Dependency Injection & Mocking

Lukáš Daubner

Contents

- Dapper
- Dependency Injection
- Mocking

Dapper

Dapper

- So-called micro-ORM
- Lightweight
- Basically a wrapper around ADO.NET
- Uses a lot of SQL

SELECT

```
var _connection = new SqliteConnection("Filename=:memory:");
_connection.Open();

var selectQuery = @"SELECT Id, Name, Start, Location, Contact
                    FROM Courses WHERE Id = @Id;";

Course course = await _connection.QueryFirstAsync<Course>(selectQuery, new
{
    Id = courseId
});
```

INSERT

```
var _connection = new SqliteConnection("Filename=:memory:");
_connection.Open();

var insertQuery = @"INSERT INTO Courses (Id, Name, Start, Location, Contact)
                    VALUES (@Id, @Name, @Start, @Location, @Contact);";

await _connection.ExecuteAsync(insertQuery, new
{
    Id = Guid.NewGuid(),
    Name = course.Name,
    Start = course.Start,
    Location = course.Location,
    Contact = course.Contact,
});
```

Dependency Injection

Inversion of Control (IoC)

- A.K.A. Hollywood principle
  - Don't call us, we'll call you
- A general idea that a framework should instantiate dependencies
- Framework is the coordinator
- We just implement the parts that the framework can bind

Dependency Injection (DI)

- Concrete IoC technique
- All the dependencies are bind outside of the class that needs them
- When framework instantiates a class, it provides the dependencies
- They are "injected" into the class
- By constructor/property/method

Dependency Inversion Principle (soliD)

- "High-level modules should not depend on low-level modules. Both should depend on abstractions."
- "Abstractions should not depend on details. Details should depend on abstractions."
- Essentially this forces loose coupling
- Instead of calling (depending on) a concrete class, we depend on an interface.

Example

```
public class CourseService
{
    private readonly CourseRepository _courseRepository;

    public CourseService()
    {
        _courseRepository = new CourseRepository();
    }
    ...
}
```

Example

```
public class CourseService : ICourseService
{
    private readonly ICourseRepository _courseRepository;

    public CourseService(ICourseRepository courseRepository)
    {
        _courseRepository = courseRepository;
    }
    ...
}
```

IoC/DI Containers

- Frameworks implementing dependency injection
- Handling the RRR lifecycle
  - Register
  - Resolve
  - Release
- How exactly they treat the objects depends on the container
  - Some track the whole object lifecycle
  - Some just returns an object based on specification

IoC/DI Containers in .NET

- Many to choose from
  - AutoFac
  - SimpleInjector
  - Castle.Windsor
  - Ninject
- They have different features, support different object lifecycles
- Currently, they are quite overshadowed by Microsoft.Extensions.DependencyInjection

Lifecycles

- Typical lifecycles are:
- **Transient** - each resolution gives new instance
- **Singleton** - one global instance
  - Preferred way of implementing singleton!
- **Scoped** - depending on an explicit scope
- **PerGraph** - one per dependency graph
- Keep in mind that not every container implements them or calls them the same

AutoFac Example

Register

```
builder
.RegisterType<CourseService>()
.InstancePerDependency()
.As<ICourseService>();
```

Resolve

```
var courseService = scope.Resolve<ICourseService>();
```

Demo

Mocking

Mock

- Mock is a fake object used for testing
- The aim is to remove dependencies and focus on the tested component
- Allows us to define deterministic behavior for the tests
- Essential for true unit testing

Mocking, DI, and soliD

- Dependency Injection and Dependency Inversion Principle allows easy definition of mocks
  - Because the dependencies are always outside of the tested class
  - The class depends on interfaces, they do not care about implementation (which can be the mock)

Mocking using Moq

```
var expectedGuid = Guid.NewGuid();

Mock<ICourseRepository> _courseRepositoryMock =
  new Mock<ICourseRepository>();

_courseRepositoryMock
    .Setup(x => x.CreateAsync(It.IsAny<Course>()).Result)
    .Returns(expectedGuid);

CourseService service = new CourseService(_courseRepositoryMock.Object);
```

Demo

Task

- Register the CourseService and EnrollmentService to AutoFac and try to resolve it
- Implement CourseService and EnrollmentService using TDD
  - First write the tests
  - Then write the code
- Mock all dependencies
  - *HINT:* Keep it simple, do not re-implement the code using mocks