Transcript - English

• Entity Framework 5: Planning Ahead

o Introduction

Hello, this is Julie Lerman for Pluralsight. Welcome to my course on Entity Framework 5. This first module is aimed to help you prepare for working with EF5 whether you're new to Entity Framework or moving to EF5 with experience from earlier versions of Entity Framework.

o Objectives

I'll begin with a high level overview of what Entity Framework is and why you might want to use it for your data access needs. Then I'll make sure you're clear how Entity Framework 5 works in relation to .NET 4 and .NET 4.5. For newbies, I'll provide a very quick first look at using Entity Framework, and then I'll talk about what's been added to Entity Framework in version 5. You may be moving an existing solution that's already using EF4. So I'll give you some pointers about moving from EF4 to EF5. Later modules in this course will focus on the different modeling workflows. So in this module, I'll give you some guidance for choosing which modeling workflow you'll want to use. Finally, because Entity Framework 6 is already in the works, you'll learn a little bit about what's in EF6 so you can keep those upcoming features in mind as you move through the rest of this EF5 course.

o Why Entity Framework

Why would you choose Entity Framework as the data access tool for you applications? Entity Framework is an Object Relational Mapper or ORM. ORMs are aimed to increase developer productivity by relieving you of the tedium and redundant task of persisting the data that you use in your applications. I can say, after many years of software development, when I want to focus on my business domain and the user experience, having to worry about the database and how to get data in and out of it is such a pain. Having to do that over and over and over again is really a bore. So taking advantage of its default behaviors, Entity Framework can generate the necessary database commands for reading or writing data and execute them for you in the database. If you're querying, you can express your queries against your own domain objects using LINQ to entities. Entity Framework will execute the relevant query in the database and then materializes results into instances of your domain objects for you to work within your app. So you focus on your domain and Entity Framework takes care of the database work. There are other ORMs in the marketplace such as NHibernate and LLBLGen Pro. Most ORMs typically map domain types directly to the database schema. Entity Framework has a more granular mapping layer so you can customize mappings, for example, by mapping the single entity to multiple database tables or even multiple entities to a single table. Entity Framework is Microsoft's recommended data access technology for new applications. As far as I can see, they've even stopped referring to it as ADO.NET Entity Framework it's so different than ADO.NET. ADO.NET now seems to refer directly to the technology for data sets and data tables. And that's not going way anytime soon. There are too many apps that depend on it. But Microsoft isn't making any more investment in ADO.NET. What about LINQ to SQL? Another ORM in the Microsoft step. It, too, will stick around and it has had some minor investment mostly bug fixes. But Entity Framework is where all of the forward moving investment is being made. And that's been the case for a number of years already. Microsoft recommends that you use Entity Framework over ADO.NET or LINQ to SQL for all new development.

o EF's Conceptual Model

For developers who are used to database focused development, the biggest shift with Entity Framework is that it lets you focus on your business domain. What it is that you want your application to do without being limited by what the database is able to do. With Entity Framework, the focal point is referred as a conceptual model. It's a model of the objects in your application not a model of the database you use to persist your application data. Your conceptual model may happen to align with your database schema or it may be quite different. You can use a Visual Designer to define your conceptual model which can then generate the classes you'll ultimately use in your application. Or you can just define your classes and use a feature of Entity Framework called Code First. And then Entity Framework will comprehend the conceptual model. Either way, Entity Framework is able to work out how to move from your conceptual model to your database. So you can query against your conceptual model objects and work directly with them. If this is your first look at Entity Framework, you might be interested in a simple list of its basic features. I've created this list based on the things I think are most notable about Entity Framework and also from frequently asked question about Entity Framework. Even though Entity Framework is a Microsoft tool, you may not realize that it's being developed as an Open Source product. This happened when Entity Framework 5 was released. The EF team was the ultimate responsibility of making commits to the code base, but the developer community is already getting involved in discussions, code review, and even submitting their own code, some of which are already been committed towards Entity Framework 6. Entity Framework is no longer tied to the .NET release cycle. .NET is released about every two years. Since EF isn't built in to .NET as of EF 6, it can be released when it's ready. So we'll get new features for Entity Framework much more rapidly that .NET updates. While Microsoft provides Entity Framework support in the SQL Server provider, there are many third party providers that allow to you to use Entity Framework with a variety of databases from commercial databases like Oracle to Open Source databases like MySQL and Firebird. You write your queries with LINQ, LINQ to entities and Entity Framework breaks those down into something called command trees. Then the provider will transform the command tree into the appropriate SQL for the targeted database. If you hard code a value into LINQ query, Entity Framework will do the same with the SQL. The value will be hard coded into the query. But if you use variables to supply those values then Entity Framework will create a parameterized SQL query. Entity Framework is able to keep track of changes made to objects that it's aware of, including adding new objects or deleting some. You'll have to follow its rules though the rules have gotten more flexible overtime then you can call an Entity Framework command which is called SaveChanges and it'll check the changes you've made and create and execute the appropriate Update, Delete and Insert commands in the database. If you're designing disconnected apps where Entity Framework won't be around at the time the objects are actually being edited, there are patterns you can follow to let Entity Framework know what changes it should be aware of when you pass the objects back for Entity Framework to say back to the database. As mentioned earlier, you can use a Visual Designer to create your model and then leverage Code Generation to create classes from the visual model, or you can use the Code First feature which lets Entity Framework work with your own class definitions. Entity Framework has a variety of ways that it supports toward procedures. With EF5, the best support is when you're using the Visual Designer to define your model. You can map your own Insert, Update, or Delete procedure to a model object, and then Entity Framework will use those procedures when it calls save changes instead of creating its own SQL commands. The Code First method of defining a model doesn't support this behavior with Entity Framework 5 but it is a plan feature for Entity Framework 6. Finally, I want to point out that Entity Framework have some very robust default behavior which has lead to its nickname The Magic Unicorn. But don't be fooled by this simplicity. Entity Framework has a complex API that let's you have granular control over everything from its modeling to it's runtime behavior. I've mentioned a few times that Entity Framework isn't dependent on the .NET Framework release cycle. Versioning has been somewhat confusing with Entity Framework, so I want to clarify what that means in Entity Framework 5 and also what it'll mean in Entity Framework 6. Part of Entity Framework 5 lives inside of .NET. And another part of it lives inside of an additional assembly that's distributed using NuGet. What this means is that the core functionality of Entity Framework is built in to the .NET Framework. But the Code First support, that's what lets Entity Framework use your classes in lieu of a visual model, and a lighter way API for interacting with EF are in the NuGet package. The core is what provides the querying, change tracking and all of the transformation from your queries to SQL queries and from data return from the database into your objects. You can use the EF 5 NuGet package with both .NET 4 and with .NET 4.5. But here's one big point of confusion, .NET 4.5 added support for enums and spatial data to the core Entity Framework APIs. That means if you're using EF 5 with .NET 4, you won't get these new features. You'll only get them when combining EF5 with .NET 4.5. Now let's look at what's going to happen in EF6. This is pretty cool, the team is taking the core APIs that currently live inside of .NET and pulling them into the NuGet package. That means all of the Entity Framework lives insides this assembly that's distributed by NuGet. That also means we won't be dependent on .NET to provide specific features like the Entity Framework enum support and special data support. So you'll see that one of the features of EF6 is that it supports enums and spatial data for .NET 4 and .NET 4.5. I'll talk a bit more at the end of this module about what else to expect from EF6.

o EF's Basic Workflow

Let's take a quick look at a simple workflow for working with Entity Framework. The first thing you need is a model. You can start with a database and empty Designer a code. If you start with a database such as this small sample customer's database, you can use Visual Studio's Entity Framework Designer to reverse engineer the database into a model. In turn, the Designer will generate code classes for you to use in the application along with a context which manages those classes in behalf of Entity Framework. You can also start with a blank model and design the model and generate the database from there. Or you can just start with your own classes and use Entity Framework's Code First to have Entity Framework work with them. Once you have a model, you can use the context to let you create queries using LINQ to entities. The queries are written against the classes generated from your model or the classes you created for Code First. So you don't have to write your queries against the database and Intellisense is a big help. Entity Framework will determine the SQL and execute it in the database. Here's the query that SQL Profiler tells us was run for the customer query. Then Entity Framework takes the results and creates object instances from them. And here's a quick look at some of the values from those objects in a Console window. As long as the context is aware of the objects, it will keep track of the changes you make. If it's a disconnected application, we do have to do a little extra work when reconnecting the data to the context. Then you can call Entity Framework SaveChanges command. That will figure out any UPDATE, INSERT, or DELETE commands for the database and execute them one at a time. And they are executed in a transaction, so that will roll back if anything fails.

o What's new in EF5

For those of you coming from prior versions of Entity Framework, it's nice to have a specific list of exactly what's changed. In my opinion, the most significant change in Entity Framework 5 is one that improves query performance dramatically. Translating your LINQ to entity's query into a SQL query is an expensive operation. So, now Entity Framework will cache the SQL that it works so hard to create and reuse it when you execute the same query. This will also take into account parameters. Entity Framework will recognize parameterized queries and reuse the cache query even when the value of predicate has changed. When I first read about this feature, I wondered how easy it would be to move Entity Framework for projects to EF5 just to get this performance benefit. But it turns out that to get that one benefit, your project doesn't have to use Entity Framework 5 or even target .NET 4.5. Thanks to the in-place upgrade from .NET 4 to 4.5. As long as .NET 4.5 is installed in the machine where the Entity Framework logic is running, your app will get the benefit. So you don't even need to recompile your app with .NET 4.5, and because this performance increases pretty significant, the fact that you can get it for free, so to speak, even in your .NET 4 apps is pretty impressive. As mentioned already, we gained support for enums and for spatial data like SQL geography. Both of these are supported whether you're using the Entity Framework Designer or using Code First. Remember, these are dependent on .NET 4.5. In EF5, if you're using the Designer, you can map to table valued functions in your database and execute them easily from the context. If you're using Code First in evolving your model, there's a nice change when you're adding new entities to your model that require new tables to be added to the database. Instead of noting this as a breaking change to your model that requires a drop and create, Code First will just add the new tables to the database and move on. There are some great improvements to the Entity Framework Designer. Color coding entities is pretty nice. I think the most significant, though, is that you can create different views of your model using a diagram feature. This makes it easier to focus on subsets of the model when you're working with it visually. Another big change is that by default, the Designer will use the Code Generation T4 template that creates simple classes in a DbContext. And just because it's really important, I'll repeat, that as of EF5, Entity Framework is now open source. I'll give you a quick tour around its codeplex pages towards the end of this video.

o Upgrading EF4 Projects

When the Visual Studio 2012 Beta was released, I created a short video on moving a project from .NET4 with Entity Framework 4.3 where I was already using Code First DbContext and migrations to .NET 4.5 and EF5. There are few critical things to keep in mind when upgrading your projects. So, I'll do a quick review of moving an existing solution using Code First to EF5 and then I'll do the same with a project that uses a visual model. This will be simple solution so I can just focus on the key ingredients of moving. Here's my older solution and all the projects in here target .NET 4 and the Console app and the Data Layer projects have references to Entity Framework 4.3. Before upgrading, I'll show you the secret sauce of the Entity Framework 5. Here's a folder of an Entity Framework 5 solution that's using .NET 45. In the solution folder, there's a packages folder for all the NuGet packages used in the solution. Drilling in to the folder for EF5 then the libraries, notice there's a folder for NET 4 and a folder for NET 5. Entity Framework has two assemblies. One is targeted to .NET 4 and the other to .NET 4.5. That way, you don't have to worry about which version of the Entity Framework 5 to install. You just install EF5 and it'll pick up the correct version of the assembly for you. So, flipping back to the project I want to upgrade, if I install the Entity Framework 5 right now, it'll download both assemblies but it will add a reference to the .NET 4 version of Entity Framework 5. That means that first I need to change the target framework of my projects before updating to EF5. Once I've done that, I can go ahead and update the NuGet package to EF5. I'm only using Entity Framework in these two projects, so those are the only ones I want to update. And now you can see that Entity Framework assembly I'm using in my project is indeed version 5.0.0.0 and that's the one that does target .NET45. So people have been really confused by that in the past because they're upgrading but they don't change the framework version and they don't understand why they're not seeing some of the new features like the enum Support, when they've upgraded to EF5 but the version that's targeting the wrong version of the framework. Now, here's an old Entity Framework 4 solution that uses a visual model and the object context API. It's not even using the DbContext API but instead the original core stuff. I can open this up in Visual Studio 2012 and everything just works. You can see I ran my test they all passed. So that's not really upgrading to EF5 but just using the original app in Visual Studio 2012. But there are a number of ways to upgrade. For example, I might want to get the enum support that's been added to EF5. But for that, all I need to do is change the .NET target to .NET 4.5. I don't need to download the NuGet package to get that. So I've made that change to all of these projects, but when I go to open the model in the Designer, I get this error message that says the XML namespace for this model is incorrect because the specified namespace is targeting the older version of Entity Framework. And it suggest you can change the XML namespace by opening up the model in XML but if you just click on this modify link, it'll do the job for you. So, click on modify, now the model opens up, and I want to show you the XML so I'm going to switch to that. This is by right clicking on the model and choosing Open With, and I need to say that before I can open it in a different editor. So I want the XML. And what it changed is this line here with the EDMX that was targeting the namespace. Originally, it was EDMX version 2 and the namespace was, I think, it was ADO/2008/-- something or other. So the Designer went ahead and fixed the problem on my behalf. Now that I've got the model open the Designer, you can see now I've got the ability to create enum types in the model. That's there because now I'm targeting .NET 4.5 where enum types are supported. If the project that contains this model was still targeting .NET 4, enum types wouldn't even be listed in the model browser because the Designer understands the difference between the target frameworks. And now that I've got everything upgraded to .NET 4.5, even though I haven't made any other changes to the project or to the model, I went ahead and rerun my test, everything still works. So now I can move forward. Another way I might want to upgrade this project is to change the Code Generation. Currently, this is using the old original Code Generation which creates an object context and where each of the entities inherits from the entity object. So if you wanted to change this to use the DbCOntext, it's easy enough to change the Code Generator to use a different T4 template, but you have to be careful because the DbContext API works differently than the object context API. So code that you've already written that depends on the fact that we're currently using object context and entity object will need to go through big changes, but the same big changes you would have had to go through to change from Entity Framework 4 to Entity Framework 4.1. So it doesn't make a difference. If you're going from EF 4 to EF 4.1 or if you're going from EF4 to EF5 when it comes to making that bigger change of switching the T4 template, and this particular module is not the place to drill into those kinds of changes.

o Choosing DB First, Model First or Code First

Other modules in this course will focus on building models with the Designer or just using Code First. So in preparation for that, I wanted to provide some guidance here on which modeling workflow you might want to follow. First, I should be sure you know what I mean by the different modeling workflows. You've already seen examples of Database First modeling and a Model First modeling workflow earlier in this video. Both of these workflows used the Designer but one starts with the database to create a model and the other starts at the model to create a database. But not everyone is happy using a Visual Designer plus code generation. For those developers, we have a completely different workflow called Code First. The typical workflow for Code First is great for brand new applications where you don't even have a database. You define your classes and code and then let Code First figure out what you're database should look like. It's also possible to start Code First with a database and yes, that makes Code First a bit of a misnomer. But there's a tool to let you reverse engineer a database into classes which is a great way to get a head start on the coding. So given these options, let's look at this Decision Tree. If you prefer to work with a Visual Designer in generated code, then you'll want to choose one of the work flows that involves the EF Designer. If your database already exist, then Database First is your path. If you want to use a Visual Designer on a brand new project without a database, then you'll want to use Model First. If you just want to work with code and not a Designer, then Code First is probably for you along with the option of using the tool that reverse engineers the database into classes as a one-shot way to get a leg up on building those classes. If you have existing classes, then your best bet is to use them with Code First. I'll dedicate separate modules of this course to each of these different options.

o What's Coming in EF6

When Entity Framework 5 was released, work on EF6 had already begun. I think it's important to keep an eye to the not too distant future when learning about EF5. So, here's a quick list of what to expect in Entity Framework 6. You can follow Entity Framework 6 progress and even contribute to it at entityframework.codeplex.com. Nightly builds are release if you're interested in playing with them as soon as possible. Or you can wait for the periodic releases on NuGet. The basic way of working with Entity Framework won't change. Much of EF6 is devoted to more advanced scenarios, for example, some of it to long awaited features such as stored procedure support coming to Code First. Here are some of the other features you can look forward to in EF6. The new Async/Await pattern introduce some .NET 4 5 will be supported by EF 6. You'll be able to create your own conventions for Code First. If you're working with multi-tenant database, Code First migrations will provide support for that scenario. I mentioned earlier in the video that the core Entity Framework APIs that are currently part of .NET will be moved to the Entity Framework NuGet package. Thanks to this, we'll be able to use enums and spatial data even in .NET 4 applications. The runtime isn't the only place where Entity Framework 6 will bring changes. The tooling is also changing. The EF PowerTools will be incorporated into the Entity Framework Designer. These are some of the more prominent changes coming in EF6 but keep tabs on the codeplex site for more information.

o Summary

I hope this course has provided you with sufficient information to plan ahead for using Entity Framework 5 whether you're new to Entity Framework or you've worked with earlier versions. I tried to clarify some of the confusion around versioning, shown the basics of how EF does its job and talked about what's new in EF5 and what's currently being worked on EF6. I've also shown you some tricks for moving your apps from EF4 to EF5 and provided some guidance for choosing a modeling workflow. The rest of this course will drill into specific features and tasks for getting started with the Entity Framework 5.

o Resources

I'm Julie Lerman, and thanks for watching this video on planning ahead for Entity Framework 5. Here are some relevant resources and references you might find helpful. ( Silence )

• Entity Framework 5: Database First Modeling

o Introduction

Hello, this is Julie Lerman for Pluralsight. In this video, I'll show you how to create an Entity data model using an existing database and then show you how to understand and interact with that model in the Entity Framework Designer.

o Outline

I'll begin by creating the model using Visual Studio's Entity Data Model Wizard. Then we will take a closer look at what the model created for us. Next, I'll show you two new features introduced in Visual Studio 2012. The ability to add color to your models and to break them up with diagrams. Then, you will learn how to update the model when your database schema changes and more about having your model leverage your existing database stored procedures. Then you will see how to use the model to take advantage of the new Enum and spatial data support added to Entity Framework in .NET 4.5. Finally, I'll show you how to fix up some problems with mapping to database views that the Wizard could only take a stab at, but needs some help to get right. Since this video is all about working with the designer, I'll stick with the designer and not bother with any more PowerPoint slides. You're welcome.

o Creating and inspecting the model and it's generated code

I'll start with a new Class Library project to host my model. I generally create separate projects for each model in my solution and this also allows me to keep my modeling logic separate from other logic in my application. I won't need this class file, so, I'll go ahead and delete that and then I'll add my model. It's easier to find it if I filter down to the data items and there it is on the top. I'll call mine AW for Adventure Works AWModel. EDMX is the name of the file that will contain the model. Because I'm doing a database first work flow, I'll generate my model from an existing database. And I'm pointing to a local DB database that I've already setup in my solution. When it's time to code, we'll be spending a lot of time working with this Entities class that is generated from the model. So, I like to keep the name of that a little simpler. I'll just rename that AWEntities. Now it's time to choose what's going to go in the model and I can choose from all the objects in the database. Well, almost all of them. I can select from Tables, from Views or from Stored Procedures and Functions. The Functions I have available to me will be any user-defined functions or table value functions that are defined in the database. The first thing I want to point out here, is that the Wizard is grouping my objects by their schema names, so before we look at that, I just want to flip over to the database, so you can see what that is about. In the database, I have a custom schema that I've created, called AWLT. And my important tables are all tied to that schema. I've left tables that don't really have to do with the actual business domain inside of the DBO Schema and If I expand views, you'll see that my views are all tied to AWLT schema and if I go into Programability, it's the same with my Stored Procedures. And again, I have a few that aren't really about the domain, so I left them with DBO and my Table-Valued functions are AWLT and I also have a Scalar-Valued function tied to AWLT. So, now back in my Entity Data Model Wizard, you can see that if I expand AWLT, that will list all of my tables and these are really the ones I'm interested, so I'll go ahead and select that and that selects all of those tables. I have no need for the DBO tables in my model, so I'll go ahead and select all the views and I'm going to select all but a few of the stored procedures and Functions, so you can see the stored procedures and Functions are collected in this common area. I'm going to select all of them and then unselect DeleteCustomer, InsertCustomer, and UpdateCustomer. I'll get back to those shortly. So, now I've selected all of the objects that I want to be pulled into my model. Before I move on, let's focus on the rest of what is in this Wizard. I have three checkboxes and by default, they are all checked. It's most common to want these options. The first is about how the names are pluralized or singularized, if the Wizard finds database tables that have plural names, it will make sure that each entity has a name that is just singular, because the entity represents a type, not a collection of things. In the background, Entity Framework has something called a pluralization service and the pluralization service is pretty good, but is based in English, so you may find that after you've generated the model you might want to change some of the names. The next option, is include foreign key columns in the model. Again, this is the best default behavior. If you've got relationships defined in your database and there is primary key, foreign key constraints that are defined, Entity Framework will recognize these and it will build Relationships. There are two different ways to build relationships, the most direct and the simplest to work with when you are creating your applications is using the Foreign Key Columns. The last refers to how the Wizard will handle the stored procedures and functions. What it will do with this option selected, is not only make the model aware of them, it will set things up so that you can use things directly in code right away. Before Visual Studio 2012, we had a similar ability, the stored procedures and functions would come into the model, but we had to select them one by one and generate this extra functionality in the model. So now with this option, the Wizard will go ahead and perform that action for all the stored procedures and functions we have brought in. I'll leave the Model Namespace at this Default, which is just the database name plus the word model and I'll go ahead and click Finish. Now, a lot is happening, but it happened quickly because I didn't have an enormous amount of tables to pull into my model. The first thing I want to point out is that I'm seeing messages and warnings based on this model generation in the error list. The messages and warnings are the same. It's nothing to be concerned about. It is something that you should have an understanding of. Entity Framework depends a lot on something called an Entity Key in each one of the entities. It's kind of like a primary Key in your database and it uses this Entity key to track your object instances as you are working with the application and it's a rule that every entity must have an Entity Key, so typically your tables in your database all have primary keys defined. So, Entity Framework is able to figure that out. But database views don't have primary keys, so what the Wizard has done is infer a primary key composed of all the nonnullable fields in the table and then match that with a composite entity key made from all the same properties in the entity and the message is just telling you that it did this for the four different views. There's a good chance you will want to modify those keys in the model. I'll get back to that task later in this video. So, besides the warnings and errors, there was a lot that happened here and let's just focus on one thing at a time. The first and most obvious thing is that a model was generated, which you can see in this diagram, so these are all of the objects that got brought in. You can see already there are lines between some of these objects. These are defining relationships that Entity Framework discovered based on primary key/foreign key constraints that are already defined in the database. There are some things that don't seem to have relationships with anything, most likely these will be entities that are mapping to the views, so I've moved things around to make it a little easier to look at what's in my model, but already you're seeing one of the problems with having a whole model in a screenful. Now, granted, I have the resolution on this monitor a little lower than usual and that's just for the sake of making a recording that you can actually see it well. So, typically I would I would have a higher resolution and be able to see a lot more here, but still imagine if we had a ton of entities in here, it really does make it hard to get around. So, one of the features that we have in Visual Studio 2012 Designer is that we can break these up, we can take a single model, but break it up into a variety of diagrams. Another new feature in the designer is that we can use color to differentiate the entities and I'll show you how to do that shortly. So, I entities in my designer that represent all of the tables and all of the views that I selected to be in the model, what I don't have here, is a representation of the functions and stored procedures. They are not part of the this visual model, the conceptual model that represents the objects that I am going work with as I'm using Entity Framework. So, this diagram is only giving me a partial view of everything that is in the model. The way to see everything is using this Model Browser. So, here's the Model Browser that represents my entire model. Now, all of my Entity Types are listed here. The place I can see the stored procedures and functions is inside of the Function Imports. So, these become functions in my application, the functions that are executed. The functions will return results and what the Wizard did was for each one of these functions that were generated from my stored procedures and my table value functions, it created something called a Complex Type. These Complex Types are not true entities in my model. They are special types. They have a special representation in the model. It makes it possible for me to use these functions that were created and return results that are a known type. The only thing we are not seeing so far in the Model Browser is the Scalar-valued function that I pulled in. A Scalar-valued function is something that I'll end up using, for example, in queries. So it is represented differently, we will be able to use it in our code, but we won't see it in the Model Browser. So, we looked at the messages and warnings that we got from the model. We looked at what's on the model designer surface. We've looked at the Model Browser to see additional objects that aren't on the surface of the diagram and there's one other important thing that happened as a result of creating this model that you need to see and that's related to our classes. First, I want to point out that I now have a reference to Entity Framework in my project. This is a reference to the NuGet package for Entity Framework 5. So as part of this process, the Wizard added a reference to Entity Framework. The next thing I want to focus on now is the model in this project. The model that we are seeing on the design surface is a visual representation of an XML file and that XML is physically inside of this EDMX file. The XML file just describes the model. It isn't what we work with in our applications. What we work with are classes that are generated directly from this model. Entity Framework uses the T4 Code Generator, which stands for Text Template TransformationToolkit and those TT files you see are T4 template files. Let me collapse this so it's not as overwhelming. So attached to the EDMX, I have a T4 template for generating a context and I have a T4 template for generating model classes. Let's first look at the model classes. So, you can see for each of the entities that I have in the model, I have a generated class. So, there is a customer entity in my model and I have a customer class that has been generated from that. And here is that class that the Code Generator created based on what it learned about my customer class from the EDMX. It's a very simple class. This is referred to as POCO, plain old CLR objects. It doesn't have any dependencies on the Entity Framework API or any other external APIs. I simply have a list of properties that match the properties described in the model and not only do I have the Scalar properties, but I also have these navigation properties for customer addresses and sales order headers. Let's flip back over to the model so you can see what that is all about. If we look at the customer, we can see customer has a relationship to customer addresses, a one-to-many relationship, and it has a one-to- many relationship to sales order headers. So, the Wizard by default created properties that allow us to navigate directly from the customer to those different types and in code what that ends up representing is customer addresses will represent a collection of customer address types related to a particular customer. Sales order headers will represent a collection of sales order header types that are related to a particular customer. There are a few other things about this generated code that I would like to point out. The first is the fact that my navigation properties are ICollection types and are marked virtual. Both of these are a setup for you to be able to use Lazy Loading in your application if you want. Lazy Loading lets you retrieve related entities, for an instance on the fly without extra coding. In order for Lazy Loading to work though, it has to be enabled on the context instance and then it will work with any navigations that are both ICollection and marked virtual. The generated code prepares your class so that if you do enable Lazy Loading on a context instance, then the rest of the plumbing is already in place. Another benefit of the ICollection is that you have a very generic way of declaring the set and can return the property into any number of types that implement ICollection. And then the last bit of code in here that was generated is just during the constructor to make sure that those navigation properties have been instantiated. That's not special to Entity Framework, it's just that if you try to do something with those navigation properties before they've been instantiated, you'll get an exception. So, it just avoids that. I also want to point out we got generated classes from those complex types that represent the results of this functions and stored procedures. Here's a generated class for the result ufnGetAllCategroies. Now, here's one of the first things I do after I've generated a model, the Wizard uses defaults to name the functions and types that it generates from the stored procedures and functions that I chose and it does the best job it can, but the results aren't usually anything you'd really want to use in code, so I go in and fix those up. I'll flip back over to the Model Browser and take a look at those complex types that were created to capture the results of my stored procedures and functions. I don't like these default names. For example, the type that was created to capture the get all categories function was named ufnGetAllCategories results, so that is the pattern that it uses, but that's not a real type name that I'd be wanting to use in my application. So, I am going to rename that to Category Trimmed because this has less properties in it than my full blown product category type. Now, if I go back to the function and I edit that function, we can see this actually is something you can do manually and before Visual Studio 2012, we had to create each one of these manually, but the Wizard did this job for us. It created a default name, created the complex type for me based on inspecting what the results of that function are and named that type. So, you can see that my complex type rename is showing up here, so the function already knows that it's going to be returning category trimmed types instead of that other one. I can also change the function import name. That's also an important thing for me to do. The database is using a naming convention for its function, but that naming convention doesn't really apply to my application. So, I'm going to name it Get All Categories Trimmed so that it's a little more obvious and then it's going to return the category trimmed. These are some of the kind of changes that I would be making to the model that was generated for me. Now let's return to the Solution Explorer and the generated classes. Notice that the type name didn't change in the generated class, but once I save the model it forces the code to regenerate and there is the new class name.

o Model to Database Mappings

Before returning to the designer surface to show you how to organize and colorize entities I want to be sure you are familiar with the table mappings. What you've seen so far in the designer is referred to as the "conceptual model." It's what you'll work within your application, but that XML I showed you earlier contains more than just the XML that represents this model. It also contains an XML representation of the database schema, which is in this storage model section and another set of XML that tells Entity Framework how to get from your entities and their properties to the database tables and columns. This is called the mapping section. The mappings are what Entity Framework uses for executing queries, returning query results, and performing updates to the database. The whole kitten caboodle of the conceptual model, database model, and mappings is referred to the model metadata. But, you don't have to blind yourself with XML in order to see how that's setup. There is a Table Mapping Viewer in the designer. One way to get to it is by right clicking any entity in the designer and choosing table mapping. Here you can see what database table the entity maps to and each column that maps to a property in that entity. Because I reversed engineer directly from the database and I haven't done any customizations, the mappings are really straight forward. Here, the entity is a direct reflection of the database table, but that's just the default. There is so much you can do to customize the model and change how it maps to the database. You can make a change as simple as changing the name of a property in the entity and you can see right away the mapping details updated. The designer understands that the property still maps to the same column even though I changed the name of the property. Don't miss this new designer feature that we got in Visual Studio 2012. You can reorder properties to your liking. We used to have to do this in the XML, which was not so much fun. You can't drag properties around. You'll have to use the context menu, just right click a property and you'll find this new Move Properties section, which has a number of ways you can move a property. I'd rather have the last name be above first name in my customer entity. So, I'll just move it up. You can use undo to revert these changes if you need to. There are a slew of other things you can do from creating inheritance hierarchies to merging tables into a single entity. There is a full 2 hour course on customizing your model in the designer that I did for Visual Studio 2010 and Entity Framework 4. While there've been improvements to the designer in Visual Studio 2012, most of the customization features remain the same, so I'll point you to that course to learn the ins and outs of customizing the model once you've created it. That way I can focus on the new features of the designer here.

o Adding Color to Model Entities in the Designer

A new property for entities on the design surface is fill color. By default, all the entities are this blue color, so the fill color property is a typical Visual Studio color selector custom web system. The only thing you can't do is create your own color, but there's plenty here to choose from, so hopefully they will be sufficient. So, you can just select a color. That will change the color of the entity, it's pretty obvious, and then you can also select a number of entities at one time and change their color altogether. If you were to change it to a lighter color, then notice the foreground text, which is currently white, will turn to black so you'll have contrast and you can read it. Although, be warned that that foreground setting isn't true for all colors. Here I selected yellow and it doesn't change to black. Its back to white, so it's not really legible. So, here I've provided some visual organization to my model using colors on the entities. Now I can easily look at this and my eye will be able to see all of the entities that are related to products, the entities that are related specifically to a sales order, the entities related to customers and their addresses. Down on the bottom, I've made the views all in olive color so I can visually say "ah" those are the views and I can find them easily. This is a really big help, but again when you have a really, really big model, even the colors might not be enough to make it easier to interact with the model. This is where you can take advantage of the diagram feature.

o Using Diagrams to Visually Break up your Model in the Designer

Back to my AWModel, in this particular model, the colors actually are probably going to be enough to help make it easy enough for me to interact with this model, but I'm still going to use this model as my way of demonstrating how to use the diagrams to break things up visually. One thing to keep in mind before I start creating these various diagrams is that I'm not breaking up the model. The model is still all of these entities. So, what I'm doing is I'm just creating different Design-Time visual representations of the model. They are called diagrams not views, although in my mind, it makes sense to say it's just a different view of the model, but the problem is the word "view" has so many meanings in the data and database world. So, instead the term is diagram. So, you can already see in the Model Browser, that I do have this section called diagrams and this default diagram is the one that's represented. I do like to keep a single diagram that does represent the entire model. So, I would probably rename this to Complete and then I'll just remember to leave that around. There are a number of ways to create new diagrams. One is to start in the Model Browser and add a new diagram. Then you can start dragging the entities you want onto the designer surface. I'll create a new diagram by right clicking Diagrams and selecting Add New Diagram. Now I can drag entities from the Model Browser onto the surface. I want to create a diagram that lets me focus on customer information. So, I'll start with a Customer Entity. Notice that even though I have no related entities in this diagram, we are seeing the navigation properties for customer addresses and sales order headers. That's because those are part of the customer entity as the model sees them. It doesn't matter if they are part of this particular diagram. Now, I'll add customer addresses and you can see that the designer automatically represents the association between the two entities that is already defined in the model. Also, notice I'm getting the default blue color of the entities. The colors are specific to each diagram, so the designer doesn't say, "Oh, she made customer, and customer address entities red before, so they should always be red." I'd have to change that in this diagram if I wanted that color here as well, but I'll skip that for this demo. There's another way to add entities to the diagram. You can add entities that are related to ones already in the diagram. So, from an entities context menu, you can select Include Related. I'll use that to bring in any entities related to customer addresses and that just happens to be one entity, which is address. Notice in the Navigation Properties that address also has a relationship to Sales Order Header, but that wasn't pulled in too. That's because Include Related will only pull in entities that are directly related to the one you started with. I'll go ahead and explicitly pull in Sales Order Header even though I don't really want it in this diagram. I'm just going to use this to show you another feature, which is how to remove entities from a diagram. So, I don't really want this entity here. If I select it and hit the Delete key, I get a dialogue about deleting the entity from the model. I don't want to delete the entity from the model, just from the diagram. So, you need to look carefully at this dialogue, especially if you aren't familiar with it. It's been around since Visual Studio 2010's Entity Framework Designer and it's not saying delete from model, are you sure? It's saying, hey I'm about to delete this entity from the model, so while I'm at it do you also want me to delete the metadata that describes the database table that this entity maps to. So, if I said "yes," then the entity will go away from the model, not just the diagram, but the whole model and the metadata won't even know that the related table exists in the database. If I say "no," to the question, the entity will still be deleted from the model completely, but the metadata will still be aware of the database table. The real answer in this case is that you want to cancel this operation completely. I'm not deleting anything. I just want to remove this from the diagram. So, the fact that I'm seeing this dialogue is a red flag and I'm happy that it alerted me to the fact that hitting the Delete key was not I really wanted to do. I don't want to delete it. So, I'll go ahead and cancel and instead, I'll right click and choose the real option that I wanted, which is to remove the entity from the diagram. After doing that, I can look over at the Model Browser and you can see that sales order header entity is still there. So, this is what I want in my customer diagram and I'll go ahead and give it a proper name, (CustomerView). Another way to create a new diagram is from an existing one. Let's say I want to diagram that focuses on the order. I can select SalesOrderHeader and SalesOrderDetail and then from the Context menu, I can choose Move to New Diagram. The designer creates a new diagram and places the entities on it, but if we look back at the diagram we just came from, the entities are gone from the complete diagram. Move means move. It doesn't mean copy and paste. So, sometimes you'll want that behavior where you want it to remove it from one diagram and place it onto a new diagram. So, that's good, but in this case, it's not because I really want to keep my complete model intact. So, let's go ahead and undo that. I have to click Undo three times before I see the entities again. That's because I'm not really undoing one step, but three. When I did the move that actually entailed creating a new diagram, cutting the entities from the complete diagram, and then pasting them into the new one. So, with Undo I am literally undoing each of those three actions.

o Updating Your Model when the Database Changes

Now, I'd like to show you another feature of the Entity Framework Designer, updating a model from the database. I've made some changes to the database. I've added a column to the Customer table. In the SalesOrderShipping table, I changed the ship method from a randomly entered string to an int. So, we can select a method from the reference lists instead of typing something in and I added a brand new table. So, now I want to apply these changes to the model. In the designer surface, I can right click for the context menu and then choose Update Model from Database. The Add tab shows anything in the database that is not already represented in the metadata. I don't mean that it's not in the conceptual model, but rather in the section of XML that the Wizard created earlier to represent the database objects. So, any new tables, views, procedures or functions will be listed in here along with any that you didn't choose earlier like these three stored procedures. Also, if I had deleted an entity from the model and said "yes" to the question about removing the related store schema information, then that table will also show up in this list, so, we are seeing the new table I created, as well as the stored procedures that I didn't select before. If you've modified a table schema, it's not in the Add list. Instead, the Refresh page lists all the items that were selected earlier and the Wizard will automatically update those. This is where the changed ship method type will get picked up, as well as my new column in the customer table. If I'd removed objects from the database that the model is dependent on, they will be listed here on the Delete Pages. So, before I can go ahead and finish this up, I want to be sure to go back to Add and select my new table. I want that to be brought into my model. Now I can go ahead and click Finish. Right away I can see the Customer Entity has been updated to show the new column I added to the customer table. Since the Customer Entity is mapped to the table, then the designer just assumed that since it knows about a new column in the table, it should add that to the mapped entity. Now, I don't have the entity that maps to SalesOrderHeaderShipping on this particular diagram. So let me go back to the Model Bowser and open up the complete diagram and we'll look at that property. So, first of all the properties of the property still show that the type is string. Now, originally it was a string because in the database it was originally a end varchar, but I changed it in the database. This is one thing that updating the model from the database won't do. It won't alter or remove existing properties, so even though the type has changed in the source schema, the type for the property and the entity didn't get automatically updated because that could create a problem down the road in any code that's already dependent on this and a problem like that, you might not find until runtime, but if I try to validate this model, the problem will show up right away, which is this validation error here. This is telling me that the mapping is invalid because currently ship method, which is a string, is mapping to ship method in the store schema, which is an .int. So, let's look at the mappings here, the Table Mappings. You can see it didn't remove the mapping, but the mapping is now broken. Because Entity Framework is not able to coerce an .int to a string or vice versa. So, the way the designer handles this, it kind of ensures that you are going to see the problem before runtime. I'm seeing this problem right away and I can deal with the problem right away. What I'll do is change the ship method to be an int. If I already have existing code that is dependent on that, I'm going to have to go through and fix up that code. That's not a small shift in domain logic to go from, "I want to just type a random string in here to I want to have a drop down list." So, it's no surprise that there is going to be a little more work to do in your application. But now if I validate the model again, the validation is successful. It just says Validation Completed. Don't forget that there was also a new table in the database. The new table that I pulled into the model didn't get automatically added to an existing diagram, but it is part of the model and you can see it in the Model Browser, but because I want the complete diagram to represent everything that's in the model, I'll go ahead and drag and drop this onto the diagram. Now, this table doesn't have any relationships to the other table. It's not even important what's in there, I just wanted you to see that process.

o More on Stored Procedure Modeling

When I originally selected database objects for the model, I skipped some of the stored procedures and said I would get back to them in a bit. You've seen that the stored procedures I did select were transformed into methods that I can call from my DBContext class and for those that return results, Complex Types were automatically generated also. I want to talk more about the stored procedures I already pulled and then return to those that I skipped at the time. We've already looked a bit at the functions and complex types that were created from the stored procedures and functions I brought in earlier. It just so happened that all of the procedures were ones, which returned results. But you can also import procedures that just do something in the database and don't return anything. In that case, you'll get the function and the method in the context class, but there won't be a complex type created. Another thing I wanted to be sure you saw about these functions is that most of these procedures have parameters. For example, this annual customer sales procedure has a parameter for fiscal year. If we look at the function in the Model Browser, we can see that it too has a parameter and the method that was created by the Code Generator also has a parameter. So, the designer and the Code Generator ensure that the parameter that the procedure needs is collected and passed along when the procedure is executed. You can also have multiple parameters, as well as output parameters. Another notable point about how the Wizard handled the procedures and functions is that by default, they will always created a complex type for the results, but it may not always be necessary and it's up to you to correct this. For example, if you look at the Order Details Procedure, it returns all of the columns from Sales Order Detail. Because I haven't customized the Sales Order Detail entity, it too maps to all of the columns of the same table. That means the Order Details Procedure will return a type that matches our entity. So, rather than having a new complex type, I can edit the function mapping to indicate that the result is the entity type and then I can remove the complex type completely. So, I can do that by right clicking on the order details function import and opening up the edit Wizard and instead of returning a collection of order details result, I'll go ahead and select the Sales Order Detail entity and then click OK to save it. Once I save the model itself, the Entity Framework Designer forces the code to be regenerated and you can see that the order details method now returns object result of Sales Order Detail, not object result of Order Details result. Another benefit of returning an entity rather than a complex type is the entity will get change tracked by Entity Framework, whereas a complex type doesn't. So, now I can use this procedure to return editable data. So, now to the procedures I skipped earlier. The designer supports another use of stored procedures where we can use those procedures whenever we need to update, insert or delete entities in the database. That will override the default behavior, which is that Entity Framework creates update, insert, and delete commands on the fly when it's time to save changes. The reason I didn't select them earlier was because I didn't want import functions to be created for these procedures along with the others. So, now I want to go pull them in with the Update Model from Database Wizard. Here are my other three stored procedures Delete Customer, Insert Customer, and Update Customer. The insert and update procedures involve all of the columns of the customer table in the database and again, because my customer entity still maps directly to that table, I know I'll be able to provide all of the correct parameters for these procedures. The delete just takes an ID and deletes the appropriate row from the database table. So, I'll go ahead and select all three of those and then the important thing that I want to do here is I want to deselect import selected stored procedures and functions. That is the feature of the Wizard that automatically created the import functions along with the complex types when they were necessary. So, for these I don't want import functions. I could have selected them originally and then just deleted the import functions that had been created, but I wanted you to see this feature as well. So I went the longer route. So, I'll go ahead and click Finish. If we open up Function Imports, you can see that I don't have new function imports related to the stored procedures that I just selected. As a matter of fact, the stored procedures are not represented anywhere in this model, but we can look at the storage part of the model. This is the part that represents the schema of the database and it's now aware of those three stored procedures. It didn't know about them before because we hadn't pulled them in. Now what I want to do is something called Stored Procedure Mapping. What I'll do with this mapping is I'm going to map the customer entity in the model to the stored procedures. So, I'm going to right click on here and choose Stored Procedure Mapping and you can see that this gives me the ability to map this customer to something that will perform the insert, something that will perform an update, and something that will perform the delete. I'll select the appropriate method for insert. You can see that the designer did its best job of matching up the parameter names to existing properties. Since we had changed the first name property to First, I have to give it a little assistance here. I'll say that goes to First. All of the parameters that are required for the insert are now mapped to the correct properties of the entity. What will happen, is if in my code I add a new customer and then I call Save Changes instead of Entity Framework creating on the fly a SQL command to do the insert, it will now use my stored procedure in the database and pass all the parameters into it. It overrides the default behavior. Keep in mind that later on in this course when I'm working against this stored procedure, I am going to make a tweak to the mapping against the insert customer command. So, I can do the same for Update and then Delete and Delete only has a Customer ID parameter. So that maps correctly. It just passes the customer ID and the stored procedure that has logic that will delete that row in the database. So, now for any changes to Customer that get sent up to the database, I can use my database stored procedures rather than SQL that Entity Framework would create. I should point out that I created that someNew Column in the database and that's now in customer entity as well. I created that column just to demonstrate the feature of updating the model from the database so it's not really involved in my stored procedures, which is why I don't have a parameter for someNew Column. There is a lot more to learn about working with stored procedures in the designer. Since that hasn't changed much since Entity Framework 4 and Visual Studio 2010, I'll point you to the Stored Procedure focused module of my Designer Supported EDM customization course to learn more about modeling with stored procedures.

o Using Enums with your Entity Data Model

One of the most requested features for Entity Framework that was finally added in the EF5 was the ability to use Enums. Without Enum support, we had to use workarounds, but now the model does support Enums. As I tried to make clear in the first module of this video, this support is only available with .NET 4.5. When EF6 is released it will be possible to have Enums with .NET 4 as well, but with EF5 to get Enum support you must combine it with .NET 4.5. Along with this, the Entity Framework Designer in Visual Studio 2012 allows you to use Enums in an EDMX. I'll show you how that works. Remember that I changed the Ship Method database column so now it's an integer in the database and in my model. I want to use that integer as an Enum, so I'll need to do two things. First, I'll define an Enum in the model and then I can set the type of the Ship Method property to the Enum. After I demonstrate this, I'll show application or even in .NET. To add a model-based Enum for use with an EDMX, you start by defining your Enum in the Model Browser. The designer surface is only for displaying entities and their associations. Visual Studio 2012 knows that I'm using .NET 4.5 and therefore it is giving me access to Enums in Model Browser. I can right click the Enum Types and select Add New Enum Type. I'll name my type Ship Method. Enum types can support ints or bytes, but since my property is already an int, then I'll just use the default of Int32. Now I can specify the Enums. My ship methods will be Land, Sea, Air, and of course, Star Trek Transporter. Although the values of the Enums are optional, I like to supply explicit values. That gives me the option of adding, editing or removing Enums in the future without getting the values out of sync. The Flags attribute lets you define bitwise Enums, which means that when defining an Enum property, you can combine the members. In other words, you can say that a ship method is by Land and by Sea. I'll show you what this other option is for shortly and then I'll go ahead and click OK. So, now you can see that my ship method is listed in my Enum types. And in the designer, the only way to see what the different members are is to go back and edit it again. I can do that by double clicking or right clicking and say edit. So, I'll just cancel that. Once I save the model and the Code Generator runs again, you can see a new file has been generated for ship method. And there is the Enum that was generated based on the Enum I defined in my model. Now that I've got my Enum defined and the model is aware of it and the designer is aware of it, I can go ahead and change the type of the ship method to my new Enum. Notice it is not alphabetical, but it gets placed at the bottom of the type list. So, now when I want to define a query using that ship method or I want to set or get the value of that, I can use the Enum and not integers. When Entity Framework creates commands for queries or for updating the database, it will automatically translate the Enum into the actual value 1, 2, 3, 4. Whatever it is. So, that is creating an Enum in the model. Now what I want to do is show you the alternate way, which is to let the model know about an Enum that exists already. Whether that is one that you created yourself or its part of another API or its part of the .NET framework. So, I'll start by deleting the Enum that I created. I have recreated the same Enum in code and I've given it a different name "AltShipMethod," so that you can really see that it's different than the one that I originally called ship method. So, this exists in code, but the model still needs to be aware of it because when we query and when we do updates, Entity Framework depends on that metadata in the model. We need to make the model aware of our external Enum. So, I'm back in the Model Browser and I'll go ahead and add a new Enum type again. This is important. My Enum type name has to match the Enum that is defined in the class. It doesn't matter what the name space is, but the actual name of the Enum. So, I'll call this one AltShipMethod. Now, I don't have to specify the members because they already exist and then I will click Reference external type and here I have to put in a strongly typed name and I'll go ahead and click OK. I'll save this. Now I can go back to the properties for the ship method property and choose AdventureWorksSuperLTModelAltShipMethod. So, I'm not selecting the Enum type that I defined myself in the class, I'm selecting the Enum type I defined in the model and then Entity Framework will figure out in the background that that actually points to the code-based Enum type that I created. So, this doesn't have to be my own definition. I could have used something that is part of another API or the .NET framework. But, you can also use your own Enum. So, that is what I have defined here. Now, in a later module in this course, I'll show you how to use these things in code. In this particular module, I'm just focused on showing you how to set all of this stuff up. The last thing I want to do with respect to the Enum types is to look at the generated code again. Since the Enum already exists in code, it didn't get regenerated by the Code Generator. However, I do want to show you that the Sales Order Shipping Class is aware that this ship method is based on that Enum. It is nullable because it was originally was nullable when it was a string when I converted it to an int, I left it as nullable. So, the nullability here is really based on my own configuration, not something that happened that happened just because I'm using an Enum.

o Using Spatial (Geography or Geometry) in your Model

Another new Entity Framework modeling feature that was added in .NET 4.5, was support for spatial data. That is geography and geometry data types for mapping applications. SQL Server introduced geography and geometry data types in SQL Server in 2008, but because the Entity Data Model didn't support those types directly we had to use some workarounds to leverage that data. Now Entity Framework 5 does support these data types and the Entity Framework Designer does as well. If you are working with other databases that have spatial data, you'll need to confirm that the relevant database provider includes support for Entity Framework's use of spatial data. As far as database first modeling goes, getting this new type into your model is pretty simple. In the database I have already added a new geography column to the address table. I'll update my model and walla, here's the new property in my address entity and it's a geography type. Now the type is comprehended for queries, reading and writing property values, and updating data back to the database without having to use workarounds to emulate the data type when working with address entity and looking at the address class, after the code has been regenerated, you can see that this property aligns with System.Data.Spatial.DBGeography. So, in our code we'll be working with the DBGeography class, but then Entity Framework will understand how to move this .NET type through the metadata and to the database and back. Later in this course, I'll show you how to work with this type in your application code.

o Fixing Up the Keys of Entities that Map to Views

When I originally created this model, I pointed out the messages about the keys created for entities that mapped to a database view. Before wrapping up, I want to spend a little more time with that because I want you to see how to make sure that the keys are properly defined. Recall that every entity must have a key in order for Entity Framework to do change tracking, build relationships, and merge query results. The Wizard uses a database table's primary key to define the entities entity key. For example, the customer table's primary key is CustomerID, so the Wizard marked the CustomerID property in the entity as its key. But, when you pull in a database view, since those don't have primary keys, the Wizard does its best job to infer a key. The pattern it uses is to compose a key based on all of the nonnullable properties. Now, let's go back and take a closer look. It's easy for me to find the views in my complete model because I colored them olive. You can see what properties are a part of each entity's key by the little key icon. In vGetAllCategory there is only one, ParentProductCategory name. But in vProductAndDescription, all five of them are marked as keys. Another possibility is that if the view doesn't have any nonnullable keys at all, the Wizard will just skip pulling it into the model because there is no way for it to create a key. You'll get a message about that when you create the model. You'll have the same problem with a table. If there is no primary key identified in the table, and there are nonnullable columns, the Wizard won't pull that table in either. And again, you'll get an error message. So, you'll be alerted to that problem. So, what I want to do is focus on just on just one of the views, vProductAndDescription. What this view does is combine some properties from the product table and some properties from the product description table. And then just creates a narrow view over those two tables. All the columns in the database view were nonnullable and therefore, all the properties in the entity are nonnullable. I happen to know that ProductID is really the one property that we can use to identify each separate instance of this particular type. I can either right click the property and uncheck Entity Key or I can go to the Properties page for that property and change its entity key value to false. So, I'll just do the last two here and save it. So, now this particular entity has a more logical entity key defined. I don't want to ignore what might cause possible confusion with the vGetAllCategory entity, so I'll take a look at that also. The Wizard marked ParentProductCategoryName as the entity key. But that's a nonnullable field, but what really ought to be an entity key is ProductCategoryID. So, why didn't that also get marked as a key? It is an integer and integers are nonullable by default. The Wizard didn't explicitly determine that this was nonnullable. You can see the nullable setting is not set to true or false, it is just set to none. So, it wasn't able to presume that this was nonnullable and therefore could be used as a key, but I can safely change this and mark that as not an entity key and mark this as an entity key and I'll go ahead and validate the model and it's valid. So, I can manually make that change. That wasn't a problem.

o Summary

In this video, you saw how to create a model from an existing database and learned quite a bit about what was actually created. We looked at the model metadata and learned how to use the new color and diagram features of Visual Studio 2012's Entity Framework Designer. You learned how to update the model when the database schema has changed and how to take advantage of existing stored procedures. You also saw how to define Enums in the designer or use ones that are defined elsewhere. Finally, you saw how to fix up the keys for entities that map to views. While this has been an in-depth look at database first modeling, it's not meant as a thorough look at all you can do to customize a model. Don't forget to go back to the other Entity Framework Modeling videos on Pluralsight that are referenced in this video. I'm Julie Lerman and thanks for watching this video on building database first models with Entity Framework 5 and Visual Studio 2012.

o References

Here are some more resources you might find helpful.

• Entity Framework 5: Model First Modeling

o Introduction

Hello. This is Julie Lerman. In this module of the Entity Framework 5 course you'll learn about creating an entity data model in the designer using the workflow refer to as Model First.

o Why Model First

In the first module of this course I talked about the different modeling workflows. Database first, Model First and code first. I want to start this model reminding you of what the Model First workflow is and why you'd want to choose that over database or code first. Model First is great for when you're starting a new project where the database doesn't even exist yet. And most importantly that you want to work in a visual designer. With Model First, you define your model in an Entity Framework designer then generate SQL which will create database schema to match your model and then you execute the SQL to create the schema in your database. That's what we'll do in this video.

o Outline

We'll start by creating an empty model and adding your first entity into the model. We'll explore the attributes of properties in the entities, setting different data types and how to control things like the length of the strings in your model and then the resulting database. Then we'll show you how to create an Enum on the fly when defining properties. I'll create some more entities and then define relationships between them including showing you how to build a many-to-many relationship. After I've designed small model we'll make some changes to the model's container and then generate the database schema while looking at some details of the metadata before and after running that tool. Your first step at the model is probably not one that's ready for production so I'll talk about what to do if your model or your database changes. Finally we'll look at how you can use the new Visual Studio 2012 designer features for coloring and diagrams when creating models with Model First.

o Creating a Model and your First Entity

I'll start by creating a new project that I'll use to host my model. This will be a Class Library project and then I'll add a new item and choose the data category just to make it easier to find the item template for ADO.Net Entity Data Model. Next I'll change the default name to SalesModel and then add it. The Wizard ask if I want to start with the database or with an empty model. From Model First you start with an empty model and then I'll click finish. Now I have a completely empty design surface. Also, notice that I've gotten new references in my project, Entity Framework and System.Data.Entity. The designer pulled in the Entity Framework 5 NuGet package for me, that's where I'm getting the Entity Framework reference. It's also added the code generation templates for context and from my classes. This will use the DB context that's in the Entity Framework assembly and also it will use simple POCO classes. We'll look at these after we've actually created some entities in the new model. One way to create new entities on the designer is to use the toolbox. You can just drag and drop entity onto the designer and it creates a default entity for you. I prefer to use the context menu to create entities instead of using the toolbox. So I can right click on the designer surface and then say add a new entity. The dialogue gets you started. My first entity will be Customer so I'll type that name in. The designer automatically creates a pluralized name for the entity set. The entity set is how Entity Framework groups entities for querying and for data management. Keep an eye on the entity set name though as it's being created. The pluralization service works best with the English but you'll find some edge cases even with English words where the pluralization isn't just right. So you can fix the name yourself if you need to. Every entity needs a key which is how Entity Framework tracks the entity. That's like a primary key in the database. So this little wizard starts you off with a property named ID, that's your key property. You can name it whatever you want. I'll change this to custom ID. It's also possible to compose a key from multiple properties but you'd need to do that outside of this wizard 'cause we don't have any properties yet. Also you can use GUIDs or Strings or other types for an entity key. I'll stick with the INT 32 default. So here's the starting point for my entity. Notice the little key glyph next to the CustomerId property. That just lets me know that the property is the key. Now, I can start adding more properties. I know I want a FirstName and a LastName. There are two ways to add properties. When is from the context menu for the entity. So I'm adding a scalar property and I'll call it FirstName. But default, properties are non-nullable strings. I'll probably want to be sure that the first and last properties are filled so the default is good for me. Entity Framework won't let me save a customer unless these properties are filled now. Since we'll be creating a database from this model this also means that the first and LastName columns in the data base table that we'll get created will also be non-nullable. The other way to add a property is just by hitting the insert key on your keyboard. That only works if your cursor is on another property in the entity. So, I'll use that way to create my LastName property.

o Properties and Attributes of Entity Properties

Just like FirstName, LastName is a non-nullable string and I'll leave it that way. So, now let's look at some of the other properties of the entity properties. I'll start with CustomerId. As I selected the CustomerId property you may have noticed that the property lists just shrink. I lost three properties and those are ones that are only relevant to strings since CustomerId is an int we don't see those in its properties window. I'll start with the most critical properties for CustomerId. You can see the entity key is true. I set that on a wizard when I created the entity, but you can set it here in the properties window if you want. Entity key is also in the context menu. So, if you prefer you can check or uncheck it from there as needed. This is how you can create composite keys if you want by setting entity key to true for additional properties. Store Generated pattern lets Entity Framework know if the store that is the database is in charge of setting the value of a column and therefore the value of the map to property. There are three options, Computed, Identity, and None. With computed or identity selected, Entity Framework won't bother trying to save the property value back to the database. Computed means that the database has formula that defines the value, maybe one that's dependent on other columns in the table. That's not something we can even use with Model First since you can't define a database column formula in your model. But if you're using database first, your database might already have columns define with formulas. So, you can let Entity Framework now about that. Identity tells Entity Framework that the property maps to an identity column that the database will increment. When we insert new rows Entity Framework will make sure to retrieve new identity and computed values after the database is created them and then apply them back to the in memory object that's being inserted. For an update the computed values will be pulled back from the database in conjunction with an update command.

o Controlling Strings in Entities and Database Tables

Now, let's get back to one of the string properties. The property list expanded again because of this facet section that's relevant to string properties. Facets are specifically for describing the databases columns that a property maps to. They're used when Entity Framework generates the DDL to create the database schema. It's really important to keep in mind that they have nothing to do with the actual conceptual model. For example, if you look at the customer class you can see that it's super simple. Now, we'll change FirstNames max length to 50 and save. Saving causes the code gen to run again. And now, back to the customer class you can see that that hasn't changed. There's no logic in here to avoid the FirstName being over 50 characters. Unless you add your own validation logic, if the user does enter more than 50 characters and saves to the database it will be the database that sees that the FirstName is too long and throws an error. You could build logic into the class using something like an IValidatableObject that would force Entity Framework to catch the problem when you call save changes. I've already created a course on using Entity Framework validation where you can learn more about that. By default Entity Framework assumes that strings map to variable link database fields like nvarchar. Setting the fixed length facet to true will change that behavior. Not only will the DDL specify a fix length field, but when inserting or updating data to the database Entity Framework won't have value to match the fixed length. Fixed length is either true or false. And if you don't specify a max length it will default in SQL server to an nchar whose length is 4, 000. Your database provider is responsible for default scheme of behavior. So, be default MicrosoftSQLServerProvider.system.data.sqlclient map string properties to nvarchar max database columns. You do have a max option available in max length facet as a drop down option, but if you don't set max length at all, that's the default anyway.

o Default Values, Spatial Data and Enums

Notice that default value is not in the facet section. This is specifically for your model and has nothing to do with the database. In other words this won't define a default value in your database column. I'll change the default value to Julie and save to rerun the code gen. This time the generated class did change. Notice that it sets the default value for FirstName in the class constructor. I'll add another entity to my model now, order, and as it create properties for it, we'll look at some other data types. My first property will be ordered date. I'll leave it a non-nullable, but change its type to DateTime. I've talked already about the fact that .Net 4.5 brought spatial data support to Entity Framework. This means you can map directly to the geography and geometry types in SQL server and for other databases, the effect will depend on the features of that database and your database provider. We can define our entity properties with these spatial types. When your project is targeting .Net 4.5, Visual Studio 2012 will let you define your entity properties using these spatial types. If your project is targeting .Net 4, you won't see the spatial types as a possible type for your property. The confusion with defining this in Model First is that you'll see a lot of different spatial types in the drop down, not just geography and geometry. Those are all there because the OData API supports a variety of types that are in the system.spatialnamespace. Entity Framework doesn't support this yet, but if you're building your model to be used for WCF data services then you can benefit from these granular types and OData will let you work with them. Otherwise, they'll just be treated the same as plain old geography and plain old geometry types. No matter which flavor of geography you choose you'll get a geography type in the database and the generated class will have a DB geography type. Another new feature brought to Entity Framework and .Net 4.5 is Enum support. You can go about creating Enums for Model First in a few ways. You can pre-define an Enum in the model browser or you can define the Enum on the fly as you're creating a property for the entity. I'll start with that way. I'll add a new property called OrderSource. The context menu for property let's you convert it to an Enum, but only when it's type that can be coerced to an Enum that would be either an Int16, Int32, Int64, a Byte, or an SByte. My property is a string so the convert to Enum option is isn't even available on the context menu. I'll change the type to Int32 and now you can see the "Convert to Enum" option. Clicking on this will be open up the Enum wizard. Here, I can create a name for the Enum and define the members. I'll call this Enum OrderSource also. I like to explicitly set values from my Enums so that I won't run into problems down the road if I want to add or remove members. In the database first module of this course I went into a lot more detail about creating Enums, how to use the plugs option to create bit wise Enums, and how to define an Enum in the model that points to an Enum that exist in code in another API. So, I won't go into these details here. In fact since I also showed how to create Enums using the model browser in that module, I won't even repeat that here. If you skip over the database first module of this course you might want to at least check the Enum portion of that video. Now that I finished up defining the Enum, notice that the property type is set to the OrderSource Enum. The Enum is now listed in the model browser and the code generator created an OrderSource file with the Enum code listing. Now that the Enum exist code I can use it in my application code whether I'm writing queries or setting values.

o Adding Relationships

I've added a few more entities to my model, LineItems for the orders and product so we know exactly what's being ordered and product category. It's time to create relationships between my entities. In the model relationships are called associations. As with the entities you define associations from the toolbox or from the context menu. I'll start by showing a tool box created association to define my one-to-many relationship between customer and order. I can select Association in the toolbox and then click on each end of the relationships. So, I'll start with customer and then I'll click on order. So, now you can see I have an association line between customer and order. And by default the designer assumes a relationship is one-to-many relationship. Now, I need to tweak this relationship which I can do in the properties window for this association. So, one-to-many happens to be exactly what I want here and that happened because I started by clicking on customer and then I click order as my second entity. So, that worked out just right, but all this did so far is defined the two ends of my relationship. I still have to define how that relationship works out and you do that using a referential constraint. The only problem with going this route by starting with the toolbox is that there's one more missing piece. I don't have a foreign key in my order that points back to the customer. So, I find it easier to use the wizard from the context menu and build associations that way. So, I'm going to delete this association and I'll start again. I'll right click on customer and say add new association. And by default it has presumed that the other entity association is order. Maybe that's just because it's the next thing that was defined in the model. I really don't know. That's totally random but I locked out. If it didn't pick the end that I was hoping for, all I have to do is drop down and choose the end that I want. So I'll leave at that order. And again by default, it chooses a one-to-many relationships so that happened to work out okay. When I used the tool box to create the association it also created navigation properties in each entity. So it gave me an order's property and customer and it gave me a customer property in orders and it's doing that by default here. If I don't want it to, I can just turn this off. Maybe I never want to go from customer to orders, but I will want to go from an order directly to its customer. In this case I'm going to leave them both there. Another benefit of going this route and using the Add Association dialogue is it gives me the opportunity to automatically create a foreign key in the dependent entity. Notice the check box says Add foreign key properties to the Order Entity. So I'm going to let it do that. I do want a foreign key property. With that foreign key property defined, I'll then be able to define the referential constraints between the customer and the order for this relationship and the Wizard will be able to figure that out for me. So I don't even have to manually do that. That will be taken care of. So let me go ahead and click Okay, and you can see the association, one-to-many association line between customer and order. And here is the order's navigation property and the customer entity and the customer navigation property and the order entity. And if we just expand this a little bit, you can see the new foreign key property that was added to order. I don't like the way the designer names that though so I'm going to go ahead and edit that and just call it CustomerId. So it was using the name of the entity and the name of the key that I was mapping to. Now let's go ahead and look at the properties for the association. Remember when I use the tool box, this referential constraint was empty and I didn't have the foreign key either. So I was going to have to create the foreign key before I could create this referential constraint. So the Wizard took care of this for me and you can see that the way it's defined it is that the principal entity is customer, the dependent is order. So it's like the parent and the child, thus constraint has to be between an entity key and the principal. So that's CustomerId that's in the customer entity and then on the other end it's saying, map this to the foreign key property CustomerId that's in the order entity. So you can see that's why I prefer using the Wizard in Model First to create associations with me because it just throws all that stuff in there for me. If I want it, the Wizard gives me a lot of options and also I can come into the Properties window for the association and make different kinds of changes if I want to also. I want to point out one other thing in this Properties window which is the Undelete properties. You can see one on End1 and one on End2. This is how Cascade deletes or controlled with the Entity Framework. So if I wanted to have a Cascade delete, that's said, if I delete a customer then delete any of the orders that exist for it, then I would change Undelete for the customer end of this relationship to Cascade. That means if I delete a customer then its related orders will also be deleted. It's really important to understand the difference between Cascade delete as it's understood by Entity Framework and Cascade delete as it's understood by the database. In Entity Framework, a Cascade delete will work with whatever objects you have in memory. So if there are orders in the database that you haven't loaded up, Entity Framework won't be deleting those in memory. Entity Framework can only delete the orders it's aware of, the ones that have been pulled into memory. However, because we're using Model First here, the presumption will be that we also want a Cascade delete in the database. By defining this new association property, Entity Framework will act on it when I delete a customer in memory, in my code, but additionally when I generate the DDL for the database scheme, it will assure that a Cascade delete is defined in the customer and order relationship in the database. You're getting both of them. It just so happens that the next two relationships I want to define are also one-to-many relationships between Order and LineItem and then from Product to LineItem. One product might be in many different LineItems.

o Many to Many Relationships

There's one more association I wanted to define and that's between Product and Category. In my domain, I want to be able to specify multiple categories for a product. For example if I'm selling clothes, I might have a product that's an item of clothing for children. So I might have a children category and I might have a clothing category. So I'd want to tag that with both of them. And then of course I might have many products within a particular category. So let me go ahead and define a many-to-many relationship between these two. So I'll add new association and it's going to be between category and product and this will be many-to-many. Now notice that Add foreignkeys has been disabled. We can't have a foreign key on either side because that would presume I'm pointing to a single thing. So for many-to-many relationships, Entity Framework is dependent on a join table in between the two related entities. The table will exist in the database but it doesn't exist in the model. In the model, it understands it's a many-to-many relationship. So we'll have a direct relationship between product and category. The database will have a join table in between the two and Entity Framework will understand how this relationship exists. When you execute queries or you return data or your building up a graph or related objects, Entity Framework will understand how to compose the queries based on the fact that there is a join table in the database and how to deal with updates and returning data as well. So, this is a nice benefit of having the model. As I'm doing my coding, I don't have to worry about the fact that there's a join table. I don't have to write join queries. All I have to worry about is that category has many products and product has many categories and I code based on that. Now, let's look at the association properties. Notice that there's no referential constraint here. Again, a referential constraint is really based on a primary key, foreign key relationship. This association will actually be defined by way of mappings, the mappings that describe how to get from the model to the database. So when I do my database generation Entity Framework will go ahead and create that mapping for me. So I don't have to worry about defining that. All I have to do here is say to many-to-many relationships and I'm good.

o Entity Container Properties

There are few more things I want to do and look at before I go ahead and generate the database and that's in relationship to the model itself. So not the individual entities but the model that there are all contained in. So I'm clicking just anywhere on the background of the diagram and I'll click properties and now I have the properties of the actual model not a particular entity. So I want to point out a few of the options here that you might want to be aware of. The first set is specific to co-generation. It has nothing to do with the database generation. That's just how the designer deals with the fact that we've got a specific template we're working with instead of the regular code gen. The next is lazy loading enabled is true and again this is for co-generation and what this will do is ensure that the definition of the context that we use when we're coding is set up so that lazy loading is enabled. I'll just leave that as is. Database script generation, these are defaults that at this time with Visual Studio 2012, there's really no easy way to change this defaults. With Visual Studio 2010, we had an additional tool that we could get from the Visual Studio gallery that would let us change the workflow and the DDL templates. What this is telling me is that the default workflow that it's going to be using when it creates the database schema for me is that any place where it sees inheritance defined to the model which I haven't done here but any place that it does see inheritance it will create the database using a table per type strategy. Meaning that for each entity in the hierarchy, it'll create a separate table. So say I had customer and that was BaseEntity and then I had good customers and bad customers. What that would do is create a table for customer and it'll create another table for good customer and another table for bad customer. And that's table per type and that's a default strategy. Moving down entity container access is public; I see that more as a co-generation option that the class that's created for the context is public in scope. Entity container name this is important. Currently the default is SalesModelContainer. I named the model SalesModel and so it names the container. That's the container that contains all of these entities. It just appended the word container to it. I don't happen to use that name in convention. I might call it SalesModel or SalesEntities or SalesContext. Because I've been doing so much work with code first, my instinct now is to name it context. The class set its generator from the container is what's going to give me all of my capabilities for interacting with the database and tracking the state of my objects as I'm working with them. So I do want that to be a name that makes sense to me and it's not really long and because I'm going to be using it a lot. It reminds me of something I skipped which is Database Schema Name. I just wanted to point out that this is dbo. When the database is generated, everything will be generated under the schema name dbo. So that's the default. You can absolutely control that to be whatever schema you want to be used for all the database objects that are going to be generated. Namespace is another property that will affect the generated code. All of my classes will be a Namespace called SalesModel. The rest of the properties aren't specifically related to a Model First so, I may eager to go ahead and generate this database so, let's go ahead and start looking at that piece of the puzzle.

o Generating the Database Schema

The first thing I want to do here is just to be sure, I'm going to right click on my design surface and validate the model to make sure that I don't get any errors and that's fine. Now, notice I do have some warnings. If I look at the warnings you can see that they all are related to the fact that my entities aren't mapped to anything yet and that's because so far all I have is my conceptual model. I don't have any database schema and I don't have any mappings yet. So that's perfectly fine and it's expected. And as a matter of fact if we take a look at the row XML of this model I think that would be useful to see. Just as a side note you can only view either the designer or the row XML. You can't have both of those open at the same time. So that's what that was about. So, notice that in the row XML, the SSDL section, that's the section that describes the database that basically empty because the database doesn't exists yet. The metadata doesn't know anything about the database. CSDL, that's my whole model that we've been looking at in the designer, the conceptual model. And then if we look at the mapping section again that's essentially empty also. There's no mapping because there's no database to map to. So that will change, once we generate the DDL. At the same time that the wizard creates the DDL at the schema for the database, now it knows what the schema of the databases so it's able to update the schema information and the mapping information. So, I'll open the model up again in the designer, and finally I'll go ahead and right click and choose Generate Database from Model. Now I believed I mentioned this before, this is kind of a misnomer because it's not really generating the database. The step is going to generate the DDL, or the SQL to go ahead and create the schema. So I'll go ahead and run that. It wants to know in advance what database I'm going to be generating this for, my database doesn't exists yet so I'll go ahead and say, new connection. I'm just going to use my default server and I'll name the database PluralsightSales. I'll click okay, and I'll get a message that says PluralsightSales doesn't exist here. Do you want me to go ahead and create, create it? So yeah, it will create it for me. So what this is doing is simply creating the shell of the database. It's not actually creating all the schema in the database but the database does need to exist in advance. Keep in mind that this particular step is not part of Model First. It's just the common Visual Studio data tool. Model First needs to point to an existing database so Visual Studio's tool is jumping in to help with that to make sure the database does exist. Then I'll get back to the Model First workflow for generating the schema for this database. So I'll go ahead and let that happen, now I have something that I can point to. I'll go ahead and click next. And here's the SQL that I can use to create the schema definition in the database I'd just created. Once I click finish here, a SQL file will be created in my project and the model metadata will be changed to include the database schema and the mappings between that and the conceptual model schema. I could look over the SQL in this window and if it doesn't look right, I can cancel and go back to make some more changes to my model then run this again. But I'm going to go ahead and click finish and we'll take a look at the SQL. The SQL files is the same as the DDL that we were using. It starts by pointing into an existing database even though I just happen to create it on the file. And then it will go ahead and create my customers table, and remember I set max length for my FirstName property to 50. I left the max length as the default for LastName, and therefore, the default there is nvarchar max. And in order, you can see you've got the datetime for OrderDate. I've got a geography type for destination NOT NULL. There's the order source property which in my database is going to be an int but in my application it understands it as an Enum. So Entity Framework who will take care of transforming the Enum value to the int as I go in and out of the database. Now I did talk about the fixed length property but I didn't actually select anything that's why we're still seeing nvarchars for the strings not just nchars. We also can see some of the foreign keys for example here in LineItems, I have a foreign key pointing back to the order table and I have a foreign key pointing to the product table. Here's the categories table and that just has CategoryId and name. And there's category product, there is the join table that provides the many-to-many relationship that I define between product and category. And then further down we can see where the database is going to setup all the primary keys that relate to my entity keys and also any foreign keys that I have defined in my relationships. So there's the SQL, my database schema is still empty there's none these objects exists in there. I'm going to have to execute the SQL. Before I do that though I want to go back over to the edmx file and show you that before I've actually run the SQL the metadata has been generated. So you can see now there's my SSDL section. This describes the schema of the database and there's my mapping section which describes how to get from the conceptual model to the schema of the database and back and forth. As I'm executing queries, as data is being returned from queries and as I'm doing updates to the database. So now let me go ahead and I will run this, I can just right click on the SQL, Visual Studio has the capabilities for doing this for executing SQL and doing some different database things in here. And if we go all the way down here on the menu, I can see execute, and I want to make sure I'm pointed to right server. And it's gone ahead and completed successfully. So the last piece of this puzzle is that we should go ahead and take a quick look at the database that was created. You can see that inside of my PluralsightSales database I now have all of those tables, customers, orders, LineItems, products, categories and CategoryProduct. And I've expanded customer so you can see I've got CustomerId it's a primary key, it's integer, it's not nullable. There's FirstName, it's an nvarchar 50, it's also non-nullable and LastName nvarchar max, also a non-nullable. So now you can see my model that I defined in the designer, as I understood my domain in the model, the properties that I needed, the relationships I needed between them, and there is the database that matches what I describe in my model. And Entity Framework with this metadata understands how to get back and forth between the two so, I can go ahead and use my conceptual model to do all of my coding and have Entity Framework take care of all the database work for me. Now I can also take the SQL that I used to create this database schema and bring it to the DBA when it's time to move to the production server and instead of, you know, giving me access through Visual Studio to do all that stuff. We're not going to do that. We'll make sure our professional is in charge with that.

o When the Model or Database Changes

One other thing people ask about when looking at Model First is what happens if my model changes? Or what happens if I decide to make a change in the database, or I give my DBA the SQL and they say, "Hey, you know, we should make this change in the database, you should do this, you should do that to, you know, make it better." If we change the model and rerun the wizard to create the SQL what would happen is by pointing to that PluralsightSales database this time it will see that all these tables and keys and index. That all these tables already existed, and all the keys existed. So you'd actually see SQL in these sections for drop existing constraints, drop existing tables. So that will get rid of everything and recreate everything. So you need to be conscientious of that especially if you started putting sample data into your database since of course we're not working against our production database. So what about the other way around? What if we've made changes to the database and we want those to be reflected in the model? At that point you can consider that you no longer using Model First, all right you've used your model to get a stake in the ground that your database. From then on what you can use is the same feature that I demonstrated in the database first video of this course which is that you can use the context menu to update the model from the database. So if you make changes to the database, now you're going to go from the database schema into the model. You can find that in the database first module of this course.

o Using Color and Diagrams to Visually Separate your Model

There are two new designer features that I already discussed in the previous database first module, coloring entities and using diagrams. While our current model is pretty small. Both of these features help you when you're working with larger models that can become more difficult to manage on the designer surface. I won't dwell on the entity colors again, but at least be aware that you can change in entity color and its properties window. Diagrams are interesting with Model First with Database First if you have reverse engineered a sizable database into a model it make sense, it might want to break the model visually into different diagrams. With Model First you can also take advantage of it. Let say for example we want to track contact information for a customer but we want to do that in unique entities. I could just add those entities to the model in this default diagram or I could do it in another diagram so I can focus on sales in this one and more on the customer in the other. I can use a common entity in multiple diagrams. So I'll start by opening the model browser and giving this diagram a better name. Now I'll add a new diagram and that's something you can only do in the model browser. All the entities in my model are listed in the model browser regardless of which ones are on which diagrams. I want to see the customer in my new diagram so I can drag it right on to the surface. Now I create an entity for addresses and one for contact info. Notice that I've also created an Enum for Address Type. A customer can have multiple addresses including billing and shipping maybe even a primary shipping address and a secondary one. I only have a single contact record for each customer. That's why I have a one to zero, or one relationship between those two. Now I have two diagrams, my entities are spread across the diagrams, and I have customers showing on both of them but if you look at the generated code, customers only there once. The diagrams are not what drives the code generation or the database schema generation. There are only views of the model. It's the model itself that the designer uses to create the code and the database schema. And you can always see the full model by looking in the model browser. So you can benefit from the entity coloring and the diagrams whether you're using the designer for Database First or Model First modeling.

o Summary

In this video you learned how to create models from scratch in the Entity Framework designer using the Model First workflow. I created an empty model and then added entities to the model one by one. We looked at controlling the attributes of the entity properties and the difference between Facets used to control the generated database schema and properties that affect the model itself. I showed you how to define associations between entities and why I prefer to do that with the context menu not from the toolbox. You've seen how to define Enums for entities as you create properties and how to handle model or database changes. Finally, you learned about the coloring and the diagram features of the designer and how you can benefit from those when designing a model in the Entity Framework designer. This is Julie Lerman and thanks for watching this video on creating entity data models using the Model First Workflow in Visual studio 2012.

o References

Here's some more resources that you might find helpful.

• Entity Framework 5: Code First Modeling

o Introduction

Hello. This is Julie Lerman, and welcome to the fourth module of my Getting Started with Entity Framework 5 course. This module is Code First modeling. In the previous two modules, you saw me build models using the Entity Framework Designer. First, I reverse engineered an existing database into a model, and then I created a model from scratch in the Designer and created a database schema from that model. In both cases, I let the Designer trigger Code Generation to build classes from the model on the designer. While some developers prefer to work with the Designer in Code Generation, others would rather just work with their code, no Designer, no Code Gen. For those developers, Entity Framework has a third modeling workflow referred to as Code First.

o Why Code First?

Code First is really made up of a set of puzzle pieces. First are your domain classes. The domain classes have nothing to do with Entity Framework. They're just the items of your business domain. So if you run a brewery, you'd have objects like recipe, beer, employee, supplier, inventory, things like that. Okay, I know nothing about brewing beer. I just like to drink it. Entity Framework, then, has a context that manages the interaction between those classes and your database. The context is not specific to Code First. It's an Entity Framework feature. What Code First adds is a model builder that inspects your classes that the context is managing, and then uses a set of rules or conventions to determine how those classes and the relationships describe a model, and how that model should map to your database. All of this happens at runtime. You'll never see this model, it's just in memory. Code First also has the ability to use that model to create a database if you wanted to. It can also update the database if the model changes using a feature called Code First Migrations. Finally, for the many possible scenarios where Code First might miss the boat when inferring the model or database based on convention, you can add configurations to tweak how the model and database are defined. Let's walk through this starting with some classes and then see how we can get a model and database created then I'll start tweaking things to meet my needs.

o Outline

In this video, I'll show you the basics of using Code First to leverage existing classes to describe model for Entity Framework to use at runtime, and how to let Code First create a database for you if you want to use that feature. Also, show you how to handle changes to the model which will affect the database. You'll learn about adding configurations using annotations and also fluently in the DataLayer. And I'll show you a tool that will allow you to reverse engineer an existing database into classes that can be used with Code First modeling workflow. Not a lot has changed since I did the in-depth Code First course for Entity Framework 4.1 or the course on Code First Migrations. So this particular video will be designed to get you started modeling with Code First and then you can take advantage of the more in-depth content of the earlier courses to learn how to benefit from the various features of Code First and Migrations.

o The Domain Classes

Here's my project with a handful of domain classes and since I'm obviously not qualified to brew beer, I thought I would stick with a simple sales domain. So the customers and orders and line items, and things like that. My classes are pretty simple. So here's the address class. Address has a navigation property customer, so I have a relationship with the customer as well as the CustomerId foreign key. Also notice, I have an address type property which uses the enum type address type, which you can see the enum up there. So this is new for Entity Framework 5 and .NET 4.5 that we can use enums. And with Code First, it's just no different than writing any other .NET code. Here's my simply defined enum with three members in it. And then I can just use it in my address class. So you don't have to do anything special with your enums in order to use them in Code First with Entity Framework, which is really nice. So here's customer and customer has navigation properties for orders and addresses. And that's a .NET pattern, not something that's specific to Entity Framework. I'm instantiating those in the constructor of the customer so that the first time I go to add something to my orders, I don't get an exception that says, orders is null. So I'm instantiating those in advance. That's a good pattern to use. ( Pause ) Order does the same thing in instantiating its line items in advance, and it has a reference back to customer as well as a CustomerId foreign key. Now, it may seem redundant to you to have a foreign key and the navigation property, but not only is this my habit for creating classes, you'll find in the long run that it's lot easier to manage relationships Entity Framework when you have both the foreign key and the navigation property. You can certainly use just one or the other, but then you'll have to understand more about the underpinnings of how Entity Framework handles the relationships. In addition to yet another enum here which is the OrderSource enum, I also am taking advantage of something else that's new to Entity Framework 5 which is the ability to use spatial data or geography data as properties directly in my classes. So you can see I'm using the DbGeography class to define destination lat/long. So that'll be a point in space. I want to point out, then, in order to use the DbGeography type, I need a reference to .NET's system.data.entitydll in my project. This assembly contains the system.data.spatial namespace where the DB geography type resides. Although my domain classes have nothing to do with Entity Framework, I have to add this reference solely for this reason. Hopefully, in the future version of .NET, this type will be moved into system.data so that the reference for system.data.entity doesn't raise eyebrows about the claim that I'm using POCOs in this project.

o Creating a DbContext Data Layer

So these are my classes, they have nothing to do with Entity Framework. So how am I going to engage Code First Entity Framework? I'll do that by now creating a data layer that uses Entity Framework. And I'll point that data layer to the classes. It will comprehend the classes and be able to manage the classes with respect to Entity Framework features of getting data in and out of the database, managing change tracking, things like that. I've created a new project in my solution, and it's a class library project. And so far, the only thing I've got in here is a new empty class called SalesModelContext. The first thing I want to do for this project is make sure it's aware of Entity Framework 5. And I could do this from the project, or I could do with solution Y. By doing with solution Y, then as I add other projects in here that need Entity Framework, I can just go to the solution and say, "Okay, now, use it here, use it here, use it here." So I'll start with the solution and manage NuGet packages for this solution, for the entire solution. Go online and look for Entity Framework. And there it is, right at the top. If I wanted to play with the alphabets of Entity Framework 6 instead of using Entity Framework 5, I would use Include Prerelease, and then I would see the latest version of the Alpha or the Beta of the next version of Entity Framework. So I just want to point out, I'm definitely using stable only, and I'm using version 5 of Entity Framework. Because I'm managing packages for the entire solution, it's saying, I'll go ahead and put Entity Framework in all the projects. But I don't want it in the domain classes, I only want it in the data layer. Now, you can see that if the package is filed in my project, and that I've got the EntityFramework.dll as a reference in my project, NuGet will put that in there for me. And there's a new App.config file. If we look at that, you can see that it's got some default configuration. It's added a config section to make sure that the config file is aware of the Entity Framework API, and then an Entity Framework section where it specified the default connection factory, by default, it's going to use the local Db. Local Db is part of SQL Server 2012, and it's now the new default development database that you would use instead of SQL Express. So it gets installed along with Visual Studio 2012. If I didn't have this connection factory, Entity Framework would revert to its default behavior which is to use SQL Express by default. So this overwrites that. So now, by default, we're going to be using local Db, unless we otherwise specify different database, for example, in a connection string. Notice there's no connection string here, though. We can forget about App.config for a while. So now that I have the Entity Framework 5 referenced here, what I want to do is make sure that my class inherits from Entity Framework's DbContext. And notice that that's in the system.data.entity namespace. EntityFramework.dll extends the system.data.entity namespace that's also inside of the system.data.entitydll that's in .NET 4.5. Now, I have my SalesModelContext, it's going to understand how to do all those Entity Framework and Code First stuff because it inherits from DbContext. So what I want this context to do is manage my classes domain. The way it does that is in a type called DbSet. I'll be able to manage things through those DbSets. A DbSet of customers will allow Entity Framework to manage instances and querying and updating to the database of the customer objects Before I can do that, I need this data layer to be aware of my domain objects. So I'll go ahead and add that reference. Now it can go ahead and add my DbSets. So I've added DbSets for all of my classes except for one. I didn't know how to DbSet for ContactDetail. The reason is that these DbSets are what give me the ability to query one of this set. If I want to write and execute a query to go find customers, DbSet allows me to do that. Without a DbSet for ContactDetail, I won't be able to do an explicit query to go find ContactDetail records. But ContactDetails have a one to one or actually it's a one to zero or one relationship with customers. I know that I won't have a need to explicitly look for ContactDetails, I just want to get them along with customers. So because ContactDetail has relationship with customers, the context will be aware of it, the model that Code First pass will be aware of it and then I can take advantage of the relationship between customer and ContactDetail to pull in the ContactDetails whenever I query for customers or go get them after the fact. Another thing that I want to point out here is that I have a very small domain. So I've put all of my classes in a single context. So I'll use this one context throughout my entire application. If you have a large domain and you have lots and lots of classes, you'll most likely not be working with all of those classes in each area of your application. I created a course for Pluralsight called Entity Framework in the Enterprise which specifically deals with that scenario. The second module of that course is about building domain driven design bounded context with Code First and DbContext.

o Building, Debugging and Fixing the Code First Model with Configurations

A critical function of Entity Framework is its ability to interact with the database. In doing so, it can transform LINQ queries against your context and DbSets into database queries. It can materialize results from the database into class instances and it can Send, Insert, Update and Delete commands to the database to reflect changes made to those objects or new ones that have been created. Entity Framework relies on a set of metadata that describes your domain model, describes the database schema and also describes the mappings to move between the two. If you watch the previous videos in this course, I showed you the XML metadata of entity data models that stored in an EDMX file when working in the Designer. We don't have a Designer or an EDMX file when we're working with Code First. So how does entity framework get from this context to an entity data model when we're using Code First? It does so by building that model metadata on the fly at runtime. DbContext infers this metadata based on the classes that the context is managing and then further by the properties of and relationships between those classes. One of the difficulties of defining a model with Code First is that you have to kind of cross your fingers that all of that info that your classes and context provide do in fact build the model you intended and properly mapped to the database if it already exists. There's a way to relieve that guessing, though. You can use a tool that will visualize the model for you. This is one of the features of the entity framework power tools which you can install from the visual studio extension manager. I already have the tools installed in Visual Studio. The feature of this tool that I want to focus on is its ability to create or read only visual model from a context. What you can do, once the tools installed in Visuals Studio is right click on any class that inherits from DbContext, and from the Entity Framework menu item choose View Entity Data Model. There are couple of rules about making this work, and I'll show you those. The first one I'm going to break right now and I just want you to see what the errors look like so you understand. This is not going to be successful. So it's actually going to take a little longer than it would were it to be successful. I won't make you wait all that time now. When we get back to the regular IBM cursor, I am getting my visual model. You can see down in the bottom, there's a message that says, "An error occurred while trying to build the model. See the output window for details." So I'm going to look at the output window, if we get back up to the beginning of this error, we can see now an error occurred will get provider information from the database. This can be caused by Entity Framework using an incorrect connection string. Now I'm going to stop there, but this is why I wanted you to see the message. The message can be misleading. Remember I said earlier, Code First doesn't necessarily need a connection string but it does need some kind of information about building the connection. So the real problem here is that it's not even finding that information in the App.config file and that's because right now, to start a project in the solution is the domain classes project. And that doesn't even have the App.config file in it. At some point in development, you probably have a project and the solution that is actually the executing code and that's where your App.config would be and that would probably be your start up project. But right now, the only source, the Entity Framework for understanding anything about connecting to a database is inside of this App.config inside of the SalesModelDataLayer. So I'm just going to for the time being let that be the start up project. So that problem was just one that prevented the tool from even running. And I want to be clear that the tools I'm trying to connect to the database, mine doesn't even exist yet. It just needs to know the database provider when creating the metadata. But the tool could also help you validate your model. If there are problems that prevent the generated model from complying to the schema rules that Entity Framework needs, they'll be discovered and reported here. And I think this is a much better place to find and fix those problems than when debugging your app because by that point, you've got more puzzle pieces involved and you may not realize it's a modeling problem and waste a lot of time focusing on your code looking for where the problem is coming from. I have two more issues that I know will create a problem with the schema. So let's run the tool to get rid of them. We'll only find one at a time. So we'll have to grab the first one, fix it and then fund the next one and then fix that. The first one is based on an Entity Framework rule that every entity needs to have a key. Notice that this time, the error came up right away because the tool was struggling to find the connection information. Using convention, Code First is enabled to determine the entity key for the ContactDetail type. It has two conventions for finding the key. Look for a property named ID or look for a property that combines the type name and ID. There is no such property here. In my class, my key property is CustomerId because of my relationship with customer. Code First gives us two ways to add configuration details when convention is enabled to infer things properly. One is using annotations directly in the class. I'll do that now and a bit later, I'll show the other way to configure which is using the flow in API. The annotations are in a namespace called System.Componentmodel.DataAnnotations and also in a subname space called schema. I want the data annotation called Key that tells Code First which property is the key. So now let's run the tool again and once again, the validation fails because of one of the more confusing conventions in Code First. When you have a one to one or one to zero or one relationship, Code First needs to understand which is the principle and which is the dependent and of the relationship. That will determine which table in the database has a foreign key pointing to the other. You'll always need to add configuration information in this case. Relationships are definitely a point of confusion in modeling especially when they're aren't relationships that a convention can figure out. In this case, I have to specify the foreign key. But just saying this is the foreign key isn't enough. You need to indicate foreign key to what and the navigation property is how you do that. I can say that CustomerId is a foreign key to the customer type but I don't do that by using the type as the parameter. Instead, I have to indicate the property that knows the type. In this case those names are the same, customer and customer. It would be more obvious that I'm using the property names of parameter the foreign key if the property name wasn't exactly the same as the type name. Notice that the foreign key annotation is in the Schema namespace. So one last time, I'll go ahead and generate the visual model and there it is, it worked out. Finally it was able to display the model and made sure that all the Entity Framework rules were being honored. Now the second part is to make sure that my model came out the way I intended it to be. And I don't like the way it's laid out the diagram so I'll go ahead and just move things around a little bit. ( Pause ) You don't want to put too much investment into making this a perfect and beautiful diagram 'cause it's a read only diagram and I'm just going to throw it away when I've gotten the information that I need out of it. And that information is just looking at it to make sure that all of my entities and the relationships are as I wanted them to be. The tricky is that relationship between customer and ContactDetail but that did work out as I wanted. It's a one to zero or a one relationship, that's just what I wanted it to be. So I can see my relationships between the entities, I see my navigation properties. I've got a many-to-many relationship where I want to be between category and product. I've got one-to-many relationships where I want them. So the model looks great. I was able to validate the model and also visually verify that the model worked out the way I wanted it to work. So I shouldn't have any surprises when it comes time to write my code against this model. I'm a huge fan of using that tool just for this purpose. I hope that something it will take advantage of this as well.

o Initializing a Database from the Model

Now we'll look at another of the important Code First features database initialization. Code First doesn't worry about the database until runtime. And it needs to do two things. The first is to locate the database and second is to create the database if necessary. Here's how that works. The first time a particular context needs to interact with the database at application runtime is when Code First will perform database initialization. What it does first is look for connection string using the same method any that app would use. If it can't find the connection string, then it infers one. By default, it will use the SQL express as the database. But if something else has been specified as you saw with the local Db information in our App.config file, then it will use that instead. Then it looks for a database with a name that matches the strongly typed name of the context. In the case of my context, that would be Salesmodel.Datalayer.SalesModelContext. Now, it can build a connection string in memory. Once it has a connection string, it will check to see if the database exists or not. This is another default behavior that can be changed. If the database doesn't exist, then it will create it and it uses the metadata that the model builder created to determine the schema of the database it's going to create. Once the database exist, then it can go ahead and do whatever task was being executed on the context in the code. So let's watch that default behavior in action. I created a little ConsoleApp in my solution. I made sure that the ConsoleApp also had a references to the Entity Framework NuGet package which created the App.config file for me which is the same as the App.config I had in the DataLayer project. Now this is gong to be my executing application. So I don't really need App.config in my data layer, so I've moved that out. When I started creating the code for this, I realized that I made a mistake I commonly make because of my old days as a VB programmer. I hadn't made my SalesModelContext class public. So I fix that before I was able to write up my code. So it just have one simple method in here, get customers, and what it does is it instantiates my context and then executes a simple query which is to get all the customers and return them as a list. Two list is a LINQ method and it's also an executing method, so it will cause my query context.Customers to actually execute go and hit the database and return customer objects. So far in this course, I've focused on building models whether it's Database First, Model First, or Code First. And we haven't really seen any executing code yet. In the next video in this course, we'll spend more time with actually using the models. So I'm using the model here just so that you can see the database initialization. So this is the point when Code First says, "Ah, I need to do something with the database. Let me go find the connection string and check to see if the database exists yet." So it's not when the context is initialized but it's the first time in an application process that a context actually gets used to hit the database, to do some kind of interaction with the database. So once I've executed the query and return the customers as a list, I iterate through the list and spit the names out in the Console. Now, of course, we don't have a database yet. There won't be any customers in there so there won't be any first name listed in the console. So go ahead and run that. And doesn't look like much happened. But if we open up SQL Server Object Explorer and look in the local Db which is where I know that this database is going to get created, there is my new SalesModel.DataLayer.SalesModelContext database that just got created by Code First when I executed that query. So expand that. There are all the tables that reflect the model. So this is Code First, right, it's the model of my code that's driving what the structure of the database is going to look like. So the database looks just like my model. So you can see, I've got the customers table and their orders and the line items and all those tables. And if we expand them, you can see that they have all the columns that match what my model inferred from the classes. So customer had a customer ID, that was the key in my class, and therefore, this column is the primary key in my database. First name and last name were strings and by default because I didn't add any other configurations. SQL Server renders those as nvarchar max. That's the default behavior and notice that they are nullable. If we expand ContactDetails, we'll see that that customer ID is indeed not only a primary key in this table but it's also a foreign key. And we drilled in, we would see that the constraint-- the primary key/foreign key constraint is built between ContactDetails and customers. There's orders and orders has its customer ID foreign key pointing back to the customer. And let's not forget the new features we got in Entity Framework 5 and .NET 4.5. There's my order source field which is-- where we'll store the other source enum values. And the geography type which was inferred directly from the Db geography type I used as property in the order class. And if we expanded all of these tables, you would see columns reflecting other properties in these classes. The one table that you may not have expected to see here is the category products table. I defined a many-to-many relationship between category and products in the way Entity Framework and the database work that out is to have a joined table in the actual database. So when I'm working in my application I'm writing code, I'm just doing with many-to-many relationship. But when it comes time to create the SQL for doing queries and updating or inserting records, Entity Framework will translates my many-to-many relationship into SQL that understands about the joined table and handles that appropriately.

o Using Migrations to Update Database Schema when the Model Changes

So what happens if I want to change something in my model? I've added a date of birth property to the customer type. If I were to run the application again, I get an exception and that's because the default behavior of the database initialization does not allow for changes to the model. There's a number of ways you can change how the database initialization works. One of them is instead of using initialization, tell Code First to use its feature that's called Migrations. And that's what this message is saying. The default behavior that we're hitting is that by default, Code First will only create the database if it does not yet exist. So what's happened is Entity Framework has created the metadata in memories in the model builder, and when it created the database before, it actually stored a hash of the model of the metadata. So Code First is just compared its current metadata to the metadata that was stored in the database and seeing that something is changed. So what I'll do here is I'll switch over to Migration. So I'm just going to do it in a simple way, because I have a whole course on Code First Migrations. And Migrations haven't really changed very much since that course, so there's no need to go into detail here. I'll start by enabling Migrations in my DataLayer project. You need to do this in the Package Manager Console window using the command Enable Migrations. This will jumpstart some set up code and other actions that I'll need for migrations to work. There are two ways to use migrations, automatic which is suitable for basic needs and one called Code Based where you have total control over how and when migrations are performed. You can even customize what's going to be done to the database. But I'll use automatic for this demo which means I need to append the enable automatic migrations parameter to the command. When it's completed, you can see that it created a new for me called configuration in a new folder called Migrations. The configuration class inherits a Db migrations configuration. And its constructor indicates that will be using the automatic migrations. Remember, I haven't yet able to run my apps since I added the new date of birth property to the customer type. Now that I have automatic migrations enabled let's run the app again and see what happens. But I'm getting the same error, okay. I knew that was going to happen. The database initialization is still using the default. I have to specifically tell Entity Framework to use migrations instead of the initialization. There are four initializers. The default is called CreateDatabaseIfNotExists. That's the behavior we're experiencing. Another is DropCreateDatabaseIfModelChanges. I could use that but if I had any data or other objects that I've added to my database, I'd lose all of that. DropCreateDatabaseAlways is a great option for automated integration testing. And finally, MigrateDatabaseToLatestVersion. That's the one we need to use for migrations. In addition to my migrations course, my DbContext API course has a module on database initialization if you want to dig deeper into that topic. You can change the initializer either in a startup code for your application. So here, that's in my Console's main module. If I were writing a web app, that would be in Global asax. You can also set the initializer in your application's config file. In Entity Framework 6, we'll have yet another way to specify initializers because we're going to have code-based configuration, so we wouldn't have to do that work in the config file. But I'll do the job in my startup code. I'm using Entity Framework's Database.SetInitializer method, specifying to use a MigrateDatabaseToLatestVersion initializer, and notice that I have to do that by creating an instance of that type. Then I have to indicate what context I want this initializer to work on. And finally, I have to specify the migrations configuration type to use. Also, I want to point out that by default, that class's scope is set to internal. So I needed to change it to public in order to use it in the SetInitializer code. Now, I when I run my app, no exceptions. I'll open up the schema of the customer's table and refresh it, and there is the new date of birth property. I want you to see a few more of the migrations features, so I'm going to mess around with a model a little. First, I'll remove the date of birth property from the customer class. And when I try to run the app again, migrations isn't happy because, by default, it protects you from removing columns that might have data in them. This is referred to as data loss. So I can change the configuration to override that by allowing data loss. And then when I run the app again, that works just fine. So now, I'm just going to add some data manually into the customer's table. And then I'll restore the property and run the app again. Because I'm restoring the property, that will force another migration again and you could already see from the Console window but we'll look in the database as well. You could see that even though I ran a migration, the data is still there. And because I added that new property, SQL Server provided a default value for it.

o Seeding a Database During a Migration

I want you to see a few more of the migrations features, so I'm going to mess around with a model a little. First, I'll remove the date of birth property from the customer class. And when I try to run the app again, migrations isn't happy because, by default, it protects you from removing columns that might have data in them. This is referred to as data loss. So I can change the configuration to override that by allowing data loss. And then when I run the app again, that works just fine. So now, I'm just going to add some data manually in to the customer's table, and I'll restore the property and run the app again. Because I'm restoring the property, that will force another migration again, and you could already see from the console window, but we'll look in the database as well. You could see that even though I ran a migration, the data is still there. And because I added that new property, SQL Server provided a default value for it. You can also create seed data when migrating the database using the override of the configuration seed method. If you're using one of the other initializers, there's also a way to seed databases after Code First creates them. You can see more about seeding during initialization in that database initialization video I've already mentioned. The AddOrUpdate method which is part of the migrations API, will use a specified property to check for already existing data before adding any new rows. But be aware that if a match is found, then Code First will replace the entire existing row. It won't just update one or two values. My seed method code creates two new customers and then uses the AddOrUpdate method to seed the database with them. Instead of checking for a single property as a sample code did, I'm comparing both the first and last name, and you can see that you need to use initializer code to pull that off. Also, notice that AddOrUpdate expects an array. So I took the list that I'd created and I had created it as a list just because that's what comes naturally, and converting it to an array to pass into AddOrUpdate. When you're using automatic migrations, there's something to keep in mind with respect to the migration seed method. Prior to Entity Framework 5, this method would only run during a migration which meant that I'd have to go and make a change to my model right now to get this seed to happen. And that's kind of a pain because I don't really want to make a change to my model. So now, with Entity Framework 5, the seed method is simply triggered by database initialization, even if it turns out that no migration is necessary. So that solves the earlier problem but it creates a new problem. The seed method is going to get run every time. I want it to run the first time I've made changes to it, but I don't want it to run every time I run my application. That's going to cause a lot of unnecessary database activity. So right now, there are few ways you can disable this without having to delete the seed code. And none of them are pretty. So let me go ahead and get that seed data in my database now. So I'll go ahead and run this. So I haven't made a change to the model, and you can see that I am hitting the code. So now, my seed data is going into the database. Now, that's done and I don't want that to happen again. One thing I could do is just disable the initializer. So I'll run this again and you'll see that I don't hit the breakpoint in the seed method. Another is to use a compiler directive. So when I don't want this to run or even get compiled into my application, I can set that to false. And if I make a change to it, so I do want it to run, I can set it to true. I'm hoping that in Entity Framework 6, we're just going to have another configuration that we can use during automatic migrations so we don't have to mess around with these hacks.

o Using Data Annotations to Configure Mappings

( Pause ) Earlier in this video, you saw me use some data annotations to fix up my model. I used a key data annotation to let Code First know that the CustomerId property of the ContactDetail class was the key property. And I also helped Code First better understand the nature of the relationship between ContactDetail and the customer class using the foreign key property. There are many ways to enhance a model with the configurations when Code First convention doesn't meet your needs. Data annotations are one way. These are simple to apply by just annotating your classes and their properties. There are other APIs that understand data annotations as well. For example, MVC can read them. In fact, there are a slew of data annotations that MVC and the Entity Framework will read for the purpose of validating your data. Here's a list of the data annotations you can append in your classes that are relevant to Entity Framework. There are more data annotations that you can use for other purposes. The interesting thing about the validation annotations is that Entity Framework will use the max portion of StringLength as well as the MaxLength and required attributes to help to find the schema of the database. For example, a property marked as required will map to a non-nullable column in the database. There's a lot more that you can do with validation in Entity Framework, and I have yet another course that's focused specifically on using Entity Framework validations. All of the annotations and the schema sub namespace are for Entity Framework and deal with mappings. For example, convention presumes that a database column name matches a property. You can use the column attribute to specify the column name when that's not the case. My earlier code for this course covers the data annotations in detail. Not just showing how to apply them, but you'll see the effect on the database schema as well. So I won't go into them further here. Keep in mind, though, that in Entity Framework 5, the schema namespace is new. That's actually a change that was made to .NET 4.5. So if you're using Entity Framework 5 with .NET 4, or an earlier version of Entity Framework, the schema annotations are in the System.ComponentModel.DataAnnotations namespace and live in the Entity Framework dll assembly. There was no schema namespace back then. Otherwise, there have really have been no other changes with respect to the data annotations for Entity Framework.

o Configuring Mappings with the Fluent API

You may not want to put your database schema mapping configurations in your domain classes and dirty them up. I hope I didn't sound too prejudice against data annotations there. Data annotations really are convenient and if you want to also benefit from something like MVC Client Side Validation, they're a great way to go. But there's an alternative way to configure the mappings to your model without putting this information which is about your persistence layer into your domain classes. That's the fluent API. With this method, you can write code for the model builder to use-- to apply configurations as it's building the model from your classes. That way, you keep the mappings out of your domain classes. There's another interesting benefit. The data annotations is only a subset of all the configurations you can achieve with Code First. I'll give you a quick look at using the flow in API and then point you back to that earlier Code First course where I went in-depth on this topic as well. In order to apply configurations fluently, you need to interrupt the model building process to let it know about these additional configurations. DbContext has a virtual method called OnModelCreating. That's where the configurations go. You can just add the configurations directly into this method or if you have a lot of them, you can group them in their own classes and then add those classes into the model builder. You use a combination of lambda expressions to specify which entity and which property you're configuring. For example, specifying a key is an entity method. In other words, you're saying, "The key for this entity is," and then express the property. You can add configurations to a property also. First, you have to specify the entity then drill to the property, and then, add the configuration. Here's the one for changing the column name. Relationship configurations are pretty confusing. You actually have to describe both ends of the relationship even if you're only changing one of them. If you can remember that the pattern for defining relationship is a has with pattern, that will help a lot and you can start from either end. So, remember that one-- so, remember that one to zero or one relation-- remember the one to zero or one relationship between ContactDetail and customer? If I start with ContactDetail, then I need to say, "It has a required relationship with customer," and then, I have to say, "How customer sees ContactDetail." That has an optional relationship with ContactDetail. Remember with the annotation, I had to also specify the foreign key but here, Code First already has enough information to figure that out. If you do need to specify a foreign key even if that's the only thing you need to configure, the relationship is fine, you still have to describe the full relationship and then append the has foreign key method at the end of it. Here's the same relationship with configuration starting with customer. So now I need to remove the data annotations that it just duplicated using the flow in API. So, if I ran the app again right now, no migration will happen on the database because the model hasn't really changed. I just used a different way to describe it. You can combine annotations and fluent configurations if you have a need to do so. I tend to use one of the other. I'll add one more data annotation to this type. Now, I'll run the app and this time I will have a model changed so the migration will happen. If we look at the database now, you can see, I still have my primary key foreign key, CustomerId in ContactDetail. The CellPhone column has the name I configured it fluently, and the HomePhone column has the length that I configured with the annotation. If you have a lot of configurations, the modelBuilder method could get cluttered. An alternate way to specify the configurations is in entity configuration classes. Here, I've created and entity configuration class to contain configurations for ContactDetails. Notice that the configurations go in the class constructor. Instead of these modelBuilder methods up here, I have to let the modelBuilder know about the ContactDetail mapping. I just add an instance of that as a configuration. Another benefit to the separate classes is that if you reuse any of your domain classes in another context, you can just add that class to the configurations this way instead of copying and pasting every configuration into onModelCreating which can of course create a maintenance problem. Here are the types of configurations you can specify with the fluent API whether you're configuring an entity's table mappings or configuring mappings related to properties. This is where I want to be sure you remember that you can learn about this in detail in the fluent API specific module of my earlier Code First course. There is one type of configuration that's not for a property or an entity but for the model itself. That's a configuration that tells the model to ignore an entity. Before I show you that, you should understand how the model builder determines what entity is going to a model. The modelBuilder includes all the entities for which you've defined the DbSet. Then it pulls in any entities that are reachable by any of those entities. That's how the ContactDetail entity got pulled in. Finally, if you have a fluent configuration for an entity, even if there's no DbSet for it or connected to it, the modelBuilder will pull that in also which is the only way you can respond to the configurations. I've had a lot of cases where an entity was included in the model because the relationship even though it didn't want the model to be aware of it. This is where the model configuration I mentioned comes into play. Let's say I removed the category's DbSet, but category will still be in the model because it's related to product, and so the modelBuilder will pull it in. I can use the modelBuilder's ignore method to force the model to ignore the category. Now, using the entity framework power tool, I can visualize the model and see that the category entity isn't there anymore.

o Reverse Engineering a Database for Code First

I've shown you one of the features of the Entity Framework power tools. Another important feature is that if you have an existing database, you can reverse engineer that database into classes and fluent configurations. I'll give you a quick run through of that feature. I've created a new class library project that I'll use for this example. If I right click on the project and choose Entity Framework, there's an option for reverse engineering Code First. I have to select the database I want and then let the tool go to work. When it's done, I have a slew of new classes. Their classes for each of the tables that were found in my database even some which aren't really related to my application. You may even notice that I didn't get to select which were the objects in my database I wanted-- the way you can do with Database First modeling. You'll get a class for every table and view and the database that's that. In my Entity Framework in the Enterprice course, I talked about how I'd use this tool to do the grunt work of knocking a lot of code for me and then I reorganize things into separate context when I have a large domain with lots of classes. These are all simple POCO classes that are aware of relationships to the other classes. All of the mappings are configured in separate entity type configuration classes in this mapping folder. Every mapping is configured even those which Code First can determine by convention and that may seem overkill but one reason for this is if you change something, say the name of a property, it's easier to modify the existing configuration than to go to the trouble of not realizing that you've broken the conventional mappings and dealing with the potential confusion that might come from that. And believe me, I have been there. Notice that I'm having some namespace issues. The tool doesn't also pull Entity Framework in. So, I'll just install any framework package to get a reference to that in my project. And then everything is happy again. The tool also created the context for me with DbSets for each of the classes that it generated and then in OnModelCreating, it added all of those mapping files in using configurations.Add.

o Summary

So, for you lovers of code over Designers, I hope you've seen the Entity Framework is for you, too. You saw that we could use our own domain classes to define a model. I showed you my favorite tool for debugging and verifying that Code First will see the model the way you intended. You learned about database initialization and migrations as your model evolves, and you learned about configuring mappings declaratively with the annotations or imperatively with the fluent API. And I showed you that Code First is not always Code First since you can reverse engineer an existing database to get a leg up on all of that coding. And now you've seen all three methods of creating models for entity framework to use. Database First, Model First, and Code First. In the next video, we'll get coding against those models and put Entity Framework through its paces and a little application or two. I'm Julie Lerman and thanks for watching this video about Entity Framework 5 Code First modeling.

o References

Here are some more resources you might find helpful. ( Silence )

• Entity Framework 5: Interacting with Your Data Model

o Introduction

Hello, welcome back to Getting Started with Entity Framework 5. This is Julie Lerman, and this video will focus on coding against your data model with Entity Framework 5.

o The DbContext API

In the last three modules of this course, you learned three different ways of defining an entity data model. Two of them, Database First and Model First, depended on the Entity Framework designer combined with code generation; and the third, Code First, let you skip a visual designer and just write your own code. Regardless of which path you choose, you'll end up with domain classes and one or more Entity Framework DbContext classes that allow you to retrieve and persist data relevant to those classes. You got a look at some of DbContext classes in those modules. In this module, I'll focus on the basics of using The DbContext API in your applications as a bridge between your classes and your database. The DbContext is one of the most important classes in the Entity Framework. It enables us to express and execute queries; it takes query results from the database and transforms them into instances of our model classes; it can keep track of changes to entities, including adding and deleting, and then triggers the creation of insert, update and delete statements that are sent to the database on demand. Your domain classes don't need to be aware of the Entity Framework; as long the DbContext is aware of your classes and aware of the database, it can do its job. When you use the Entity Framework designer, that will generate a class that inherits DbContext and is aware of the classes you described in your model. When you're using Code First, it's up to you to code up a DbContext class unless you've used the Entity Framework power tool to reverse-engineer a database; that tool creates the DbContext class for you.

o Outline

In this video, we'll get started with using the DbContext to interact with your database and domain classes to define and execute queries and send data to the database. We'll keep an eye on what's happening in the database with a profiler. We'll look at working with graphs of related data and also look at working with stored procedures in your database. Let's play a bit with the DbContext class and then drill into some of what's going on.

o Your First Queries

Since the three models I built are similar, I'll grab the solution where I used Code First to build my model. When you're writing it up it doesn't make a difference which workflow you use to define your model, except for the fact that the Code First model doesn't support direct mapping to stored procedures in the way that using EDMX and the designer lets you do. Just as a quick reminder, here are my domain classes, Customer, Address, Order and some more detailed types, as well as a few enums. If I flip over to the Database First model, I have some of the same classes to work with. The only difference is that these classes were generated, whereas I built my other classes by hand. In fact, the same goes for the class that inherits from DbContext. When I used the designer, that was generated; when I didn't use the designer, I created it by hand. They're not terribly different. The only extra code you see in the DbContext that's going to use Code First is that I've described some of my mappings in code instead of doing that in the designer. I'll start with a simple method, one that lets me get all of my customers. In order to do any interaction with the database, I need to start with an instance of my SalesModelContext, which is a DbContext. And I can express a query using one of the DbSets that I defined in the context. So that would be Customers. But all I have here so far is a DbSet. If I run this line of code, I'll get a DbSet instance, but the context won't execute a query. I need to use one of the query execution LINQ methods, such as ToList. When I call ToList, that tells Entity Framework, "Okay, I want a list of the Customers," and in response, Entity Framework has to go get that list, which means executing the query. There are a number LINQ methods that will cause a query to execute, whether you want to return a set of results, like this list of Customers, or an Object where you would use one of the singleton methods. So I'll create a list of customers, then to prove to you that I really got them, I'll iterate through that list and print out the first name of each customer. There's a lot happening behind the scenes, which I'll explain later. I just want to focus on the code right now. I don't really need to have my query in a separate statement, I could put it directly in the foreach statement instead. In fact, if I do that I don't even need to call ToList, the foreach statement has the same effect because the app is saying, "Okay, I need those Customers now," when it's doing the iteration, and that causes the query to be triggered. And there are my Customers again. What if I just wanted to get a single customer? I could write a query, but there's a method I can use to find an entity using its Id. It's a DbSet method called Find. Besides how simple it is to write, another great benefit of the Find method is that it'll check for the entity in memory before it bothers with a call to the database. But keep in mind that Find is a method; it's not a query-able. So if you want to do anything fancy in addition, you'll have to revert to a query. You'll see more querying examples as I move forward. Now let's see what it's like to add a new customer. I don't need Entity Framework's context to create a customer; I can just do that the normal .NET way. Yeah, you'd probably like to know how old I am, wouldn't you? But, hey, I wasn't born just yesterday. But for now I'll just say I was born today. Now I need Entity Framework. I'll instantiate a context and add this new customer to the Customers DbSet. By using the Add method, I'm specifically telling the context that this is new and needs to be inserted into the database. Even if this entity already had a CustomerId, the context doesn't care, it'll still insert it into the database because I called Add. Of course, if I insert a customer with an existing Id, then the database is going to throw an error, because I'm trying to insert duplicate primary key, but this customer is brand-spanking-new so I don't have to worry about that. DbContext has a SaveChanges method. That'll cause the context to inspect all of the entities it's tracking and figure out what commands need to be sent to the database and execute them. I'm using a SQL server provider, so I'll help Entity Framework construct the correct TSQL. If I was using Oracle or MySQL provider or one of the other database providers, then I'd get the appropriate SQL for those databases. The same thing happened when I executed the queries: the SQL client provider I'm using helped Entity Framework write the TSQL. When I'm done, I do want to prove that this new customer really did get into the database, so I'll call the GetCustomer method again. That uses its own context, so it won't know about the customer that this context is already tracking. So it will truly go to the database to look for the record. I need to make a small change to my GetCustomer method, though, because I hard coded the CustomerId into it. I'll change it so I can pass a variable in instead. But I'm depending on Entity Framework's behavior for inserts. SaveChanges won't just send an Insert() command; it'll also add SQL that will retrieve the Id that the database created for the new row and update the Customer entity with this new CustomerId. So now I can just pass customer.CustomerId into the GetCustomer method, because by this point, I'll have the true database value for the new customer row. Now I can run the InsertCustomer method and you can see that the call to GetCustomer is showing me the customer I just added to the database.

o Inserting, Updating and Deleting Data

Next is updating. This time I'll use FirstOrDefault for my query instead of Find because I don't happen to have the Id to pass into the Find method. I suppose it's silly to say I was born today, so I'll get a little closer to the truth and change this to say that I was born 25 years ago. What's a little white lie between friends? And then I need to call SaveChanges. Since the context was tracking this Object when I made the change, it made note of it. So it'll note a construct and Update method and it will only update the property that changed. We'll look at the SQL in a bit so you can see. To prove that the change got saved, I'll modify the GetCustomer method to write out the birthdate, and I'll also write out the date just before I edit the customer. Remember that GetCustomer is getting the customer from the database, not dealing with the customer instance that I'm using in the UpdateCustomer method. When I run the method you can see that my initial birth date is February 2, 2013. Then after I save it GetCustomer finds the new rule on the database and prints out the name and birthdate, which is now February 2, 1988. I've added one last method. This one's for deleting and I used it to delete the new record that I inserted. In fact, I'll get rid of that and any other records for Julie that are in the database. So here's a LINQ query against the Customers DbSet where I'm searching for all customers whose first name is Julie. The Wheremethod isn't an executing method, so I still have to add something to execute and my default is to use ToList. Once I've retrieved the list I can iterate through it and remove each of the customers. Remember earlier when I said I could just put the query without ToList into the foreach statement. In this case that would break one of .NETs rules; without ToList, I'd be removing results as they were being streamed back to me. And if I start doing that before the stream's reached its end then the foreach will totally get confused about where in the list it's pointing. So I need to perform this operation on a list and therefore cxplicitly use the ToList method. Again, since the context is keeping track of these items when I call Remove, when I call SaveChanges, it knows to create Delete commands for each of the items that are removed. When I'm all done I'll call GetCustomers again to be sure that there are no more Julie customers in my database. Now I want to run through all these methods again and see what's happening in the database. I'm using Entity Framework Profiler from Hibernating Rhinos, so I've added a reference to that in my ConsoleApp project. And because I used new Get to pull that in, Entity Framework Profiler also added some start-up code for me so that it will note a profile and send the results to its user interface. So I'll uncomment all of my methods and run the whole kit and caboodle, then flip over to the UI to see what it captured. The first few methods are a bit of migrations codes, so I'll skip those. Here's the call from my first method, GetCustomers, the query explicitly projects every column that maps to my Customer entity. That's about it. The Stack Trace shows you what code in your app triggered the particular command, and you can see it was GetCustomers. The next method was GetCustomer where I used the Find method. Find will trigger the same query that LINQ's single method uses, which is to SELECT TOP 2. The reason it does this is to be sure there's only one result in the entire database. If you get two, then an exception will be thrown, because Find's job is to return a single result, if there's more than one, then something's wrong. And notice that it passed the value to search for as a parameter. Checking Stack Trace again, you can see that this was in fact from my GetCustomer method. I can even double-click and get directly to the code. Next was my Insert method. I created a customer attached it to the context and called SaveChanges. And look, the Insert method is wrapped in a transaction. This is Entity Framework's default behavior. Whether the context is doing one or more updates, they'll all be wrapped in a single transaction. And if anything fails, all the commands that have executed up to that point will roll back. It's also possible to override this behavior and use your own transaction scope, but that's getting ahead of this getting started course. So here's the Insert command, and remember I said that Entity Framework will be sure that the new Id gets returned. There's the SQL that makes that happen. Remember that after I inserted, I passed the new Id into my GetCustomer method which called Customers.Find to get the new row from the database. Well, there's the SELECT TOP 2 again, and you can see that it passed in 21 which must be that new Id. Now for my update. First I want to point out that Entity Framework Profiler is showing that I called Select and Update from the same context. But this Select isn't parameterized. There's a reason for that. I didn't use a variable; I hard-coded the predicate directly into my LINQ query instead of using a variable. So Entity Framework translates that into a straight query without a parameter. Now here's my Update method that saved changes pushed through. There's the transaction again, and since I made the edit and called SaveChanges in the same context instance, Entity Framework knew exactly which property was changed. And so it's only updating that one. It also knows to use the Id of the customer for that update. And remember I called GetCustomer again, so here's the query in the separate context instance inside the GetCustomer method. And finally, my Delete method. Remember I wrote a query to get all customers with the name Julie. Here's the resulting TSQL; again, it's not parameterized because I hard coded my name into the query. After I called Customers.Remove for all of the Julie customers, and there was really one, I called SaveChanges. Here's the Delete command that Entity Framework constructed and sent to the database using the Id of that customer. And finally, one last call to GetCustomers to make sure that there were no longer any Julie's among them.

o Building and Inserting Graphs

So far you've seen me work with single entities. You can also work with graphs of data; for example, an order and its line items, or a customer and her orders and line items for those orders. I'll start by inserting a graph of related data, and then we'll look at different ways of retrieving related data. I've added a method to my app called CreateCustomer with the order. I've had to do a little advanced data retrieval, because when I create an order I'll be adding line items, and I need to know what my products are. In a real app, the products would already exist before creating an order. So I'm grabbing a few products that I've already got in my database. Now I'll start building my graph. Just like the earlier insert, this part has nothing to do with Entity Framework; I'm just writing normal .NET code. Here's my customer and I'm getting really fancy here and adding a ContactDetail object on the fly to go along with the contact because what's the point of having a customer that can't Tweet about your awesome company? I could build the graph up in one enormous statement, but I don't find code like that to be very readable or maintainable. I don't mind one level of on-the-fly object initializers, like I did with the contact detail. I'll create an Order in its own variable. Here you can see one of the simplest ways to populate a DbGeography type. Remember that Entity Framework 5 understands this data type now, so we'll be able to persist it and retrieve it from the database with no problem. And I'm using one of my enums to define order source. I'm creating the line items on the fly as part of the order instance. Notice that for each of the line items I'm setting the product to one of the product objects that I retrieved at the outset. Currently the order and the customer aren't aware of each other. I could have set the customer when I created the order, but I'm being careful about the precedence of my relationship. If my context was already tracking these objects, then when I set the relationship, Entity Framework will ensure that it's automatically two-way. In other words, if I say myOrder.Customer=myCustomer, then myCustomer will also know that the order is in its Orders collection. But I'm not inside the context at the moment, so the customer won't be aware of the order that way. Instead I'll explicitly add the order to the customer's orders property. At the moment, then, the order doesn't know about the customer, but that's okay. You'll see how this works out. Now I'm ready to work with the context. I'll add the customer to the context Customers DbSet, just like I did earlier with the InsertCustomer example. But this time, since the customer has a contact detail hanging onto it, and an order -- and that order has two line items, and each line item has a product attached to it -- when I add the customer, the whole graph, all of those objects are also added to the context. And because I used the Add method, everything in the graph is marked as added. When I call SaveChanges, Entity Framework will create insert statements for everything. One of the really nice things here is that it takes care for InCase for you. You might remember having to write this code, insert the new customer, get the new Id for the customer, put that into the contact detail, and then insert the contact detail. Now put the new Id into the order and insert the order, and so on and so forth. Remember looking at the SQL for the insert and seeing that the new Id was also queried as part of the Insert operation? Well, here's another benefit for inserting graphs. Let's go ahead and run this method. I've set a few break points. Here's the first. I wanted to look at the Customer graph before Entity Framework gets involved. So here's the customer. It's got a contact detail and an order. The order has its two line items, and the line items have their product. And that product instance looks a little nasty; there's a very specific reason that it's like this, and it has to do with the fact that I reverse-engineered some of the classes using the Entity Framework power tools. I'll talk about this later when I discuss loading related data. So for my current intention this is actually perfectly okay, but I didn't want to just ignore the elephant in the room. So I'll run through the rest of the method, and everything must have been okay. But I've already given you a hint that there might be a problem. But let's go back and take a look at the Profiler now. The last context is where I called SaveChanges. Here you can see that all of the commands from SaveChanges are surrounded by a single transaction. Now I've got my customer insert with the return of the new CustomerId. And I insert the contact detail, and look, here's the CustomerId that just came back from that first insert. Now I insert the order, and there's the CustomerId again. Oh, and here you can see how Entity Framework handled transforming the DbGeography type into a SQL Server geography data type. And remember I used an enum to specify the Order source. Entity Framework translated that into the enum value that I defined on my enum type. So there's that value, which is 5. And there's the CustomerId I'll use to associate with this order. The next insert is for the Juicy Dog Bone product. Yeah, Entity Framework is inserting another copy of my product. I definitely don't want that to happen, and there was no indication of a problem when I ran the code. Entity Framework did just what I told it to do. So I'll go back and fix that in a minute. Here's the other line item, and the second product that I use. And I don't want this one to get inserted either. Okay, so let's go back and look at the code where I accidentally inserted the product and see what's going on. I queried for the products in their own context. By the time they got back into my CreateCustomer method, they were just plain old product instances. Entity Framework has no knowledge of them. It doesn't know that they came from the database. We know, and we can tell by seeing that there's an Id, but Entity Framework won't know. As a matter of fact, at this point, Entity Framework isn't even around. So then I added those objects to my graph. When I added the complete customer graph to the context using the Add method, everything in the graph got marked by Entity Framework as added. It didn't know that the products were special, and it didn't stop and say, "Hey, there's already an Id here. Are you sure you want to add it?" It just did what I told it to do. I said, "Add this graph," and so it applies added to everything. So this is a really common problem when we're dealing with disconnected data. If we had queried for the products within the context, then the context would be aware of them and wouldn't change their state to added when I added the customer graph, but they weren't. They were coming from elsewhere. So that's what caused that problem, and there are a few ways to avoid it. But my preferred way is to set the relationship using the foreign key and not the navigation property. That way I just avoid all confusion. If you don't use foreign key properties in your classes, then you have to learn a lot more about Entity Framework's behavior. You can learn a lot more about working with disconnected graphs in my Entity Framework in the Enterprise course. So let me do the painless fix using the Ids. And I have the products, so the Ids are right there for me to use. When I run the method again, voila: the context doesn't insert the products this time.

o Querying with LINQ Expressions or Methods

Before looking at any more queries, I want to make a note about the LINQ syntax that I've been using. There are two ways to express queries with LINQ. One is with LINQ methods and lambdas, which you've seen me use; the other is with LINQ expressions, which look a little more like composing a SQL expression. When I first started using LINQ, I was daunted by the lambdas and so I mostly used expressions, but now I prefer using LINQ methods with lambdas and they've become second nature. The basics of querying with LINQ to entities hasn't changed dramatically since my earlier course on Querying with the Entity Framework, except that in the earlier course, I was querying from an object context rather than the newer, more efficient DbContext. Keep in mind that there are some syntax differences when you use LINQ in VisualBasic. You can also dig further into LINQ with these courses from Scott Allen, but I'll continue to use LINQ methods and lambdas in this video.

o Loading Related Data

Now that I've created some related data in the database, let's take a look at retrieving related data with Entity Framework. There are two categories for getting related data: one is to get a full graph of data at once from the database in a single query, and the other is to load related data after the fact. Pulling a graph from the database can be done either with EagerLoading which uses the Include method on a DbSet, or with query projection, where you can specify navigation properties be selected along with the root entity that you're querying. Alternatively, once you have objects in memory, you can load the related data after the fact. This is possible either by explicitly calling a Load method or implicitly by taking advantage of a feature called LazyLoading. Let's have a look at the four methods. This first method explores EagerLoading with the Include method. You can specify which related data you want as a lambda or as a string. The ability to use a lambda for Include is relatively new and helps you avoid typos. When you're using the lambda with Include, in order to get IntelliSense to help you, you do need to have a reference to the System.Data.Entity namespace. Otherwise, you'll be trying to type the lambda, and you won't be getting any help from IntelliSense. You're not limited to a single relationship with Include, but if you use too many includes, you can run into performance problems, because the generated SQL degrades as the query gets more and more complex. It's easier to describe multiple layers of Include with a string and dot notation, such as this one, where I'm saying, "Get the customers, and their orders, and all of the line items for those orders." The last query does that and adds one more level, which is to bring back the related product for each item. You can drill into graphs with Include, but you can also get siblings. For example, I could include orders and also include addresses. I'll run this method and then look at one of the customer graphs returned by the final query. So here's that customer, and you can see there's only one order. And that order comes along with its line items, and then each line item has the product that's related to that one. If I look over in the Profiler, you can see the four queries that were executed in the database as a result of my four separate LINQ queries in my method. I need to point something out first, though, in case it's catching your eye at the moment. That first duration looks quite long; that's not related to the actually query, but the startup cost of using Entity Framework in the application. You pay that price only once per application process for each context that you query against. So the first time I use the sales model context in my app, there's some work to be done in the background, but then the results of that work stays in memory and doesn't need to happen again, no matter how many times you instantiate the context or execute queries on it. : Here's a great article on the MSDN website about performance with Entity Framework and some things that you can do to reduce or even eliminate some of those startup costs and other performance issues. Okay, so back to focusing on the EagerLoading. The first two queries are exactly the same. In one, I used a lambda to include orders and the other I used a string. So you can see that in response, the TSQL has a left outer join between customers and orders and then projects all of the relevant fields. In the next query, I included the orders and their line items, so the SQL gets a little longer, partly because there are more fields to project, but there's also some more logic here to handle the additional relationship to the items. Finally here's the query where I also pulled in the products. Since there's only one product per line item, this one doesn't grow as dramatically as the previous one did. But you can image that if you go deeper and deeper into a graph, the SQL will become more and more complex, and at some point it makes sense to break up the queries or pull in some data and then explicitly load the rest of it. Another drawback to Include is that you can't limit or sort the related data. It's all or nothing. Using a projection to EagerLoad can help you get around that. First, here's a simple projection. I'm querying customers and saying to select the customer and the customer's orders. A new type is created with one property being a customer type and the other a collection of orders. The result is now a list of complex types instead of a nice, neat list of customers. But it's easy enough to drill into che Customers and once you have a customer you'll see that Entity Framework's actually worked out the graph of that customer along with its orders, even though those orders are in that other property. So you don't really gain much over the Include with this example, but the next query you can see that in my Select, I don't just get to see orders, but instead I am sorting the orders by date and then grabbing the first one. So that's how you can get around that all-or-nothing limitation of Include. The resulting SQL queries have a left outer join for the ones that aren't filtered and then an outer join to filter the order query. So those are ways to get related data to be returned from the same query as your base entities, and we call it EagerLoading. Sometimes you might want to query data and then, in your application logic, go get some more data that's related to what you already have in memory. This is where the Explicit and LazyLoading come in, so you don't have to create and execute more queries. First I'll show you the ExplicitLoading, I'm grabbing a random customer. Now it's in memory, and you can see that this is all that happened in the database. Now this exact code isn't a great use case because it would have made more sense to EagerLoad the orders, but imagine you've got some brilliant logic where it makes sense to work with the customer that's in memory, and then at some point go get that customer's orders. It's not a job of the customer, but a job of the context to load the orders. So you start by using the context.Entry method to access the tracking information about the customer. Then you say you want to do something with one of its collection properties and use a lambda to say which one, the orders. And finally what is it I want to do with this collection? I want to load it. This method doesn't return anything. I'm not creating a variable to capture the results. The results will just be pulled into memory and become part of the customer's graph. So now I'll execute that line of code, and there you can see Entity Framework created and executed a query for me. Now I can drill into the customer's details and display the number of orders that are attached to the customer now. You can also make this happen implicitly without having to call Load with a feature that's called LazyLoading. Let me copy this method and change its name, and then I'll comment out the call to Load. If I ran this right now, I'll get the customer but no LazyLoading. I have to enable it, and there are two keys to LazyLoading. The first is that you need to identify which properties in your types you want to be able to LazyLoad. You do this by adding the virtual keyword to the property. So now this is a runtime flag for Entity Framework to know that it's allowed to LazyLoad the orders property. But that will only work when the context is set to LazyLoad. I'll stick that here for now, and I'll run the method, and there's the order. You can make sure the context is always configured for LazyLoading by setting the configuration in the constructor for the context rather than as needed in your application. But right now, Orders is the only property that will LazyLoad. So even if you have LazyLoading enabled for the context, only the navigation properties marked "virtual" will have that behavior. So for customer, contact detail and addresses will not be lazily loaded. You might recall seeing that bizarre product type in the customer graph earlier. In the line item class, the product navigation property is also marked "virtual" for LazyLoading. Whether or not LazyLoading is enabled, at runtime when Entity Framework sees that the property is virtual, it wraps it in a proxy which delivers the logic that will make the LazyLoading work. So it's not really a simple product type anymore, but for all intents and purposes in your code, it still is.

o Working with Stored Procedures

Now we'll look at working with stored procedures. Remember that when working with a visual designer, we can include information about database-stored procedures and their mappings directly in the model. There's no way to do this when you've defined your model with Code First, but it's still possible to call stored procedures and other commands directly from the DbContext. Let's first look at the mapped procedures. I flipped over to the solution where I created a model using Database First against an existing database. So here I'm using the designer, and with a designer you may recall this is where I do have access to a lot more stored procedure mappings. So one of things that I did in the DbFirst module of this course is that I mapped existing stored procedures that could Insert, Update and Delete a customer. So here I am in the model, select customer and mapping details, and I'm on the stored procedure mapping section of mapping details. And you can see that I've got mappings to a stored procedure called InsertCustomer, one called UpdateCustomer, one called DeleteCustomer. So anytime I do a modification to a customer that's being tracked, and I call SaveChanges, instead of constructing a command, Entity Framework will use the appropriate stored procedure. And also remember that I mapped properties of the entity to parameters in the stored procedure. Now that's one way of mapping with a designer. The other is to create function imports, and you can those in the model browser. You might remember I had a whole bunch of other stored procedures, but they weren't modification-stored procedures that I could map directly to a particular entity in my model. These are stored procedures that I just need to execute randomly. And in this case, all of these happen to be stored procedures where I was returning data, where I was pulling data back into my application, although I could also have done this with stored procedures where I am just pushing something in. So I'm going to show you the effects of working with the mapped stored procedures. Those are the Answered, Update and Delete ones. And what I've done is I've added a console application to this solution. It's just very simple. It has a reference to my model, and it also has a reference to Entity Framework. And in here I created a single method that will run through an Answer, an Update, and a Delete against the customer. So here I'm creating a customer and then calling SaveChanges, so we'll see what happens when an Insert is executed. Then I'll take that same customer while it's still being tracked, even though it's no longer new, still being tracked by the context. I'll make some edits to it, and I'll call SaveChanges again to see the Update happen, and then finally with that same customer as it's still being tracked by the context, I'll call Remove. And when I call SaveChanges, that will trigger my Delete stored procedure. And you can see I'm failing already when I call SaveChanges after the Insert. So I'm going to drill into this error. And I can see down here that it's saying that there's a problem with converting datetime2 to datetime. Now, I'm very familiar with that particular error, and I know what that means is that I'm trying to push a bad date into the database. And I know exactly what the problem is. The problem is that I haven't set modified date here. So I could just add the modified date here, which I'll do for now, but in a real app, I would want a better way to consistently update the modified date field. I'll show you how to do that in the next module where I talk a little more about application design with Entity Framework. So let's run this again and see what happens this time. I'm getting another error, but this time the error's getting thrown when I try to save the Update. What this error is telling me is that I was trying to either call an Update, Insert or Delete, but it couldn't find a row in the database to act on. Let's take a look at EF Profiler to see what happened. So the problem wasn't at the Insert. The Insert worked just fine. The problem was with the Update.Customer. And if you look at the parameters that got sent in to Update.Customer, you can see that the CustomerId is 0. Okay, it's not using the CustomerId of the new customer that I inserted. There's a reason for that. Remember a little earlier when I showed you the inserts that Entity Framework was composing, that it would not only do the insert, but it was pulling back the primary key of the new row that got inserted, and then Entity Framework would automatically take that key and update the entity key value, in this case, the CustomerId of the customer with that value. That's not happening with the stored procedure. In fact, the stored procedure is designed to do that, but I didn't take that into account when I did the stored procedure mapping. So let's fix that problem. Here's the stored procedure and I want to show you that, in fact, the stored procedure was designed to return that new primary key at the end of the procedure. What I didn't do when I did the stored procedure mapping was take that into account. So now I'll show you how to do that last piece. So I'm back in the Customer model, in the mapping details, and just after all the parameter mappings, there's a section here called Result Column Bindings. So this stored procedure doesn't just do an Insert; it sends back results. Now I can capture that result, which I know will be the value of the key, and map it to the CustomerId. Now my code ran error-free and you can see in the Profiler I have an Insert wrapped in its transaction because I called them all separately. Then I made an Update and called SaveChanges. There's Update.Customer and finally I Removed.Customer, called SaveChanges, and there's the Delete. So now you've seen some simple use of the stored procedure mapping, where stored procedures replace command generation when SaveChanges is called. Now, I don't have stored procedures mapped to any of the other entities, so those entities will get command generation. And as you drill more into the stored procedure mapping, you'll see that there's rules about mapping stored procedures across inheritance hierarchies and in relationships and thing like that. As a reminder, I also created those function imports for some of the other stored procedures in my database. Now, if we look at some of the generated code from that model, you can see that the T4 template is designed to take those into account. So here's just some of the stored procedures. One of them was called AnnualCustomerSales, the way the function import is defined in the model, it's aware of the fact that it takes in a string parameter called fiscalyear. So the method signature takes that into account. It uses it to create an ObjectParameter, then does a little trick to get from the DbContext that we're working with to the ObjectContext that's really behind the DbContext. And from there, calling an ExecuteFunction method, expecting a return of a type called AnnualCustomerSales\_Result. Remember, the function import wizard creates those complex types then passes in the name of the stored procedure to execute along with its parameter. When working with Code First, you don't have the benefit of these mappings between functions and complex types, but we can take advantage of a number of different ways for DbContext to execute commands directly on the database on your behalf. It has a method called ExecuteSqlCommand, so you can see that here I'm asking it to execute my AnnualCustomerSales start procedure and I'm passing in a parameter. Also be aware of the fact that there's a SQL query method that you can take advantage of as well.

o Summary

You've probably noticed that I keep wanting to tell you everything I know about Entity Framework here. But you have had a good look at getting started with using the DbContext to interact with your model and domain classes in this module. You've seen some basic querying and updating, what's going on in the database. We've seen Entity Framework's affinity for working with graphs, for creating and sending them to database, and also looked at a number of ways to load related data. We looked at working with database stored procedures, and even when procedures can't be mapped to the model, call commands directly on the database. In the next and final video of this course, we'll move out of this little single-file ConsoleApp and look at some of the considerations for using Entity Framework in your application architecture. I'm Julie Lerman, and thanks for watching this video on Coding Against your Entity Data Models.

o References

Here are some resources you might find useful.

• Entity Framework 5: Using EF in Your Solutions

o Introduction

Hello! This is Julie Lerman. Welcome back to my course on getting started with Entity Framework 5. In this final course module, I'll show you how Entity Framework fits into your software.

o Outline

First, I'll show you a few common techniques for making your models better citizens in your application architecture. I'll show you how to keep your designer generated domain classes separate from the model and data access portion of your solution. Another important technique for developers using the designer is how to extend the generated classes with your own logic. I'll show you how you can override the context-saved changes method to add logic that you might want to perform to your entities before pushing them into the database, then uphold the data acces logic out of the UI code and encapsulate it into repositories. With those tools in hand, I'll then show you a WPF App that's using the domain classes, the model and the repository we've just built. I'll then do the same in an MVC App. And finally, because services and server site APIs are becoming increasingly important for providing data to mobile apps, I'll take a quick look at using our model in a WCF Data Service and consuming it in my first ever Windows store app. Well, that part is more of a hack job, but you know where you can find great courses on developing Windows 8 apps anyway. So, let's get started.

o Separating Generated Domain Classes from EDMX

When I created my code first model, I kept my domain classes in their own project along with that ANIM, so all the logic for my domain is in one project and all the logic for my data layer, all the Entity Framework stuff with a context and migrations, they are in their own project. Except for one very specific use case which is that in the order class, I'm using a DbGeography, and DbGeography needs System.Data.Spatial which is currently in Entity Frameworks system.data.entity.dll. Other than that very, very special exception, I have no need at all for Entity Framework in my domain classes. I have nothing to do with Entity Framework. So, I'd like to have that separation of concerns in my application, my domain is in one area and, you know, maybe if it's a large domain I might have that split up into many more projects and my Entity Framework code is in its own project. Now, let's look at my database first solution. I have a project specific for the model that has the edmx file in there and it is in code generation to create my domain classes, but the way the designer works by default in Visual Studio is that all of that generated code sits in the same project along with the model. Now, if you're just creating a simple little application, I don't see a problem with this, but if you're creating something more enterprise level, you probably want to think about that separation of concerns again where you have your domain logic in one area and your data access, your Entity Framework logic in another project. And this gets it highlighted when you want to start customizing these classes, but what that would mean in this particular scenario is that in addition to my domain classes being in here, all the rest of my related business logic for those classes when I want to extend them, maybe have custom properties or some methods. I have to keep stuffing all of that into that same project which is really my Entity Framework project. So, what I want to do first is show you how you can separate that out. So, here's my domain classes' project and that's where I want the generator to put my classes, so I want the model-- so, I want the model in the model project, I want to have the domain classes project contain my domain classes. Now, I could just take all of these generated classes and copy them and paste them into the new project, but then when I change the model, that code generation won't be able to update those classes. So, I still need to have some kind of link between my model and these classes so that the code generation can happen properly. There are few ways to solve this problem. One of them, I'm not a big fan of but you are going to see a lot of references to it if you're googling or talking to other people who have done something similar. So, I want you to be aware of it and explain to you why I personally don't happen to like it, but it might be perfectly satisfactory for your scenario. What that is, is to use something in Visual Studio that allows you to make a reference from one project to another and not to the other project but actually to the class. Here's how you can do that, I'm going to go to my domain classes, I'm going to add an existing item and I'm going to add that TT-class that generates the domain classes. So, I'll go to my Adventure Works Model, I had to have all files. There's the template for the context. I don't want that, I want the template for the actual model. And instead of just clicking add, I'm going to add as link. Notice that now the tt files there and it automatically regenerated all those other files. And if I select, the tt file appear, and if I select this tt file down here, notice that it keeps pointing up to the file in the other project. Now, I do have my domain classes inside of my domain classes' project, kind of. They're really just mirrors over there, and if I make a change in my model and then save it, I can see that the customer file in my Adventure Works Model class is getting updated, and if I open the one inside the domain classes it's also updated. So, really what this is it's just a mirror of what's inside of the other class. There's something else I don't like about this which is my domain class still have the same name space of the project that they're coming from. I don't have to go and modify the template used, the domain classes' name space. It's totally doable and totally solvable, but I think it's messy and I don't like it. So, I'm going to show you another way, it does mean making a minor change to the template, but I'm okay with that. So, I'm back to starting from scratch again. I have added the new domain classes' project into the solution. I haven't done anything with the tt class yet. So, what I want to do is I want to take the template out of the Adventure Works Model project and put it into the domain classes. And like I said before, you can't just copy and paste from here because the way Visual Studio's handling this, the tt file, has a dependency on the edmx file. So, what I'll do instead is take this AWModel.tt file out of my adventure works folder and put it into my domain classes folder instead. Now, I have to include that file in my domain classes. Don't worry about an error with the transformation yet, I haven't given it all the information it needs. And then I'm going to just delete it from the model project. So, that's deleted along with all of the domain classes that it was looking for. Now, I have my template down here, the only problem is this template can't find the model, the default template assumes that the model is in the same file path, so I'm going to modify this template to tell it where to find the model. So, now I've given it an explicit path and I'll save this and notice that it automatically generated all the classes for me. So, there are the classes and I'll open up customer and there's the correct name space. Now, what happens if I make a change to my model? So, I'll go back to customer and I'll change first back to first name, I'll save the model, but I'm not getting a warning about that customer file being overwritten. So, you can see that in customer I've still have first. So, I do have to explicitly Run Custom Tool to get the classes regenerated. There are two other things I need to do to my new domain classes' project. One is because I'm using DbGeography, I do have to have system.data.entity, I had to do the same thing with code first. If I wasn't using any geography types in my model and therefore my classes, I wouldn't need that. The other is I've got this ANIM that I defined out in my data layer, and that ANIM now belongs, really, with the domain classes. There's one last problem I need to address here which is that using the default template, the context expects all of those classes to be in the same name space as the context, but they are no longer there, they are in a domain classes' name space. So, I also need to make an edit to the template for the context. If we scroll down here, you can see, here is to place in the context where it spits out the various usings for the different name spaces that we need. So, I will add it using domain classes, and you can see that the file got regenerated but it can't find domain classes, that's because I need a reference to the domain classes project in this project. Now, it's happy with domain classes and everything builds properly. Now, you've seen the difference between explicitly putting the template into this project and what I have to do to make it work, I have to make some template modifications and I do have to regenerate the classes anytime I change the model. I'm good with that because I just don't like the messiness of-- I personally don't like the messiness of the linked files, but there are a lot of people who are very happy with them. So, it's just a personal preference.

o Adding Logic to Generated Code using Partial Classes

One of the problems with working with generated code is that you can't add custom logic into the generated classes or the next time the code gets generated, your custom logic goes away. However, the code generation templates create partial classes for all of the generated classes including the context class. So, it's easy enough to extend them by creating additional partial classes. One of the important roles for partial classes is that they have to be in the same project at a compiledtime.net pulls the logic of all the partial classes for single class into one class but this still stay in separate files in your project. I've added a partial classes folder inside of my domain classes project and created a customer class and this customer class will extend the generated customer classes. In declaring this class, I say that it's a partial class. So, if compiledtime.net will see this partial customer class and the other partial customer class and pull them together. And also, because of the partial, it's not going to say, "Wait a minute, I already have a customer class inside of domain classes." I also want to point out that by default, when I create this class, visual studio will make the name space be domainclasses.partialclasses because I've put it in a folder. I do need this customer to be in the same name space as my other customer. So, I fix that name space to be just domain classes. I've created a property called full name so that anytime I want to use a full name in my application, I don't have to keep concatenating first name and last name together. So, I'll just have a get return there and just to show you that it's aware of the fact that this class is part of the other class, I say these, there are all the properties of the customer class. So, I can say this last name probably want to trim it. So, now I have a full name property, easily accessible in my customer class. A really important thing to understand about these custom properties is that you can't use this in a query because full name doesn't really map to anything in the data base. So, your query needs to still pull back the last name and the first name but in your application code when you're interacting with this customer object, you can use the full name property. And notice, I don't have a setter. I don't want to suggest that you could set the full name and have that be persisted back to the data base when you call SaveChanges. Now, in back in my console application, I've added a reference to domain classes. So, that its aware of my classes now that they are moved into a different place and I created a little method called GetCustomers. It will go and grab first five customers for me and iterate through them. Now, my code isn't quite complete yet because I can't just write out customer. So, I just want you to see that there is a full name in IntelliSense and I can go ahead and run this and they are all their full names, the last name comma first name which is the definition of full name for the first five customers. Another customization developers frequently ask about when using an IdiomaX and code gen is how to apply data annotations to the generated code. The best way to do this is using a dot net feature called MetadataTypes. The feature is more commonly known as buddy classes and even ugly buddy classes by a few. Like partial classes, you can have logic to the generated code in a separate file but then the compiler will zip it all together at build time. Fellow Pluralsight authors Steve Smith has a great blog post on how do use Metadata in classes on his blog here at Ardalis.com.

o Overriding SaveChanges

In the last module, when I show you how the map stored procedures worked, we ran into a problem quickly because I hadn't supplied value for the modified date property and customer when I was trying to insert a new customer. And entity framework didn't even catch that, it was the database that caught the problem. What happened was, since I had left the value null .net put in its default minimum date time value and then that was part of the insert into the data base. Well, .net's minimum day is a few centuries earlier than SQL server's minimum date. Whenever you use that value to insert into a data base, you're going to get an insert into a SQL server database you'll get an exception. What I did is a quick fix for the problem. It was just to set the modified date in the insert customer and on at the same time, I said, "This isn't really a sustainable solution because when I have to do that all over my code and that I would show you a better way in the next module." So, here we are with that problem. Now, it happened to be in the code first application now, not working with a generated code. The solution that I'll show you if I was doing with the generated code, I could do it in a partial class to extend the context as I just showed you for extended logic for the customer. In this solution, I've gone through and added a modified date field to each of the classes. Now, I could also have done that by creating a base class. It has modified date and making each of this classes inherit that base class. I'll shift to that next. So, what I want to do is avoid having to explicitly set modified date every time I create a new entity or update an entity. And the way I can do that is by overriding the contexts saved changes method. So, here I am back on the context class and overriding the saved changes. I'm going to start by writing this code with a presumption that any class I'm possibly dealing with does have a modified date property. I've added in the logic to go ahead and set the modified date to now for anyone of those properties. So, I want to explain this code to you and then show you more efficient way to apply this logic. So, what I'm doing is I'm taking advantage of entity frameworks ChangeTracker. You can say that ChangeTracker is a property of the context because of that this.ChangeTracker, and the ChangeTracker gives me the ability to take a look at all of the StateInformation that's being stored in the state entries and then drill through them. So, I'm saying, look at all the entries that represent the StateInformation for each of the objects that you're tracking and then filter that down to only those whose state is added or modified. So, now, I'm not going to be bothering with unchanged ones or deleted ones. Now, I've got that filter down, I want to change that modified date for each one of the entities that the entries represent. So, remember, the entry is not an entity. The entry just has the StateInformation and it does give me access to the actual entity. I can't just say entry.ModifiedDate because ModifiedDate is really a property of a customer or an address. I can get access to the properties of the objects that the entry is working with but still, once I've got property, I still don't know what the actual type is. I don't know if it's a customer or an address but I can use a string to pass in the name of the property that I want to change. So, this modified date set the current value of that to date time now. Once that's done, then I say, "Okay, keep going down the path of whatever saved changes was about to do. So, that way, I don't have to worry about the modified date field anywhere in my application logic. Anytime I call saved changes, saved changes will take a look at anything that it's about to insert or update and make sure that the modified date is set. Let me show you another way to approach this where you can have the benefit of working with strongly typed classes instead of passing in this magic string ModifiedDate hoping that you typed it write and hoping that it's is-- the property name is exactly the same everywhere. I've added a new class into my domain classes called StateInfo. This is really going to be a base class. The StateInfo class has one property called modified date and it's a date time. Then, I went back to all of my classes and I removed that modified date field and instead I made each of the classes inherit from StateInfo. Now, back in the saved changes class, another nice feature of that state entry is that it does give me pointer back to the actual object that particular entry is representing. So, I can get to the object by saying e.Entity, and what I can do here now is instead of looking at every entry, only bother looking at those entries who inherit from StateInfo or where e.Entity is a StateInfo class. So, focus on StateInfo but then, only bother with the added and modified entities and add one more bit of logic here. Instead of working with the actual entry, what I want to do is pull back the entity that the entry represents. So, now, I'm working with the strongly typed class whether it's a customer or an address as long as it's inheriting from StateInfo. And I have to also specifically cast it to a StateInfo. So, now, all of these entities coming back are StateInfos. So, I'll change this, so my code is more clear and I'll call this for each StateInfo because I'm really pulling back the StateInfos. Now, I can't say StateInfo, modified date equals date time now. So, once you have a better understanding of working with the change tracker and the entries and some of the things you can do when you have access to that detailed information. Now, I've got some more efficient code. I've also made my classes cleaner instead of adding modified data every single one I've got the base class with a StateInfo. That StateInfo I might use to add more information to it that I want to share, more properties that I want to share across my classes. So, let me return to the program and I will remove the ModifiedDate. You can run the app and I've put a breakpoint inside of my saved changes, overwrite and you can see I'm about to change the modified date. Up to this point, I hadn't supplied a value for modified date, so .net put in a default date value of 110001, but there's the new date getting applied.

o Building a Repository to Access your DbContext

Here's some of the logic of the console app, if we look at insert customer there is absolutely nothing in here that has anything to do with the user interface of the console app. Now, as a point of comparison, let's take a look at the GetCustomer method, the GetCustomer method has logic that writes information out to the console app. That's interaction with the console, that's interaction with the user, something for the user to see. So, if we go back up to the insert customer, clearly, the using sales model context and adding the customer and saving changes has nothing to do with the UI. What I want to do is have the logic in the UI be focused on user interface activity. It doesn't really make sense for the UI code to have an awareness of where its data is coming from or how it's happening. This follows the pattern called separation of concerns. The UI should only be concerned with the UI, then it makes it easier to maintain user interface related code without having data access logic getting in the way. So, instead, what I want to do is encapsulate the logic into a repository that's dedicated to the data access. For now, I'll just use a single repository class to push my data access code into, but in a more architected application I want a variety of repositories which you can see in my entity framework in the enterprise course. So, I'm going to go ahead and do some refactoring now to move the data access logic into a new repository class. In creating the customer can be a little questionable in this totally demo-wear scenario presumably the values for the customer are coming from the UI, the user has typed in Julie, typed in Lerman and probably put in a date. So, that's going to come from the UI would most like then take those pieces of data, pass them elsewhere and then have that create a customer from the data that came from the UI. I won't go to that extreme, I'll just say-- I'll just create the new customer from the "data" that was entered in the UI, and then I'll do the interaction with the sales model context. So, I've created my simple repository class and for now I've just put it in the console apps project. And here's the class, I've got five methods and I declare and instantiate the context when the class is instantiated. So, the insert method takes in to customer, adds it to the customers of the already existing context instance and then goes ahead and SaveChanges right away. ( Pause ) So, I'm explicitly using ref here although the behavior would be the same without it but I just want to be explicit about that. Back in the program file of the console, I declare and instantiate the repository at start ups, the repository is just available for the lifetime of this app. And then my insert customer method, I'm still creating the customer but instead of instantiating the context, adding the customer in and then saving it, instead, I'm just passing the customer to the insert customer and save method. So, I've run that, when I pass that customer and as a reference and it got inserted, remember the database creates a new ID and pushes it into the customer instance and because it is a reference it's the same instance so that customer ID is populated and then I could go ahead and display the customer ID. DisplayCustomer is really my old GetCustomer method, but I renamed it because, really, all I'm doing here is displaying and what I did there was go ahead and retrieve the customer again and display it's ID. And I just moved my find method on the context into the repository. So, I find that and then write out that name and the birthday. Now, things are working a little bit differently in here because when I use the program I was explicitly using separate contexts, but in this particular scenario I'm actually working with the same context. So, in this particular case where I called find since I'm working with the same context, it didn't need to go back to the database, it found the same customer instance that was already being tracked by the context. Let's take a look at some of these other methods, GetCustomersAndOrders, I'm doing eager loading here and returning a list of customers, plain old GetCustomers just returns a list of the customers. What this means is that my program doesn't need to know anything about entity framework. A little bit of entity framework profiler code here doesn't count, that's just for debugging that's not for my real application. Notice all these using statements that are not even necessary anymore. One of those is system.data.entity and the other is the data layer. So, by using the repository and taking the reliance on entity framework completely out of this piece of my application, this is my user interface. My user interface shouldn't have to know anything about where that data is coming from or where the data is going to or how that's all happening. I name my repository simple because it really is very simple. I'm not taking into account many of the bigger architectural questions here. If we're using a connected application or a disconnected application, some of the patterns you need to follow. Again, I've talked a lot more about these things in my course Entity framework in the enterprise.

o Using your EF Data Layer in a Client App (WPF): Modifying the classes

But I still want to take another repository and use it in something more than a console app. So, I want to show it to you in a clientside apps, so I'll use WPF, I'll show it to you in a little MVC application so that will show you a disconnected application. And I also want you to see using the model in a data service, so WCF data service that will provide OData to an application and that won't use the repository because the data service works directly with the model. WCF data service as an OData are really important if you want to create data to be consumed by mobile apps like Windows 8 application or phone apps. I've enhanced my repository and some of the domain classes and use those along with the data context in this simple WPF app. The app is just this single screen where I have a list of existing customers and then I can select a customer to view or edit their key information, the information that comes in their contact detail record and any orders. I don't have the details of the orders in here, just the fact that they didn't make an order. So, I have three customers in here, myself "Yogi" the bear who doesn't have a lot of information and "Samson" the newfie who you may know, he is my dog who even has his own Twitter account, I cannot edit the existing records, Samson doesn't really have a Facebook page, I thought a Twitter account was enough, and save that, maybe I'll make a little edit here. And then notice I'm editing the contact detail, I'm not even the main record and I can get prompted to save. I can edit or add new orders with the default date and select one of the order sources, these are coming from my ANIMs, save that. I can create a new customer, maybe Yogi's friend BooBoo bear, and let's say, he placed an order also on a 12th and we like to see BooBoo bear so we had him coming in person. So, I'm going to close this and then rerun it again so that you can see that all of these changes are real, they really got stored in the database. So, there is BooBoo bear with the order. There is my change with the new order and also the home phone, Samson's Facebook page. I don't think I did anything to Yogi. So, what I'm working with here is at the top I'm working with a list of customers and the data entry form, I'm working with a graph of a customer, it's related contact detail, and it's collection of orders. I'm retrieving a graph and then I'm saving the graph all in one file swoop. So, let's look at some of the code here. I'm not going to go through it in detail. My goal here is to give you a look at using entity framework inside of a simple WPF app. I'm not going to build up the entire app in front of your eyes. So, in my solution, I have my domain classes that you're familiar with. My data layer that you're familiar with. I've moved my simple repository into its own project and I have made some changes to this classes and to the repository to make them richer for the sake of using any application, and also some of the logic is relevant to the fact that I'm using a client side application. I'm doing all the entity framework work inside of the app instead of using, for example, services on the other end. So, this is a client side application. In a connected application, you can use a long running context but in a disconnected app, you're more likely to be working with context that you instantiate, use your database interaction and then dispose it quickly. You can certain build a repository that's flexible enough to be shared in both scenarios but this little advance for this course. Instead, I've constructed some of the logic to be more specific to the long running context and this WPF example. First, I want to show you something I've done to the domain classes which was I added more logic into my StateInfo to get inherited by my classes. I've done two things here. One is, I've made StateInfo implement INotifyPropertyChange. INotifyPropertyChange it's a really useful interface when you're building WPF app so that when properties do change, you can bubble information back up to the UI. I've also added and isDirty property. Now, notice that I've used one of entity frameworks schema data annotations here called NotMapped, isDirty is a property do need to set and get but it's not one that I want to persist to the data base. So, by default, code first would assume that this property that has both a getter and a setter should be mapped to the data base. So, I'm specifically saying, I don't want it mapped. I can also set that configuration using fluent API. I use the data annotations so that it would be really in your face when I was showing you this class. Property change event handler is part of INotifyPropertyChange. The only thing I added to it was to set my isDirty flag equals true. And the reason I want an isDirty flag is so that if somebody makes an edit to a record and then tries to go to another record, I pop up a massage asking if you want to SaveChanges. And being that I'm in a client side application and working with a long running context, I can almost execute that particular logic using StateInformation from the context. There was one little tricky thing that I wasn't able to implement. So, the isDirty property gave me better control. So, that's why I implemented it. So, now, everyone at the class is implements INotifyPropertyChange because they all inherit from StateInfo. And then, in those classes, I've picked specific properties that will trigger the property changed event, so not all of them. In order to do that, my properties can't be auto implemented where I just have the little get set after them. I have to call on property changed when I change a value. So, you can see that the changing the first name will trigger that, little set isDirty to true changing last name will do that. Changing date of birth will do that. None of the others will trigger isDirty to get set of-- set to true when I change something on the customer. I've also modified some of the properties and contact detail. So, that as we move through. So, that-- so if I make any changes to any of the contact information like the Twitter alias or the Facebook page, that will get flagged as dirty as well. Does it in most critical things I did to the domain classes, I provided a little additional logic to make my application a little nicer. So, those aren't really critical to show you. I've made a change to how my SaveChanges override works. So, I want to show that to you. The first iteration is unchanged but then I explicitly call SaveChanges instead of just returning the result of SaveChanges. I call SaveChanges and then, I want to go through all the StateInfos and change IsDirty back to false and then I go ahead and return whatever the result of SaveChanges is. So, the most interesting changes are in the repository. I've definitely made a lot of changes there.

o Using your EF Data Layer in a Client App (WPF): The tweaked repository

I've made some executive decisions about the repository. I'm still instantiating a context at the start of the repository to be used for the life of the repository and a single repository instance will be used for the life of the app. So, in effect, there's only one context instance at work for the entire life of this application. This is fine for an app of the scenario scope and of course that it's all happening on the client. The context in my app will have direct access to the database, for example, within a company internet. Here what its methods look like now. First, I have a method that will create a new customer and pass it back to the calling code which is coming from the user interface. Then the user will fill out the details of the customer and save it. This takes the onus of constructing a customer out of the UI logic. Also, notice that I'm creating the contact detail for the customer in the method. So, I'm passing back a whole graph. If a user wants to edit a customer, the calling code passes in the idea of the customer that the user selected and the repository passe back the entire graph of the customer, its contact details and its orders. Since, this is in a complex graph, I'm using eager loading for both the contact detail and the orders. GetCustomer ID won't get use by my app but it will just return a customer without the other related objects. GetCustomers returns all the customers in the database and returns them as a list. You saw this in the earlier version of the repository. Customers in memory is useful for disconnected application. Since my contacts is hanging around, it will remember all of the customers that I've queried already. So, there's no need to return to the database for them. This is just for the customer list, not for the individual graphs. This method takes advantage of a Db set property called Local which doesn't ever go to the database. It only returns what the context is tracking. Local returns an observable collection so that the contact starts tracking another entity or stops tracking one, the collection will automatically take that in account. So, if I create a new customer and I used the new customer with contact detail method that adds a customer to my context. So, now, my observable collection will also be aware of that new customer. I don't have to go back to the database to get it. And notice, I have a little bit of logic in here to make sure that I actually have queried for the customers already. So, if I don't have anything in my local customers, I do go and hit that GetCustomers method. In the earlier version of my repository, I've saved at the same time I inserted a new customer. But now, I'm keeping save as its own method. This way, the user can trigger save regardless of whether she created a new customer and relay the data or edits an existing one. So, in the insert customer contact detail method, I don't save to the data base right away just after I've created a new essentially empty customer. I wait until the user clicks the save button. The long running connected context will be aware of inserts and updates. So, we don't need to be explicit about what we're saving. I'm doing a little more than SaveChanges in here. What I'm doing is checking for any added customers that the user never even touch because I don't want to save them into the data base but instead of iterating through the local customers using for each, I'm using a different pattern. If I iterated through the collection of the for each and started to removing things and changing the size of the collection, then the iterator gets all confused. So, instead, the pattern is to-- is to go through using for and to start from the end. My last method is for deleting a current customer. Now, in this one, I'm explicitly calling save at the same time that I'm removing my customer from the context. In the flow of my application, deleting falls out of the scope of the other actions. A user will delete by right clicking a customer in the list and choosing delete and it doesn't make sense for the user to go click a save button after deleting a customer. So, save will take care of my inserts and updates while delete will take care-- deleting and saving that action all at once.

o Using your EF Data Layer in a Client App (WPF): Debugging and Profiling

If we look at the WPF sales project, you'll see that there is no reference here to the data layer to entity framework or system.data.entity, and the code inside of my code behind is only code that's directed towards interaction with the user interface or making calls to the repository. So, what I'll do is I'll put a couple of breakpoints in here I'll debug through the app and point out some of the key logic. As I'm loading the Window, the first thing I do is get a list of the customers from my list box. So, I call the CustomersInMemory method and remember, if there aren't any CustomersInMemory at, that will first go and call my GetCustomers method. And then there's a UI related method for making sure the list box is sorted and then I'm just creating pointers to customer view source and contact detail view source which are classes in my example that will allow me to data binding. And finally, I force the customer list box to point to the first item in the list. When I select the item in the list, I'm hitting an event for selection changed on the list box and the most important bit of code in there is that I'm taking the ID of the selected item and using it to call get customer graph so I'm getting the graph, setting it as my current customer and then I'm calling a method that will take care of all the data binding of the customer, the contact and the orders to the user interface. And so my first item in the list is BooBoo bear and that's what got selected and I pulled back the graph for that and did all the data binding. I'll make an edit to the Twitter alias and save and the code behind calls save in the repository. There aren't any added customers so we just call save changes. I'll create a new customer and in response to that, I call the repository new customer with the contact detail. And remember, that gives me a new customer along with the new contact detail record, there is the constructor for customer. I add that to my context so now my context is aware of this new customer while it's still empty and then I return that customer. Remember that observable collection, I've bound that to the customer list box, notice the customer list box is all ready aware of this new customer because I added it to the context, it's keeping track of what customers are coming in and out of the context. So, I'll go ahead and fill this out and Sampson forgot to buy valentines present for his girlfriend so she's going to become a customer and buy her own dog cookies. And she's going to place her first order today, she telephone that in, she's a really, really clever dog. So, I'll go ahead and save that and my code behind called the Save method again. Now, as I iterate through, I'm on the new record and the entity state of this is added, the context knows that it's added but both the customer and contact detail are dirty, I'm not going to remove it from the context. I'm going to go ahead and save the changes. In my UI code, I've ensured that that new item is selected, that triggered the selection change event so we ran through that again. Now, I've got the profiler running so I wanted you to see that when I called save changes, I had an insert into customers and insert into contact details and the insert into orders. Because I had a graph and the context was tracking everything as it happen, it understood the graph, it understood that everything was added, I'm going to call saved changes, it inserted everything in the graph. I should show you an update so I'm going to go ahead and place another order and I'm just going to e-mail it in cause I'm a little busy working on this plural site course. And while I'm at it, I'm just going to add my mobile phone number so I'll go ahead and save it and now I'm back in entity framework profiler and notice, I don't have another context, okay. Everything that I've been doing so far has been happening on the same context so even the profiler is aware of that. So, here's my latest transaction, I updated the contact details with that new cellphone number and I inserted a new order into the orders. So, the context was keeping track of all of that at the same time and then when I called save changes, it just pushed everything up to the database. And again, because it's a long running context and because the entire application is connected and because a missing data binding, the context is aware of the changes that are being made to my entities as the user is working and interacting with it. So, when it's time to save changes, I don't have to look at what the values and the properties of the form are I don't have to attach orders to customers, the context is aware of all of that information and all of that state information, that's the beauty of being connected.

o Using your EF Data Layer in a Web App (ASP.NET MVC4)

Here's a little MVC for app that's using my domain classes, my repository, well a tweeked version of the repository that we've been working with. And the repository uses entity framework in my data layer. This app list my customers, I can drill into a customer, view and edit their details and also view a list of their orders and even go in and edit anyone of those orders. I can create new orders and notice that this source is coming from my ANIMs in my application so there is the new order. Let me show you a little bit of the code and then we'll debug through this so you can see the interaction between the UI and the code. So, here's my solution and instead of a WPF project, I now have in MVC for project. And I just want to point out that this has no reference to entity framework or system.data.entity or my data layer project. And before I debug through, I do want to show you that I made some changes to the repository. First of all, since I'm going to interact with the orders separately just to keep my code separate and easier to maintain, I put to the repository logic for working with orders into a separate class. One thing I added in here because of the disconnected nature is an update customer method because the customer will be coming from MVC and it's not being changed tracked, I need to make sure that as I'm making the context aware of it, I'd let the context also know that this has been modified otherwise, it'll just consider it unchanged and I won't do anything with the database when I call save changes. I'm explicitly setting the state to modify it when I'm passing in a customer that's to be updated. I'm still using my get customer graph, they won't be using customers in memory because the context isn't sticking around so there won't ever be customers in memory anyway so there's no point using that. Because we're disconnected, I'm not going to have to worry about that weird state of new customers where I may have added an empty customers to the context but the user didn't edit the customer so I don't really want to say that the database that's not going to happen. The user's going to edit on the screen, those values will come to me as a customer and I'll add that customer to the context directly so I don't have to worry about that so I took that extra code out of there. The order repository is similar to customer except I have fewer methods that I need to worry about. I instantiate a context as this repository is instantiated. I have a Save method, I can get order by ID, I can add a new order, I can update an order and I can delete an order. I'm not bothering with getting a list of order cause the only time I need to see a list is as part of a customer graph so I'm seeing the list not editing it but just seeing a simple list on the customer page anyway, I don't have a need for that here. So, I put a whole bunch of breakpoints in the code and I've started the app up and the first thing it's trying to do is create the page where it lists the customers. So, I've hit my repository method that's retreaving the customers. You can see that that had been called by the customer controller's index action result method. And there is the resulting page. I simply just grab the list to the customers and I'm displaying them. So, we go ahead and edit one of these customers. So, if you're not familiar with MVC, notice, there are two methods related to edit. One is when I'm saying I want to edit, the other, I'll hit when I post back from the edit. So, I want to edit and I need to get a customer so I'm going to call the get customer graph method that will hit the repository, pull back the customer graph and then that gets passed on to the view. The markup in the view knew what to do with all the data that were sent to it. So, I'll edit the customer, I'll click on save, this will cause the pause back of the edit screen which is that second edit method and you can see what that's going to do is pass the customer that MVC rendered for us. MVC took all the values from the page and created a customer instance from them and pass that in as a parameter here so here are all the values. Now, I'm going to take that customer and send it to the repository update method and then call Save. And now, I'm going back to my index view and so you can see that full name is now Julie. Let me go back in there retrieving the graph again. So, you can see, nothing is getting stored in memory, I keep going back to the database and pulling information back on to the screen again. And if I create a new order, I'm in the order controller instantiating the order repository and I'm on the create screen. Now, I'm posting back, I'm instantiating a new controller and instantiating a new repository which will instantiate a new context. Now, I'm hitting the post back with the create and you can see that's going to go into the repository and pass the order that came back into my add new order method which calls add and then I save back to the database and there is my save changes overwrite method in my data layer. That's just a quick look at a simple application where I'm using my domain and my repository, I've got everything separated. So, this is a little more architected than a typical demo ware MVC app where you would have all of the entity framework activity directly in the controllers. One last thing to note about this MVC application is that if you look at the profiler, you can see that I have all these different context. I have one context per each select for an update for another select for an update for an insert and again that's the nature of this disconnected application. Where I don't have the long running context so I need to be explicit about the stake of all of my entities as a I'm interacting with the context.

o Accessing your Data from Mobile Apps

While you can use entity framework in your client apps and in the server site code of your web apps, you can't use it directly in totally disconnected applications. Entity frameworks depends on being able to interact directly with the database. So, you can't use it from a phone app or a Window Store app. Totally disconnected apps rely on services to provide their data. A service layer that has access to the database is where you can use entity framework. If you're using Web API or Web Services, you can use code similar to what you just saw in the MVC controllers and that code can interact with the repository or directly against a context if you're not building a big service. Well, Web API can now produce OData. Another way to create OData is to wrap your context inside of a WCF Data Service. But WCF Data Services work directly with the context. So, you won't be using a repository or any kind of intermediary code between them. There's a Pluralsight course on creating WCF data Services. So, I won't go into detail but I do want to show you my model working in a data service. Here's my data service project and you can see I've got a simple app that contains a data service and I have my data layer and my domain classes. The data service project has references to my data layer and my domain classes. Now, this isn't a Data Service project per say. I created a WCF service project because that gave me all the underpinnings that I need and then removed the actual service interface that the template provides and then I added a WCF Data Service item into the project. This service wraps my sales model context. When you consume the service, you're working with OData APIs and then those do all of the querying and updating with entity framework. Unless you want to override some of the service behavior, you won't be writing any entity framework code in the service. Now, data service starts out as completely locked down. I've exposed a little bit of my data through the data service. So, I said it's okay to look at all of the data and its okay to read, write and delete from the customers and pay attention to my note. This is a specific way to make data available. It's usually available for public consumption although there's all kinds of stuff that that you can do to secure it. So, I can explore this service directly in a browser. And the default is that it just shows you what entity sets are available and I made all of them available for reading. I can look at some of the raw data in the browser just so I can make sure everything is working. Here's the customer data. Here's the customer data along with orders for the customer and it's until you get down to the third customer where there's actually some order data. I won't dwell on how to interact with OData. Again, there's a whole course on that. I just want to you to see that it's exposed this way. And once you have it exposed this way, then you can consume it from a disconnected type of an application. For example, here is my first ever attempt at a Window Store Application and I'm not doing much for that. I manage to get it to read from the service. I added a Service Reference that I named CustomerOData and that points to my WCF Data Service. So, if I run my little app and there it is. My application, I manage to get this application to read from my OData service and display a list of my customers and that's as far as I got. As soon as I get this course published I'll be off to watch the Windows 8 Application courses on Pluralsight. I also want to point out that you'll be able to expose OData through Web API. Here's a recent blog post from Microsoft about that. Along with another blog post also from Web Dev about securing access to the OData when you're using the Web API.

o Summary

Well, I hope by now you feel that you've really gotten started with the Entity Framework 5. That and more I'm sure. In case your head is still spinning from my Window Store App, let me remind you where we've been in this module. I did some work with the EDMX Generated Code. First, pushing the Domain Classes into their own project and then adding custom logic to those generated classes. Then, I extended the SaveChanges method to show you how to do some work to your objects before they're stored in the database. Recall that I used the base class with the modified date property and then updated the date as needed in SaveChanges. Then I pulled the data access logic out of the consuls UI code and put it into its own repository to help with separation of concerns. With that in hand, I then used all of this code with some tweaks in a WPF app where we had the benefit of a long running context. And then in an MVC app where you have to pay attention to state because you're constantly working with new context as your Web App Postback. Finally, I showed you how a Data Service can consume your model directly and expose OData. And then there was that Window Store App but we don't need to talk about that anymore.

o References

Thanks for watching this course on getting started with Entity Framework 5. I'm Julie Lerman and here's some more resources that you might find helpful. ( Silence )