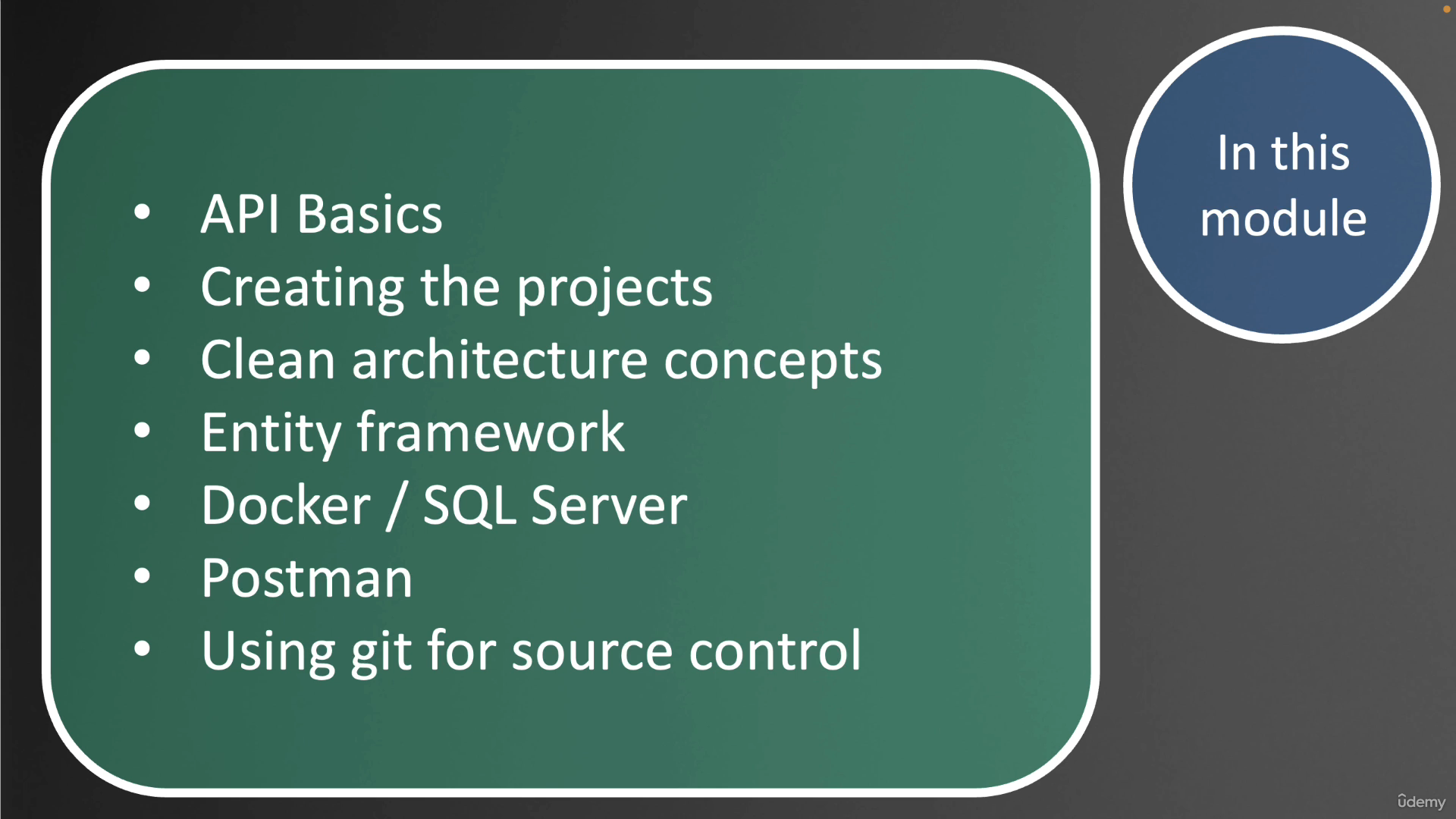
[e-commerce app with .Net Core and Angular](https://kumaran.udemy.com/course/learn-to-build-an-e-commerce-app-with-net-core-and-angular/)

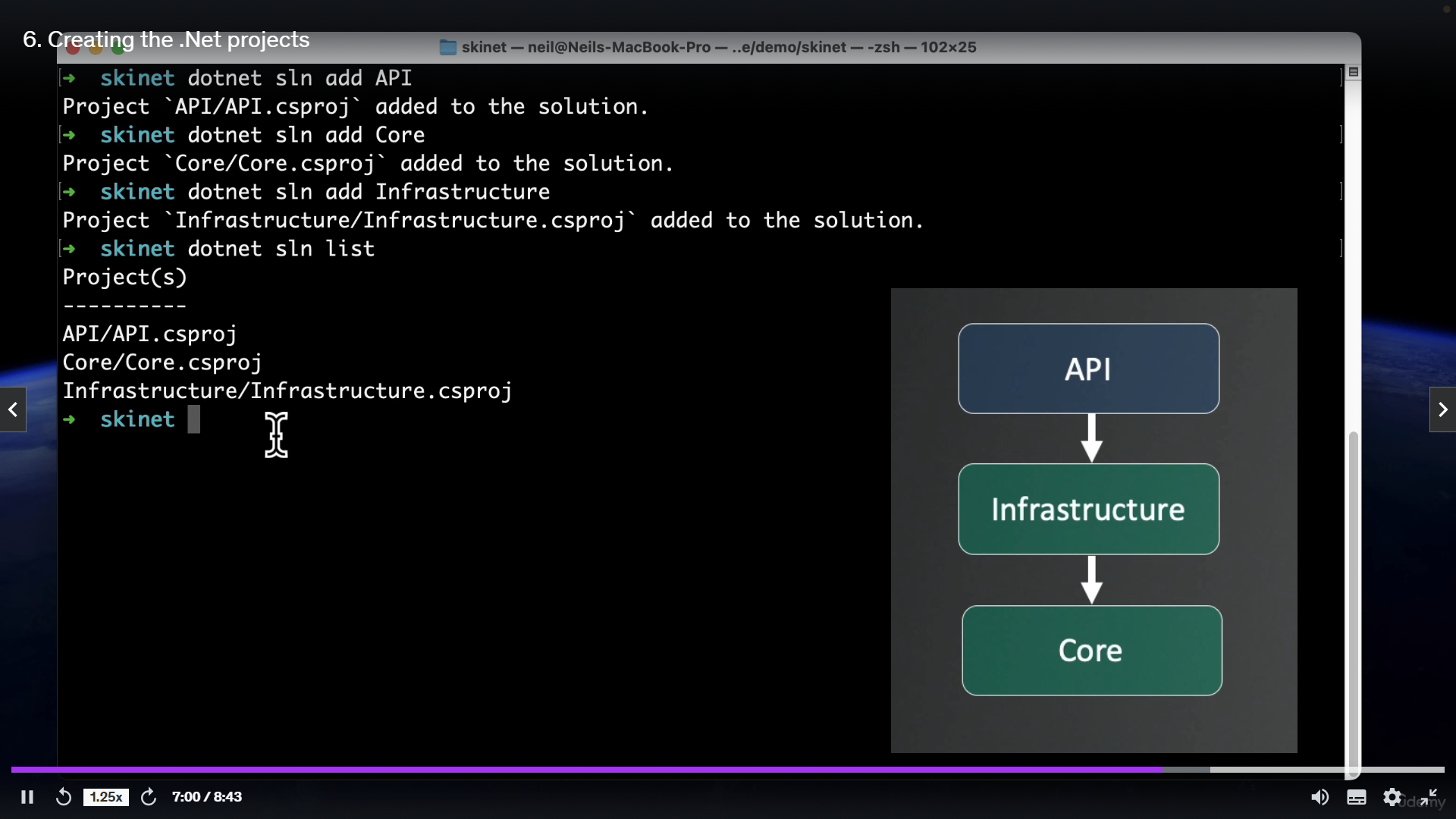
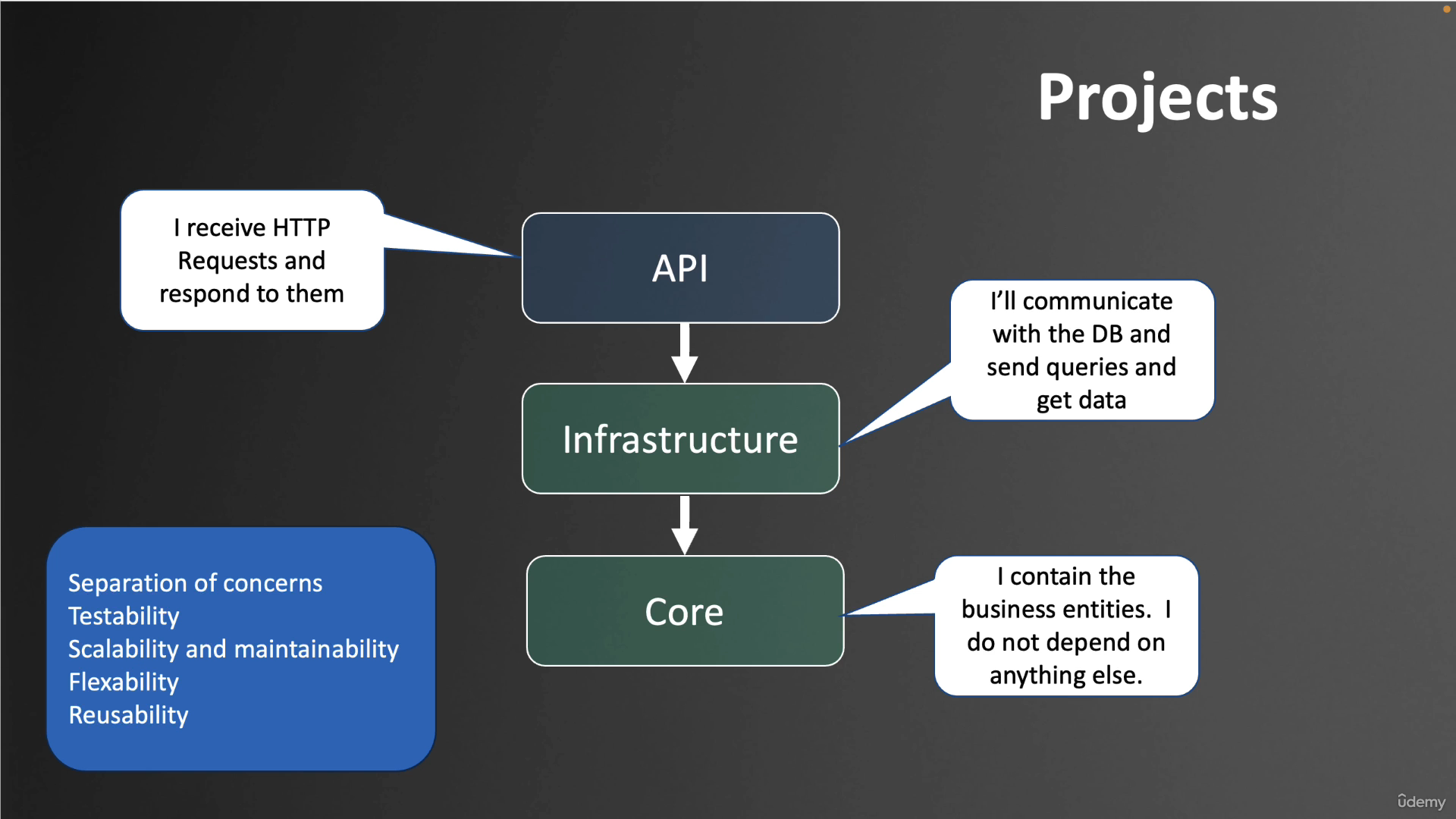
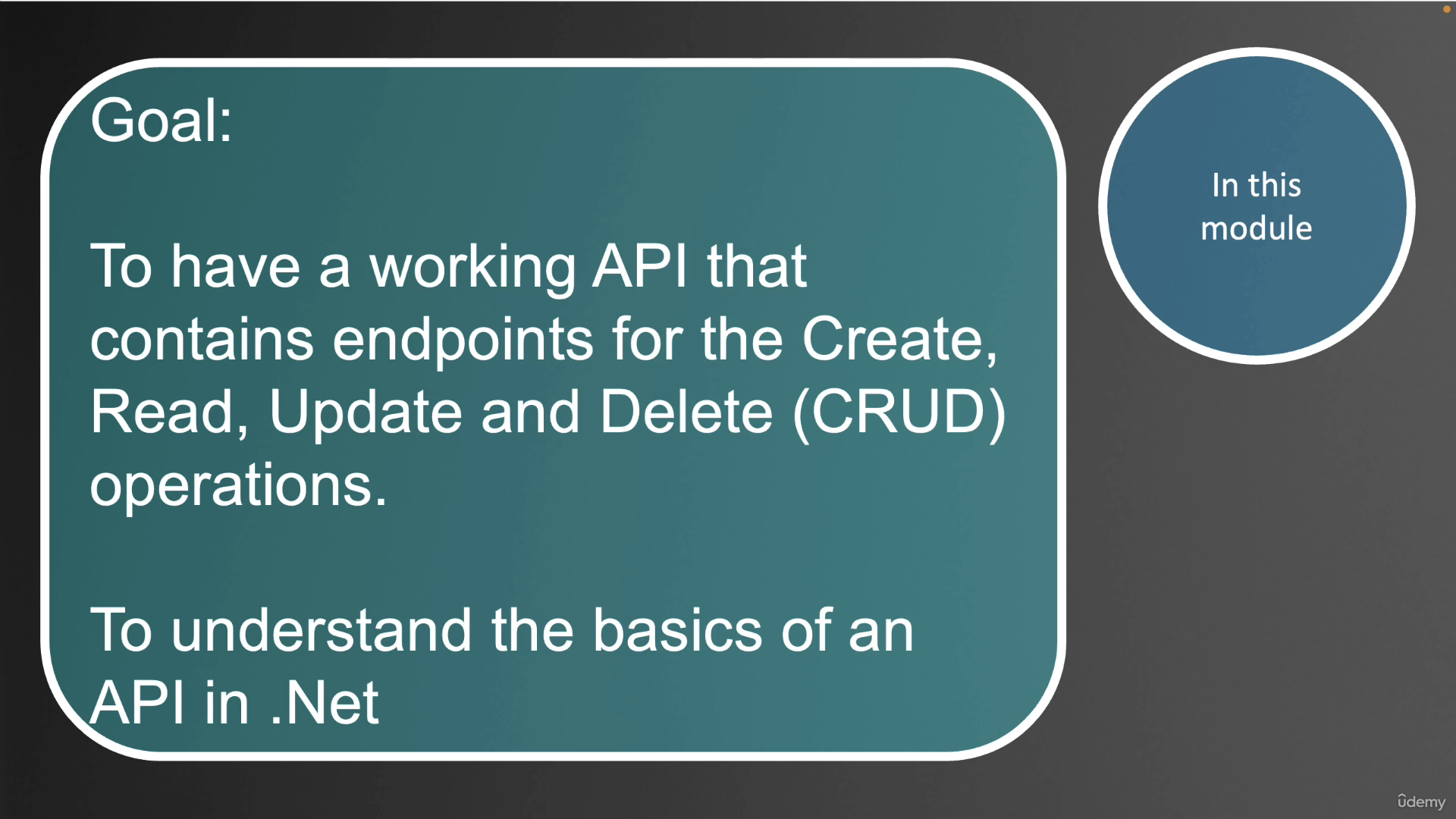






* Vs code / VS
* Docker
* Git
* Postman
* SQLServer





dotnet tool install --global dotnet-ef --version 8.0.8

"ConnectionStrings": {

"DefaultConnection": "Server=lochost;Database=skinet;Trusted\_Connection=True;TrustServerCertificate=True"

}

dotnet ef migrations add InitialCreate -s API -p Infrastructure

namespace Infrastructure.Config

{

public class ProductConfiguration : IEntityTypeConfiguration<Product>

{

public void Configure(EntityTypeBuilder<Product> builder)

{

builder.Property(x => x.Price).HasColumnType("decimal(18,2");

builder.Property(x => x.Name).IsRequired();

}

}

}

public class StoreContext : DbContext

{

public StoreContext(DbContextOptions options): base(options) { }

public DbSet<Product> Products { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.ApplyConfigurationsFromAssembly(typeof(ProductConfiguration).Assembly);

}

}

SEED Data

namespace Infrastructure.Data

{

public class StoreContextSeed

{

public static async Task SeedAsync(StoreContext context)

{

if (!context.Products.Any())

{

var productsData = await File.ReadAllTextAsync("../Infrastructure/Data/SeedData/products.json");

var products = JsonSerializer.Deserialize<List<Product>>(productsData);

if (products == null) return;

context.Products.AddRange(products);

await context.SaveChangesAsync();

}

}

}

}

**Program.cs**

app.MapControllers();

try

{

using var scope = app.Services.CreateScope();

var services = scope.ServiceProvider;

var context = services.GetRequiredService<StoreContext>();

await context.Database.MigrateAsync();

await StoreContextSeed.SeedAsync(context);

}

catch (Exception ex)

{

Console.WriteLine(ex);

throw;

}

app.Run();

namespace Core.Interfaces

{

public interface IGenericRepository<T> where T : BaseEntity

{

Task<T?> GetByIdAsync(int id);

Task<IReadOnlyList<T>> ListAllAsync();

void Add(T entity);

void Update(T entity);

void Remove(T entity);

Task<bool> SaveAllAsync();

bool Exists(int id);

}

}

namespace Infrastructure.Data

{

public class GenericRepository<T>(StoreContext context) : IGenericRepository<T> where T : BaseEntity

{

public void Add(T entity)

{

context.Set<T>().Add(entity);

}

public bool Exists(int id)

{

return context.Set<T>().Any(x=>x.Id == id);

}

public async Task<T?> GetByIdAsync(int id)

{

return await context.Set<T>().FindAsync(id);

}

public async Task<IReadOnlyList<T>> ListAllAsync()

{

return await context.Set<T>().ToListAsync();

}

public void Remove(T entity)

{

context.Set<T>().Remove(entity);

}

public async Task<bool> SaveAllAsync()

{

return await context.SaveChangesAsync() > 0;

}

public void Update(T entity)

{

context.Set<T>().Attach(entity);

context.Entry(entity).State = EntityState.Modified;

}

}

}

namespace Core.Interfaces

{

public interface ISpecification<T>

{

Expression<Func<T, bool>> Criteria { get;}

}

}

namespace Core.Specification

{

public class BaseSpecification<T>(Expression<Func<T,bool>> \_criteria) : ISpecification<T>

{

public Expression<Func<T, bool>> Criteria => \_criteria;

}

}

namespace Infrastructure.Data

{

public class SpecificationEvaluator<T> where T : BaseEntity

{

public static IQueryable<T> GetQuery(IQueryable<T> query, ISpecification<T> spec)

{

if (spec.Criteria != null)

{

query = query.Where(spec.Criteria); // x => x.Brand == brand

}

return query;

}

}

}

namespace Core.Interfaces

{

public interface IGenericRepository<T> where T : BaseEntity

{

Task<T?> GetByIdAsync(int id);

Task<IReadOnlyList<T>> ListAllAsync();

**Task<T?> GetEntityWithSpec(ISpecification<T> spec);**

**Task<IReadOnlyList<T>> ListAsync(ISpecification<T> spec);**

void Add(T entity);

void Update(T entity);

void Remove(T entity);

Task<bool> SaveAllAsync();

bool Exists(int id);

}

}

public class GenericRepository<T>(StoreContext context) : IGenericRepository<T> where T : BaseEntity

{

public async Task<T?> GetEntityWithSpec(ISpecification<T> spec)

{

return await ApplySpecification(spec).FirstOrDefaultAsync();

}

public async Task<IReadOnlyList<T>> ListAsync(ISpecification<T> spec)

{

return await ApplySpecification(spec).ToListAsync();

}

private IQueryable<T> ApplySpecification (ISpecification<T> spec)

{

return SpecificationEvaluator<T>.GetQuery(context.Set<T>().AsQueryable(), spec);

}

}

namespace Core.Specification

{

public class BaseSpecification<T>(Expression<Func<T,bool>> \_criteria) : ISpecification<T>

{

protected BaseSpecification() : this(null) { }

public Expression<Func<T, bool>>? Criteria => \_criteria;

}

}

namespace Core.Specification

{

public class ProductSpecification : BaseSpecification<Product>

{

public ProductSpecification(string? brand,string? type) : base(x =>

(string.IsNullOrWhiteSpace(brand) || x.Brand == brand) &&

(string.IsNullOrWhiteSpace(type) || x.Type == type)

)

{

}

}

}

[HttpGet]

public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts(string? brand,

string? type,string? sort)

{

var spec = new ProductSpecification(brand, type);

var products = await repo.ListAsync(spec);

return Ok(products);

}

namespace Core.Interfaces

{

public interface ISpecification<T>

{

Expression<Func<T, bool>> Criteria { get;}

**Expression<Func<T,object>>? OrderBy { get;}**

**Expression<Func<T,object>>? OrderByDescending { get;}**

}

}

namespace Core.Specification

{

public class BaseSpecification<T>(Expression<Func<T, bool>> \_criteria) : ISpecification<T>

{

protected BaseSpecification() : this(null) { }

public Expression<Func<T, bool>>? Criteria => \_criteria;

**public Expression<Func<T, object>>? OrderBy { get; private set; }**

**public Expression<Func<T, object>>? OrderByDescending { get; private set; }**

**protected void AddOrderBy(Expression<Func<T, object>> orderByExpression)**

**{**

**OrderBy = orderByExpression;**

**}**

**protected void AddOrderByDescending(Expression<Func<T, object>> orderByDescendingExpression)**

**{**

**OrderBy = orderByDescendingExpression;**

**}**

}

}

namespace Infrastructure.Data

{

public class SpecificationEvaluator<T> where T : BaseEntity

{

public static IQueryable<T> GetQuery(IQueryable<T> query, ISpecification<T> spec)

{

if (spec.Criteria != null)

{

query = query.Where(spec.Criteria); // x => x.Brand == brand

}

if (spec.OrderBy != null) {

query = query.OrderBy(spec.OrderBy);

}

if(spec.OrderByDescending != null)

{

query = query.OrderByDescending(spec.OrderByDescending);

}

return query;

}

}

}

namespace Core.Specification

{

public class ProductSpecification : BaseSpecification<Product>

{

public ProductSpecification(string? brand,string? type,string? sort) : base(x =>

(string.IsNullOrWhiteSpace(brand) || x.Brand == brand) &&

(string.IsNullOrWhiteSpace(type) || x.Type == type)

)

{

switch (sort)

{

case "priceAsc":

AddOrderBy(x=>x.Price);

break;

case "priceDesc":

AddOrderByDescending(x=>x.Price);

break;

default:

AddOrderBy(x=>x.Name);

break;

}

}

}

}

[HttpGet]

public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts(string? brand,

string? type,string? sort)

{

var spec = new ProductSpecification(brand, type,**sort**);

var products = await repo.ListAsync(spec);

return Ok(products);

}

namespace Core.Interfaces

public interface ISpecification<T,TResult> : ISpecification<T>

{

Expression<Func<T, TResult>>? Select { get; }

}

namespace Core.Specification

public class BaseSpecification<T, TResult>(Expression<Func<T, bool>> criteria)

: BaseSpecification<T>(criteria), ISpecification<T, TResult>

{

public Expression<Func<T, TResult>>? Select { get; private set; }

protected void AddSelect(Expression<Func<T, TResult>> selectExpression)

{

Select = selectExpression;

}

}

namespace Infrastructure.Data

public static IQueryable<TResult> GetQuery<TSpec,TResult>(IQueryable<T> query, ISpecification<T,TResult> spec)

{

if (spec.Criteria != null)

{

query = query.Where(spec.Criteria); // x => x.Brand == brand

}

if (spec.OrderBy != null)

{

query = query.OrderBy(spec.OrderBy);

}

if (spec.OrderByDescending != null)

{

query = query.OrderByDescending(spec.OrderByDescending);

}

var selectQuery = query as IQueryable<TResult>;

if (spec.Select != null) {

selectQuery = query.Select(spec.Select);

}

return selectQuery ?? query.Cast<TResult>();

}

namespace Core.Interfaces

Task<TResult?> GetEntityWithSpec<TResult>(ISpecification<T,TResult> spec);

Task<IReadOnlyList<TResult>> ListAsync<TResult>(ISpecification<T,TResult> spec);

namespace Infrastructure.Data

public async Task<TResult?> GetEntityWithSpec<TResult>(ISpecification<T, TResult> spec)

{

return await ApplySpecification(spec).FirstOrDefaultAsync();

}

public async Task<IReadOnlyList<TResult>> ListAsync<TResult>(ISpecification<T, TResult> spec)

{

return await ApplySpecification(spec).ToListAsync();

}

private IQueryable<TResult> ApplySpecification<TResult>(ISpecification<T,TResult> spec)

{

return SpecificationEvaluator<T>.GetQuery<T,TResult>(context.Set<T>().AsQueryable(), spec);

}

namespace Core.Interfaces

bool IsDistinct { get; }

namespace Core.Specification

public bool IsDistinct { get; private set; }

protected void ApplyDistinct()

{

IsDistinct = true;

}

namespace Infrastructure.Data

public class SpecificationEvaluator<T> where T : BaseEntity

if (spec.IsDistinct)

{

query = query.Distinct();

}

if (spec.IsDistinct)

{

selectQuery = selectQuery?.Distinct();

}

namespace Core.Specification

namespace Core.Specification

{

public class BrandListSpecification : BaseSpecification<Product,string>

{

public BrandListSpecification()

{

AddSelect(x => x.Brand);

ApplyDistinct();

}

}

}

BaseSpecification

public class BaseSpecification<T, TResult>(Expression<Func<T, bool>> criteria)

: BaseSpecification<T>(criteria), ISpecification<T, TResult>

{

**protected BaseSpecification() : this(null!) { }**

public Expression<Func<T, TResult>>? Select { get; private set; }

protected void AddSelect(Expression<Func<T, TResult>> selectExpression)

{

Select = selectExpression;

}

}

TypeListSpecification

namespace Core.Specification

{

public class TypeListSpecification : BaseSpecification<Product, string>

{

public TypeListSpecification()

{

AddSelect(x=>x.Type);

ApplyDistinct();

}

}

}

**Section 5 – Sorting,Filtering,Searching and Pagination**

Paginaion - Performance

* Parameters passed by query string :

/api/products?pageNumber=2&pageSize=5

* Page size should be limited
* We should always page results

Deferred Execution

* Query commands are stored in a variable
* Execution of the query is deferred
* IQueryable<T> creates an expression tree
* Execution :
  + ToList(), ToArray(), ToDictionary()
  + Count() or other singleton queries

namespace Core.Specification

{

public class ProductSpecParams

{

private List<string> \_brands = [];

public List<string> Brands

{

get => \_brands;

set

{

\_brands = value.SelectMany(x => x.Split(',',StringSplitOptions.RemoveEmptyEntries)).ToList();

}

}

private List<string> \_types = [];

public List<string> Types

{

get => \_types;

set

{

\_types = value.SelectMany(x => x.Split(',', StringSplitOptions.RemoveEmptyEntries)).ToList();

}

}

public string? sort { get; set; }

}

}

namespace Core.Specification

{

public class ProductSpecification : BaseSpecification<Product>

{

public ProductSpecification(ProductSpecParams specParams) : base(x =>

(specParams.Brands.Count == 0 || specParams.Brands.Contains(x.Brand)) &&

(specParams.Types.Count == 0 || specParams.Types.Contains(x.Type))

)

{

switch (specParams.sort)

{

case "priceAsc":

AddOrderBy(x=>x.Price);

break;

case "priceDesc":

AddOrderByDescending(x=>x.Price);

break;

default:

AddOrderBy(x=>x.Name);

break;

}

}

}

}

namespace API.Controllers

[HttpGet]

public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts([FromQuery]ProductSpecParams specParams)

{

var spec = new ProductSpecification(specParams);

var products = await repo.ListAsync(spec);

return Ok(products);

}

Adding Pagination

namespace Core.Interfaces

{

public interface ISpecification<T>

{

int Take { get; }

int Skip { get; }

bool IsPagingEnabled { get; }

}

}

namespace Core.Specification

{

public class BaseSpecification<T>(Expression<Func<T, bool>> \_criteria) : ISpecification<T>

{

public int Take { get; private set; }

public int Skip { get; private set; }

public bool IsPagingEnabled { get; private set; }

protected void ApplyPaging(int skip,int take)

{

Skip = skip;

Take = take;

IsPagingEnabled = true;

}

}

}

namespace Infrastructure.Data

SpecificationEvaluator

if (spec.IsPagingEnabled) {

query = query.Skip(spec.Skip).Take(spec.Take);

}

if (spec.IsPagingEnabled)

{

selectQuery = selectQuery?.Skip(spec.Skip).Take(spec.Take);

}

namespace Core.Specification

{

public class ProductSpecParams

{

private const int MaxPageSize = 50;

public int PageIndex { get; set; } = 1;

private int \_pageSize = 6;

public int PageSize

{

get => \_pageSize;

set => \_pageSize = (value > MaxPageSize) ? MaxPageSize : value;

}

}

}

namespace Core.Specification

{

public class ProductSpecification : BaseSpecification<Product>

{

public ProductSpecification(ProductSpecParams specParams) : base(x =>

(specParams.Brands.Count == 0 || specParams.Brands.Contains(x.Brand)) &&

(specParams.Types.Count == 0 || specParams.Types.Contains(x.Type))

)

{

ApplyPaging(specParams.PageSize \* (specParams.PageIndex - 1),specParams.PageSize);

}

}

}

namespace API.RequestHelpers

{

public class Pagination <T>(int pageIndex, int pageSize, int count, IReadOnlyList<T> data)

{

public int PageIndex { get; set; } = pageIndex;

public int PageSize { get; set; } = pageSize;

public int Count { get; set; } = count;

public IReadOnlyList<T> Data { get; set; } = data;

}

}

namespace Core.Interfaces

{

public interface IGenericRepository<T> where T : BaseEntity

{

Task<int> CountAsync(ISpecification<T> spec);

}

}

namespace Core.Interfaces

{

public interface ISpecification<T>

{

IQueryable<T> ApplyCriteria(IQueryable<T> query);

}

}

namespace Core.Specification

{

public class BaseSpecification<T>(Expression<Func<T, bool>> \_criteria) : ISpecification<T>

{

public IQueryable<T> ApplyCriteria(IQueryable<T> query)

{

if(Criteria != null)

{

query = query.Where(Criteria);

}

return query;

}

}

}

namespace Infrastructure.Data

{

public class GenericRepository<T>(StoreContext context) : IGenericRepository<T> where T : BaseEntity

{

public async Task<int> CountAsync(ISpecification<T> spec)

{

var query = context.Set<T>().AsQueryable();

query = spec.ApplyCriteria(query);

return await query.CountAsync();

}

**}**

**}**

[HttpGet]

public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts([FromQuery]ProductSpecParams specParams)

{

var spec = new ProductSpecification(specParams);

var products = await repo.ListAsync(spec);

var count = await repo.CountAsync(spec);

var pagination = new Pagination<Product>(specParams.PageIndex,specParams.PageSize,count,products);

return Ok(pagination);

}

**Create API Base Controller**

namespace API.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class BaseApiController : ControllerBase

{

protected async Task<ActionResult> CreatePagedResult<T>(IGenericRepository<T> repo,

ISpecification<T> spec, int pageIndex,int pageSize) where T : BaseEntity

{

var items = await repo.ListAsync(spec);

var count = await repo.CountAsync(spec);

var pagination = new Pagination<T>(pageIndex, pageSize, count, items);

return Ok(pagination);

}

}

}

namespace API.Controllers

[HttpGet]

public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts([FromQuery]ProductSpecParams specParams)

{

var spec = new ProductSpecification(specParams);

return await CreatePagedResult(repo,spec,specParams.PageIndex,specParams.PageSize);

}

Adding Search Functionality

namespace Core.Specification

{

public class ProductSpecParams

{

private string? \_search;

public string Search

{

get => \_search ?? "";

set => \_search = value.ToLower();

}

}

}