ENTITY FRAMEWORK

Seminar 4

Software Engineering
ETS Software Engineering
DSIC – UPV

Goals

- To know the persistence model of Entity Framework
- To learn the application of the code-first approach
- To develop an example App based on Entity Framework code-first approach

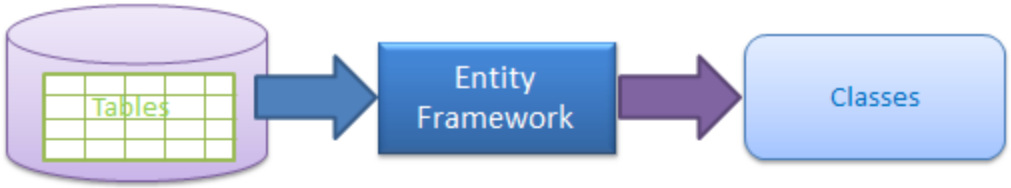
Contents

Introduction

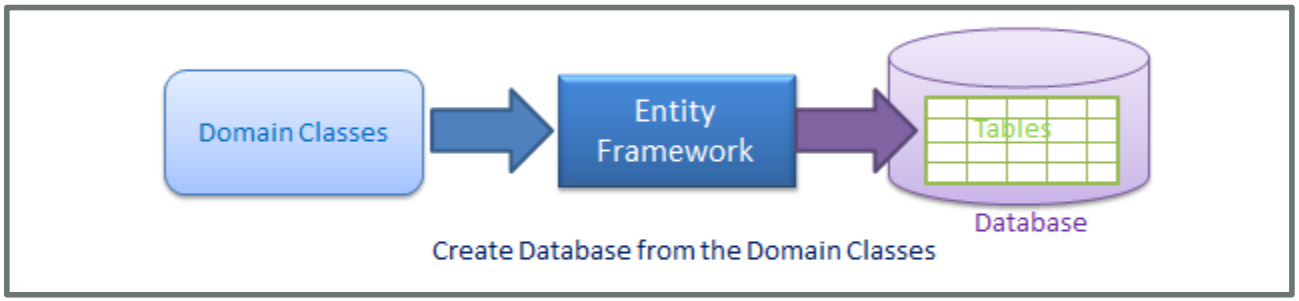
- EF is an **Object/Relational Mapping** (O/RM) framework
  - Keep our database design separate from our domain class design.
  - Automate standard CRUD operations (Create, Read, Update & Delete) so that the developer doesn't need to write them manually.

- EF supports three development approaches:
  - **Database-first**: you already have existing database or you want to design your database ahead of other parts of the application
  - **Code-first**: you want to focus on your domain classes and then create the database from your domain classes
  - **Model-first**: you want to design your database schema on the visual designer and then create the database and classes

Introduction

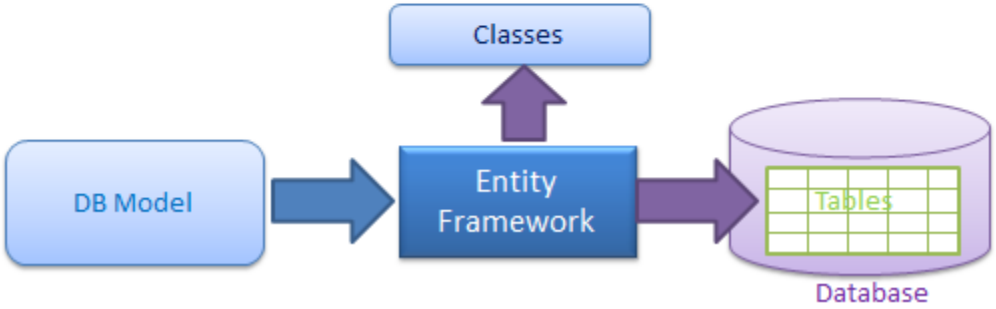


Generate Data Access Classes for Existing Database



Create Database from the Domain Classes

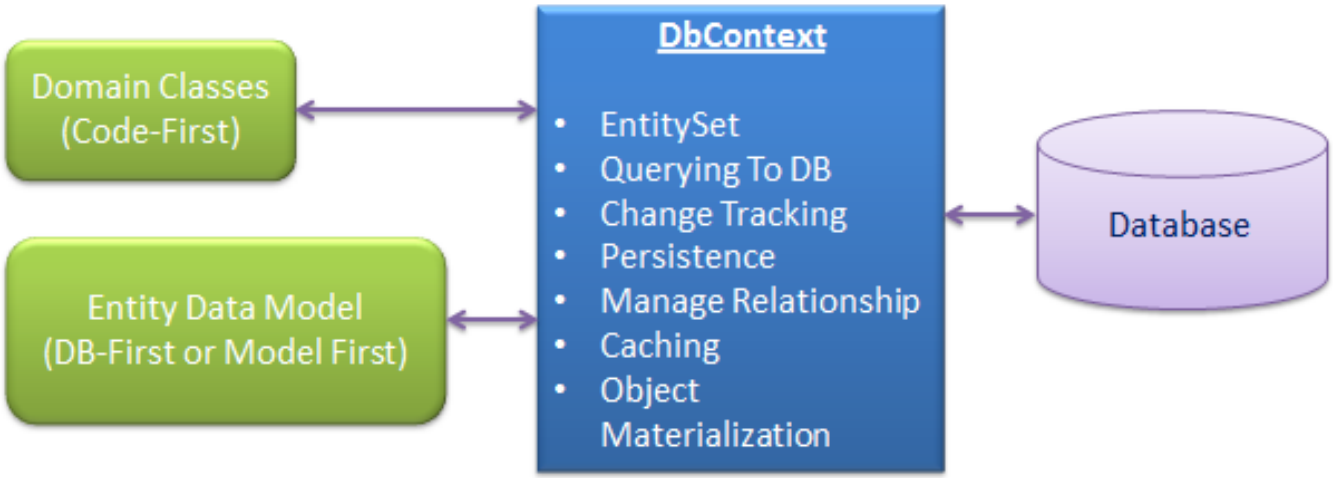
Code-first



Create Database and Classes from the DB Model design

The Context: DbContext Class

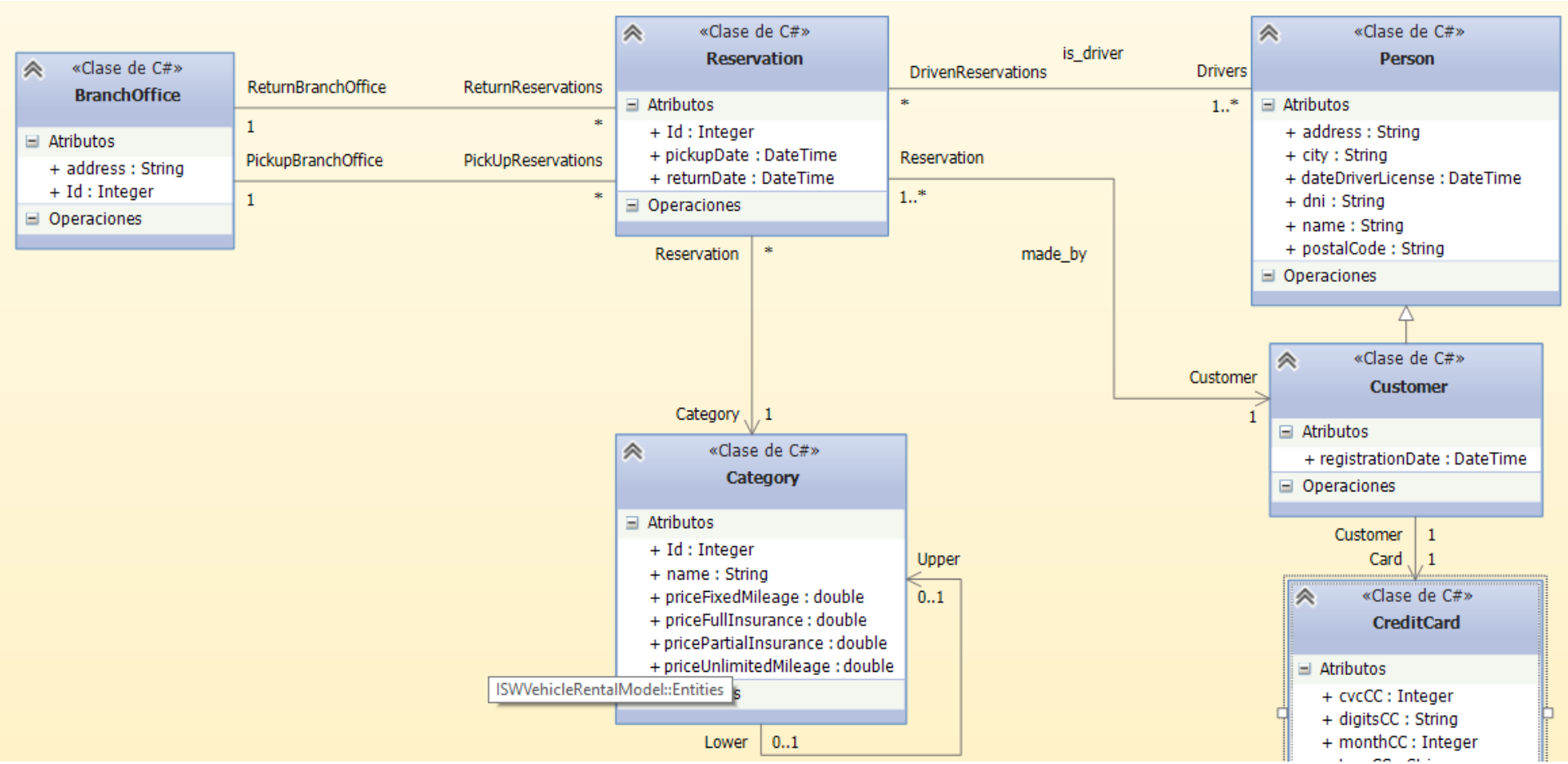
- DbContext is an important part of Entity Framework. It is a bridge between your domain or entity classes and the database.



EF DbContext Functionality

- **EntitySet**: DbContext contains entity sets for all the entities which are mapped to DB tables (DbSet<TEntity>)
- **Querying**: DbContext converts LINQ-to-Entities queries to SQL queries and send them to the database.
- **Change Tracking**: It keeps track of changes that occurred in the entities after they have been retrieved from the database.
- **Persisting Data**: It also performs the Insert, Update and Delete operations to the database, based on what the entity states.
- **Caching**: It does first level caching by default. It stores the entities which have been retrieved during its life time.
- **Managing Relationships**: DbContext also manages relationships using fluent API in Code-First approach.
- **Object Materialization**: DbContext converts raw table data into entity objects.

Example Domain Model



Example DbContext

```
internal class VehicleRentalDbContext : DbContext
    {
        public VehicleRentalDbContext() :
                base("Name=VehicleRentalDbConnection")
        {
        }

        public IDbSet<BranchOffice> BranchOffices { get; set; }
        public IDbSet<Reservation> Reservations { get; set; }
        public IDbSet<Category> Categories { get; set; }
        public IDbSet<Person> Persons { get; set; }
        public IDbSet<Customer> Customers { get; set; }
        public IDbSet<CreditCard> CreditCards { get; set; }
    }
```

Entity Sets

IDbSet & DbSet

- IDbSet<TEntity> represents the collection of all the entities of a given type that may be managed in persistent repository (DB).
- `DbSet<Tentity>` is a concrete implementation of IDbSet.
- `DbSet` objects are created from a `DbContext` object using the method `DbContext.Set<TEntity>()`

Code-First Conventions

- Code First APIs create the database and map domain classes with the database using default code-first conventions
  - **Type Discovery** Convention
  - **Primary Key** Convention
  - **Relationship** Convention
  - **Foreign key** Convention
  - **Inheritance** Convention

- A convention is a set of default rules to automatically configure a conceptual model based on domain class definitions

Type-Discovery Convention

- Code-First will create tables for classes included as DbSet properties

- Code-First also includes any referenced types included in these classes

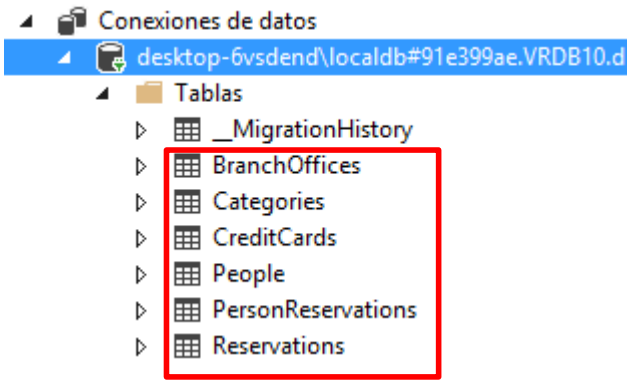
Example Type-Discovery Convention

- EF automatically generates a table for each DbSet Entity

```
public IDbSet<BranchOffice> BranchOffices { get; set; }
public IDbSet<Reservation> Reservations { get; set; }
public IDbSet<Category> Categories { get; set; }
public IDbSet<Person> Persons { get; set; }
public IDbSet<Customer> Customers { get; set; }
public IDbSet<CreditCard> CreditCards { get; set; }
```

Names of tables in Plural. E.g. People instead of Person

Additional tables for many-to-many relationships. E.g. PersonReservations

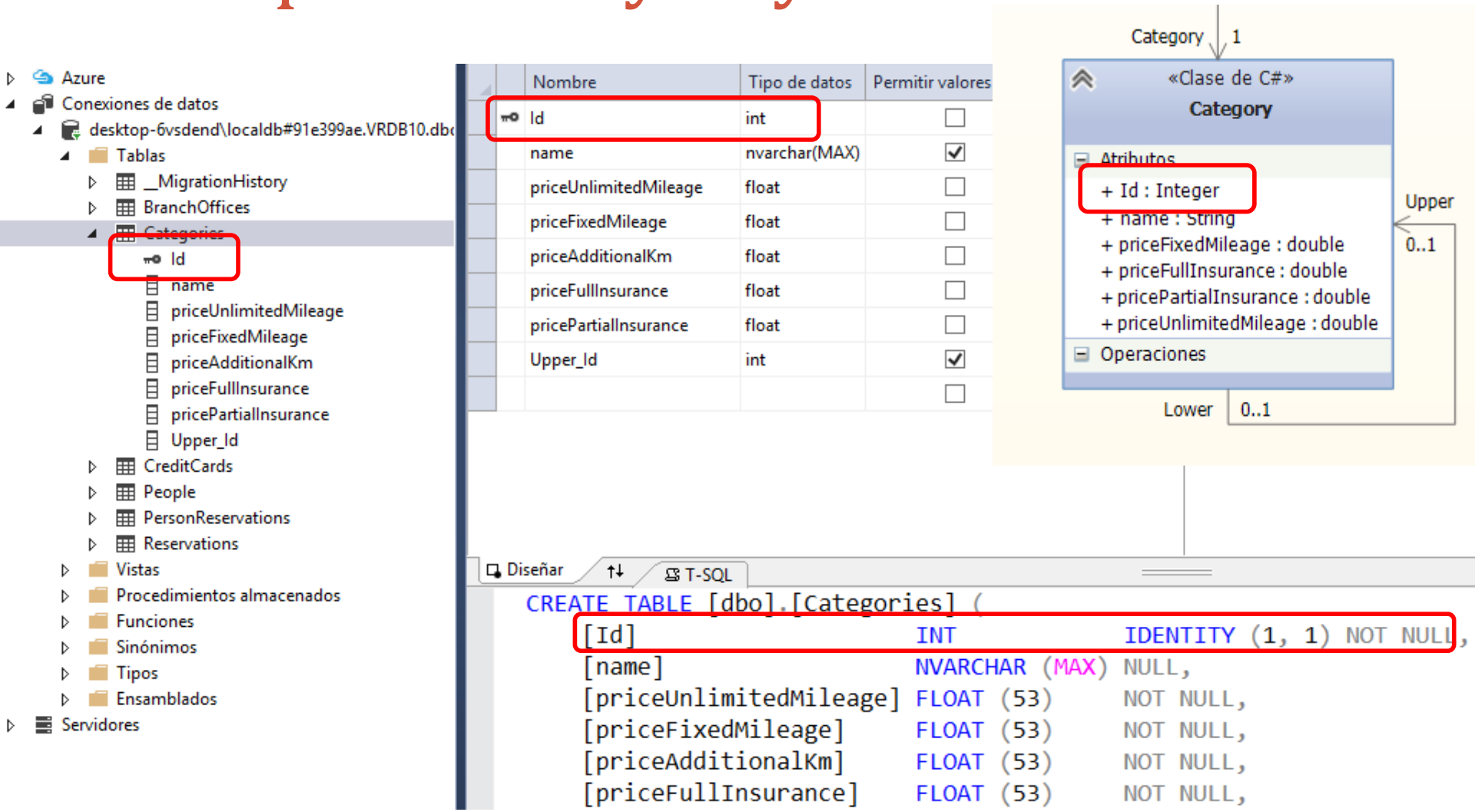


Primary-Key Convention

- Code-First will create a primary key for a property if the property name is Id or <class name>Id

- The type of a primary key property can be anything, but if the type of the primary key property is numeric or GUID, it will be configured as an identity column

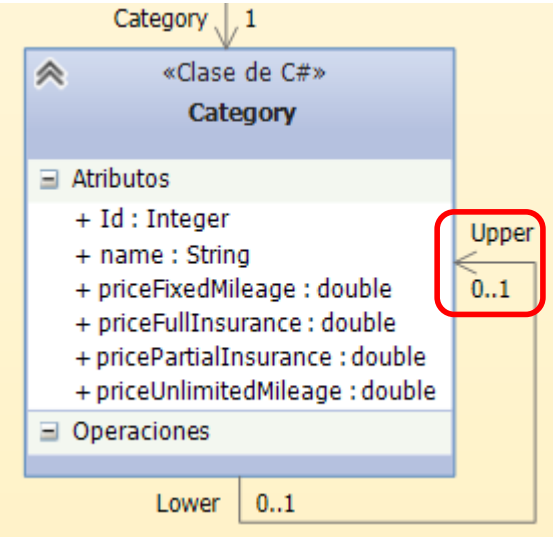
Example Primary-Key Convention



Relationship Conventions

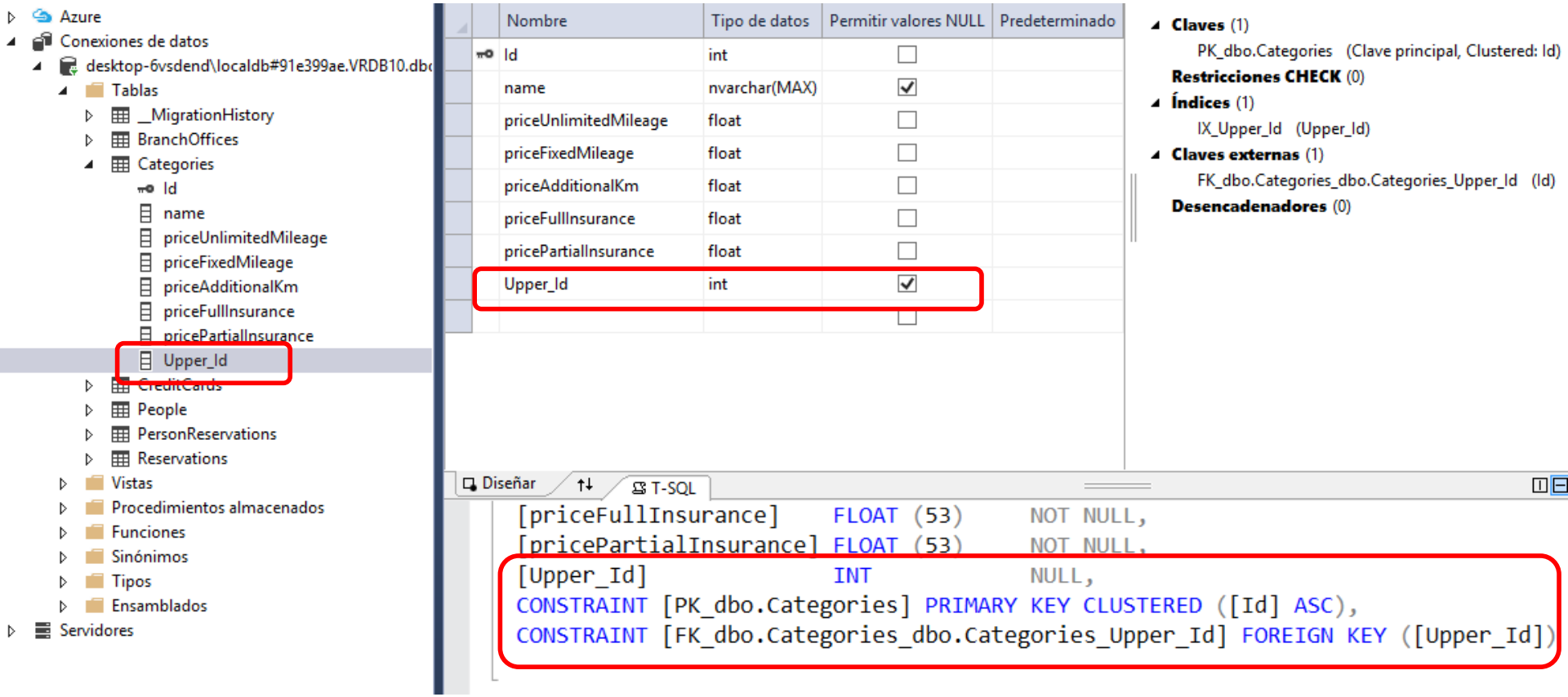
- If your classes include two reference properties, Code First will assume a **one-to-one relationship**
- If your classes contain a reference and a collection navigation property, Code First assumes a **one-to-many relationship**.
- If your classes include two collection properties, Code First will use a **many-to-many relationship**.

One-to-One Relationship Example

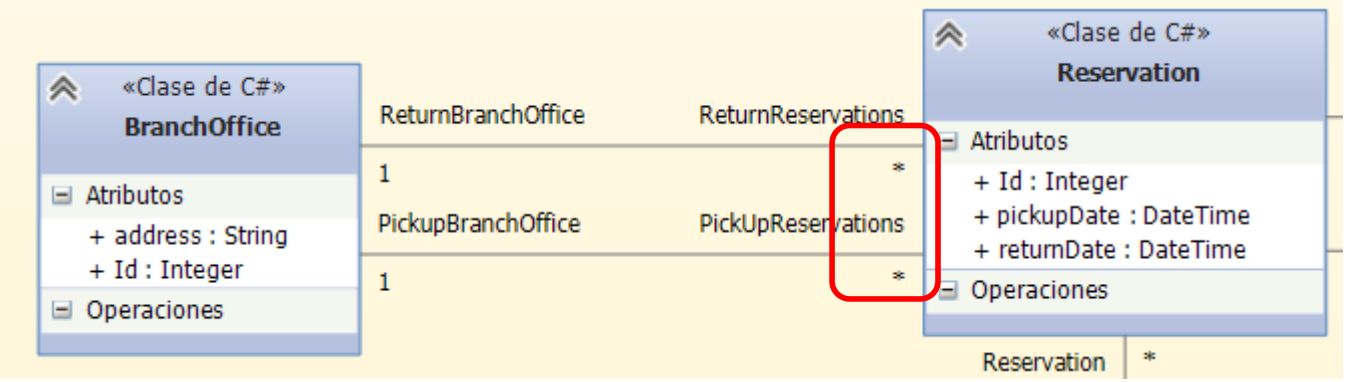



```
public partial class Category
{
    public double priceUnlimitedMileage...
    public double priceFixedMileage...
    public string name...
    public double priceFullInsurance...
    public double pricePartialInsurance...
    public int Id
    {
        get;
        set;
    }

    public virtual Category Upper
    {
        get;
        set;
    }
}
}
```

One-to-One Relationship Example



One-to-Many Relationship Example

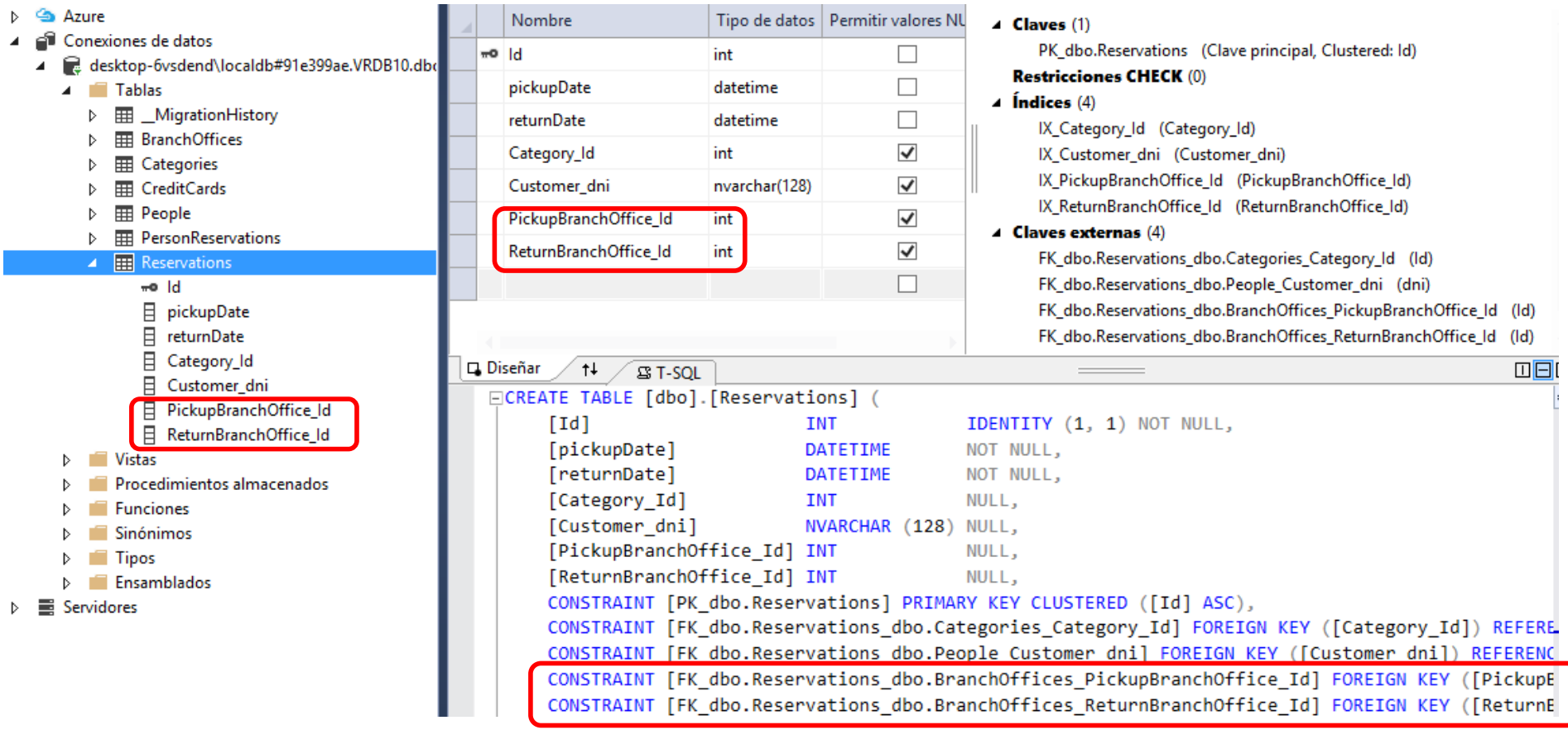



```
public partial class BranchOffice
{

    public string address...
    public int Id...
    public virtual ICollection<Reservation> PickUpReservations
    {
        get;
        set;
    }

    public virtual ICollection<Reservation> ReturnReservations
    {
        get;
        set;
    }

}
}
```

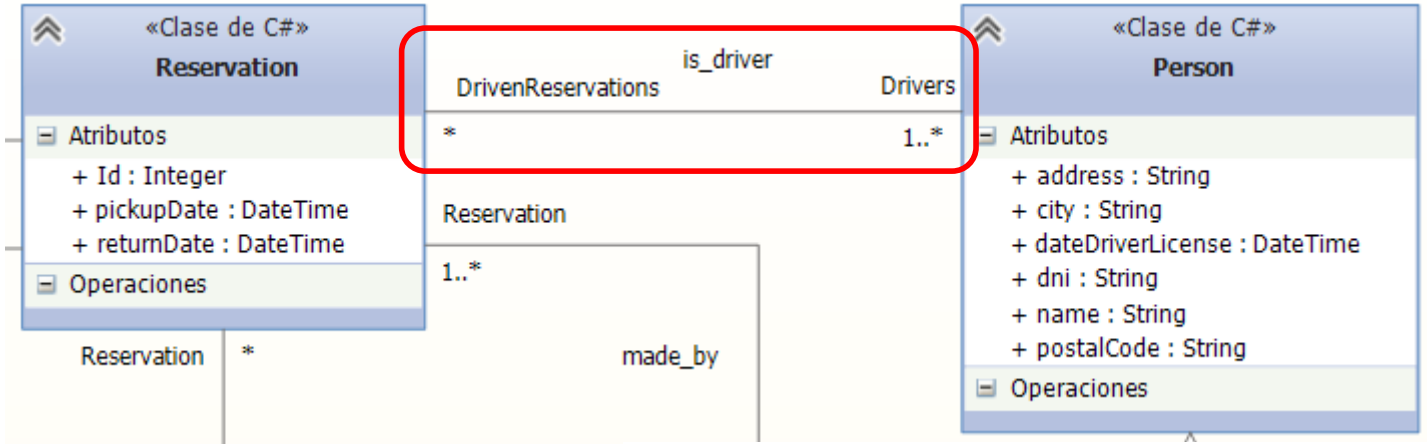
One-to-Many Relationship Example


```
public partial class Reservation
{

    public int Id...
    public DateTime pickupDate...
    public DateTime returnDate...
    public virtual Category Category...
    [InverseProperty("PickUpReservations")]
    public virtual BranchOffice PickupBranchOffice
    {
        get;
        set;
    }
    [InverseProperty("ReturnReservations")]
    public virtual BranchOffice ReturnBranchOffice
    {
        get;
        set;
    }
```

One-to-Many Relationship Example



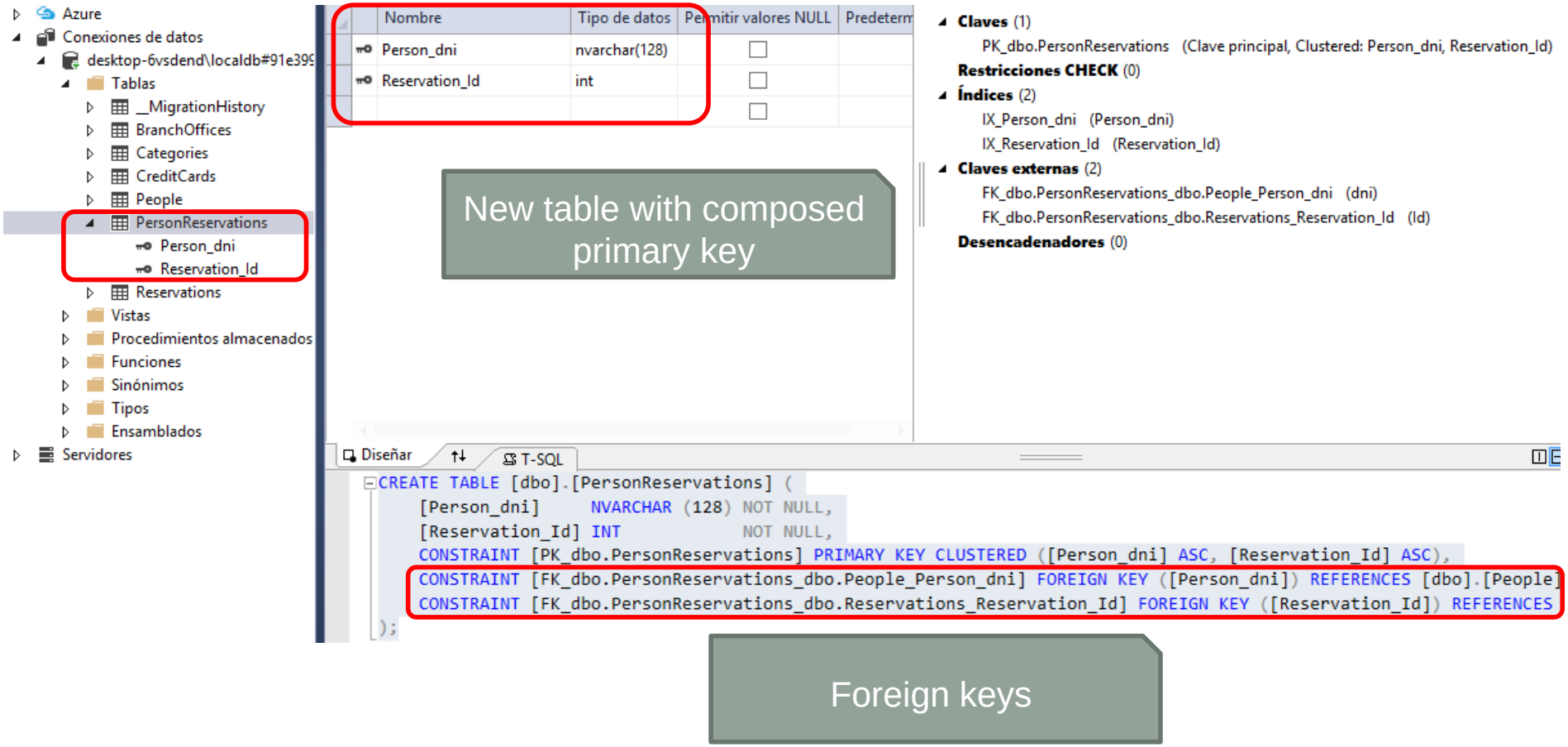
Many-to-Many Relationships Example




```
public partial class Reservation
{

    public int Id...
    public DateTime pickupDate...
    public DateTime returnDate...
    public virtual Category Category...
    [InverseProperty("PickUpReservations")]
    public virtual BranchOffice PickupBranchOffice...
    [InverseProperty("ReturnReservations")]
    public virtual BranchOffice ReturnBranchOffice...
    public virtual ICollection<Person> Drivers
    {
        get;
        set;
    }
```

```
public partial class Person
{
    [Key]
    public string dni...
    public string name...
    public string address...
    public string city...
    public string postalCode...
    public DateTime dateDriverLicense...
    public virtual ICollection<Reservation> DrivenReservations...
}
```

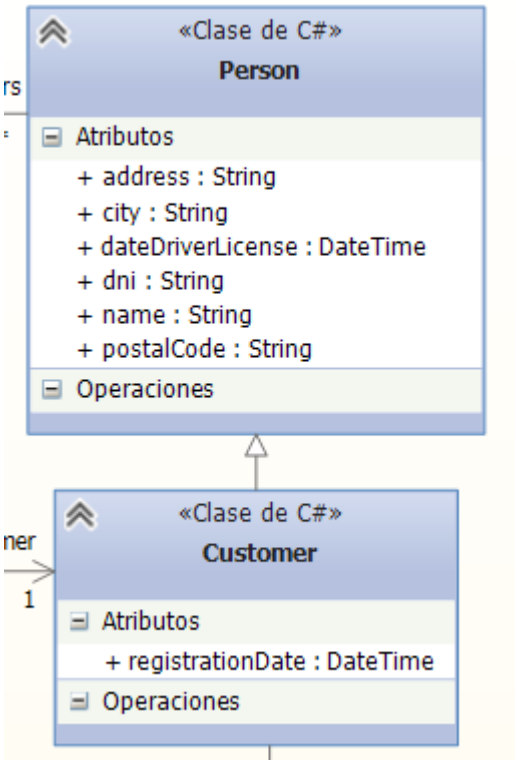
Many-to-Many Relationships Example

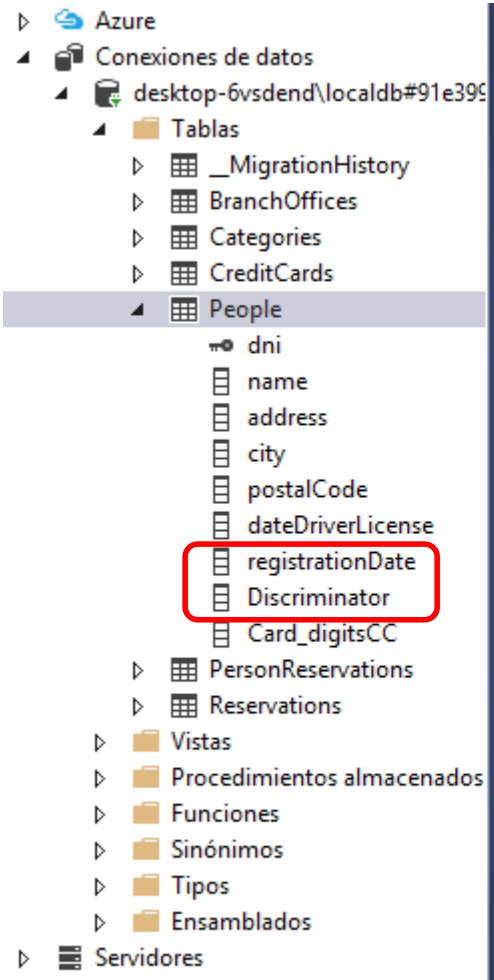


Inheritance Convention

- **Table per Hierarchy (TPH):** This approach suggests one table for the entire class inheritance hierarchy.
  - Table includes discriminator column which distinguishes between inheritance classes.
  - Default inheritance mapping strategy in Entity Framework.
- **Table per Type (TPT):** This approach suggests a separate table for each domain class.
- **Table per Concrete class (TPC):** This approach suggests one table for one concrete class, but not for the abstract class.
  - The properties of the abstract class will be part of each table of the concrete classes.

Table per Hierarchy Example





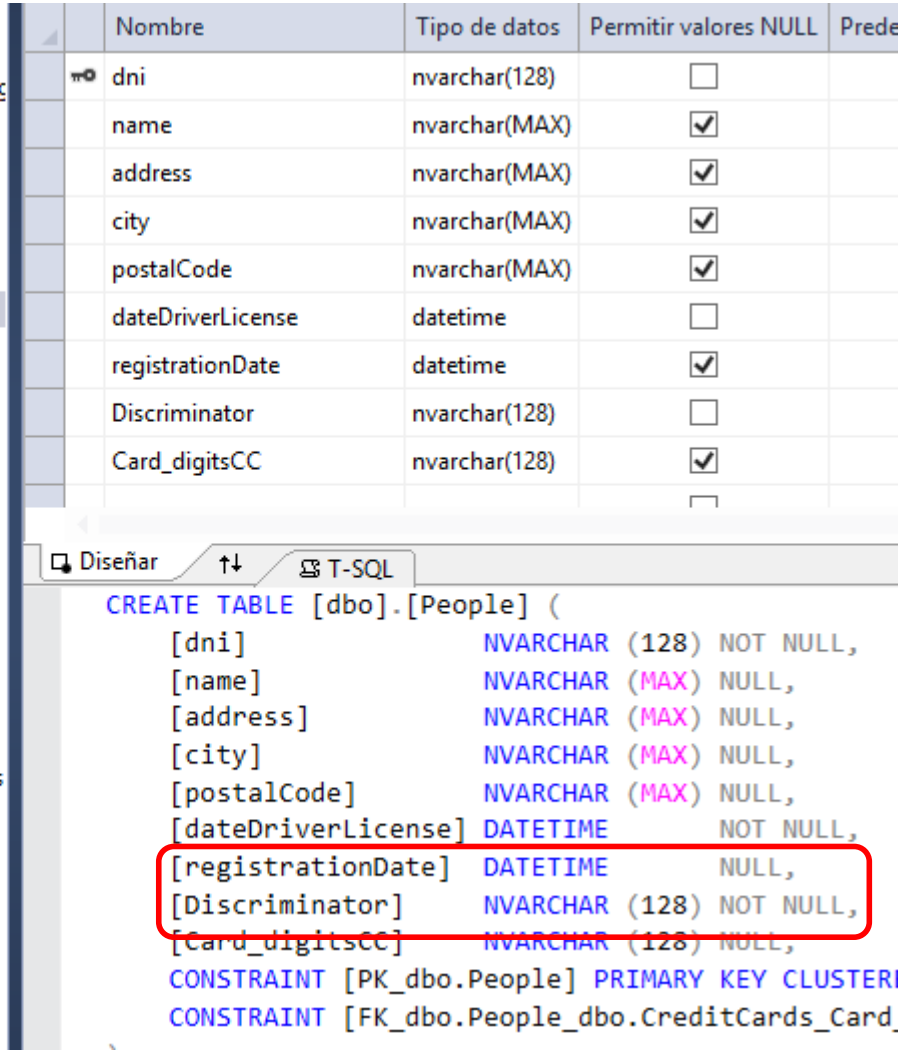
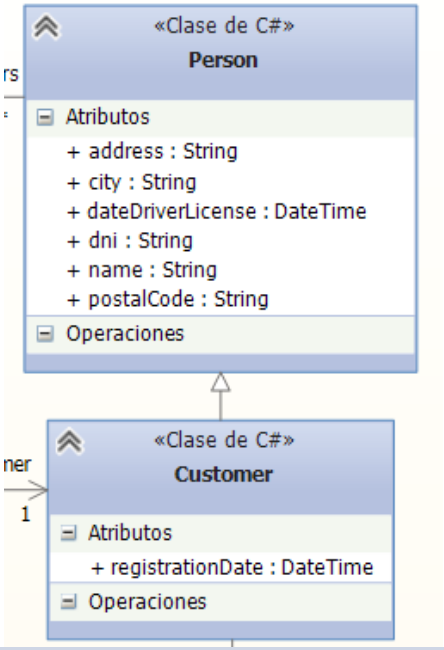


Table per Hierarchy Example



dni	name	address	city	postalCode	dateDriverLice...	registrationDate	Discriminator
1	asdf	sdf	asdf	dsf	16/03/2016 5:56..	16/03/2016 6:04...	Customer
11111111A	Javier Jaen	Camino de Vera...	Valencia	46960	23/05/2014 0:00..	11/12/2015 0:00...	Customer
2	asdf	sdf	asdf	dsf	16/03/2016 5:56..	*NULL*	Person
22222222B	Vicente Nacher	C/ Goya, 13	Valencia	46023	15/03/2016 17:2..	15/03/2016 17:2...	Customer
3	asdf	sdf	asdf	dsf	16/03/2016 5:56..	*NULL*	Person
33333333C	Patricia Pons	C/Goya 16	Valencia	46960	15/03/2016 17:2..	15/03/2016 17:2...	Customer
44444444D	asdf	asdf	asd	asdf	15/03/2016 17:3..	15/03/2016 17:3...	Customer
NULL	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*

Conventions Key Facts

Default Convention For	Description
Table Name	<Entity Class Name> + 's' EF will create DB table with entity class name suffixed by 's'
Primary key Name	1) Id 2) <Entity Class Name> + "Id" (case insensitive) EF will create primary key column for the property named Id or <Entity Class Name> + "Id" (case insensitive)
Foreign key property Name	By default EF will look for foreign key property with the same name as principal entity primary key name. If foreign key property does not exists then EF will create FK column in Db table with <Dependent Navigation Property Name> + "_" + <Principal Entity Primary Key Property Name> e.g. EF will create Standard_StandardId foreign key column into Students table if Student entity does not contain foreignkey property for Standard where Standard contains StandardId
Null column	EF creates null column for all reference type properties and nullable primitive properties.
Not Null Column	EF creates NotNull columns for PrimaryKey properties and non-nullable value type properties.
DB Columns order	EF will create DB columns same as order of properties in an entity class. However, primary key columns would be moved first.
Properties mapping to DB	By default all properties will map to database. Use [NotMapped] attribute to exclude property or class from DB mapping.
Cascade delete	Enabled By default for all types of relationships.

Domain Classes Configuration

- To **override** the previous conventions by configuring your domain classes to provide EF with the information it needs
- Two ways to configure your domain classes
  - **DataAnnotations**: Attribute based configuration, that may be applied to domain classes and their properties
  - **Fluent API:** An advanced way of specifying model configuration that covers everything that data annotations can do in addition to some more advanced configuration not possible with data annotations

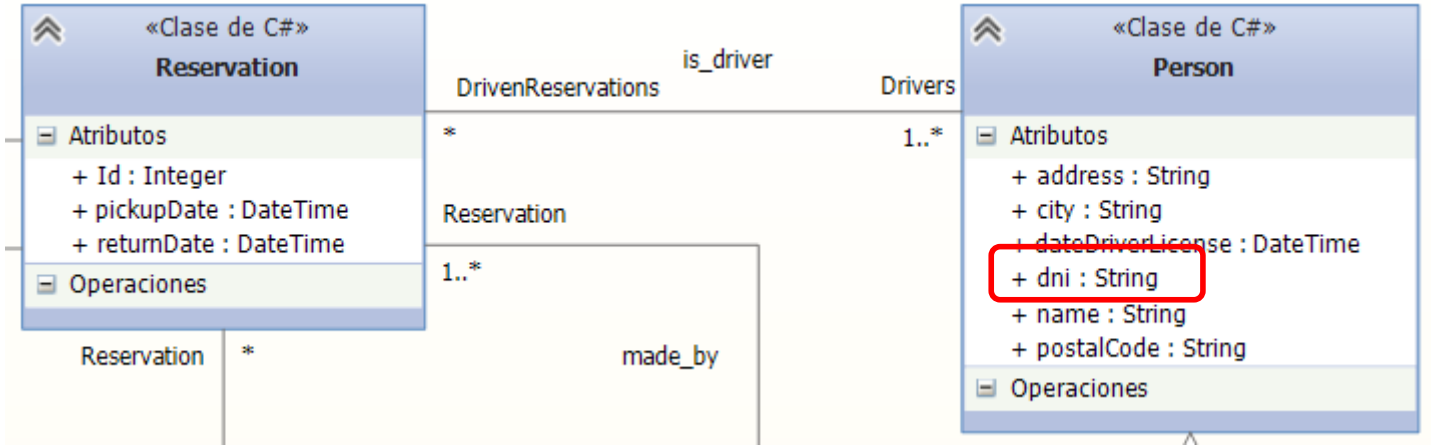
Data Annotations

- **System.ComponentModel.DataAnnotations** includes the following attributes that impacts the nullability or size of the column.
  - Key
  - Timestamp
  - ConcurrencyCheck
  - Required
  - MinLength
  - MaxLength
  - StringLength

Data Annotations

- **System.ComponentModel.DataAnnotations.Schema** namespace includes the following attributes that impacts the schema of the database.
  - Table
  - Column
  - Index
  - ForeignKey
  - NotMapped
  - InverseProperty

Data Annotations Example




```
public partial class Person
{
    [Key]
    public string dni...
    public string name...
    public string address...
    public string city...
    public string postalCode...
    public DateTime dateDriverLicense...
    public virtual ICollection<Reservation> DrivenReservations...

}
```

Data Annotations Example


```
public partial class Reservation
{

    public int Id...
    public DateTime pickupDate...
    public DateTime returnDate...
    public virtual Category Category...
    [InverseProperty("PickUpReservations")]
    public virtual BranchOffice PickupBranchOffice...
    [InverseProperty("ReturnReservations")]
    public virtual BranchOffice ReturnBranchOffice...
    public virtual ICollection<Person> Drivers
    {
        get;
        set;
    }
```

Tasks

- Understand the meaning of the different data annotations:

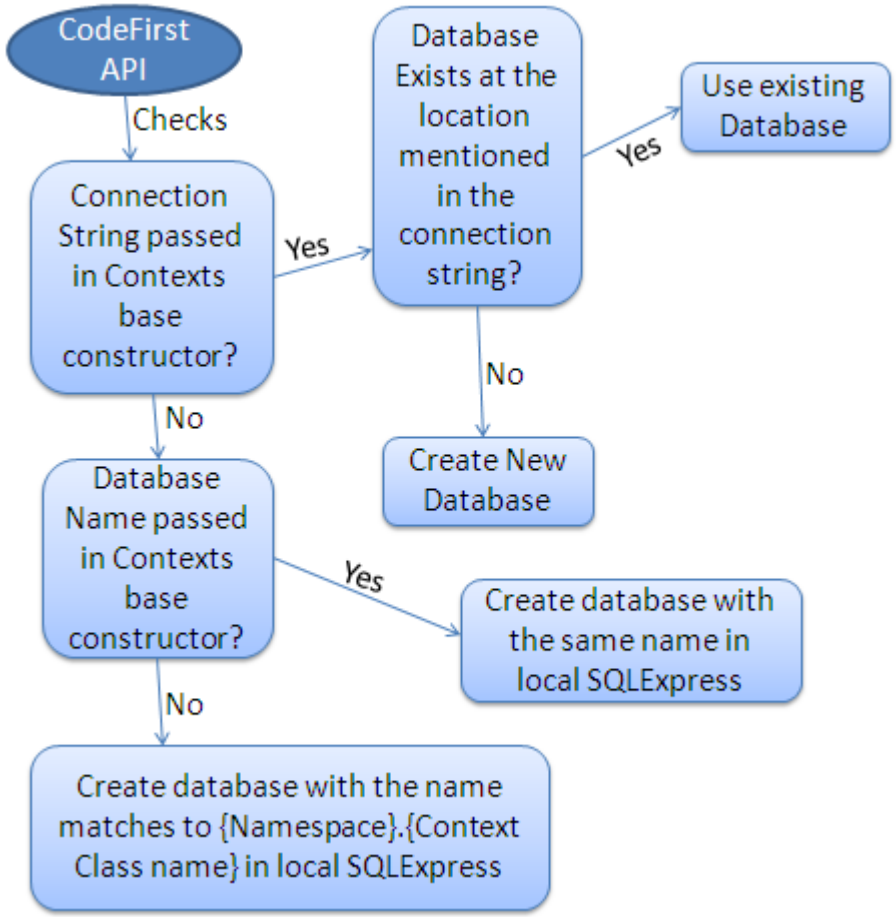
  http://www.tutorialspoint.com/entity_framework/entity_framework_data_annotations.htm

- Advanced Task. Understand how Fluent API Works

  http://www.tutorialspoint.com/entity_framework/entity_framework_fluent_api.htm

Database Initialization

- Code First creates a database automatically according to the following workflow



Database Initialization

- No Parameter in base constructor of DbContext class
  - A database in local SQLEXPRESS server with a name that matches {Namespace}.{Context class name}
- Database name as a parameter in a base constructor of DbContext class
  - A database with the name you specified in the base constructor in the local SQLEXPRESS database server

Database Initialization

- Define connection string in App.config or web.config and specify connection string name starting with "name=" in the base constructor of the context class

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
    </startup>
  <connectionStrings>
    <add name="DBConnectionString" connectionString="Data Source=(LocalDB)\MSSQLLocalDB;
         Initial Catalog=VRDB10;Integrated Security=True;Connect Timeout=15"
         providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

Database Operations: Using DbContext

- IDbSet offers methods to add, remove and search objects
- Enables to express and execute queries
- Takes query results from the database and transforms them into instances of our model classes
- 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

Example of operations in EF

```
VehicleRentalDbContext context = new
VehicleRentalDbContext();

context.Categories.Add(new Category("luxury",
23, 12, 2, 32, 14));
context.People.Add(new Person("222222222A", …);
Person p = context.People.Find("12345678A");
context.People.Remove(p);
context.SaveChanges();

context.People.Where(person => person.Id ==
“123456789A”)
```

Task

- Explore VehicleRentalApp and identify the different EF elements
- What data Access pattern has been implemented and where is it implemented in the project?
- What are the benefits of the proposed architecture and of the selected EF technology?
- What are the benefits of having the IDAL interface

Conclusions

- EF is an **Object/Relational Mapping** (O/RM) framework
- Automate standard CRUD operations (Create, Read, Update & Delete) so that the **developer doesn't need to write them manually**
- **Code-first**: you want to focus on your domain classes and then create the database from your domain classes
- Code First APIs create the database and map domain classes with the database using default code-first conventions
- DBContext enables to express and execute queries, keeps changes tracking and materializes objects

References

- Hirani, Z., et al. Entity Framework 6 Recipes 2013.
- Entity Framework Documentation (MSDN)
  - Entity Framework Code First Conventions
  - Entity Framework Code First Data Annotations
  - Entity Framework Fluent API – Relationships
  - Entity Framework Fluent API - Configuring and Mapping Properties and Types
  - Entity Framework Loading Related Entities
- Tutoriales Online
  - http://www.entityframeworktutorial.net
  - http://www.tutorialspoint.com/entity_framework/

LOADING STRATEGIES

Entities Loading Strategies

- **Eager loading**: a query for one type of entity also loads related entities as part of the query.
  - Achieved by the use of the **Include()** method.

- **Lazy loading:** An entity or collection of entities are automatically loaded from the database the first time that a property referring to the entity/entities is accessed.
  - Delaying the loading of related data, until requested.
  - Achieved by creating instances of derived proxy types and then overriding virtual properties to add the loading hook.
  - Default loading mechanism

- **Explicit loading**: if disabled the lazy loading, it is still possible to lazily load related entities with an explicit call.
  - No ambiguity or possibility of confusion regarding when a query is run.
  - Use the **Load()** method on the related entity's entry.

Eager Loading Example

```
class Program {

   static void Main(string[] args) {

      using (var context = new UniContextEntities()) {
         // Load all students and related enrollments
         var students = context.Students
            .Include(s => s.Enrollments).ToList();

         foreach (var student in students) {
            string name = student.FirstMidName + " " + student.LastName;
            Console.WriteLine("ID: {0}, Name: {1}", student.ID, name);

            foreach (var enrollment in student.Enrollments) {
               Console.WriteLine("Enrollment ID: {0}, Course ID: {1}",
                  enrollment.EnrollmentID, enrollment.CourseID);
            }
         }

         Console.ReadKey();
      }
   }
}
```

Explicit Loading Example

```
class Program {

   static void Main(string[] args) {

      using (var context = new UniContextEntities()) {

         context.Configuration.LazyLoadingEnabled = false;

         var student = (from s in context.Students where s.FirstMidName ==
            "Ali" select s).FirstOrDefault<Student>();

         string name = student.FirstMidName + " " + student.LastName;
         Console.WriteLine("ID: {0}, Name: {1}", student.ID, name);

         foreach (var enrollment in student.Enrollments) {
            Console.WriteLine("Enrollment ID: {0}, Course ID: {1}",
               enrollment.EnrollmentID, enrollment.CourseID);
         }

         Console.WriteLine();
         Console.WriteLine("Explicitly loaded Enrollments");
         Console.WriteLine();

         context.Entry(student).Collection(s => s.Enrollments).Load();
         Console.WriteLine("ID: {0}, Name: {1}", student.ID, name);

         foreach (var enrollment in student.Enrollments) {
            Console.WriteLine("Enrollment ID: {0}, Course ID: {1}",
               enrollment.EnrollmentID, enrollment.CourseID);
         }

         Console.ReadKey();
      }
   }
}
```