Architectural Unit Testing

- Posted by:
- travis
- No Comments

I'm going to assume that you already buy into the advantages of unit testing your code, and the merits of doing so don't need to be enumerated in yet another article. Perhaps more interesting and unique in today's software development practices is the idea of unit testing your architecture!

Architecture itself has MANY definitions and may mean something slightly different to each person (i.e. "the hard stuff"). In this context, when I refer to *unit testing architecture*, I mean to refer to the common expectations and contracts of what modules or projects are allowed to reference and use. Even more specifically, a project may rely upon certain agreed-upon naming conventions of objects and classes or interfaces. Often these types of discussions and patterns are communicated between team leads and initial design patterns put down on project start. Very quickly the project grows, evolves and other contributors begin working on a project, and these initial non-codified "architectural" requirements are quickly forgotten and you've lost your architectural purity.

Architectural purity… why do I care about that? The project still seems functional and the team continues to release daily. Perhaps pushing for that clean dependency graph between modules was a waste of time in the first place? If you are asking these questions, then your project is likely not mature enough to suffer from the aftermath of this erosion. Circular references, shared monolithic libraries, and zero domain driven design abstractions will quickly push your project to a halt and make any type of refactoring incredibly difficult (not to mention the extended and exhausted build times that could ensue).

In my experience, the only way to ensure that any requirement (business or technical) lasts for the length of a project is to have it codified into the repository somewhere. The easiest method of that is to include into the "README.md" or other related documentation that exists and evolves alongside the source code. Unfortunately as markdown, none of this information will be enforced and requires some pre-knowledge of its existence (but hopefully is made to be decently self-discoverable by the team). Of course, the other method requires a bit more work, but if you can begin to code these rules as unit tests, then you can begin to enforce the architectural requirements without an engineer having any pre-knowledge of it at all. While you can achieve this using your own code and Reflection libraries, it looks pretty ugly and unmaintainable. That's where a framework like ArchUnit comes in.

ArchUnit (Java)

ArchUnit is a Java library that you can add to your unit tests that provides a very fluent API method of validating some of the architectural constraints that you might often want to add to a project.

ArchUnit allows you to be pretty specific to a particular class or broader in a given package/import. As an example, you can easily look for all classes in a given package with a particular annotation to validate that the name is prefixed with a value (such as "Service"):

```
@ArchTest
public static ArchRule services_should_be_prefixed =
  classes()
    .that().resideInAPackage("..service..")
    .and().areAnnotatedWith(MyService.class)
    .should().haveSimpleNameStartingWith("Service");
```

The possibilities of what this API will allow you to test are pretty endless. There are some great example projects to look at and review. To name a few of the scenarios:

- **Module Dependencies** – test proper layers of module references, specifically look for cyclic dependencies, identify modules that should never reference each other.

- **Limit External Dependency** – enforce that a project must use a certain dependency or not use a certain dependency.
- **Naming Conventions** – test for naming conventions of services and controllers. Additionally, apply broader rules for private variable naming or even module naming.
- **Coding Conventions** – test that objects are instantiated in a certain way, dependency injection is enforced, etc.
- **DAO** – enforce that all data access to your databases happens in a certain module and that the DAO is only referenced by certain service level implementations.

Great article from Java Magazine on ArchUnit for more a in-depth look.

NetArchTest (.NET)

Inspired by ArchUnit style tests, the NetArchTest project allows you to test your architectural constraints in .NET. A lot of the teams I work on are either Java or .NET based, and these two libraries provide very similar feature parity and capability for creating architectural unit tests. NetArchTest is also fully fluent and you'll feel right at home.

A simple example from GitHub that shows how you can validate that references between presentation or UI layer do not directly link to the data access repositories:

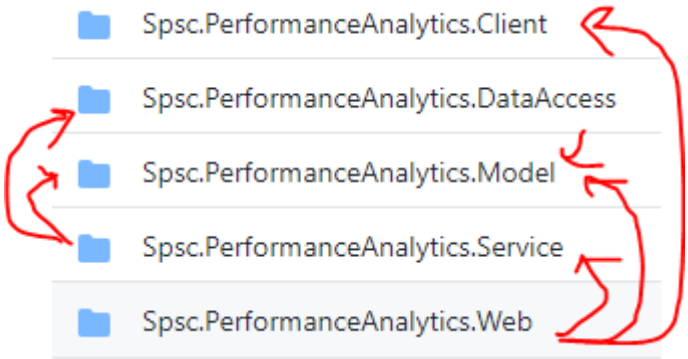
```
// Classes in the presentation should not directly reference repositories
var result = Types.InCurrentDomain()
  .That()
  .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
  .ShouldNot()
  .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
  .GetResult()
  .IsSuccessful;
```

While I have had good success with the NetArchTest library, you should be aware that ArchUnit has an official library now that supports .NET, ArchUnitNET. A little bit newer and different licenses on the library, definitely consider using ArchUnitNET if it is an advantage to have a similar interface between your Java and .NET projects. That being said, I have not had the opportunity to explore ArchUnitNET in-depth… leave some comments if you have tried it and how it compares!

Examples

The following is an example of how one of our teams at SPS uses the architectural unit tests to initialize a new service in .NET Core.

The .NET Core seed solution contains 5 projects for a layered architecture, each with a specific purpose and very intentional set of dependencies on other projects.



As you'd expect, our Web project specifically is only allowed to reference the Service project (i.e. not the DataAccess). Additionally, we want to ensure that the Service layer is entirely agnostic of the Web or UI project. What may seem a bit funny is that the Web project depends on the Client project. The client project

is often a high fidelity HTTP Client built with Refit, but they share the contractual web models defined in a single spot in the client (to prevent the necessity of creating a separate distributed NuGet package for the shared model). This is a really fantastic, but not so intuitive pattern that I hope a future blog post can reveal a bit more. For now, understand it is essential that Web depends on Client, but that NOTHING else can and Client can have no other internal project references. We can codify these rules like this:

```
[TestMethod]
public void ClientDependsOnNothingTest()
{
  var result = Types
    .InAssembly(typeof(SupplierConfigurationServiceFactory).Assembly)
    .ShouldNot()
    .HaveDependencyOnAny(
      "Spsc.PerformanceAnalytics.DataAccess",
      "Spsc.PerformanceAnalytics.Model",
      "Spsc.PerformanceAnalytics.Service",
      "Spsc.PerformanceAnalytics.Web")
    .GetResult();

  Assert.IsTrue(result.IsSuccessful,
    "HTTP Client Project must not reference any other projects.");
}

[TestMethod]
public void ModelDependsOnNothingTest()
{
  var result = Types
    .InAssembly(typeof(AppSettings).Assembly)
    .ShouldNot()
    .HaveDependencyOnAny(
      "Spsc.PerformanceAnalytics.DataAccess",
      "Spsc.PerformanceAnalytics.Service",
      "Spsc.PerformanceAnalytics.Web")
    .GetResult();

  Assert.IsTrue(result.IsSuccessful,
    "Model Project must not reference any other projects.");
}

[TestMethod]
public void ServiceAgnosticDependencyTest()
{
  var result = Types
    .InCurrentDomain()
    .That()
    .ResideInNamespace("Spsc.PerformanceAnalytics.Service")
    .ShouldNot()
    .HaveDependencyOnAny(
      "Spsc.PerformanceAnalytics.Web",
      "Spsc.PerformanceAnalytics.Client"
    )
    .GetResult();

  Assert.IsTrue(result.IsSuccessful,
    "Service Project must not reference Web or Client projects.");
}

[TestMethod]
public void DataAccessAgnosticDependencyTest()
{
  var result = Types
    .InCurrentDomain()
    .That()
    .ResideInNamespace("Spsc.PerformanceAnalytics.DataAccess")
    .ShouldNot()
    .HaveDependencyOnAny(
      "Spsc.PerformanceAnalytics.Service",
      "Spsc.PerformanceAnalytics.Web",
```

```
      "Spsc.PerformanceAnalytics.Client")
  .GetResult();

  Assert.IsTrue(result.IsSuccessful,
    "Data Access Project must not reference any other projects except model.");
}
```

For the Architect and the Enterprise

As an architect myself, I'm constantly interested in how software development practices around coding, style, CI/CD, security, and SDLC in general hold up when applied at the enterprise level to an organization. Currently, the discussion above and patterns at SPS is not something that is universally applied to all teams, but rather adopted by individual teams as they begin to see the benefit.

From the Architect's perspective, it is nice to be able to codify these fitness functions directly as a project starts to ensure its longevity in support of this purity. In the example shown above, it is pretty simple and not incredibly comprehensive to the full capability of the framework. But it is worth asking the question of how far you would actually want to take the usage of this framework and testing, especially in a Microservice or even a TinyService world. In some cases, a framework like ArchUnit is a lot more valuable in the organization when looking at monolithic style applications or mono-repo style applications where such enforcement is more of a problem. By default we see more natural barriers form for multi-repo and Microservices that make Architectural purity at this level less of a concern (you have a lot of other concerns to deal with in the distributed application architecture where this framework won't be able to assist you).

As with anything related to "Architecture", the question on if and how you should implement architectural testing in your organization will come to "it depends". That being said, it seems like there is an appropriate niche for architectural unit testing as a decorator to enhance or to fill the gap based on the architectural form-factor you are following.

Moving forward, I'd like to consider how Architectural Unit Tests can be pre-baked into standardized organizational service templates, seed templates, and scaffolding to help encourage its usage and pre-knowledge of its usage for other team leads and architects.