ADO.NET
Entity Framework

CSIS 3540

Client Server Systems

Class 10

Topics

- Entity Framework Overview
- DBContext, DBSet
- CodeFirst model generation from scratch or using existing database
- LINQ and EF
- Use of Windows Forms
- Data Annotations
- Database migration

Overview of Entity Framework

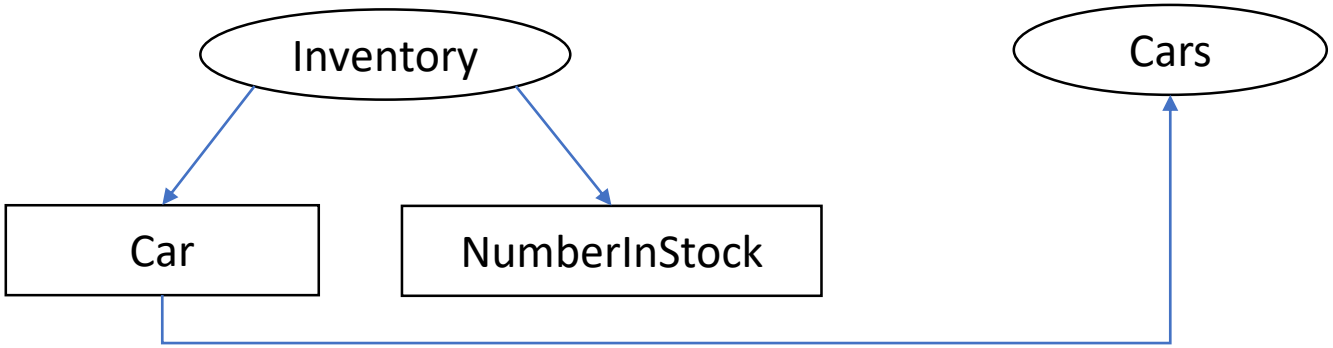
- Interact with data from relational databases using an object model that maps directly to the business objects (or domain objects) in your application. For example,
- Rather than treating a batch of data as a collection of rows and columns, instead operate on a collection of strongly typed **objects** termed *entities*.
  - Entities (objects) have associations/relationships between them
- These entities are also natively LINQ aware, and can query against them.
- The EF runtime engine translates LINQ queries into proper SQL queries.
- Similar to strongly typed DataSet/DataTable in Disconnected Layer.
- No SQL needed.

When to use diffent layers

- Connected: interact with a single table
- Disconnected: use of underlying SQL features such as stored procedures
- EF
  - Multi-table applications
  - Preference to use C# as the object relationship manager (ORM)

Entities

- An entity is a class that maps directly onto a table
- However, you can name/rename this class in C# to be anything
  - EF will take care of the mappings to the DB table automatically.
- Database relationships become object associations
  - Inventory and Cars objects get translated to DB tables



Database - DBContext

The Role of the DbContext Class

The `DbContext` class represents a combination of the Unit of Work and Repository patterns that can be used to query from a database and group together changes that will be written back as a single unit of work. `DbContext` provides a number of core services to child classes, including the ability to save all changes (which results in a database update), tweak the connection string, delete objects, call stored procedures, and handle other fundamental details. Table 23-1 shows some of the more commonly used members of the `DbContext`.

Table 23-1. *Common Members of DbContext*

Member of `DbContext`	Meaning in Life
`DbContext`	Constructor used by default in the derived context class. The string parameter is either the database name or the connection string stored in the `*.config` file.
`Entry` `Entry<TEntity>`	Retrieves the `System.Data.Entity.Infrastructure.DbEntityEntry` object providing access to information and the ability to perform actions on the entity.
`GetValidationErrors`	Validates tracked entries and returns a collection of `System.Data.Entity.Validation.DbEntityValidationResults`.
`SaveChanges` `SaveChangesAsync`	Saves all changes made in this context to the database. Returns the number of affected entities.
`Configuration`	Provides access to the configuration properties of the context.
`Database`	Provides a mechanism for creation/deletion/existence checks for the underlying database, executes stored procedures and raw SQL statements against the underlying data store, and exposes transaction functionality.

`DbContext` also implements `IObjectContextAdapter`, so any of the functionality available in the `ObjectContext` class is also available. While `DbContext` takes care of most of your needs, there are two events that can be extremely helpful, as you will see later in the chapter. Table 23-2 lists the events.

DbContext Methods and Properties

- SaveChanges()
  - Updates the database (emits SQL)
- Database.Log
  - Can be set to a delegate function where database traces (ie SQL code) can be processed
  - Database.Log = s => Debug.Write(s);
- Database.Create()
  - Creates the db if it does not exist or reinitializes
- Database.Delete()
  - Deletes the db (zeroes all the tables)

DbSet – a table

The Role of DbSet<T>

To add tables into your context, you add a DbSet<T> for each table in your object model. To enable lazy loading, the properties in the context need to be virtual, like this:

```
public virtual DbSet<CreditRisk> CreditRisks { get; set; }
public virtual DbSet<Customer> Customers { get; set; }
public virtual DbSet<Inventory> Inventory { get; set; }
public virtual DbSet<Order> Orders { get; set; }
```

Each DbSet<T> provides a number of core services to each collection, such as creating, deleting, and finding records in the represented table. Table 23-3 describes some of the core members of the DbSet<T> class.

Table 23-3. *Common Members of DbSet<T>*

Member of DbSet<T>	Meaning in Life
Add AddRange	Allows you to insert a new object (or range of objects) into the collection. They will be marked with the Added state and will be inserted into the database when SaveChanges (or SaveChangesAsync) is called on the DbContext.
Attach	Associates an object with the DbContext. This is commonly used in disconnected applications like ASP.NET/MVC.
Create Create<T>	Creates a new instance of the specified entity type.
Find FindAsync	Finds a data row by the primary key and returns an object representing that row.
Remove RemoveRange	Marks an object (or range of objects) for deletion.
SqlQuery	Creates a raw SQL query that will return entities in this set.

DBSet – other methods

- Load() – force load all records from the db
  - Eager loading
- Local – explicit local copy of the object/table
- Local.Clear() – will have the same effect as Remove, but will clear all records
  - Must follow by context.SaveChanges()
- DbSet<Inventory> Inventories
  - context.Inventories.Local.Clear() will clear all Car(s) from the set/table

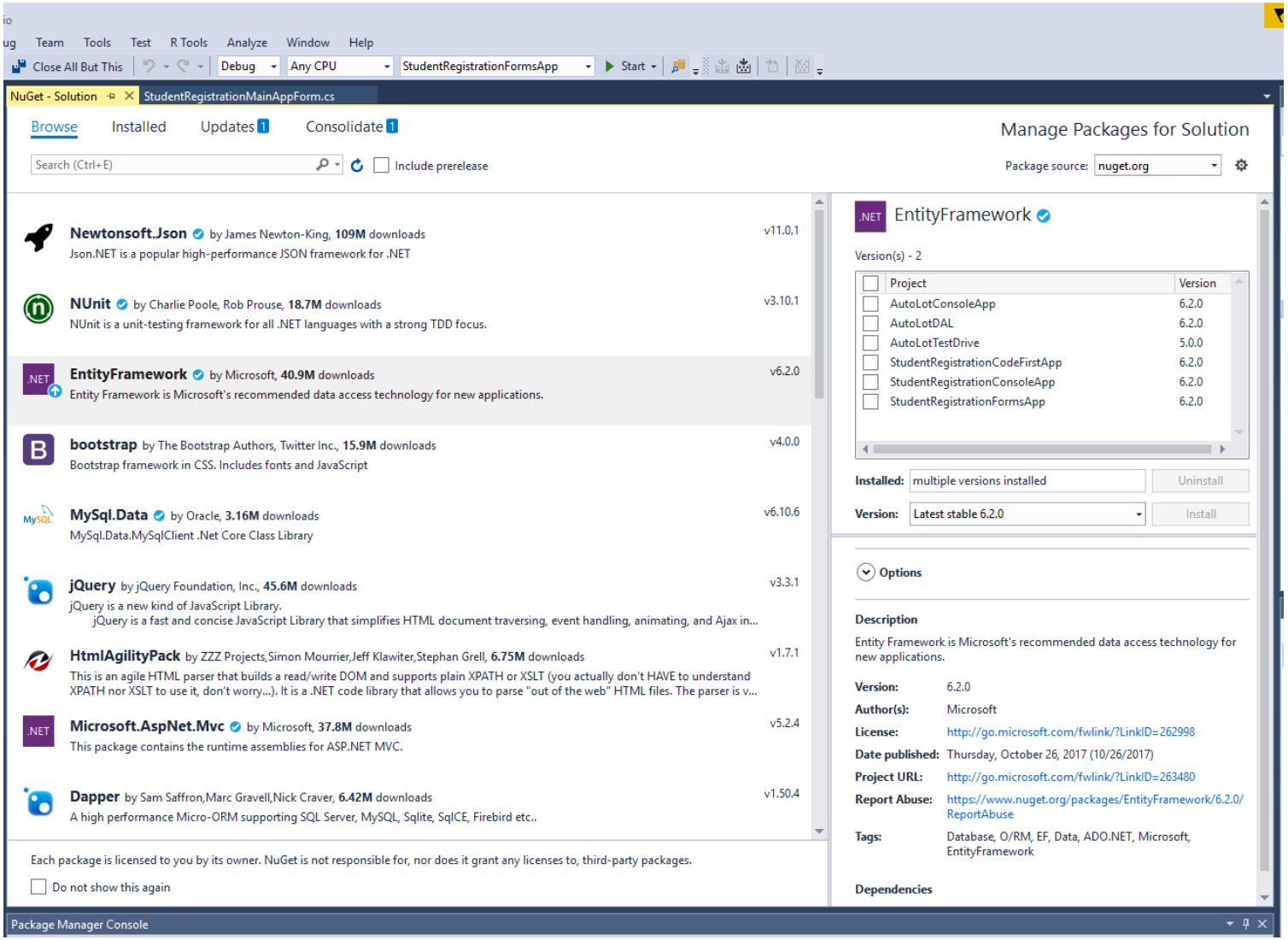
Entities and SQL

- Can execute a direct SQL command
  - MyEntities context = new MyEntities();
  - context.Database.ExecuteSqlCommand(sqlcmd);
  - Useful for maintenance functions
- DbContext tracks the state of all objects within it
  - Sends SQL commands when SaveChanges() is called.
- Entire application can be written without ever issuing specific SQL commands

Need to Add EntityFramework

- Need to add this for every project in a solution
- Includes executables to generate models and databases
- Tools
  - -> NuGet Package Manager
  - -> Manage NuGet Packages for Solution

Add EntityFramework to projects



CodeFirst approach

- If you have a database
  - VS will generate all of the classes you need to interact with the tables as objects
- If not
  - VS creates a skeleton for you to fill in
  - You create your own classes (that will map onto tables)
  - VS will generate the DB for you automatically
- An older approach uses Entity Framework Designer, which will NOT be supported in the next release of EF.
  - Future versions of VS will have more powerful diagramming tools as well
- We will only use CodeFirst!!

CodeFirst from a Database

- Let's look at StudentRegistrationConsoleApp first
- Need to create a database first
  - Create a blank database named **StudentRegistration** exactly under localdb
  - Set the StudentRegistration project to be the startup project
    - Take a look at the tables and relations
  - Hit Start, and the database tables will be created.
  - Check this in the Object Explorer
- See also AutoLotConsoleApp, which is based on the AutoLot database

Generate model (classes)

- Create a new Console project -- StudentRegistrationTest
- Create a new **EF Classes** folder
- Right Click on the **EF Classes** folder
  - Add ADO.NET model under Data
  - Name it **StudentRegistrationTestEntities**
    - This is the database (DBContext)
  - Follow the wizard, create from database
    - Use StudentRegistration as the database
    - Define StudentRegistrationConnection as the connection string
    - Examine App.Config
- Classes will be created in the **EF Classes** folder.

StudentRegistrationEntities

- Take a look at the test project classes created.
- It will look similar to below
- We will now use StudentRegistrationConsoleApp project, which is a complete implementation of a demo app

```
public StudentRegistrationEntities()
    : base("name=StudentRegistrationConnection")
{
}

public virtual DbSet<Course> Courses { get; set; }
public virtual DbSet<Department> Departments { get; set; }
public virtual DbSet<Student> Students { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Course>()
        .HasMany(e => e.Students)
        .WithMany(e => e.Courses)
        .Map(m => m.ToTable("Registration")
            .MapLeftKey(new[] { "CourseId", "DepartmentId" }).MapRightKey("StudentId"));

    modelBuilder.Entity<Department>()
        .HasMany(e => e.Courses)
        .WithRequired(e => e.Department)
        .HasForeignKey(e => e.CourseDepartmentId)
        .WillCascadeOnDelete(false);
}
```

Student class

- Note
  - using System.ComponentModel – this will be needed in other files as well
  - Data Annotations in square brackets
    - Gives direction to the model implementation and UI interfaces

```
    using System;
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;
    using System.ComponentModel.DataAnnotations.Schema;
    using System.Data.Entity.Spatial;

    public partial class Student
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Student()
        {
            Courses = new HashSet<Course>();
        }

        public int StudentId { get; set; }

        [Required]
        [StringLength(50)]
        public string StudentFirstName { get; set; }

        [Required]
        [StringLength(50)]
        public string StudentLastName { get; set; }

        [StringLength(10)]
        public string StudentMajor { get; set; }

        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<Course> Courses { get; set; }
    }
```

Using the Student class

- Look at Program.cs
  - note **using StudentRegistrationConsoleApp.EF_Classes;** to pick up the various classes.
- Create the context (DBContext).
- Add the students to the context.Students entity.
- Save changes
- Done!
- Now we can query context.Students as any other collection.
  - Can use LINQ
- If you change class names, refactor the code!
  - In fact, EF generates Cours instead of Course, this has to be refactored.

```
var context = new StudentRegistrationEntities();

foreach (Student s in students)
{
    context.Students.Add(s);
}
context.SaveChanges();
```

Database Logging

- To look at the SQL being sent to the database, set the following.
- Log is set to a Delegate, which is Debug.Write
- Need to include **using System.Diagnostics;**
- Output window will contain SQL trace

```
using System.Diagnostics;

context.Database.Log = (s => Debug.Write(s));
```

Find and Delete records

- context.Student.Find(“Jim”).FirstOrDefault() will return a Student or null
- Then context.Student.Remove(myStudent) will remove the record
- Don’t forget to context.SaveChanges()

Using Windows Forms

- Exact same set of steps to use existing database
- See StudentRegistrationFormsApp
  - **using System.Data.Entity; // need this for ToBindingList()**
  - Columns were initialized with headers
  - Simply set DataGridView.DataSource to Local.ToBindingList()

```
// need Load() for BindingList, which allows the gridviews to be sorted/edited
            //  and sync'd to database

            context.Students.Load();
            context.Courses.Load();

            // show the Students and Courses tables

            dataGridViewStudents.DataSource = context.Students.Local.ToBindingList();
            dataGridViewCourses.DataSource = context.Courses.Local.ToBindingList();

            // set up registration gridview

            updateRegistration();
            context.SaveChanges();
```

No need for Registration table!

- many-to-many handled with object relationships instead
  - look at class diagram
- Registration table updated automatically!
- Works in both directions!

```
// set up initial registration
            // note that we can add a student to a course or
            // add a course to a student, both will work and set up proper links

            courses[0].Students.Add(students[0]);
            courses[0].Students.Add(students[1]);
            courses[1].Students.Add(students[0]);
            courses[4].Students.Add(students[0]);

            students[2].Courses.Add(courses[2]);

            context.SaveChanges();
```

Use of LINQ

- Note use of actual class rather than anonymous
- Use of Data Annotations so column headers are set up correctly.
- In the next project, we will do this for all gridviews

```
 // Notice no need to use a join in EF, as the parent-child link exists
            //  so we can just walk from parent to child (Student to Courses)
            // If we use an anonymous class, the gridview can't be sorted or edited by the user
            // So we use StudentCourseRegistration and set the fields from the Students and Courses
tables

            var query =
                 from student in context.Students
                 from studentreg in student.Courses
                 orderby student.StudentLastName
                 select new StudentCourseRegistration
                 {
                     StudentID = student.StudentId,
                     StudentLastName = student.StudentLastName,
                     CourseID = studentreg.CourseId,
                     Department = studentreg.CourseDepartmentId,
                     CourseName = studentreg.CourseName,
                 };

            // set gridview datasource to a List

            dataGridViewStudentRegistration.DataSource = query.ToList();
```

Register and Add Student events

- Register
  - Note use of row selection in gridview
  - Use of Find() and Add() DbSet methods on the Students and Courses tables
  - Registration table is behind the scenes
- Add Student
  - Use of new form
  - Note how data is passed
  - Use of Add() DbSet method
  - Is not really necessary, as gridview allows for adding rows

CodeFirst blank

- Set StudentRegistrationCodeFirstApp as startup project
- Note creation of EF Classes folder
- Running the wizard ONLY produces a few files and the App.Config
- You provide all of the classes
- Code walk through
  - RegistrationClasses.cs contains each Entity
    - Note use of ForeignKey()
  - StudentRegistrationEntities.cs not much different
  - Better use of object relationships
  - No Registration table (although one is created for us)
- All three tables are displayed, with a report table showing registration.
- Extensive use of Data Annotations in classes
  - Helps with gridview – note gridview is NOT preconfigured!!

Data Annotations

Table 23-5. *Data Annotations Supported by Entity Framework*

Data Annotation	Meaning in Life
`Key`	Defines the primary key for the model. This is not necessary if the key property is named `Id` or combines the class name with `Id`, such as `OrderId`. If the key is a composite, you must add the `Column` attribute with an `Order`, such as `Column[Order=1]` and `Column[Order=2]`. Key fields are implicitly also `[Required]`.
`Required`	Declares the property as not nullable.
`ForeignKey`	Declares a property that is used as the foreign key for a navigation property.
`StringLength`	Specifies the min and max lengths for a string property.
`NotMapped`	Declares a property that is not mapped to a database field.
`ConcurrencyCheck`	Flags a field to be used in concurrency checking when the database server does updates, inserts, or deletes.
`TimeStamp`	Declares a type as a row version or timestamp (depending on the database provider).
`Table` `Column`	Allows you to name your model classes and fields differently than how they are declared in the database. The `Table` attribute allows specification of the schema as well (as long as the data store supports schemas).
`DatabaseGenerated`	Specifies if the field is database generated. This takes one of `Computed`, `Identity`, or `None`.
`NotMapped`	Specifies that EF needs to ignore this property in regard to database fields.
`Index`	Specifies that a column should have an index created for it. You can specify clustered, unique, name, and order.

■ **Note** In addition to data annotations, EF supports a Fluent API to define your table structure and relationships. Although you saw a small example in the earlier section, the Fluent API is beyond the scope of this chapter. You can find more information on defining tables and columns using the Fluent API here:

Manage migrations

- When you want to update your model
- Open NuGet Package Manager Console from Tools menu
- Commands
  - enable-migrations
  - update-database
- Errors are sometimes hard to fix
- Look at Configuration.cs under Migrations
  - Set **AutomaticMigrationsEnabled = true;**
  - **Warning: this may destroy data and it can change the DB schema**