Implementing Data Access and Dependency Injection

Alex Wolf
.NET Developer

www.thecodewolf.com

Exploring Database Options

Relational Databases

- SQL Server
- MySQL
- Postgres
- Oracle

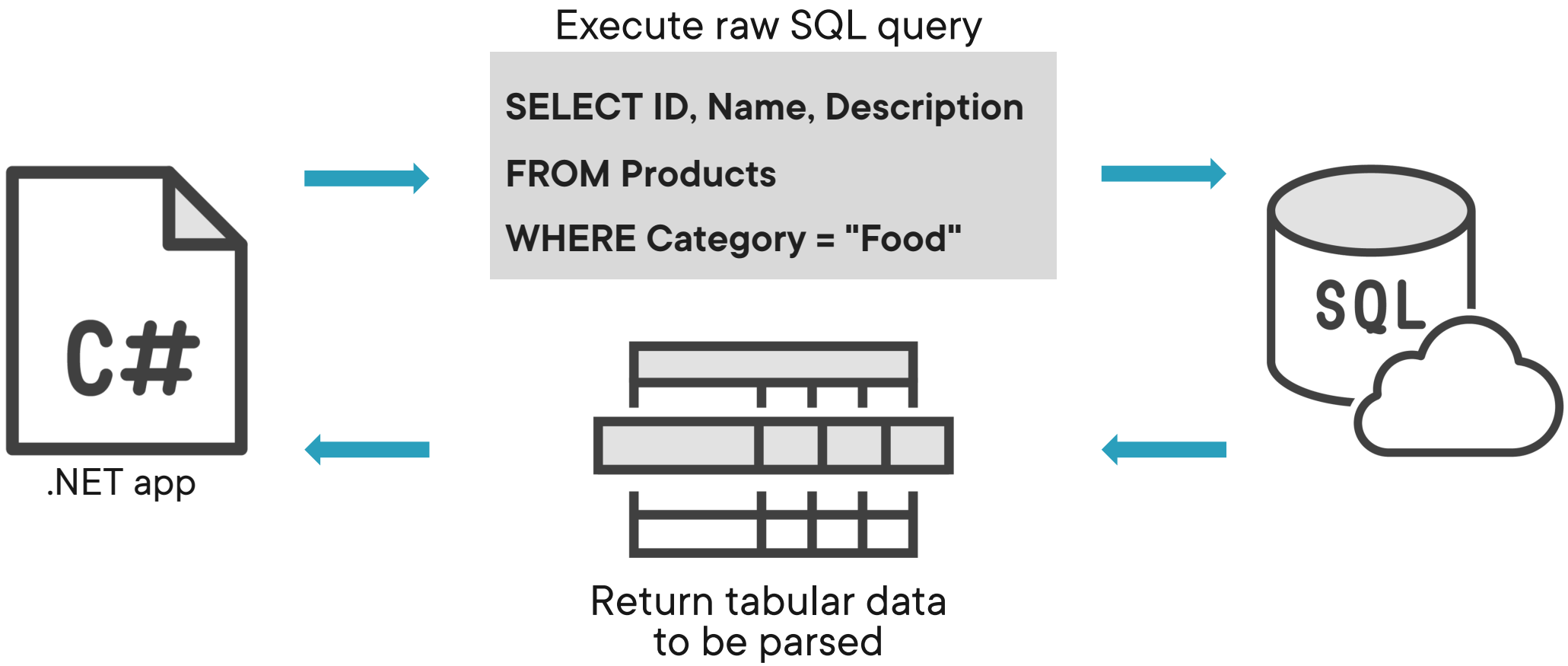
NoSQL Databases

- CosmosDB
- DynamoDB
- MongoDB
- Redis

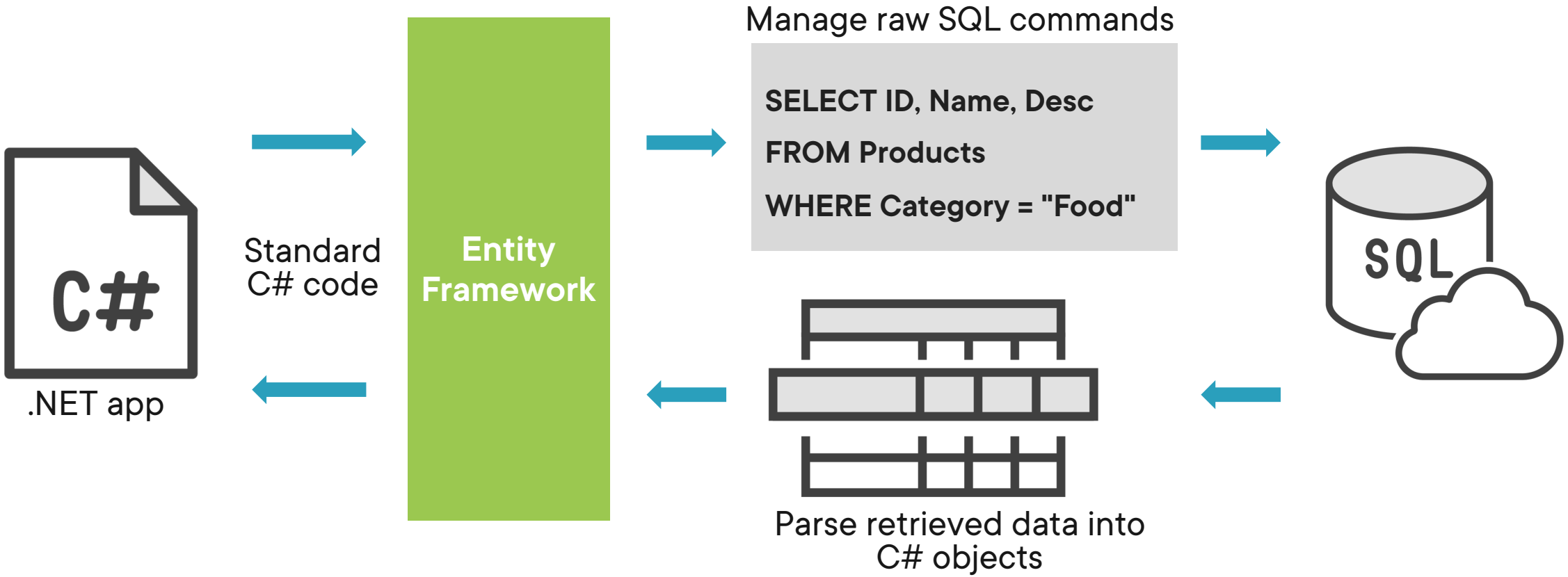
Entity Framework Core

A lightweight, cross-platform object-relational mapper for .NET.

Traditional Database Workflows



Understanding Object-Relational Mappers



A note about Entity Framework.

More information

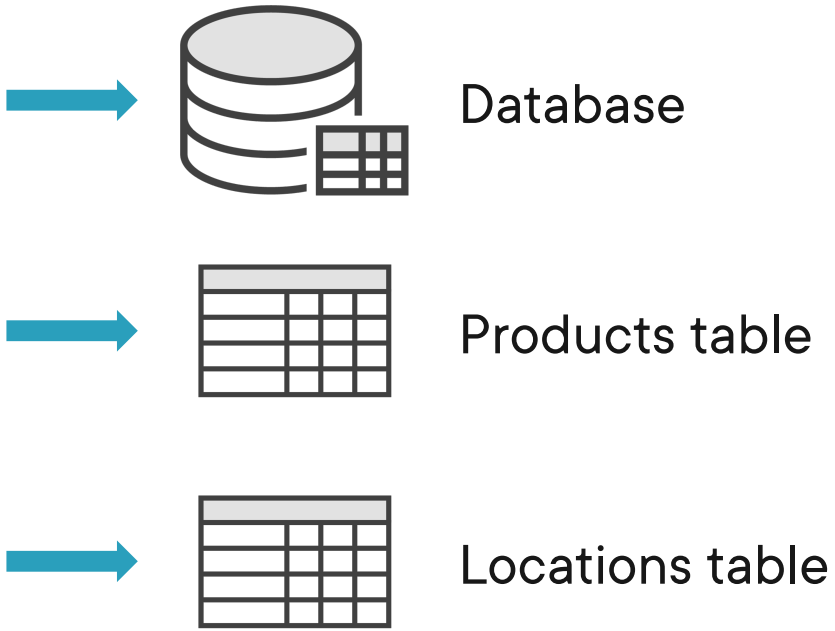
Entity Framework Core: Getting Started

Julie Lerman

Entity Framework Core Concepts

Essential Entity Framework Components

```
WiredContext  :  DbContext
{

    DbSet<Product> Products


    DbSet<Location> Locations
}
```



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

[MaxLength(500)]
public string Description { get; set; }

[NotMapped]
public IFileUpload Upload { get; set; }

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

- Database column cannot be empty
- Database column cannot exceed 500 chars
- Property should not be mapped to database
- Configure database primary key

Entity Framework Approaches

Code First

Create the database from the defined code model

Database First

Generate the code model from the database

```
builder.Services.AddDbContext<WiredContext>(
    options => options.UseSqlServer(
        builder.Configuration.GetConnectionString("WiredBrain"))
));
```

Registering the DbContext

The DbContext is registered in the program.cs file

The registration also configures the database type and connection settings

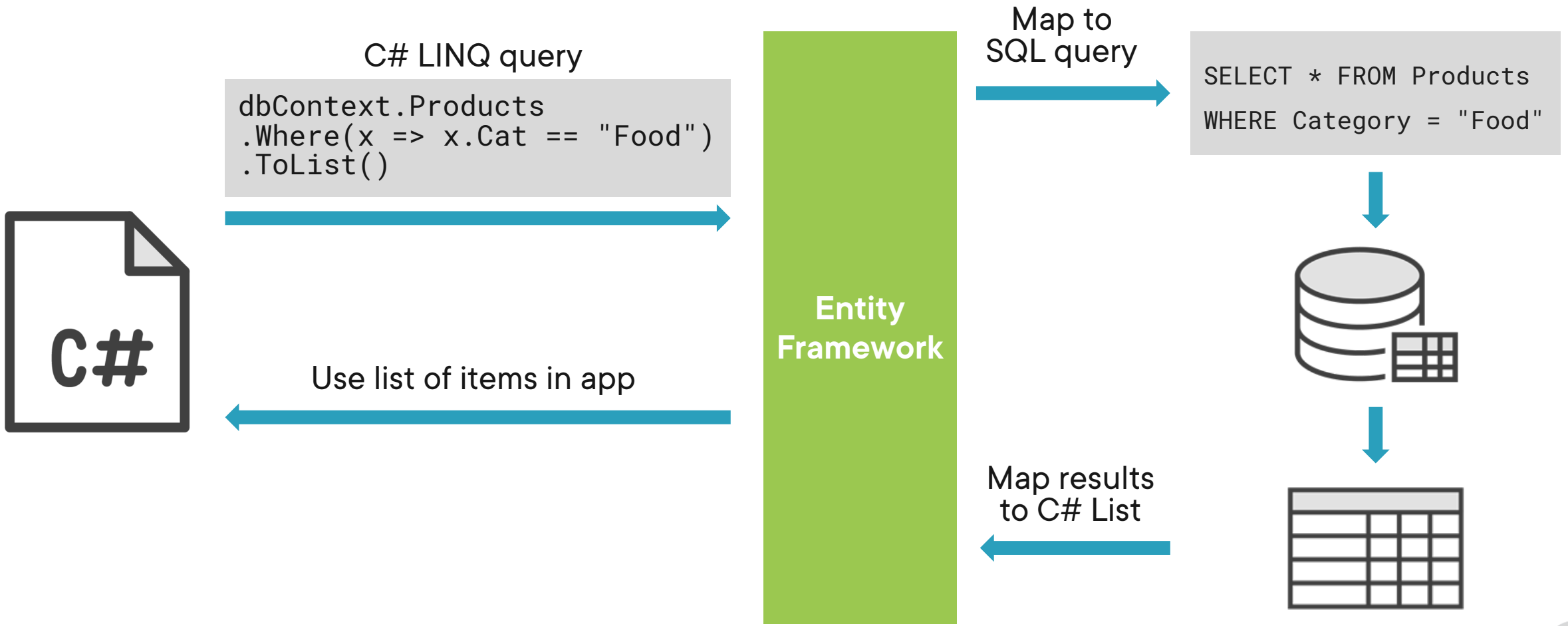
```
if (ModelState.IsValid)
{
            dbContext.Products.Add(NewProduct);

    dbContext.SaveChanges();

    var products =    dbContext.Products.ToList()

    return RedirectToPage("AllProducts");
}
```

- Add new product
- Commit changes to database
- Retrieve products from database

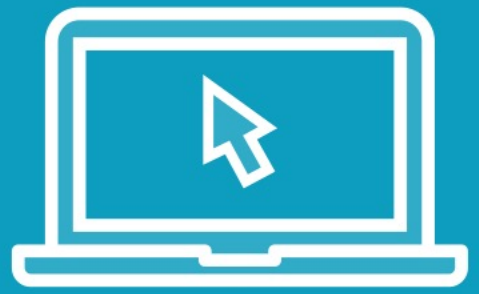
Customizing Queries with LINQ

LINQ method	Purpose
Products.Where(x => x.Category == "Food")	Filters items using inline logic
Products.First(x => x.Id == 1)	Returns the first matching item
Products.OrderBy(x => x.Category)	Orders items based on a property
Products.Take(10)	Retrieves x number of items from the set
Products.GroupBy(x => x.Created)	Groups items by a given property

A Complete Entity Framework Workflow

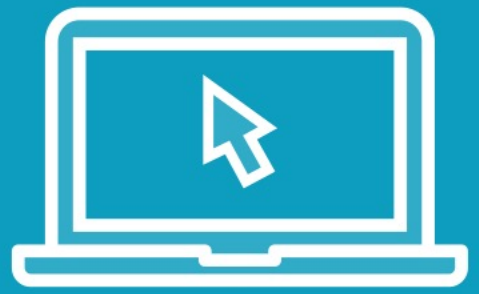


Demo



Setting up the Entity Framework classes

Demo



Applying essential configurations

Understanding Migrations

Managing Data Model Changes

Product class

```
string Name { get;set;}

string Category { get;set;}

string SKU { get; set; }
```

New property

Store location class

```
string Name { get;set;}

string Address { get;set;}

decimal Id { get;set; }
```

New entity

Matching column does not exist

Matching table does not exist

Original database schema

Entity Framework Migrations

Provide a way to create or update the database schema to align with the application data model

Exploring Migration Workflows

Initial setup

```
WiredContext : DbContext
{
  DbSet<Product> Products
}
```

Create migration

Generated migration code

Run migration

Generated database

New entity

```
WiredContext : DbContext
{
  DbSet<Product> Products
  DbSet<Location> Locations
}
```

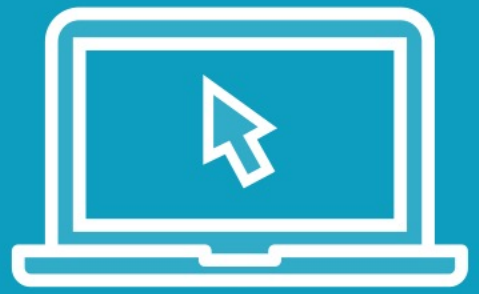
Create migration

Generated migration code

Run migration

Updated database

Demo



Creating the database using migrations

Dependency Injection Basics

Dependency Injection

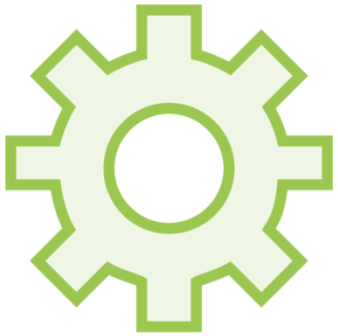
A design pattern that makes classes more independent from their dependencies

Abstractions vs. Implementations

Abstractions:
Interfaces

IEmailService

Implementations:
Classes



OutlookMailService

A Simple Dependency Injection Example

Decoupling Components

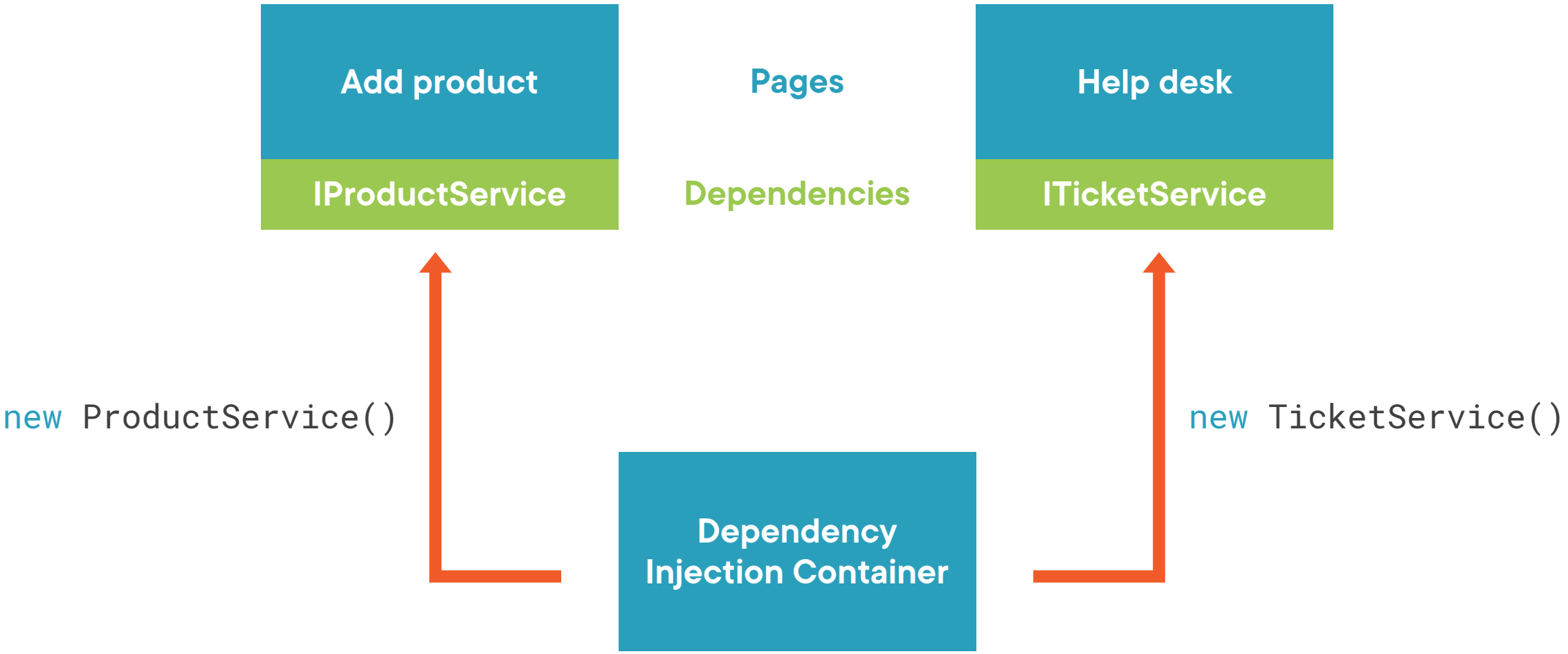
AddProduct.cshtml.cs

```
public void OnPost()
{
    var mailer = new OutlookMailService();
    mailer.SendEmail();
}
```

AddProduct.cshtml.cs

```
private IMailService mailer;

public AddProduct(IMailService mailService)
{
    this.mailer = mailService;
}

public void OnPost() {
    mailer.SendEmail();
}
```

The Dependency Injection Container



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IEmailService, OutlookService>();
    services.AddTransient<IProductService, ProductService>();
}
```

Configuring Dependency Injection Containers

Dependencies are registered in Program.cs using the service collection

We generally bind an interface type to a class implementation type

A Sample Dependency Injection Workflow

AddProduct.cs

```
public AddProduct(IProductRepository productRepo) {
        // Constructor
}
```

Hey, I depend on this product repo contract

Okay, here is an implementation

Program.cs (DI container)

```
services.AddTransient<IProductRepository,ProductRepository>();
```

Dependency Injection Benefits

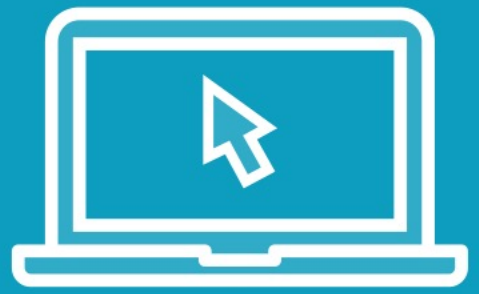
Loose coupling

Improved testability

Service lifecycle management

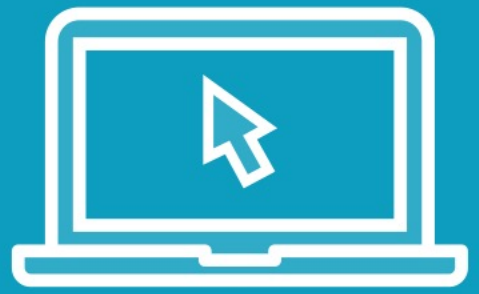
Readability and maintainability

Demo



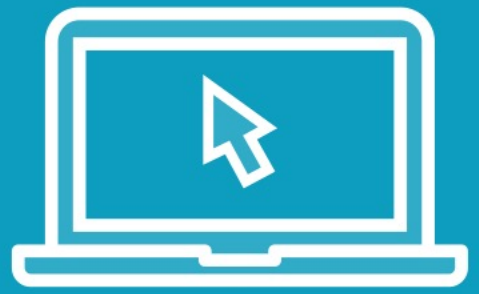
Saving new products to the database

Demo



Preparing the form for image uploads

Demo



Saving the uploaded product images

Summary

Entity Framework (EF) is an object relational mapper

ORMs create a code abstraction over a database to handle SQL queries and low-level operations

EF uses a class called DbContext to represent and manage interactions with a database

DbContext defines DbSet properties with type parameter to represent our database tables

EF Migrations keep our database in sync with the code data model and state of our application

Dependency injection (DI) is a pattern that makes classes more independent from their dependencies

.NET provides a built in DI container to supply dependency instances and manage their life cycles