Entity Framework

Alice Colella

Junior Developer @icubedsrl

Alice.Colella@icubed.it

Roberto Ajolfi

Senior Consultant @icubedsrl

Roberto.Ajolfi@icubed.it

Entity Framework

ORM di Microsoft basato sul .NET Framework

Insieme di tecnologie ADO.NET per lo sviluppo software

Definisce un modello di astrazione dei dati

Traduce il nostro codice in query comprensibili dal DBMS

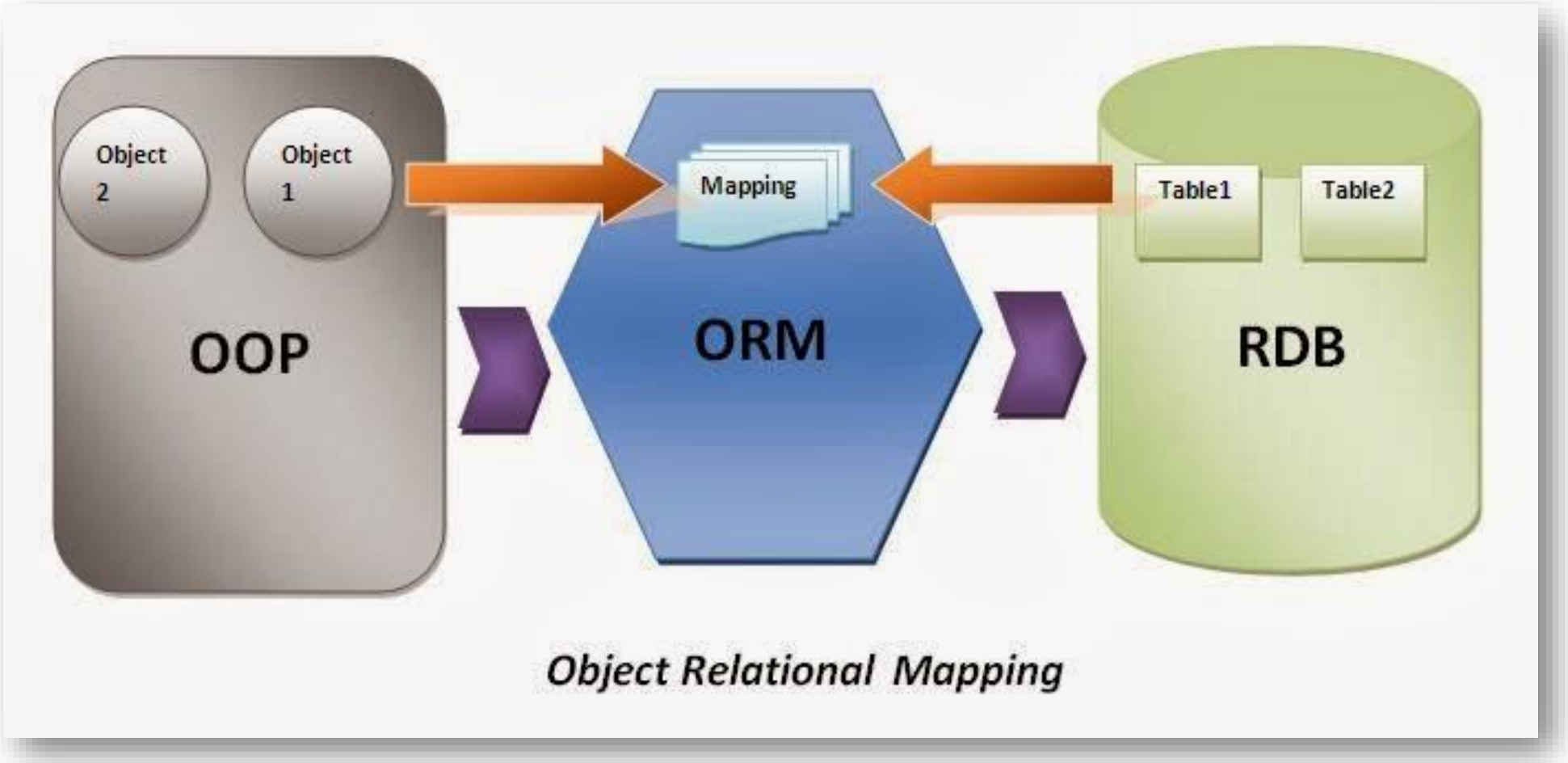
Disaccoppiamento tra applicazione e dati

- Posso mantenere la stessa rappresentazione anche se cambia il modello fisico (es. da SQL Server ad Oracle)

Open source

- https://github.com/aspnet/EntityFramework

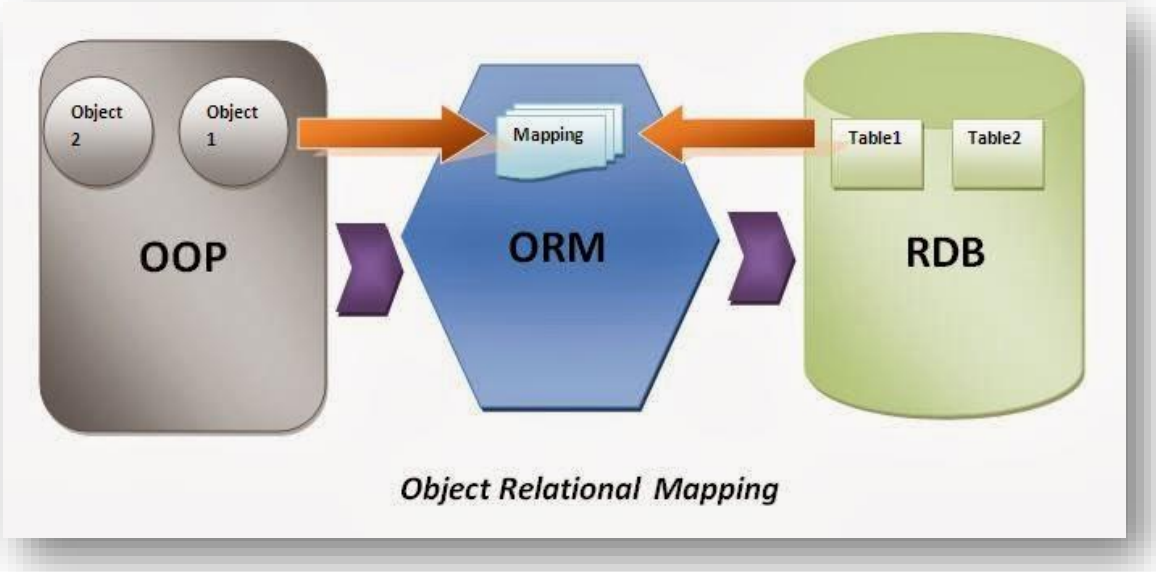
Cos’è un ORM?



Cos’è un ORM?

È una tecnica per convertire dati da type system incompatibili

Da database ad object-oriented



3 caratteristiche fondamentali

- **Mapping**
  - Definisce come il database si «incastra» negli oggetti e viceversa
- **Fetching**
  - Sa come recuperare i dati dal database e materializzare i rispettivi oggetti
- **Persistenza del grafo**
  - Sa come salvare le modifiche agli oggetti, generando le query SQL corrispondenti

Entity Framework

Entity Client Data Provider

- Livello di astrazione, che rende utilizzabile EF con più sorgenti dati

Entity Data Model

- Rappresenta il modello di mapping tra database ed oggetti

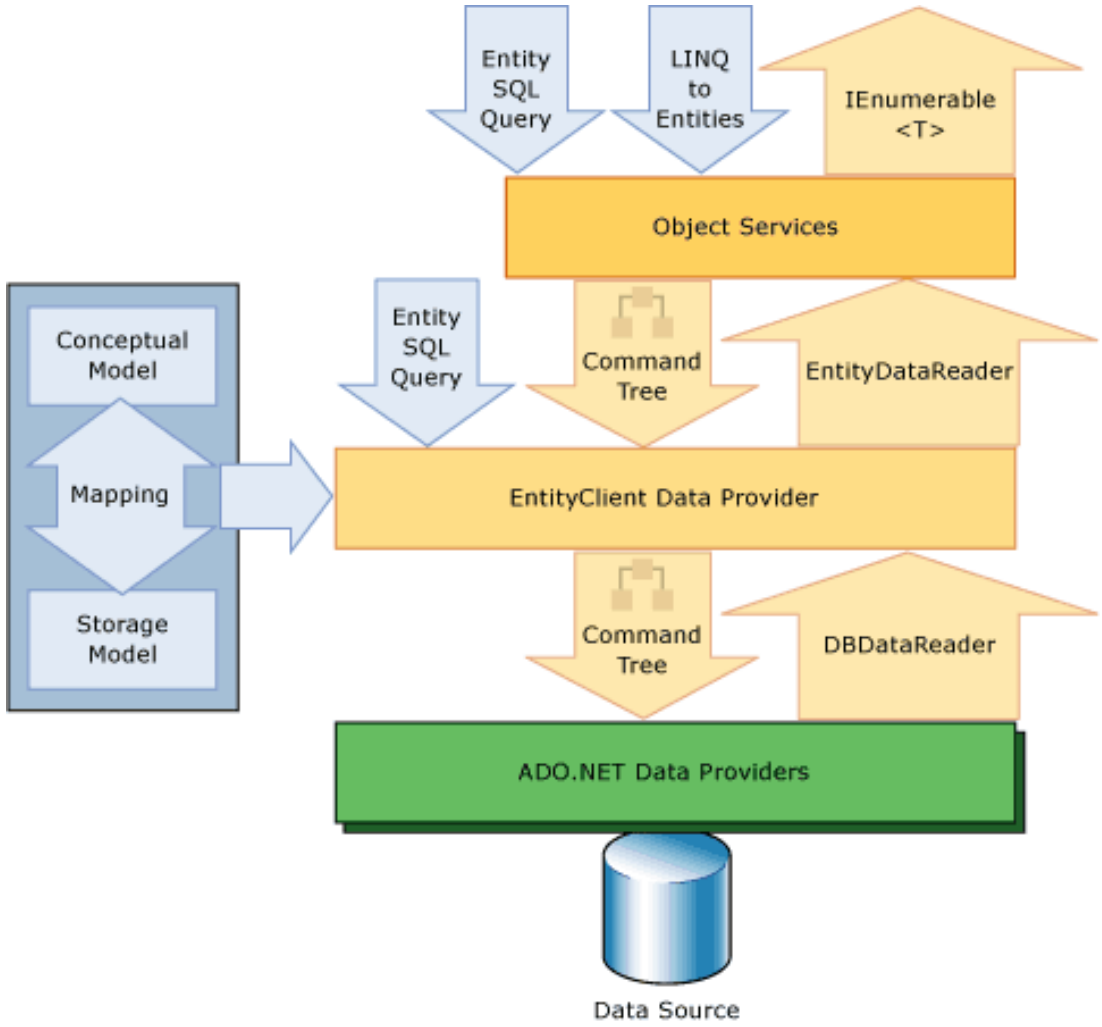
LINQ to Entities

- Flavour di LINQ che consente di utilizzare tutti gli operatori in unione con le entity di EF

Entity SQL

- Linguaggio speciale per interrogare EF con un linguaggio indipendente dal database utilizzato, consentendo di creare facilmente query dinamiche

Come funziona



icubed

Diversi approcci

Database-First

- Il modello viene importato da un DB esistente
- Se modifico il database posso (quasi) sempre aggiornare il modello

Model-First

- Il modello del database viene creato dal designer di Visual Studio
- L'implementazione fisica è basata sul modello generato
- Non favorisce il riutilizzo del codice né la separazione tra contesto ed entità
- Poichè il modello definisce il DB, eventuali sue modifiche verranno perse

Code-First

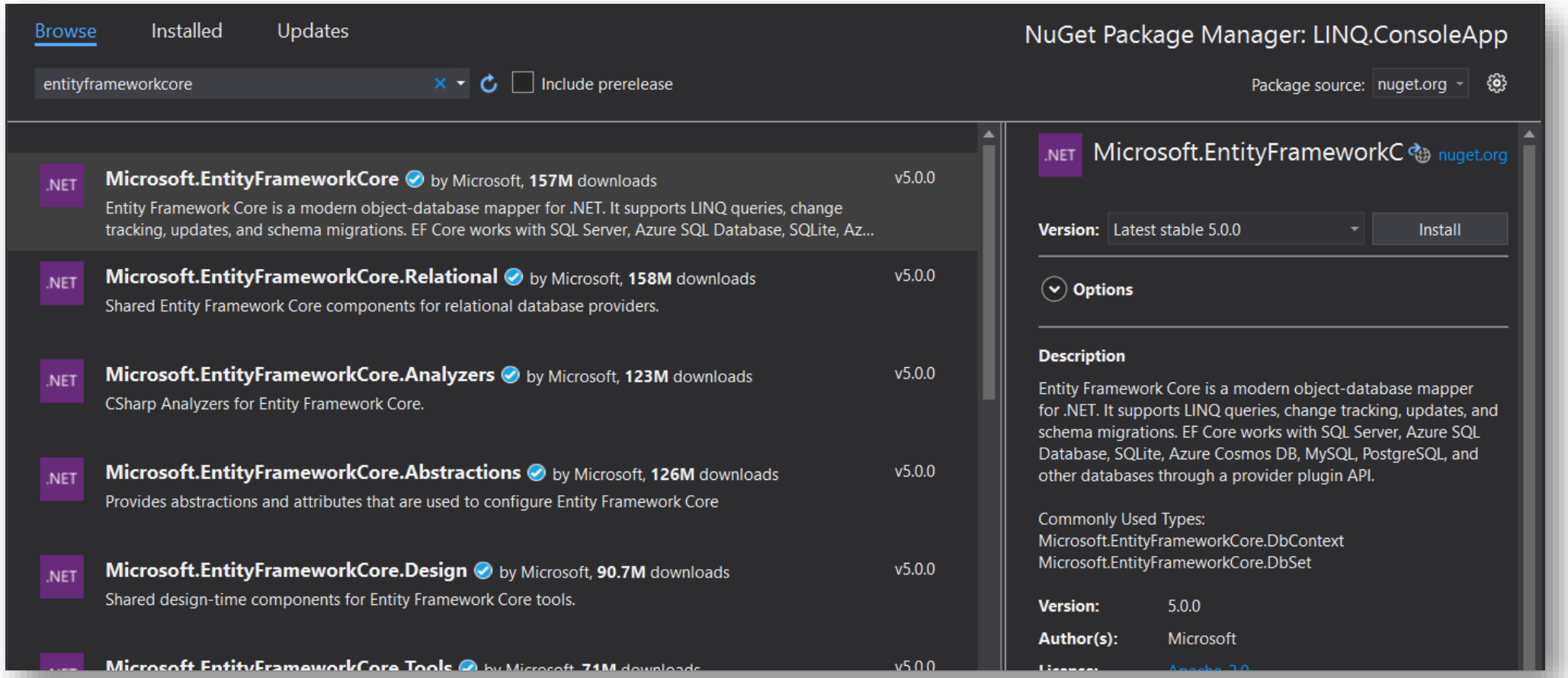
- Il modello viene creato dal nostro codice
- L'implementazione fisica è basata sul nostro codice

Perché Code-First?

- Focus sul domain design
- C# potrebbe risultarci più familiare delle query di SQL
  - E sarebbe l'unico linguaggio da apprendere
- Possiamo mettere facilmente sotto source control il nostro database (niente script SQL solo codice C#)
- Evitiamo la mole di codice auto generato da EDMX

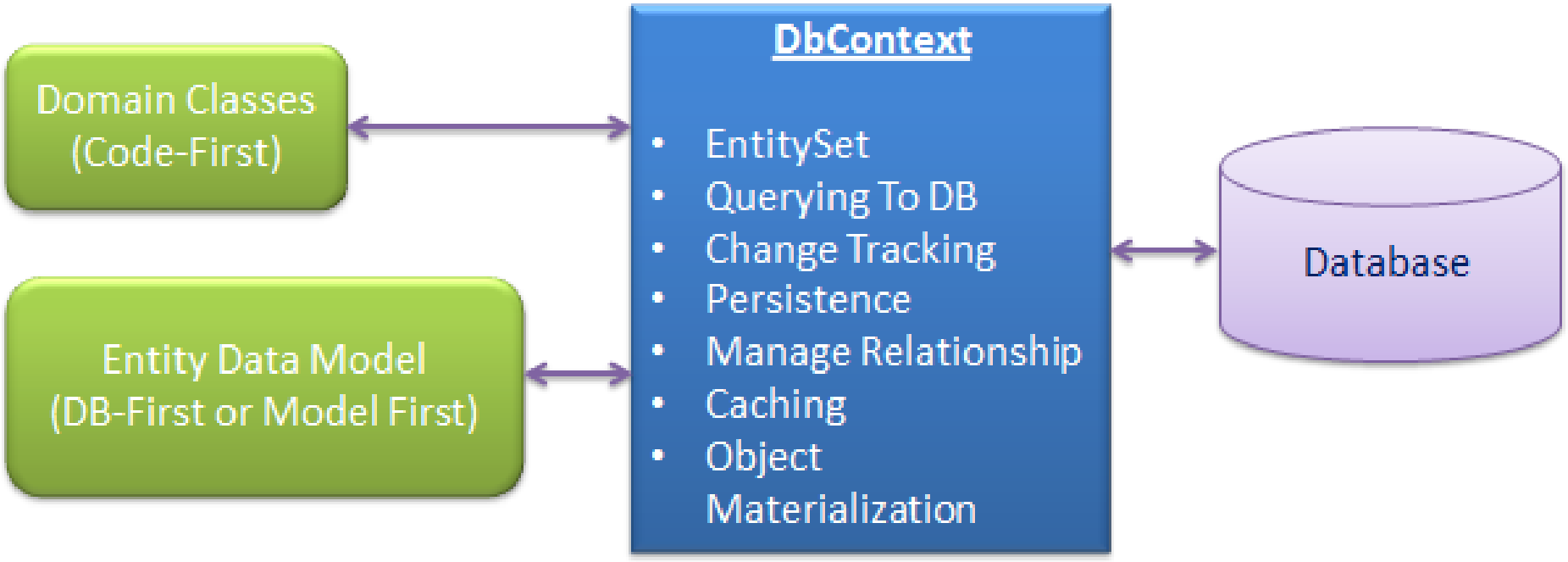
- Se scegliamo di sviluppare in .NET Core, l'EDMX non è supportato

Configurazione di EF



icubed

Il DbContext 1/2



Il DbContext

```
public class Context : DbContext
{
   public DbSet<Student> Students { get; set; }

   public Context() : base() { }

   public Context() : base("MyContext") { }
}
```

Il DbContext

```
public class Context : DbContext
{
   public Context(
      DbContextOptions<TicketingContext> options
   ) : base(options) { }
}
```

```
{
  // ...
  "ConnectionStrings": {
    "TicketingDb": "Server=tcp:democrito.database.windows.net,1433;Initial Catalog=Ticketing;
        Persist Security Info=False;User ID=sa;Password=xxxxxxx;MultipleActiveResultSets=False;
        Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"
  }
}
```

Il DbSet

E’ una classe che rappresenta le entity

Serve per fare operazioni CRUD

E’ definito come `DbSet<TEntity>`

I metodi più utilizzati sono:

- `Add, Remove, Find, SqlQuery`

```
public DbSet<Student> Students { get; set; }
```

Type Discovery

Nel *DbContext*:

```
public DbSet<Student> Students { get; set; }
```

Definizione della classe *Student*:

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime DateOfBirth { get; set; }

    public Teacher Teacher { get; set; }
}
```

Primary Key

```
public class Student
{
      public int StudentID { get; set; }

      public string StudentName { get; set; }
      public DateTime DateOfBirth { get; set; }
      public Teacher Teacher { get; set; }
}
```

```
public class Student
{
      public int MyPrimaryKey { get; set; }

      public string StudentName { get; set; }
      public DateTime DateOfBirth { get; set; }
      public Teacher Teacher { get; set; }
}
```

Convenzione sul nome della chiave primaria:

- *Id*
- *<NomeClasse>ID*

Non usa la convenzione di code-first

Genera una *ModelValidation Exception* se non gestita con le *DataAnnotations*

Foreign Key

```
public class Student
{
      public int StudentID { get; set; }

      public string StudentName { get; set; }
      public DateTime DateOfBirth { get; set; }

      public int CourseId { get; set; }
      public Course Course { get; set; }
}
```

```
public class Course
{
      public int CourseId { get; set; }

      public string CourseName { get; set; }
      public Teacher Teacher { get; set; }
      public ICollection<Student> Students { get; set; }
}
```

La *ForeignKey* viene generata automaticamente da code-first ogni volta che viene individuata una navigation property

Sempre bene rispettare le stesse convenzioni della *PrimaryKey*

Navigation Properties

Cos'è la proprietà Teacher?

```
public class Student
{
    // ...
    public Teacher Teacher { get; set; }
}
```

È una Navigation Property.

È la rappresentazione in EF di una Relazione tra due entità.

DataAnnotations e Fluent API

Le DataAnnotations sono attributi che servono a specificare il comportamento per fare l'override delle convenzioni di code-first

Possono influenzare le singole proprietà

- Namespace *System.ComponentModel.DataAnnotations*
- *Key, Required, MaxLenght…*

Possono influenzare lo schema del database

- Namespace *System.ComponentModel.DataAnnotations.Schema*
- *Table, Column, NotMapped…*

Le DataAnnotations sono limitate. Per il set completo bisogna andare di Fluent API

DataAnnotations: Key

Override della convenzione sulla *PrimaryKey*

Viene applicato alle proprietà di una classe

```
[Key]
public int MyPrimaryKey { get; set; }
```

DataAnnotations: Required

Indica al database che quella colonna non può essere NULL

In ASP.NET MVC viene usato anche per la validazione

```
[Required]
public string Name { get; set; }
```

DataAnnotations: MaxLenght, MinLenght

Possono essere applicati a stringhe o array

Possono essere usati in coppia

EntityValidationError se non rispettati durante una update

```
[MaxLenght(50), MinLenght(8)]
public string Name { get; set; }
```

DataAnnotations: Table

Rappresenta l’override del nome della tabella

Può essere solo applicato ad una classe e non alle proprietà

Si può anche inserire uno schema differente

```
[Table("Studente", Schema = "MySchema")]
public class Student { /* ... */ }
```

DataAnnotations: ForeignKey

Rappresenta l’override della convenzione sulla chiave esterna

Viene applicato solo alle proprietà di una classe

```
public class
{
    public i
    public i

    [Foreign
    public C
}

public class
{
    public i
    public I
}
```

```
public class Student
{
    public int StudentId { get; set; }
    public int CourseId { get; set; }

    public Course Course { get; set; }
}

public class Course
{
    public int CourseId { get; set; }
    public ICollection<Student> Students { get; set; }
}
```

icubed

DataAnnotations: DatabaseGenerated

Utile per chiavi primarie auto incrementanti

```
[DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public int Id { get; set; }
```

Fluent API

Sono una alternativa completa alle DataAnnotations

Si definiscono dentro l'override di *OnModelCreating*

Tre tipologie di mapping supportate:

- **Model**: Schema e convenzioni
- **Entity**: Ereditarietà
- **Property**: chiavi primarie/esterne, colonne e altri attributi

Model e Entity Mapping

Configurazione dello schema per tutto il database

Configurazione dello schema per singola tabella

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasDefaultSchema("Admin");

    //Map entity to table
    modelBuilder.Entity<Student>().ToTable("StudentInfo");
    modelBuilder.Entity<Student>().ToTable("StandardInfo", "anotherSchema");
}
```

Property Mapping

Configurazione della chiave primaria

```
modelBuilder.Entity<Student>().HasKey<int>(s => s.StudentId);
```

Configurazione di altre proprietà

```
modelBuilder.Entity<Student>().Property(p => p.Age)
                              .HasColumnName("Eta")
                              .HasColumnOrder(3)
                              .HasColumnType("datetime")
                              .IsRequired();
```

Fluent API Configurations 1/2

Tutte le configurazioni sono fatte via Fluent API

- Problema: troppo codice dentro *OnModelCreating*, diventa ingestibile!
- Soluzione: organizziamo le configurazioni

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfigurations<Student>(new StudentConfiguration());
    modelBuilder.ApplyConfigurations<Course>(new CourseConfiguration());
}
```

Fluent API Configurations 2/2

```
public class StudentConfiguration : IEntityTypeConfiguration<Student>
{
    public void Configure(EntityTypeBuilder<Student> builder)
    {
        builder.ToTable("StudentInfo");

        builder.HasKey<int>(s => s.StudentId);

        builder.Property(p => p.Age)
            .HasColumnName("Eta")
            .HasColumnOrder(3)
            .HasColumnType("datetime")
            .IsRequired();
    }
}
```

Relazioni molti-a-molti 1/4

Scenario: uno studente è iscritto a più corsi e ogni corso può avere più studenti

```
public class Course
{
    [Key]
    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public virtual  ICollection<Student> Students { get; set; }
}
```

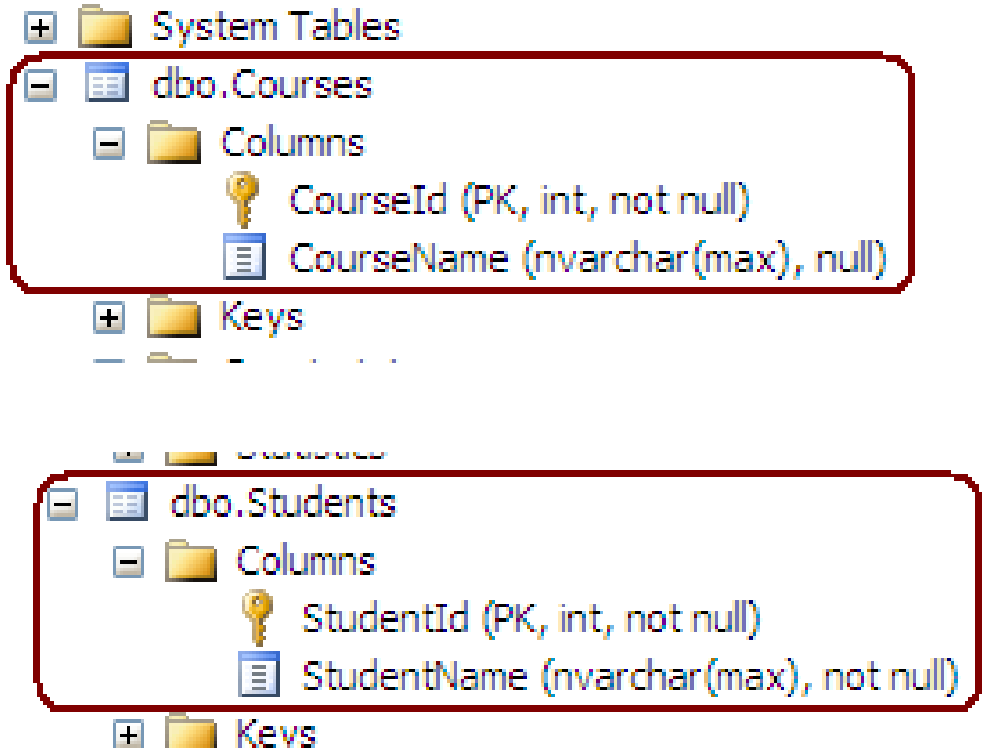
Relazioni molti-a-molti 2/4

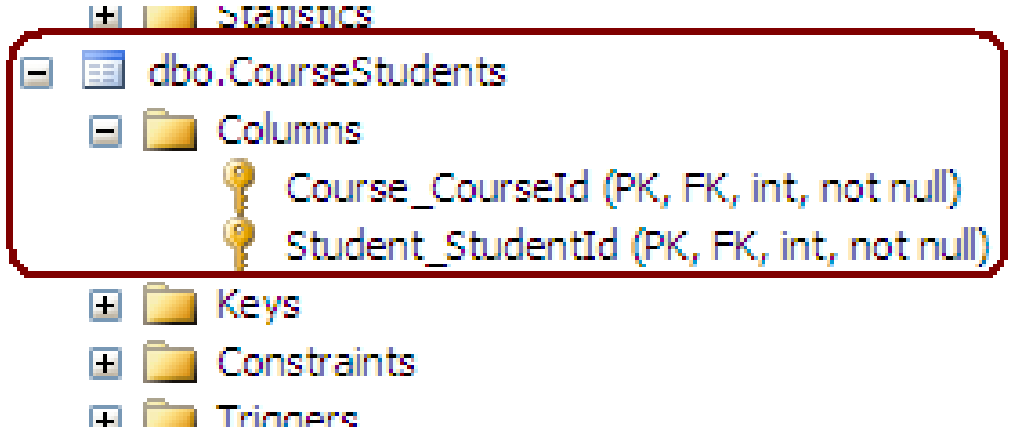
Scenario: uno studente è iscritto a più corsi e ogni corso può avere più studenti

```
public class Student
{
    public Student()
    {
        Courses = new List<Course>();
    }

    [Key]
    public int StudentId { get; set; }
    public string StudentName { get; set; }
    public virtual ICollection<Course> Courses { get; set; }
}
```

Relazioni molti-a-molti 3/4





icubed

Aggiungere dati

Utile sapere:

- Il pattern *IDisposable*
- *Async/await*

```
using (var ctx = new Context())
{
   var person = new Person("Matteo", "Tumiati");
   ctx.People.Add(person);
   await ctx.SaveChangesAsync();
}
```

Fare query sui dati

Utile conoscere:

- Il pattern *IDisposable*
- *Linq*
- I dati che si vogliono ottenere ☺

```
using (var ctx = new Context())
{
   ctx.Students.Where(x => x.Name.StartsWith("A"))
               .OrderBy(x => x.Age)
               .ToList();
}
```

Errori comuni 1/2

Giusto

```
var ages = dbContext.People
.Where(x => x.LastName.StartsWith("A"))
.OrderBy(x => x.Age)
.Where(x => x.City == "Bologna")
.ToList();
```

Sbagliato

```
var ages = dbContext.People
.Where(x => x.LastName.StartsWith("A"))
.ToList()
.OrderBy(x => x.Age)
.Where(x => x.City == "Bologna")
.ToList();
```

Errori comuni 2/2

Giusto

```
var person = dbContext.People.Find(1);
```

Quasi giusto ☺

```
var person = dbContext.People
                      .Where(x => x.Id == 1);
```

CUD (Create / Update / Delete)

EF offre diversi modi per aggiungere, aggiornare o eliminare i dati nel database. Un'entità verrà inserita o aggiornata o eliminata in base al valore della sua proprietà `EntityState`.

Esistono due scenari per salvare i dati di un'entità:

- **Connesso**: la stessa istanza di `DbContext` viene utilizzata per il recupero e il salvataggio delle entità

- **Disconnesso**: l'istanza di `DbContext` utilizzata per il recupero e il salvataggio delle entità è diversa

CUD (Create / Update / Delete)

Connected Scenario

Inserimento

```
using(var ctx = new MyContext())
{
    var prd = new Product()
    {
        ProductCode = "PR0001"
    };

    ctx.Products.Add(prd);

    ctx.SaveChanges();
}
```

CUD (Create / Update / Delete)

Connected Scenario

Update / Delete

```
using(var ctx = new MyContext())
{
    var prd = ctx.Products.FirstOrDefault<Product>();

    // UPDATE
    prd?.ProductCode = "PR0002";
    ctx.SaveChanges();

    // DELETE
    if(prd != null)
        ctx.Products.Remove(prd);
    ctx.SaveChanges();
}
```

CUD (Create / Update / Delete)

Disconnected Scenario

Inserimento

```
using(var ctx = new MyContext())
{
    var prd = new Product()
    {
        ProductCode = "PR0001"
    };

    ctx.Entry<Product>(prd).State = EntityState.Added;

    ctx.SaveChanges();
}
```

CUD (Create / Update / Delete)

Disconnected Scenario

Update / Delete

```
Product prd;
using(var ctx =
{
    prd = ctx.Pr
}
// ...
prd.FirstName =
// ...
using(var ctx = new MyContext())
{
    // UPDATE
    ctx.Entry<Product>(prd).State = EntityState.Modified;
    ctx.SaveChanges();
}
```

```
// ...
using(var ctx = new MyContext())
{
    // DELETE
    ctx.Entry<Product>(prd).State = EntityState.Deleted;
    ctx.SaveChanges();
}
```

Query Home-Made

Si può bypassare uno strato di Entity Framework che si occupa della traduzione della query da Linq to Entities

```
var people = dbContext.Database.SqlQuery<Person>(
      "SELECT * FROM Person WHERE FirstName LIKE 'a%'")
.ToList();
```

Utilizzare le Stored Procedure

SqlQuery mi permette anche di richiamare delle stored procedure.

```
CREATE PROCEDURE [dbo].[stpGetTicketByID]
                    @id int
                AS ...
```

```
var par1 = new SqlParameter("@id", 1);
var par2 = new SqlParameter("@active", true);

var people = dbContext.Ticket
.FromSqlRaw("exec stpGetTicketByID @id @active", par1, par2);
```

```
var people = dbContext.Ticket
.FromSqlRaw("exec stpGetTicketByID @id", param);
```

Utilizzare le Stored Procedure

```
var people = dbContext.Ticket
    .FromSqlRaw("exec stpGetTicketByID @id");
```

- Il risultato deve essere un tipo di entità
  - Ciò significa che una stored procedure deve restituire tutte le colonne della tabella corrispondente di un'entità
- Il risultato non può contenere dati correlati
  - Ciò significa che una procedura memorizzata non può eseguire JOIN per formulare il risultato
- Le procedure di inserimento, aggiornamento ed eliminazione non possono essere mappate con l'entità
  - quindi il metodo SaveChanges non può chiamare le procedure memorizzate per le operazioni CUD

Eseguire altri comandi SQL

- Le procedure di inserimento, aggiornamento ed eliminazione non possono essere mappate con l'entità
- Occorre utilizzare `ExecuteSqlCommand`

```
var rowsAffected = context.Database
  .ExecuteSqlCommand("Update Students set FirstName = 'Bill'
where StudentId = 1;");
```

Transazioni

In EF, il metodo `SaveChanges()` crea una **transazione implicita** e la usa per racchiude tutte le operazioni di INSERT, UPDATE e DELETE necessarie.

Se il codice contiene più chiamate al metodo `SaveChanges()`, EF crea più **transazioni implicite separate**, esegue le operazioni CRUD ed esegue il commit di ciascuna transazione.

Transazioni

Per creare ed utilizzare una singola transazione con più chiamate `SaveChanges()`:

- `DbContext.Database.BeginTransaction()`: crea una nuova transazione

```
using var transaction = ctx.Database.BeginTransaction();
try {
    // ...
    ctx.SaveChanges();
    transaction.Commit();
} catch(Exception ex) {
    transaction.Rollback();
}
```

Transazioni

- `DbContext.Database.UseTransaction()`: ci consente di passare un oggetto transazione esistente

```
using var transaction = ctx.Database.BeginTransaction();
// ...
try {
    // ...
    ctxTwo.Database.UseTransaction(transaction.GetDbTransaction());
    // ...
    ctxTwo.SaveChanges();
    transaction.Commit();
} catch(Exception ex) {
        transaction.Rollback();
}
```

Migrations

Sono il meccanismo che consente, utilizzando l'approccio code-first, l'aggiornamento del database a fronte di modifiche al modello.

Esitono 2 tipi di migrazioni

- *Automatiche*: poco invasive
- *Manuali o code-based*: richiedono un intervento specifico sul database

Migrazioni automatiche

Per abilitare le migrazioni bisogna avviare un comando dalla Package Manager Console

- `Enable-Migrations –EnableAutomaticMigration:$true`

Se il comando ha successo, allora verrà creato il file */Migrations/Configuration.cs* che rappresenta la nuova strategia di inizializzazione

```
internal sealed class Configuration : DbMigrationsConfiguration<Context>
{
    public Configuration()
    {
        AutomaticMigrationsEnabled = true;
    }

    protected override void Seed(Context context) { }
}
```

Migrazioni code-based

Sono utili quando vogliamo più controllo sulle modifiche automatiche.

Servono due comandi dalla Package Manager Console:

- `Add-Migrations «Migration name»`
  - Crea una nuova classe con tutte le modifiche rispetto allo stato precedente del db
- `Update-Database`
  - Aggiorna il database sulla base del modello

Si può anche fare rollback di una modifica:

- `Update-Database –Migration:"Migration name"`

Migrazioni code-based

Per ogni Migration, viene creato un nuovo file per ogni migrazione

- `TimeStamp + NomeMigrazione.cs`
- Il file contiene una classe che eredita da `Migration`
- Contiene due metodi `Up` e `Down` per l'aggiornamento del database

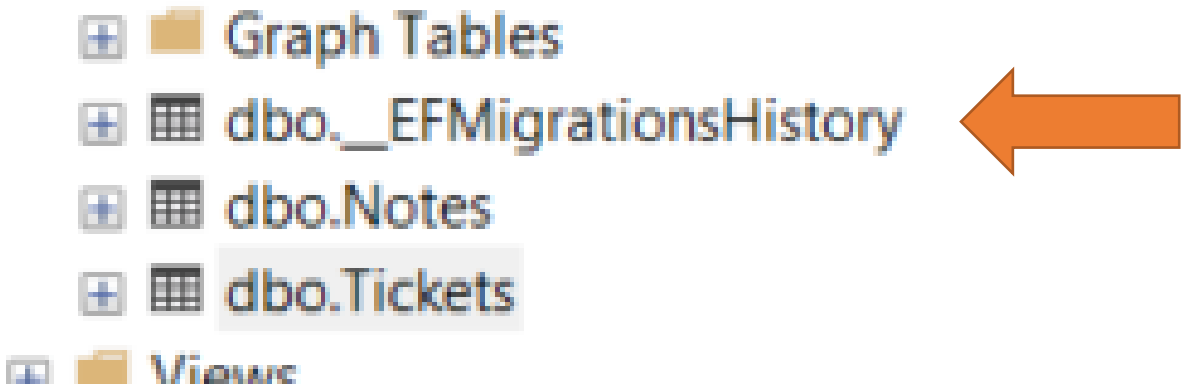
```
public partial class FirstMigration : DbMigration
{
      public override void Up()
      {
            CreateTable("dbo.Students", c => new {
                              StudentId = c.Int(nullable: false, identity: true),
                              Name = c.String(),
                              Age = c.Int(nullable: false)})
            .PrimaryKey(t => t.StudentId);
      }

      public override void Down()
      {
            DropTable("dbo.Students");
      }
}
```

icubed

Migrazioni

Se andiamo a vedere il nostro database…



Viene aggiunta una tabella al nostro database per mantenere lo storico delle Migration applicate.

Gestione della Concorrenza

- EF Core implementa il controllo ottimistico della concorrenza
- Nella situazione ideale, i cambiamenti effettuati non interferiranno tra loro e quindi potranno avere successo
  - Nel peggiore dei casi, due o più processi tenteranno di apportare modifiche in conflitto e solo uno di essi dovrebbe riuscire

Gestione della Concorrenza

Per gestire la concorrenza occorre aggiungere una proprietà `Timestamp` alle entità:

```
class Movie {
        // DATA ANNOTATION
        [Timestamp]
        public Byte[] RowVersion { get; set; }
        // ...
}
```

```
// FLUENT API
modelBuilder.Entity<Movie>()
    .Property(p => p.RowVersion)
    .IsRowVersion();
```

Gestione della Concorrenza

- Quando due o più processi vogliono modificare un record, EF effettua un check sul timestamp.
- Se il timestamp del record nel DbSet è diverso da quello nel db sono in presenza di un conflitto
  - occorre gestire l'eccezione `DbUpdateConcurrencyException`

```
try {
        // ...
        ctx.SaveChanges();
} catch (DbUpdateConcurrencyException ex) {
        // ...
}
```

Gestione della Concorrenza

La risoluzione del conflitto di concorrenza implica l'unione delle modifiche in sospeso dal DbContext corrente con i valori nel database

Sono disponibili tre set di valori per aiutare a risolvere un conflitto di concorrenza:

- Current Values: sono i valori che l'applicazione stava tentando di scrivere nel database
- Original Values: sono i valori originariamente recuperati dal database, prima che venissero apportate le modifiche
- Database Values: sono i valori attualmente memorizzati nel database

```
catch (DbUpdateConcurrencyException ex)
{
    foreach (var entry in ex.Entries)
    {
        if (entry.Entity is Person)
        {
            var proposedValues = entry.CurrentValues;
            var databaseValues = await entry.GetDatabaseValuesAsync();

            foreach (var property in proposedValues.Properties)
            {
                var proposedValue = proposedValues[property];
                var databaseValue = databaseValues[property];

                // TODO: decide which value should be written to database
                // proposedValues[property] = <value to be saved>;
            }

            // Refresh original values to bypass next concurrency check
            entry.OriginalValues.SetValues(databaseValues);
        }
        else
        {
            throw new NotSupportedException(
                "Don't know how to handle concurrency conflicts for "
                + entry.Metadata.Name);
        }
    }
```

L'eccezione contiene le entità su cui abbiamo avuto il conflitto

Per ogni entità posso accedere a Current Values e Database Values

Con queste informazioni posso decidere come gestire il conflitto.

… e poi tento di risalvare le modifiche

icubed

Caricamento dei Dati Correlati

Entity Framework Core consente di utilizzare le Navigation Property nel modello per caricare entità correlate

Esistono due modelli comuni utilizzati per caricare i dati correlati:

- **Eager Loading**: i dati correlati vengono caricati dal database come parte della query iniziale
- **Lazy Loading**: i dati correlati vengono caricati in modo trasparente dal database quando si accede alla proprietà di navigazione

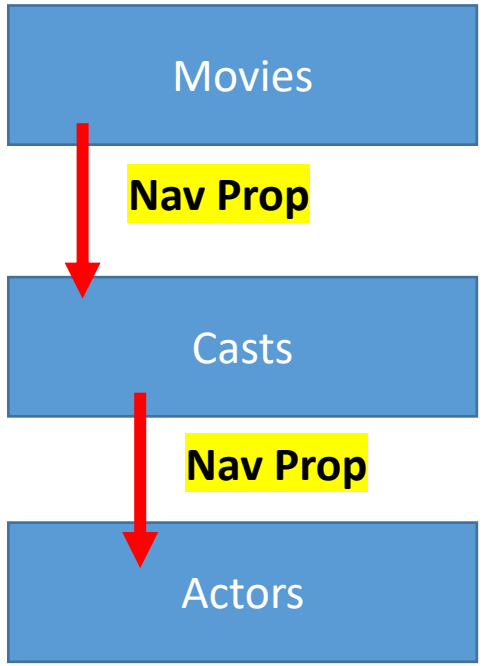
Caricamento dei Dati Correlati

Eager Loading

È possibile utilizzare il metodo `Include` per specificare i dati correlati da includere nei risultati della query

```
using var ctx = new MyContext();

var data = ctx.Movies
    .Include(m => m.Casts)
    .Include(c => c.Actors)
    .ToList();
```



Caricamento dei Dati Correlati

Lazy Loading – Metodo 1

Il modo più semplice per utilizzare il Lazy loading

- installare il pacchetto `Microsoft.EntityFrameworkCore.Proxies`
- abilitarlo con una chiamata a `UseLazyLoadingProxies()`

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);
}
```

Caricamento dei Dati Correlati

Lazy Loading – Metodo 1

EF Core abiliterà quindi il Lazy Loading per **qualsiasi** Navigation Property che **dovrà** essere `virtual` e di un tipo che può essere ereditato (**classe non `sealed`**)

```
class Actor  // non sealed
{
    // ...
    // Nav Prop virtual
    public virtual IEnumerable<Casts> Cast { get; set; }
}
```

Caricamento dei Dati Correlati

Lazy Loading – Metodo 2

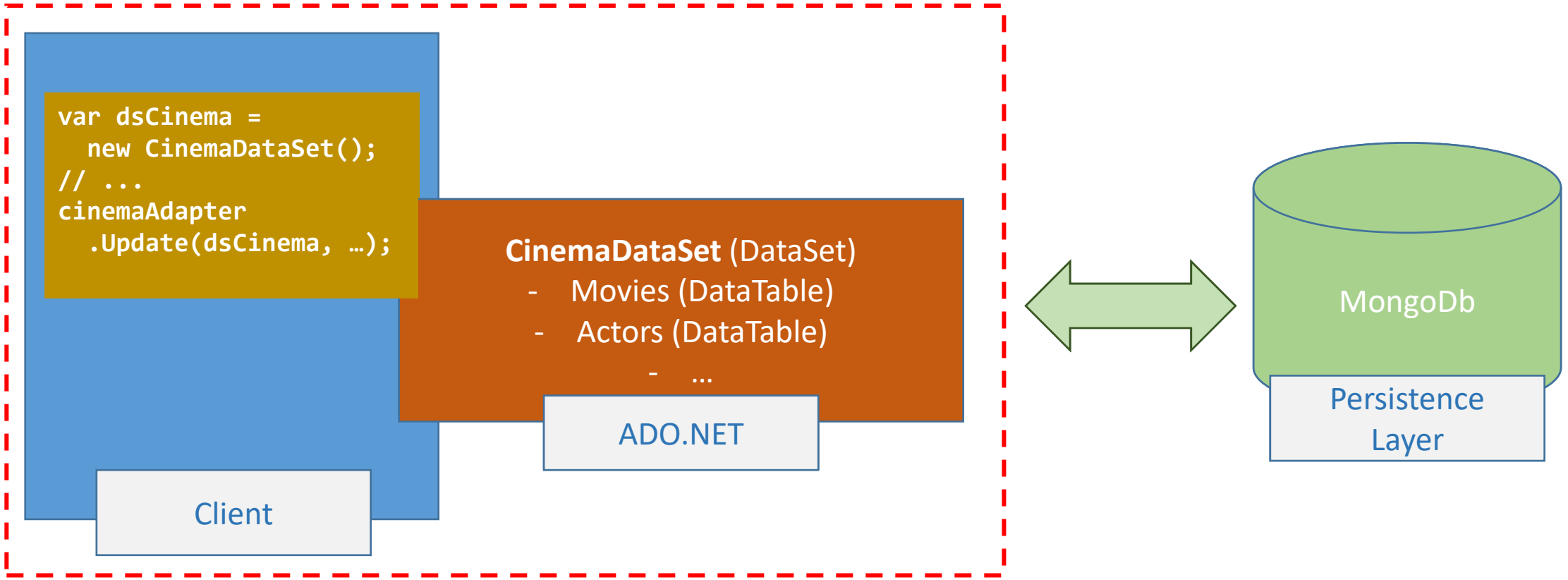
```
class Actor
{
    public Actor(ILazyLoader lazyLoader) {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }
    // ...
    private ICollection<Casts> _cast;

    public ICollection<Casts> Cast
    {
     get => LazyLoader.Load(this, ref _cast);
     set => _posts = value;
    }
}
```

Il Lazy Loading in EF Core può funzionare anche iniettando il servizio `ILazyLoader` in un'entità e modificando le Navigation Property.

Repository Pattern

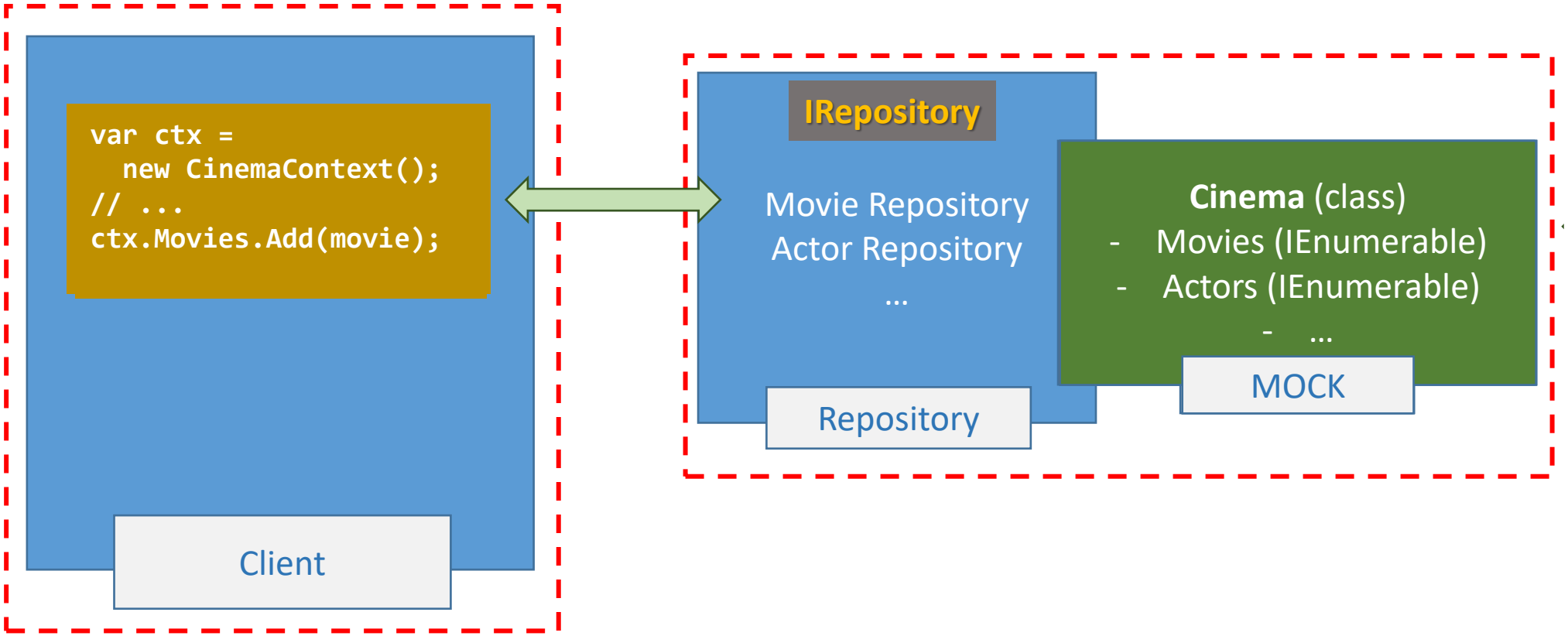


Repository Pattern

Un repository media tra il client e il/i layer di mappatura dei dati, agendo come una raccolta di oggetti in memoria

- gli oggetti possono essere aggiunti e rimossi dal Repository, così come da una semplice raccolta di oggetti
- il codice di mappatura incapsulato dal Repository eseguirà le operazioni appropriate dietro le quinte
- un Repository contiene tutto il codice relativo alla persistenza ma nessuna logica di business
- L‘obiettivo di questo pattern è quello di separare il codice per la persistenza dal codice di business

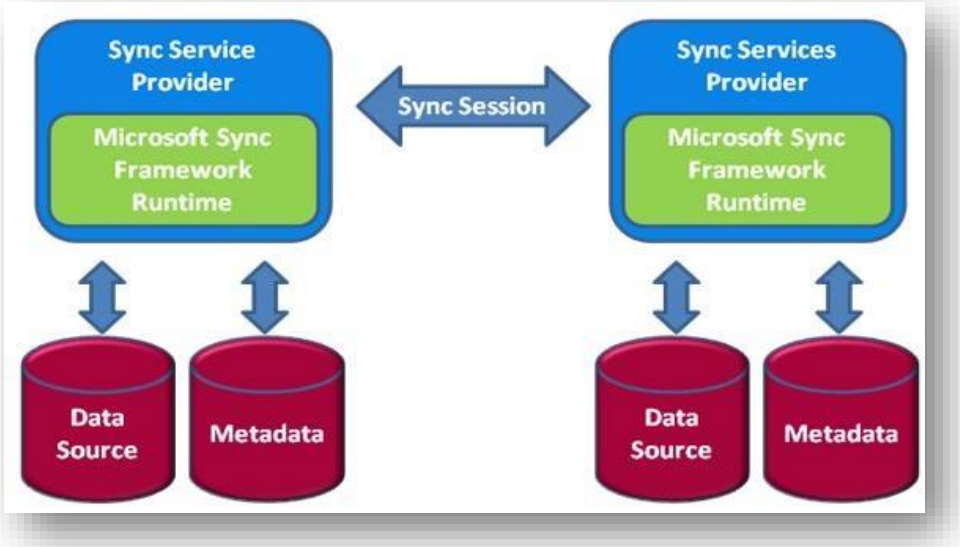
Repository Pattern



Sync Framework

- Microsoft Sync Framework è uno strumento **del 2011** che Microsoft aveva introdotto al principio dell'era Mobile

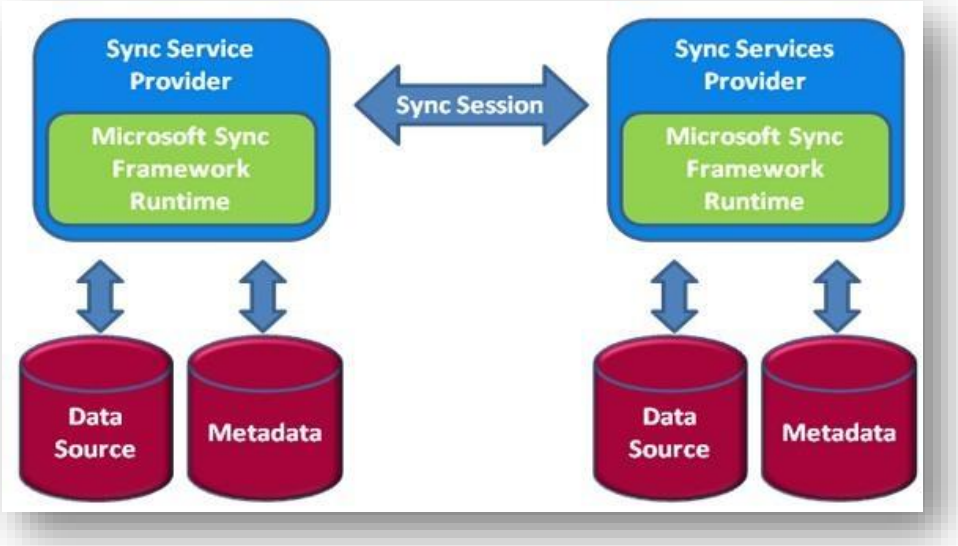
- Serviva per creare in automatico una esperienza di **sincronizzazione** tra una base dati Microsoft SQL Server Standard, con una corrispondente base dati Microsoft SQL Compact Edition



icubed

Sync Framework

- Ogni modifica fatta innescava una conseguente operazione di variazione dei dati sul database gemello
  - evitando allo sviluppatore di occuparsi delle tematiche infrastrutturali tra i due storage, e gestendo in automatico tematiche di conflitto o di ridondanze di informazione
- A seguito dell'avvento di REST, **Microsoft ha dismesso la piattaforma**, orientando le sue soluzioni sui database nativi esposti da HTML5



icubed

Web Services

Alice Colella

Junior Developer @icubedsrl

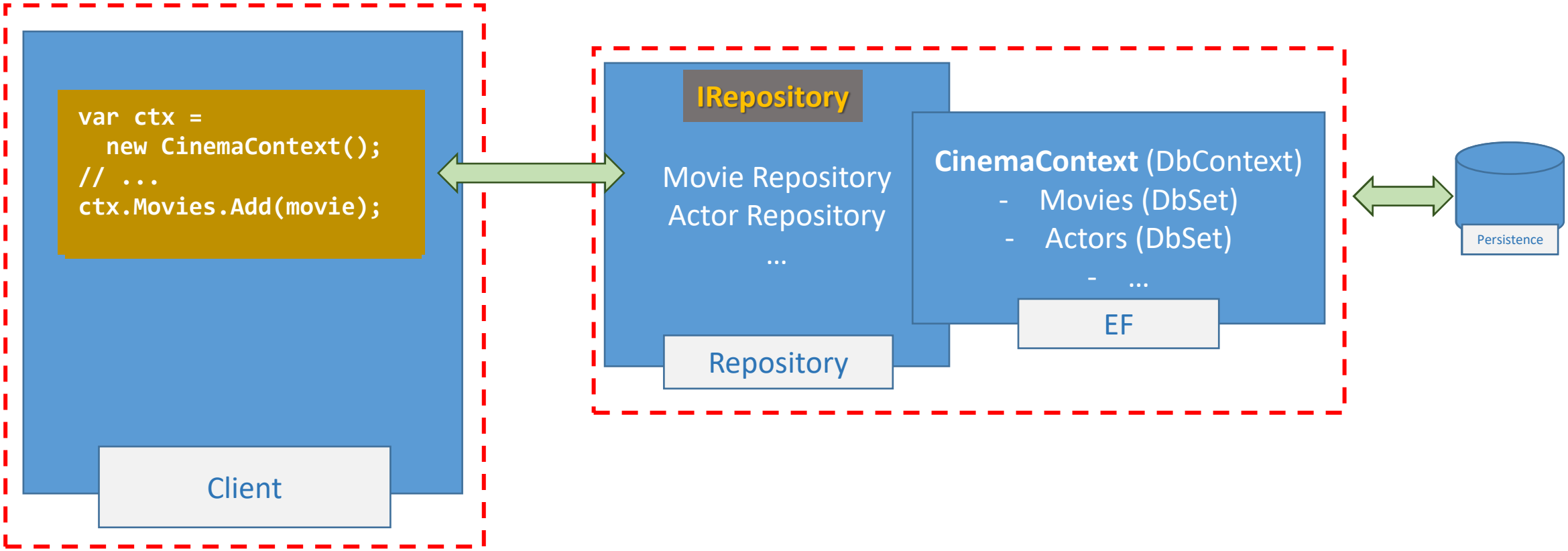
Alice.Colella@icubed.it

Roberto Ajolfi

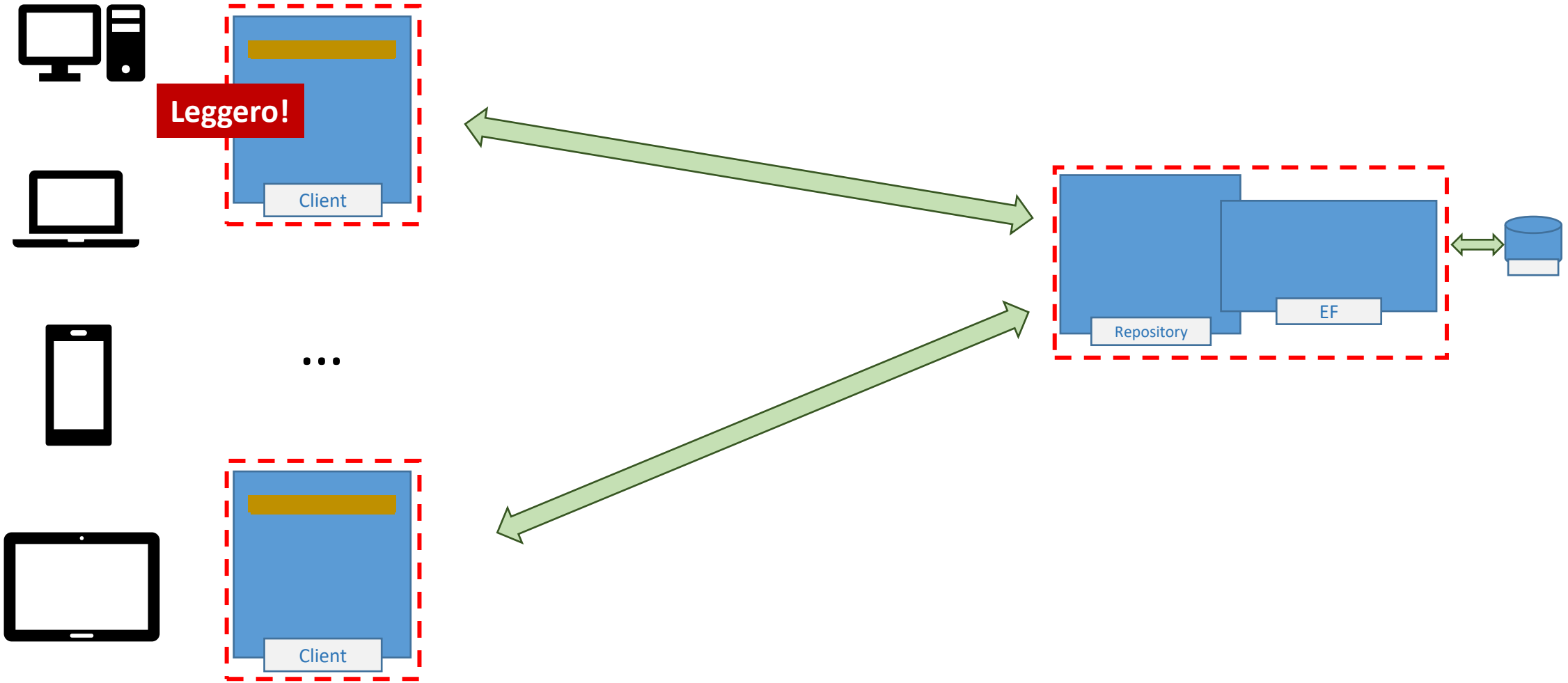
Senior Consultant @icubedsrl

Roberto.Ajolfi@icubed.it

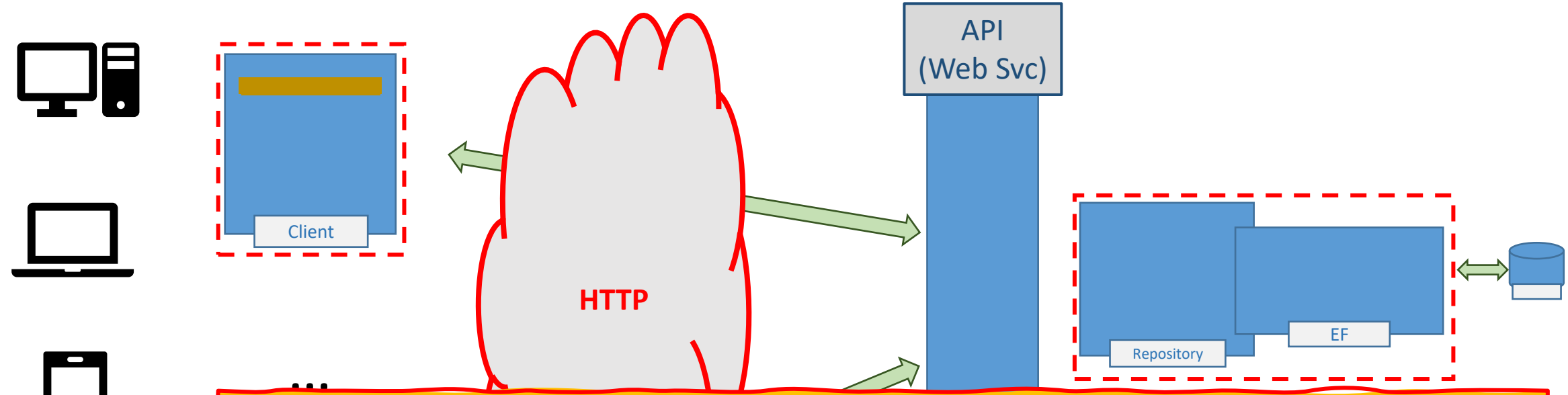
Data Services



Data Services



Data Services



Un'API (Application Programming Interface) è un'interfaccia che definisce le interazioni tra più software.

In particolare, definisce i tipi di chiamate o richieste che possono essere effettuate, come effettuarle, i formati dei dati da utilizzare, le convenzioni da seguire, ecc.

Tipologie di Web

- Si basano sul protocollo
- I dati vengono scambiati
- Lo scambio avviene defi
  dati all'interno di un file **V**
  Language)
- Per accedere al servizio
  faccia da **client** e delle c
  WSDL

```
<?xml version="1.0"?>
<definitions name="StockQuote"
          targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

 <types>
  <schema targetNamespace="http://example.com/stockquote.xsd"
        xmlns="http://www.w3.org/2000/10/XMLSchema">
   <element name="TradePriceRequest">
    <complexType>
     <all>
      <element name="tickerSymbol" type="string"/>
     </all>
    </complexType>
   </element>
   <element name="TradePrice">
    <complexType>
     <all>
      <element name="price" type="float"/>
     </all>
    </complexType>
   </element>
  </schema>
 </types>

 <message name="GetLastTradePriceInput">
  <part name="body" element="xsd1:TradePriceRequest"/>
 </message>

 <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetLastTradePrice">
   <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
   <input>
    <soap:body use="literal"/>
   </input>
   <output>
    <soap:body use="literal"/>
   </output>
  </operation>
 </binding>

 <service name="StockQuoteService">
  <documentation>My first service</documentation>
  <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
   <soap:address location="http://example.com/stockquote"/>
  </port>
 </service>

</definitions>
```

WCF

- In .NET l'implementazione di un SOAP Service viene fatta utilizzando WCF (Windows Communication Framework)
  - È un framework per la creazione, la configurazione e la distribuzione di servizi distribuiti in rete
  - Si basa sul concetto di comunicazione basata su **messaggi**, in cui una richiesta HTTP è rappresentata in modo uniforme
  - rende possibile avere un'API unificata indipendentemente dai diversi meccanismi di trasporto

- WCF è stato rilasciato per la prima volta nel 2006
  - Ultima versione: WCF Core 3.1.0 ➔ November 26, 2019

Tipologie di Web Service - REST

- E’ l’acronimo di **“Representational State Transfer”**, ed è definito come uno “stile architetturale software” che definisce una serie di vincoli per creare servizi web

- Fornisce interoperabilità tra sistemi complessi di computer su Internet

Tipologie di Web Service - REST

- Un “RESTful web service” permette di richiedere, accedere e manipolare risorse remote semplicemente usando rappresentazioni “testuali” delle stesse.
- Per la manipolazione si utilizzano indirizzi “uniformi” e un elenco predefinito di operazioni “stateless”

Accesso ai dati (REST Web Service)

Per accedere ai dati tramite un servizio REST bastano

- Un URL

```
https://servername/api/resourcename
```

- L'utilizzo degli HTTP Verbs

HTTP Verbs	CRUD Operations
GET	READ (one or many records)
POST	CREATE
PUT	UPDATE
DELETE	DELETE

Accesso ai dati (REST Web Service)

Esempi di chiamate

HTTP Verbs	URL	Body	Returns (*)
GET (list of resources)	`https://servername/api/resourcename`	-	JSON array of resources
GET (single resource by ID)	`https://servername/api/resourcename/ID`	-	Single resource as JSON
POST	`https://servername/api/resourcename`	Resource as JSON	-
PUT	`https://servername/api/resourcename/ID`	Resource as JSON	-
DELETE	`https://servername/api/resourcename/ID`	-	-

(*) errors in case of issues

icubed

Pragmatic REST oppure REST “Strict”

Come per ogni cosa nella vista, ci sono almeno due modi per fare le cose…e non è detto che uno dei due deve essere per forza quello sbagliato:

- REST “Strict” è un modo di sviluppare servizi REST (quindi API) seguendo i dettami di REpresentational State Transfer, che definisce che il modello dati deve essere sempre accessibile e gestibile dall’esterno

- Pragmatic REST è un approccio allo sviluppo di servizi REST che tende a centralizzare la logica di business sul servizio stesso invece che distribuirla sui client

Perché REST è così importante

- Un servizio REST è HTTP “nudo e crudo”, nessuna “sovrastruttura”
  - GET, POST, PUT, DELETE, OPTIONS, MERGE
  - Headers HTTP
  - Request all’interno del body in formato JSON (o in querystring)
  - Authentication passata all’interno di ogni singola richiesta

Perché REST è così importante

- Nessuna sessione server: completamente Stateless = meno carico server = maggiori prestazioni
- Molto più leggero di SOAP anche per applicazioni enterprise (meno overhead)
  - I risultati JSON sono “digeriti” da una SPA senza trasformazioni…è sempre JavaScript!

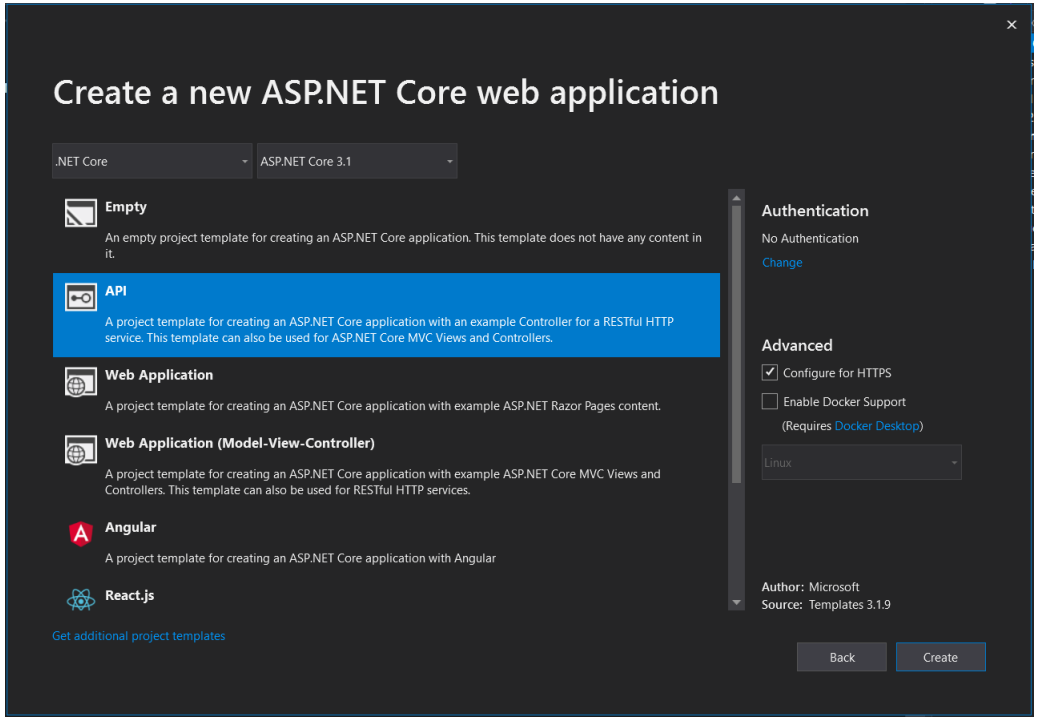
La Sicurezza nei servizi REST

- L’autenticazione viene gestita in genere tramite l’aggiunta di un Authentication Header alla chiamata HTTP
  - BASIC Authentication
  - JWT (Token based)
  - …
- HTTPS ! Always !!!

Servizi REST con ASP.NET Core

Per scrivere un servizio REST in .NET Core si utilizza

ASP.NET Core



ASP.NET Core è un framework per lo sviluppo web

- Open source
- Modulare
- Cross platform

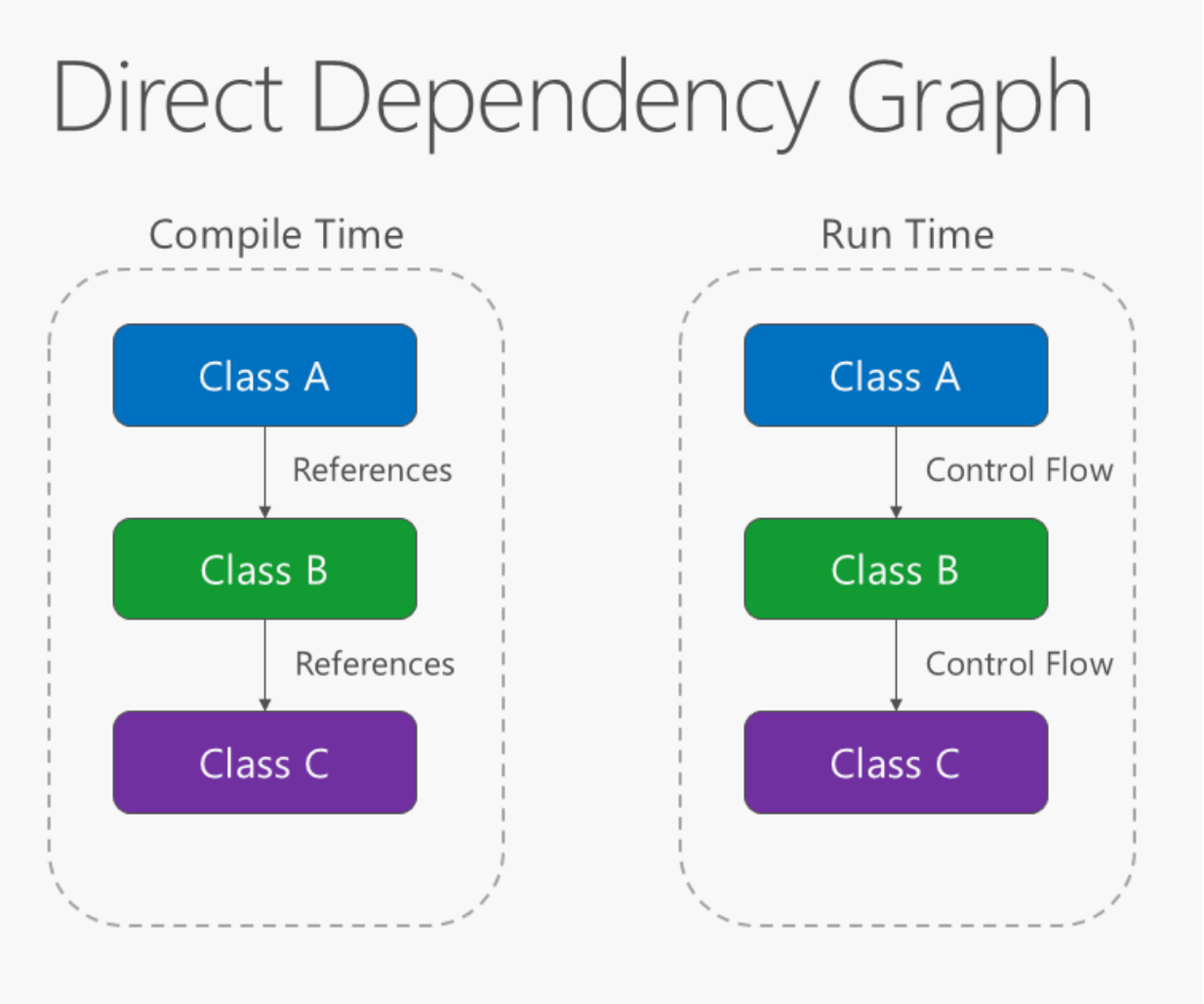
icubed

Demo

REST Service (ASP.NET Core)

icubed

Dependency Injection



Dependency Injection

Dependency injection (DI) è un design pattern della Programmazione orientata agli oggetti il cui scopo è quello di semplificare lo sviluppo e migliorare la testabilità di software di grandi dimensioni.

Per utilizzare tale design pattern è sufficiente **dichiarare le dipendenze** di cui un componente necessita (dette anche **interface contracts**).

Quando il componente verrà istanziato, un **iniettore** si prenderà carico di risolvere le dipendenze (attuando dunque l'inversione del controllo).

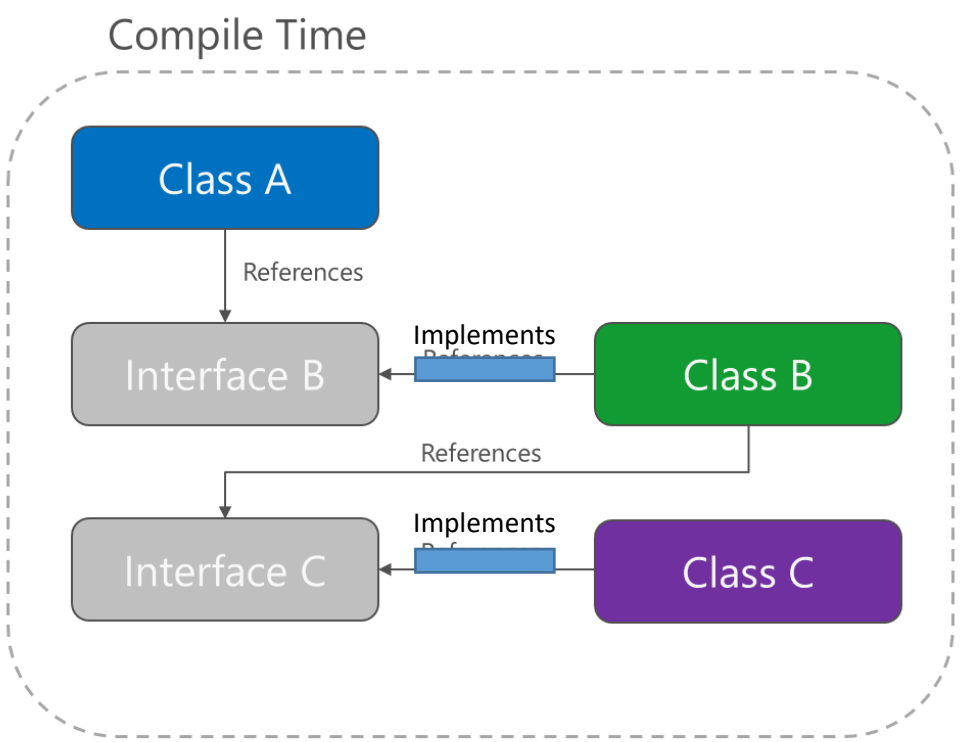
Dependency Injection

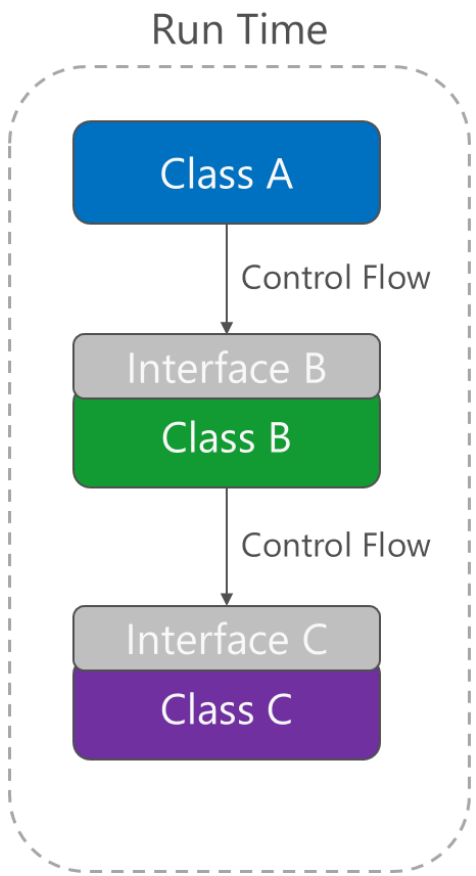
Il pattern Dependency Injection coinvolge almeno tre elementi:

- una componente dipendente
- la dichiarazione delle dipendenze del componente, definite come interface contracts
- un injector (chiamato anche provider o container) che crea, a richiesta, le istanze delle classi che implementano delle dependency interfaces

Dependency Injection

Inverted Dependency Graph





1cubed

© 2020 iCubed Srl

iCubed s.r.l.

Piazza Durante, 8 20131 MILANO

Phone: +39 02 57501057

P.IVA 07284390965