Design Patterns and Unit Testing

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

- Data Access and Coupling
- Repository Pattern
- Unit of Work Pattern
- Implementing IDisposable
- Unit Testing with xUnit
- Mocking with MOQ

Get the Bits

github.com / tonysneed /

Kliant.AspNetCore-Angular

Design Patterns

Data Access and Coupling

Avoid **coupling** to data access API

- Changes render app obsolete
- Difficult to test

Instead program to **abstractions**

- Implementations supplied at runtime
- Leverage dependency injection

Repository Pattern

Repository **interfaces** decouple app from data access API

- Async method signature
- Return Task<T>

```
public interface IProductRepository {
    Task<Product> FindAsync(int id);
}
```

Constructor Injection

Classes **declare** their dependencies

- Pass interfaces to constructor
- DI container supplies “real” instance

```
public class ProductsController : Controller {

  private readonly IProductRepository _productRepository;
  public ProductsController(IProductRepository productRepository) {
        _productRepository = productRepository;
    }
}
```

Repository Implementation

Implementation uses specific **data access API**

```
public class ProductRepository : IProductRepository {

   private readonly NorthwindContext _dbContext;
   public ProductRepository(NorthwindContext dbContext) {
        _dbContext = dbContext; }

    public async Task<Product> FindAsync(int id) {
        return await _dbContext.Products.FindAsync(id);
    }
}
```

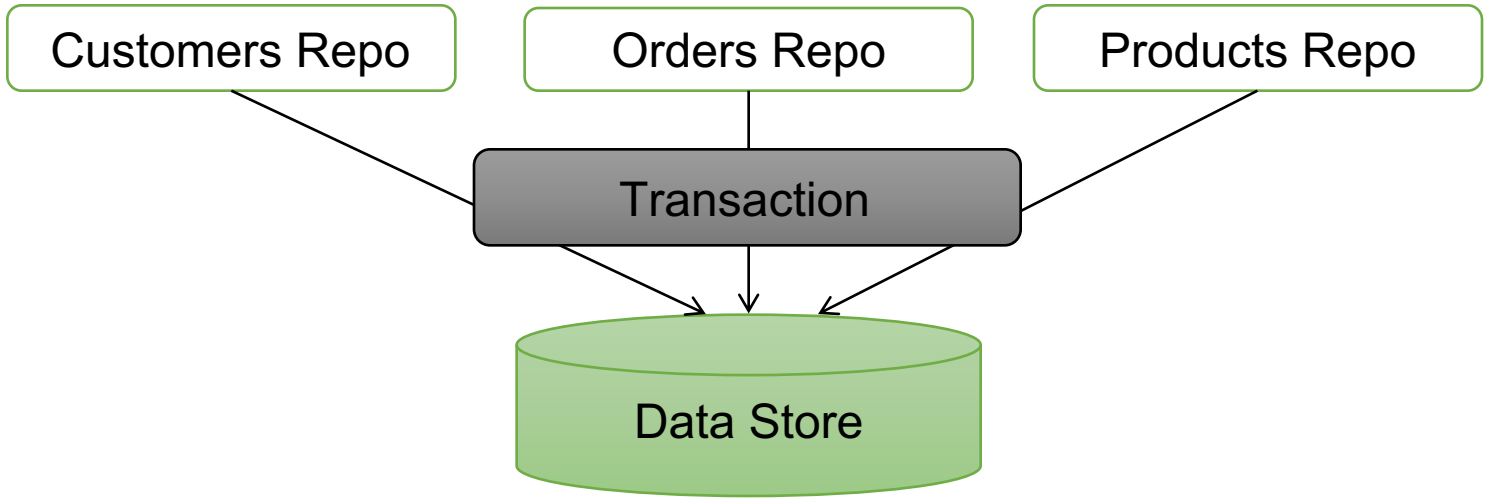
Preparing Entities for Persistence

```
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);
    } }
```

Problem: Transactions

Entities from **multiple** repositories

- Saved within same transaction



Solution: Unit of Work Pattern

Unit of work spans **multiple** repositories

- Expose each repository as a property on UoW interface

```
public interface IUnitOfWork
{
    ICustomerRepository CustomerRepository { get; }
    IOrderRepository OrderRepository { get; }
    IProductRepository ProductRepository { get; }

   Task<int> SaveChangesAsync();    // Persist all changes
}
```

Unit of Work Implementation

```
public class UnitOfWork : IUnitOfWork, IDisposable
{
    public UnitOfWork(ICustomerRepository custRepo,
        IOrderRepository orderRepo,
        NorthwindContext dbContext) { // Code elided ...

   public ICustomerRepository CustomerRepository {  // Repositories
       get { return _customerRepository; } }
    public IOrderRepository OrderRepository {
        get { return _orderRepository; } }

   public async Task<int> SaveChangesAsync() {
        return await _dbContext.SaveChangesAsync(); }
}
```

Clean-Up: IDisposable

```
public class UnitOfWork : IDisposable {

    public void Dispose()
    {
        // Safely cast to IDisposable, then call Dispose
        if (_disposed) return;
        var disposable = _dbContext as IDisposable;
        if (disposable != null) disposable.Dispose();
    }
}
```

Controllers and UoW

```
public class ProductController : Controller {
   private readonly IUnitOfWork _unitOfWork;
    public ProductController(IUnitOfWork unitOfWork) {
        _unitOfWork = unitOfWork; }

    [HttpGet("{id}")]
    public async Task<IActionResult> Get(int id) {
       return await _unitOfWork.ProductRepository.FindAsync(id); }

   [HttpPost]
    public async Task<IActionResult> Post(Product product) {
        _unitOfWork.ProductRepository.Insert(product);
        await _unitOfWork.SaveChangesAsync();
        return product; } }
```

Demo: Design Patterns

Unit Testing

Problem: Manual Testing

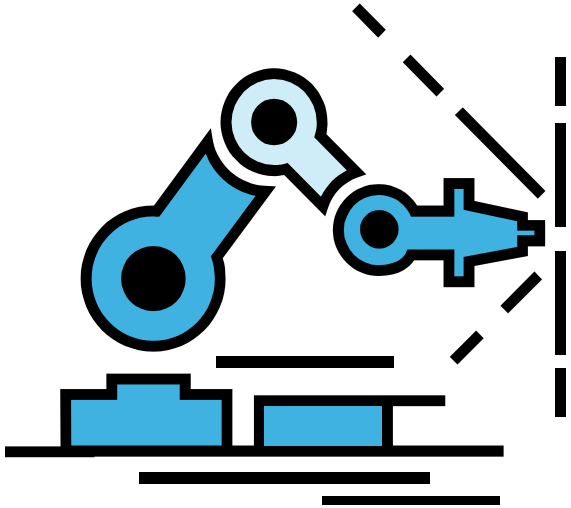
Problems with **manual** testing

- Time-consuming
- Difficult to reproduce
- Non-regressive
- Error-prone

Solution: Automated Testing

Benefits of **automated** testing

- Document expected behaviors
- Verify fix doesn't break something else
- Can run tests with CI builds



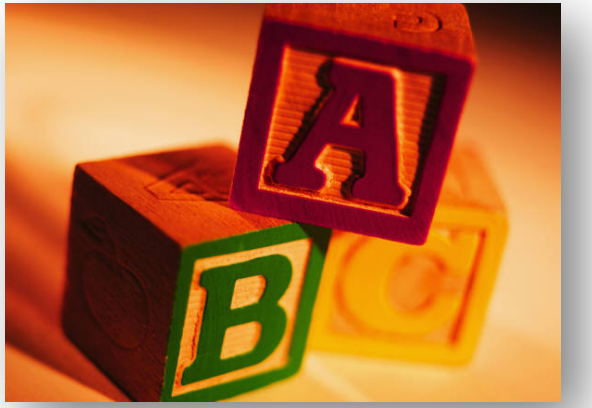
Unit Tests vs Integration Tests

Unit Tests

- Serve as specifications
- Demonstrate just one behavior
- Decoupled from external dependencies
- Can be run in parallel

Integration Tests

- End-to-end from client to server
- Can include external dependencies
- Includes all components in the stack



Preferred Framework: xUnit

Scaffold with DotNet CLI or Yeoman

C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Source>mkdir HelloXUnit

C:\Source>cd HelloXUnit

C:\Source\HelloXUnit>dotnet new -t xunittest
Created new C# project in C:\Source\HelloXUnit.
```

Running Unit Tests

Run unit tests from the command line

Select C:\Windows\System32\cmd.exe

```
C:\Source\HelloXUnit>dotnet test
Project HelloXUnit (.NETCoreApp,Version=v1.1) was previously compiled.
Skipping compilation.
xUnit.net .NET CLI test runner (64-bit win10-x64)
  Discovering: HelloXUnit
  Discovered:  HelloXUnit
  Starting:    HelloXUnit
  Finished:    HelloXUnit
=== TEST EXECUTION SUMMARY ===
   HelloXUnit  Total: 1, Errors: 0, Failed: 0, Skipped: 0, Time: 0.130s

SUMMARY: Total: 1 targets, Passed: 1, Failed: 0.
```

Mocking Frameworks

Only implement members required for testing

Preferred Mocking Framework: Moq

```
public void GetProductShouldReturnProduct() {

    var expectedProduct = new Product{ ProductId = 1, ProductName = "Test" };
    var productsMockRepo = new Mock<IProductsRepository>();
    productsMockRepo
        .Setup(x => x.Find(It.IsAny<int>())).Returns(expectedProduct);

    var controller = new ProductsController(productsMockRepo.Object);
    var product = controller.Get(1);
    Assert.Equal(expectedProduct.ProductName, product.ProductName);
}
```

Demo: Unit Testing

Questions?