Unit Testinng - Connected Services - Docker

with Visual Studio 2022

Mühlehner & Tavolato GmbH

UNIT TESTING IN C#

Mühlehner & Tavolato GmbH

Why Unit Testing?

Benefits of Unit Testing

- Catches bugs early in development
- Acts as documentation
- Makes code changes safer
- Encourages better code design
- Enables continuous integration
- Speeds up development in the long run

Testing Pyramid

Unit tests should form the foundation of your testing strategy

Testing Frameworks in .NET

Popular Frameworks

- **MSTest**: Microsoft's built-in framework
- **NUnit**: Widely-used open-source framework
- **xUnit**: Modern, flexible framework used by .NET Core

We'll focus on **xUnit** for this workshop.

xUnit Features

- **[Fact]**: For simple test cases
- **[Theory]**: For parameterized tests
- Constructor for setup, IDisposable for teardown
- Built-in assertions
- Extensible architecture
- Parallel test execution

Your First Unit Test

Calculator Class

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public int Subtract(int a, int b)
    {
        return a - b;
    }

    public double Divide(int a, int b)
    {
        if (b == 0)
            throw new DivideByZeroException();
        return (double)a / b;
    }
}
```

Basic xUnit Test

```
public class CalculatorTests
{
    private readonly Calculator _calculator = new();

    [Fact]
    public void Add_TwoNumbers_ReturnsSum()
    {
        // Arrange
        int a = 5, b = 3;

        // Act
        int result = _calculator.Add(a, b);

        // Assert
        Assert.Equal(8, result);
    }
}
```

Parameterized Tests with [Theory]

```
public class CalculatorTests
{
    private readonly Calculator _calculator = new();

    [Theory]
    [InlineData(5, 3, 2)]
    [InlineData(10, 5, 5)]
    [InlineData(0, 0, 0)]
    [InlineData(-5, -3, -2)]
    public void Subtract_TwoNumbers_ReturnsDifference(
        int a, int b, int expected)
    {
        // Act
        int result = _calculator.Subtract(a, b);

        // Assert
        Assert.Equal(expected, result);
    }
}
```

Benefit: Test multiple scenarios with a single test method

Testing Exceptions

```
public class CalculatorTests
{
    private readonly Calculator _calculator = new();

    [Fact]
    public void Divide_DivideByZero_ThrowsDivideByZeroException()
    {
        // Arrange
        int a = 10, b = 0;

        // Act & Assert
        Assert.Throws<DivideByZeroException>(
            () => _calculator.Divide(a, b)
        );
    }

    [Theory]
    [InlineData(10, 2, 5.0)]
    [InlineData(7, 2, 3.5)]
    public void Divide_ValidDivision_ReturnsQuotient(
        int a, int b, double expected)
    {
        // Act
        double result = _calculator.Divide(a, b);

        // Assert
        Assert.Equal(expected, result);
    }
}
```

Test Isolation with Mocks

Code Under Test

```
public class UserService
{
    private readonly IUserRepository _repository;

    public UserService(IUserRepository repository)
    {
        _repository = repository;
    }

    public bool UpdateUserEmail(int userId, string newEmai
    {
      // implementation goes here
    }
}
```

Testing with Moq

```
[Fact]
public void UpdateUserEmail_ValidEmail_ReturnsTrue()
{
    // Arrange
    var user = new User { Id = 1 };

    var mockRepo = new Mock<IUserRepository>();
    mockRepo.Setup(r => r.GetById(1)).Returns(user);
    mockRepo.Setup(r => r.Save(It.IsAny<User>()))
            .Returns(true);

    var service = new UserService(mockRepo.Object);

    // Act
    bool result = service.UpdateUserEmail(1, "new@example.com");

    // Assert
    Assert.True(result);
    mockRepo.Verify(r => r.Save(It.Is<User>(
        u => u.Email == "new@example.com")), Times.Once);
}
```

Testing Best Practices

Naming & Structure

- Use descriptive test names:

  `Method_Scenario_ExpectedResult`
- Follow the Arrange-Act-Assert pattern
- One assertion per test (conceptually)
- Test behaviors, not implementation details
- Avoid test interdependence

Coverage & Quality

- Test happy path and error cases
- Test edge cases and boundaries
- Don’t target 100% coverage blindly
- Keep tests fast and deterministic
- Treat test code as production code
- Use test data builders for complex objects

LIVE UNIT TESTING IN VISUAL STUDIO 2022

Mühlehner & Tavolato GmbH

Introduction to Live Unit Testing

What is Live Unit Testing?

- Automatically runs tests as you type
- Shows real-time test results in the editor
- Visual indicators of code coverage
- Available in Visual Studio Enterprise

Benefits

- Instant feedback on code changes
- Encourages Test-Driven Development
- Identifies affected tests immediately
- Increases confidence when refactoring
- Visualizes code coverage directly in editor

Live Unit Testing in Action

Visual Indicators

- **Green ✓**: Code covered by passing tests
- **Red ×**: Code covered by failing tests
- **Blue -**: Code not covered by tests

Demo: Live Unit Testing

1. Go to Test > Live Unit Testing > Start
2. Make a change to the Calculator.Add method:

```
public int Add(int a, int b)
{
    // This will break tests
    return a + b + 1;
}
```

3. Watch the indicators change to red 4. Fix the code and see them turn green again

Configuring Live Unit Testing

Settings (Tools > Options > Live Unit Testing)

- Test run frequency
- Test project inclusion/exclusion
- Run tests after build only
- Custom test runner arguments

Performance Tips

- Include only relevant test projects
- Use tight feedback loops during TDD
- Temporarily pause when writing many new tests
- Filter to specific tests when focusing on a feature

Test-Driven Development with Live Unit Testing

TDD Workflow

1. **Red**: Write a failing test first
2. **Green**: Write just enough code to pass
3. **Refactor**: Improve code while tests pass

Live Unit Testing enhances this cycle with immediate feedback.

Example: Adding a Prime Number Check

```
// Step 1: Write the test first
[Theory]
[InlineData(2, true)]
[InlineData(3, true)]
[InlineData(4, false)]
[InlineData(17, true)]
public void IsPrime_ChecksIfNumberIsPrime_ReturnsCorrectResult(
    int number, bool expected)
{
    bool result = _calculator.IsPrime(number);
    Assert.Equal(expected, result);
}

// Step 2: Implement the method to pass
public bool IsPrime(int number)
{
  // your code
}
```

Mühlehner & Tavolato GmbH

Exercise: TDD with Live Unit Testing

Task

Implement a `Factorial` method using TDD:

1. Write tests for factorial calculation
2. Use Live Unit Testing to guide implementation
3. Ensure your solution handles edge cases

Test Cases to Consider

- Factorial of 0 (should return 1)
- Factorial of positive numbers
- Handling negative numbers (throw ArgumentException)
- Potential overflow issues with large numbers

DOCKER SUPPORT FOR .NET APPLICATIONS

Mühlehner & Tavolato GmbH

Introduction to Docker

What is Docker?

- Platform for developing, shipping, and running applications in containers
- Containers package code and dependencies together
- Ensures consistent environments across development, testing, and production

Benefits for .NET Applications

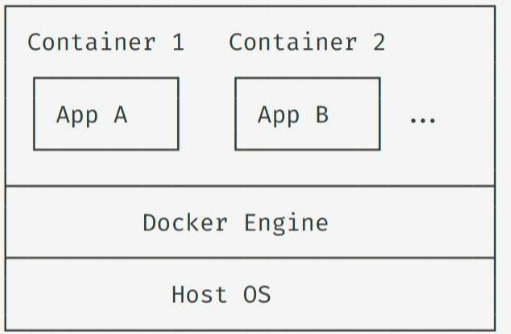
- Consistent deployment across environments
- Simplified dependency management
- Isolation from other applications
- Efficient resource usage
- Improved development workflows
- Platform independence

Docker Concepts

Key Terms

- **Image**: Blueprint for a container
- **Container**: Running instance of an image
- **Dockerfile**: Recipe to build an image
- **Registry**: Storage for Docker images
- **Docker Compose**: Tool for multi-container apps

Docker Architecture



Adding Docker Support in Visual Studio

Steps

1. Right-click on project in Solution Explorer
2. Select 'Add > Docker Support'
3. Choose Linux or Windows containers
4. VS generates a Dockerfile with appropriate settings
5. Debugging support is automatically configured

VS Docker Integration Features

- One-click containerization
- Integrated debugging in containers
- Container Tools window
- IntelliSense for Dockerfiles
- Built-in container orchestration
- F5 experience for containers

Anatomy of a .NET Dockerfile

```
# Build stage
FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /src
COPY ["WebApi.csproj", "./"]
RUN dotnet restore "WebApi.csproj"
COPY . .
RUN dotnet build "WebApi.csproj" -c Release -o /app/build

# Publish stage
FROM build AS publish
RUN dotnet publish "WebApi.csproj" -c Release -o /app/publish /p:UseAppHost=false

# Final stage
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "WebApi.dll"]
```

Multi-stage build: Smaller final image with only runtime dependencies

Mühlehner & Tavolato GmbH

Understanding Docker Images for .NET

Common .NET Images

- **dotnet/sdk**: Development environment
- **dotnet/aspnet**: ASP.NET runtime
- **dotnet/runtime**: .NET runtime
- **dotnet/monitor**: .NET monitoring tools

Image Tags

- **7.0**: Major.Minor version
- **7.0.5**: Major.Minor.Patch
- **7.0-bullseye-slim**: Version + OS variant
- **latest**: Most recent version (avoid in production)

.NET provides official images for multiple architectures (x64, ARM64)

Running .NET Applications in Docker

Basic Commands

```
# Build image from Dockerfile
docker build -t myapp .

# Run container from image
docker run -p 8080:80 myapp

# View running containers
docker ps

# Stop container
docker stop <container_id>

# View logs
docker logs <container_id>
```

VS Integration

- Press F5 to build and run in container
- Set breakpoints for container debugging
- View container logs in output window
- Use Container Tools window to manage containers

Visual Studio handles `docker build` and `docker run` commands behind the scenes.

Multi-Container Applications with Docker Compose

What is Docker Compose?

- Tool for defining and running multi-container applications
- Uses YAML file to configure all services
- Manages networking between containers
- Simplifies container startup and shutdown

Adding Container Orchestration

1. Right-click on solution in Solution Explorer
2. Select 'Add > Container Orchestrator Support'
3. Choose Docker Compose
4. VS generates docker-compose.yml and updates configuration

Sample docker-compose.yml

```yaml
version: '3.4'

services:
  webapi:
    image: ${DOCKER_REGISTRY-}webapi
    build:
      context: .
      dockerfile: WebApi/Dockerfile
    ports:
      - "5000:80"
    depends_on:
      - sqlserver
      - redis

  sqlserver:
    image: mcr.microsoft.com/mssql/server:2019-latest
    environment:
      - ACCEPT_EULA=Y
      - SA_PASSWORD=YourStrong!Passw0rd
    ports:
      - "1433:1433"
    volumes:
      - sqlserver_data:/var/opt/mssql

  redis:
    image: redis:alpine
    ports:
      - "6379:6379"

volumes:
  sqlserver_data:
```

Configuring Application for Docker Environment

Connection Strings

```
// appsettings.json
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=sqlserver;Database=MyDb;U
    "Redis": "redis:6379"
  }
}
```

Note service names used as hostnames

Service Registration

```
// Program.cs
var builder = WebApplication.CreateBuilder(args);

// Database
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(
        builder.Configuration.GetConnectionString(
            "DefaultConnection")));

// Redis Cache
builder.Services.AddStackExchangeRedisCache(options =>
{
    options.Configuration =
        builder.Configuration.GetConnectionString("Redis")
    options.InstanceName = "SampleInstance";
});
```

Exercise: Dockerize a .NET Application

Task

1. Add Docker support to the task manager API
2. Modify it to use SQL Server in a container
3. Configure Docker Compose
4. Run and test the application

Project Structure

- TaskManager.API (ASP.NET Core Web API)
- TaskManager.Data (EF Core)
- TaskManager.Models (Domain models)

CONNECTED SERVICES

with Visual Studio 2022

Mühlehner & Tavolato GmbH

Introduction to Connected Services

What are Connected Services?

- Tools to easily integrate external services
- Code generation for API consumption
- Configuration for authentication and endpoints
- Service-specific NuGet packages

Supported Service Types

- Azure services (Storage, Key Vault, etc.)
- gRPC services
- OpenAPI (Swagger) definitions
- WCF services
- SQL Server databases
- REST APIs

Adding Connected Services

Steps

1. Right-click on project in Solution Explorer
2. Select 'Add > Connected Service'
3. Choose service type
4. Configure service-specific options
5. Click 'Add'

VS adds NuGet packages, configuration, and possibly generated code.

Connected Services Window

- Provides overview of all connected services
- Allows managing service configurations
- Simplifies updating service references
- Displays dependencies for each service

Azure Storage Connected Service

Configuration

```
// appsettings.json
{
  "ConnectionStrings": {
    "AzureStorage":
    "DefaultEndpointsProtocol=https;AccountName=mystorageaccount;AccountKey=mykey;EndpointSuffix=core.windows.net"
  }
}
```

```
// Program.cs
builder.Services.AddSingleton(x ⇒
    new BlobServiceClient(
        builder.Configuration.GetConnectionString(
            "AzureStorage")));
```

Usage Example

```
[ApiController]
[Route("[controller]")]
public class FilesController : ControllerBase
{
    private readonly BlobServiceClient _blobClient;

    public FilesController(BlobServiceClient blobClient)
    {
        _blobClient = blobClient;
    }

    [HttpPost("upload")]
    public async Task<IActionResult> UploadFile(
        IFormFile file)
    {
        var containerClient =
            _blobClient.GetBlobContainerClient("files");
        await containerClient.CreateIfNotExistsAsync();

        var blobClient =
            containerClient.GetBlobClient(file.FileName);
        using var stream = file.OpenReadStream();
        await blobClient.UploadAsync(stream, true);

        return Ok(new { url = blobClient.Uri.ToString() });
    }
}
```

Mühlehner & Tavolato GmbH

OpenAPI (Swagger) Connected Service

Adding the Service

1. Right-click on project > Add > Connected Service
2. Select 'OpenAPI'
3. Enter URL to Swagger document (e.g., https://petstore.swagger.io/v2/swagger.json)
4. Configure namespace and other options
5. Click 'Add'

VS generates client classes for the API.

Using Generated Client

```
public class PetController : ControllerBase
{
    private readonly PetStoreClient _client;

    public PetController(PetStoreClient client)
    {
        _client = client;
    }

    [HttpGet("available")]
    public async Task<IActionResult> GetAvailablePets()
    {
        var availablePets =
            await _client.FindPetsByStatusAsync(
                new[] { "available" });

        return Ok(availablePets);
    }
}
```

Azure Cosmos DB Connected Service

Configuration

```
// appsettings.json
{
  "CosmosDb": {
    "Endpoint": "https://myaccount.documents.azure.com:443/",
    "Key": "mykey==",
    "DatabaseName": "Tasks",
    "ContainerName": "Items"
  }
}
```

```
// Program.cs
builder.Services.AddSingleton<CosmosClient>(sp =>
{
    var config = sp.GetRequiredService<IConfiguration>();
    return new CosmosClient(
        config["CosmosDb:Endpoint"],
        config["CosmosDb:Key"]);
});
```

Usage Example

```
public class TasksRepository : ITasksRepository
{
    private readonly CosmosClient _client;
    private readonly Container _container;

    public TasksRepository(CosmosClient client,
                           IConfiguration config)
    {
        _client = client;
        var dbName = config["CosmosDb:DatabaseName"];
        var containerName = config["CosmosDb:ContainerName"];
        _container = _client.GetContainer(dbName, containerName);
    }

    public async Task<TaskItem> GetTaskAsync(string id)
    {
        try
        {
            var response = await _container.ReadItemAsync<TaskItem>(
                id, new PartitionKey(id));
            return response.Resource;
        }
        catch (CosmosException ex) when (ex.StatusCode == HttpStatusCode.NotFound)
        {
            return null;
        }
    }
}
```

Exercise: Add and Use Connected Services

Task

1. Add Azure Cosmos DB Connected Service
2. Configure it to store and retrieve data
3. Bonus: Add OpenAPI service to connect to an external API

Sample Application

- Task management API
- Needs persistent storage for tasks
- Could benefit from integration with external API for task categorization

Wrap-up and Next Steps

What We've Covered

- Unit testing with xUnit and Moq
- Live Unit Testing in Visual Studio 2022
- Docker support for .NET applications
- Connected Services for external integrations

Further Learning

- CI/CD with GitHub Actions or Azure DevOps
- Kubernetes for container orchestration
- Advanced testing techniques (integration, performance)
- Cloud-native development with .NET
- Microservices architecture

The End