Entity Framework

EF 6 Code First

Lesson Objectives

- EF Code First Overview
- Code-First Conventions
- Database Initialization
- Database Initialization Strategies
- Seed data

Section 1

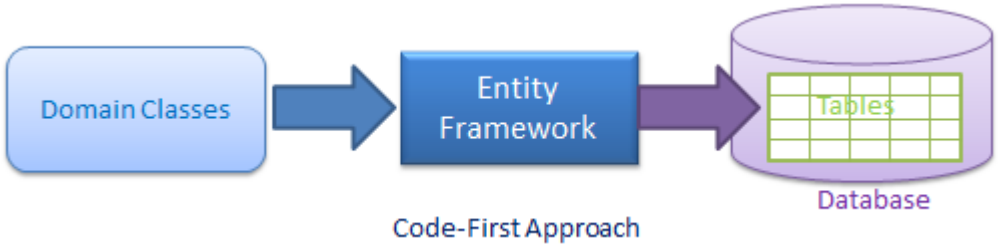
EF CODE FIRST OVERVIEW

EF Code First Overview

- Do not have an existing database for your application?
- Do not have much experience in SQL?
- Do not want to working with SQL directly?
- Prefer to writing entities and context class first?

EF Code First Overview

- Code-First is mainly useful in Domain Driven Design.
- In the Code-First approach,
  - ✓ developer focus on the domain of application
  - ✓ create classes for domain entity
  - ✓ EF generate database from code



Code-First Workflow

- Step 1: Create or modify domain classes
- Step 2: Configure these domain classes using Fluent-API or data annotation attributes
- Step 3: Create database using automated migration

Code-First Workflow

- Loop:
  - ✓ Step 2.x: Add or update the domain classes
  - ✓ Step 3.x: Update database schema using code first migration
- End loop

Section 2

CODE-FIRST CONVENTIONS

EF 6 Code-First Conventions

- Conventions are sets of default rules which automatically configure a conceptual model based on domain classes when working with the Code-First approach.
- Conventions are defined in the System.Data.Entity.ModelConfiguration.Conventions namespace.

EF 6 Code-First Conventions

- **Rules**:
  - ✓If no specification, use default convention
  - ✓If define specification, use defined

Default code first conventions

- **Schema**: EF creates all the DB objects into the **dbo** schema
  - ✓ Example: entity Student => **dbo**.Students
- **Table Name**: plural form of the entity class name
  - ✓ Example: entity Student => table Students
  - ✓ Example: entity Category => table Categories

Default code first conventions

- **Primary key Name**: EF will create a primary key column for the property named Id or <Entity Class Name> + "Id" (case insensitive)
  - ✓ Id
  - ✓ <Entity Class Name> + "Id“
- If data type of the field (to be come primary key) is **int**, EF set Identity for the primary key, with seed is 1, increment step is 1

Default code first conventions

- What happen if the entity class contains both?

Default code first conventions

- **Foreign key property Name**: EF will look for the foreign key property with the same name as the principal entity primary key name.
  - ✓ Use pair: public int CategoryID { get; set; }, public Category Category { get; set; }
  - ✓ SQL: CategoryID (FK, int, null)

Default code first conventions

- If the foreign key property does not exist, then EF will create an FK column in the Db table with <Dependent Navigation Property Name> + "_" + <Principal Entity Primary Key Property Name>
  - ✓ Use only: public Category Category { get; set; }
  - ✓ SQL: Category_ID(FK, int, null)

Default code first conventions

- Null column: EF creates a null column for all reference type properties and nullable primitive properties
  - ✓ Example: string, class type property
- Not Null Column: EF creates NotNull columns for Primary Key properties and non-nullable value type properties
  - ✓ Example: int, float, decimal, datetime
- To create null column for int, float, decimal, … use Nullable or ?
  - ✓ Example: Nullable<int> or int?

Default code first conventions

- DB Columns order:
  - ✓ EF will create DB columns in the same order like the properties in an entity class.
  - ✓ However, **primary key** columns would be moved first.
- Cascade delete
  - ✓ Enabled by default for all types of relationships.

Default code first conventions

- Properties mapping to DB:
  - ✓ All public properties (with **get** and **set**) will map to the database.
  - ✓ Otherwise, the property will not mapped to the database

Default code first conventions

- Examples:
  - ✓ public string Description { get; set; } => mapped
  - ✓ public string Description { get { return Name; } } => **not** mapped
  - ✓ public string Description { set { value = Name; } } => **not** mapped
  - ✓ public string Description; => **not** mapped
  - ✓ private string Description { get; set; } => **not** mapped
  - ✓ protected string Description { get; set; } => **not** mapped

Data type mapped

C# Data Type	Mapping to SQL Server Data Type	C# Data Type	Mapping to SQL Server Data Type
int	int	byte	tinyint
string	nvarchar(Max)	short	smallint
decimal	decimal(18,2)	long	bigint
float	real	double	float
byte[]	varbinary(Max)	char	*No mapping*
datetime	datetime	sbyte	*No mapping (throws exception)*
bool	bit	object	*No mapping*

Relationship Convention

- EF 6 infers the One-to-Many relationship using the navigation property by default convention.
- EF 6 does not include default conventions for One-to-One and Many-to-Many relationships.
- You need to configure them either using Fluent API or DataAnnotation.

Conventions for One-to-Many Relationships

- Convention 1:
  - ✓ includes a reference navigation property of parent in the child entity class
- Convention 2:
  - ✓ includes a collection navigation property of children in the parent entity class
- Convention 3:
  - ✓ includes navigation properties at both ends will also result in a one-to-many relationship
- Convention 4:
  - ✓ a fully defined relationship at both ends will create a one-to-many relationship

Conventions for One-to-Many Relationships

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

    public int GradeId { get; set; }    [4]
    public Grade Grade { get; set; }    [1, 3, 4]
}

public class Grade
{

    public int GradeId { get; set; }
    public string GradeName { get; set; }

    public ICollection<Student> Student { get; set; } [2, 3, 4]
}
```

Section 3

DATABASE INITIALIZATION

Database Initialization

- Based on the parameter passed in the base constructor of the context class, EF decides the database name and server while initializing a database
- The DbContext constructor overloading
  - ✓ No Parameter
  - ✓ string: nameOrConnectionString

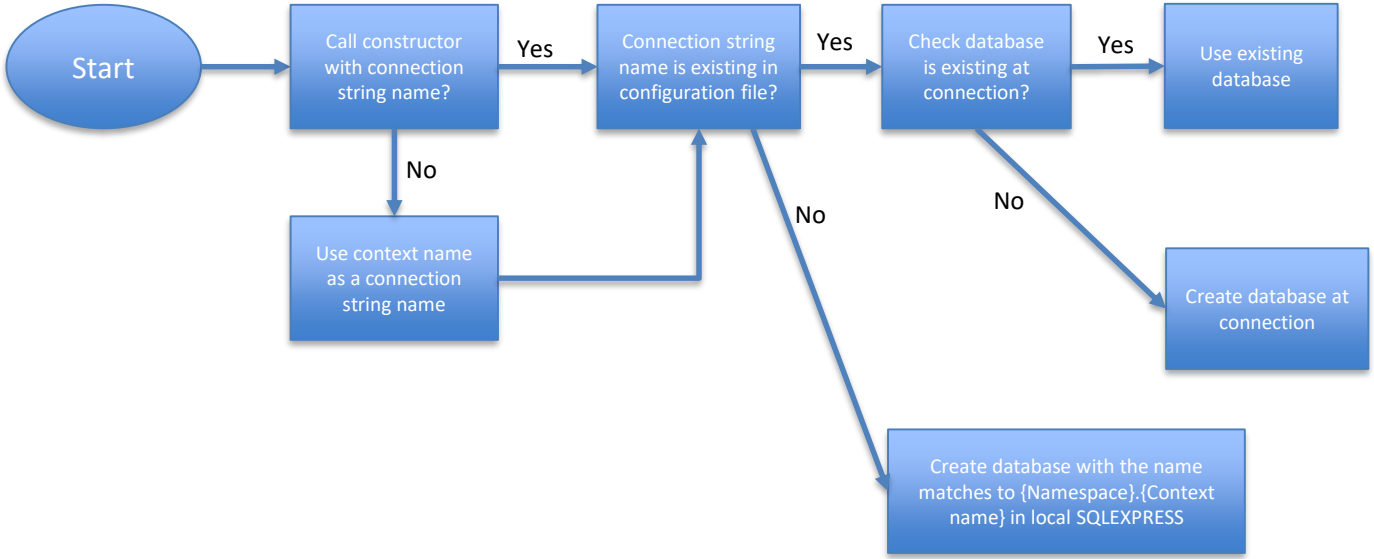
No Parameter

- If we do not specify the parameter in the base constructor of the context class, EF use DbContext class name as a connection string name
  - ❖ Create: public DbShopContext() : base()
  - ❖ Equals with: public DbShopContext() : base("DbShopContext")

Connection String Name

- Define the connection string in app.config or web.config
- Pass connection string name as a parameter of base constructor
  - ✓ DbContext class:
    - public DbShopContext() : base("DbShopContextConnectionString")
  - ✓ Web.config/App.config:
    - <add name="DbShopContextConnectionString" connectionString="....." providerName="System.Data.SqlClient"/>

Database Initialization



Section 4

DATABASE INITIALIZATION STRATEGIES

Database Initialization Strategies

- CreateDatabaseIfNotExists
- DropCreateDatabaseIfModelChanges
- DropCreateDatabaseAlways
- Custom DB Initializer
- Disabled

CreateDatabaseIfNotExists

- This is the **default** initializer.
- It will create the database if none exists as per the configuration.
- If the model class is changed then run the application with this initializer, then it will throw an exception.

DropCreateDatabaseIfModelChanges

- If model classes (entity classes) have been changed
  - ✓ Drops an existing database
  - ✓ Creates a new database
- We don't have to worry about maintaining database schema, when model classes change.
- We may lost important data
- Only use this strategy in development environment

DropCreateDatabaseAlways

- Drops an existing database every time you run the application, irrespective of whether your model classes have changed or not.
- This will be useful when we want a fresh database every time we run the application.
- Only use this strategy in development environment

Custom DB Initializer

- Create custom DB Initializer by inherit one of three strategy above
- Usually used to apply configuration or create Seed data

```
public class ShopContextInitializer<T> : CreateDatabaseIfNotExists<DbShopContext>
{

    protected override void Seed(DbShopContext context)
    {
        base.Seed(context);
        //// Create seed data
    }
}
```

Turn off the DB Initializer

- When you are sure, the database is existing already.
- When you don't want to lose existing data in the production environment
- Working process:
  1. Deploy stage: Initialize database, add Seed data
  2. Test stage: Test application, update database (if need)
  3. Product stage: Turn off the DB Initializer

Set an initializer in configuration

- To change initializer without update code/re-deploy
- Same code in difference environments: coding, testing, UAT, PROD, maintain
- To detect/investigate problems

Set an initializer in configuration

- Syntax:

```
<appSettings>
    <add key=”DatabaseInitializerForType [MyAssemblyQualifiedContextType]”
    value=”[MyAssemblyQualifedInitializerType]” />
</appSettings>
```

- Example:

```
<appSettings>
    <add key=”DatabaseInitializerForType
            LazyUnicornTests.Model.BlogContext, LazyUnicornTests”
    value=”LazyUnicornTests.Model.BlogsContextInitializer, LazyUnicornTests” />
</appSettings>
```

Set an initializer in configuration

- The entry goes in the appSettings section of configuration file
- The key must always start with the string “DatabaseInitializerForType” followed by whitespace
- …

Set an initializer in configuration

- The second part of the key is the assembly-qualified name of the context type for which the initializer should be set
- The value is the assembly-qualified name of the initializer to set, which must expose a public, parameterless constructor so that EF can create an instance of it

Turn off the DB Initializer

```
<appSettings>
    <add key="DatabaseInitializerForType
    SchoolDataLayer.SchoolDBContext, SchoolDataLayer"
        value="Disabled" />
 </appSettings>
```

- Value can be: **Disabled** or **empty string**

Section 5

SEED DATA

Seed Data

- To insert data into database tables during the database initialization process.
- Use to provide some test data for the application
- Use to create some default master data for the application

Seed Data steps

- Step 1: create a custom DB initializer
  - ✓ public class ShopContextInitializer<T> : CreateDatabaseIfNotExists<DbShopContext>
- Step 2: override the Seed method
  - ✓ protected override void Seed(DbShopContext context)
- Step 3: don’t forget call the base Seed
  - ✓ base.Seed(context);
- Step 4: add code to add data

Summary

- Use EF Code First approach: create entities class and DbContext first, then generate database
- Code-First Conventions: EF uses conventions to create database, tables, columns
- There are strategies to create database
- Always consider to use appropriate strategy

Thank you