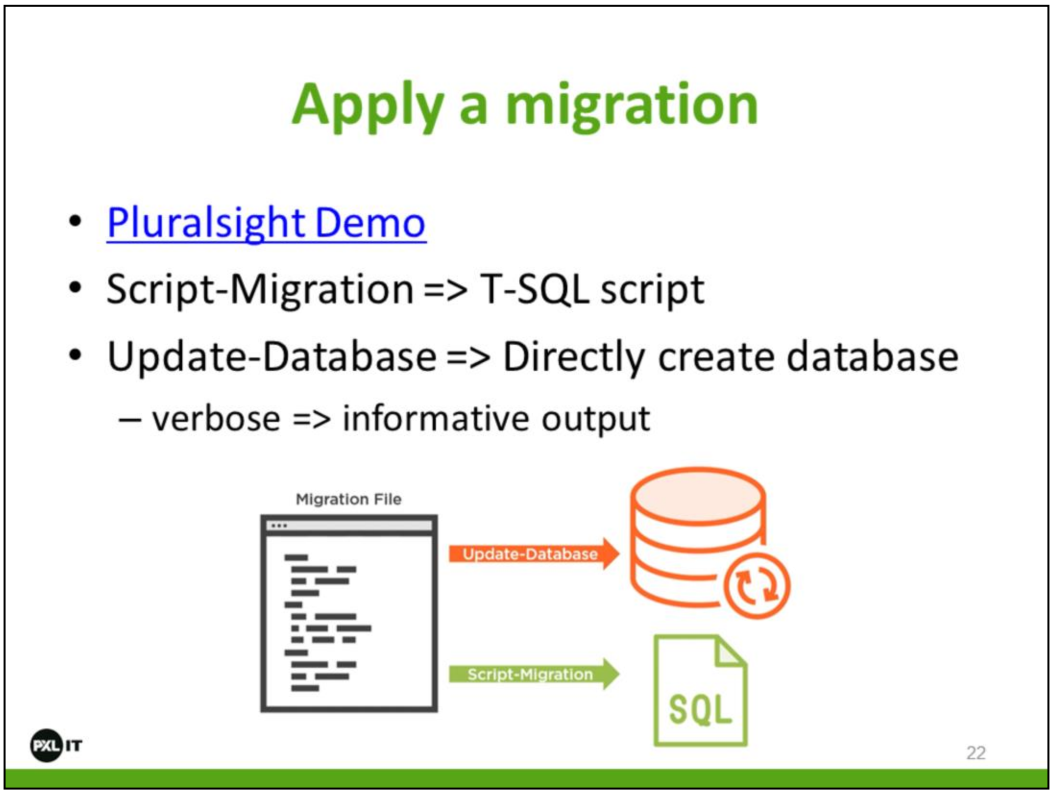
Scrolling through the migration file, you can see that it's creating tables, and for the tables, it's creating the columns along with particular attributes for those columns.

Also, notice that it's specifying primary keys and foreign keys, as well as constraints between those primary and foreign keys.

It even specified a few indexes and EF Core's conventional rule behind indexes is to create an index for every one of the foreign keys that it discovers in the model.

All of that is part of a method called *Up* and here is another method called *Down*, that's used if we ever want to unwind this particular migration.

So that's it for this first migration. The next step will be to have the migration tools create the database.



Apply a migration

- Pluralsight Demo
- Script-Migration => T-SQL script
- Update-Database => Directly create database
  - verbose => informative output

Migration File
Update-Database
Script-Migration
SQL

With the migration in place, we now have everything we need in order to create a database.

Not only do migrations give me the ability to have Entity Framework create the database for me directly, I can also have it generate a script, which is an important feature for working with a production database or sharing your development database changes with your team.

EF Core's ***script-migration*** command will build up the relevant SQL.

In this case, it's T-SQL because I'm using SQL Server based on what is discovered in the migration file.

Now with this T-SQL in hand, you can let your resident database expert take care of executing it on the production database.

They may even want to make some tweaks to it.

With my development database, I'll typically let migrations go ahead and create and update the database for me on the fly.

But with a production database, a more common scenario is to take advantage of the ability to generate the SQL and taking more control over how and when it's applied to the production database.

Alternatively, you can let EF Core migrations directly affect the database with the ***update-database*** command.

Before I run that, I do need to point out that I deleted the database, which was created by *EnsureCreated* in the previous module.

So now the database does not exist at all. In fact, if we look back at the migration, there's nothing in there that says create the database.

That's actually handled by the internal code inside of migrations, which first checks to see if the database exists or not before the migration's run.

If it doesn't exist, then EF Core will first create that database at the given location.

If you've created a script and you're running that, you will be responsible for creating the database yourself.

I won't be using any of the *update-database* parameters, although I will use a parameter that's common across PowerShell, which is the *-verbose parameter*.

This lets you see everything that the *update-database* command is doing, and you can see it's discovering the correct projects, it's rebuilding the solution, and then finding all of its assets.

Finally, it applies the migration, and it's done.

Now, after I refresh the SQL Server Object Explorer, you can see the new database, *SamuraiAppData*, and if I expand that further, you can see the new tables: *Clans*, *Quotes*, and *Samurais*.

Those pluralized names were driven by the way I named the *DbSets* in my *DbContext*.

So what about that history table?

Its job is to keep track of which migrations have been run on the database, and its columns are *MigrationId* and the *ProductVersion* that was used.

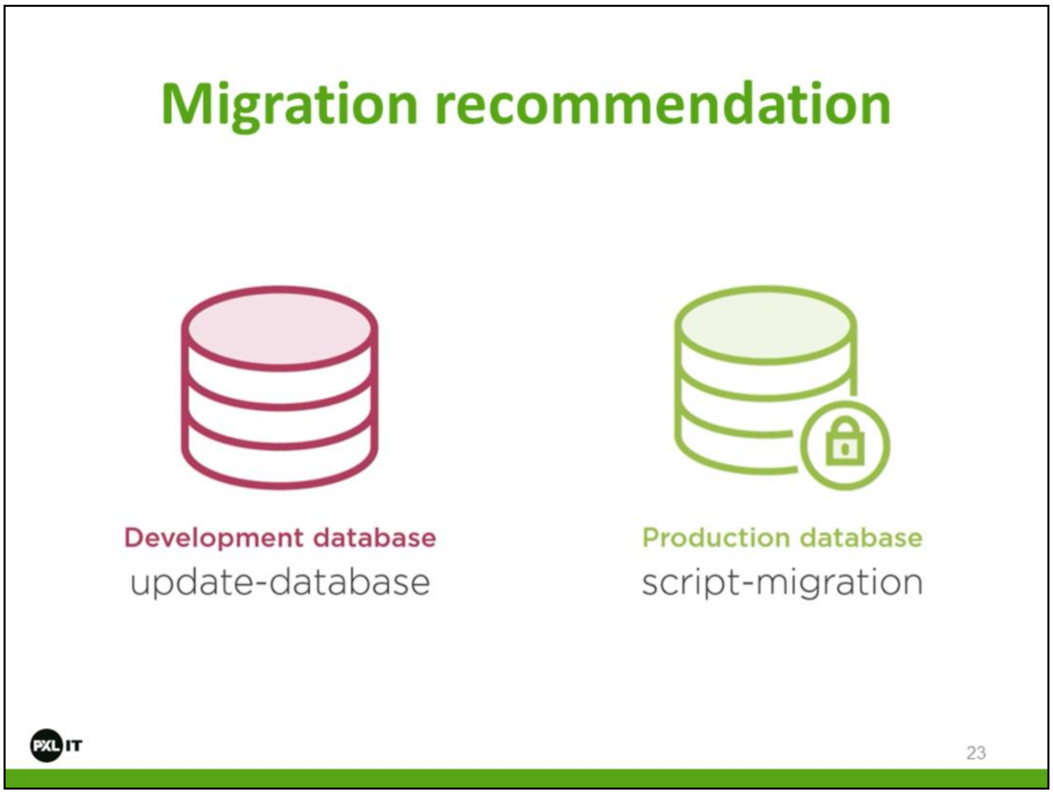
Let’s look at the tables from a different perspective by looking at the database schema (using SQL Server Object explorer or SQL Server Management Studio).

You can see that **it created primary keys**, and that's thanks to one of EF Core's **convention**s based on the fact that the property names in my classes were named *Id*.

You can also see that the default data type derived from my *string* properties is an *nvarchar(MAX)*, which is actually a default that's driven by the database provider, not by EF Core.

We saw in the migration code that it was working out not just the primary keys, but also foreign keys.

And you can see those relationships in the database schema along with the names that EF Core's conventions provided for those constraints.



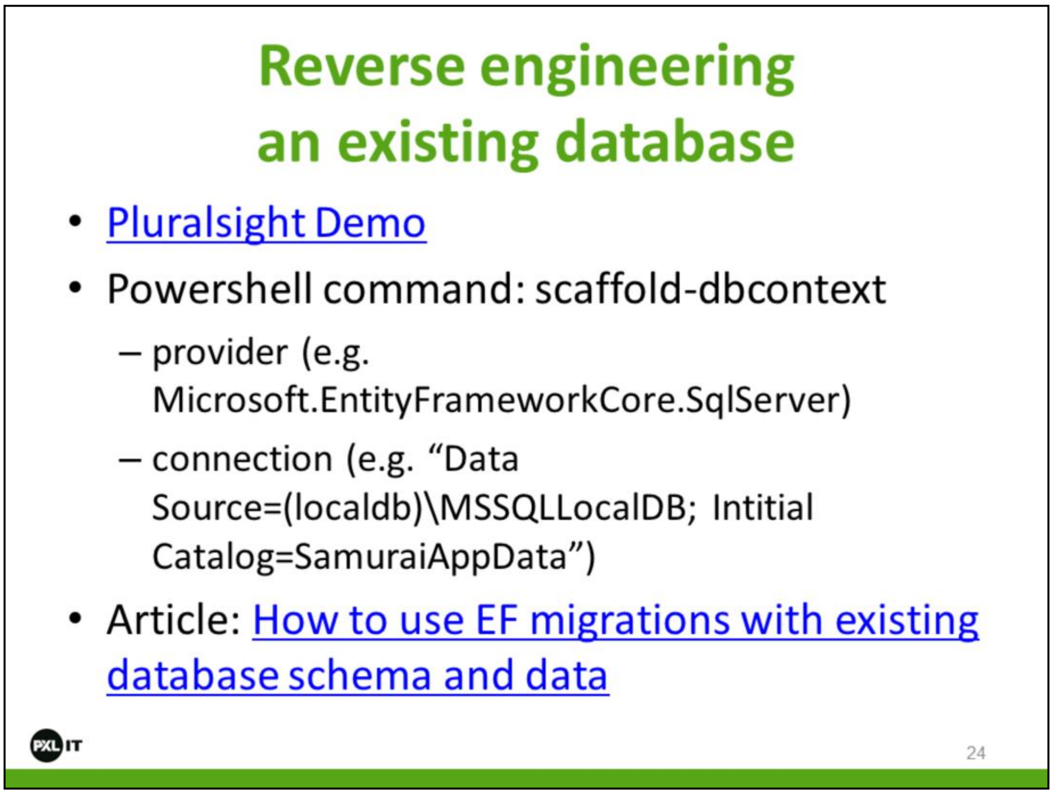
Migration recommendation

Development database
update-database

Production database
script-migration

PXL IT

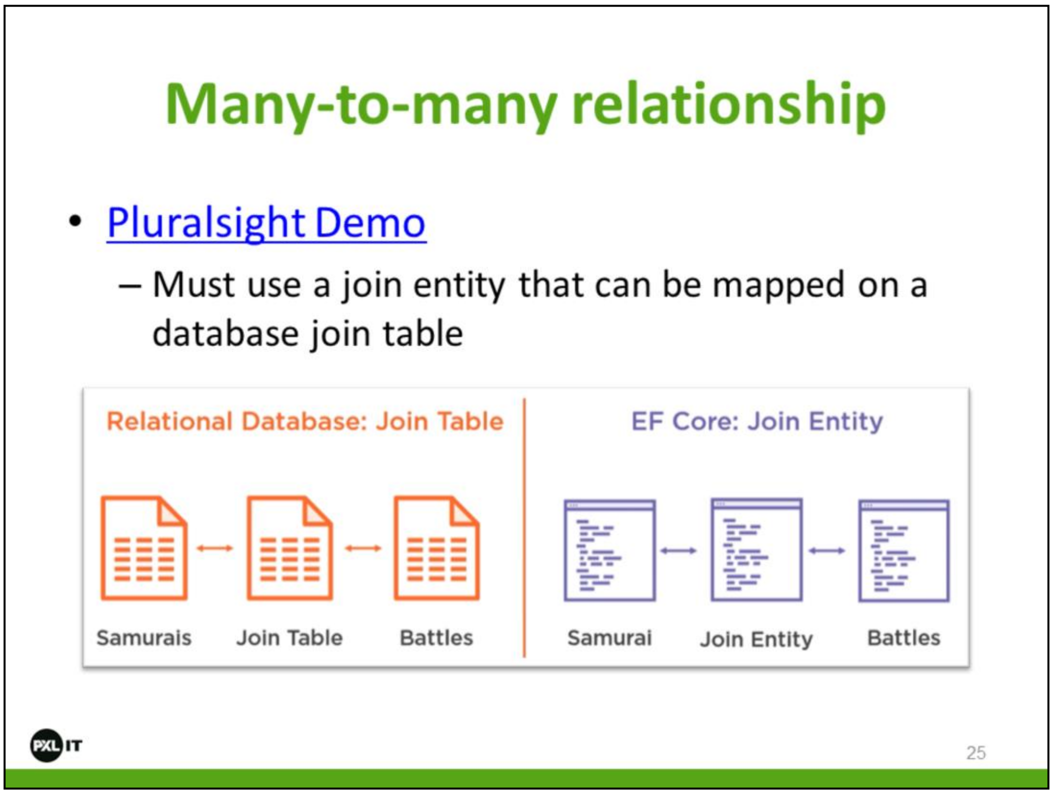
23


Reverse engineering an existing database

- Pluralsight Demo
- Powershell command: scaffold-dbcontext
  - provider (e.g. Microsoft.EntityFrameworkCore.SqlServer)
  - connection (e.g. “Data Source=(localdb)\MSSQLLocalDB; Intitial Catalog=SamuraiAppData”)
- Article: How to use EF migrations with existing database schema and data

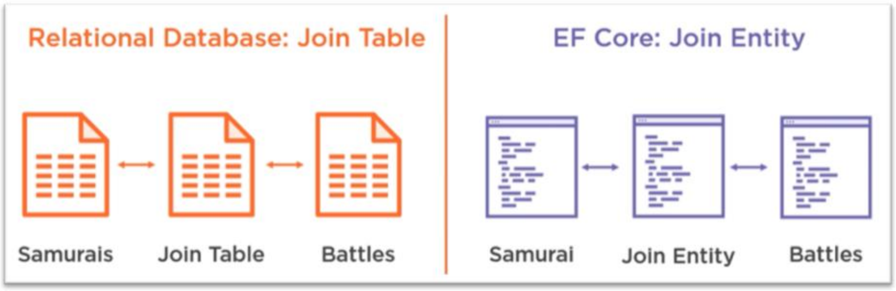
24

Article: How to use EF migrations with existing database schema and data



Many-to-many relationship

- Pluralsight Demo
  - Must use a join entity that can be mapped on a database join table



25

It's important to understand how many-to-many relationships work in EF Core, and also why.

One of the big jobs of a samurai is to fight in battles, so I'll be adding a new battle class into the domain in order to keep track of information about the battle.

And then we'll need to introduce a many-to-many relationship between samurai and battle.

That will let us keep track of all the battles the samurai fought in and all the samurais that fought in a particular battle.

In EF Core you have to represent a **many-to-many** relationship explicitly with a **join entity**.

This aligns with how it's done in a relational database.

There are pros and cons in using a join entity:

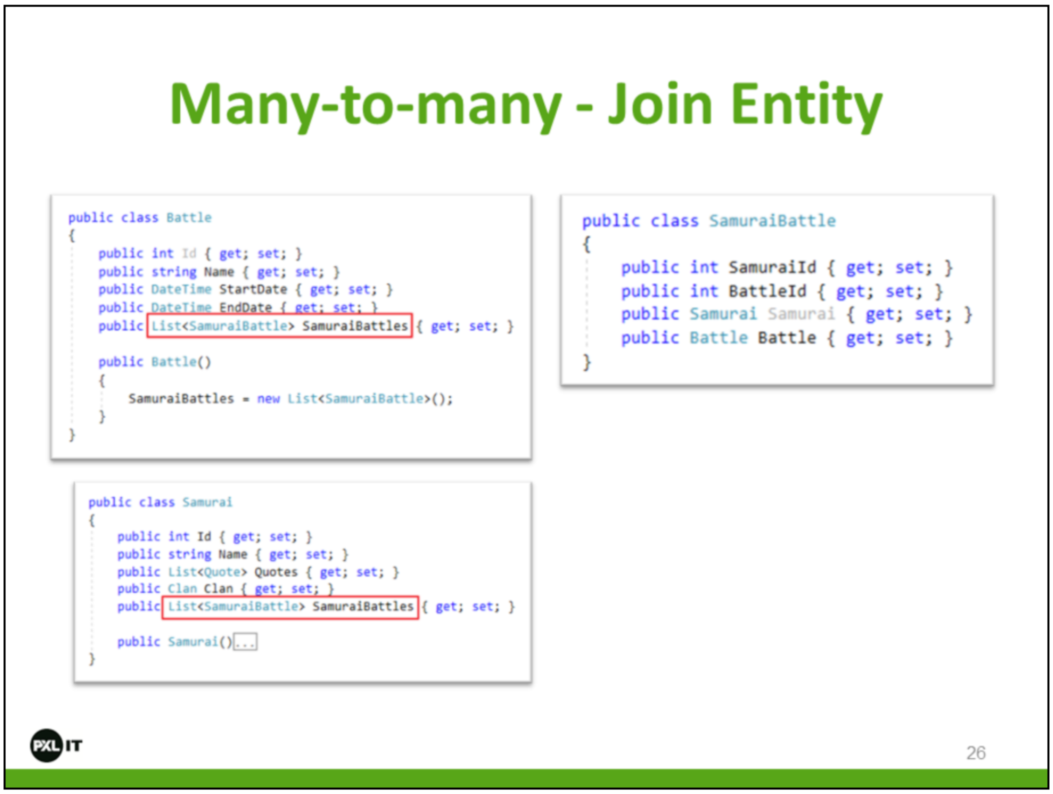
- When defining my business objects, I do want to be able to easily navigate from a samurai to all the battles which the samurai's fought in,
  and I would like to easily be able to look at a battle, and then see all of the samurais that fought in a particular battle.

Even if I design my entity classes this way, EF Core isn't able to recognize this pattern in order to persist the data.
  And because we have to put that join entity in between, it does create some complications for us in how we interact with that data in our code.
  So from that perspective, being forced to use the join entity does present a problem;

- However, here's another perspective.
  I don't like a lot of magic to happen that I can't control. EF would need to do some tricky transformations behind the scenes in order to persist many-to-many relationships to the database.
  That's fine when things are simple, but when you run into complications, it's a little more difficult to sort out because you're not in control of how it works.

For this course, it's best to take the simple path and follow the pattern prescribed by the EF team, which is to implement the join entity.

And later in the course, you'll see how to interact with these classes, including the join entity, when retrieving and saving your data.



Many-to-many - Join Entity

```
public class Battle
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime StartDate { get; set; }
    public DateTime EndDate { get; set; }
    public List<SamuraiBattle> SamuraiBattles { get; set; }

    public Battle()
    {
        SamuraiBattles = new List<SamuraiBattle>();
    }
}
```

```
public class SamuraiBattle
{
    public int SamuraiId { get; set; }
    public int BattleId { get; set; }
    public Samurai Samurai { get; set; }
    public Battle Battle { get; set; }
}
```

```
public class Samurai
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Quote> Quotes { get; set; }
    public Clan Clan { get; set; }
    public List<SamuraiBattle> SamuraiBattles { get; set; }

    public Samurai()...
}
```

PXL IT

26

Let’s take a closer look at the new classes.

The **Battle** class got its *Id*, the *Name* of the battle, and its *StartDate* and *EndDate*.

In addition to the new *Battle* class, I've created the join entity class named ***SamuraiBattle***.

This has a *SamuraiId* and a *BattleId* property, and these are required because they'll become foreign keys pointing back to the *Samurai* class and the *Battle* class.

I've also added navigation properties to *Samurai* and *Battle*, but these are optional.

Whether or not you need them depends on how you plan to interact with these objects in your business logic, but they're not needed by EF Core in order to comprehend the model and how to persist it.

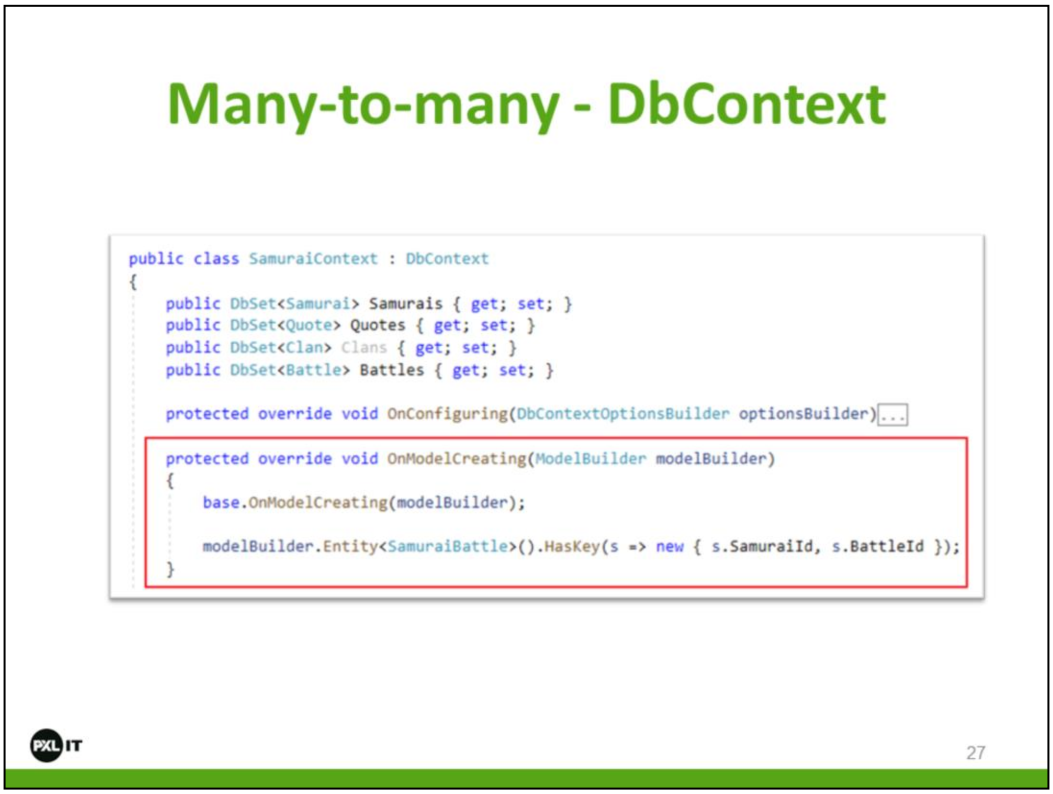
Next, I need a list of *SamuraiBattles* in the ***Samurai*** class, and we'll need to instantiate that in the constructor, so you don't run into any surprises because of it being null.

And we also need to add the *SamuraiBattles* list into the *Battle* class and instantiate in its constructor as well.

And with those, it's now possible for a Samurai to have many *SamuraiBattles*, which will connect it to any of the battles that those lead me to.

In the *Battle* class, the *SamuraiBattles* list will connect the battle back to the *Samurai* represented in each of those connections.

The classes involved in the many-to-many are now set up properly, but **EF Core can't infer this relationship on its own**…



Many-to-many - DbContext

```
public class SamuraiContext : DbContext
{
    public DbSet<Samurai> Samurais { get; set; }
    public DbSet<Quote> Quotes { get; set; }
    public DbSet<Clan> Clans { get; set; }
    public DbSet<Battle> Battles { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)...

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<SamuraiBattle>().HasKey(s => new { s.SamuraiId, s.BattleId });
    }
```

27

We have to give EF Core some more information so that it can map to the database, as well as translate queries and updates.

Here in the SamuraiContext class, I'm using the **Fluent API** to specify the last critical detail of the many-to-many relationship.

The place to add custom mappings (from data model to database schema) is in het DBContext's ***onModelCreating*** method.

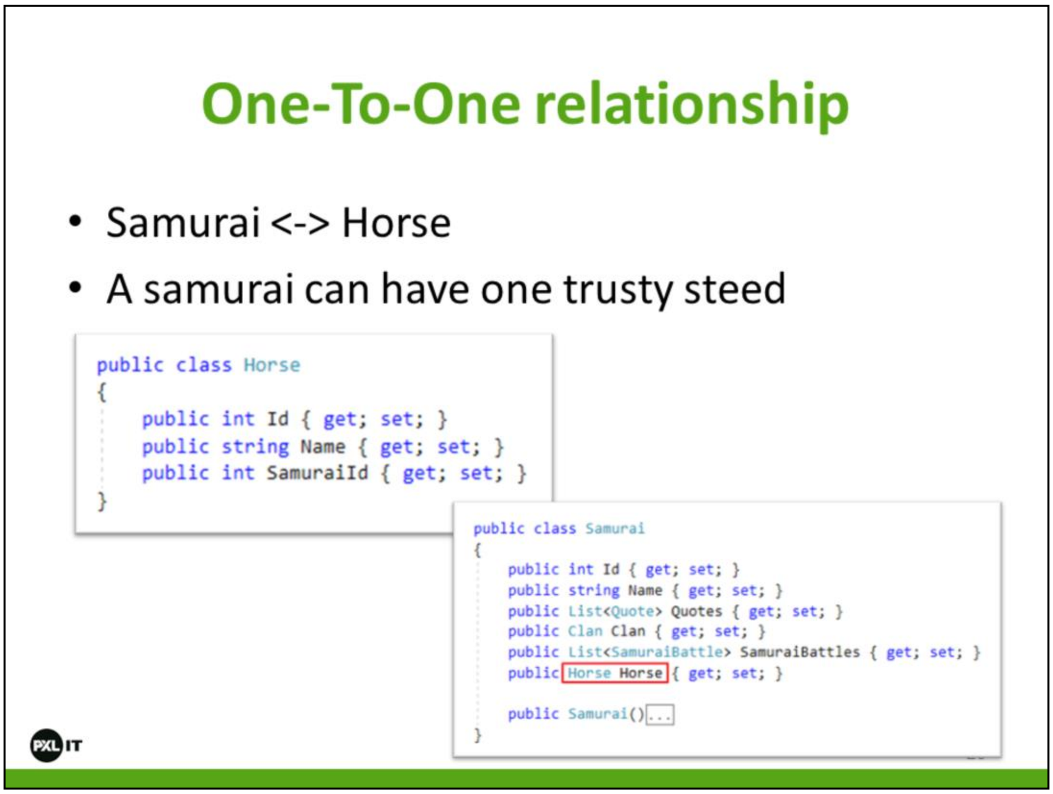
This method **gets called internally at runtime when EF Core is working out what the data model looks like**.

Using the *modelBuilder* object that EF Core has passed into the method, I've told it that the *SamuraiBattle* Entity has a key that's composed from its *SamuraiId* and *BattleId* properties.

Now EF Core will be able to understand the database schema that's related to this, as well as to build the SQL for queries and database updates that respect this many-to-many relationship.

Fluent API reference:

https://www.learnentityframeworkcore.com/configuration/fluent-api



One-To-One relationship

- Samurai <-> Horse
- A samurai can have one trusty steed

```
public class Horse
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int SamuraiId { get; set; }
}
```

```
public class Samurai
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Quote> Quotes { get; set; }
    public Clan Clan { get; set; }
    public List<SamuraiBattle> SamuraiBattles { get; set; }
    public Horse Horse { get; set; }

    public Samurai()...
}
```

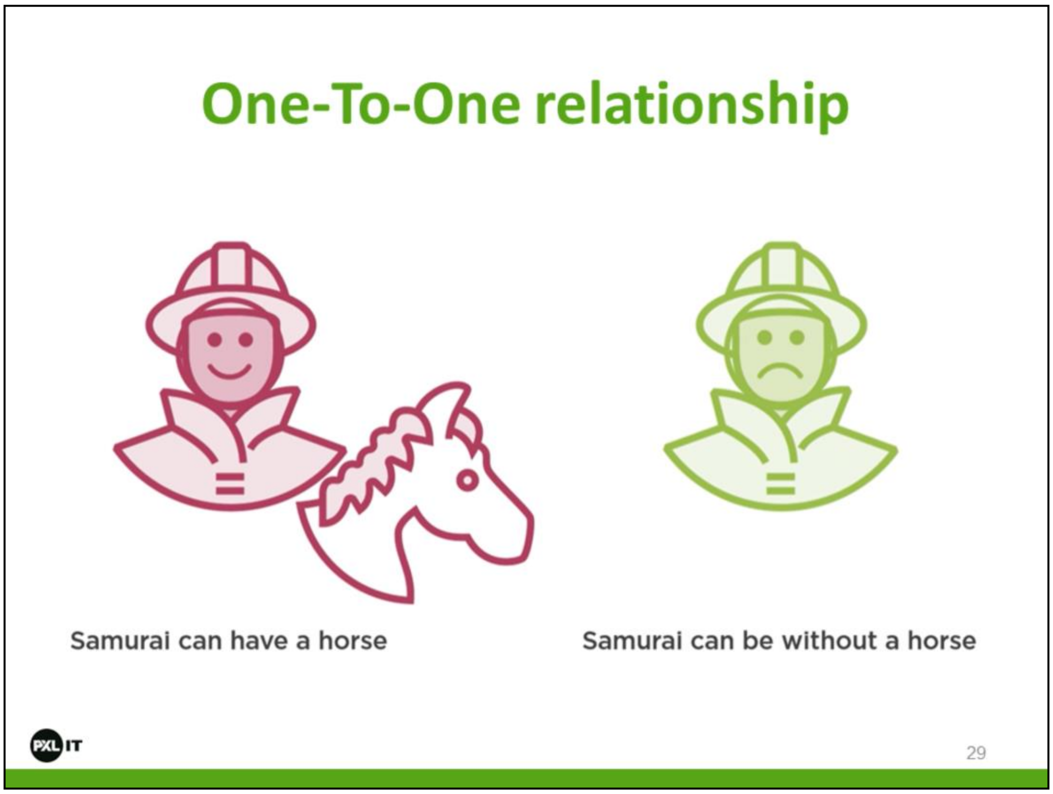
Now let's move on to the one-to-one relationship.

I've created a new class called *Horse* to keep track of each samurai's trusty steed.

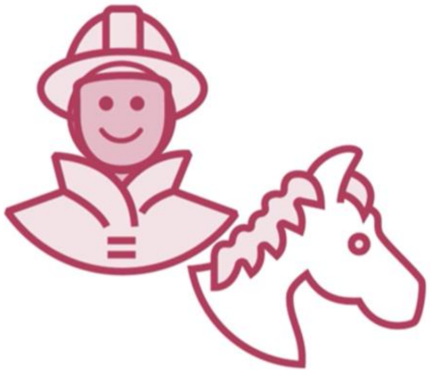
The new class has an Id, the horse's name, and the Id of the samurai that it belongs to.

The *Samurai* class now has a navigation property to define the samurai's horse.

With the navigation property in *Samurai* and the foreign key in the *Horse* class, it's enough for EF Core to comprehend the relationship without any more information from me.



One-To-One relationship



Samurai can have a horse

Samurai can be without a horse

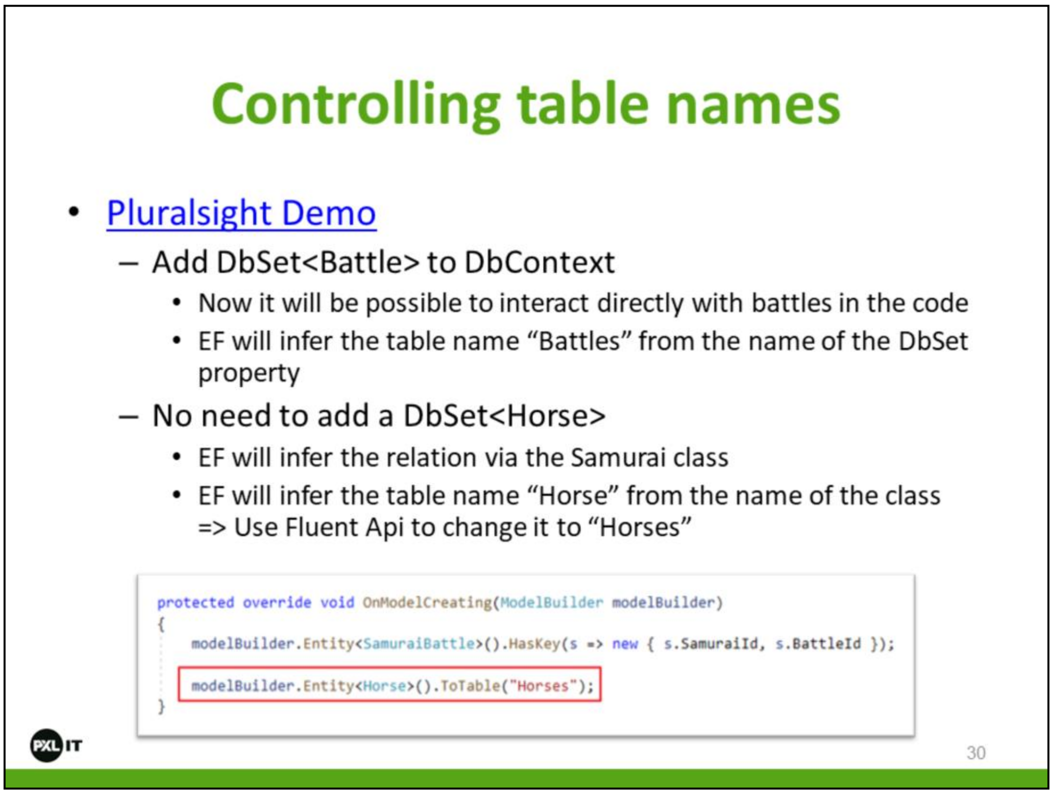
29

Keep in mind that the dependent end of a one-to-one relationship, in this case the horse, is always optional as far as EF Core is concerned.

There's no way to apply that constraint in the model or, for that matter, in the database.

If you want to require that samurai always has a horse, you'll have to do that in your business logic.

But I'll keep that simple, and we just might have to have some samurais that have to walk.


Controlling table names

- Pluralsight Demo
  - Add DbSet<Battle> to DbContext
    - Now it will be possible to interact directly with battles in the code
    - EF will infer the table name "Battles" from the name of the DbSet property
  - No need to add a DbSet<Horse>
    - EF will infer the relation via the Samurai class
    - EF will infer the table name "Horse" from the name of the class => Use Fluent Api to change it to "Horses"

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<SamuraiBattle>().HasKey(s => new { s.SamuraiId, s.BattleId });

    modelBuilder.Entity<Horse>().ToTable("Horses");
}
```

30

There is one last change to make, and that's in the *DbContext*.

Further on in the course, I want to be able to interact directly with battles through the *DbContext*, so I've added a new *DbSet* called *Battles*.

EF Core will also be able to find the *Horse* class because it's connected to the *Samurai*.

However, by **default**, EF Core will create a **table name that matches the name of the class since I haven't specified a *DbSet***.

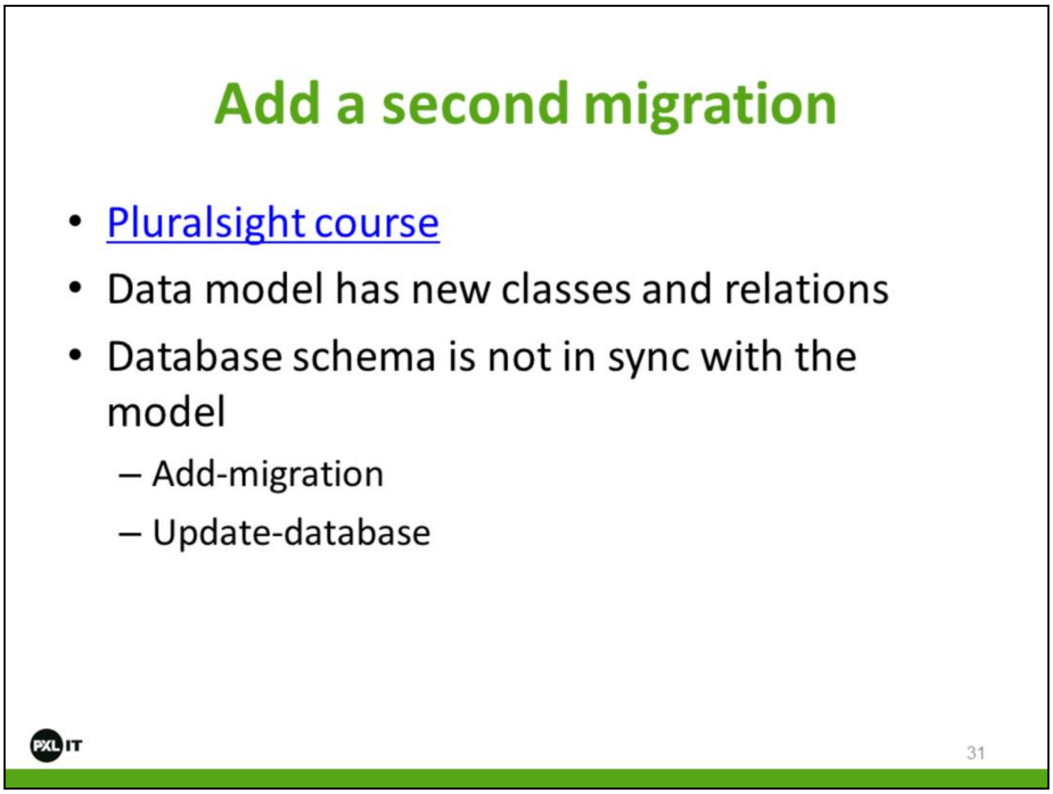
And that's going to be singular, horse, which doesn't align with my convention for table names.

But I don't want to add a *DbSet* just for that reason because then someone working with my API might be tempted to interact directly with horses, which doesn't align with my business rules.

There is another way to affect the table name, and this is another great example of how flexible the mappings are.

I can use the **fluent API** or data annotations to change that.

So in *OnModelCreating*, I've added a line of code to tell the *modelBuilder* that the *Horse* entity maps to a table called *Horses*, and I'm using the ***ToTable*** method to do that.

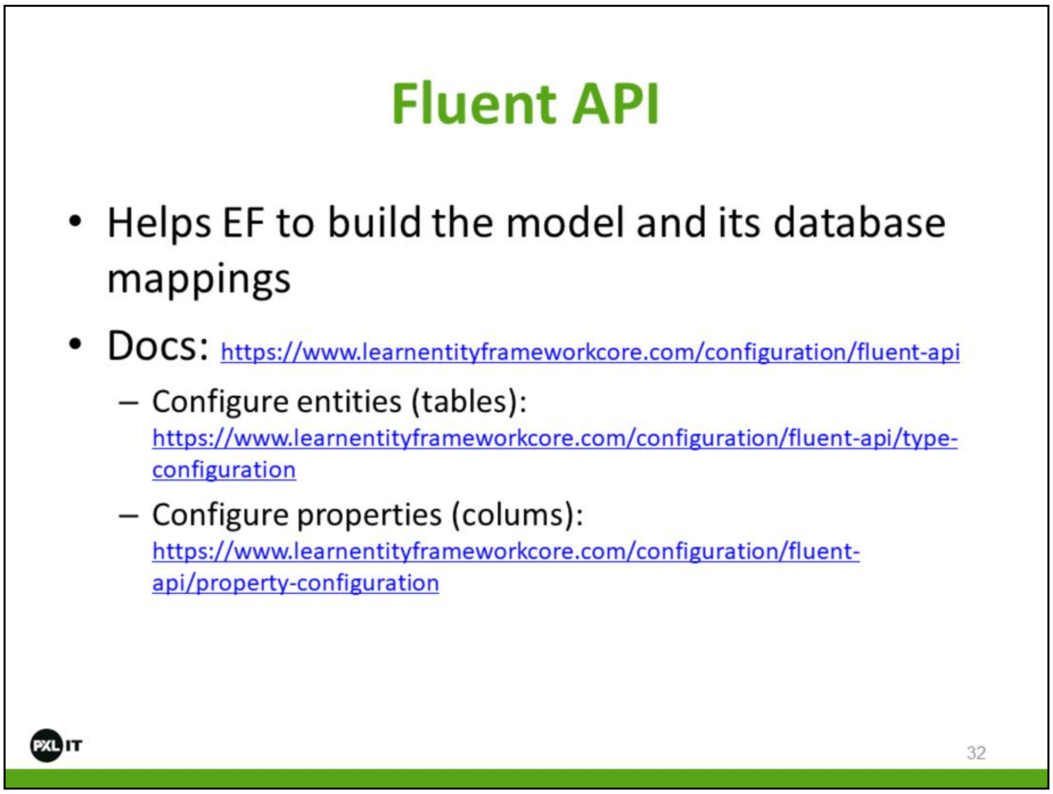


Add a second migration

- Pluralsight course
- Data model has new classes and relations
- Database schema is not in sync with the model
  - Add-migration
  - Update-database

PXL IT

31



Fluent API

- Helps EF to build the model and its database mappings
- Docs: https://www.learnentityframeworkcore.com/configuration/fluent-api
  - Configure entities (tables): https://www.learnentityframeworkcore.com/configuration/fluent-api/type-configuration
  - Configure properties (colums): https://www.learnentityframeworkcore.com/configuration/fluent-api/property-configuration

32

Configurations are applied via a number of methods exposed by the Microsoft.EntityFrameworkCore.ModelBuilder class.

The **DbContext** class has a method called **OnModelCreating** that takes an instance of **ModelBuilder** as a parameter.

This method is called by the framework **when your context is first created to build the model and its mappings in memory**.

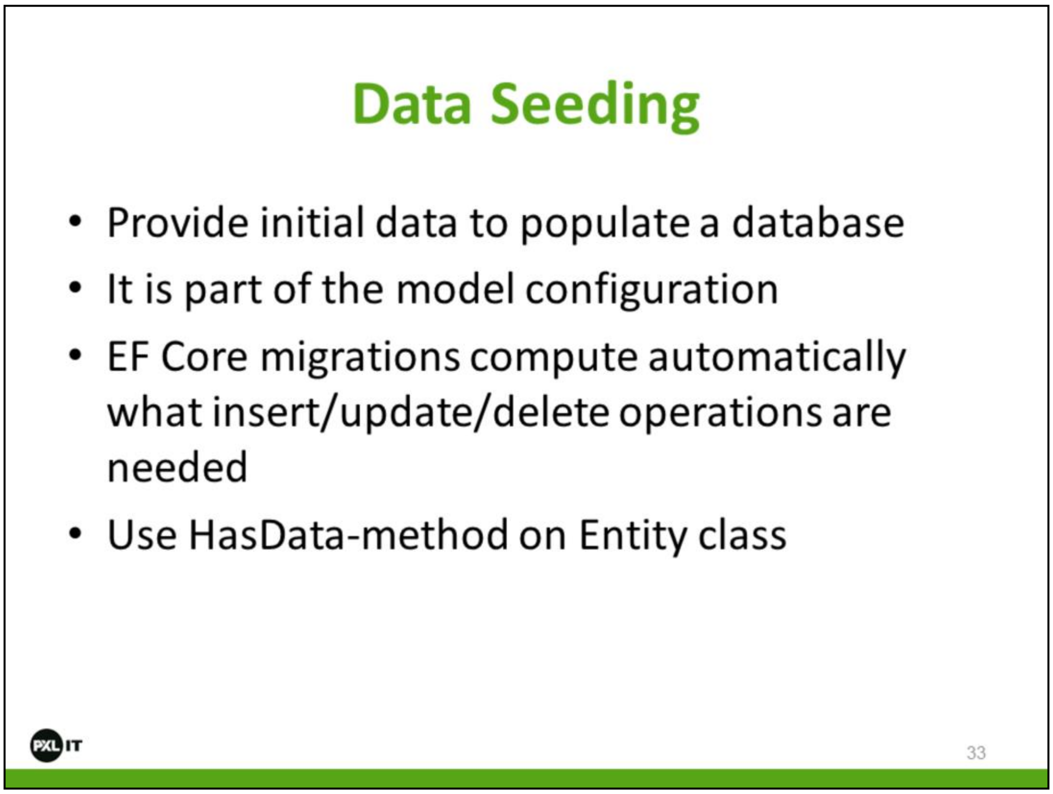
You can **override** this method to add your own configurations

Docs:
https://www.learnentityframeworkcore.com/configuration/fluent-api

Configure entities (tables):
https://www.learnentityframeworkcore.com/configuration/fluent-api/type-configuration

Configure properties (colums):

https://www.learnentityframeworkcore.com/configuration/fluent-api/property-configuration



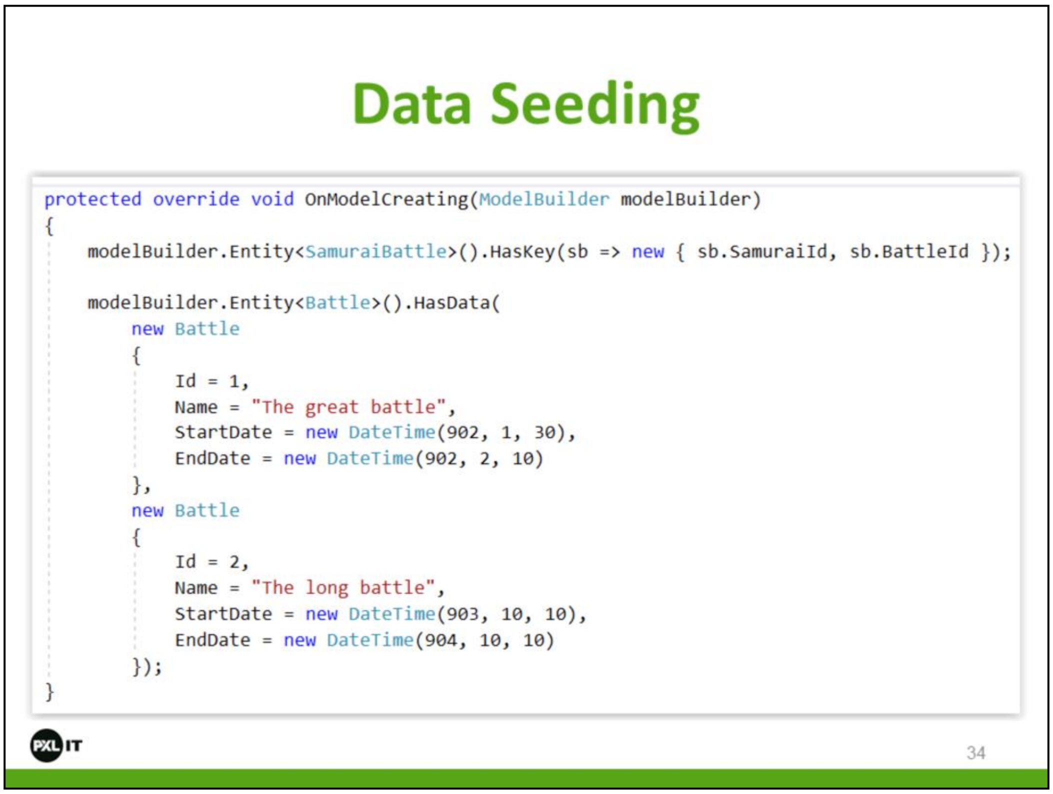
Data Seeding

- Provide initial data to populate a database
- It is part of the model configuration
- EF Core migrations compute automatically what insert/update/delete operations are needed
- Use HasData-method on Entity class

33

https://docs.microsoft.com/en-us/ef/core/modeling/data-seeding

This feature is available since EF Core 2.1



Data Seeding

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<SamuraiBattle>().HasKey(sb => new { sb.SamuraiId, sb.BattleId });

    modelBuilder.Entity<Battle>().HasData(
        new Battle
        {
            Id = 1,
            Name = "The great battle",
            StartDate = new DateTime(902, 1, 30),
            EndDate = new DateTime(902, 2, 10)
        },
        new Battle
        {
            Id = 2,
            Name = "The long battle",
            StartDate = new DateTime(903, 10, 10),
            EndDate = new DateTime(904, 10, 10)
        });
}
```

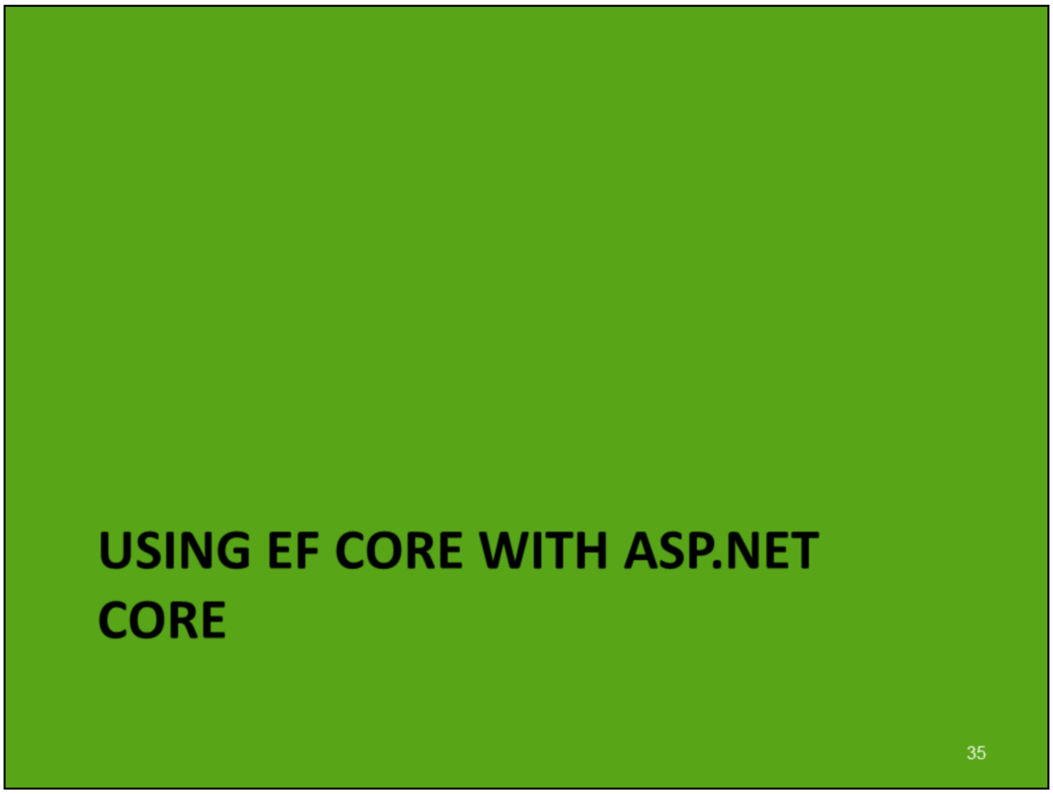
PXL IT

34

With HasData you configure entities with data. Then you perform a new migration to add these to the migration code:

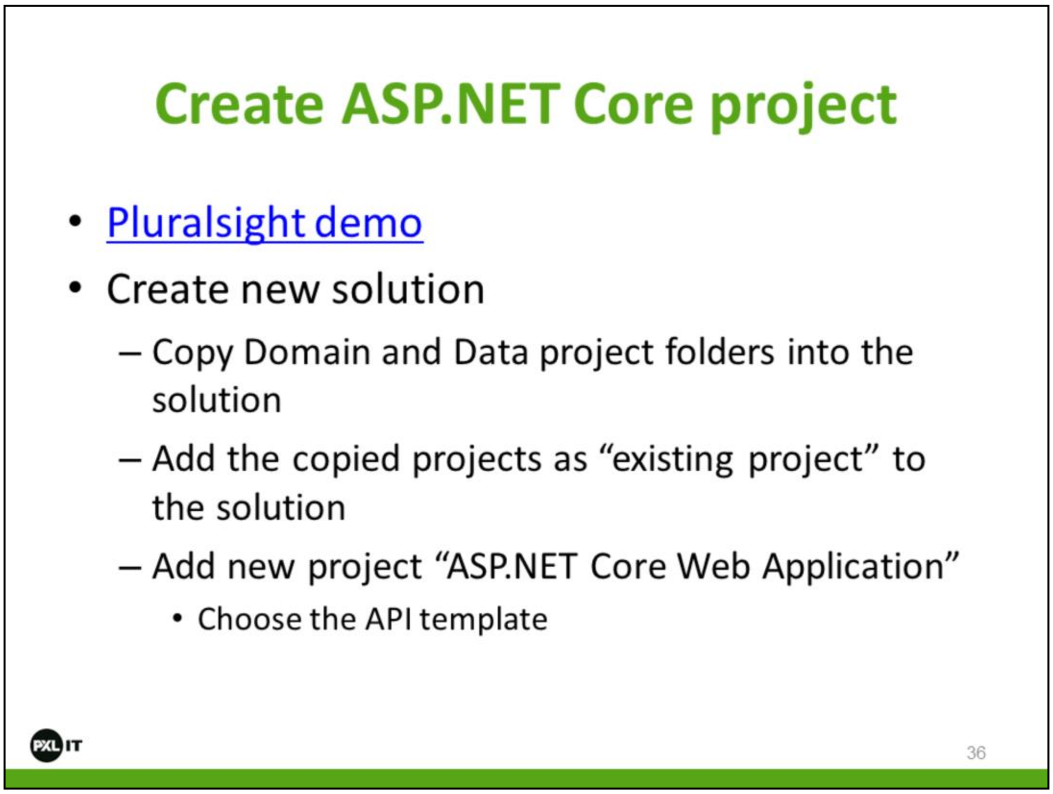
Add-Migration battledata

Update-Database triggers migrations and therefore creates the necessary data.



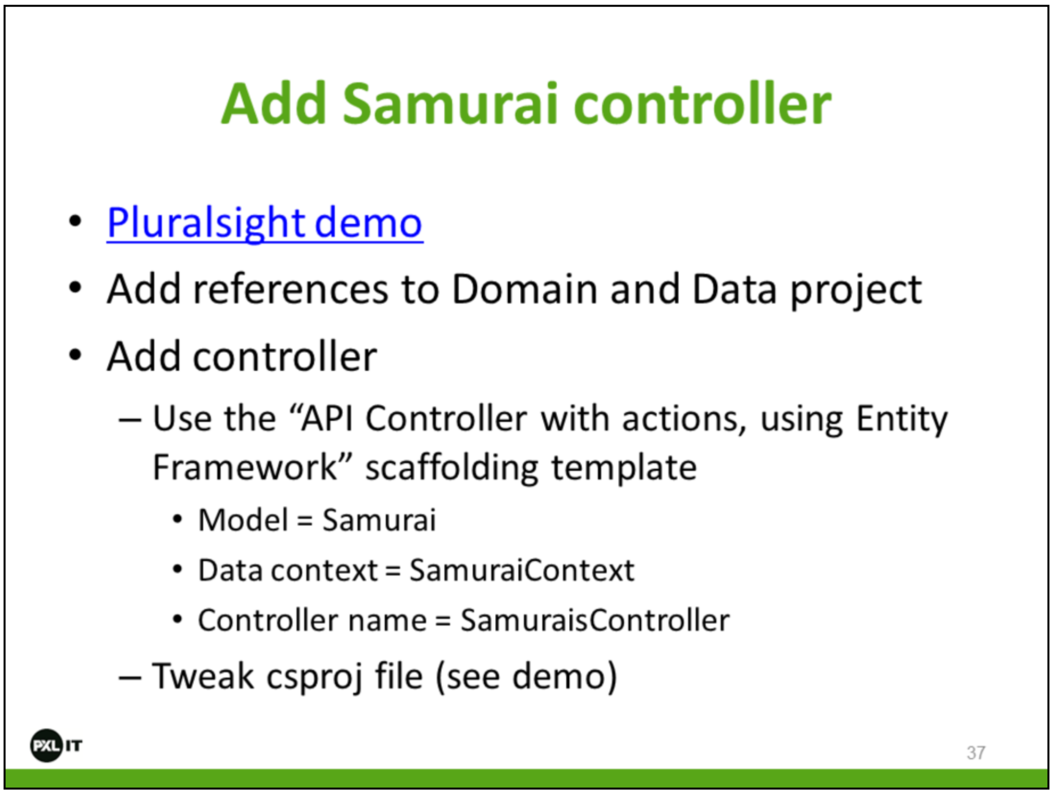
USING EF CORE WITH ASP.NET CORE

35


Create ASP.NET Core project

- Pluralsight demo
- Create new solution
  - Copy Domain and Data project folders into the solution
  - Add the copied projects as “existing project” to the solution
  - Add new project “ASP.NET Core Web Application”
    - Choose the API template

36



Add Samurai controller

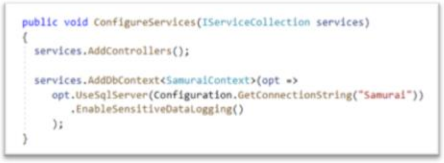
- Pluralsight demo
- Add references to Domain and Data project
- Add controller
  - Use the “API Controller with actions, using Entity Framework” scaffolding template
    - Model = Samurai
    - Data context = SamuraiContext
    - Controller name = SamuraisController
  - Tweak csproj file (see demo)

PXL IT

37

Wiring up ASP.NET Core App with our DbContext

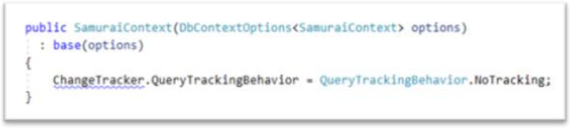
- Pluralsight demo
- Startup.cs
  - Add SamuraiContext as a service so that it can be injected in the controllers



```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllers();

  services.AddDbContext<SamuraiContext>(opt =>
     opt.UseSqlServer(Configuration.GetConnectionString("Samurai"))
        .EnableSensitiveDataLogging()
     );
}
```

- SamuraiContext
  - Add constructor that takes in DBContextOptions
    - Disable EF Tracking
    - Remove OnConfiguring method



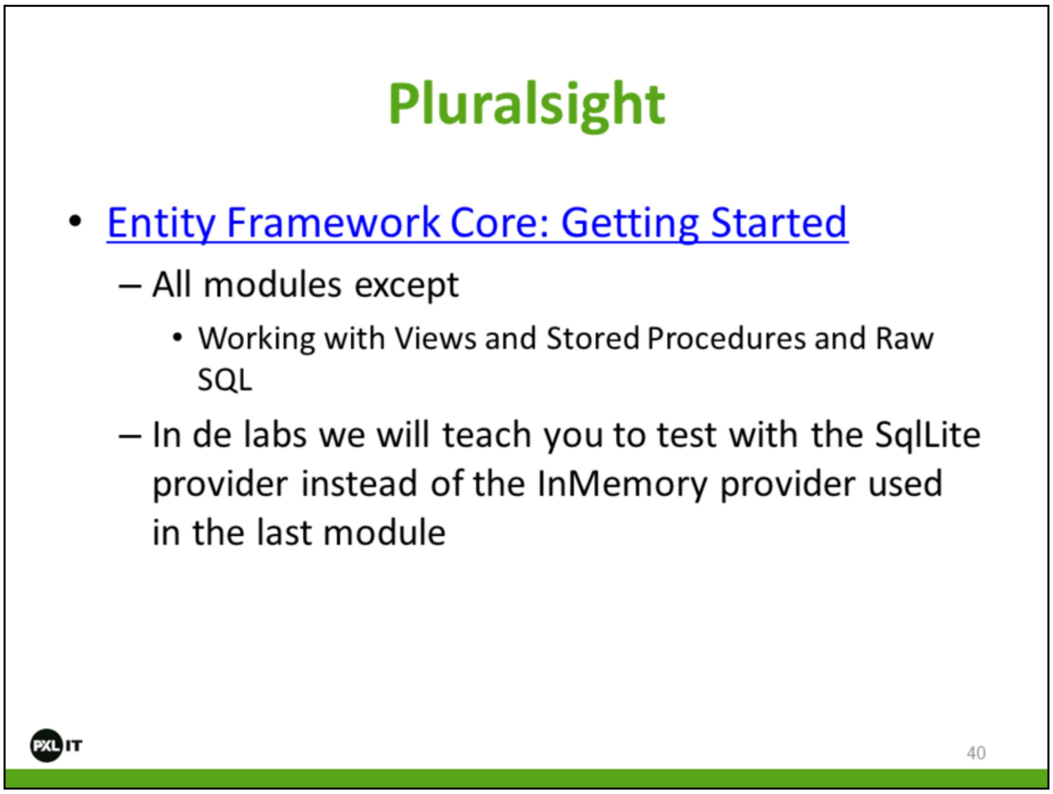
```
public SamuraiContext(DbContextOptions<SamuraiContext> options)
  : base(options)
{
    ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
}
```

PXL IT

38

REFERENCES

39



Pluralsight

- Entity Framework Core: Getting Started
  - All modules except
    - Working with Views and Stored Procedures and Raw SQL
  - In de labs we will teach you to test with the SqlLite provider instead of the InMemory provider used in the last module

40

Pluralsight Course: Entity Framework Core: Getting Started

Need to dig deeper?

- Recommended:
  - Read the official Microsoft documentation when you want to have a deeper understanding of a concept
  - https://docs.microsoft.com/ef/core

PXL IT

41

https://docs.microsoft.com/ef/core