Enforcing software architecture governance for .net projects using unit tests

- Post published:July 5, 2020

Discussions around software architecture is the most important phase in any enterprise software engineering project. Most software architectural decisions are documented at the start of a project. This documentation is vital to the software development team since it shares a common language and understanding of how components interact with one another.

Normally software architects and/or technical leads are responsible for ensuring that the development team adhere to the architectural decisions by continuously

- involving the development team in architectural discussions
- keeping a record of each architectural decision
- maintaining architectural decision records
- reviewing software designs,
- implementing and maintaining static code analyzers, and
- performing intensive code reviews.

Today, most Agile methodologies are encouraging rapid software development to implement or enhance features as quickly as possible. As a result, the project continuously grows with a code base getting larger and larger, thus forcing architects and/or technical leads to overlook lines of source code when performing code reviews. Also, software architecture documents are not updated continuously and these documents seem to be forgotten as the code base increases and as developers come and go. Overtime, the governance around the software architecture reduces.

Architecture unit testing is one way of enforcing governance of architectural decisions. This type of unit testing focuses on testing the code base in the context of the software architecture and can be easily automated in a build pipeline.

Having architecture unit testing provides the following automated architecture governance checking

- dependencies between projects
- dependencies between classes, members and interfaces
- dependencies between layers
- cyclic dependencies and more.

Two common open source libraries that can be used to write application-specific architecture tests in .net are

- NetArchTest and
- ArchUnitNet.

Getting started

Read the onion architecture article located here since the remainder of this article will focus on writing a few architecture unit tests to ensure that the onion architecture is adhered to. The examples below use NetArchTest with xUnit.

NetArchTest can be installed using nuget

```
Install-Package NetArchTest.Rules
```

Example

The domain layer is in the very center of the onion architecture. Since all coupling in an onion architecture is toward the center, the domain is only coupled to itself.

Thus, in the application core, the domain layer should not reference the repositories layer and the services layer. Using NetArchTest, two architecture unit tests can be written. Consider the below test that checks if the domain layer references the repositories layer.

```
[Fact]
public void Architecture_DomainLayer_ShouldNotHaveReferenceToRepositoriesLayer()
{
    Types types = Types.InAssembly(typeof(ICoreModule).Assembly);

    bool actualResult = types
        .That()
        .ResideInNamespace("Demo.Core.Domain")
        .ShouldNot()
        .HaveDependencyOn("Demo.Core.Repositories")
        .GetResult()
        .IsSuccessful;

    actualResult.Should().BeTrue();
}
```

DomainTests.cs hosted with ❤ by **GitHub** **view raw**

As can be seen from the above code base, NetArchTest provides a very simple fluent API to write architecture unit tests. Similarly, a test to check that the domain layer does not reference the services layer can be written as follows.

```
[Fact]
public void Architecture_DomainLayer_ShouldNotHaveReferenceToServicesLayer()
{
    Types types = Types.InAssembly(typeof(ICoreModule).Assembly);

    bool actualResult = types
        .That()
        .ResideInNamespace("Demo.Core.Domain")
        .ShouldNot()
        .HaveDependencyOn("Demo.Core.Services")
        .GetResult()
        .IsSuccessful;

    actualResult.Should().BeTrue();
}
```

DomainTests.cs hosted with ❤ by **GitHub** **view raw**

The above architecture unit tests are considered to be rules and are tested independently. NetArchTest also provides the ability to group rules into policies. Consider the below policy for the domain layer.

```
[Fact]
public void Architecture_DomainLayer_ShouldAdhereToTheOnionArchitectureDomainLayerPolicy()
{
    PolicyDefinition domainLayerPolicy = Policy
```

```
5           .Define("Onion Architecture Domain Layer Policy", "The domain layer should not have
6           .For(Types.InAssembly(typeof(ICoreModule).Assembly))
7           .Add(t => t.That()
8                       .ResideInNamespace("Demo.Core.Domain")
9                       .ShouldNot()
10                      .HaveDependencyOn("Demo.Core.Repositories"),
11                "Enforcing Domain Layer Policy", "Domain layer should not have references to t
12          )
13          .Add(t => t.That()
14                      .ResideInNamespace("Demo.Core.Domain")
15                      .ShouldNot()
16                      .HaveDependencyOn("Demo.Core.Services"),
17                "Enforcing Domain Layer Policy", "Domain layer should not have references to t
18          );
19
20      bool actualResult = domainLayerPolicy.Evaluate().HasViolations;
21
22      actualResult.Should().BeFalse();
23  }
```

DomainLayerPolicy.cs hosted with ❤ by **GitHub** **view raw**

Policies are an effective way to group rules. After executing a policy, the result of each failed rule can also be outputted in a test. An example of this can be seen in the NetArchTest sample.

Summary

Enforcing software architecture governance is a challenge in most enterprise software engineering projects. To make this process more robust, architecture unit testing can be implemented as a possible solution to enforce governance of architectural decisions.

By having automatic architecture tests, the

- architecture can evolve as the project grows,
- architects/developers can ensure that new components/features adhere to the software architecture, and
- developers are encouraged to understand and contribute to architectural decisions.

Thanks to the hard working open-source community, libraries such as NetArchTest and ArchUnitNet exist to enable easy creation and maintenance of architecture unit tests, thereby enforcing and automating software architecture governance.

The source code used in this article can be found here.

Further information on architecture unit testing can be found at the following links:

- Writing arch unit tests for dotnet
- NetArchTest
- ArchUnitNET
- ArchUnit

Software versions used in this article

- Jetbrains Rider v2020.1.3
- NetArchTest.Rules v1.2.5

- xUnit v2.4.0
- FluentAssertions v5.10.3