Entity Framework

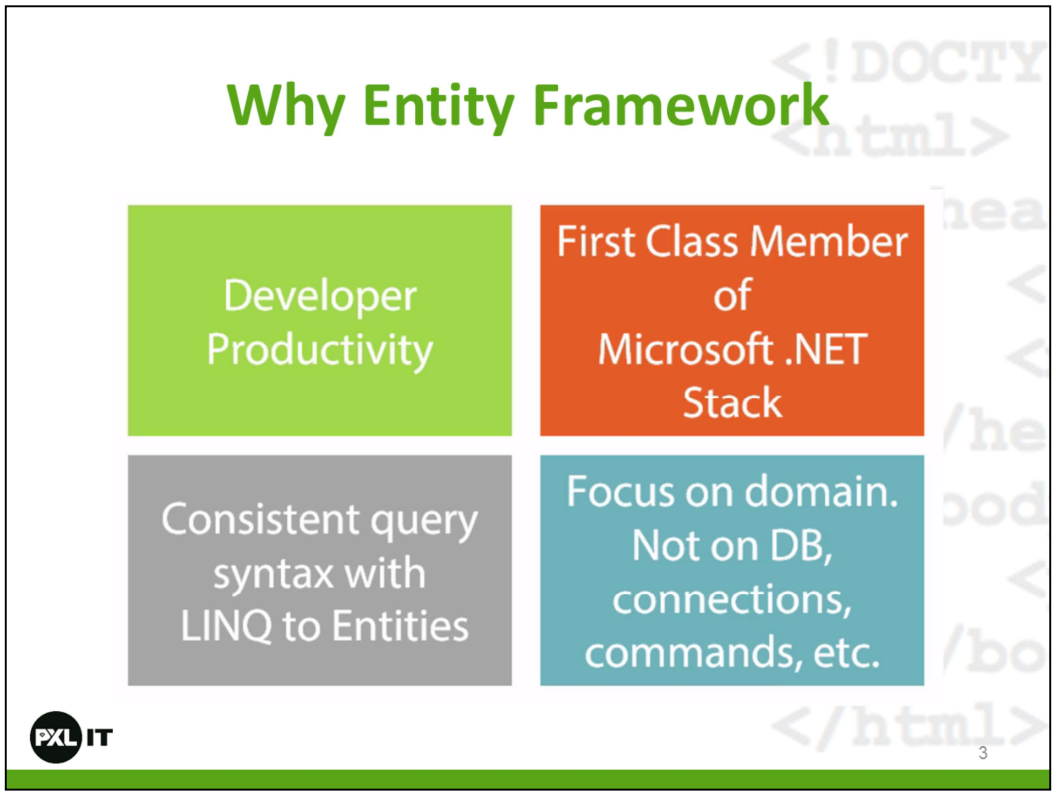
Enterprise & Mobile .Net

DE HOGESCHOOL
MET HET NETWERK

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

OVERVIEW OF EF

2



Why EF?

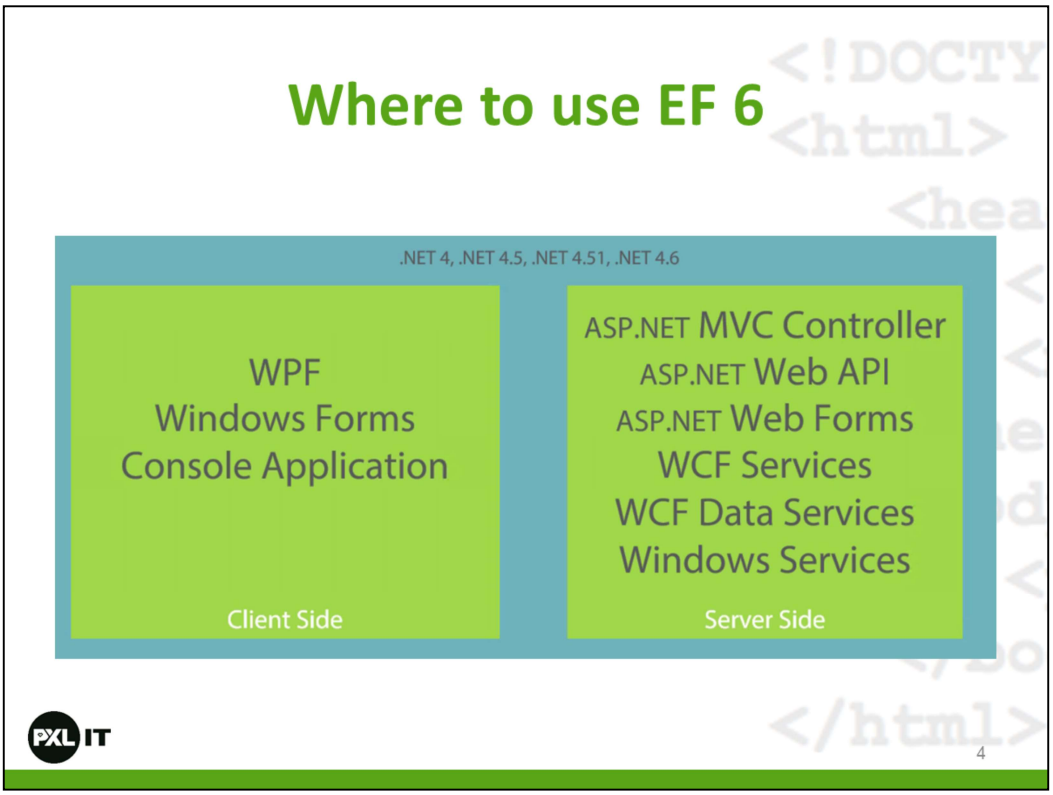
Because it eliminates a large amount of reduncant data interaction tasks, EF can enhance *developers productivity*. It also provides *consistency in the tasks* it does rather than having various members of your team invent their own means of designing their data access. There is a learning curve for EF, especially if you want to leverage it's advanced features. But in simple scenario's, the learning curve can be quick.

EF is a first class member of the Microsoft .NET Stack. EF6 is open source and had a swift release cycle. EF Core is the newest edition of EF, existing *alongside* EF6 and running on several OS (Windows, Mac and Linux). *This course uses EF6*.

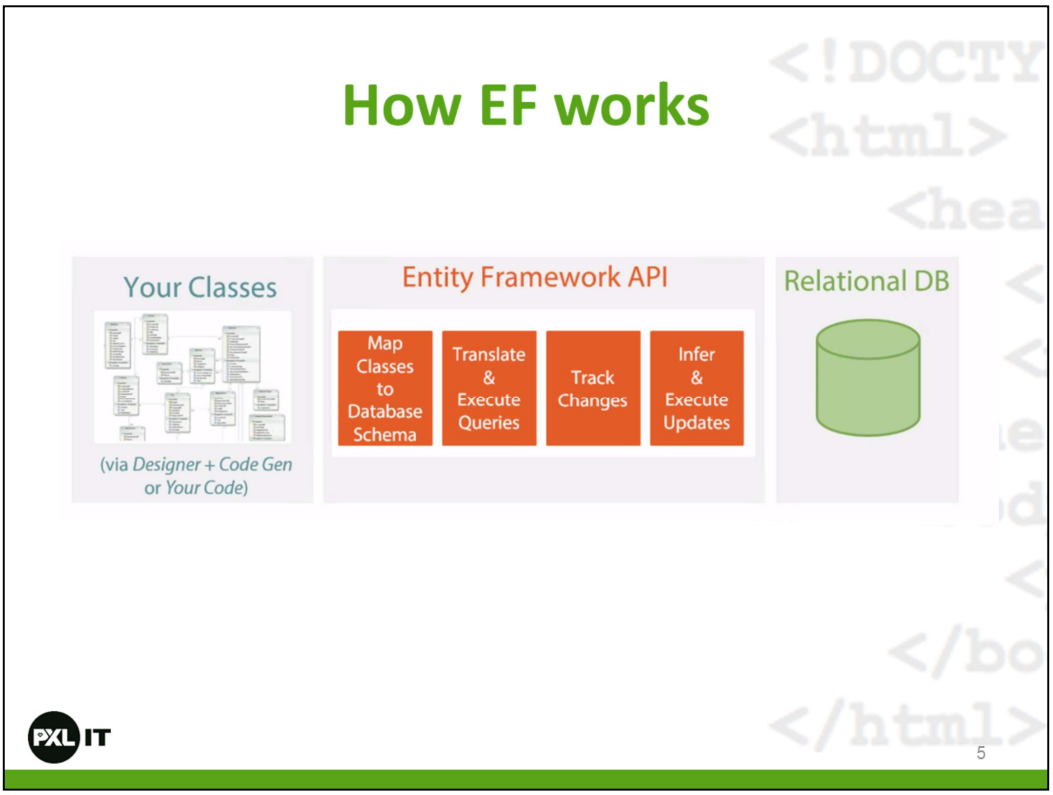
Rather than writing the relevant SQL to target whatever relational database you're working with, EF uses the LINQ syntax that's part of the .NET Framework. This allows developers to use a consistent and a strongly typed query language regardless of which database you're targetting. And they're writing these queries against their domain objects, not against the database schema. Aditionally, LINQ is used for querying other elements than .NET, even in memory objects. So developers can benefit from their knowledge of LINQ whether they're using LINQ to entities, LINQ to objects, or some other flavour

of LINQ.

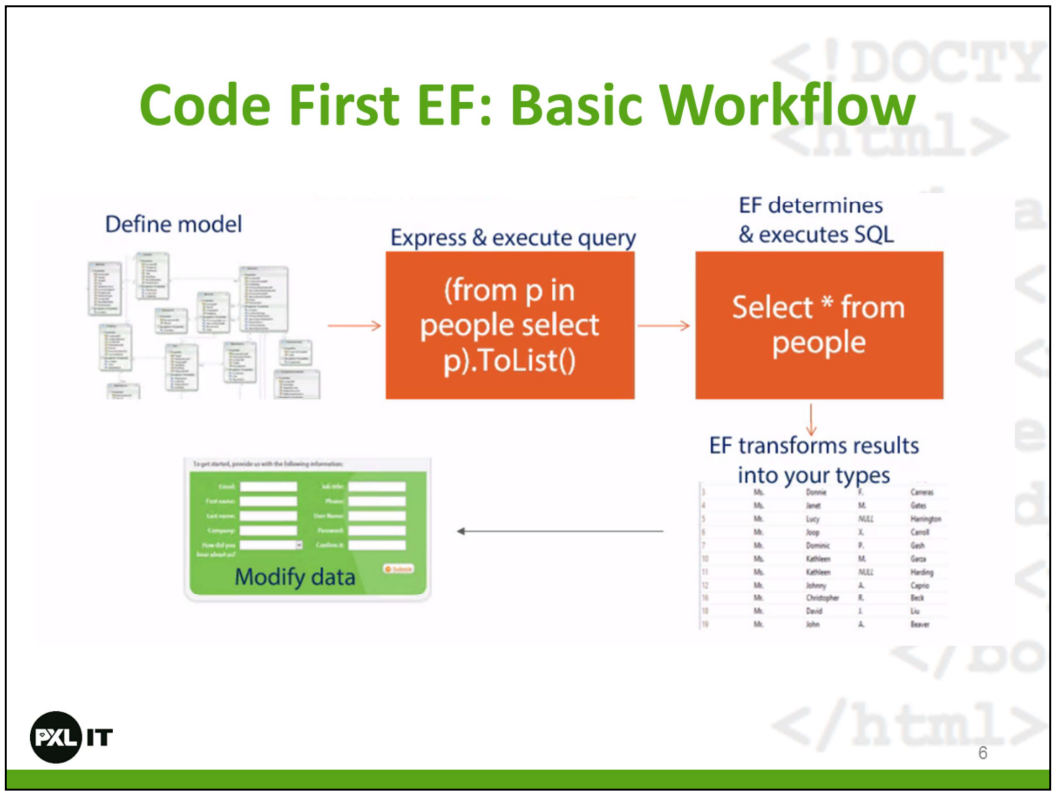
Using an ORM (=Object Relational Mapper) allows developers to focus on their domain, their business rules and their business objects. They don’t have to worry about direct interaction with the database or being familiar with the database schema. Developer still need to understand how EF works and some of it’s nuances with regards for example to how it tracks changes that need to persist to the databse. Or patterns for working in disconnected applications. But that follows the importance of understanding how your tools work. And not just blindly using them in your software.



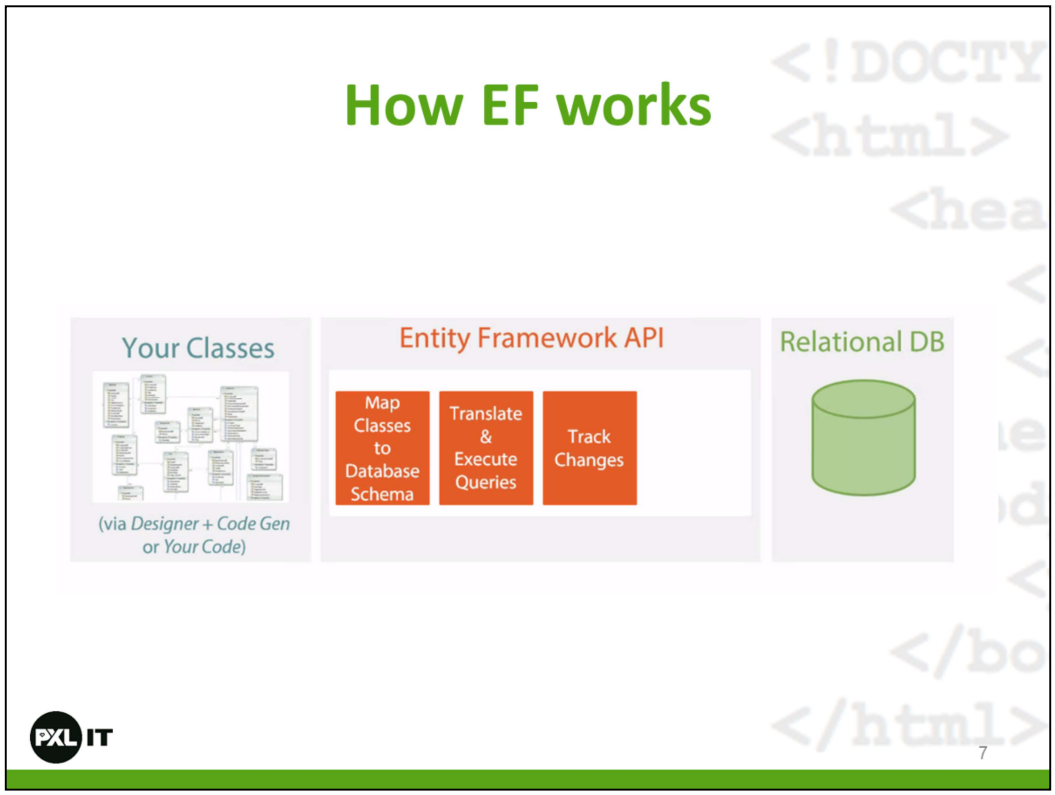
EF 6 relies on the .NET framework. You can use it with any version, starting with .NET 4. That means you can use it directly in client-side .NET applications such as WPF and server-side applications: ASP.NET and WCF. Keep in mind that with .NET services or API's you can of course support other front-ends that consume that services. And it's good to be aware that EF Core will support having EF directly on devices. So you won't be dependant on an internet connection to interact with those services or Api's. you can actually store and interact with data, directly on you device, for instance using SQLite as a database back-end.



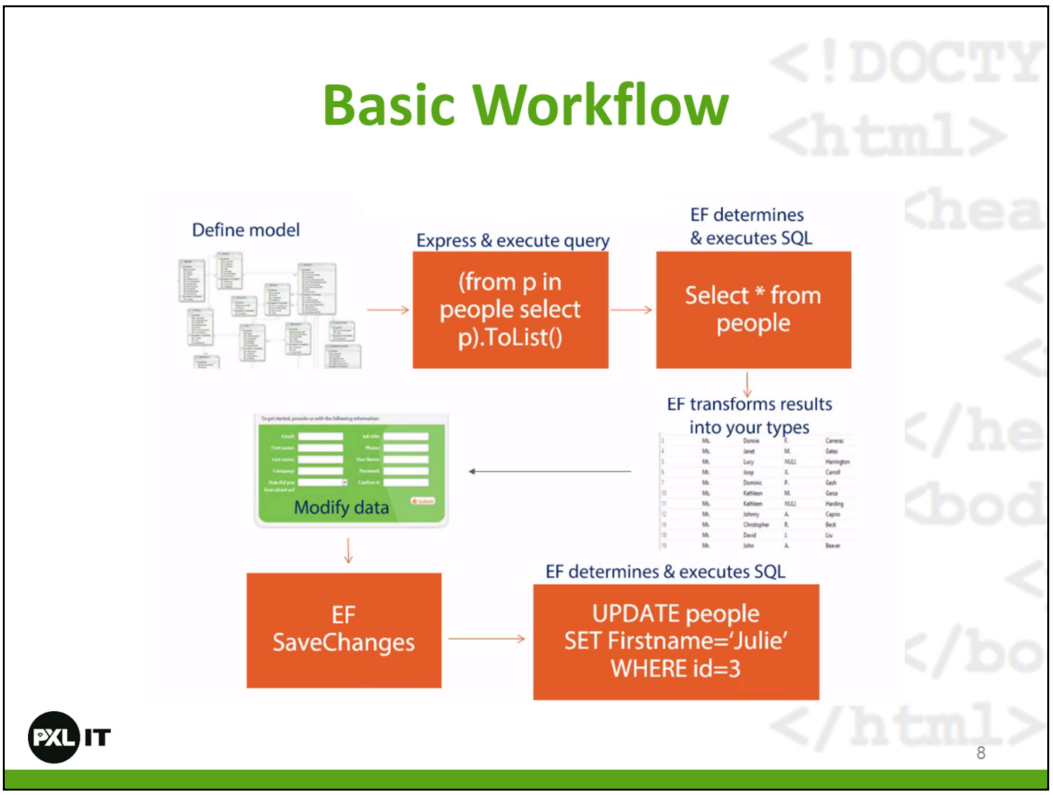
You should have a basic comprehension what EF is doing for you. It really all starts with your domain classes, not with EF. But with these classes in hand, than you can use EF Dbcontext API to wrap those classes into a model and instruct EF as to how those classes in the model map the database schema.



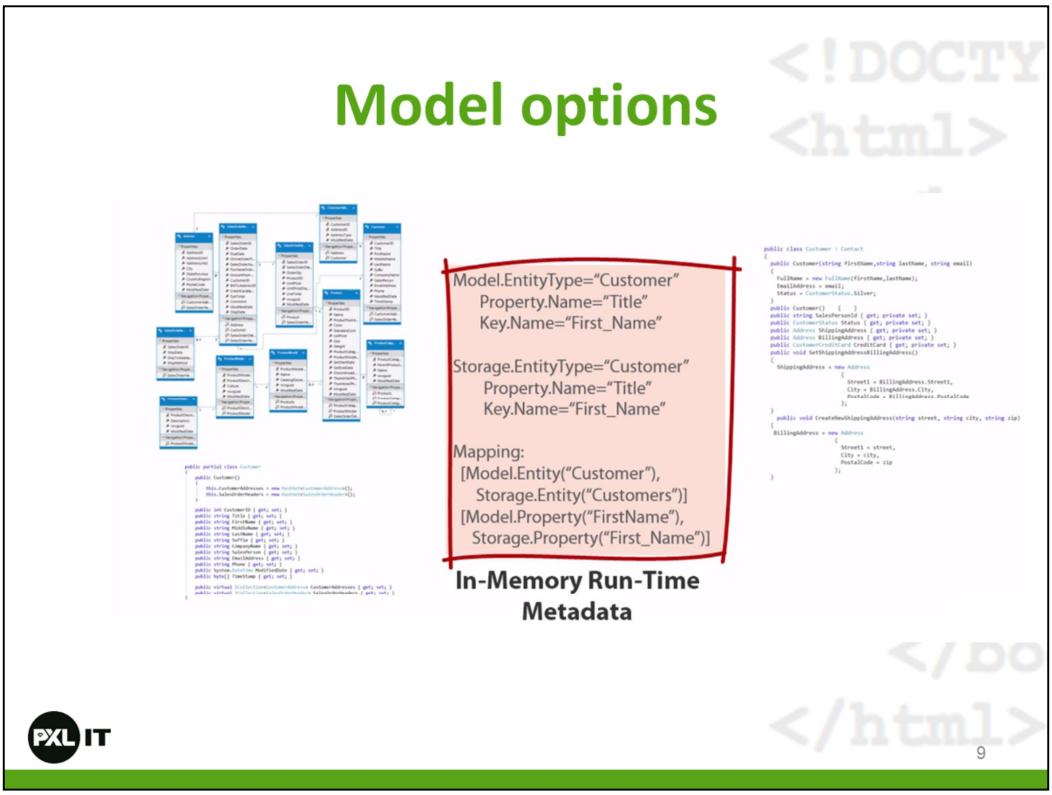
With this understanding and the help of the relevant database provider, EF can translate queries that you write in LINQ to entities against your classes into SQL that's understood by your database. It then executes the query and uses the results to return populated instances of your objects. Again, taking away all that redundant work that we normally have to do.



You can also map to views. Instead of tables, and if you need to, you can even execute Stored Procedures. Maybe if EF just can't create efficiently performing SQL, or if the queries are just to hard to express with LINQ. EF can keep track of the objects as long as they are in scope. E.g. in a client app. If you edit or add or delete an object EF will be aware of that. With disconnected apps though, we have patterns to inform EF of the state of an object, when it comes back from whatever was working with it.



Then, with a single command SaveChanges, EF will use that state information to build and execute the relevant insert, update or delete commands on the database.



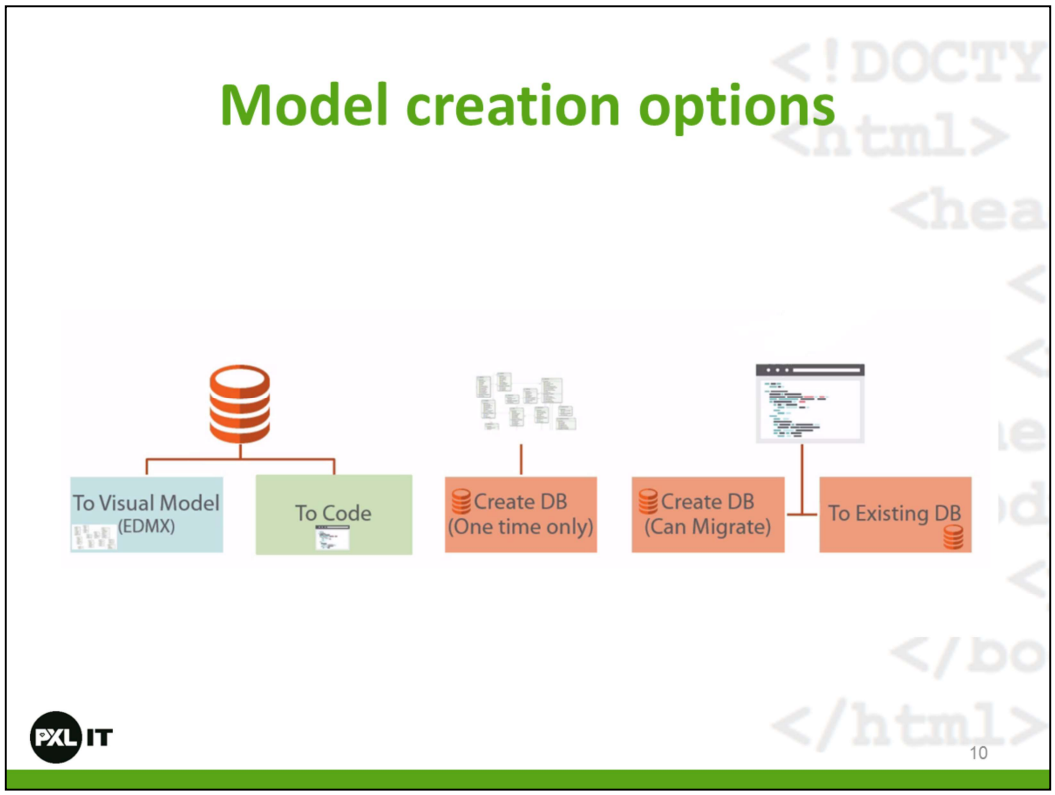
There are number of ways to define a model. There're two types of models that EF can use. One is a visual model and the other's a model that inferred from your classes and some additional code. The visual model is supported by a designer in VS as well as some 3rd party designers. If you use the designer it creates an XML file referred to as **edmx**. That's entity data model and the X for XML. Then, the designer generates classes based on that edmx.

If you use straight code then there's no code generation needed and you can totally use the code in your classes. At run time, if EF finds an EDMX, it reads its XML and generates an in-memory model which is it's representation of how the class is mapped to the database. But if EF sees that there's no EDMX, and discovers a model in your code, it will use it's **code first API** to generate that same in-memory model.

Once that in-memory model exists, regardless of where it came from, whether there was an EDMX or from code, everything that EF does is the same. Some people misunderstand this, thinking that EF behaves differently if you use the designer. But the only difference is how it's able to create that in-memory model at runtime.

Then, it's just one set of rules and behaviours after that. So that are **the 2 different types** of models: the **EDMX** and the one **based on code**. Now, how

do you create these models?



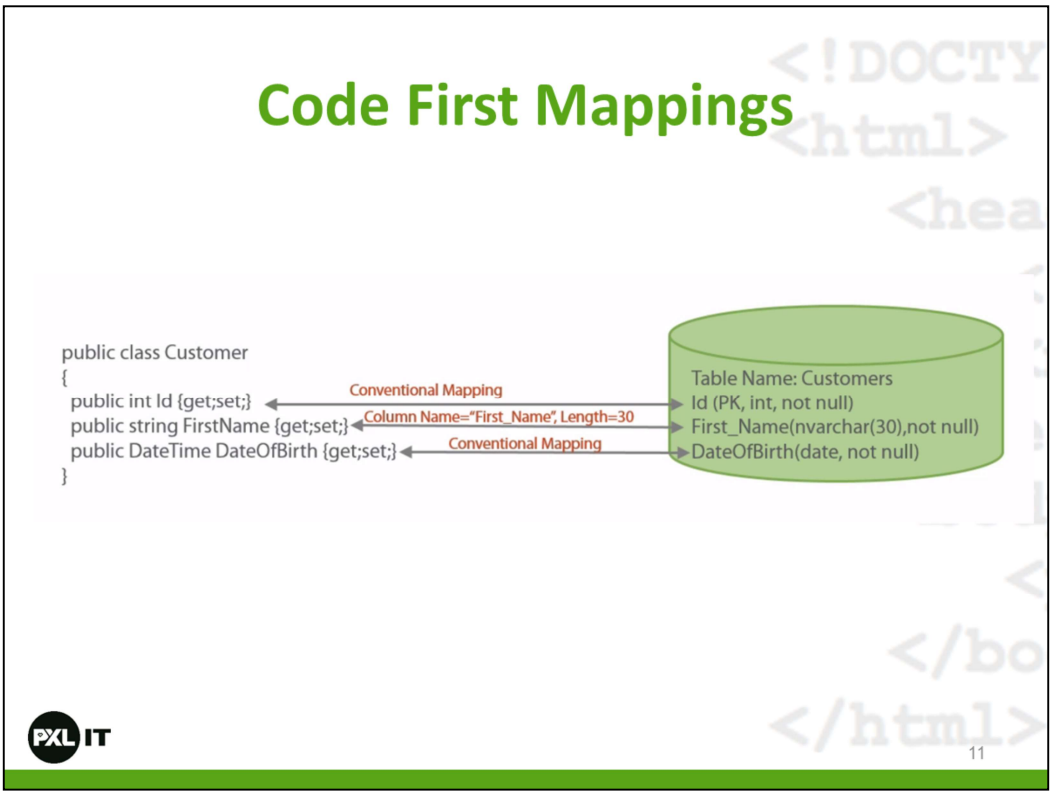
There are a few paths.

You can reverse engineer from an existing database to a model and you can do this whether you want to create a designer based EDMX or a model based on code. If you use an EDMX, it's possible to update that EDMX when you modify the database schema. But if you're reverse engineering to code, that's a one shot deal, just to get you started with the code. You can't update the code based on changes to the database.

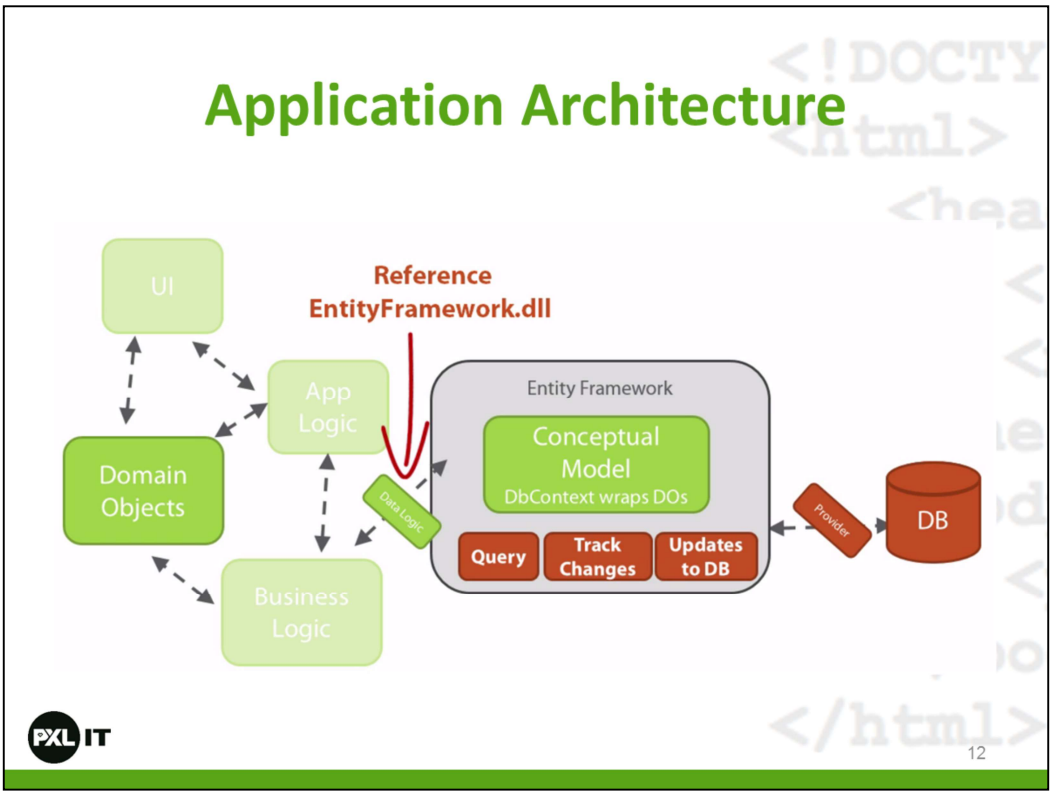
It's also possible to start in the designer. And design the model and its entities then from the generated database. This too is a one shot deal. And the feature is added quiet a while ago and doesn't have the abbility to migrate the database when you change the model. This is called **model first** and because of that limitation it does have limited uses.

The **most popular** path is to **just use code**. This is typically done if you don't have a database. Keep in mind that if you take advantage of that reverse engineer feature, to reverse engineer from an existing database to code, that's basically nothing more than a shortcut to this path starting with code. If you do start with code you also have the option to let EF determine the model and what the database schema should be and from there, create the database or create SQL to create the database. This workflow also has the ability to let you

migrate the database when your model classes or mappings change.



When you use the code based modelling and EF uses its conventions to infer what the database looks like. But as mentioned a few times already, you can apply your own mappings to override those conventions. Some examples of the default conventions are that it assumes that column names are the same as the property names. Another is, if you have a string in your class, EF relies on a database provider to determine what the default attributes of the mapped column should be. For example, Microsoft's SQL Server provider defaults to an nvarchar(4000). Conventions are not just about properties to column mappings but also they control things like indexes, relationships, constraints, and more.



A typical demo app might have a single project with the UI and the data access code, all bundled together, This isn't a bad way to just see how EF works but it can be misleading with the respect to where EF fits into a more realistic application architecture.

Of course, every problem has different needs. So this is just one perspective.

EF provides a way to define which Domain Objects it will work with in a particular model. This is done with an EF class called a DbContext. You might have more than one DbContext to define a model that you use in different areas of your software. This data logic piece is where you would write and execute your EF logic queries and calls to SaveChanges. You might even work in here to affect how EF handles your objects. The dbcontext can then execute LINQ queries, track changes and perform saves to the database. And you trigger these actions with the code that you write in the datalayer logic of your application. The rest of this application structure that you see here is simply your own code. The UI, the domain objects with their business rules. Other layers where you have application of business logic to find. This area of your application doesn't need to be aware of EF at all. Only the data logic layer where you tell the dbcontext to retrieve data for you or to save data or again perform other persistence related tasks, needs to be aware of EF.

CREATING A CODE BASED MODEL AND DATABASE

13

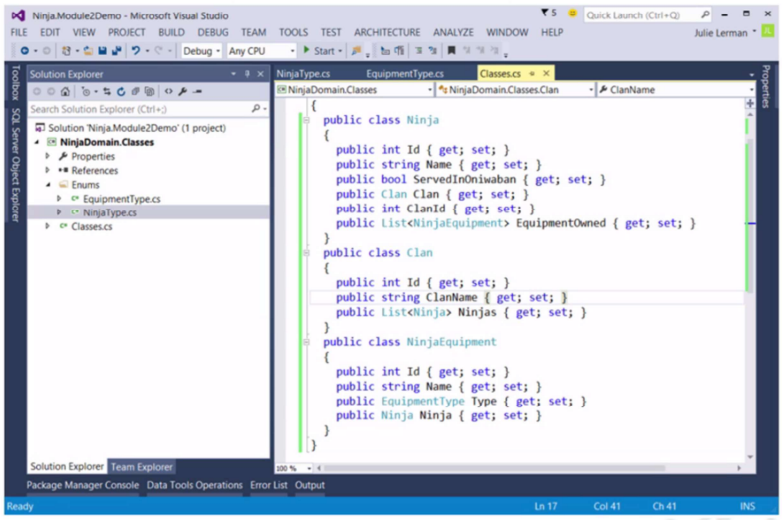
In this module

- Define a Code-Based Model from Classes
- Create a database from the model
- Modify the model
- Migrate the DB using model changes
- Code-based & visual models from a DB

PXL IT

14

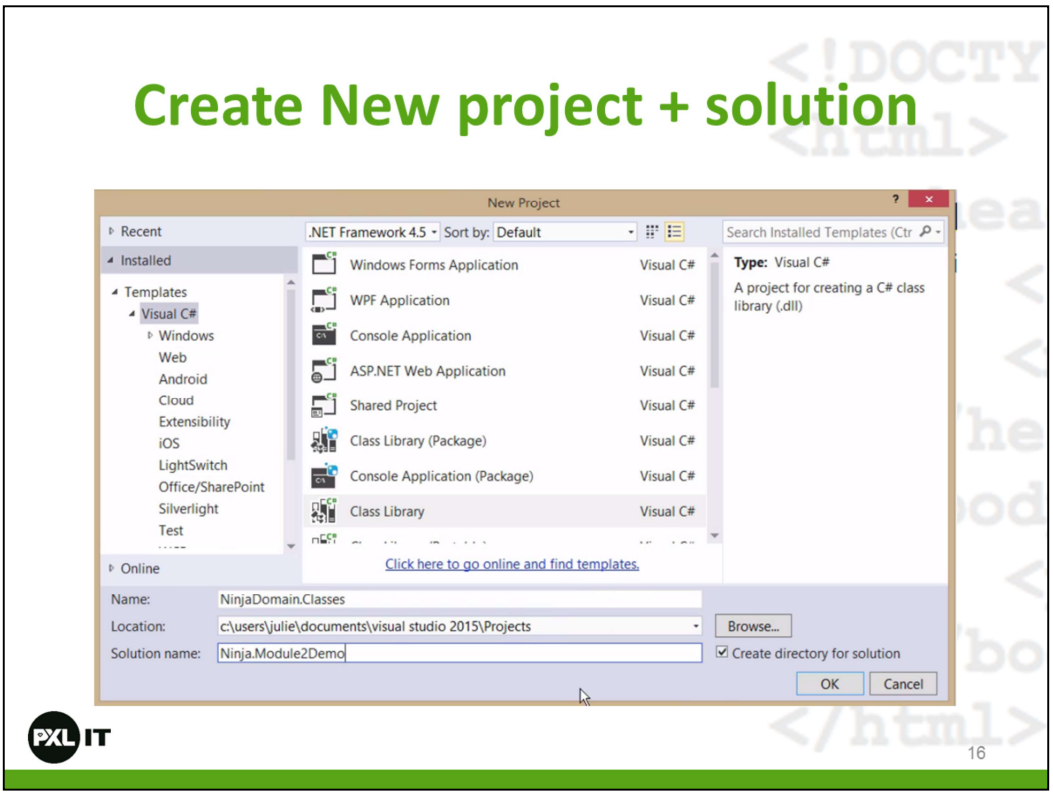
Creating the starting solution



PXL IT

15

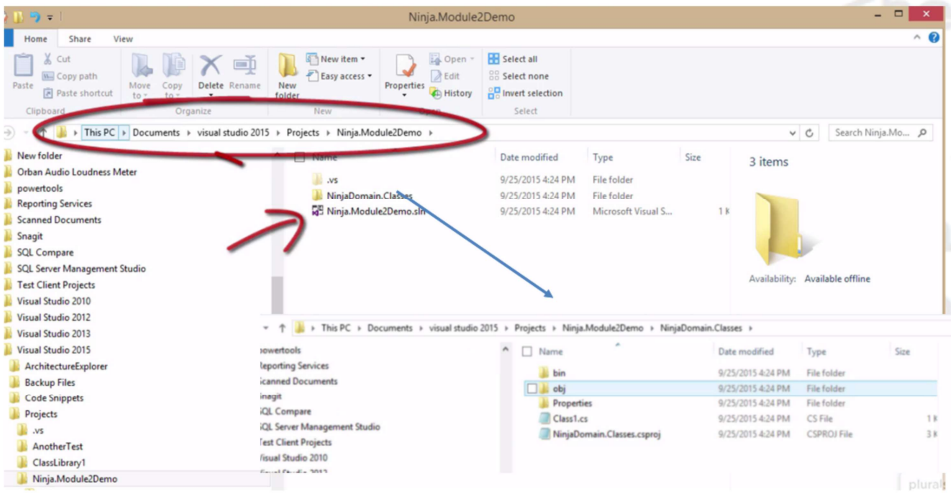
Create a solution with some Plain Old CLR Objects, aka POCO's. This part of the demo has nothing to do with EF yet.



- Create a new solution named Ninja.Module2Demo → Choose a Class Library Project.
- Name the class library: NinjaDomain.Classes

I'm starting with a small solution that has a single project, and the project type is just a class library. Within this project, there are some classes I defined, that are relevant to my domain. I have a Ninja class, a Clan class so ninja's are in clans, and then a class to keep track of each Ninja's Equipment, and they don't share Equipments. So each piece of equipment is unique to the Ninja who owns that equipment. Ninjas have a name, and we've got a boolean to see if they served in Oniwaban, which is like the CIA, and that's about it. So this has nothing to do with the EF. You can see in the references nothing about EF.

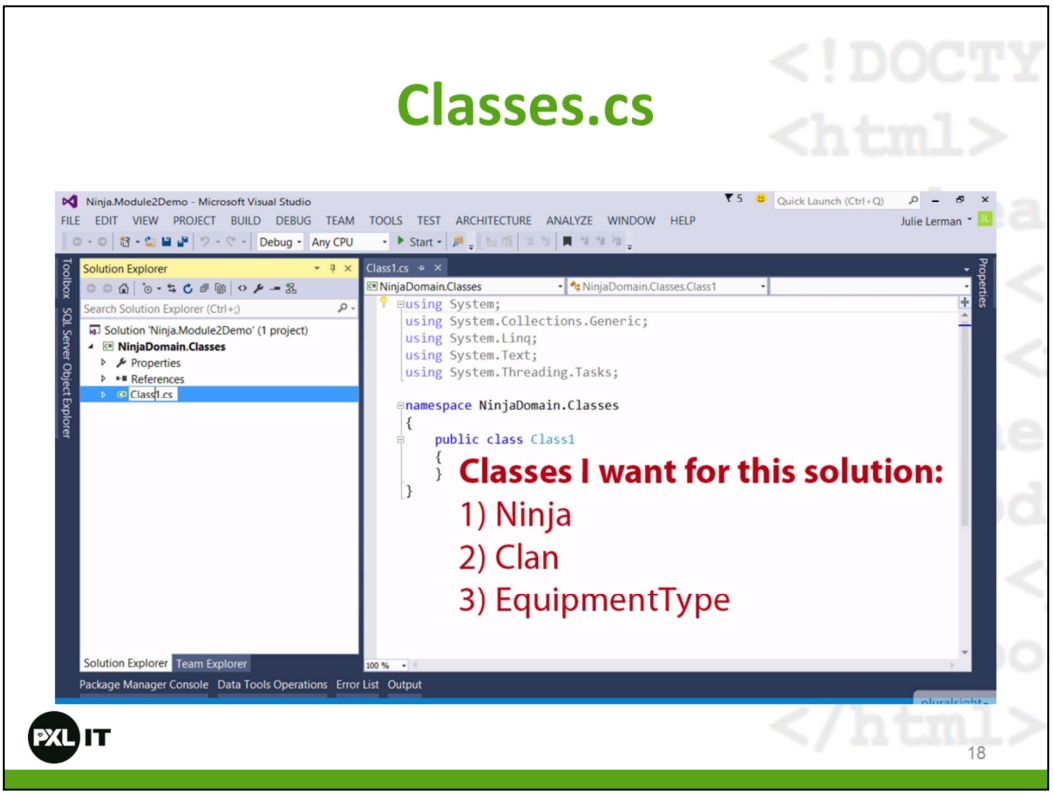
Folder



PXL IT

17

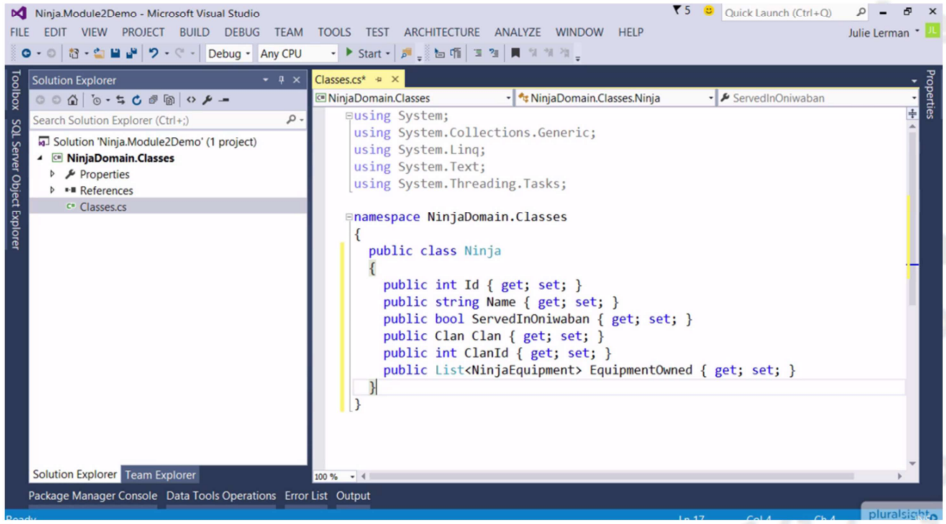
So VS created that for me . I right-click on the project and use the context menu to open the folder where all of this was created on my computer. There's a project folder and the solution file. You'll see the classes folder and inside of it the classes



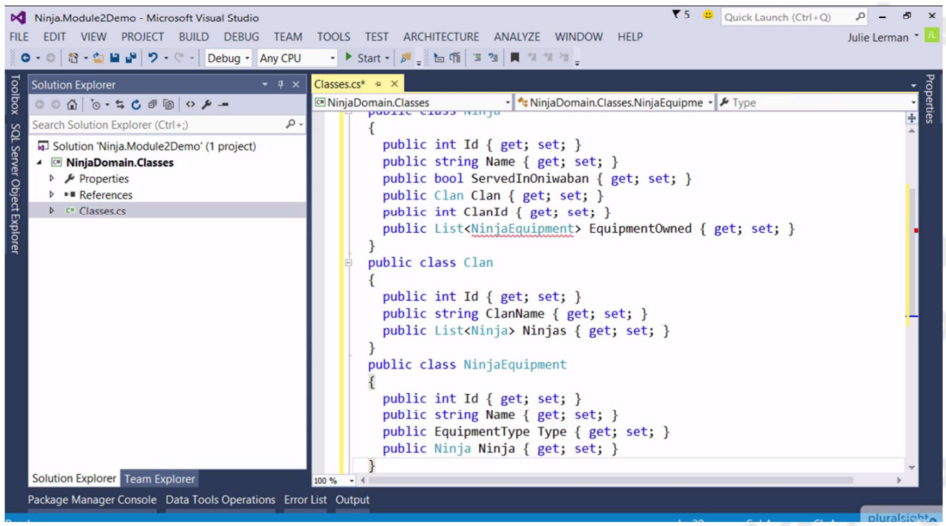
By default, the project template created a file called Class1.cs. I have 3 classes that I have to create, but I want to put them all-in a single file so that it is easier to see them. I'll change the name of Class1 to Classes.cs here in the solution explorer. Notice that is also changes the class name to Classes as well inside of the file. But I don't need a class by that name. What I want inside of this file are my 3 different classes.

NOTE: putting all classes inside one file is not best practice, but done for the sake of brevity in this demo.

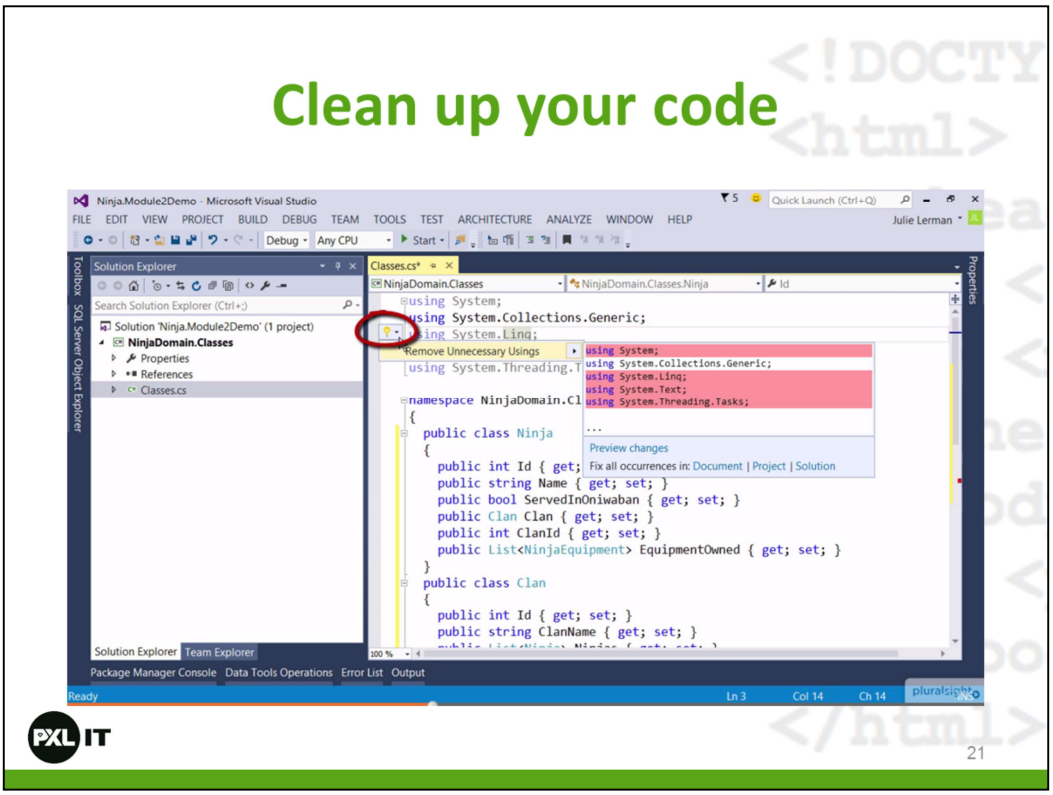
Ninja class


Ninja.Module2Demo - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Julie Lerman
Debug
Any CPU
Start
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Ninja.Module2Demo' (1 project)
NinjaDomain.Classes
Properties
References
Classes.cs
Classes.cs*
NinjaDomain.Classes
NinjaDomain.Classes.Ninja
ServedInOniwaban
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace NinjaDomain.Classes
{
public class Ninja
{
public int Id { get; set; }
public string Name { get; set; }
public bool ServedInOniwaban { get; set; }
public Clan Clan { get; set; }
public int ClanId { get; set; }
public List<NinjaEquipment> EquipmentOwned { get; set; }
}
}
Solution Explorer
Team Explorer
Package Manager Console
Data Tools Operations
Error List
Output
Toolbox
SQL Server Object Explorer
Properties

Clan and NinjaEquipment classes



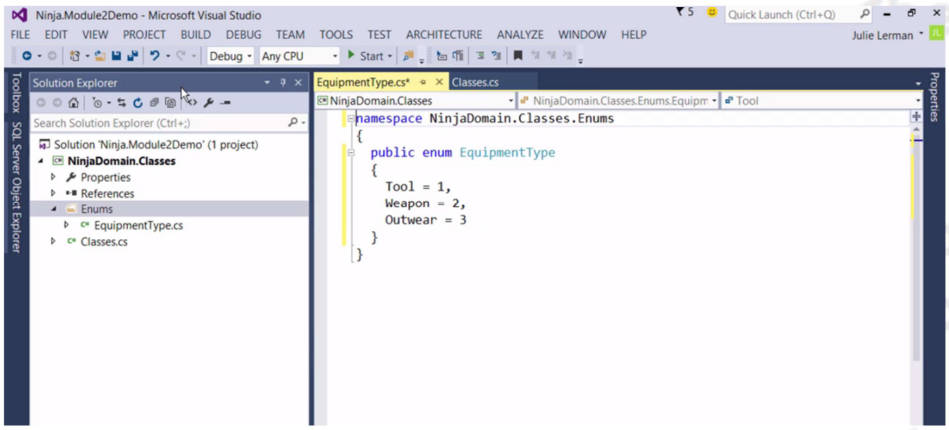
PXL IT



Something that I really like to do to clean up my code is get rid of the default using statements that I don't happen to need in the code that I'm writing. So I'll take advantage of this VS feature that provides me with this little lamp and the menu option to get rid of unnecessary using statements. That is in VS 2015.

Shortcut: put your cursor on the spot containing red squiggly lines and hit CTRL-;

Enums: EquipmentType

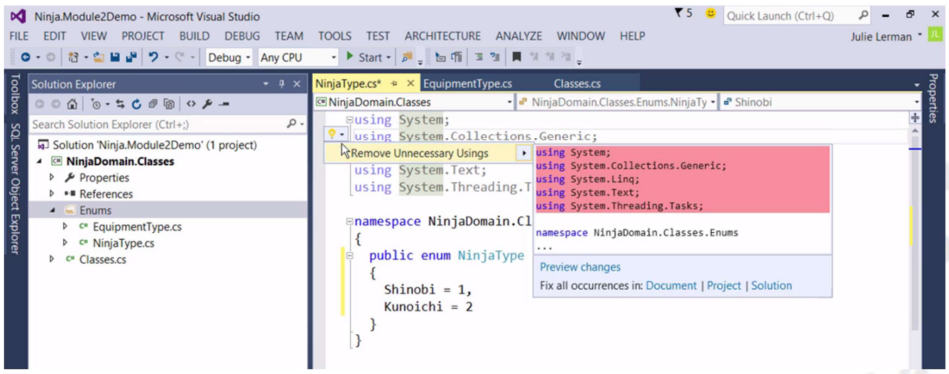


PXL IT

22

Put the enum EquipmentType in a Enums folder.

Enums: NinjaType



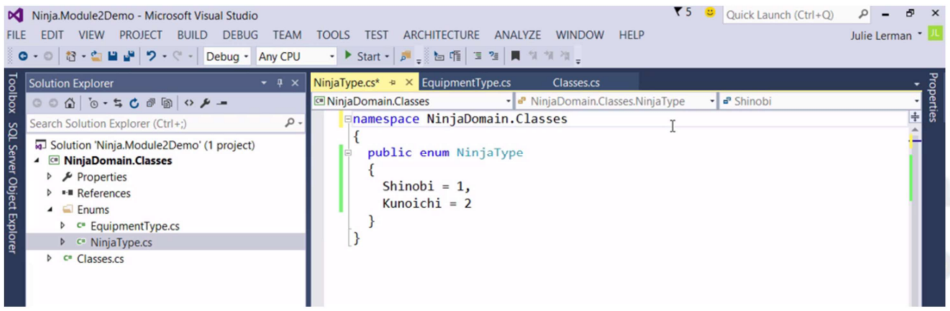
PXL IT

23

Now, I create another enum and this one is called NinjaType.

I'll make it public. By default C# types are private, having come from visual basic, so you have to explicitly mark them as public. Here the enum items for NinjaType: Shinobi and Kunoichi. And again I'll go ahead and clean up the using statements.

Enums – Classes: in the same namespace



PXL IT

24

Now going back to the classes we can see there's that EquipmentType enum, but notice it's still not being recognized. This is because the Enum is created inside another folder. VS infers a namespace that includes that foldername. So I get rid of that .Enums at the end so all of my classes and enums are in the same namespace. I'll save that and go back and look at the classes again and notice EquipmentType isn't a problem any more thus it is being found by the compiler.

The Domain Classes

- Ninja class

```
public class Ninja
{
  public int Id { get; set; }
  public string Name { get; set; }
  public bool ServedInOniwaban { get; set; }
  public Clan Clan { get; set; }
  public int ClanId { get; set; }
  public List<NinjaEquipment> EquipmentOwned { get; set; }
}
```

The Domain Classes

- Clan class

```
public class Clan
{
    public int Id { get; set; }
    public string ClanName { get; set; }
    public List<Ninja> Ninjas { get; set; }
}
```

- NinjaEquipment class

```
public class NinjaEquipment
{
    public int Id { get; set; }
    public string Name { get; set; }
    public EquipmentType Type { get; set; }
    public Ninja Ninja { get; set; }
}
```

PXL IT

The enums

- EquipmentType enum

```
namespace NinjaDomain.Classes
{
    public enum EquipmentType
    {
        Tool=1,
        Weapon=2,
        Outwear=3
    }
}
```

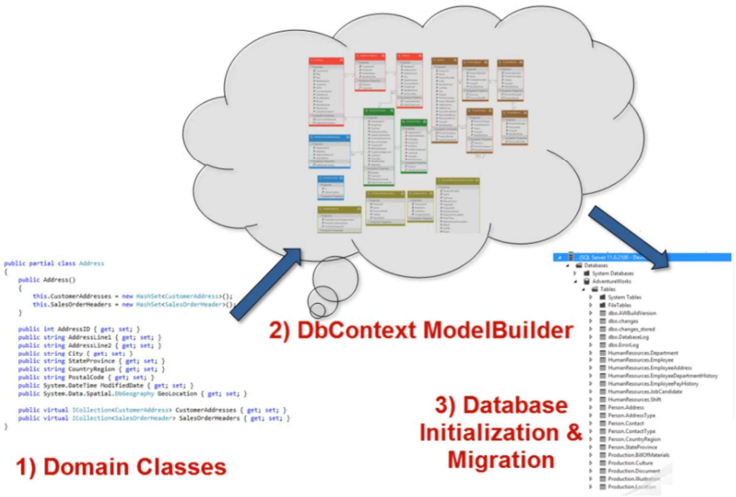
- NinjaType enum

```
namespace NinjaDomain.Classes
{
    public enum NinjaType
    {
        Shinobi = 1,
        Kunoichi = 2
    }
}
```

CREATING an EF MODEL for a Set of Domain Classes

Define Model

- Start from classes (code)



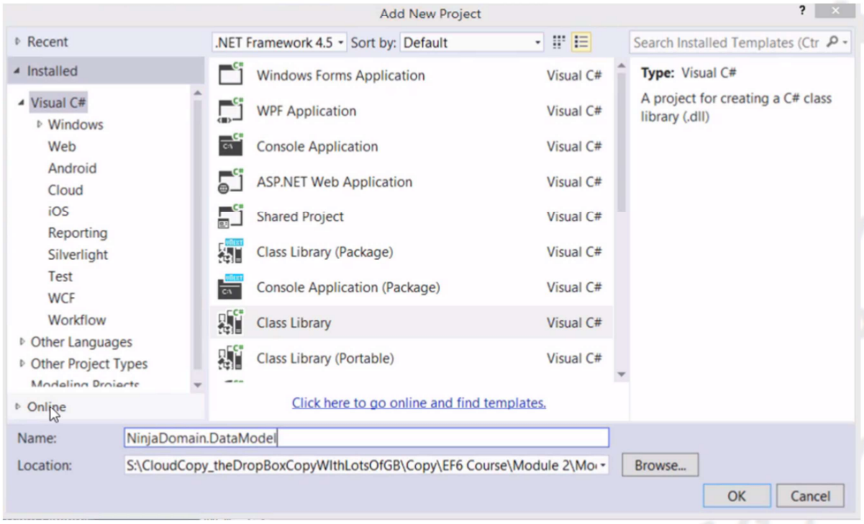
Code First

1. Create your domain classes
2. Create a DbContext

2b. Build, Debug and Fix your Code First Model with Configurations

3. Database Initialization & Migrations

Add a Class library project NinjaDomain.DataModel

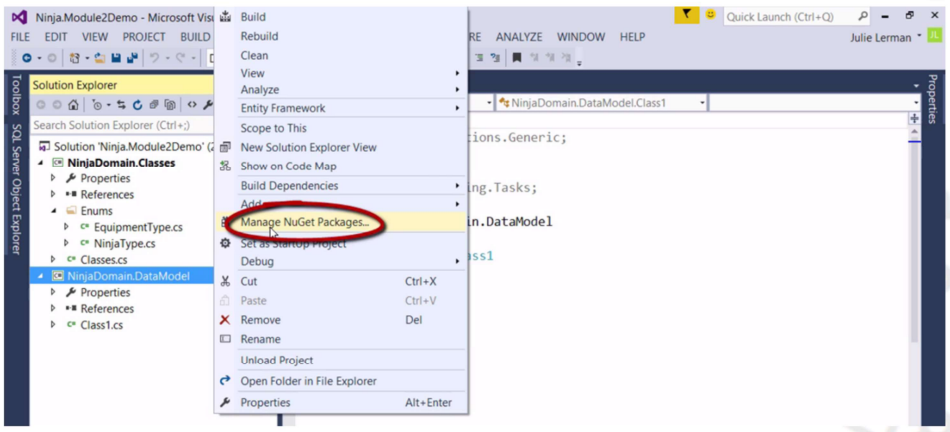


PXL IT

31

I create a second class library project within the same solution. I name it NinjaDomain.DataModel

Open Nuget Package Manager



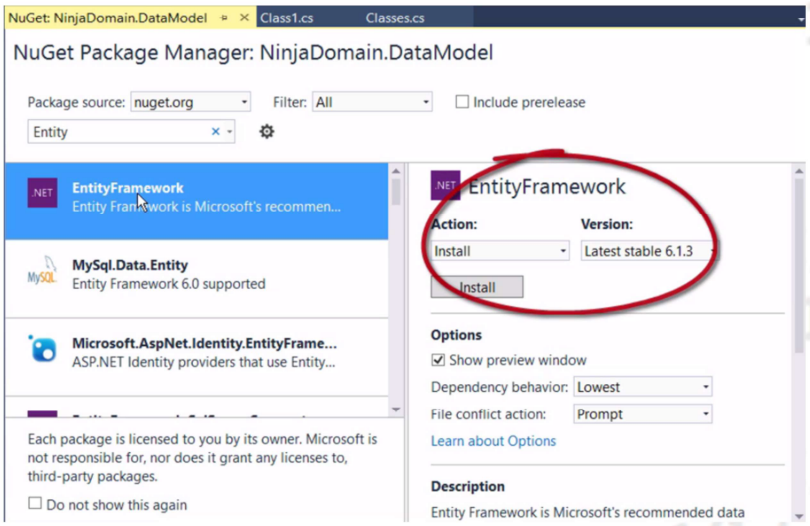
PXL IT

32

What I'll do now, is add another project which is the project that will enable my interaction with the database and other features that EF is able to perform for me. So I'll add another project and this one again will be a class library and the first thing I want to do with this datamodel is add in the EF API's. I can do this with Nugets Package manager. I do that by right-clicking the project and choose Manage Nuget Packages.

I do the same thing for the first project.

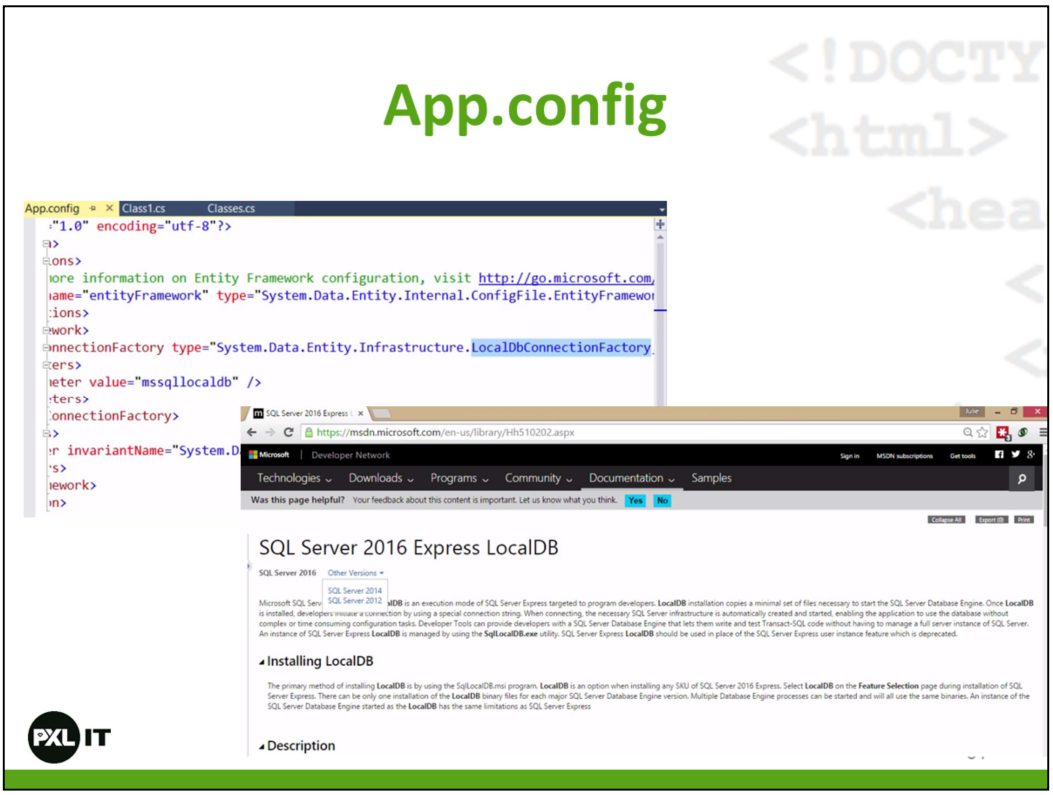
Adding EF to the project



PXL IT

33

I'll filter on EF to get to it more quickly, and there it is with the latest version. I select it and I can go ahead and install. You can see in the output window that the package manager is working and is downloading and installing the packages.. In addition to the packages having been installed, it made references to EF and entityframework.sqlserver.

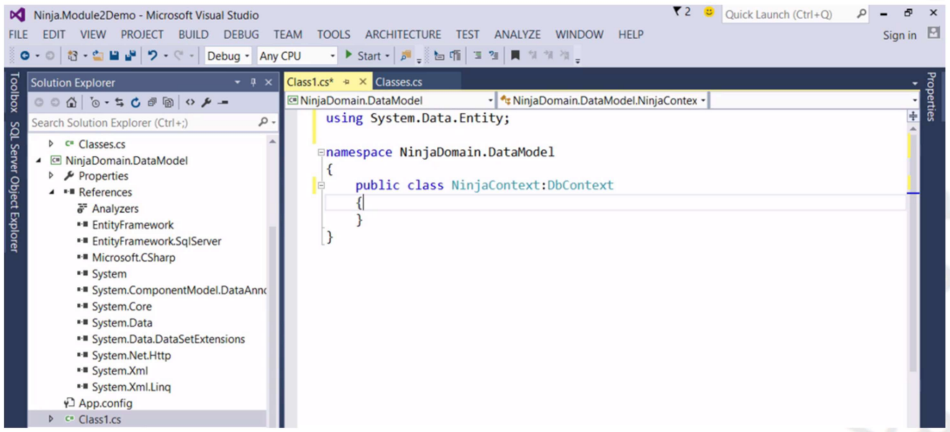


When the package manager installed EF it saw that there was no app.config file. In this project, so it created one for me because it needs to put inside some default information that EF will use. By default it will use SQL Server localdb as a database. LocalDb is a Sql server that is essentially a lightweight version of Sql Express to be used for development. LocalDb was first introduced with SQL server 2012 and it's included with the VS 2013 and 2015 installations.

You can read more about localdb for various versions of sql server in the msdn library:

https://msdn.microsoft.com/en-us/library/hh510202.aspx

NinjaContext class

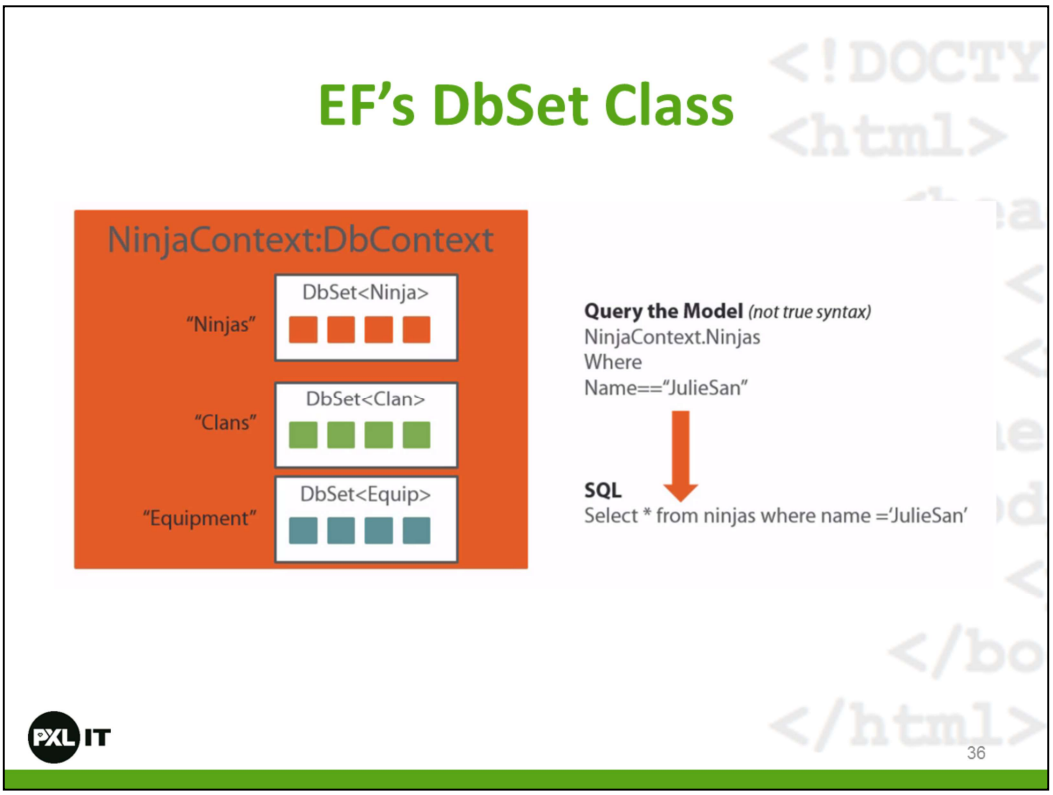


PXL IT

35

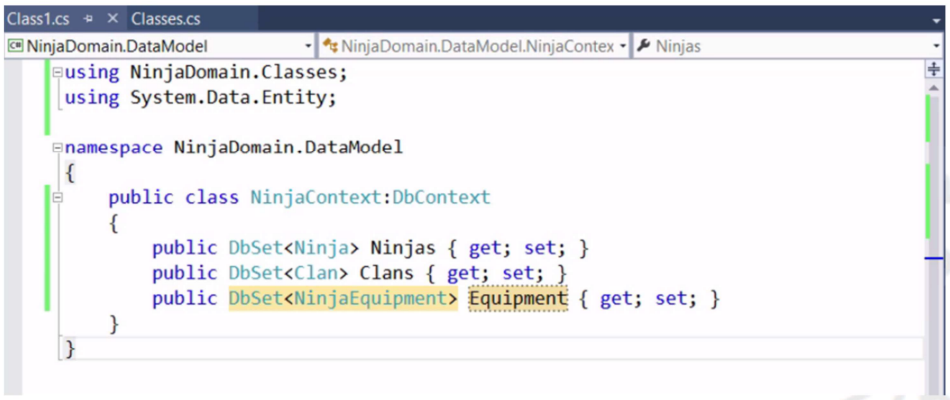
The class that was created, I just gonna change its name to NinjaContext that needs to inherit from the Entity Frameworks DbContext, which is in System.Data.Entity namespace.

The context is what really manages everything, but this dbcontext still doesn't know anyting about my other classes. My Ninja class, Equipment, Clans. The Dbcontext class in this API orchestrates all of the interactions in the database and the changetracking.



Another critical class in the API is the DbSet. The DbSet is the organizational unit that maintains a particular set of types. If you're familiar with the repository pattern, a DbSet is a repository. It's responsible for maintaining the in-memory collection of entities which could be a collection of ninjas, of a collection of clans, or a collection of any of your entities in your model. You perform your queries through DbSets., asking the DbContext to query by way of the DbSet of Ninjas or the DbSet of Clans. And in term, EF interprets that into a database query. So when we define a DbContext, we need to tell it what DbSets are in this model.

NinjaContext.cs



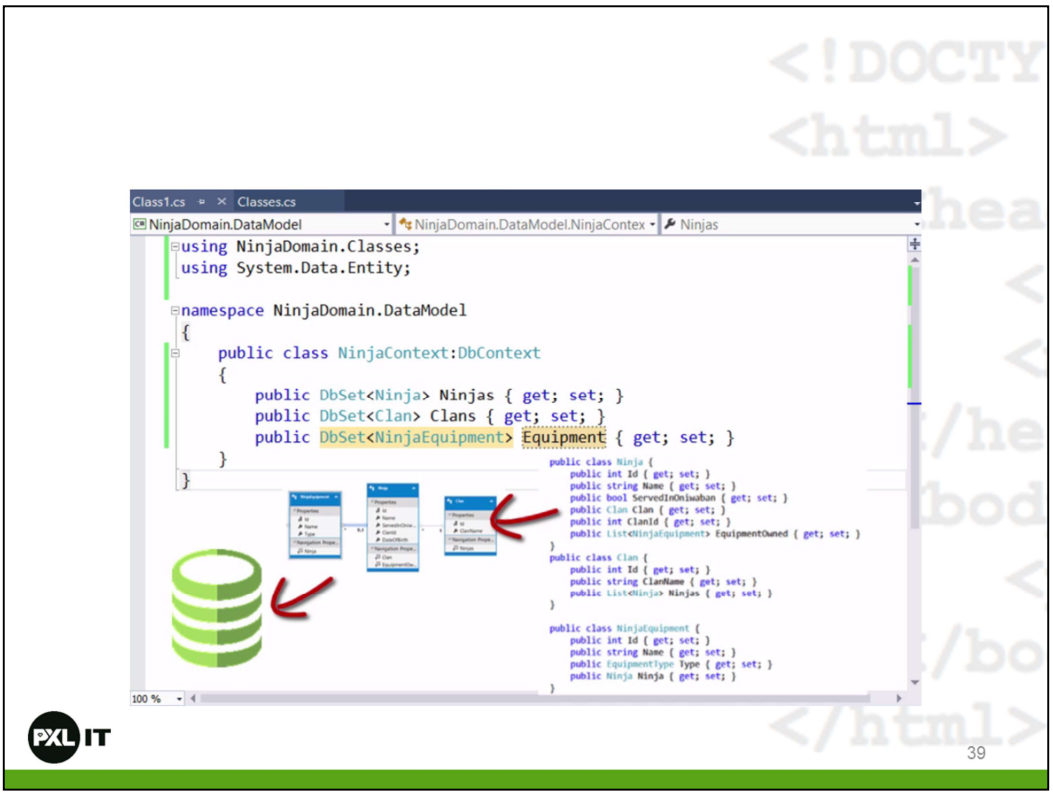
PXL IT

37

So I'll tell the DbContext that it should have a DbSet of Ninjas, one for NinjaEquipment and one for Clans. That means that I will be able to query and persist these entities directly. It's also possible to have entities in your model that aren't part of the DbSet. Those are reached by relationships to entities that are in a DbSet. So I have to create DbSets for all the types. I have 2 choices here now, One is to cross my fingers and hope that EF interprets everything the way I mented to be or I can take an advanced look at how it's going to interpret all of that and that's what we will do next.

VALIDATING YOUR EF model

PXL IT



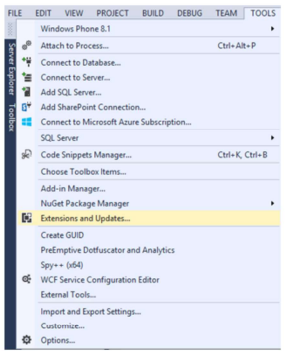
At run-time when EF sees this DbContext class, it will think,, I've got a set of Ninjas , a set of Clans, a set of the Equipment, It will go back to the classes and look at the schema of those classes, the relationships between them and infer a model as well as infer how that model relates to a database schema.

2b. Building, Debugging and Fixing your Code First Model with Configurations

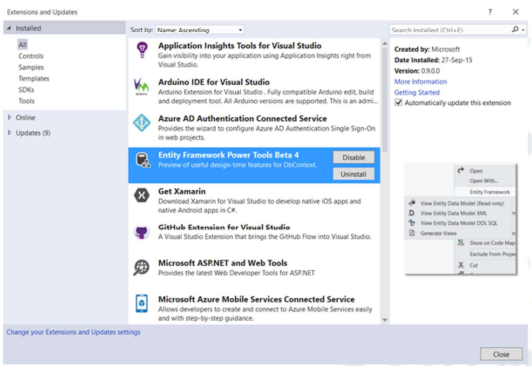
- We don't have a Designer or an EDMX file when working with Code First (←→ Database First, Model First)
- How does EF generate an entity data model from this context ?
- → On the fly at runtime ➔ can use a tool that will visualize the model for you

EF Power Tools

- Tools → Extensions and updates



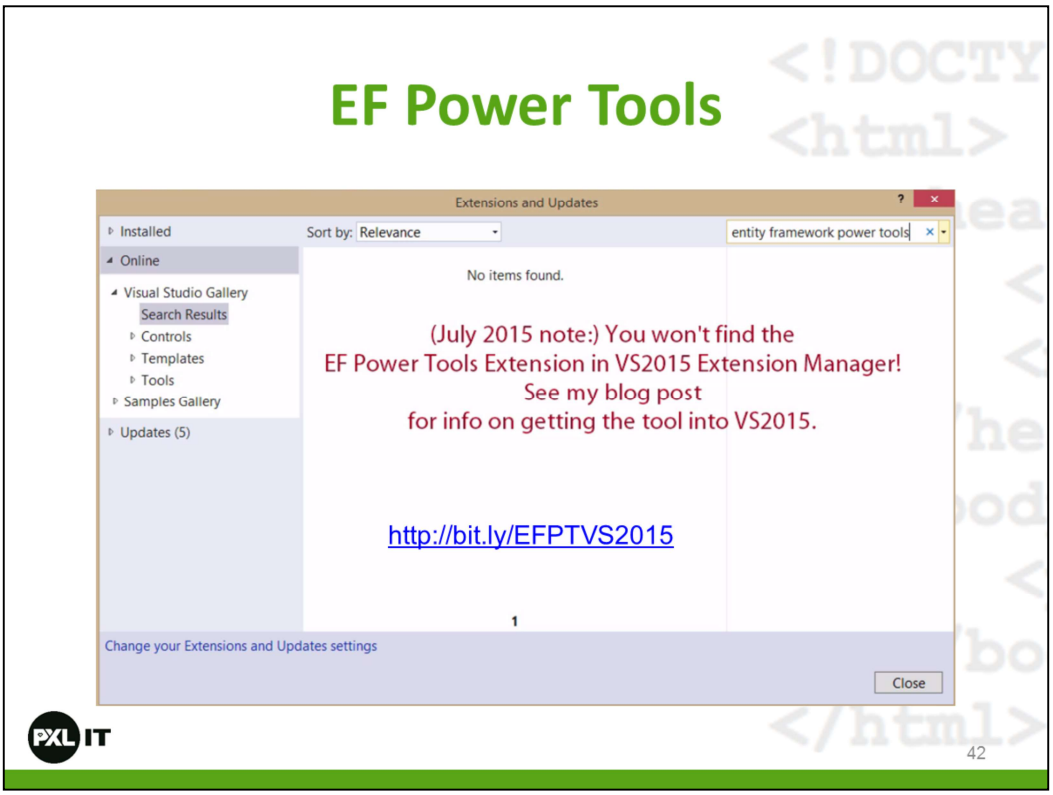
- Entity Framework Power Tools



PXL IT

41

In order to validate your model, you can use a tool called Entity Framework Power Tools which is created by the Microsoft EF team and it's still considered as beta. Eventually, they're pulling more and more of its features into the EF designer.

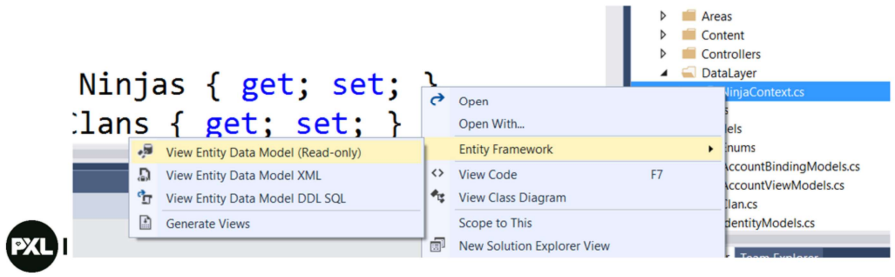


Keep in mind that the installer is not available for Visual Studio 2015

Installation instructions here: http://bit.ly/EFPTVS2015
TL;DR: just download the VSIX at the top of the article and doubleclick it.

Power Tools

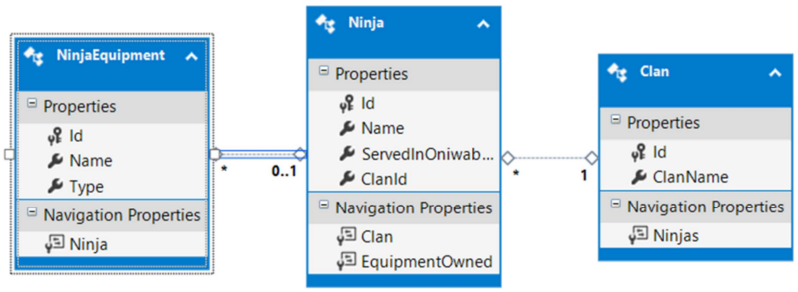
- !! In VS2015: You won't find the EF Power Tools Extension Manager !! → bit.ly/EFPTVS2015
- RightClick Domain Classes in VS



43

So Entity Framework will go to the same process as it would at runtime to infer the model from the DbContext but I do need to give it some help. That tool needs to find the app.config file, so I do have to make the DataModel Project, the startup project. When you right-click on the NinjaContext class in the Solution Explorer, in the submenu you can click on Entity Framework → View Entity Data Model (Read-Only) to generate the model as its infered.

View Entity Data Model

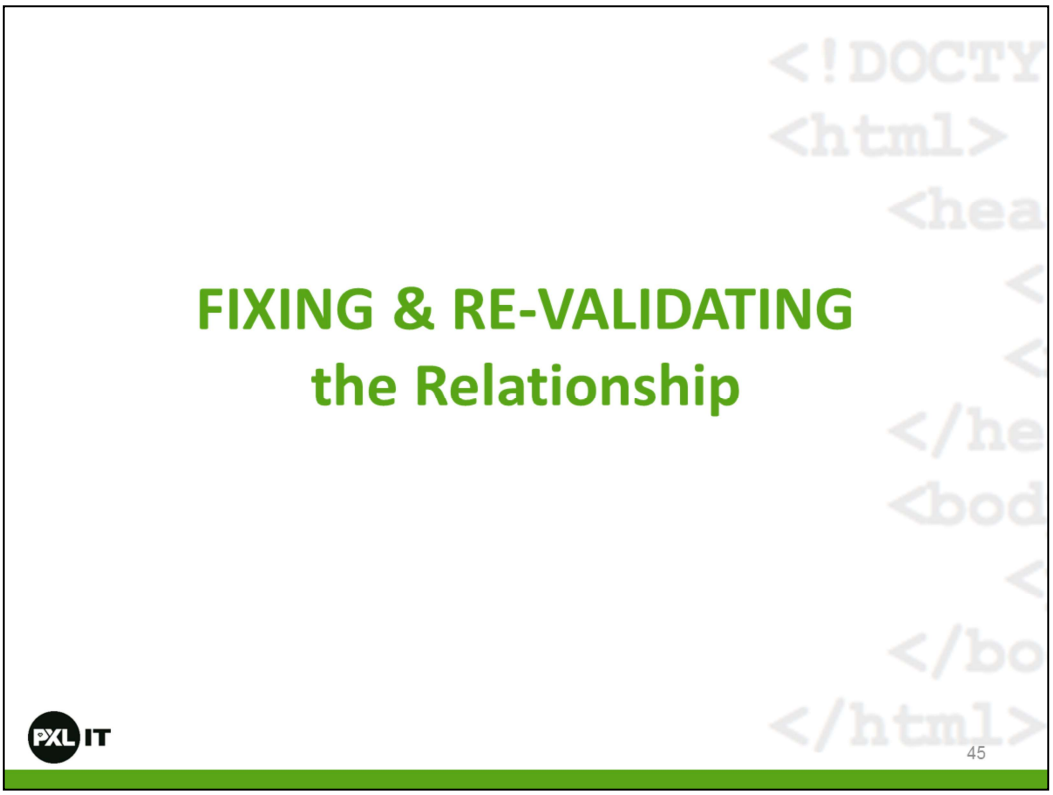


Between Ninja and NinjaEquipment: Has to be 1..* relationship instead of 0/1..* relationship

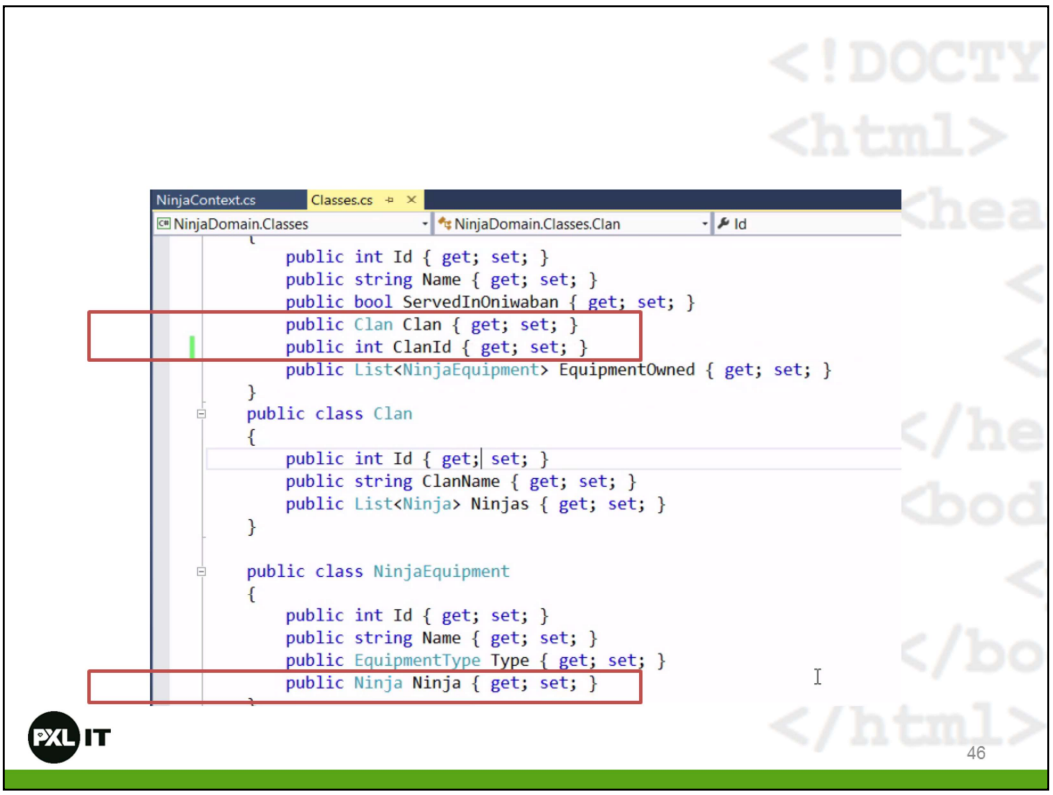
PXL IT

44

I want to have a one to many relationship between Clan and Ninja, so a Clan can have many Ninjas in it and it inferred that there is a zero or one to may relationship between Ninja and NinjaEquipment. It assumes that I can have an Equipment that isn't assigned to a Ninja. And that isn't how I really wanted it to be. So it is really important that I saw this with this tool, otherwise at runtime, things would be acting differently and being persistent differently than I had wanted. This is all happening based on the conventions. I can go in and make some changes. What I want is that there is a one to many relationship between Ninja and NinjaEquipment.

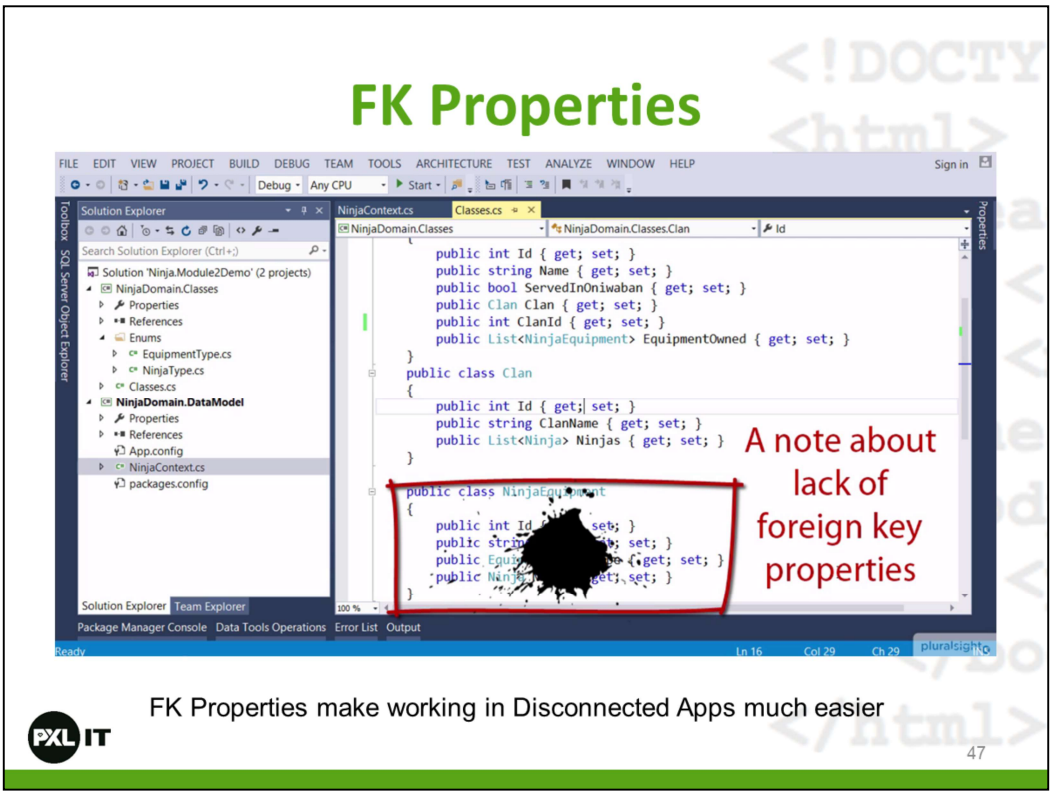


Why did EF interpret this relationship differently ?



If we look back in the classes, notice that the Ninja class I also have a FK property for the ClanID. I put that in there, because I wanted an easy way to identify the clan if I happen to have the ID without having to worry about having an instance of the clan at my disposal.

If we look at the NinjaEquipment class, we don't have a NinjaID because my intention is to identify the relationship between Ninja and their equipment through the EquipmentOwned property in the Ninja class and I don't anticipate needing to create a piece of NinjaEquipment and just set the NinjaID.



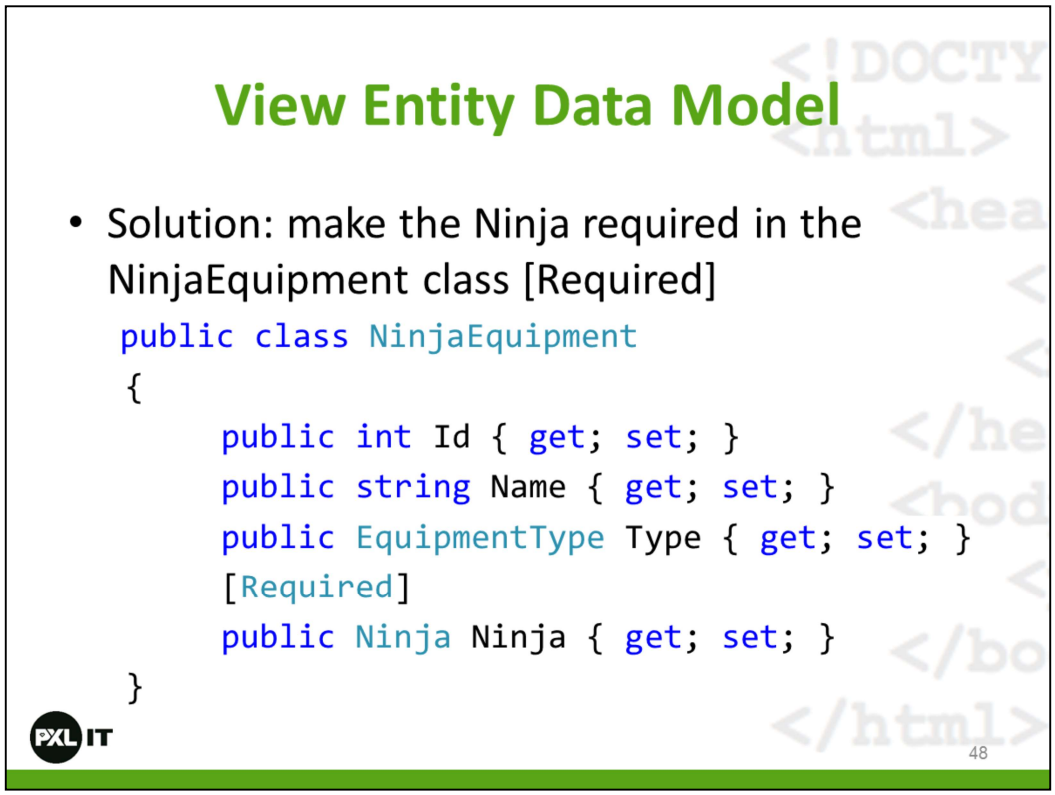
This looks nice and clean not having the FK property in there. It's actually something that I strongly discourage.

Having the FK seems unnessacery in your class and some people go as far as saying that it dirties their class. But there are places where you might pay heavily for asking EF to perform its work only based on having this navigation property. You won't notice any problem in the course as I'm working on the client-side of the applications,

But when I'm using this model in a web application where things are disconnected, the problem will be very obvious and we'll make sure you don't miss it. So for now, I leave the class just this way. Because when we hit the problem later on, it's going to be a memorable lesson.

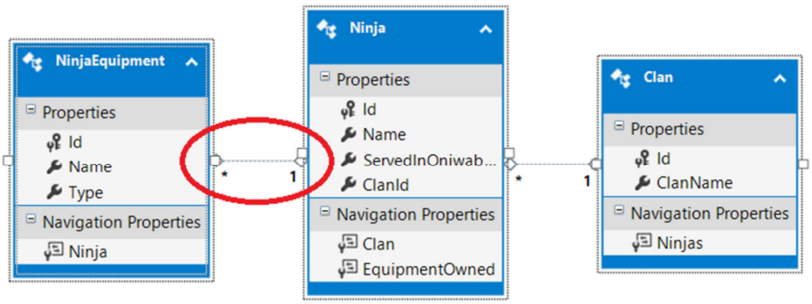
So why did EF interpreted a 0..1 relationship between Ninja and NinjaEquipment?

The fact that I have ClanID (integer) and an integer is not-nullable and therefore required, EF interprets: if ClanID can't possible be nullable, there must always have to be a Clan. Therefore it's going to be a 1 to many relationship. But I don't have that in NinjaEquipment there is nothing to say that you must have a Ninja.



Therefore there are a couple of ways to solve that. The easiest way to add the [Required] data annotation to the Ninja property. Now, the Ninja is required. EF comprehenses that data annotation.

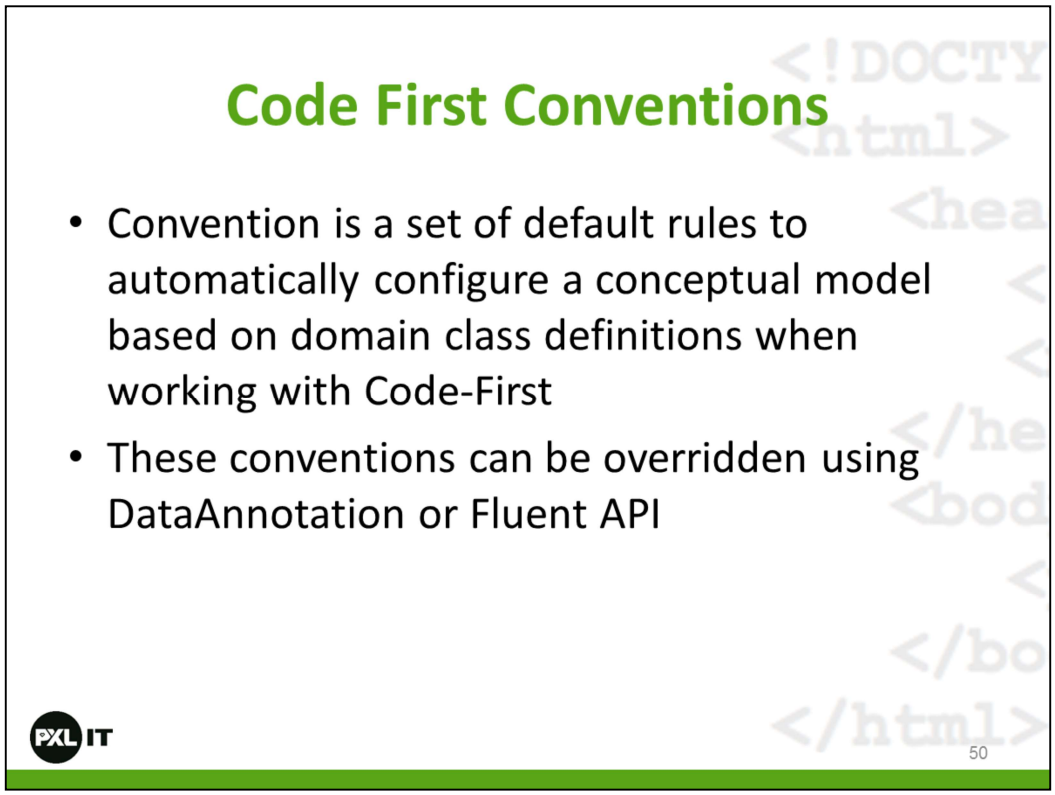
View Entity Data Model



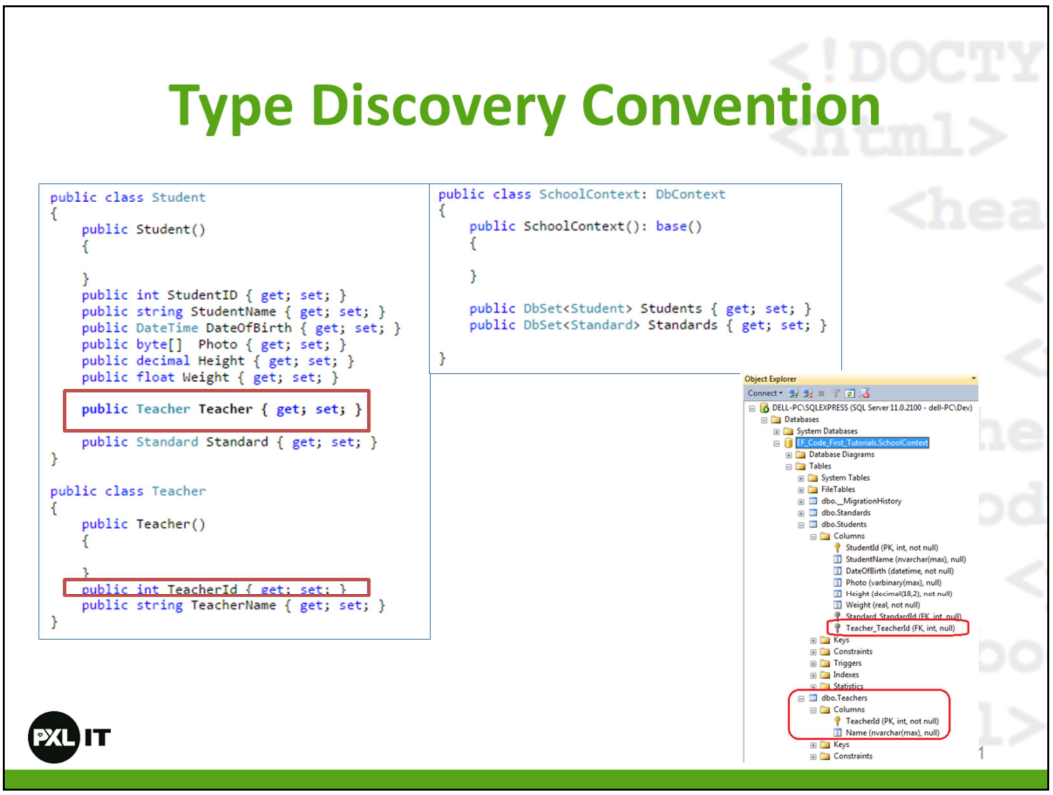
PXL IT

49

So if I go back and generate the model again, we have a one to many relationship. Between Ninja and NinjaEquipment.



There is another way to apply these rules, especially rules that are really about how the classes should be mapped to the database and aren't related to the business rules of the classes. That's by overriding the conventions with what's called Fluent API configuration

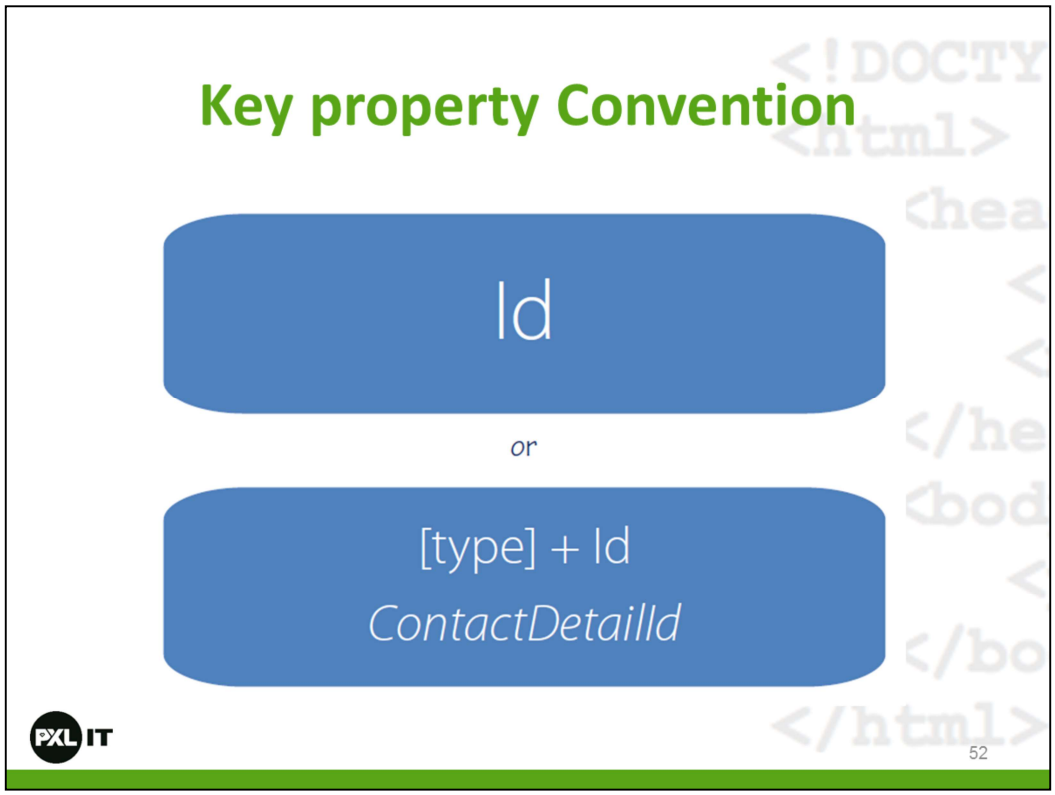


The following Student entity class includes reference of Teacher class. However, context class does not include Teacher as DbSet property. So, even if Teacher is not included as an entity set in the context class, Code-First will include it in conceptual model and create a DB table for it, as shown on the slide.

Code-First also includes derived classes even if the context class only includes base class as a DbSet property.

The conventions for the type discovery are:

- Code-First includes types defined as a DbSet property in context class.
- Code-First includes reference types included in entity types even if they are defined in different assembly.
- Code-First includes derived classes even if only the base class is defined as DbSet property.



The default convention for primary key is that Code-First would create a primary key for a property if the property name is **Id** or **<ClassName>Id** (NOT case sensitive). The data type of a primary key property can be anything, but if the type of the primary key property is numeric or GUID, it will be configured as an identity column.

If you have defined key property other than Id or <ClassName>Id then ModelValidationException will be thrown. For example, consider the Standard class:

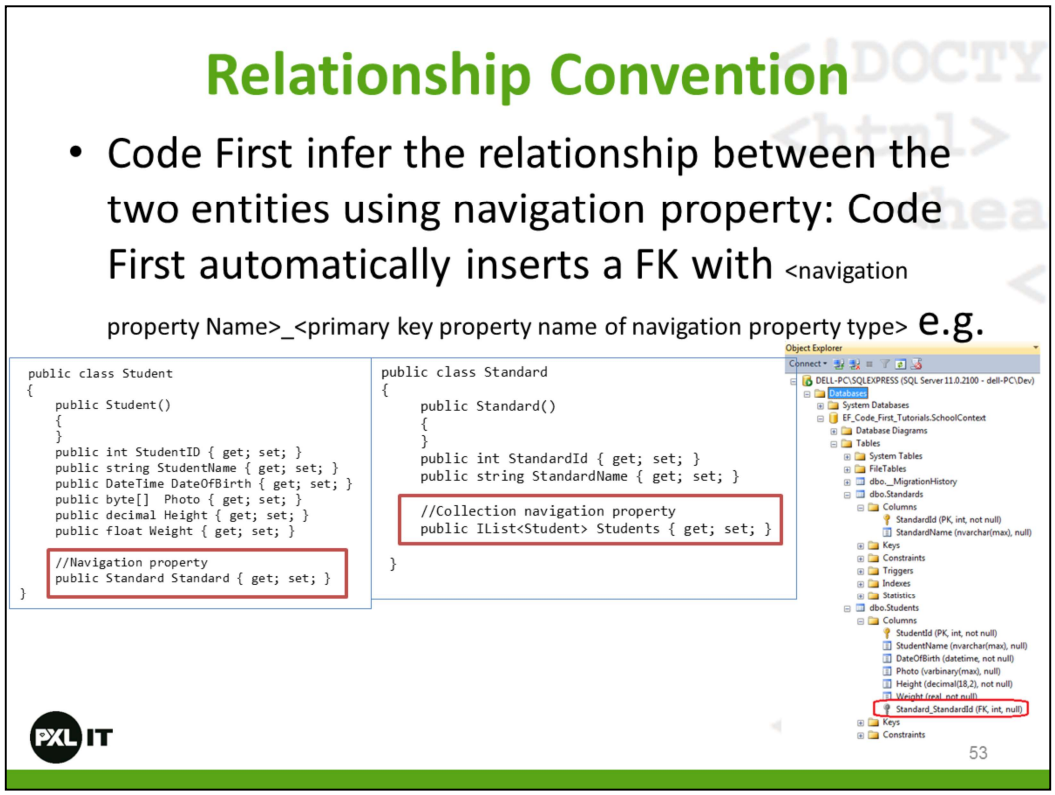
```
public class Standard
{
    public Standard() { }
    public int StdId { get; set; }
    public string StandardName { get; set; }
    public IList<Student> Students { get; set; }
}
```

As you can see in the above code, Standard class is defined with **StdId** key property. Entity Framework will throw the following exception for this.

'System.Data.Entity.ModelConfiguration.ModelValidationException' occurred in

EntityFramework.dll
EntityType 'Standard' has no key defined. Define the key for this EntityType.

If you want to define StdId as primary key then you have to use DataAnnotations or Fluent API to configure it as primary key. We will see how to do it later in these tutorials.

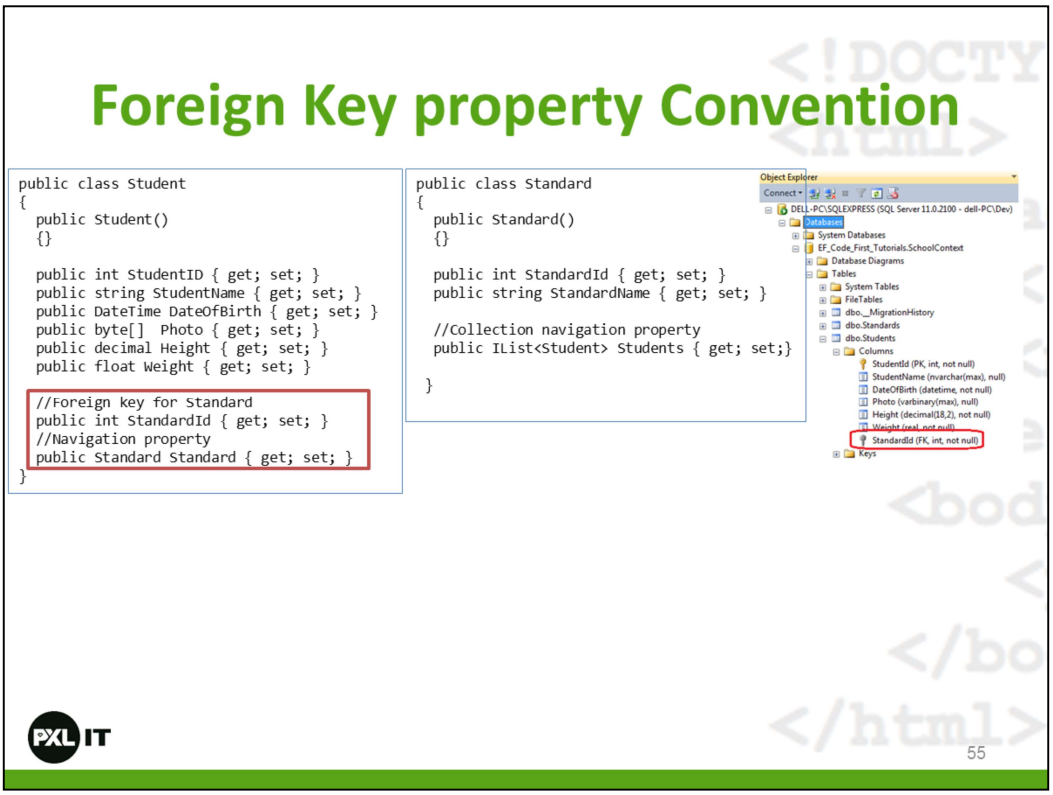


Code First infers the relationship between the two entities using navigation property. This navigation property can be simple reference type or collection type. For example, we defined Standard navigation property in Student class and ICollection<Student> navigation property in Standard class. So, Code First automatically created one-to-many relationship between Standards and Students DB table by inserting Standard_StandardId foreign key column in the Students table.

Thus, the default code first convention for relationship automatically inserted a foreign key with**<navigation property Name>_<primary key property name of navigation property type>** e.g. Standard_StandardId.

Foreign Key property Convention

- It is recommended to include a foreign key property on the dependent end of a relationship (instead of using the Relationship Convention (previous slide))
- Entity Framework interprets a property as a foreign key property if it's named *<navigation property name><primary key property name>*



We have seen above that Code First automatically inserts a foreign key when it encounters a navigation property. It is recommended to include a foreign key property on the dependent end of a relationship. Consider the example on the slide: As you can see in the above code, Student class includes foreign key StandardId, which is the key property in Standard class. Now, Code First will create StandardId column in Students class instead of Standard_StandardId column, as shown below.

Notice that StandardId foreign key is not null in the above figure. This is because int data type is not nullable.

Code First infers the multiplicity of the relationship based on the nullability of the foreign key. If the property is nullable then the relationship is registered as null. Otherwise, the relationship is registered as NOT NULL. Modify data type of StandardId property from int to Nullable<int> in the Student class above to create a nullable foreign key column in the Students table.

Complex type Convention:

Code First creates complex type for the class which does not include key property and also primary key is not registered using DataAnnotation or Fluent API.

This was an overview of code first conventions. These conventions can be

overridden using DataAnnotation or Fluent API. Also, you can define custom conventions for your application using Entity Framework 6.0, onwards.

More on code first conventions: https://msdn.microsoft.com/en-us/data/jj679962.aspx

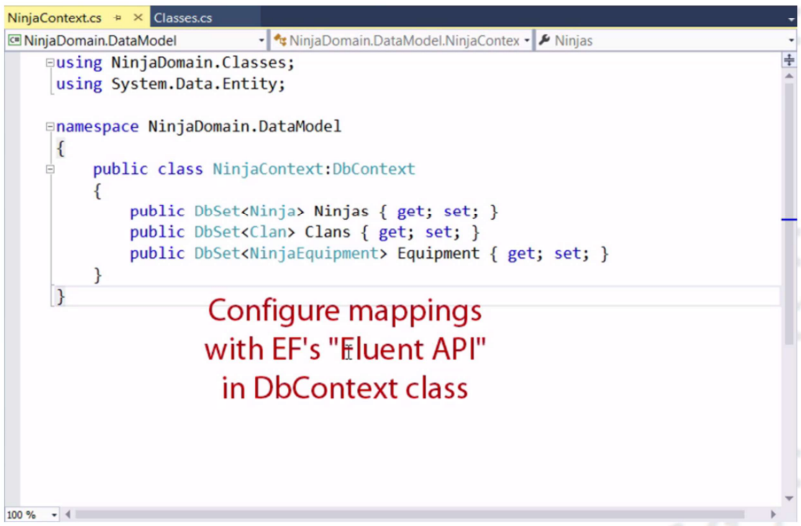
Annotations

```
public class ContactDetail
{
    [Key, ForeignKey("Customer")]
    public int CustomerId { get; set; }
    public string MobilePhone { get; set; }
    public string HomePhone { get; set; }
    public string OfficePhone { get; set; }
    public string TwitterAlias { get; set; }
    public string Facebook { get; set; }
    public string LinkedIn { get; set; }
    public string Skype { get; set; }
    public string Messenger { get; set; }

    public virtual Customer Customer { get; set; }
}
```

PXL IT

Fluent API



PXL IT

57

You write your Fluent API code in th DbContext class.

Configuring mappings with the Fluent API

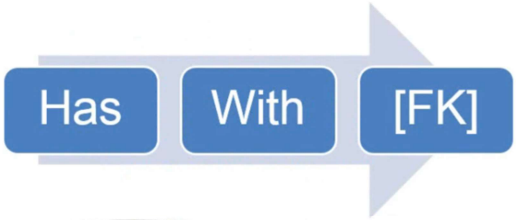
- Fluent API examples (another model is used)

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<ContactDetail>().HasKey(c => c.CustomerId);
   modelBuilder.Entity<ContactDetail>().Property(c =>
     c.MobilePhone).HasColumnName("CellPhone");
   base.OnModelCreating(modelBuilder);
}
```

  - Set CustomerId to PK
  - Set Column Name of MobilePhone to CellPhone

Fluent Relationships

You have to describe both ends of the relationship



Or

```
modelBuilder.Entity<ContactDetail>().HasRequired(c => c.Customer).WithOptional(cu => cu.ContactDetail);
//  modelBuilder.Entity<Customer>().HasOptional(cu => cu.ContactDetail).WithRequired(cd => cd.Customer);
```

PXL IT

59

Fluent API

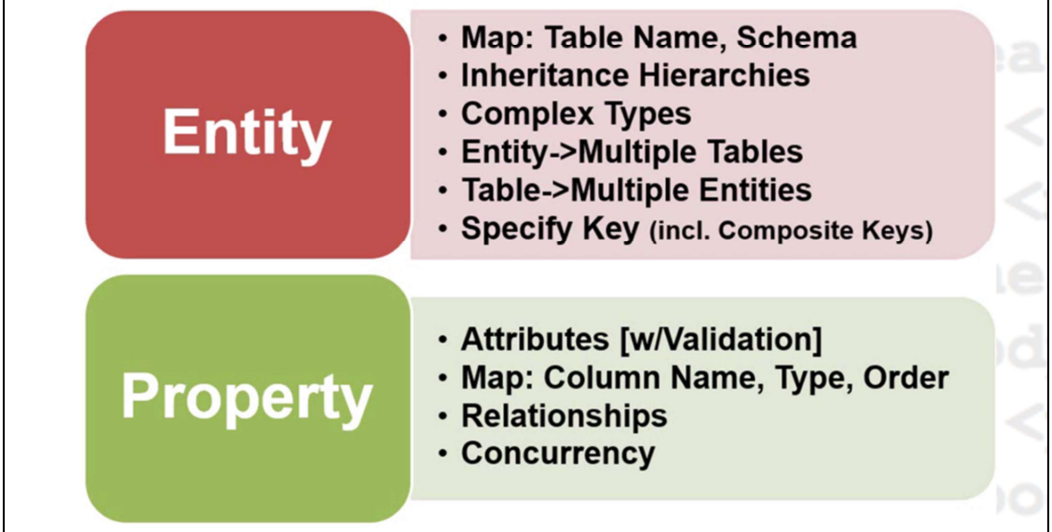
- Alternative (when you have a lot of configurations) → create class

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
  //modelBuilder.Entity<ContactDetail>().HasKey(c => c.CustomerId);
  //modelBuilder.Entity<ContactDetail>().Property(c => c.MobilePhone).Ha
  ////modelBuilder.Entity<ContactDetail>().
  ////  HasRequired(c => c.Customer).WithOptional(cu => cu.ContactDetail
  //modelBuilder.Entity<Customer>().
  //  HasOptional(cu => cu.ContactDetail).WithRequired(cd => cd.Customer
  modelBuilder.Configurations.Add(new ContactDetailMappings());
  base.OnModelCreating(modelBuilder);
}
```

```
public class ContactDetailMappings : EntityTypeConfiguration<ContactDetail>
{
  public ContactDetailMappings()
  {
    this.HasKey(c => c.CustomerId);
    Property(c => c.MobilePhone).HasColumnName("CellPhone");
    HasRequired(c => c.Customer).WithOptional(cu => cu.ContactDetail);
  }
}
```

PXL IT

Fluent API



PXL IT

61

Fluent API

How ModelBuilder Discovers Entities

DbSet

DbSet<Customer>

Reachable by known entity

Customer.ContactDetail

Fluent API Configuration

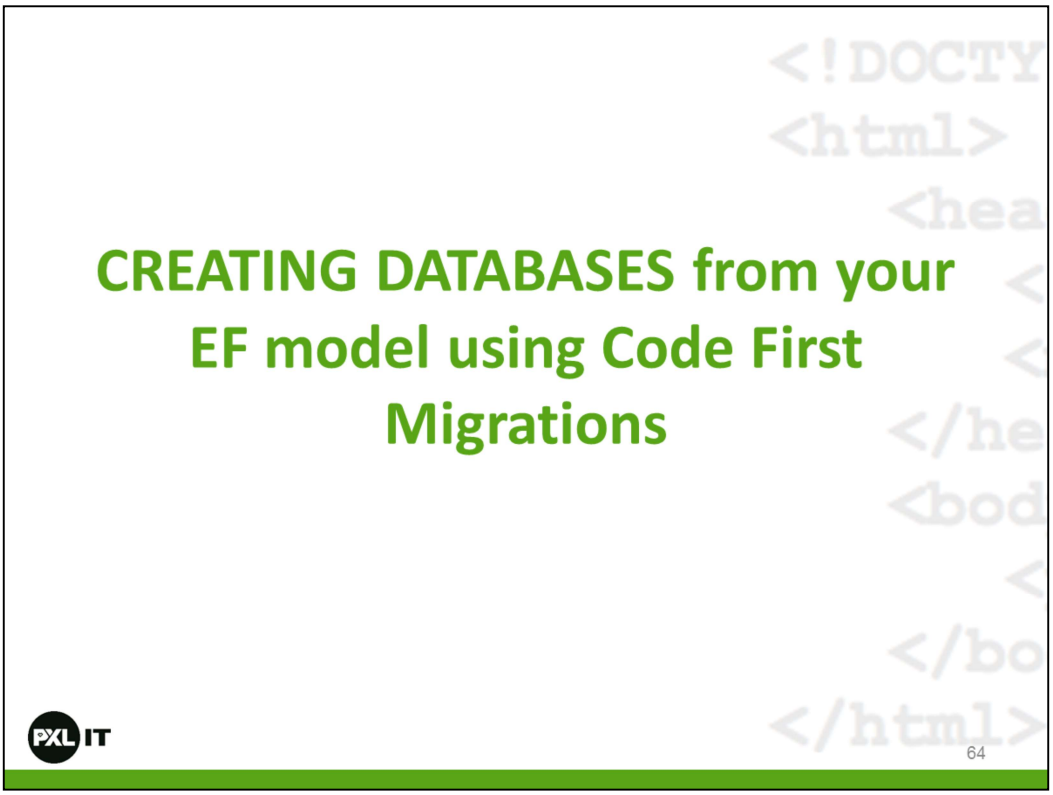
ModelBuilder.Entity<RandomDisconnectedEntity>

PXL IT

62

Fluent API

- Comment out a DbSet statement in the DbContext class → Entity still in the entity model if it has a relationship with another Entity which is in the DbContext class
- `modelBuilder.Ignore<EntityName>();`
  → Entity will no longer be in the entity model



There are number of ways that let you create or migrate a database from your model. Some of these are automated and can happen at runtime and I like to use those when performing unit tests. Otherwise the design-time code first migrations feature is the path where you can have the most consistency and most control.

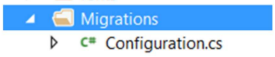
With Code First, when we talk about migrating a database, its in reference to updating the database schema to line with changes you made to the model. EF has an API who helps you do this. It can compare an existing model to its previous syncronisation and figure out what should be changed to the database to line with its changes it's seen in the model. With the help of the database provider, it can read the code that describes the changes and then transform it into the appropriate SQL. The API can also run that SQL or save it for you as a script file to be run another time.

Database Migration

Open Package Manager Console :

View → Other Windows → Package Manager Console

1. Enable Migrations: `enable-migrations`

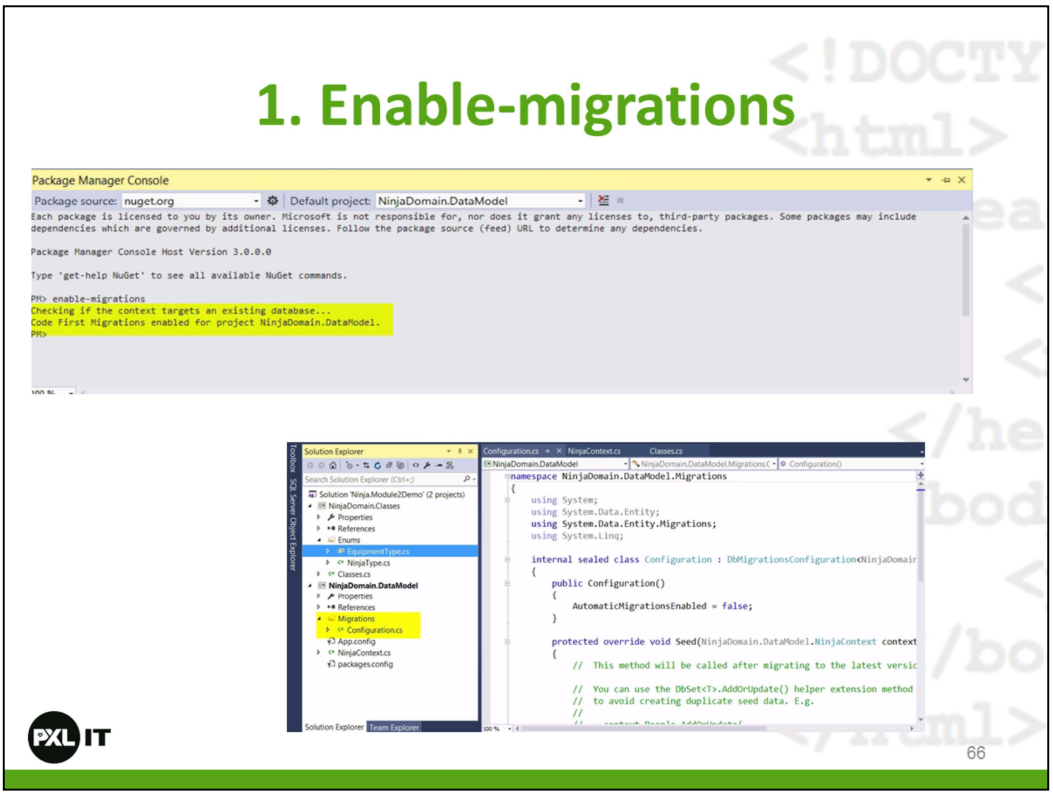


2. Let Migrations create a database
   - In Configuration class (Configuration.cs)
     → `AutomaticMigrationsEnabled = false;`
   - In Package Manager Console:
     `Add-Migration Initial`

PXL IT Creates a Migration File

65

The migrations Api can also create a database for the first time. So we'll start there. I'll tell EF6 that I wan't to use the EF6 Code First Migrations API and then we'll walk through using that to create a database that I can use to presist my ninja data. Then we'll use migrations API to migrate or change that database after making changes to the model

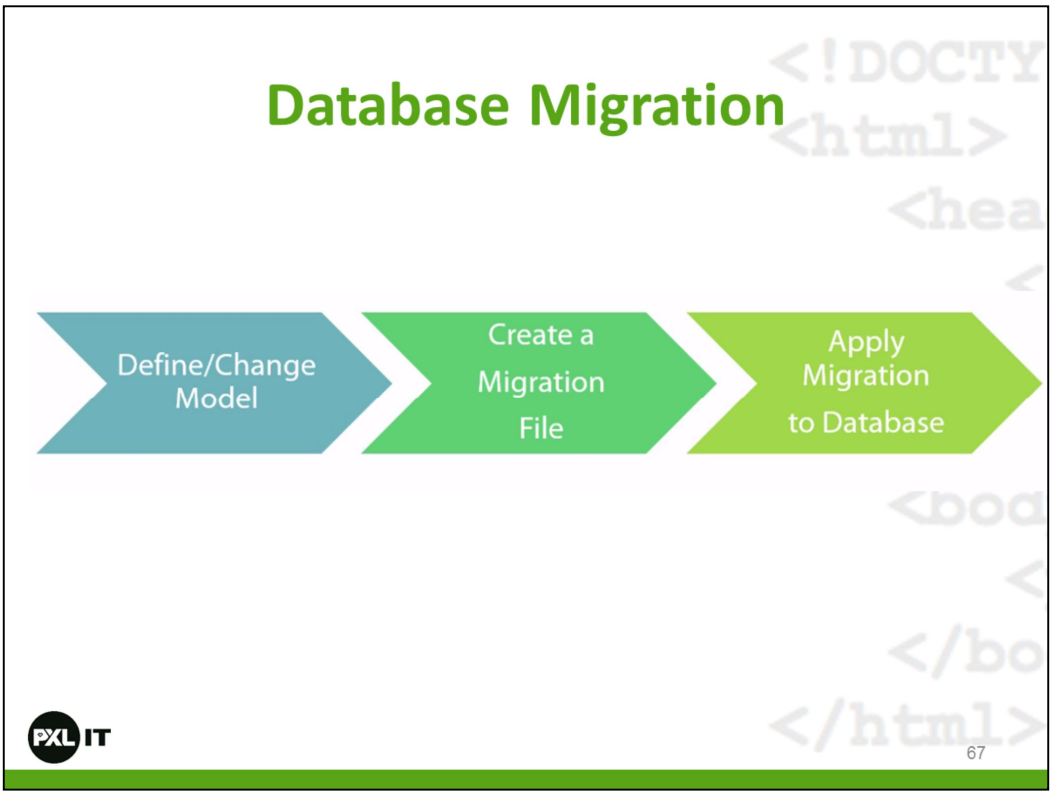


Because I already have EF installed, I can use its **enable-migrations** command in the package manager window. The command has some parameters you can use, but I'm just gonna stick with the default for this demo.

What enabling migrations does, first you can see it checks to see if the context is targeting a database that already exists and in my case there isn't an existing database yet and then it performs the tasks which define enabling migrations in the project.

It's not installing new packages, or new api's, All the migrations logic is inside of the EF dll, but what it did do was it created a folder called migrations and then it created a configuration file which is specific to migrations and so the class by default is named Configuration, but its inheriting from DbMigrationsConfiguration class and you can see that it's setting up a configuration for my Ninja context class. **The only configuration in here is to ensure that migrations won't magically run on their own.** There are a lot of downsides to the automated migrations so they are disabled by default. You better leave them off !

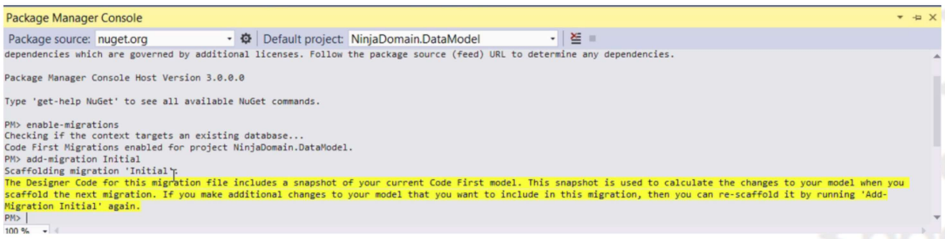
Also, if you want to push some default data into the database, on each migration, there's some sample code provided in the default class that will only seed with data that doesn't exist already.



Performing migrations is actually a 3-step proces: Make changes to the model, Create a migration file that's gonna be a file that defines the things that need to happen to the database in order to reflect the current state of the model. And finally you apply the migrations to the database.

We already have our model defined and in fact we did make a change to it but as I start working with migrations just gonna work with the current state of the model.

Add-migrations



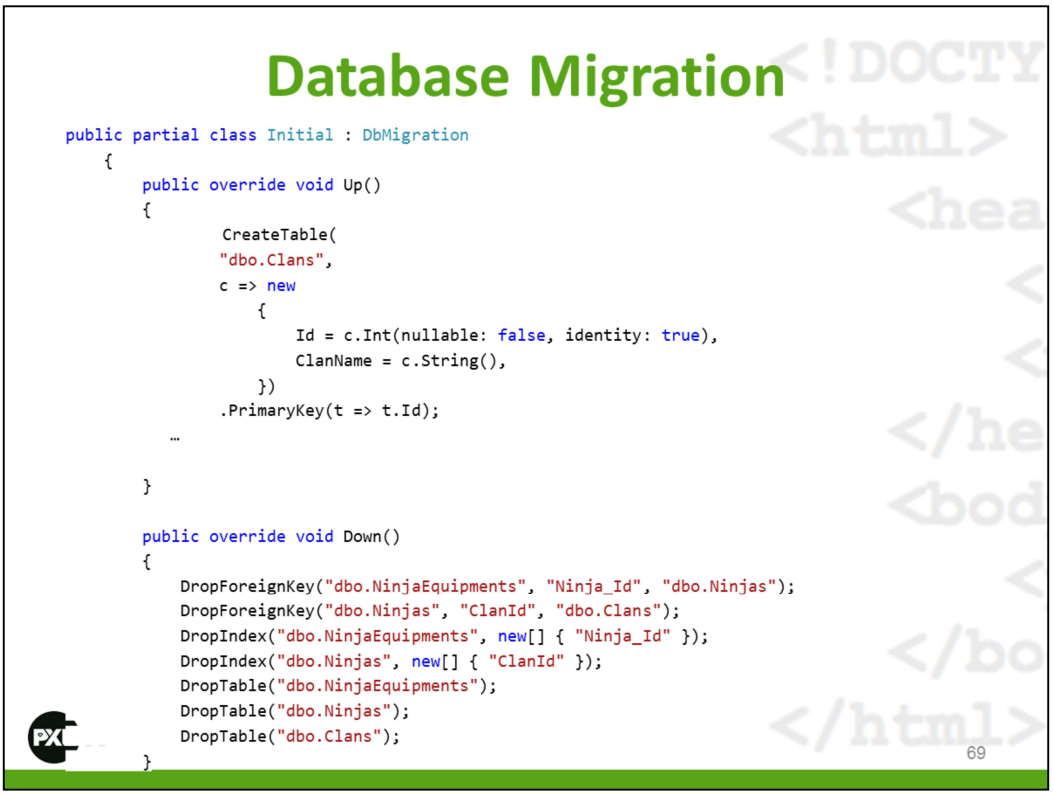
PXL IT

68

So all the work is done in the package manager console. The first thing I'm going to do is say that I want to add a migration and the command for that is add-migration So in this context to migration, EF creates a class to describe the changes that will be needed in the database. This command also has a number of parameters, so you can be really specific about how you want it to work, but I'm going to let it just do it's default behaviour. However, one parameter that you do need to supply is a name for the migration. If you forget it will actually prompt you to it.

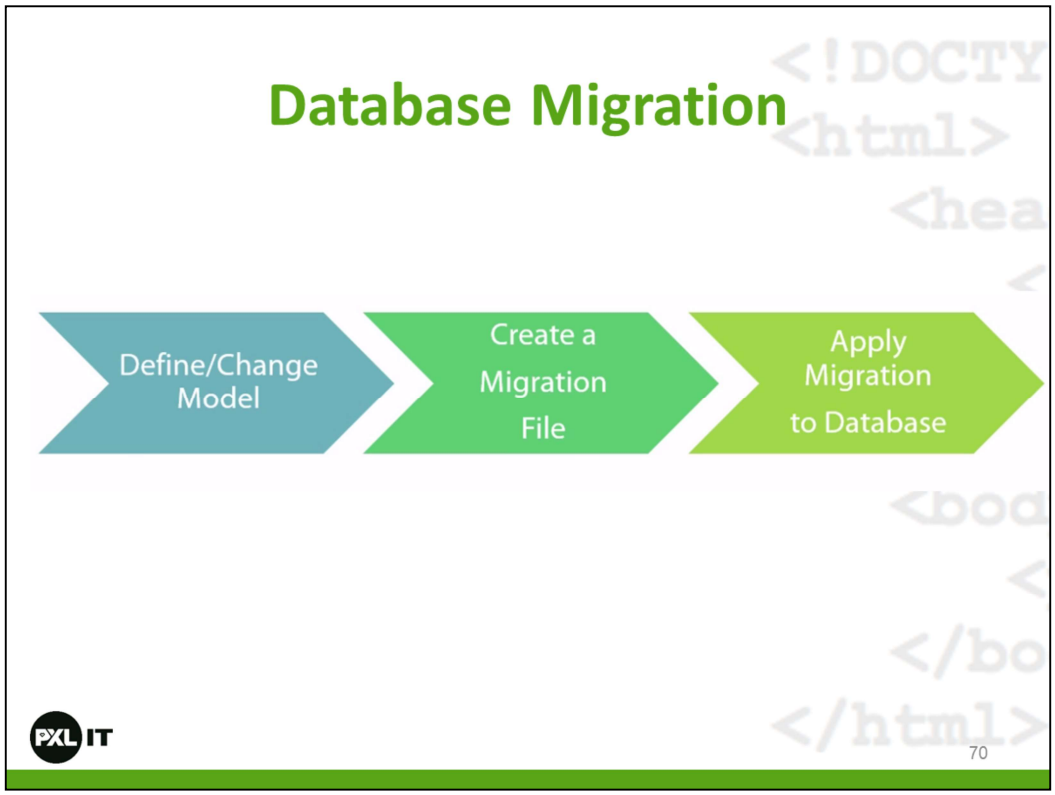
The first migration, I name it Initial.

So what it's done is first it looked to see if I had any migrations already and I didn't so it's going to build up a migration based on the fact that this is the first time we're doing it and the databse doesn't exist yet. So what is it migrating from is nothing. Therefor it needs to create everything from scratch.



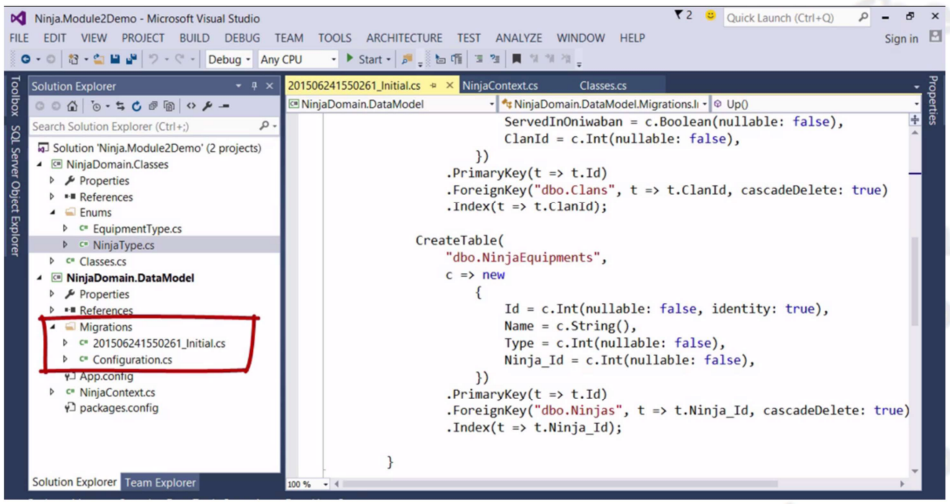
This isn't SQL code It has specific syntax, and the provider will translate that into the appropriate SQL. The migration code says to create a clans table and that should have a column for id that's based on a CLR int. As well as a column for ClanName which is based on the CLR string. Notice that it recognizes the ID to be a PK. It also has instructions to create the Ninjas table with it's relevant fields. In addition to the PK, it's aware that it will need the PK-Fk constraint to the Clans table using the ClanId property. And by default EF presumes that constraints included cascade deletes.

Another default is to create an index on the FK property. And finally, in the NinjaEquipment table, it's doing the same thing.



So Step 1 was defining the model, Step 2 was creating this migration class.

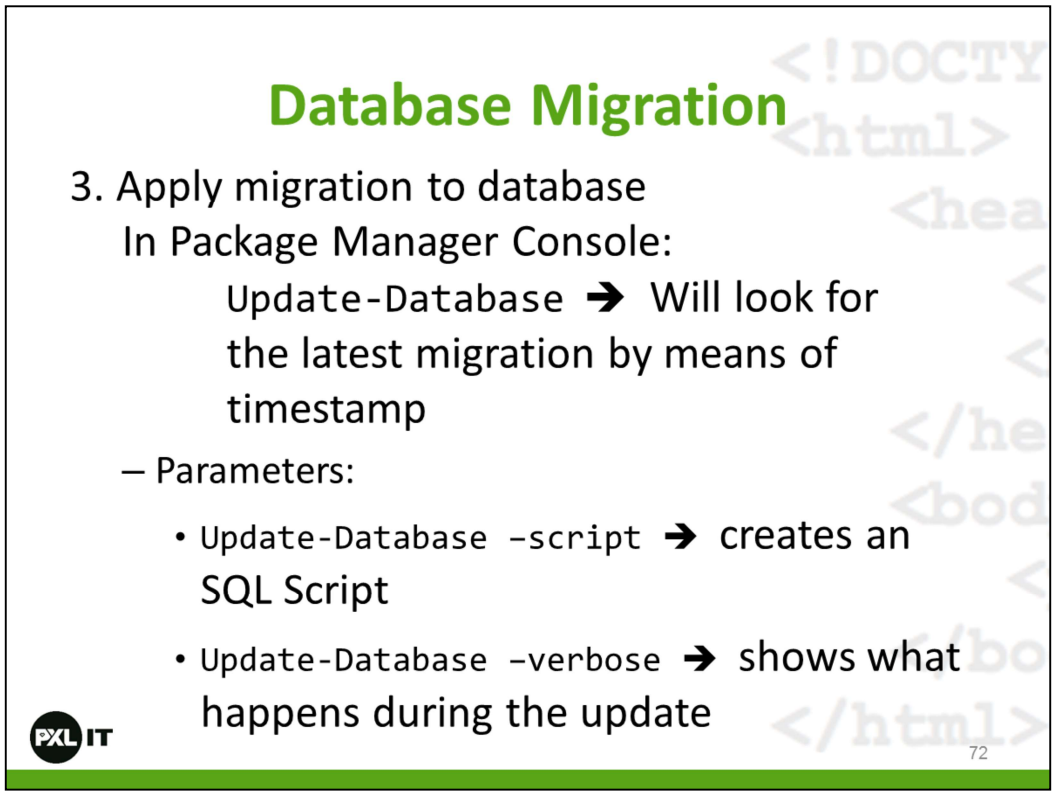
Database Migration



PXL IT

71

Notice also that these classes right here inside of the migrations folder

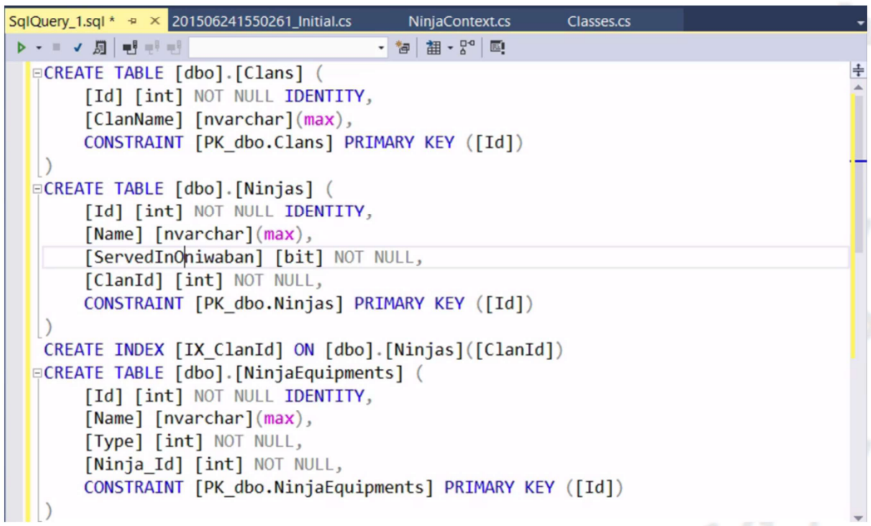


The last piece of this steps is to apply the migration and that's done with the command called update-database

There are a number of parameters that really control how this works. There are 2 parameters that I want to make sure that you see,. One is the script parameter. It make the difference between literally working againts the database versus simply creating an SQL script.

I will perform the command update-database –script. Because of the information I supplied in the app.config (mssqllocaldb + db provider SqlClient)

Database Migration

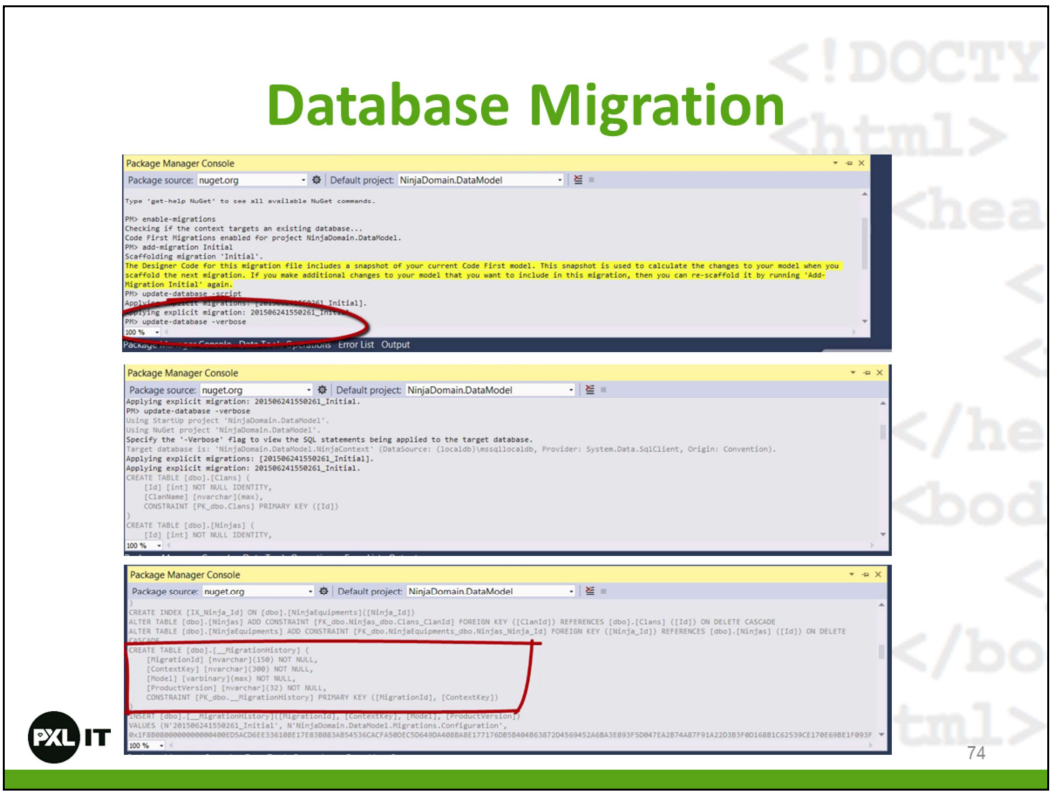


PXL IT

73

Based on the information it found in the Migration class, it was able to create this T-SQL script.

I can run it, give it to my DBA, and have him run it.



This time, I'm not gonna use the script but I will use a different parameter which is verbose and what that will do is it will show me in the package manager console exactly what's happening in the database . Otherwise it will do all these steps behind the scenes.

It created the db for me, It ran that same T-SQL that I've shown you before. And the last piece it did is it keeps track of the migration literally in the database. It's creating a Migration history table and this is where it will have in the db keeping track of the latest migration, this is just a hash of the migration that was just applied. It's a representation of the model itself, which comes into play the next time I want to create a migration, so lets go and look for that database, with its tables, columns, PKs, FKs,… There is no data yet, it's just the database.

Migrating Database when your model changes

- Add property to the Ninja class:

```
public DateTime DateOfBirth { get; set; }
```

- In Package Manager Console:

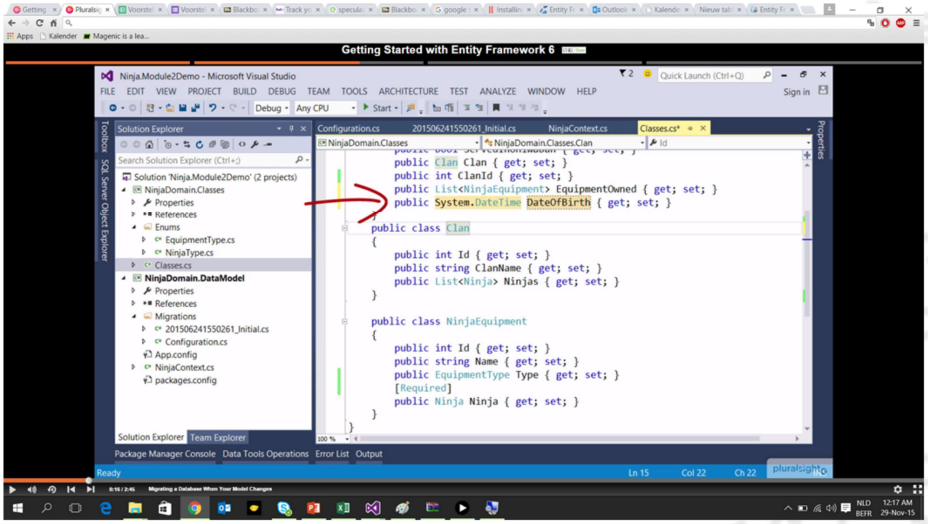
```
Add-Migration AddBirthdayToNinja
```

➔ Adds `<timestamp>_AddBirthdayToNinja.cs` to the migrations folder in the Solution Explorer

PXL IT

75

Database Migration

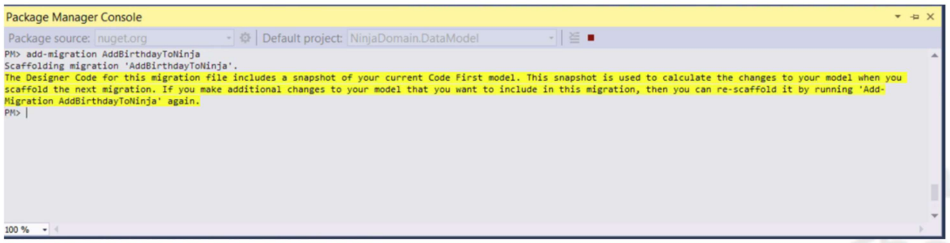


PXL IT

76

Let me make another change to the model. Then I'll go through the migration process again, so you can see how I'm then able to migrate the database. I added a simple scalar property to the Clans class, just to have a quick change going on.

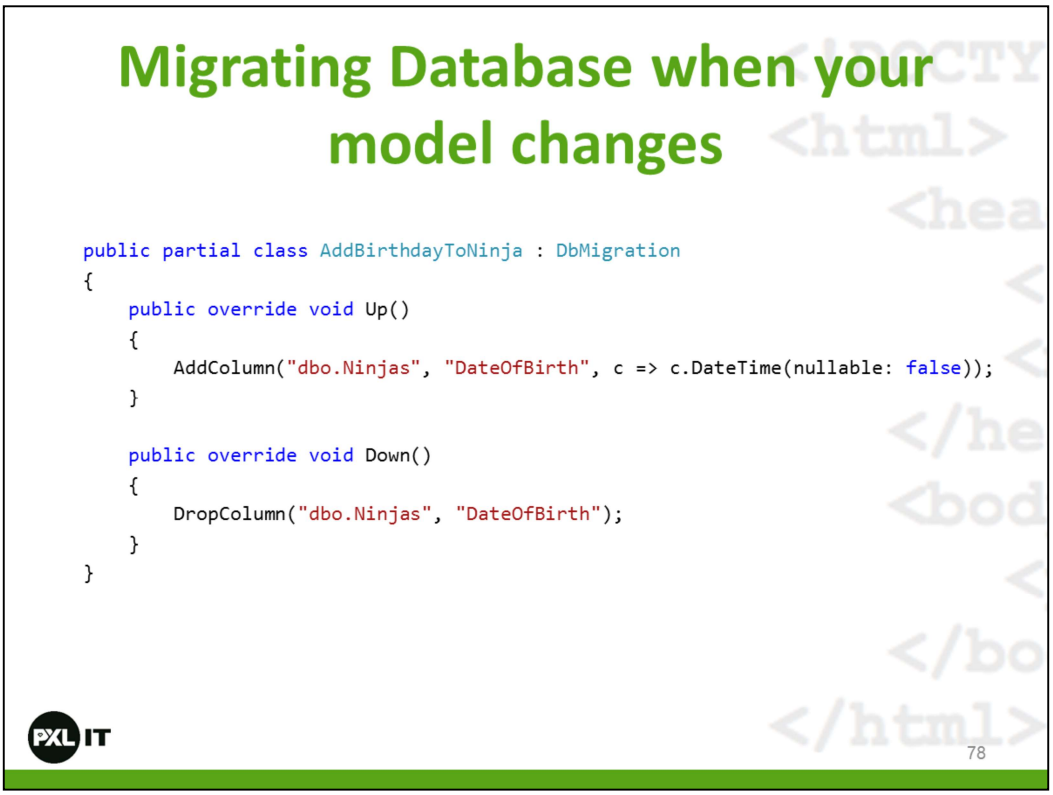
Database Migration



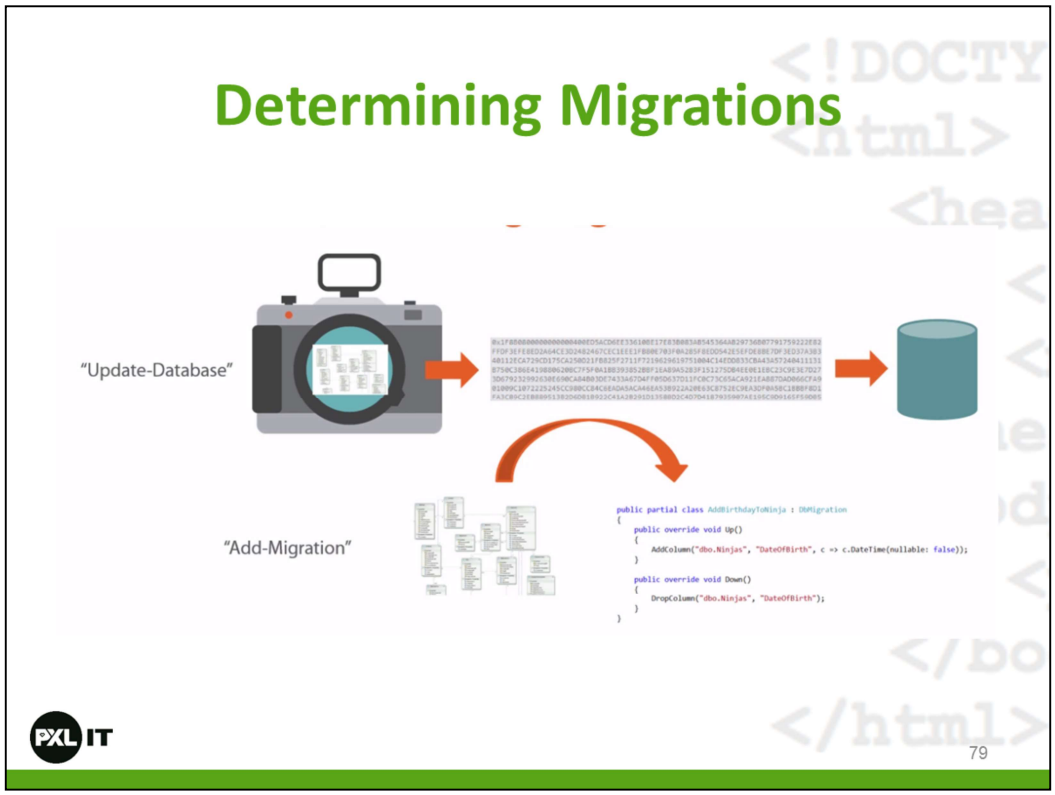
PXL IT

77

And now I will add another migration and I'm going to give it a name, so I know what I did,



And here's the code. Now notice that I'm not creating tables, or do anything like that, all I'm doing is adding the column. That's the beauty of the migrations, you can have an existing database , you can have data in that database, You're dba have may gone in there and tweeks some things for performance, adding more indexes triggers, things like that, Nothing of that will get messed up All we're gonna do is adding a column.



Remember when I update the database, and pointed out migration history, table, what EF does is, it takes a snapshot of the model, creates a hash from that, and then stores it into that history table. Then, the next take you add a migration, EF compares the hash of the current model to the most recent hash that was stored in the db. Then, based on the differences it finds, it can determine what migriations need to be applied. There is logic in there to protect you from deleting columns that might have data in them, and you can even override that.

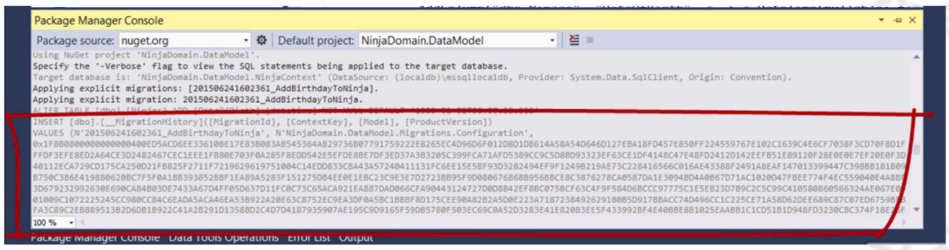
Determining Migrations

PXL IT

80

The one thing you're probably wondering about is , I have an Up method, with changes that need to be applied to the database, and there's also a Down method. The Down method is used if you need to do more management of this migration. Maybe you need to reverse some of the migrations. So if you need to do some migrations to be ondone, The Code First Migrations API will understand to read the Down. You can do this by: Update-Database -Target:migrationsClassName

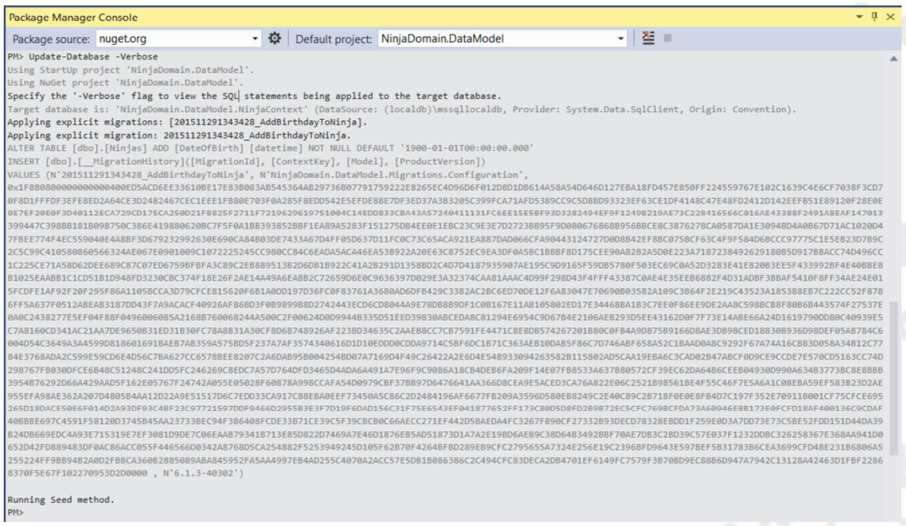
Determining Migrations



PXL IT

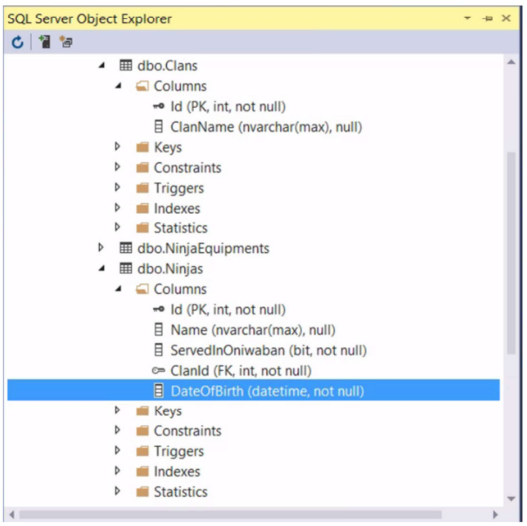
In the package manager, you can do an update-database –verbose, so you can see the SQL code. You can see that I'm adding the current version of the model to that history table.

Determining Migrations



Do an update-database –Verbose to add the column to the ninja table

Determining Migrations



PXL IT

83

And of course if we look at the ninja's table, you can see now DataOfBirth is there. And again, we don't have data in the database yet,

Seeding a database using migrations

- Seed method: Runs after upgrading to the latest migration to allow seed data to be updated

Seeding a database using migrations

- AddOrUpdate method (part of the migrations API):
  - uses specified property to check for already existing data before adding any new rows
  - if a match is found → Code First will replace the entire existing row (won't just update one or two values)