Part 1 – Entity Framework's Convention:

Code First enables you to describe a model by using C# or Visual Basic .NET classes. The basic shape of the model is detected by using conventions. Conventions are sets of rules that are used to automatically configure a conceptual model based on class definitions when working with Code First. The conventions are defined in the System.Data.Entity.ModelConfiguration.Conventions namespace.

Primary Key Convention

Code First infers that a property is a primary key if a property on a class is named "ID" (not case sensitive), or the class name followed by "ID". If the type of the primary key property is numeric or GUID it will be configured as an identity column.

```
public class Department
{
    // Primary key
    public int DepartmentID { get; set; }

    . . .

}
```

Relationship Convention

In the Entity Framework, navigation properties provide a way to navigate a relationship between two entity types. Every object can have a navigation property for every relationship in which it participates. Navigation properties allow you to navigate and manage relationships in both directions, returning either a reference object (if the multiplicity is either one or zero-or-one) or a collection (if the multiplicity is many). Code First infers relationships based on the navigation properties defined on your types.

In addition to navigation properties, we recommend that you include foreign key properties on the types that represent dependent objects. Any property with the same data type as the principal primary key property and with a name that follows one of the following formats represents a foreign key for the relationship: '<navigation property name><principal primary key property name>', '<principal class name><primary key property name>', or '<principal primary key property name>'. If multiple matches are found then precedence is given in the order listed above. Foreign key detection is not case sensitive. When a foreign key property is detected, Code First infers the multiplicity of the relationship based on the nullability of the foreign key. If the property is nullable then the relationship is registered as optional; otherwise the relationship is registered as required.

If a foreign key on the dependent entity is not nullable, then Code First sets cascade delete on the relationship. If a foreign key on the dependent entity is nullable, Code First does not set cascade delete on the relationship, and when the principal is deleted the foreign key will be set to null. The multiplicity and cascade delete behavior detected by convention can be overridden by using the fluent API.

In the following example the navigation properties and a foreign key are used to define the relationship between the Department and Course classes.

```
public class Department
{
    // Primary key
    public int DepartmentID { get; set; }
    public string Name { get; set; }

    // Navigation property
    public virtual ICollection<Course> Courses { get; set; }
}

public class Course
{
    // Primary key
    public int CourseID { get; set; }

    public string Title { get; set; }
    public int Credits { get; set; }

    // Foreign key
    public int DepartmentID { get; set; }

    // Navigation properties
    public virtual Department Department { get; set; }
}
```

Note:*If you have multiple relationships between the same types (for example, suppose you define the* **Person** *and* **Book** *classes, where the* **Person** *class contains the* **ReviewedBooks** *and* **AuthoredBooks** *navigation properties and the* **Book** *class contains the* **Author** *and* **Reviewer** *navigation properties) you need to manually configure the relationships by using Data Annotations or the fluent API.*

Complex Types Convention

When Code First discovers a class definition where a primary key cannot be inferred, and no primary key is registered through data annotations or the fluent API, then the type is automatically registered as a complex type. Complex type detection also requires that the type does not have properties that reference entity types and is not referenced from a collection property on another type. Given the following class definitions Code First would infer that Details is a complex type because it has no primary key.

```
public partial class OnsiteCourse : Course
{
    public OnsiteCourse()
    {
        Details = new Details();
    }
```

```
    public Details Details { get; set; }
}

public class Details
{
    public System.DateTime Time { get; set; }
    public string Location { get; set; }
    public string Days { get; set; }
}
```

Connection String Convention

1- Use Code First with connection by convention:

If you have not done any other configuration in your application, then calling the parameterless constructor on DbContext will cause DbContext to run in Code First mode with a database connection created by convention. For example:

```
namespace Demo.EF
{
    public class BloggingContext : DbContext
    {
        public BloggingContext()
        // C# will call base class parameterless constructor by default
        {
        }
    }
}
```

In this example DbContext uses the namespace qualified name of your derived context class—Demo.EF.BloggingContext—as the database name and creates a connection string for this database using either SQL Express or LocalDb. If both are installed, SQL Express will be used.

2- Use Code First with connection by convention and specified database name:

If you have not done any other configuration in your application, then calling the string constructor on DbContext with the database name you want to use will cause DbContext to run in Code First mode with a database connection created by convention to the database of that name. For example:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("BloggingDatabase")
    {
    }
}
```

In this example DbContext uses “BloggingDatabase” as the database name and creates a connection string for this database using LocalDb.

3- Use Code First with connection string in app.config/web.config file:

You may choose to put a connection string in your app.config or web.config file. For example:

```
 <configuration>
  <connectionStrings>
    <add name="BloggingCompactDatabase"
    connectionString="Data Source=.;Initial Catalog=BloggingDatabase;Integrated
Security=True"/>
  </connectionStrings>
</configuration>
```

If the name of the connection string matches the name of your context (either with or without namespace qualification) then it will be found by DbContext when the parameterless constructor is used. If the connection string name is different from the name of your context then you can tell DbContext to use this connection in Code First mode by passing the connection string name to the DbContext constructor. For example:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("BloggingCompactDatabase")
    {
    }
}
```

Alternatively, you can use the form “name=<connection string name>” for the string passed to the DbContext constructor. For example:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("name=BloggingCompactDatabase")
    {
    }
}
```

This form makes it explicit that you expect the connection string to be found in your config file. An exception will be thrown if a connection string with the given name is not found.

Removing Conventions

You can remove any of the conventions defined in the System.Data.Entity.ModelConfiguration.Conventions namespace. The following example removes **PluralizingTableNameConvention**.

```
public class SchoolEntities : DbContext
{
    . . .
```

```
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Configure Code First to ignore PluralizingTableName convention
        // If you keep this convention, the generated tables
        // will have pluralized names.
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

Custom Conventions

Custom conventions are supported in EF6 onwards. Sometimes these default conventions are not ideal for your model, and you have to work around them by configuring many individual entities using Data Annotations or the Fluent API. Custom Code First Conventions let you define your own conventions that provide configuration defaults for your model.

Let's start by defining a simple model that we can use with our conventions. Add the following classes to your project.

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

public class ProductContext : DbContext
{
    static ProductContext()
    {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<Produc
tContext>());
    }

    public DbSet<Product> Products { get; set; }
}

public class Product
{
    public int Key { get; set; }
    public string Name { get; set; }
    public decimal? Price { get; set; }
    public DateTime? ReleaseDate { get; set; }
    public ProductCategory Category { get; set; }
}

public class ProductCategory
{
    public int Key { get; set; }
    public string Name { get; set; }
    public List<Product> Products { get; set; }
}
```

Introducing Custom Conventions

Let's write a convention that configures any property named Key to be the primary key for its entity type.

Conventions are enabled on the model builder, which can be accessed by overriding OnModelCreating in the context. Update the ProductContext class as follows:

```
public class ProductContext : DbContext
{
    static ProductContext()
    {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<Produc
tContext>());
    }

    public DbSet<Product> Products { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Properties()
                    .Where(p => p.Name == "Key")
                    .Configure(p => p.IsKey());
    }
}
```

Now, any property in our model named Key will be configured as the primary key of whatever entity its part of.

We could also make our conventions more specific by filtering on the type of property that we are going to configure:

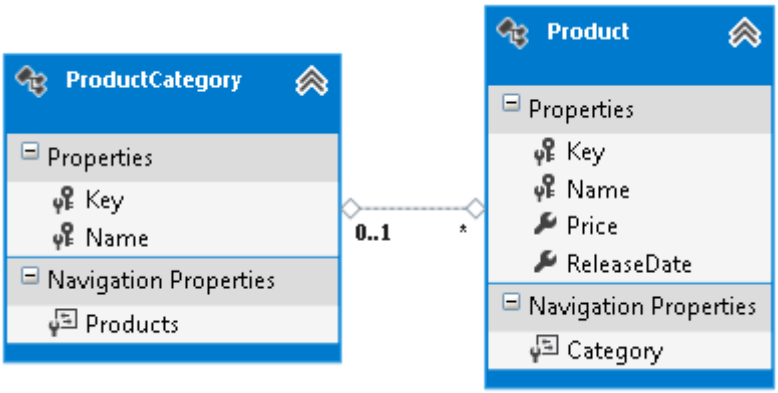
```
modelBuilder.Properties<int>()
            .Where(p => p.Name == "Key")
            .Configure(p => p.IsKey());
```

This will configure all properties called Key to be the primary key of their entity, but only if they are an integer.

An interesting feature of the IsKey method is that it is additive. Which means that if you call IsKey on multiple properties and they will all become part of a composite key. The one caveat for this is that when you specify multiple properties for a key you must also specify an order for those properties. You can do this by calling the HasColumnOrder method like below:

```
modelBuilder.Properties<int>()
            .Where(x => x.Name == "Key")
            .Configure(x => x.IsKey().HasColumnOrder(1));

modelBuilder.Properties()
            .Where(x => x.Name == "Name")
            .Configure(x => x.IsKey().HasColumnOrder(2));
```

This code will configure the types in our model to have a composite key consisting of the int Key column and the string Name column. If we view the model in the designer it would look like this:



Another example of property conventions is to configure all DateTime properties in my model to map to the datetime2 type in SQL Server instead of datetime. You can achieve this with the following:

```
modelBuilder.Properties<DateTime>()
            .Configure(c => c.HasColumnType("datetime2"));
```

Convention Classes

Another way of defining conventions is to use a Convention Class to encapsulate your convention. When using a Convention Class then you create a type that inherits from the Convention class in the System.Data.Entity.ModelConfiguration.Conventions namespace.

We can create a Convention Class with the datetime2 convention that we showed earlier by doing the following:

```
public class DateTime2Convention : Convention
{
    public DateTime2Convention()
    {
        this.Properties<DateTime>()
            .Configure(c => c.HasColumnType("datetime2"));
    }
}
```

To tell EF to use this convention you add it to the Conventions collection in OnModelCreating, which if you've been following along with the walkthrough will look like this:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Properties<int>()
                .Where(p => p.Name.EndsWith("Key"))
                .Configure(p => p.IsKey());
```

```
    modelBuilder.Conventions.Add(new DateTime2Convention());
}
```

As you can see we add an instance of our convention to the conventions collection. Inheriting from Convention provides a convenient way of grouping and sharing conventions across teams or projects. You could, for example, have a class library with a common set of conventions that all of your organizations projects use.

Execution Order

Conventions operate in a last wins manner, the same as the Fluent API. What this means is that if you write two conventions that configure the same option of the same property, then the last one to execute wins. As an example, in the code below the max length of all strings is set to 500 but we then configure all properties called Name in the model to have a max length of 250.

```
modelBuilder.Properties<string>()
            .Configure(c => c.HasMaxLength(500));

modelBuilder.Properties<string>()
            .Where(x => x.Name == "Name")
            .Configure(c => c.HasMaxLength(250));
```

Because the convention to set max length to 250 is after the one that sets all strings to 500, all the properties called Name in our model will have a MaxLength of 250 while any other strings, such as descriptions, would be 500. Using conventions in this way means that you can provide a general convention for types or properties in your model and then overide them for subsets that are different.

Part 2 – Entity Framework's Configuration:

1-DataAnnotation: EF Code-First provides a set of DataAnnotation attributes, which you can apply on your domain classes and properties. You have to include System.ComponentModel.DataAnnotations namespace to use DataAnnotation attributes. DataAnnotation basically includes attributes for server side validations and database related attributes.

Validation Attributes:

Annotation Attribute	Description
Required	The Required annotation will force EF (and MVC) to ensure that property has data in it.
MinLength	MinLength annotation validates property whether it has minimum length of array or string.
MaxLength	MaxLength annotation maximum length of property which in turn sets the maximum length of column in the database

StringLength	Specifies the minimum and maximum length of characters that are allowed in a data field.

Database Schema related Attributes:

Annotation Attribute	Description
Table	Specify name of the DB table which will be mapped with the class
Column	Specify column name and datatype which will be mapped with the property
Key	Mark property as EntityKey which will be mapped to PK of related table.
ComplexType	Mark the class as complex type in EF.
ForeignKey	Specify Foreign key property for Navigation property
NotMapped	Specify that property will not be mapped with database
ConcurrencyCheck	ConcurrencyCheck annotation allows you to flag one or more properties to be used for concurrency checking in the database when a user edits or deletes an entity.
DatabaseGenerated	DatabaseGenerated attribute specifies that property will be mapped to Computed column of the database table. So the property will be read-only property. It can also be used to map the property to identity column (auto incremental column).

DataAnnotation example:

```
    [Table("StudentInfo")]
    public class Student
    {
        public Student(){ }

        [Key]
        public int SID { get; set; }

        [Required(ErrorMessage="Student Name is Required" )]
        [Column("Name", TypeName="ntext")]
        [MaxLength(20), MinLength(2, ErrorMessage="Student name can
not be 2 character or less")]
        public string StudentName { get; set; }

        [NotMapped]
        public int? Age { get; set; }

        [ConcurrencyCheck()]
```

```
        public Byte[] LastModifiedTimestamp { get; set; }

        public int? MathScore { get; set; }

        public int? ScienceScore { get; set; }
```

Creating a Custom Data Annotation Attribute

Let's assume that you wish to validate FirstName and LastName properties so that their length must be between 5 and 50 characters and they should contain only letters. You will build your own data annotation to accomplish this task. So, add a new class to the project and name it NameAttribute. The NameAttribute class inherits from the ValidationAttribute base class and is shown below:

```
public class NameAttribute:ValidationAttribute

{

  private int intMinLength;

  public NameAttribute(int minlength)

  {

    this.intMinLength = minlength;

  }

  public override bool IsValid(object value)

  {

    string pattern = @"^[a-zA-Z]{" + intMinLength + ",50}$";

    bool result = Regex.IsMatch(value.ToString(),pattern,
RegexOptions.IgnoreCase);

    if (result)

    {

      return true;

    }

    else

    {

      return false;

    }
```

```
    }
}
```

The NameAttribute class has a private member intMinLength that stores the minimum length of a name (you will use 5 in this example). The NameAttribute class inherits from the ValidationAttribute base class and overrides the IsValid() method. The IsValid() method accepts a value that is to be validated and returns true if the value is valid, false otherwise. The validation is performed using a regular expression. The regular expression ensures that the value contains only uppercase or lowercase letters and that its length is between intMinLength and 50. The IsMatch() method of the Regex class performs the pattern matching and returns true if the pattern is found in the source string. The IsMatch() returns false if the pattern cannot be found. The result is then returned to the caller.

Using a Custom Data Annotation Attribute

Once the custom data annotation class NameAttribute is ready it can be used on the data model for performing the validations. Since Employee class is an EF designer generated class you will need to create a metadata class and apply data annotations to its properties. The Employee class that does this for you is shown below:

```
public class Employee
{
    [Required]
    [Name(5,ErrorMessage="FirstName must be between 5 and 50 characters")]
    public string FirstName { get; set; }

    [Required]
    [Name(5, ErrorMessage = "LastName must be between 5 and 50
characters")]
    public string LastName { get; set; }
}
```

The Employee class defines two properties - FirstName and LastName. Both the properties are decorated with [Required] and [Name] data annotation attributes. Notice how [Name] is used to specify the minimum length of 5 and an ErrorMessage. The ErrorMessage property comes to the NameAttribute class from the base class ValidationAttribute. Currently the Employee class is not associated with the Employee model class in any way.

2-Fluent API - Configuring/Mapping Properties & Types: The code first fluent API is most commonly accessed by overriding the OnModelCreating method on your derived DbContext. The following samples are designed to show how to do various tasks with the fluent api and allow you to copy the code out and customize it to suit your model, if you wish to see the model that they can be used with as-is then it is provided at the end of this part.

Default Schema (EF6 onwards)

Starting with EF6 you can use the HasDefaultSchema method on DbModelBuilder to specify the database schema to use for all tables, stored procedures, etc. This default setting will be overridden for any objects that you explicitly configure a different schema for.

```
modelBuilder.HasDefaultSchema("sales");
```

Property Mapping

The Property method is used to configure attributes for each property belonging to an entity or complex type. The Property method is used to obtain a configuration object for a given property. The options on the configuration object are specific to the type being configured; IsUnicode is available only on string properties for example.

Configuring a Primary Key

The Entity Framework convention for primary keys is:

1. Your class defines a property whose name is "ID" or "Id"

2. or a class name followed by "ID" or "Id"

To explicitly set a property to be a primary key, you can use the HasKey method. In the following example, the HasKey method is used to configure the InstructorID primary key on the OfficeAssignment type.

```
modelBuilder.Entity<OfficeAssignment>().HasKey(t => t.InstructorID);
```

Configuring a Composite Primary Key

The following example configures the DepartmentID and Name properties to be the composite primary key of the Department type.

```
modelBuilder.Entity<Department>().HasKey(t => new { t.DepartmentID, t.Name
});
```

Switching off Identity for Numeric Primary Keys

The following example sets the DepartmentID property to System.ComponentModel.DataAnnotations.DatabaseGeneratedOption.None to indicate that the value will not be generated by the database.

```
modelBuilder.Entity<Department>().Property(t => t.DepartmentID)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
```

Specifying the Maximum Length on a Property

In the following example, the Name property should be no longer than 50 characters. If you make the value longer than 50 characters, you will get a DbEntityValidationException exception. If Code First creates a database from this model it will also set the maximum length of the Name column to 50 characters.

```
modelBuilder.Entity<Department>().Property(t => t.Name).HasMaxLength(50);
```

Configuring the Property to be Required

In the following example, the Name property is required. If you do not specify the Name, you will get a DbEntityValidationException exception. If Code First creates a database from this model then the column used to store this property will usually be non-nullable.

Note: *In some cases it may not be possible for the column in the database to be non-nullable even though the property is required. For example, when using a TPH inheritance strategy data for multiple types is stored in a single table. If a derived type includes a required property the column cannot be made non-nullable since not all types in the hierarchy will have this property.*

```
modelBuilder.Entity<Department>().Property(t => t.Name).IsRequired();
```

Specifying Not to Map a CLR Property to a Column in the Database

The following example shows how to specify that a property on a CLR type is not mapped to a column in the database.

```
modelBuilder.Entity<Department>().Ignore(t => t.Budget);
```

Mapping a CLR Property to a Specific Column in the Database

The following example maps the Name CLR property to the DepartmentName database column.

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .HasColumnName("DepartmentName");
```

Renaming a Foreign Key That Is Not Defined in the Model

If you choose not to define a foreign key on a CLR type, but want to specify what name it should have in the database, do the following:

```
modelBuilder.Entity<Course>()
    .HasRequired(c => c.Department)
    .WithMany(t => t.Courses)
    .Map(m => m.MapKey("ChangedDepartmentID"));
```

Configuring whether a String Property Supports Unicode Content

By default strings are Unicode (nvarchar in SQL Server). You can use the IsUnicode method to specify that a string should be of varchar type.

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .IsUnicode(false);
```

Configuring the Data Type of a Database Column

The HasColumnType method enables mapping to different representations of the same basic type. Using this method does not enable you to perform any conversion of the data at run time. Note that IsUnicode is the preferred way of setting columns to varchar, as it is database agnostic.

```
modelBuilder.Entity<Department>()
    .Property(p => p.Name)
    .HasColumnType("varchar");
```

Configuring Properties on a Complex Type

There are two ways to configure scalar properties on a complex type.

You can call Property on ComplexTypeConfiguration.

```
modelBuilder.ComplexType<Details>()
    .Property(t => t.Location)
    .HasMaxLength(20);
```

You can also use the dot notation to access a property of a complex type.

```
modelBuilder.Entity<OnsiteCourse>()
    .Property(t => t.Details.Location)
    .HasMaxLength(20);
```

Configuring a Property to Be Used as a Concurrency Token

To specify that a property in an entity represents a concurrency token, you can use either the ConcurrencyCheck attribute or the IsConcurrencyToken method.

```
modelBuilder.Entity<OfficeAssignment>()
    .Property(t => t.Timestamp)
    .IsConcurrencyToken();
```

Type Mapping

Specifying That a Class Is a Complex Type

By convention, a type that has no primary key specified is treated as a complex type. There are some scenarios where Code First will not detect a complex type (for

example, if you do have a property called ID, but you do not mean for it to be a primary key). In such cases, you would use the fluent API to explicitly specify that a type is a complex type.

```
modelBuilder.ComplexType<Details>();
```

Specifying Not to Map a CLR Entity Type to a Table in the Database

The following example shows how to exclude a CLR type from being mapped to a table in the database.

```
modelBuilder.Ignore<OnlineCourse>();
```

Mapping an Entity Type to a Specific Table in the Database

All properties of Department will be mapped to columns in a table called t_ Department.

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department");
```

You can also specify the schema name like this:

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department", "school");
```

Mapping Properties of an Entity Type to Multiple Tables in the Database (Entity Splitting)

Entity splitting allows the properties of an entity type to be spread across multiple tables. In the following example, the Department entity is split into two tables: Department and DepartmentDetails. Entity splitting uses multiple calls to the Map method to map a subset of properties to a specific table.

```
modelBuilder.Entity<Department>()
    .Map(m =>
    {
        m.Properties(t => new { t.DepartmentID, t.Name });
        m.ToTable("Department");
    })
    .Map(m =>
    {
        m.Properties(t => new { t.DepartmentID, t.Administrator, t.StartDat
e, t.Budget });
        m.ToTable("DepartmentDetails");
    });
```

Mapping Multiple Entity Types to One Table in the Database (Table Splitting)

The following example maps two entity types that share a primary key to one table.

```
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);
```

```
modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);

modelBuilder.Entity<Instructor>().ToTable("Instructor");

modelBuilder.Entity<OfficeAssignment>().ToTable("Instructor");
```

3-Fluent API - Configuring Relationships

When configuring a relationship with the fluent API, you start with the EntityTypeConfiguration instance and then use the HasRequired, HasOptional, or HasMany method to specify the type of relationship this entity participates in. The HasRequired and HasOptional methods take a lambda expression that represents a reference navigation property. The HasMany method takes a lambda expression that represents a collection navigation property. You can then configure an inverse navigation property by using the WithRequired, WithOptional, and WithMany methods. These methods have overloads that do not take arguments and can be used to specify cardinality with unidirectional navigations.

You can then configure foreign key properties by using the HasForeignKey method. This method takes a lambda expression that represents the property to be used as the foreign key.

Configuring a Required-to-Optional Relationship (One-to-Zero-or-One)

The following example configures a one-to-zero-or-one relationship. The OfficeAssignment has the InstructorID property that is a primary key and a foreign key, because the name of the property does not follow the convention the HasKey method is used to configure the primary key.

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

// Map one-to-zero or one relationship
modelBuilder.Entity<OfficeAssignment>()
    .HasRequired(t => t.Instructor)
    .WithOptional(t => t.OfficeAssignment);
```

Configuring a Relationship Where Both Ends Are Required (One-to-One)

In most cases the Entity Framework can infer which type is the dependent and which is the principal in a relationship. However, when both ends of the relationship are required or both sides are optional the Entity Framework cannot identify the dependent and principal. When both ends of the relationship are required, use WithRequiredPrincipal or WithRequiredDependent after the HasRequired method.

When both ends of the relationship are optional, use WithOptionalPrincipal or WithOptionalDependent after the HasOptional method.

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);
```

Configuring a Many-to-Many Relationship

The following code configures a many-to-many relationship between the Course and Instructor types. In the following example, the default Code First conventions are used to create a join table. As a result the CourseInstructor table is created with Course_CourseID and Instructor_InstructorID columns.

```
modelBuilder.Entity<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
```

If you want to specify the join table name and the names of the columns in the table you need to do additional configuration by using the Map method. The following code generates the CourseInstructor table with CourseID and InstructorID columns.

```
modelBuilder.Entity<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
    .Map(m =>
    {
        m.ToTable("CourseInstructor");
        m.MapLeftKey("CourseID");
        m.MapRightKey("InstructorID");
    });
```

Configuring a Relationship with One Navigation Property

A one-directional (also called unidirectional) relationship is when a navigation property is defined on only one of the relationship ends and not on both. By convention, Code First always interprets a unidirectional relationship as one-to-many. For example, if you want a one-to-one relationship between Instructor and OfficeAssignment, where you have a navigation property on only the Instructor type, you need to use the fluent API to configure this relationship.

```
// Configure the primary Key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
```

```
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal();
```

Enabling Cascade Delete

You can configure cascade delete on a relationship by using the WillCascadeOnDelete method. If a foreign key on the dependent entity is not nullable, then Code First sets cascade delete on the relationship. If a foreign key on the dependent entity is nullable, Code First does not set cascade delete on the relationship, and when the principal is deleted the foreign key will be set to null.

You can remove these cascade delete conventions by using:

modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>()
modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>()

The following code configures the relationship to be required and then disables cascade delete.

```
modelBuilder.Entity<Course>()
    .HasRequired(t => t.Department)
    .WithMany(t => t.Courses)
    .HasForeignKey(d => d.DepartmentID)
    .WillCascadeOnDelete(false);
```

Configuring a Composite Foreign Key

If the primary key on the Department type consisted of DepartmentID and Name properties, you would configure the primary key for the Department and the foreign key on the Course types as follows:

```
// Composite primary key
modelBuilder.Entity<Department>()
.HasKey(d => new { d.DepartmentID, d.Name });

// Composite foreign key
modelBuilder.Entity<Course>()
    .HasRequired(c => c.Department)
    .WithMany(d => d.Courses)
    .HasForeignKey(d => new { d.DepartmentID, d.DepartmentName });
```

Configuring a Foreign Key Name That Does Not Follow the Code First Convention

If the foreign key property on the Course class was called SomeDepartmentID instead of DepartmentID, you would need to do the following to specify that you want SomeDepartmentID to be the foreign key:

```
modelBuilder.Entity<Course>()
          .HasRequired(c => c.Department)
          .WithMany(d => d.Courses)
          .HasForeignKey(c => c.SomeDepartmentID);
```

Part 3 – Entity Framework's Tricks:

1-DbSet.AddRange & DbSet.RemoveRange:

DbSet in EF 6 has introduced new methods AddRange & RemoveRange. DbSet.AddRange adds collection(IEnumerable) of entities to the DbContext, so you don't have to add each entity individually.

```
    IList<Student> newStudents = new List<Student>();
    newStudents.Add(new Student() { StudentName = "Student1 by
addrange" });
    newStudents.Add(new Student() { StudentName = "Student2 by
addrange" });
    newStudents.Add(new Student() { StudentName = "Student3 by
addrange" });

    using (var context = new SchoolDBEntities())
    {
        context.Students.AddRange(newStudents);
        context.SaveChanges();
    }
```

Similarly, DbSet.RemoveRange is used to remove collection of entities from DbSet.

```
    IList<Student> existingStudents = …..

    using (var context = new SchoolDBEntities())
    {
        context.Students.RemoveRange(existingStudents);
            context.SaveChanges();
    }
```

2-Async query and Save:

You can take advantage of asynchronous execution of .net 4.5 with entity framework. EF 6 has a ability to execute a query and command asynchronously using DbContext.

Let's see how to execute asynchronous query first and then we will see an asynchronous call to context.SaveChanges.

Asynchronous Query:

```
    private static async Task<Student> GetStudent()
    {
        Student student = null;

        using (var context = new SchoolDBEntities())
        {
            Console.WriteLine("Start GetStudent...");

            student = await (context.Students.Where(s => s.StudentID
== 1).FirstOrDefaultAsync<Student>());

            Console.WriteLine("Finished GetStudent...");

        }

        return student;
    }
```

As you can see in the above code, GetStudent method is marked with async to make it asynchronous. Return type of asynchrounous method must be Task. GetStudent returns object of Student entity so return type must be Task<Student>.

Also, query is marked with await, this is a freed calling thread to do something else until it executes the query and returns the data. We have used FirstOrDefaultAsync extension method of System.Data.Entity, you may use other extension methods appropriately, such as SingleOrDefaultAsync, ToListAsyn etc.

Asynchronous Save:

You can call context.SaveChanges asynchronously the same way as async query:

```
    private static async Task SaveStudent(Student editedStudent)
```

```
    {

        using (var context = new SchoolDBEntities())
        {
            context.Entry(editedStudent).State =
EntityState.Modified;

            Console.WriteLine("Start SaveStudent...");

            int x = await (context.SaveChangesAsync());

            Console.WriteLine("Finished SaveStudent...");
        }

    }
```

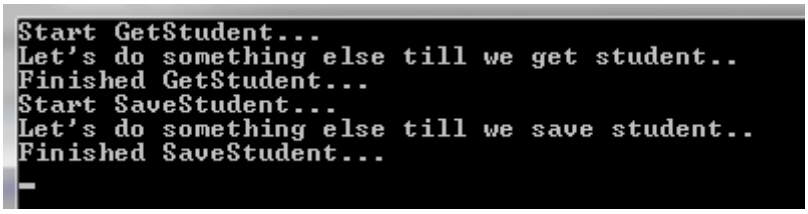
Getting async query result:

You can get the result when asynchronous using wait method as below:

```
    public static void AsyncQueryAndSave()
    {
        var student = GetStudent();

        Console.WriteLine("Let's do something else till we get
student..");

        student.Wait();


        var studentSave = SaveStudent(student.Result);

        Console.WriteLine("Let's do something else till we get
student.." );

        studentSave.Wait();

    }
```

As shown in the code above, we call async method GetStudent in the usual way and store the reference in the variable student. Then we call student.wait(), this means the calling thread should wait until the asynchronous method completes, so we can do another process, until we get result from the asynchronous method.

The code shown above will have following output:



Part 4 -Difference between Lazy Loading and Eager Loading:

1-Lazy/Deferred Loading

In case of lazy loading, related objects (child objects) are not loaded automatically with its parent object until they are requested. By default LINQ supports lazy loading.

For Example:

```
1. var query = context.Categories.Take(3); // Lazy loading
2.
3. foreach (var Category in query)
4. {
5.  Console.WriteLine(Category.Name);
6.  foreach (var Product in Category.Products)
7.  {
8.  Console.WriteLine(Product.ProductID);
9.  }
10.   }
```

Generated SQL Query will be:

```
1. SELECT TOP (3)
2. [c].[CatID] AS [CatID],
3. [c].[Name] AS [Name],
4. [c].[CreatedDate] AS [CreatedDate]
5. FROM [dbo].[Category] AS [c]
6. GO
7.
8. -- Region Parameters
9. DECLARE @EntityKeyValue1 Int = 1
10.  -- EndRegion
```

```
SELECT
[Extent1].[ProductID] AS [ProductID],
[Extent1].[Name] AS [Name],
[Extent1].[UnitPrice] AS [UnitPrice],
[Extent1].[CatID] AS [CatID],
[Extent1].[EntryDate] AS [EntryDate],
[Extent1].[ExpiryDate] AS [ExpiryDate]
FROM [dbo].[Product] AS [Extent1]
WHERE [Extent1].[CatID] = @EntityKeyValue1
GO

-- Region Parameters
DECLARE @EntityKeyValue1 Int = 2
-- EndRegion
SELECT
[Extent1].[ProductID] AS [ProductID],
[Extent1].[Name] AS [Name],
[Extent1].[UnitPrice] AS [UnitPrice],
[Extent1].[CatID] AS [CatID],
[Extent1].[EntryDate] AS [EntryDate],
[Extent1].[ExpiryDate] AS [ExpiryDate]
FROM [dbo].[Product] AS [Extent1]
WHERE [Extent1].[CatID] = @EntityKeyValue1
GO

-- Region Parameters
DECLARE @EntityKeyValue1 Int = 3
-- EndRegion
SELECT
[Extent1].[ProductID] AS [ProductID],
[Extent1].[Name] AS [Name],
[Extent1].[UnitPrice] AS [UnitPrice],
[Extent1].[CatID] AS [CatID],
[Extent1].[EntryDate] AS [EntryDate],
[Extent1].[ExpiryDate] AS [ExpiryDate]
FROM [dbo].[Product] AS [Extent1]
WHERE [Extent1].[CatID] = @EntityKeyValue1
```

In above example, you have 4 SQL queries which means calling the database 4 times, one for the Categories and three times for the Products associated to the Categories. In this way, child entity is populated when it is requested.

You can turn off the lazy loading feature by setting LazyLoadingEnabled property of the ContextOptions on context to false. Now you can fetch the related objects with the parent object in one query itself.

```
context.ContextOptions.LazyLoadingEnabled = false;
```

2-Eager loading

In case of eager loading, related objects (child objects) are loaded automatically with its parent object. To use Eager loading you need to use Include() method.

For Example

```
var query = context.Categories.Include("Products").Take(3); // Eager loading

 foreach (var Category in query)
 {
 Console.WriteLine(Category.Name);
 foreach (var Product in Category.Products)
 {
 Console.WriteLine(Product.ProductID);
 }
  }
```

Generated SQL Query will be

```
SELECT [Project1].[CatID] AS [CatID],
 [Project1].[Name] AS [Name],
 [Project1].[CreatedDate] AS [CreatedDate],
 [Project1].[C1] AS [C1],
 [Project1].[ProductID] AS [ProductID],
 [Project1].[Name1] AS [Name1],
 [Project1].[UnitPrice] AS [UnitPrice],
 [Project1].[CatID1] AS [CatID1],
 [Project1].[EntryDate] AS [EntryDate],
```

```
10.   [Project1].[ExpiryDate] AS [ExpiryDate]
11.   FROM (SELECT
12.   [Limit1].[CatID] AS [CatID],
13.   [Limit1].[Name] AS [Name],
14.   [Limit1].[CreatedDate] AS [CreatedDate],
15.   [Extent2].[ProductID] AS [ProductID],
16.   [Extent2].[Name] AS [Name1],
17.   [Extent2].[UnitPrice] AS [UnitPrice],
18.   [Extent2].[CatID] AS [CatID1],
19.   [Extent2].[EntryDate] AS [EntryDate],
20.   [Extent2].[ExpiryDate] AS [ExpiryDate],
21.   CASE WHEN ([Extent2].[ProductID] IS NULL) THEN CAST(NULL AS int)
22.  ELSE 1 END AS [C1]
23.  FROM (SELECT TOP (3) [c].[CatID] AS [CatID], [c].[Name] AS [Name],
     [c].[CreatedDate] AS [CreatedDate]
24.   FROM [dbo].[Category] AS [c] )
25.  AS [Limit1]
26.   LEFT OUTER JOIN [dbo].[Product] AS [Extent2]
27.  ON [Limit1].[CatID] = [Extent2].[CatID]) AS [Project1]
28.   ORDER BY [Project1].[CatID] ASC, [Project1].[C1] ASC
```

In above example, you have only one SQL queries which means calling the database only one time, for the Categories and the Products associated to the Categories. In this way, child entity is populated with parent entity.

Part 5 -Understanding Inheritance in Entity Framework:

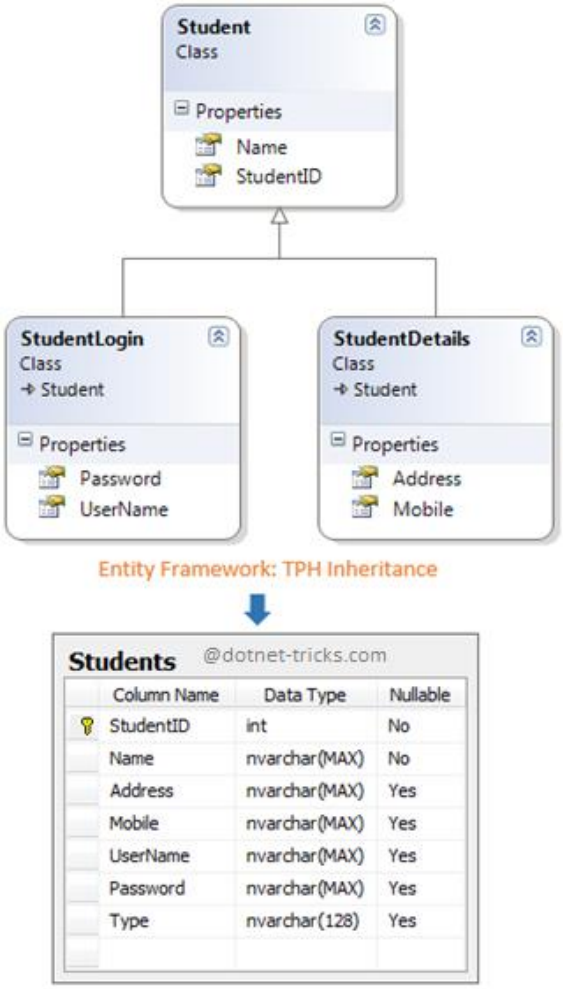
Inheritance in the Entity Framework is similar to inheritance for classes in C#. In Entity Framework, you can map an inheritance hierarchy to single or multiple database tables based on your requirements. In this article, you will learn how to map your inheritance hierarchy to database tables in SQL Server.

Types of inheritance in Entity Framework

Entity Framework supports three types of inheritances as given below-

1. Table-per-Hierarchy (TPH)

The TPH inheritance states that all entities, in a hierarchy of entities, are mapped to a single table in storage schema. It means, there is only one table in database and different Entity types in Entity model that inherits from a base Entity are mapped to that table.



C# Implementation Code for TPH

```
1. public class Student
2. {
3.  public int StudentID { get; set; }
4.  public string Name { get; set; }
5. }
6.
7. public class StudentDetails : Student
8. {
9.  public string Address { get; set; }
10.  public string Mobile { get; set; }
11. }
12.
13. public class StudentLogin : Student
14. {
15.  public string UserName { get; set; }
```

```
16.  public string Password { get; set; }
17. }
```

Entity Framework Code First Mapping for TPH

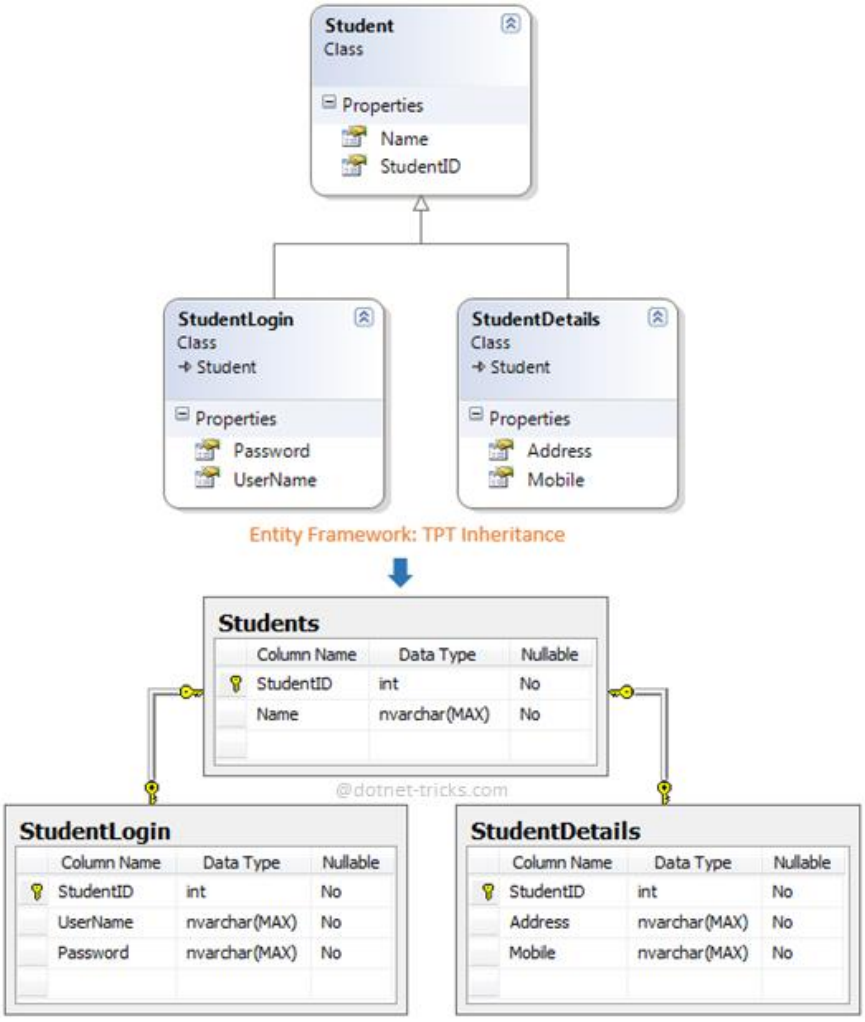
```
18. modelBuilder.Entity<Student>()
19.   .Map<StudentDetails>(m                                    =>
    m.Requires("Type").HasValue("StudentDetails"))
20.   .Map<StudentLogin>(m                                      =>
    m.Requires("Type").HasValue("StudentLogin"));
```

Note

> By default, Entity Framework supports TPH inheritance, if you don't define any mapping details for your inheritance hierarchy.

2. Table-per-Type (TPT)

The TPT inheritance states that each entity in the hierarchy of entities is mapped to a separate table in storage schema. It means, there is separate table in database to maintain data for each Entity Type.



C# Implementation Code for TPT

```
1. public class Student
2. {
3.  public int StudentID { get; set; }
4.  public string Name { get; set; }
5. }
6.
7. public class StudentDetails : Student
8. {
9.  public string Address { get; set; }
10.  public string Mobile { get; set; }
11. }
12.
13. public class StudentLogin : Student
14. {
15.  public string UserName { get; set; }
16.  public string Password { get; set; }
17. }
```

Entity Framework Code First Mapping for TPT

```
18. modelBuilder.Entity<Student>().ToTable("Student");
19. modelBuilder.Entity<StudentLogin>().ToTable("StudentLogin");
20. modelBuilder.Entity<StudentDetails>().ToTable("StudentDetails");
```

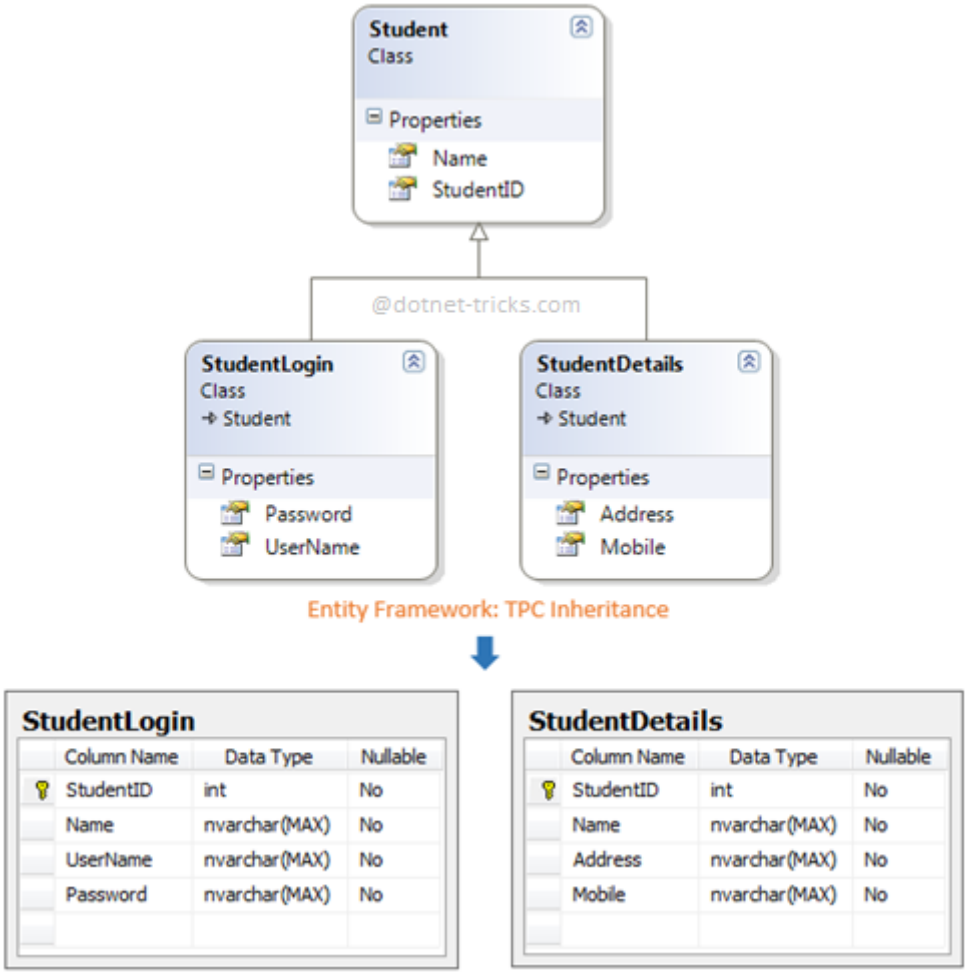
Note

> TPH inheritance patterns generally deliver better performance in the Entity Framework than TPT inheritance patterns, because TPT patterns can result in complex join queries.

3. Table-per-Concrete-Type (TPC)

The TPC inheritance states that each derived entity (not base entity) in the hierarchy of entities is mapped to a separate table in storage schema. It means, there is separate table in database to maintain data for each derived entity type.

This inheritance occurs when we have two tables with overlapping fields in the database like as a table and its history table that has all historical data which is transferred from the main table to it.



C# Implementation Code for TPC

```
1. public abstract class Student
2. {
3.  public int StudentID { get; set; }
4.  public string Name { get; set; }
5. }
6.
7. public class StudentDetails : Student
8. {
9.  public string Address { get; set; }
10.  public string Mobile { get; set; }
11. }
12.
13. public class StudentLogin : Student
14. {
15.  public string UserName { get; set; }
```

```
16.  public string Password { get; set; }
17. }
```

Entity Framework Code First Mapping for TPC

```
18. modelBuilder.Entity<StudentDetails>().Map(m =>
19. {
20.  m.MapInheritedProperties();
21.  m.ToTable("StudentDetails");
22. });
23.
24. modelBuilder.Entity<StudentLogin>().Map(m =>
25. {
26.  m.MapInheritedProperties();
27.  m.ToTable("StudentLogin");
28. });
```

Note

> The TPC inheritance is very rare but you should be aware of it and when it is needed.

References:

http://msdn.microsoft.com/en-us/data/ee712907

http://www.entityframeworktutorial.net

http://www.dotnet-tricks.com