# Entity Framework Core

# Course Map: Day 1

1. ASP.NET Core Architecture
2. Getting Started with ASP.NET Core
3. Design Patterns, Unit Testing
4. **Entity Framework Core**

# Course Map: Day 2

5. Introduction to TypeScript

6. Using VS Code for TypeScript

7. Angular 2 Architecture

8. Using Angular CLI for Client Apps

# Agenda

- EF Core vs EF 6.x
- Modeling Options, Tooling
- Model-First with Migrations
- Database-First with Scaffolding
- Queries using LINQ
- Disconnected Updates

# Get the Bits

github.com / tonysneed /

Kliant.AspNetCore-Angular

# EF Core versus EF 6.x

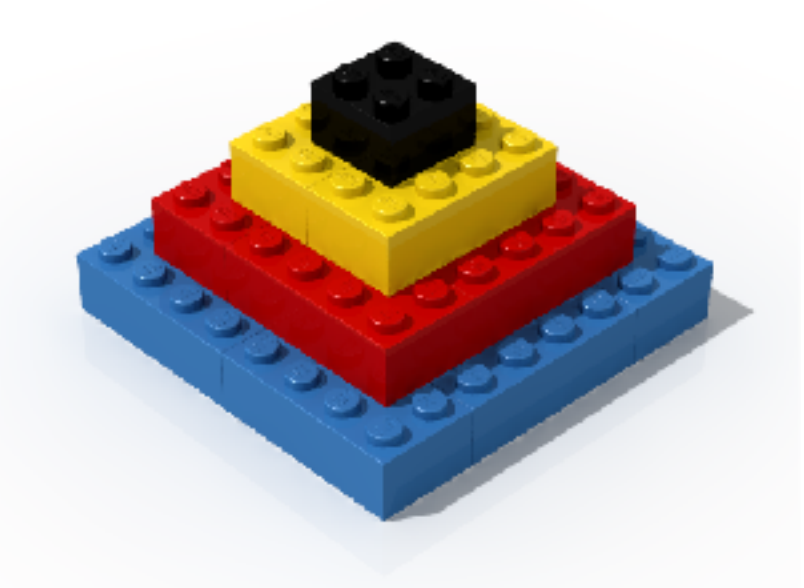

EF Core is a **brand new** data access stack

- Fixes problems with EF 6.x
- Enables non-relational providers
- Lacks some essential ORM features

Use EF 6.x **real-world** use cases

- Still recommended and supported by Microsoft
- But it only works on Windows and full .NET

# Modeling Options



1. **Model-First**
   - Database created based on models
   - Database updated as models change
2. **Database-First**
   - Scaffold models based on existing database
   - Re-generate scaffolding when tables change

# EF Core Tooling

Use EF **command-line** tools with .NET CLI

- Must be **console** or **web** app — not class library


https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dotnet

# Project.json for EF Core Tooling

```
"dependencies": {
  "Microsoft.EntityFrameworkCore.SqlServer":"1.1.0",
  "Microsoft.EntityFrameworkCore.SqlServer.Design": "1.1.0",
  "Microsoft.EntityFrameworkCore.Design": {
    "type": "build", "version": "1.1.0"
  }
},
"tools": {
  "Microsoft.EntityFrameworkCore.Tools.DotNet": "1.1.0-preview4-final"
}
```

# Model First Approach

1. Create **model** classes
   - Conventions used for modeling
   - Can override with Data Annotations or Fluent API
2. Create **DbContext** class
   - DbSet<T> property for each model
   - Override OnConfiguring to use Fluent API

# Model First: Classes

```
public class Category
{
    public int CategoryId { get; set; }            // Primary key
    public string CategoryName { get; set; }
}
```

# Model First: Classes

```
public class Product
{
    public int ProductId { get; set; }              // Primary key
    public string ProductName { get; set; }
    public decimal UnitPrice { get; set; }

    public int CategoryId { get; set; }            // Foreign key
    public Category Category { get; set; }     // Reference
}
```

# Model First: DbContext

```
public class ProductsDbContext: DbContext
{
    // Ctor accepting DbContextOptions
    public ProductsDbContext(DbContextOptions options) : base(options) { }

    // DbSet properties
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

# Connection String: appsettings.json

```
{
  "ConnectionStrings": {
    "ProductsDbConnection": "Data Source=.\\sqlexpress;
        Initial Catalog=ProductsDb;
        Integrated Security=True;
        MultipleActiveResultSets=True"
  }
}
```

# Connection String: Startup

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services
    services.AddMvc();

    // Add EF Context
    services.AddDbContext<ProductsDbContext>(options =>
        options.UseSqlServer(Configuration.
            GetConnectionString("ProductsDbConnection")));
}
```

# Model First: Migrations

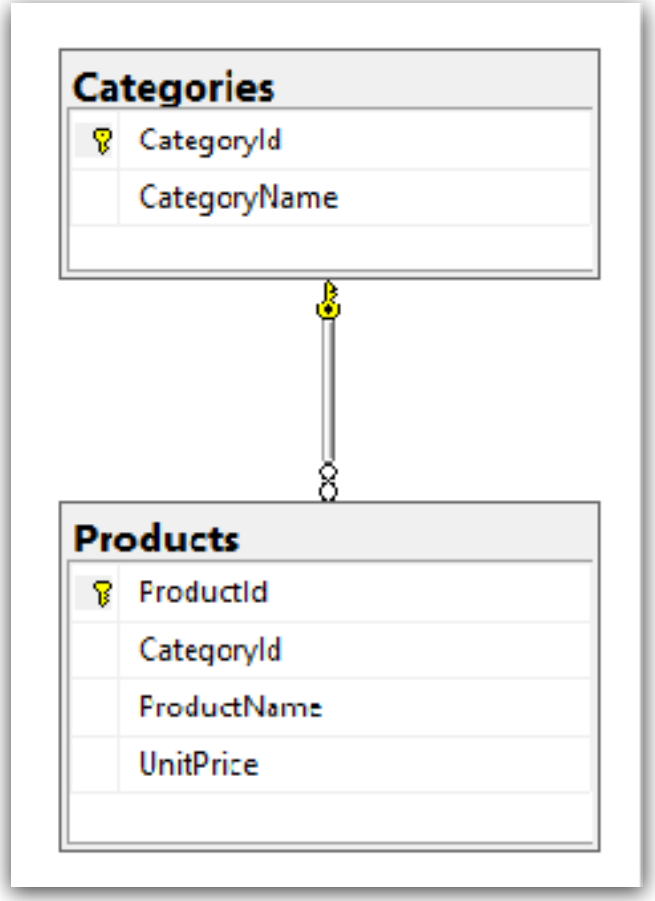
1. Create database (optional)
2. Add migration
   - Creates Migration folder with files
3. Apply migration to database

```
dotnet ef database update

dotnet ef migrations add init

dotnet ef database update
```

# Model First: Migrations




Relations inferred from class properties

# Database First: Scaffolding

1. Use **dbcontext** with **scaffold** command
   - Connection string to **existing** database
   - Specify EF provider and models folder
   - Use -f flag to overwrite previously generated files
   - Can specify individual tables

```
dotnet ef dbcontext scaffold
"Server=(local)\sqlexpress;Database=NorthwindSlim;
Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer -o Models -f
```

# Demo: EF Core Migrations

# Queries using LINQ

```
public class CategoryRepository : ICategoryRepository {
    private readonly ProductsDbContext _context;
    public CategoryRepository(ProductsDbContext context) { _context = context; }

    public async Task<IEnumerable<Category>> GetCategories() {
        return await (from c in _context.Categories
                      orderby c.CategoryName select c).ToListAsync(); }

    public async Task<Category> GetCategory(int id) {
        return await _context.Categories
            .SingleOrDefaultAsync(c => c.CategoryId == id); }
}
```

# Loading Related Entities

```
public async Task<IEnumerable<Product>> GetProducts()
{
    return await _context.Products
        .Include(p => p.Category)
        .OrderBy(p => p.ProductName)
        .ToListAsync();
}
```

# Preparing Entities for Persistence

```
public class ProductRepository : IProductRepository {

    public void Insert(Product product) {          // Mark entity as Added
        _dbContext.Products.Add(product); }

    public void Update(Product product) {          // Mark entity as Modified
        _dbContext.Products.Update(product); }

    public async Task Delete(int id) {             // Mark entity as Deleted
        var product = await _dbContext.Products.FindAsync(id)
        _dbContext.Products.Remove(product); }
}
```

# Loading Related Entities

```
public class ProductRepository : IProductRepository {

    // Load Product.Category
    public async Task LoadCategory(Product product)
    {
        await _context.Entry(product)
            .Reference(p => p.Category)
            .LoadAsync();
    }
}
```

# Saving Entities: Create

```
// POST api/values
[HttpPost]
public async Task<Product> Post(Product value)
{
    _unitOfWork.ProductRepository.CreateProduct(value);
    await _unitOfWork.SaveChangesAsync();
    return value;
}
```

# Saving Entities: Update

```
// PUT api/values/5
[HttpPut]
public async Task<Product> Put(Product value)
{
    _unitOfWork.ProductRepository.UpdateProduct(value);
    await _unitOfWork.SaveChangesAsync();
    return value;
}
```

# Saving Entities: Delete

```
// DELETE api/values/5
[HttpDelete("{id}")]
public async Task Delete(int id)
{
    await _unitOfWork.ProductRepository.DeleteProduct(id);
    await _unitOfWork.SaveChangesAsync();
}
```

# Demo: EF Core Queries and Updates

# Questions?