TESTING

Unit Test Your Architecture with ArchUnit

Discover architectural defects at build time.

by Jonas Havers

August 19, 2019

A common problem faced by development organizations is that code implementations can often diverge from the original design and architecture. The problem is common enough, especially on large projects, that a new tool has emerged to help test that a code implementation is consistent with the originally defined architecture. ArchUnit is a small, simple, extensible, open source Java testing library for verifying predefined application architecture characteristics and architectural constraints. An ArchUnit test is written and runs as a unit test that gives developers and application architects fast feedback on their work. It guarantees that a software build will break if an architectural violation is introduced.

In this article, I explain why you should test your architecture and how to get started with ArchUnit to do exactly that. You can find the code snippets from this article as well as further examples on my GitHub repository.

Why Should You Test Your Architecture?

Software architecture is an important prerequisite for the comprehensibility and changeability of codebases as well as for adherence to software quality goals. Three major goals of software architecture, when it comes to the codebase, are maintainability, replaceability, and extensibility. Maximizing the ability to achieve them enables teams to iterate quickly; that is, to add features and fix bugs quickly when an application grows. To keep a software system maintainable, replaceable, and extendable, you need to ensure that it is modular and that interdependencies are as small and correct as possible, leading to high cohesion and loose coupling.

These goals can be met by introducing certain patterns and code conventions that are comprehensively documented and communicated by and for the entire development team that has agreed to them.

An after-the-fact introduction of automated architecture verification is also suitable for an existing software system that has become hard to maintain. Tests can guide you to reduce technical debt step by step. In this way, you can monitor and verify progress towards the target project architecture.

Preparing to Write Architecture Tests

To test concepts and structures of an application architecture, you must typically map the architecture model to the technical code first or derive that from the existing application. When you do this, it is necessary not only to think about the code structure in advance but also about how domain concepts and technical concepts are or will be identifiable in the application code. This includes the identification of technical terms for

concrete application components such as “controller” and “service” as well as domain-specific terms such as “product” and “customer” and their relationships to each other.

If you have agreed on terms, relationships, and conventions, a system of static architecture rules can be created that enables developers to write and modify code that complies with the rules. That means the mental model can be mapped by implementing the concepts using layers and packages, class naming schemes, annotations, and so on.

Why Choose ArchUnit?

The ArchUnit library was initially created by Peter Gafert in 2017. It enables you to import the classes of your application into a special Java code structure that allows you to test the code and its structures with any Java unit testing framework. ArchUnit lets you implement rules for the static properties of application architecture in the form of executable tests such as the following:

- Package dependency checks
- Class dependency checks
- Class and package containment checks
- Inheritance checks
- Annotation checks
- Layer checks
- Cycle checks

Furthermore, you can create custom tests for constructors, methods, fields, members, and “code units” (that is, a method, constructor, or static class initializer that can access other code). With these checks, you can verify both specific architectural constraints as well as coding rules, such as naming conventions.

There are various tools to test for dependencies between layers and packages, which is a major concern for architecture verification, especially for a layered architecture. Although many coding guidelines (such as the naming of classes and members) can be handled by code linting tools, these tools have a limited scope and can be only partially used when the rules become more complex. As a consequence, certain rules can be enforced only by the combination of multiple tools (and sometimes not at all). The use of different tools, APIs (such as the Java Reflection API and AspectJ), and additional languages (such as Groovy) in a custom test suite eventually results in an unnecessarily difficult learning curve, especially for beginners.

In contrast, ArchUnit does not need any special infrastructure or any new language. Tests that verify the rules can be written in plain Java. ArchUnit can also handle test code and application code in other languages that are translated to Java bytecode (such as Kotlin). The rules can be evaluated with any unit testing tool. However, there is extended support for writing tests with JUnit 4 and JUnit 5.

In an evolving architecture, implementation rules evolve over time. The use of built-in, automatically executed tests forces an organization to consciously accept deviations from predefined rules instead of accidentally encountering them later in a review or never. ArchUnit can provide warnings about deviations.

In my experience, once you start to write rules with ArchUnit, you notice more and more use cases: the incorrect use of third-party libraries that can be avoided, code smells that can be banned from the project by testing against these patterns, and so on. The ArchUnit APIs stimulate creativity and make it easy to come up with new checks or to improve existing rules in order to increase the overall code quality.

Furthermore, the ArchUnit library provides typical predefined rules. Because it is difficult for a library to predict all its uses, it is all the more important that it can be extended. To this end, ArchUnit provides a

convenient way to write custom checks by using a few predefined and combinable building blocks and interfaces.

Under the Covers

ArchUnit makes use of reflection and Java bytecode analysis. Information that the library cannot obtain through the Java Reflection API is obtained at the bytecode level. For example, the Reflection API offers no way to retrieve information about access to or from a class. However, this information *is* contained in the bytecode. So dependencies between two classes can be obtained only through analysis of the Java bytecode. To analyze the bytecode, the ArchUnit library uses ASM, an all-purpose Java bytecode reading, writing, and manipulation framework (for more information, see previous coverage of ASM in this magazine).

With ASM, you can write tests that look very much like unit tests but which target architectural constraints. For example, an architectural rule written with ArchUnit might look like this:

```
ArchRule rule = ArchRuleDefinition.classes()
    .that().resideInAPackage("..domain..")
    .should().onlyBeAccessed()
    .byAnyPackage("..domain..", "..application..");
```

Here's another example:

```
ArchRule rule = ArchRuleDefinition.methods()
    .that().arePublic()
    .and().areDeclaredInClassesThat().resideInAPacka
    .and().areDeclaredInClassesThat().haveSimpleNam
    .and().areDeclaredInClassesThat().areAnnotatedW
    .should().beAnnotatedWith(RequestMapping.class)
```

ArchUnit is divided into three main API layers: Core, Lang, and the Library layer, which I'll explore after the following section.

Getting Started

You can obtain the ArchUnit library from Maven Central. You first need to declare and then pull the dependency with Maven or Gradle. After that, you are ready to write and execute tests with any Java unit testing framework you choose. Here is the POM entry for Maven users:

```
<dependency>
    <groupId>com.tngtech.archunit</groupId>
    <artifactId>archunit</artifactId>
    <version>0.11.0</version>
    <scope>test</scope>
</dependency>
```

And here's the POM for Gradle users:

```
dependencies {
    testCompile 'com.tngtech.archunit:archunit:0.11
}
```

For JUnit, ArchUnit provides an `ArchUnitRunner` that reduces boilerplate code and caches the imported classes by URL, so classes for multiple tests do not need to be imported each time. To get the runner support for JUnit 4, replace the `archunit` artifact in the dependencies above with `archunit-junit4`. For JUnit 5, use the `archunit-junit5-api` and `archunit-junit5-engine` artifacts instead. The installation guide provides additional information.

When you use the `ArchUnitRunner` with JUnit, `ArchRules` can be defined either as static fields or static methods with a single `JavaClasses` argument. In both cases, you need to add the `@ArchTest` annotation so the classes that are already imported and cached are reused by the runner and the rules are evaluated against them, for example:

```
@RunWith(ArchUnitRunner.class)
@AnalyzeClasses(
    packages = "com.company.app",
    importOptions =
        ImportOption.DoNotIncludeTests.class
)
public class ArchitectureRulesTest {

    @ArchTest
    public static final ArchRule ruleAsStaticField =
              ArchRuleDefinition.classes()
                                .should()...

    @ArchTest
    public static void ruleAsStaticMethod(JavaClasse
        ArchRuleDefinition.classes()
                          .should()...
    }
}
```

With JUnit 5, the runner does not need to be declared. So you can delete the first line: `@RunWith(ArchUnitRunner.class)`.

As I mentioned before, due to ArchUnit's nature of writing checks as unit tests, there is no need to introduce a special step in your continuous integration (CI) pipeline. You can incorporate the test suite into any CI environment and deployment pipeline.

You can exclude known architectural violations from breaking the tests by using a file named `archunit_ignore_patterns.txt` in the classpath root directory. Every line in the file will be interpreted as a regular expression and checked against reported violations. Violations with a message matching the pattern will be ignored. To ignore single tests, use the annotation `@ArchIgnore` in the test classes. Since v. 0.11.0, you can also use the `FreezingArchRule` feature, which stores the current state of violations when there are way too many rule violations to fix immediately. Usually, though, ignoring tests is a bad practice. However, by using these options, you can integrate ArchUnit into existing software projects that contain known architectural flaws and erosions. The projects can then be iteratively fixed and migrated to a clean architecture that aims to satisfy the rules.

There is also a third-party Maven plugin that enables you to easily share and enforce common architecture rules across different projects instead of copying them.

Inside ArchUnit

As I mentioned earlier, ArchUnit is divided into three principal APIs: Core, Lang, and the Library layer.

The Core API. This layer resembles the Reflection API. In addition, it deals with the basic infrastructure, such as how to import bytecode. To import compiled Java class files, use the `ClassFileImporter`. It offers many ways to import compiled Java classes, but the most convenient is to declare one or more base packages, as shown below. It allows specific locations to be filtered out, such as test classes or archives such as JARs.

```
JavaClasses classes = new ClassFileImporter()
    .withImportOption(ImportOption.Predefined.DO_NO
    .withImportOption(ImportOption.Predefined.DO_NO
    .importPackages("com.company.app");
```

What you get back from the importer is an instance of `JavaClasses`, which represents a collection of elements of type `JavaClass`. A `JavaClass` in turn represents a single imported class file.

Core objects are named after their Reflection API counterparts but with an additional Java prefix. In addition, there are representational classes in the Core API such as `JavaPackage`, `JavaMethod`, and `JavaField`. Developers who have used the Reflection API before will be familiar with methods such as `getName()`, `getMethods()`, `getRawType()`, or `getRawParameterTypes()`, which have counterparts in the ArchUnit Core API.

Additional access information from the bytecode can be obtained via `JavaMethodCall`, `JavaConstructorCall`, and `JavaFieldAccess`. For example, by calling `getAccessesFromSelf()` on a `JavaClass` instance, you can iterate over and analyze the accesses between this imported Java class and other imported Java classes. You can perform similar access checks with the object methods `getAccessesToSelf()`, `getFieldAccessesFromSelf()`, and `getFieldAccessesToSelf()`.

The Lang API. A pure use of the core classes for architecture tests is possible, but expressiveness is missing here. To address this, ArchUnit offers a higher level of abstraction with the Lang API, which provides a fluent interface that consists of three central components:

- `DescribedPredicate`: A predicate for selecting relevant classes
- `ArchCondition`: A condition that selected classes must fulfill
- `ArchRule`: A rule to define architectural concepts

Because most parts of the Lang API are also composed as a fluent API, an IDE can provide valuable suggestions on the APIs to use.

The class `ArchRuleDefinition` is used as an entry point to define an `ArchRule`. With the three API components mentioned previously, you can also go beyond the provided predicates and conditions and develop your own to satisfy your needs.

The following example verifies that domain classes should be accessed only by other domain classes or application classes:

```
ArchRule rule = ArchRuleDefinition.classes()
    .that().resideInAPackage("..domain..")
    .should().onlyBeAccessed()
        .byAnyPackage("..domain..", "..application.
```

This syntax is inspired by AspectJ pointcuts.

The "`..`" in the package notation refers to any number of packages. This syntax is inspired by AspectJ pointcuts. Thus, in this example, the `ArchRule` applies to any class inside the package `com.company.app.domain.model` (`DescribedPredicate`), for example, and it verifies that it is accessed only by classes in the package or subpackages of `domain` or `application` (`ArchCondition`).

If the check for an ArchUnit rule fails, it will report a `java.lang.AssertionError` that contains the rule text and all violations, including the class and line number.

You can also negate class rules by starting with `ArchRuleDefinition.noClasses()`. In addition to classes, you can directly test for constructors, methods, fields, members, and code units. Each of them also has a negation counterpart.

The following is an extended example of the Lang API to show what more-complex checks can look like. It is focused on the Spring

Framework and can be used to check Spring model view controller (MVC) class methods:

```
ArchRule rule = ArchRuleDefinition.methods()
    .that().arePublic()
    .and().areDeclaredInClassesThat()
          .resideInAPackage("..adapters.primary.web
    .and().areDeclaredInClassesThat()
          .haveSimpleNameEndingWith("Controller")
    .and().areDeclaredInClassesThat()
          .areAnnotatedWith(Controller.class)
    .or().areDeclaredInClassesThat()
         .areAnnotatedWith(RestController.class)
    .should().beAnnotatedWith(RequestMapping.class)
    .orShould().beAnnotatedWith(GetMapping.class)
    .orShould().beAnnotatedWith(PostMapping.class)
    .orShould().beAnnotatedWith(PatchMapping.class)
    .orShould().beAnnotatedWith(DeleteMapping.class
```

This rule checks that public methods inside a controller are annotated with any of Spring MVC's request mapping annotations, so there is no public method that is not used for handling a request. The selected classes have a simple name ending with `Controller`, are annotated with a Spring MVC `@Controller` or `@RestController` annotation, and must reside in a package or a subpackage of `..adapters.primary.web..`. With such rules, you can enforce strong coding conventions at the method level.

Once you compose a rule like the rule above, you need to check it against the imported Java classes for it to affect the test results:

```
JavaClasses classes = new ClassFileImporter() …
ArchRule rule = …
rule.check(classes);
```

The Library API. This layer contains even more abstract and complex predefined rules. For example, ArchUnit makes it possible to create definitions for layered architectures with an instance of `Architectures.LayeredArchitecture` and run checks against the individual layers. You can easily define a layered architecture by defining its layers, names, and packages. A definition for a ports-and-adapters architecture that can be expressed as a layered architecture might look similar to the following:

```
Architectures.LayeredArchitecture portsAndAdaptersA
    Architectures
        .layeredArchitecture()
        .layer("domain layer")
            .definedBy("com.company.app.domain..")
        .layer("application layer")
            .definedBy("com.company.app.application
        .layer("adapters layer")
             .definedBy("com.company.app.adapters..
```

With this architecture definition, you can now define rules against the layers. You can either append the rules directly in a single test or append them on a stored instance variable in separate tests, as shown in the next example. In both cases, the layer condition is added with the `whereLayer` method:

```
ArchRule applicationLayerRule =
    portsAndAdaptersArchitecture
        .whereLayer("application layer")
        .mayOnlyBeAccessedByLayers("adapters layer"
```

ArchUnit v 0.11.0 adds a new predefined `Architectures.onionArchitecture()` API for verifying the semantics of an Onion Architecture, as described by Jeffrey Palermo,

which is related to the ports-and-adapters architecture. It can be used in a similar fashion, but you need to define each adapter:

```
Architectures.OnionArchitecture onionArchitecture =
    Architectures.onionArchitecture()
        .domainModels("com.company.app.domain.model
        .domainServices("com.company.app.domain.serv
        .applicationServices("com.company.app.applic
        .adapter("cli", "com.company.app.adapters.cl
        .adapter("web", "com.company.app.adapters.we
```

Another ArchUnit library API exists for *slices*. Slices are basically rule definitions for subsets of Java classes. Each of these subsets matches a package infix pattern. A `SlicesRuleDefinition` is used to create the slices and to run assertions against them. This results in a `SliceRule` object from the `Slices` API. You can use the `SlicesRuleDefinition` builder to create a `SliceRule` to find cyclic dependencies; that is, to check that slices should be free of cycles or to evaluate that individual slices do not depend on each other. For example, these `SliceRule`s help to detect transitive dependencies.

```
SliceRule layersShouldBeFreeOfCycles =
    SlicesRuleDefinition.slices()
        .matching("com.company.app.(*)..")
        .should().beFreeOfCycles();

SliceRule adaptersShouldNotDependOnEachOther =
    SlicesRuleDefinition.slices()
        .matching("com.company.app.adapters.(**).."
        .should().notDependOnEachOther();
```

Again, the matching notation is inspired by AspectJ syntax. The first rule captures classes in the first package under `app` and checks that those slices are cyclic-free. The second rule groups Java classes in all subpackages of `adapters` (for example, `..adapters.primary.web..` and `..adapters.secondary.mongodb..`) and checks all those slices for any interdependencies.

For example, if you directly access a MongoDB repository (secondary adapter) in a web controller (primary adapter), ArchUnit would produce an error for the second rule above:

```
java.lang.AssertionError: Architecture Violation [Pr
<strong>Slice primary.web calls Slice secondary.mon
```

If I had used a single-star capture, the classes would have been grouped into just two slices, namely primary and secondary, and the error message would reflect that.

The Library API also contains the `GeneralCodingRules` class, which includes predefined and self-describing static coding rules that are common in many Java projects, for example:

- `NO_CLASSES_SHOULD_ACCESS_STANDARD_STREAMS` (that is, the `System.out`, `System.err`, and `printStackTrace` methods: use a logging library instead)
- `NO_CLASSES_SHOULD_THROW_GENERIC_EXCEPTIONS` (for example, rather than throwing `RuntimeException`, use `IllegalArgumentException` or better custom exceptions instead)
- `NO_CLASSES_SHOULD_USE_JAVA_UTIL_LOGGING` (use this when you want the team to use Log4j or Logback behind SLF4J)
- `NO_CLASSES_SHOULD_USE_JODATIME` (use this when you want the team use the modern `java.time` API instead)

What to Keep in Mind When You Write Tests

If you have a large codebase, you should cache class file imports for your tests. This step can greatly reduce the tests' execution time. Caching of the imported classes, for example, can be achieved with the `ArchUnitRunner` when you are using the JUnit support. For small codebases, the reimport overhead of the classes is negligible.

It is important that you write the concepts being tested as precisely as possible and that the concepts communicate clearly what the point of a rule is. For complex rules, it may make sense to use explicit and descriptive rule text instead of relying on the generated rule text. And sometimes a long rule definition can be split into multiple short ones that are easier to understand.

Furthermore, you should not follow the implemented rules blindly. Likewise, you should not change them to pass the tests quickly without thinking beforehand. Sometimes new components do not fit the existing concepts; therefore, existing rules need to be changed or extended. That is especially true in an evolving architecture. But, as mentioned earlier, you should not overuse the options to ignore tests that ArchUnit offers. This is especially true in greenfield projects.

Alternatives to ArchUnit and When to Use Them

At the language level, the Project Jigsaw module system introduced with Java 9 can be a great help to avoid inadvertently introducing dependencies across layers, because module dependencies are explicitly specified. Unfortunately, this feature does not help much with existing applications, especially those that require Java 8 or earlier. Furthermore, Jigsaw guarantees only modularity and dependencies between modules, which is only one of the quality aspects and checks that can be performed with ArchUnit.

An alternative to ArchUnit is jQAssistant, a tool that analyzes a project and stores the generated information in a Neo4j graph database. jQAssistant offers an integration with AsciiDoc, so tests can be embedded in documentation. This is practical if you are migrating to a target architecture. There are also commercial products such as Structure101 Studio, which is frequently used in system and architecture audits. In addition, there are several static code analysis tools that can perform subsets of what ArchUnit perform. These include Degraph and Deptective, among others. Some of these tools can be integrated into build environments and sometimes even into IDEs.

Conclusion

ArchUnit is a small library that can be used to unit test the architecture and internal code quality of both small and large applications. With it, you can quickly, easily, and pragmatically begin to test your code quality objectives. During the build process, ArchUnit tests can ensure that the architecture of a Java application complies with the established rules. In addition to the fluent APIs and well-written Javadoc documentation, the official guide presents the different options ArchUnit offers you for checking your codebase.

Also in This Issue

Jonas Havers

Jonas Havers (@JonasHavers) is a freelance full-stack software engineer and lecturer on software engineering from Germany. He develops web applications predominantly in ecommerce projects with a mix of Java, Kotlin, Groovy, TypeScript, and JavaScript. He is also an advocate for remote work, and he blogs frequently.

Share this Page