Special Edition 3 / 2025 2|2
JAVAPRO

Find More Great Articles Online:
www.javapro.io

CORE JAVA • FRAMEWORKS & APIs • ARCHITECTURE • CLOUD • AI

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

30 YEARS OF JAVA

SPECIAL EDITION

PART 3 2|2

07 JUMP 20 YEARS OF JAVA WITH MIGRATION ENGINEERING

20 CODE REVIEWS WITH AI: A DEVELOPER GUIDE

85 MOVE FAST, BREAK LAWS: AI, OPEN SOURCE AND DEVS

99 PUT EVENTS IN THE DRIVER SEAT TO MANAGE YOUR JAVA ANYWHERE

118 TESTING DONE RIGHT!

131 MASTERING JUNIT: NAVIGATING BETWEEN OLD AND NEW VERSIONS

www.javapro.io

With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors

Gold Sponsors

Silver Sponsors

Bronze Sponsors

Publisher:
JAVAPRO
Im Gewerbepark 29
92637 Erbendorf
Germany

E-Mail: info@javapro.io
Website: http://www.javapro.io

Editor in Chief:
Markus Kett (V.i.S.d.P.)

Editorial:
info@javapro.io

Design, Layout & Print:
Impuls Mediengruppe GmbH
Im Gewerbepark 29
92681 Erbendorf
Germany

Copyright (c) 2025
Impuls Mediengruppe GmbH

All rights reserved.

Java(TM) is a registered trademark of Oracle Corporation.

Javapro is an independent magazine and is not sponsored by Oracle Corporation.

Articles marked with a name do not necessarily reflect the opinion of the editors.

The images featured in this issue are sourced from royalty-free platforms such as Pixabay and Unsplash, contributed by our authors, or creatively generated with the help of artificial intelligence.“

30 Years of Java – Part 3 of an Ongoing Success Story

Three decades of Java prove that lasting success in technology comes from solid evolution, not fancy features. From its origins as an object-oriented language to its role in enterprise platforms and cloud-native development, Java has grown without losing its core promise: stability and portability. This third installment of our anniversary series highlights new directions in design, tooling, architecture, and use-cases practice that are shaping how Java is used today and tomorrow.

Patterns Rediscovered

Familiar concepts are reinterpreted in a modern context. Patterns gain new expression through records and sealed classes, while domain-driven design offers clarity for complex systems. Migration remains a hallmark - carrying applications across decades while adapting them to today's runtime environments and deployment models.

Breaking New Ground in Performance

Efficiency has become as important as scalability. Virtual threads simplify concurrency, while asynchronous I/O and serverless deployments extend Java into new domains. What once seemed limits are now opportunities: proof that the platform is not bound by its past but open to continuous transformation.

Tools and Teams Evolving Together

The developer experience is also shifting. Build tools are being reimagined for speed and usability, while modern testing strategies balance proven frameworks with smarter automation. AI increasingly supports reviews and quality checks, yet these advances also raise cultural questions: how teams collaborate in the future, how trust is built, and how agile practices can help or sometimes even hinder progress.

A Future Built on Renewal

The lesson after thirty years is clear: Java's vitality lies in its ability to renew itself. This third and final instalment in our anniversary series shows a platform that connects tradition with transformation, combining proven stability with the courage to embrace change. Java remains not only a language of the past, but one of the most relevant platforms for the future.

Welcome to more exciting episodes of the Java story!

JCON
GenAI
NOV 04, 2025 in Hamburg
NOV 20, 2025 in Ljubljana
www.genai.jcon.one

Markus Kett
Editor in Chief
JAVAPRO

https://linkedin.com/in/markuskett
https://twitter.com/MarkusKett

07
IDE & TOOLS
Jump 20 Years of Java with Migration Engineering
by Merlin Bögershausen

20
IDE & TOOLS
Code Reviews with AI: A Developer Guide
by Jonathan Vila

35
IDE & TOOLS
Fluent Setter: Breaking the Convention
by Sergei Chernov

49
IDE & TOOLS
Pull Requests != Code Reviews
by Javier López Fernández

56
IDE & TOOLS
JBang, the Awesome Java File Runner
by Yassine Benabbas

65
PROJECT MANAGEMENT
Agile Nightmares – When Agile Methods Become an Innovation Barrier
by Nina Nitzsche

77
PROJECT MANAGEMENT
Journey To Junior
by Andreas Monschau

85
SECURITY
Move Fast, Break Laws: AI, Open Source and Devs (Part 3)
by Steve Poole

91
SECURITY
Move Fast, Break Laws: AI, Open Source and Devs (Part 4)
by Steve Poole

99
DEVOPS
Put Events in the Driver Seat to Manage Your Java Anywhere
by Romain Pelisse

118
TESTING & QUALITY
Testing Done Right!
by Wouter Bauweraerts

131
TESTING & QUALITY
Mastering JUnit: Navigating between Old and New Versions for a Smarter Test Strategy
by Jean Donato

#JAVAPRO #IDE #TOOLS

Jump 20 Years of Java with Migration Engineering

Author:

Merlin Bögershausen is a Software Engineer, Architect and Oracle ACE with over 10 years of experience in different domains and languages. His main focus is Java Enterprise Applications with modern and Next-Generation Java. As a Migration Engineer, he helps teams and individuals utilize new features and supports them in migrating. Besides developing, speaking at conferences and his parental role, he tries to participate in community events, teach folk to land gliders (yes, he is a Flight Instructor!) and play volleyball.

https://www.linkedin.com/in/merlin-boegershausen/

For three decades, the motto in the Java environment has been Write once, run everywhere. Nothing has changed in the content of this slogan; 30-year-old Java programs are still executable today. The way programs are written has changed dramatically over the last 30 years. The language has evolved, the frameworks used have changed, and the style of programming has changed completely. All these changes mean that an update from 20-year-old source code to modern Java has become very, very complex. Often so complex that, despite the obvious

disadvantages, companies shy away from it for several years and are even prepared to accept a huge mountain of technical debt. Paying off this technical debt not only costs money in terms of maintenance, but also poses a risk to the economic success of a company. The greatest danger is not updating the software and foregoing the closing of security gaps and the performance gains from new versions.

In this article, I would like to encourage you and take away your fear of migration. I will show you the Migration Engineering tool, which makes it possible to bring 20-year-old source code into a technically new state. We will look at what migration engineering is and how we can use the tools of migration engineering to analyze the source code and then migrate it. Your projects will have specific requirements, and you will see how the existing recipes can be extended with the appropriate changes. Finally, we will look at what they can do to stay ahead of the migration wave, with always up-to-date source code and always up-to-date use of frameworks. Keep your developers happy and improve the productivity of your software development.

What is Migration Engineering, and Do I Need It?

The subject area of migration engineering deals with “How do I get from one version to another?”. If you compare this with conventional engineering disciplines such as electrical engineering, mechanical engineering, or my favorite subject, aerospace, then migration engineering is really just engineering. In other words, developing a product for the better. This improvement is usually associated with the replacement of certain components. If we take the comparison a little further, it becomes clear that the comparison is a little off. The big difference between an aircraft and our source code is that source code can be changed. An aircraft can only be modified to a limited extent. If we look at a migration of a test framework in our test code, then the complete source code file (see screenshot) that defines this test changes in the difference view.

```
@Test(expected = NoSuchBeanDefinitionException.class)
@Test
public void testLocalSchedulerEnabled() {
        assertFalse(context.getEnvironment().containsProperty("kubernetes_service_host"));
        assertFalse(CloudPlatform.CLOUD_FOUNDRY.isActive(context.getEnvironment()));
        context.getBean(Scheduler.class);
        assertThrows(NoSuchBeanDefinitionException.class, () -> {
                assertFalse(context.getEnvironment().containsProperty("kubernetes_service_host"));
                assertFalse(CloudPlatform.CLOUD_FOUNDRY.isActive(context.getEnvironment()));
                context.getBean(Scheduler.class);
        });
}
```

Diff of a migration from JUnit 4 to JUnit Jupiter

A software engineer claims that this is still the same test. That's like moving the engine of a jumbo from the center of the wing to the ends of the wing and claiming it's the same jumbo. Software development is at this level of madness and complexity every day. We change integral, fundamental components and claim that it is exactly the same software as before. Mastering this complexity is what migration engineering is all about. Migration engineering only works with a high level of test coverage, because otherwise it cannot be guaranteed that the software will really behave in the same way after the migration as it did before.

I would like to briefly describe the work of a migration engineer based on my first weeks at Moderne Inc. A day started with a message in "Spring Boot 3.4 is available, let's upgrade!". Spring Boot version migrations are usually easy to follow through the published migration guide. But the migration guide consists of rough steps. Use the following new class and dependency. A migration engineer must first break this rough step down further until the atomic code changes are known, which a software developer performs manually.

- Add a dependency to the Maven POM
- Add a new import
- Use the new class instead of the old one
- Swap the order of the call parameters
- Delete the old import
- Delete the old dependency

Once these individual steps are in place, they are aggregated and put in the correct order until they result in the target migration. It can now be applied to all existing projects.

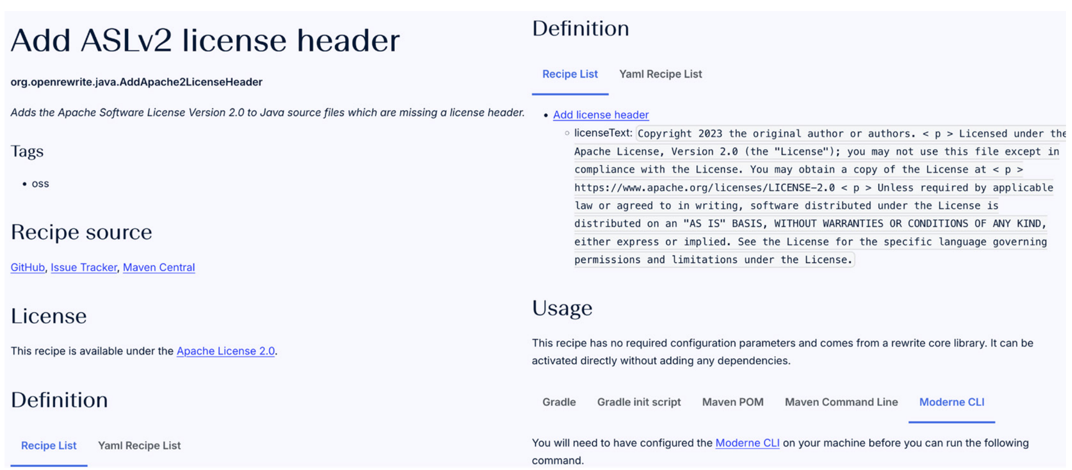
For us, this meant one pull request for each service that we operate. All in all, a lot of work due to the volume of medium-sized pull requests, which were all processed in parallel. And of course, there were queries and change requests. Individual services had to be migrated again. This meant that a migration of the migration had to take place. New recipes had to be created and applied to the existing changes. The components have now been updated, and I would like to give a rough estimate of the effort involved. I started at Moderne Inc. at the beginning of 2025 and spent roughly a week implementing the migration. I applied the migration for about 10 minutes and then worked with different people for about a week to transfer the merge requests into the source code. If this migration had been done manually by different teams, the result would have been divergent applications of the new components and a longer implementation time.

The question remains: does every project now need its own migration engineer who thinks about how the software components can be migrated from one version to the next? - I don't think so, but I do think it is important that every software developer deals with the issue and identifies migration paths for their components. Is your team part of a platform team, and do decisions influence other teams, or is a library provided for further use? Then, it is definitely the team's job to show not only what the changes are but also how they can be implemented in detail. This is the only way your teams can ensure that the styles and versions used are consistent. OpenRewrite provides the technology to formulate and apply changes in a scalable way.

OpenRewrite for Migrating Source Code

Moderne Inc. is responsible for the further development and community of OpenRewrite. This collaboration has resulted in many fundamental building blocks that are combined to create value-adding migrations. These include migrations of major versions for Spring Boot, Quarkus, Hibernate, Maven, and Gradle, as well as small helpers such as applying best practices or fixing the most common issues of static code analysis.

All applicable recipes can be found in the [Recipe Catalog](#) in the OpenRewrite documentation. To find the right recipe for your use case, you can use the search at the top left (box shortcut: Ctrl/CMD+K). This search indexes not only the names, but also parts of the documentation of a migration to enable better-tailored results. The search in the Modern SaaS at https://app.moderne.io delivers even better results; this also delivers suitable results for imprecise queries. Alternatively, the migrations are also tagged according to their subject areas and grouped according to these tags. These groups enable an exploratory search, which is particularly helpful for the first applications.



Screenshot of the Recipe Catalog for a Recipe

A team could thus become aware of the migration to Java 21. The associated [documentation page](#) is generated automatically and has the same structure for all migrations, see also the figure above. The Fully Qualified Name is specified under the heading; this is unique and is required for further use. The tags, a link to the resources and the license information are also provided. There are three main types of licenses:

- **Apache License 2.0,** the core framework is Apache-licensed so that framework authors can offer migrations for their customers. Some do, such as Micronaut and Quarkus. In cases where the framework authors do not provide such migrations, there is a marketplace for third-party migrations, including those from Modern.
- **Moderne Source Available License** are recipes that are freely available but may not be provided as part of another service. In the past, large companies have taken advantage of OpenRewrite to make their commercial AI migration solutions more reliable. To encourage products such as Amazon Q, GitHub Copilot, and IBM Watson to

cooperate, Moderne now publishes its recipes under the MSAL. This means that projects can continue to use the recipes to their full extent, but are prohibited from exploiting them commercially.

- **Moderne Proprietary** are recipes that may only be used with a license from Moderne. These are particularly value-adding recipes that are associated with a large investment by Moderne. These recipes can also be applied to open-source projects.

The various licenses serve to prohibit third parties from using the recipes in their commercial services and marketing them further without giving anything back to the project. A detailed and up-to-date list can be found in the Moderne documentation. At the time of publication, Moderne Inc. and the OpenRewrite community are still looking for a final solution. The goal is still to provide open-source projects with access to the modernization capabilities of OpenRewrite and the Moderne products.

After the legal notes, the documentation lists the migration steps to be carried out. Each step is also a migration with a documentation page of the same structure. A later paragraph deals with how such migrations can be created for your own needs and configured via YAML files. The documentation lists the ways in which this migration can be used. In addition to code adaptations, migrations also collect data for later analysis, the DataTables. Common information includes runtimes, changed files, or analysis information on the use of constructs. The migration should now be applied. After deciding on a migration, the Usage paragraph mentioned above can be used to study the application of a recipe. There are basically three ways to execute a migration.

- **Maven,** via the CLI or the Rewrite Maven plugin
- **Gradle,** rewrite plugin, or an init script
- **Modern CLI,** a standalone power user terminal tool
- **Modern SaaS,** on the OpenRewrite web application

A list of OpenRewrite recipes can be specified via the Maven CLI, which is executed with the Rewrite Maven plugin. This mode is primarily suitable for the use of one-off migrations such as framework updates; the same applies to the Gradle Init Script. Copying the examples from

the documentation for this type of application is recommended. For our JUnit Migrate to Java 21 migration, the Maven CLI call is given in the listing below. The rewrite Maven plugin Goal run is called, and the migration UpgradeToJava21 is specified with rewrite.activeRecipes. As this is not integrated in OpenRewrite, the artifact coordinates for rewrite-migrate-java must be specified under rewrite.recipeArtifactCoordinates.

```
mvn -U org.openrewrite.maven:rewrite-maven-plugin:run \
-Drewrite.recipeArtifactCoordinates=org.openrewrite.recipe:rewrite-
migrate-java:RELEASE \
-Drewrite.activeRecipes=org.openrewrite.java.migrate.UpgradeToJava21
Ende
```

In the case of the Migrate to Java 21 migration, it makes sense to run the recipe again from time to time. This is because, due to carelessness, a developer could use old Java APIs that need to be migrated again. It makes sense to use the recipe as an optional plugin in the build. To do this, the rewrite-mave-plugin is integrated as in the listing below, the UpgradeToJava21 recipe is activated, and the migration artifact is added as a dependency. With a simple mvn rewrite:run, the plugin is executed, performs the configured migration,s and applies the necessary changes to the code. With each run of the rewrite plugins, the plugin reports which migration has made a change in which file. It also creates a rough estimate of how much effort manual editing would have entailed. From here, all that remains to be done is to commit, push, and review.

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.openrewrite.maven</groupId>
        <artifactId>rewrite-maven-plugin</artifactId>
        <version>6.3.1</version>
        <configuration>
          <exportDatatables>true</exportDatatables>
          <activeRecipes>
             <recipe>org.openrewrite.java.migrate.UpgradeToJava21
             </recipe>
          </activeRecipes>
```

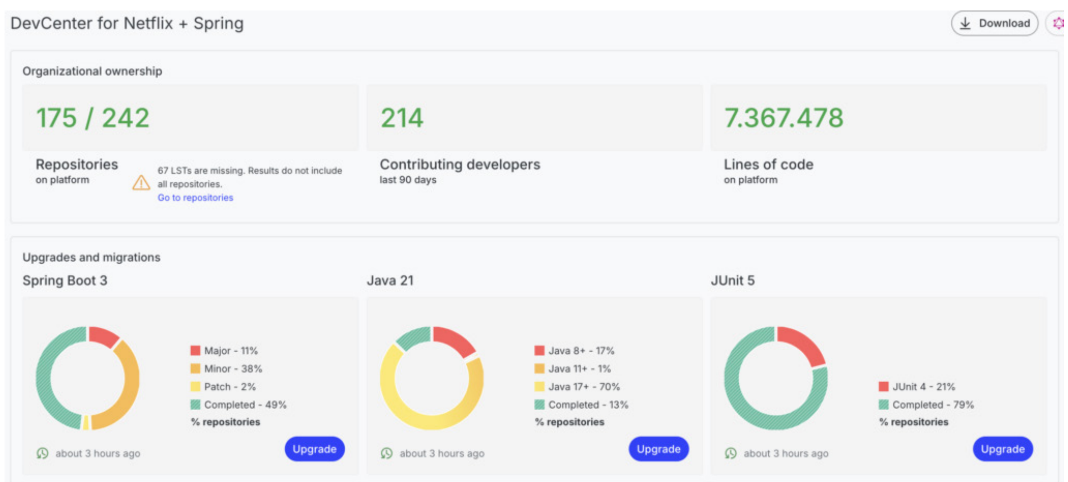
```
        </configuration>
        <dependencies>
          <dependency>
            <groupId>org.openrewrite.recipe</groupId>
            <artifactId>rewrite-migrate-java</artifactId>
            <version>3.4.0</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

The last option is the Modern CLI as a powerful tool based on OpenRewrite. The Modern CLI can be used free of charge for projects in public repositories; licensing is required for closed-source projects. In contrast to the rewrite plugins, the Modern CLI can be used on several projects simultaneously and in parallel. Furthermore, the Modern CLI can persist and reuse the generated metadata. This means that the complete metadata is not generated with every run, and the runtime is drastically improved.

The Modern CLI is installed as a native application, executables exist for all common operating systems, and a pure JAR variant is also provided via Maven Central. After installation, the CLI will be available at the terminal. The metadata must be generated before a migration can be executed. The mod build . command is used for this, which searches for projects in the current directory and generates the metadata. In addition, the list of available migrations must be downloaded with mod sync. To execute a recipe, the mod run. command and the -recipe parameter to start the migration and execute it. After the migration, the changes can be applied or studied. Basically, the process is guided by the CLI in an understandable way. After each run, it reports which next steps would be useful. A step-by-step guide can be found in the documentation.

Each execution of a recipe provides data tables in addition to code adaptations. The datatables are the basis for all analysis options with the OpenRewrite technology. The Maven and Gradle integration produces data tables for a project. In contrast, the CLI and Modern SaaS provide

data for several projects at the same time. In Moderne SaaS, there are some standard visualizations that allow migration engineers to analyze the use of different framework versions or the distribution of changes. The screenshot below shows the DevCenter, which contains information about the repositories for an exemplary organization. In addition, the achievement of strategic business goals such as Spring Boot 3 or Java 21 migrations is shown in a clear manner.



Screenshot of the DevCenter for an OpenSource Organization in Moderne SaaS

It is not uncommon for projects in companies to have special requirements for the use of certain patterns or internal libraries. In this case, the existing OpenRewrite recipes must be extended.

Custom OpenRewrite Recipes

In OpenRewrite technology, migrations are defined as recipes. A recipe instructs the OpenRewrite tool how to adapt the existing code to achieve a goal. Recipes can be defined in 3 different ways with increasing complexity and functionality:

- Declarative YML recipes
- Refaster templates in Java recipes
- Imperative Java recipes

The individual properties of the recipes are listed in the table below. The declarative recipes offer the easiest entry point; these follow a YAML schema and define the list of recipes to be executed with their configurations in addition to the necessary properties. These recipes are used to aggregate other recipes. Refaster recipes use refaster templates

as defined by the Google Error Prone project. These templates are used for type-safe search and replace method calls. If more complex manipulations are required that cannot be mapped by the basic blocks, it is necessary to formulate an imperative recipe. The full scope of Java as a programming language can be used here, and many OpenRewrite utilities are available to implement the migration logic.

Declarative Recipe	**Refaster Recipe**	**Imperativ Recipe**
YAML	Refaster Templates in Java	Pure Java
aggregate Recipes	Search & Replace	high flexible
configure Recipes	Typsafe	Visitor-Pattern
simple	Limited capabilities	multiple Utilities

Overview of different types of recipes

It is advisable to start with a declarative recipe and try to map as much of the migration as possible by configuring existing recipes. However, before a recipe can be created, a corresponding project must be set up, and the acceptance tests must be defined.

Declarative recipes are used to configure existing recipes and execute them together. In the first step, the required recipe is identified in the Open Rewrite recipe catalog, and the fully qualified name is identified. The fully qualified name is specified directly under the heading of the documentation. If recipes from your own database are to be used, the qualifying name is also used in this case. The org.openrewrite.java.ChangeType recipe can be used to replace a class reference, for example. In the moderneinc/rewrite-recipe-starter GitHub project, this recipe is configured in resources/META-INF/rewrite/stringutils.yml, see also the listing below, in order to use the correct StringUtils class.

```
type: specs.openrewrite.org/v1beta/recipe
name: com.yourorg.UseApacheStringUtils
displayName: Use Apache `StringUtils`
description: Replace Spring string utilities with Apache string
utilities.
recipeList:
```

```
- org.openrewrite.java.ChangeType:
  oldFullyQualifiedTypeName: org.springframework.util.StringUtils
  newFullyQualifiedTypeName: org.apache.commons.lang3.StringUtils
```

The type identifies it as a recipe for OpenRewrite migrations and can be addressed via the qualifying name after the name. The displayName is mainly used for the automatically created recipe catalog. The recipes to be executed are specified in the recipeList. The possible configurations are specified on the corresponding recipe pages. If the existing basic recipes in the recipe catalog are not sufficient, new recipes must be written. One possible example would be the simplification of a ternary operator to a constant expression. The use of Refaster templates from the Error Prone project is suitable here.

Error Prone Refaster can be used to make typical pattern-based changes. OpenRewrite supports the templates of Refaster and thus offers a suitable abstraction for the adaptation of method calls. A Refaster template recipe is marked as a recipe by the @RecipeDesciptor, given a name for the documentation and a description. The code passage to be replaced is specified in the @BeforeTemplate marked method. The expressions are used in the method in a type-safe manner. The method marked with @AfterTemplate defines the target state and uses type safety to create the new expression. Refaster templates can only make changes within a code block.

```
@RecipeDescriptor(
  name = "Replace `booleanExpression ? true : false` with
  `booleanExpression`",
  description = "Replace ternary expressions like `booleanExpression ⏎
  ?  true :      false` with `booleanExpression`.")
public static class SimplifyTernaryTrueFalse {
    @BeforeTemplate
    boolean before(boolean expr) {
        return expr ? true : false;
    }
    @AfterTemplate
    boolean after(boolean expr) {
        return expr;
    }
}
```

If further adjustments are required and these are not possible by combining existing recipes, an imperative recipe can be created. These imperative recipes extend the abstract class org.openrewrite.Recipe, see listing below.

```
public class TestRecipe extends org.openrewrite.Recipe {
  @Override
  public String getDisplayName() {
    return "An example Recipe";
  }
  @Override
  public String getDescription() {
    return "This Recipe is an example.";
  }
  @Override
  public TreeVisitor<?, ExecutionContext> getVisitor() {
    return new JavaIsoVisitor<ExecutionContext>() {
      @Override
      public J.MethodDeclaration visitMethodDeclaration(J.
      MethodDeclaration m, ExecutionContext ctx) {
        J.MethodDeclaration m2 = super.visitMethodDeclaration(m, ctx);
        return m2.getName().getSimpleName().equals("foo") ?
               m2.withName(m2.getName().withName("bar")) :
               m2;
      }
    };
  }
}
```

The getDisplayName method returns the display name, and getDescription the description for the automatically generated documentation. The getVisitor method returns an instance of a TreeVisitor. A TreeVisitor traverses the Lossless Semantic Tree (LST) and manipulates individual nodes in this tree. The LST is the OpenRewrite representation of the complete source code within the project, spanning languages and technologies. The LST is created when Open Rewrite is started and is used after all recipes have been executed to convert the changes back into source code. There is a separate LST implementation and customized visitors for each supported language. To manipulate Java source code

in the example, the org.openrewrite.java.JavaIsoVisitor is used. This contains a visit method for each type of element in the LST. These return the changed element. In this example, all method declarations are visited, and the J.MethodDeclaration, which has the name "foo", is searched for in order to rename it to "bar". If the currently considered method is to be retained, it is returned subverted; if null is returned, the method definition is deleted. Further complete implementations of recipes can be found in moderneinc/rewrite-recipe-starter. Among others in the class NoGuavaListsNewArrayList, this recipe migrates the use of Guava to JDK contained methods for handling ArrayLists. A detailed step-by-step workshop for the home office is available on Moderne.io and covers other important tools in addition to the techniques mentioned here.

Stay in Front of the Migration Wave

In the coming months and years, we developers will continue to be bombarded by migrations. With OpenRewrite technology and the migration engineering approach described here, organizations can put themselves in a position to act in a planned manner again instead of just continuing to react. The OpenRewrite step-by-step guides on upgrading from Java 5 to Java 21 or migrating from Java EE to Jakarta EE 10.0 offer a good starting point here. By using these two guides, teams can learn how much time can be saved when migrating 20-year-old applications. The knowledge learned can then be applied to internal migrations and libraries to keep the organization fit for another 30 years of Java.

> Back to Table of Content

#JAVAPRO #IDE #TOOLS

Code Reviews with AI: A Developer Guide

Author:

Jonathan Vila López is a Java Champion, Organiser at BarcelonaJUG and cofounder of JBCNConf and DevBcn conferences in Barcelona. Currently working as a Developer Advocate in Java at Sonar (SonarLint,SonarQube), focused on Clean Code & Security. I have worked as a (paid) developer since the first release of The Secret of Monkey Island, about 30 years ago using Go on Kubernetes for a Service Mesh layer on top of Istio | Java on Kubernetes for K8s Operator, Rest API, using Quarkus, GraalVM, Apache Camel | PHP | VB | Python | Pascal | C I am very interested in simulated reality, psychology, philosophy, and Java

https://www.linkedin.com/in/jonathanvila/

Code reviews are a cornerstone of software development. They're where we share knowledge, catch bugs early, and ensure our code meets the highest standards.

But let's be honest...

Traditional code reviews can be time-consuming and tedious and sometimes even miss subtle yet critical issues. Enter the age of AI-powered code review, a game-changer that addresses these challenges and elevates code quality to new heights.

This article dives into the common pitfalls of code reviews and explores how AI tools can revolutionize each phase of the development lifecycle.

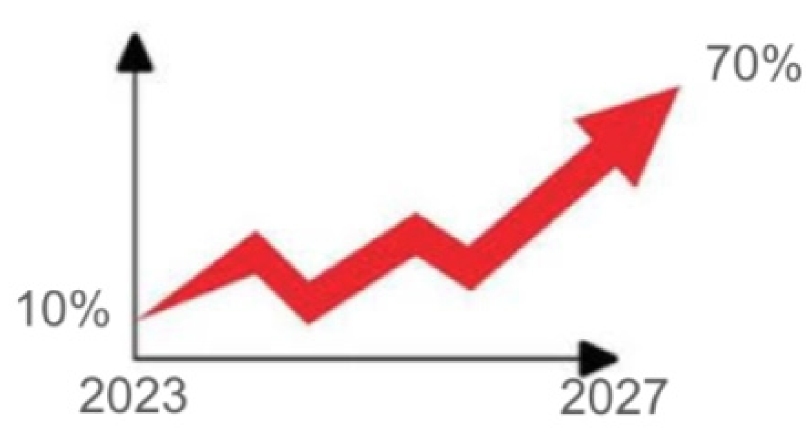
I will discuss the use and impact of AI on the different phases of the SDLC from the 2 main perspectives of the developer and the reviewer. I will also talk specifically about the use of AI in the Code Review and how to implement it productively. I will provide examples of tools used in the different phases: Github Copilot, SonarQube, Qodo, and IntelliJ.

Code Generated by AI Code Assistants

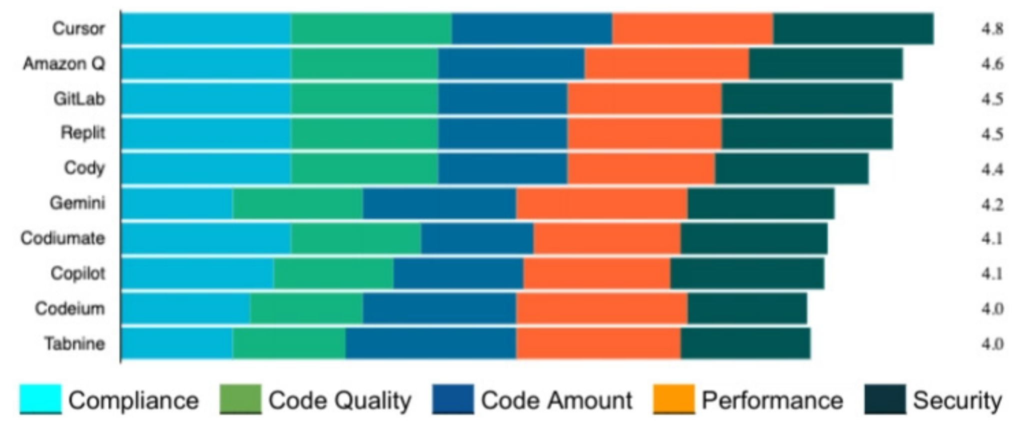
AI-powered generative code assistants take the power of AI even further by automatically generating code based on your inputs. This can dramatically reduce the time and effort required to write code, especially for repetitive or boilerplate tasks.

~55%

Generative code assistants can also help you explore different design options and identify potential problems before you start coding. By leveraging these tools, you can focus on the creative and strategic aspects of software development, while the AI handles the tedious and mechanical tasks.



There's a long list of AI code assistants providing different features, with different ranking rates considering 5 different categorizations:



https://research.aimultiple.com/ai-coding-benchmark

While these tools are powerful and feature-rich, they rely on models hosted somewhere and there is a price involved in some of the features.

The local free open-source approach ....

Other completely open-source options are also available. This option involves hosting the model to generate code locally or in your network. There are tons of free and open-source models that you can use, and you can only serve those models by installing the free tool Ollama on a machine in your network or locally.

```
➜  ~ ollama list
NAME                        ID              SIZE      MODIFIED
qwen2.5-coder:1.5b-base     02e0f2817a89    986 MB    16 hours ago
tinydolphin:latest          0f9dd11f824c    636 MB    2 months ago
tinyllama:latest            2644915ede35    637 MB    2 months ago
llama3:latest               365c0bd3c000    4.7 GB    5 months ago
codellama:latest            8fdf8f752f6e    3.8 GB    5 months ago
llama3.1:latest             91ab477bec9d    4.7 GB    5 months ago
llama2:latest               78e26419b446    3.8 GB    5 months ago
➜  ~ ollama pull deepseek-coder
pulling manifest
pulling d040cc185215... 100%
pulling a3a0e9449cb6... 100%
pulling 8893e08fa9f9... 100%
pulling 8972a96b8ff1... 100%
pulling d55c9eb1669a... 100%
verifying sha256 digest
writing manifest
removing any unused layers
success
```

```
import io.quarkus.test.junit.QuarkusTest;
import org.mockito.*;
import static org.mockito.Mockito.*;
@QuarkusTest
public class CuriousChatResourceTest {
    @InjectMocks private MockedCuriousChatRepository repository = new MockedCurio


      /* Using JUnit5 */
       import org.junit.jupiter.api.BeforeEach;
        public class CuriousTest {
           @Autowired private TestRestTemplate restTemplate ;// this is for testi
            List<CuriosityQuestion> questions = new ArrayList<>(); // pre-defined
           // your test case for asserting if rest call was successful and the re
           // your test case for asserting if rest call was successful and the re
```

Ask a follow-up

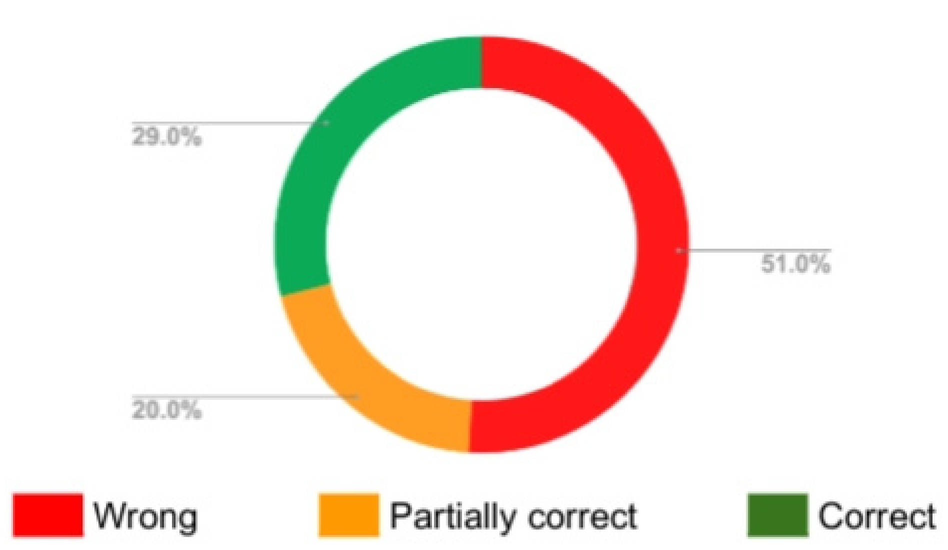
DeepSeek Coder @ ⌘⏎ @codebase ⏎ Enter

I've tried with the IntelliJ plugin "Continue", Ollama, and the models "codellama" and "deepseek-coder" and the experience was not bad at all. With this solution also you are sure your code, and your prompts are not going anywhere out of your domains.

But, every magic comes with a price.

While generative AI holds immense promise, it is not without its pitfalls. One major concern is the potential for introducing bugs and vulnerabilities into code. AI models are trained on vast amounts of data, and if this data contains errors or malicious code, the generated code may inherit these flaws.

AI-generated code correctness



Additionally, generative AI systems may not fully understand the context or intent of the code they generate, leading to nonsensical or even harmful output. Furthermore, there is a risk that generative AI could not use the full code base context in order to generate the most aligned code with our current content. It is crucial for developers to carefully review and test code generated by AI and to employ robust security measures to mitigate these risks.

With this panorama, it's clear that using AI to generate code will impact positively the speed, but also negatively in the full SDLC and more importantly in the code review process.

Let's focus on the traditional code review process and its pain points.

The Traditional Code Review Struggle: Familiar Pain Points

We've all been there. Traditional code reviews, while valuable, often suffer from:

- **Time Consumption:** Manually reviewing every line of code is a significant time investment, especially for large projects.
- **Subjectivity:** "Good code" can be subjective, leading to inconsistencies in feedback and potential disagreements.
- **Missed Issues:** Even the most experienced human reviewers can miss subtle bugs, security vulnerabilities, or performance bottlenecks. We've seen from above how code assistants can impact here.
- **Focus on Style:** Too much emphasis on minor stylistic issues can distract from more critical problems.
- **Lack of Context:** Reviewers may lack the full context of the code changes, making it harder to provide effective feedback.
- **High Cognitive Load:** Reviewing large pull requests with hundreds of lines of code can overwhelm even the most experienced developers.
- **Delayed Feedback:** Waiting for a code review can slow the development pipeline, impacting delivery timelines.

- **Team friction:** Code reviews can lead to team friction when simple issues are overlooked due to a lack of context, when subjective feedback occurs, or when poor feature testing occurs, potentially escalating into disagreements.

AI to the Rescue: Enhancing Code Reviews

AI-powered tools are transforming code reviews by automating tedious tasks, providing objective feedback, and uncovering hidden issues. Let's explore how these tools can assist throughout the development lifecycle:

1. Development Phase (IDE Integration)

We've seen that several code assistant plugins and IDEs can help us generate code. Several benchmarks (huggingface, stackeval, Mike Ravkine's) could help us choose the model to use in those assistants.

So, we have the proper tools for auto-completion and code generation, but what about verifying that the code generated does not introduce issues, vulnerabilities, or solutions that are not well suited for the language version of the code base context?

For this task, there are static analyzers in the form of IDE linters, like SonarQube for IDE, that will provide instant feedback as you write code, or CI/CD code analyzers, like SonarQube Server/Cloud, which will do a full analysis of your code and prevent or allow it to be merged.

Imagine catching potential bugs and best practices before they even make it to a code review.

These linters use static analysis, and different functions, to detect code smells, bugs, and security vulnerabilities directly in your IDE, empowering you to write cleaner and safer code from
the start.

However, this includes not only bugs and vulnerabilities but also **best practices for using certain frameworks or language versions**. AI code assistants sometimes are not well aware of the language version that you are using (e.g., Java 21) in your codebase, and the solutions they suggest do not consider the latest improvements in the language, just simply the most used approaches.

In this case, GitHub Copilot didn't suggest an approach using a feature introduced 7 years ago.

jonathanvila

generate a method that iterates a collection, doing the sum of its integer value, and calculating the average

Generated by Github Copilot

```
public double calculateAverage(Collection<Integer> collection) {
   int sum = 0;
   for (Integer num : collection) {
       sum += num;
   }
   return (double) sum / collection.size();
}
```

Manual approach considering Java's new Teeing collector, introduced in Java 12, using the consisted and language level approach to iterate collections and lazily compute values from it.

```
public double calculateAverageManual(Collection<Integer> collection)
{
   return collection.stream().collect(
Collectors.teeing(
             Collectors.summingDouble(i -> i),
             Collectors.counting(),
             (sum, count) -> sum / count)
```

```
);
}
```

or even not using the latest new features of a language. In this case, Virtual Threads were introduced in Java 21, a year and a half ago.

/generate method with multithreading accessing http://localhost:4000 and parsing the response finding a 200 http code

Code generated by Github Copilot, using platform threads

```
new Thread(() -> {
    var url = new URI("http://localhost:4000").toURL();
    var connection = (HttpURLConnection) url.openConnection();
    connection.setRequestMethod("GET");
    int responseCode = connection.getResponseCode();
}).start();
```

Manual approach using Virtual Threads, being able to create thousands of threads that will [increase the performance dramatically in blocking operations.]()

```
Thread.ofVirtual().start(() -> {
    var url = new URI("http://localhost:4000").toURL();
    var connection = (HttpURLConnection) url.openConnection();
    connection.setRequestMethod("GET");
    int responseCode = connection.getResponseCode();
}).start();
```

Luckily these linters will also warn us about the lack of best practices usage while we code and during the CI full analysis.

```
public void test() {  new *
        new 1 Thread(() -> {
            try {
                var url = new URI( str: "http://localhost:4000").toURL();
                var connection = (HttpURLConnection) url.openConnection();
                connection.setRequestMethod("GET");
                int responseCode = connection.getResponseCode();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }).start();
}
```

SonarQube: Use virtual threads for heavy blocking operations.
SonarQube: Show issue locations   More actions...
java.net.HttpURLConnection
public int getResponseCode()
throws java.io.IOException
Gets the status code from an HTTP response message. For

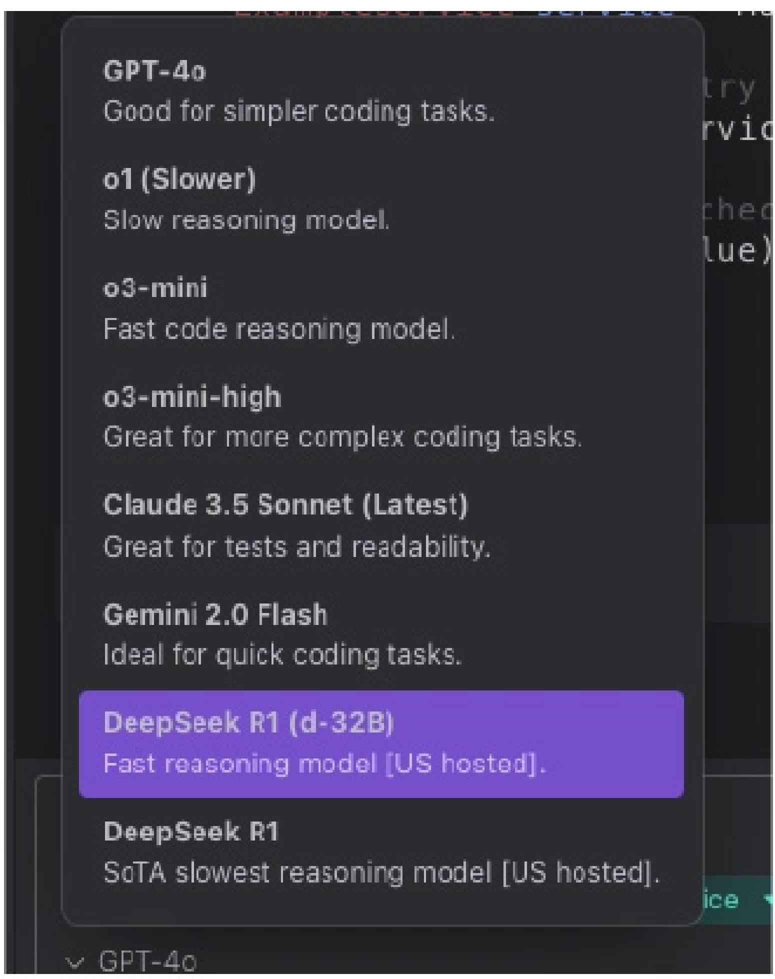
A particular benefit of some linters over others (like SonarQube IDE) is that they can analyze multiple types of files at the same time in the same project. This is not only restricted to programming languages like Java, Python, JScript, Kotlin, etc. but also to Cloud deployment files like Docker, Kubernetes, Ansible, Terraform, CloudFormation, etc., and even Secrets vulnerabilities.

2. Test Generation Phase

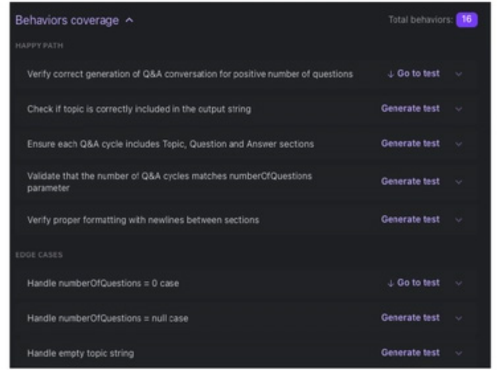
AI can analyze current code, try to understand its purpose, and generate test methods that will potentially generate the test code, ensuring high coverage. This helps to ensure that your code is thoroughly tested before it is released.

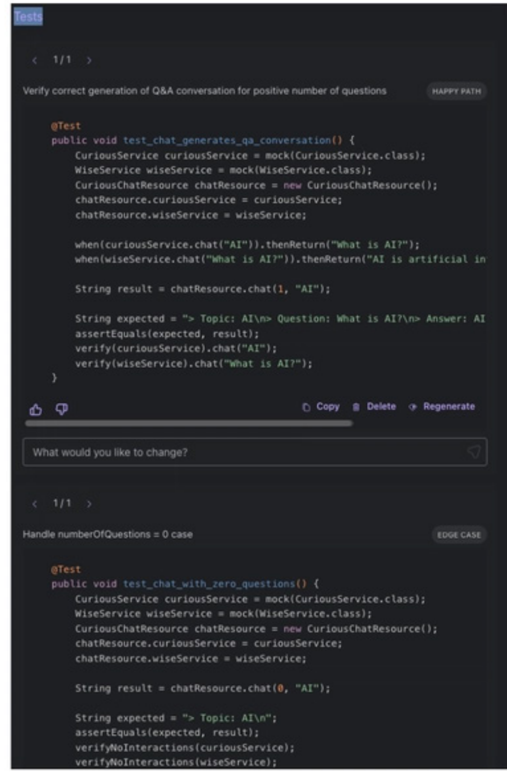
In this area, we can find tools like Qodo Gen, among others, that specialize in test generation.

I've installed it in my IntelliJ IDE and tried it with my AI project. The result is impressive, considering several test use cases in the happy path or edge cases. As with most code assistants, we can select which remote-hosted model we want to use.



Tools like Qodo will take a class method and create its tests. We will have a dashboard to see the tests and executions, and also a plan for the test generation in the Qodo plugin :





3. Pull Request Creation

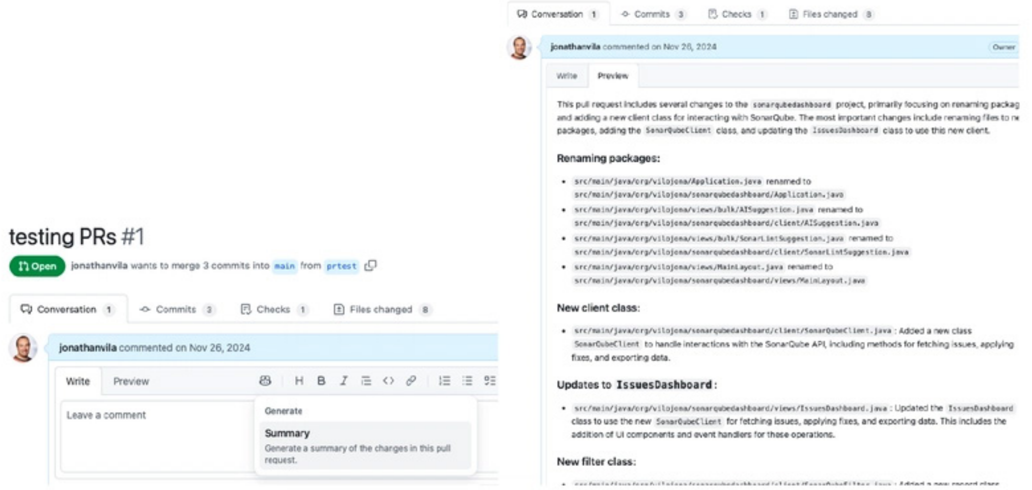
The process of creating a Pull Request is also important in order to give the proper context and details to those who will review it.

A typical workflow would usually imply:

- Follow the initial process of joining/sign-in a team
- Read the contribution guidelines
- Do the commits following a convention
- Sign all the commits (please!)
- Create a draft pull request with a good and complete description (sometimes following a template e.g. JKube project)
- Wait for all the checks to pass
- Change the PR status to ready to review.

Several guides (GitHub, pull request) can help you write good descriptions,

but we can leverage AI for this. One of the tools we can use for this is Github Copilot, which analyzes the code in the PR to provide a more detailed description.



In these two images, we see how we can ask Github Copilot to generate a summary for the PR.

We can also expand this with all the details we think can add more context and value and help reviewers in their tasks.

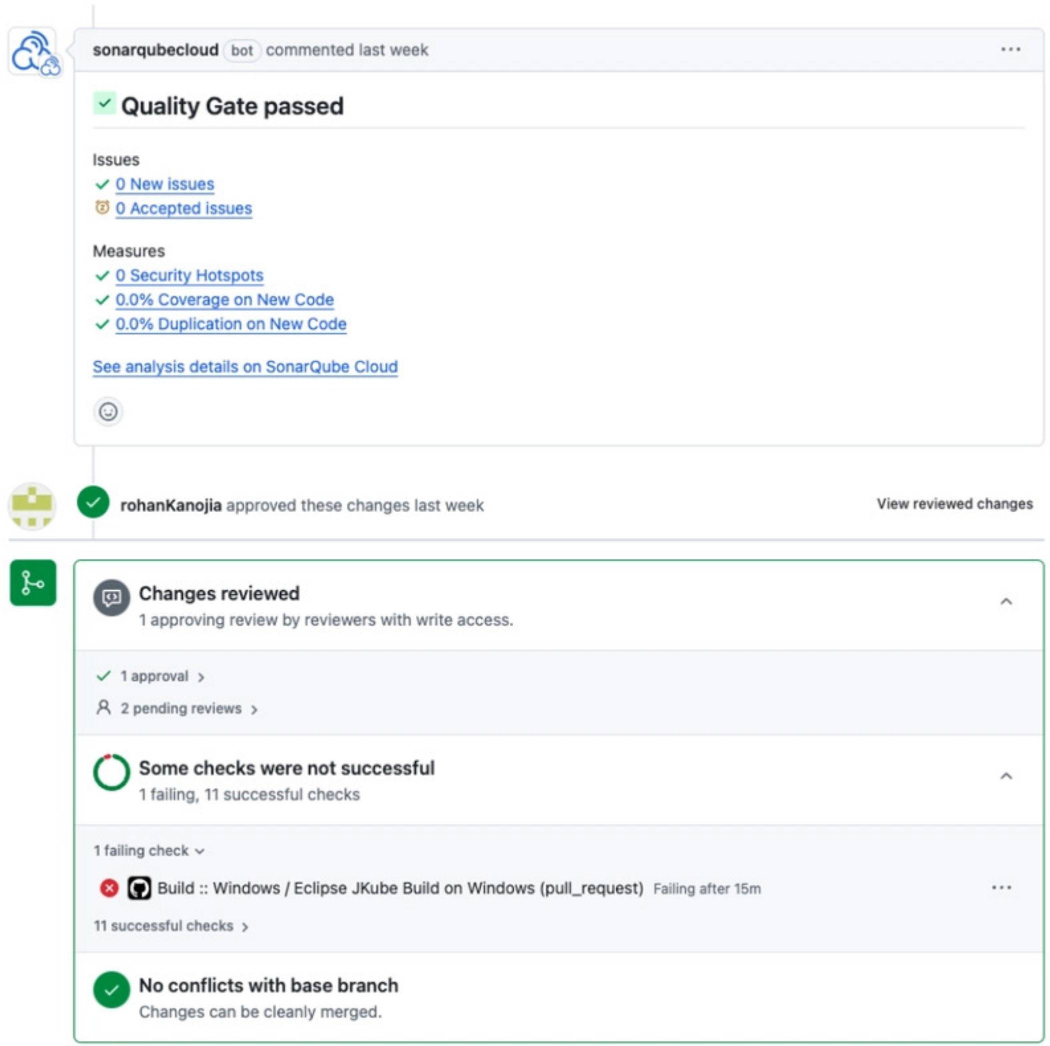
4. Pre-Code Review (Automated Analysis)

Static analyzers (like SonarQube) perform in-depth static analysis on your codebase, identifying issues that might be missed by human reviewers. This goes beyond style checks and delves into:

- **Bug Detection:** Identifying potential NullPointerExceptions, logic errors, and other bugs.
- **Security Vulnerabilities:** Detecting potential injection attacks, cross-site scripting (XSS) vulnerabilities, and other security risks.
- **Code Smells:** Highlighting code that is difficult to read, maintain, or understand. For example, overly complex methods or duplicated code.
- **Code Coverage:** Measuring the percentage of code covered by unit tests, helping to ensure comprehensive testing.

These analyzers present these issues clearly and actionably, prioritizing them based on severity.

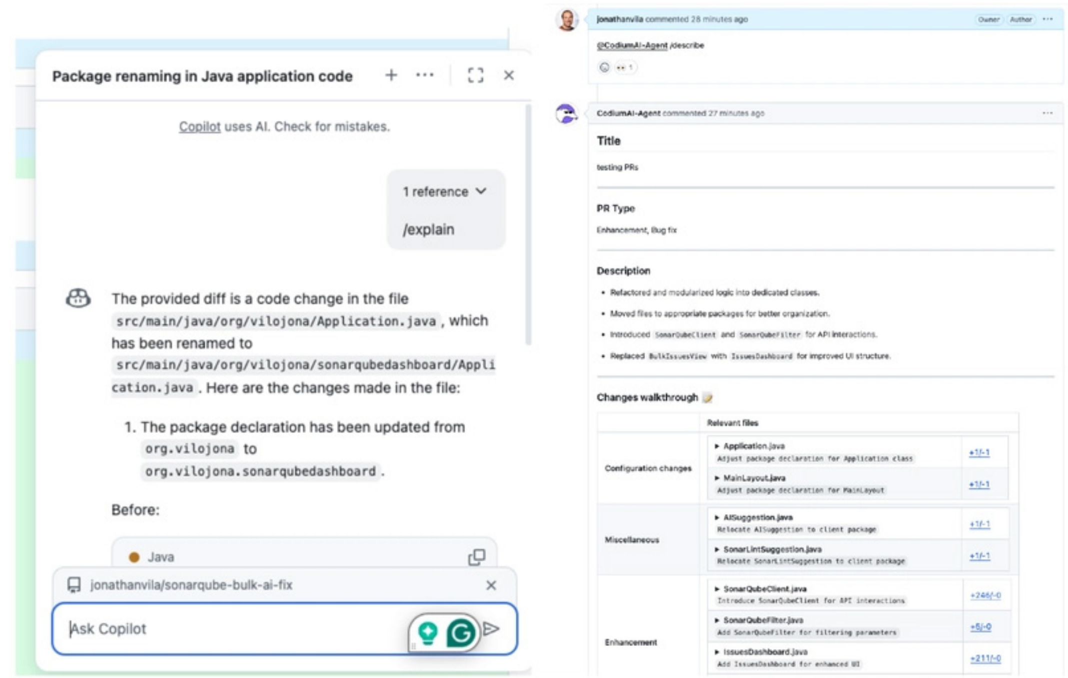
This allows developers to focus on the most critical problems first, making the review process more efficient and effective.
Connecting this step with the previous PR workflow, the tools we connect to our repository can help us to check for all the scenarios that can make our code fail, before anyone invests time in reviewing it, to just focus on working changes that need experienced review.



5. Pull Request Changes Explanation

Now it's the turn of the reviewers. They should start by reading the ticket that defines the PR's goal. After that, a careful review of the checks' status will give us an idea of whether the changes are okay to be merged.

For this, we can use several tools. I've tried Github Copilot and Qodo PR-Agent (you can find a comparison here) :



While Copilot has an explanation feature per changed file, Qodo can create a description for the entire PR. This will help the reviewers understand the details applied to files and focus on those that require more attention. It's important to reduce the time a PR needs to be merged, and definitely, the usage of AI tools can help us with that.

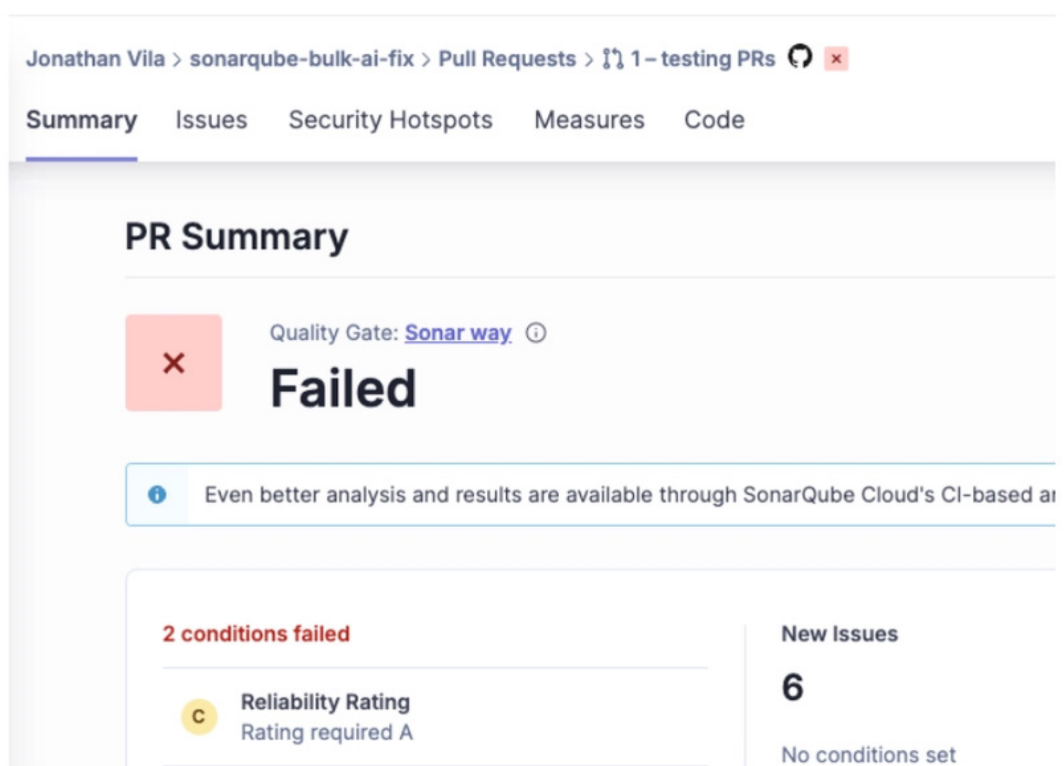
6. Changes Suggestion

In some PRs, reviewers make suggestions to improve or fix parts of the document. This is a crucial point that, if not done correctly, can add anxiety and friction among team members.

There are some guidelines in order to have safe and productive communication between the author and reviewers.

Some AI tools, like Qodo PR-Agent, can also implement the improvements and changes suggested by the AI agent directly from the PR review to the code.

Like all the changes, they need to be analyzed and checked with tools like SonarQube. If there are any issues, these tools will fail, preventing those changes from being merged into the main branch.



Addressing the Challenges with AI

Here's how AI tackles the traditional code review challenges:

- **Reduced Time:** Automation frees up developers to focus on more complex and creative tasks. Also helps reduce the cognitive load for the review process needed to understand the scope of all the changes.
- **Increased Objectivity**: Static analysis provides objective, consistent, and deterministic feedback based on predefined rules and best practices.
- **Focus on Critical Issues**: Prioritization helps reviewers focus on the most important problems.
- **Enhanced Context:** AI can read the changes and generate meaningful PR descriptions along with detailed explanations of the changes for the reviewers.

Conclusion

AI is not replacing human reviewers; it's empowering them. By automating tedious tasks and providing valuable insights, AI tools will improve the speed and comprehension of the generated code.

Tools like static analyzers will automatically check code compliance and guarantee code quality throughout the SDLC, allowing developers to focus on what they do best: designing, building, and innovating.

By leveraging AI-powered tools, we can shift our focus from simply finding bugs to building **high-quality, maintainable, and secure software.**

> Back to Table of Content

#JAVAPRO #IDE #TOOLS

Fluent Setter: Breaking the Convention

Author:

Sergei Chernov is a seasoned java developer, last years mostly focused on developer productivity engineering. Occasional open-source contributor, author and speaker.

https://www.linkedin.com/in/schernov/

What if we implement a fluent Interface, whereby the setter method returns **this** instead of **void**?

```
public class SimplePojo {
    private String value;

    public String getValue() {
        return value;
    }

    public SimplePojo setValue(String value) {
        this.value = value;
        return this;
    }
```

```
    // equals, hashCode, toString
}
```

Now, let's rewrite a standard piece of code using a fluent interface:

```
private static AssignmentGroupedActivitiesResource create() {
    return new AssignmentGroupedActivitiesResource()
        .setGrouping(new UserActivitiesGroupingResource()
            .setAlignmentScore(1)
            .setFocusScore(0)
            .setAdvancedGroups(List.of(
                new ProductivityGroupResource()
                    .setSectionName("Development")
                    .setColor("#2196f3")
                    .setSpentTime(5L),
                new ProductivityGroupResource()
                    .setSectionName("Chat")
                    .setColor("#E502FA")
                    .setSpentTime(1L)
            ))
            .setPeriodLong(10L)
            .setTotalTrackedTime(7L)
            .setIntensityScore(2));
}
```

Here's the same code, without a fluent interface:

```
private static AssignmentGroupedActivitiesResource create() {
    ProductivityGroupResource group1 = new
    ProductivityGroupResource();
    group1.setSectionName("Development");
    group1.setColor("#2196f3");
    group1.setSpentTime(5L);
    ProductivityGroupResource group2 = new
    ProductivityGroupResource();
    group2.setSectionName("Chat");
    group2.setColor("#E502FA");
    group2.setSpentTime(1L);
    UserActivitiesGroupingResource grouping = new
    UserActivitiesGroupingResource();
```

```
        grouping.setAlignmentScore(1);
        grouping.setFocusScore(0);
        grouping.setAdvancedGroups(List.of(group1, group2));
        grouping.setPeriodLong(10L);
        grouping.setTotalTrackedTime(7L);
        grouping.setIntensityScore(2);
        AssignmentGroupedActivitiesResource assignmentGroupedActivities =
        new AssignmentGroupedActivitiesResource();
        assignmentGroupedActivities.setGrouping(grouping);
        return assignmentGroupedActivities;
    }
```

By implementing a fluent interface, we use less code, we don't declare or refer to local variables, we require lower cognitive load, and we don't change the meaning. The hierarchy in the code reflects the nesting of the objects, of the fields to fill out. You can also notice the nesting visually. For example, let's consider a JSON-serializable object, a standard data exchange format for REST APIs:

```
private static AssignmentGroupedActivitiesResource create() {
    return new AssignmentGroupedActivitiesResource()
            .setGrouping(new UserActivitiesGroupingResource()
                    .setAlignmentScore(1)
                    .setFocusScore(0)
                    .setAdvancedGroups(Arrays.asList(
                            new ProductivityGroupResource()
                                    .setSectionName("Development")
                                    .setColor("#2196f3")
                                    .setSpentTime(5L),
                            new ProductivityGroupResource()
                                    .setSectionName("Chat")
                                    .setColor("#E502FA")
                                    .setSpentTime(1L)
                    ))
                    .setPeriodLong(10L)
                    .setTotalTrackedTime(7L)
                    .setIntensityScore(2));
}
```

```
{
  "grouping" : {
    "alignmentScore" : 1,
    "focusScore" : 0,
    "advancedGroups" : [ {
      "sectionName" : "Development",
      "color" : "#2196f3",
      "spentTime" : 5
    }, {
      "sectionName" : "Chat",
      "color" : "#E502FA",
      "spentTime" : 1
    } ],
    "periodLong" : 10,
    "totalTrackedTime" : 7,
    "intensityScore" : 2
  }
}
```

Why Not Use Lombok Builder?

Programmers familiar with Lombok may prefer to use Builder, instead. This option achieves the same result, and the code looks pretty much the same, but with more **builder()** and **build()** calls:

```
private static AssignmentGroupedActivitiesResource create() {
    return AssignmentGroupedActivitiesResource.builder()
        .grouping(UserActivitiesGroupingResource.builder()
            .alignmentScore(1)
```

```
            .focusScore(0)
            .advancedGroups(List.of(
                ProductivityGroupResource.builder()
                    .sectionName("Development")
                    .color("#2196f3")
                    .spentTime(5L)
                    .build(),
                ProductivityGroupResource.builder()
                    .sectionName("Chat")
                    .color("#E502FA")
                    .spentTime(1L)
                    .build()
            ))
            .periodLong(10L)
            .totalTrackedTime(7L)
            .intensityScore(2)
            .build())
        .build();
}
```

In general, Builder works well with immutable objects, as an alternative to passing a large set of parameters to the constructor. However, it's not an ideal option for mutable POJOs. Moreover, Builder **limits compatibility with a number of frameworks and libraries**.

What Frameworks Support a Fluent Setter?

Several modern frameworks support such access methods. These are the ones I know of and work with.

Lombok

It is very easy to convert the **@Data** class to use this approach.

For a class, add the **@Accessors(chain = true)** annotation:

```
import lombok.Data;
import lombok.experimental.Accessors;
```

```
@Data
@Accessors(chain = true)
public class DataPojo {
    private String value;
}
```

For package / module / project, add the following parameter to **lombok.config**:

```
lombok.accessors.chain=true
```

Fluent vs Chain in Lombok

It's important to distinguish between fluent and chain in Lombok. For example:

```
lombok.accessors.fluent=true
```

produces almost the same result as **@Accessors (fluent = true)**. The difference is that the generated setter method isn't prefixed with **set**:

```
// fluent mode of setter - no "set" prefix
public SimplePojo value(String value) {
    this.value = value;
    return this;
}
```

IDEA (Lombok Plugin)

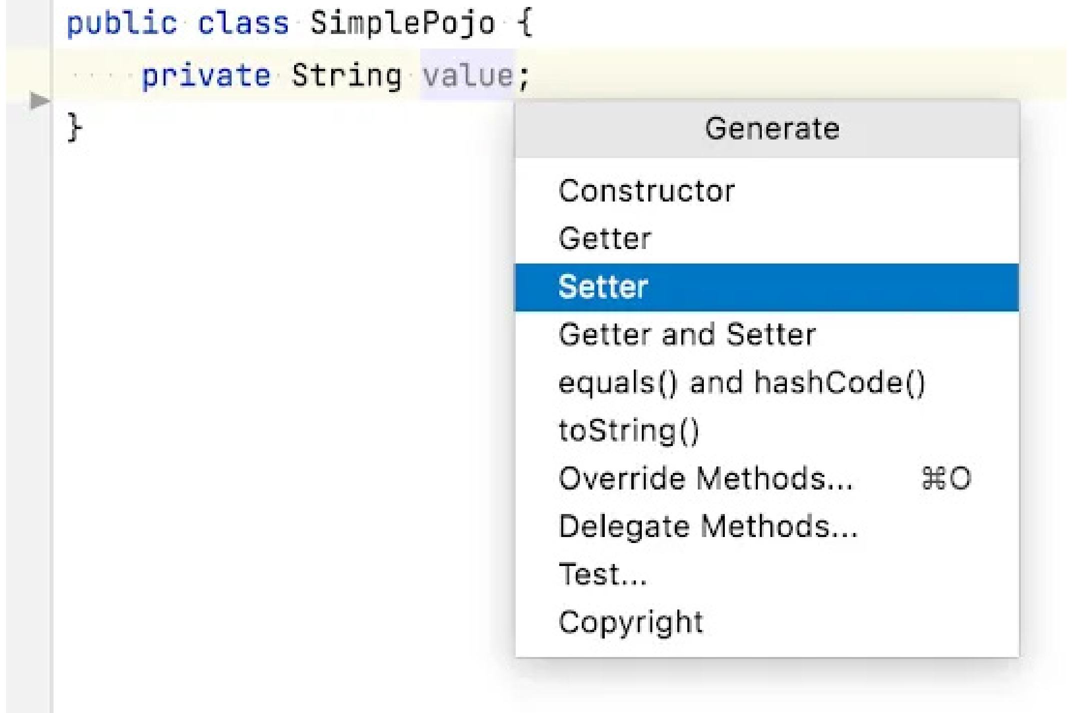
From IntelliJ version 2020.3 the Lombok plugin is [shipped with IntelliJ IDEA by default](). The plugin perfectly recognizes such configurations, both through annotations and configs, and code analysis works correctly:

```
package com.example.settersdemo.model;

import lombok.Data;
import lombok.experimenta

@Data
@Accessors(chain = true)
public class DataPojo {
    private String value;
}
```

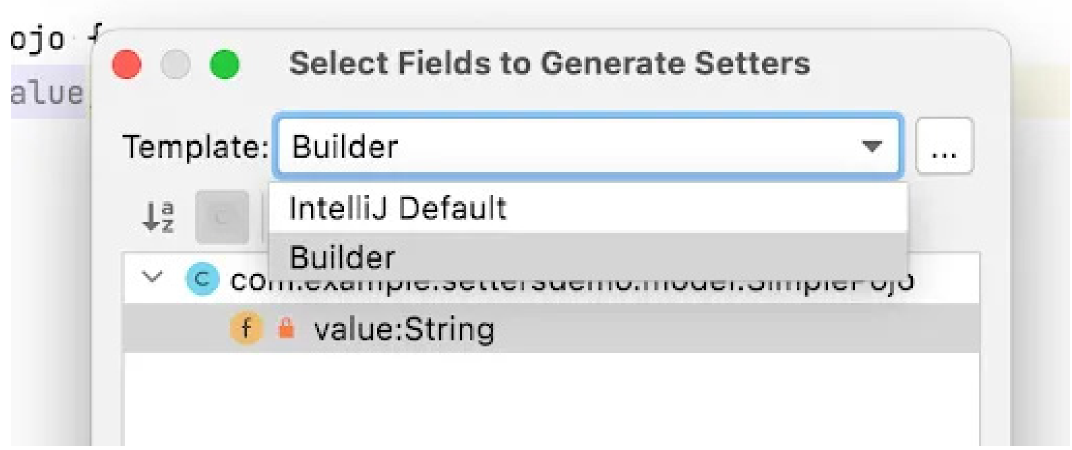
Inherited members (⌘F12) Anonymous Classes (⌘
DataPojo
DataPojo()
canEqual(Object): boolean
equals(Object): boolean ↑Object
getValue(): String
hashCode(): int ↑Object
setValue(String): DataPojo
toString(): String ↑Object
value: String

IDEA (Code Generation)

IntelliJ IDEA can generate access methods for class fields, including fluent. To do this, set a new field, and then from the context menu select *Code > Generate > Setter.*



From *Template* select *Builder:*



This generates an access method:

```
public class SimplePojo {
    private String value;
    public SimplePojo setValue(String value) {
        this.value = value;
        return this;
    }
}
```

Note that this sets the Builder pattern as the default template. To revert to the classic option, from *Template* select *IntelliJ Default*.

Jackson

Jackson, a library for serializing JSON, XML and more, doesn't require additional configuration to work with beans whose setters return this. Serialization and deserialization work without issues.

jOOQ

jOOQ, a framework that generates model classes from a database schema, can generate classes with fluent setters. Since it is generation, not analysis, it requires an explicit setting of the configuration parameter.

Hibernate

Hibernate, a database access framework, allows using classes with the proposed notation like Entity. No additional configuration required.

Spring and Spring Boot

Spring and Spring Boot (as well as derived projects like Spring Data and Spring Data REST) contain lots of logic based on "reflection" and analysis of class structures. Here are the configuration classes, the parsing of **ResultSets** through **BeanPropertyRowMapper**, and so on. **BeanUtils** is the main class responsible for parsing the properties of beans in spring, and it supports both setter methods — returning either **this** or **void**. Which means everything works out of the box.

Mapstruct

Mapstruct, a library for auto-generation of conversion classes from one class to another, makes analysis of the corresponding class structures. It requires no additional configuration, and it recognizes setter methods that return **this.**

Commons-BeanUtils

Apache Commons BeanUtils, one of the most popular java libraries, also supports fluent setter methods. But it needs explicit configuration to support this:

```
PropertyUtilsBean propertyUtilsBean = new PropertyUtilsBean();
propertyUtilsBean.addBeanIntrospector(new
FluentPropertyBeanIntrospector());
propertyUtilsBean.setProperty(bean, "field", "value");
```

> **Pay Attention About Version**
>
> If you would like to use FluentPropertyBeanIntrospector, ensure that your commons-beanutils library version is 1.10.1 or newer (as older versions have severe bugs in this class)

Kotlin

The developers of the Kotlin language have solved all the problems that the proposed approach is trying to solve in Java. There are **data** classes and named passing of parameters to constructors and methods. Moreover, Kotlin has syntactic sugar to access the properties of objects

as calls to the corresponding getter and setter methods. Kotlin honors setter methods that return **this** instead of **void:**

```
private fun create(): AssignmentGroupedActivitiesResource {
    return AssignmentGroupedActivitiesResource().apply {  this: AssignmentGroupedActivitiesResource
        grouping = UserActivitiesGroupingResource().apply {  this: UserActivitiesGroupingResource
            alignmentScore = 1
            focusScore = 0
            advancedGroups = listOf(
                    ProductivityGroupResource().apply {  this: ProductivityGroupResource
                        sectionName = "Development"
                        color = "#2196f3"
                        spentTime = 5L
                    },
                    ProductivityGroupResource().apply {  this: ProductivityGroupResource
                        sectionName = "Chat"
                        color = "#E502FA"
                        spentTime = 1L
                    })
            periodLong = 10L
            totalTrackedTime = 7L
            intensityScore = 2
        }
    }
}
```

In Kotlin we can use setter calls in Java models as property assignments. Long live Kotlin!

Groovy

Similar to Kotlin, the Groovy language has the properties notation. If the according setter method returns **this** instead of **void**, Groovy allows to use it for property assignment.

```
public class SamplePojo {
    private String value;

    public SamplePojo setValue(String value) {
        this.value = value;
        return this;
    }
    ...
}
```

Similar to Kotlin, but there it's required to do **return it** calls from Closures.

```
static SamplePojo create() {
    return new SamplePojo().with { SamplePojo it ->
        value = "value"
        return it
    }
}
```

What Does The Spec Say?

There is a JavaBeans spec that doesn't explicitly say anywhere that setter methods should return exactly **void**, although all examples are written that way.

Not everything is so rosy. The main JDK class responsible for parsing the structure of Java Beans is **java.beans.Introspector**, and it doesn't recognize setter methods, if they return something other than **void**.

What Can We Break?

java.beans.Introspector is heavily used in Swing and Java EE elements. For example, if a JSP tag class tries to return **this** in its methods, it throws an error at runtime. This is perhaps the biggest problem with this approach: compilation and unit tests can't prevent this bug, unless it's a good integration or an e2e test.

Silent bugs can also be problematic if you use Apache Commons-BeanUtils, a library to work with beans (for example, copying properties from one bean to another). By default, copying bean properties ignores properties with setters returning **this**.
The FluentPropertyBeanIntrospector extension class adds this analysis, but it has a bug. There are two solutions to address the bug: 1, 2. However, even after trying to discuss this topic on the mailing lists, this issue remains open. It looks like the project was abandoned, despite its popularity.

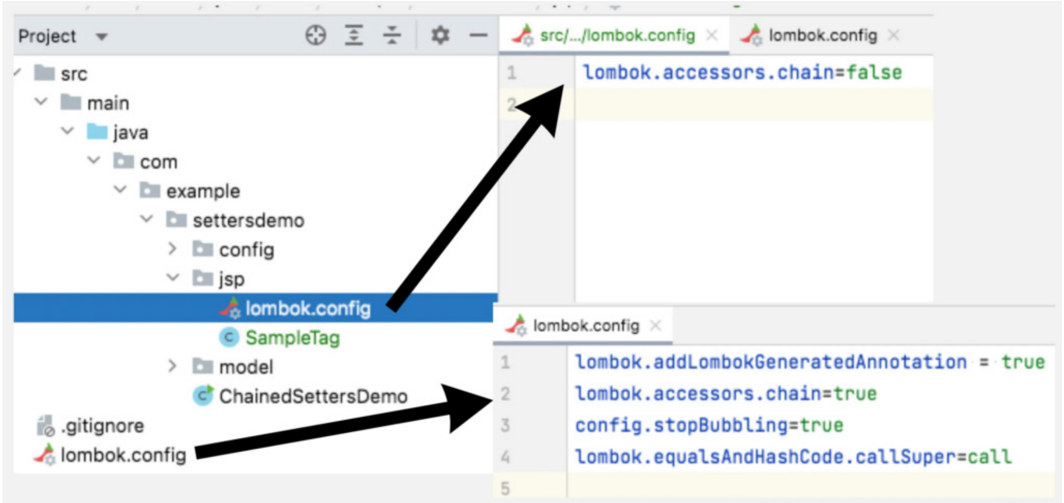
When To Use

I've been using this approach for the past few years. I apply it especially

when working with REST-APIs, because they have many classes of data models and DTOs. If a project is still in the development stage, there are fewer chances of breaking what works. Frameworks like Spring Boot are great, but if you are running on a traditional Java EE / Swing stack, it probably won't work for you.

How To Migrate

If there are no good integration tests, then migrate very carefully. Lombok allows flexibility in customization; for example, you can start with one class, package, or a new module. Or you can do it the other way around: enable chained setters at the level of the entire project, and disable them for a specific package; for example, JSP tags don't work particularly well with fluent setters:



In an existing project, it's better to implement these changes one at a time, iteratively.

Inheritance Problem

In case we inherit setters from the base class, a problem arises.

```
public class IdPojo {
    private long id;
    public long getId() {
        return id;
    }
    public IdPojo setId(long id) {
        this.id = id;
```

```
        return this;
    }
    // equals, hashCode, toString
}

public class SubPojo extends IdPojo {
    private String value;
    public String getValue() {
        return value;
    }
    public SubPojo setValue(String value) {
        this.value = value;
        return this;
    }
    // equals, hashCode, toString
}
```

Now we can't write code like this:

```
public static SubPojo create() {
    return new SubPojo()
        .setId(1) // returns IdPojo
        .setValue("value"); // compile failure
}
```

There are 4 possible solutions:

1. Often we need to get an object of the superclass at the output, not the class; we can write in reverse order, and return the supertype:

```
public static IdPojo create() {
    return new SubPojo()
        .setValue("value")
        .setId(1);
}
```

2. In the superclass, declare a constructor that accepts base fields as an exception. It's advisable to limit the number of fields. Remember that we may also need a parameterless constructor for deserialization.

```
public IdPojo() {
}
public IdPojo(long id) {
    this.id = id;
}
...
public SubPojo() {
}
public SubPojo(long id) {
    super(id);
}
```

3. Declare a generic type for **self**, and return **T** instead of **this**:

```
public class IdPojo<T extends IdPojo<T>> {
...
    public T setId(long id) {
        this.id = id;
        return self();
    }
    private T self() {
        return (T) this;
    }
    // equals, hashCode, toString
}
public class SubPojo extends IdPojo<SubPojo> {
...
}
```

4. To override methods in the inheritor:

```
public class SubPojo extends IdPojo {
...
    @Override
```

```
    public SubPojo setId(long id) {
        return (SubPojo) super.setId(id);
    }
```

Conclusions

- ✅ Reduced cognitive load
- ✅ Fewer local variables (coming up with names for local variables is one of the really difficult programming problems)
- ✅ Hierarchy of code, compact form
- ❌ Incompatibility with conservative stack
- ❌ Silent bugs

It might be a bold statement to define the use of fluent setters as a new trend. But a number of emerging technologies are going against traditional conventions. I made my choice. And I've never seen that this approach was reverted in projects that I participated.

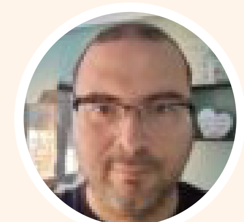
Code examples.

> Back to Table of Content

#JAVAPRO #IDE #TOOLS

Pull Requests != Code Reviews

Author:



Javier Lopez Fernandez joined Thoughtworks as a Lead Consultant Developer in September 2019 in the Spain office. He has 20 years of experience in the software industry, having worked in various sectors such as: - online travel agencies - the aerospace sector - public companies.

https://www.linkedin.com/in/

Pull request is a type of code review in contexts with lack of trust. What else do we have?

Source: Pixabay

The Fagan Inspection was first published in IBM in 1976, code reviews are old things in software. Pull Requests != Code Reviews.

> **Historically, the first code review process that was studied and described in detail was called "Inspection" by its inventor, Michael Fagan. This Fagan inspection is a formal process which involves a careful and detailed execution with multiple participants and multiple phases. Formal code reviews are the traditional method of review, in which software developers attend a series of meetings and review code line by line, usually using printed copies of the material. Formal inspections are extremely thorough and have been proven effective at finding defects in the code under review.**
>
> - *Wikipedia*

Anyone who has done code reviews in the past can see how easily it is to convert them into hard discussions.
Pull request is a type of code review in contexts with lack of trust, so they affect team morale.
A lot of times they are a tool to preserve "Status Quo" not a tool to improve quality in our product.
Bureaucracy and trust are things related, as less trust you have, more bureaucracy you introduce in the system.
Bureaucracy means long times to solve problems but doesn't mean to do a better job, just to use more time to evaluate things. But bureaucracy also means more waste, introducing more waiting times in the system, so money burnt.
The question then is how can we build more trust and what type of code reviews help us to do it?

Pull Requests (PRS)

PRs are a feature introduced by GitHub in their product from the very begining (2008). GitHub allows developers all around the world to collaborate in a distributed way in products. GitHub and Pull Requests are key to understand open source projects nowadays.

Pull Requests are **pre-merge async code reviews**. So the code review is done before the code is merged into the target branch. It is done through

the differences in code between the source branch which is modifying the code and the target branch, where the code will be merged.
Apart from the fact that they are using branches, they are also async by nature. Reviewer and reviewed are working in isolation and they communicate through the PR, they are not together.

The conversation is going to happen eventually. Reviewer will read the code and comment those parts unclear for him/her. Reviewed will be informed that reviewer has some comments. Then reviewed will try to solve them to present a new version solving these issues.

Basically, Pull Requests are async pre-merge code reviews. They are not thought to optimize lead time (latency between devs starts developing until that code is in prod), Pull Requests introduce big delays to have code deployed to production. Remember Pull Requests != Code Reviews.
In an open source scenario as described above. If lead time is crucial for a feature, the core team will do the review.

I usually say that they are a great way of doing code reviews on environments where you **cannot trust** in the people who write the code. Why, because of all the properties described above, they are gatekeepers, they try to avoid introducing bad changes in the project.

They are great for having a core team in charge of the project whose mission is to protect the product to create.

Let's imagine an open source project done by hundred of volunteer devs all around the world, the core team has to be sure no one of those devs have introduced malware in their codebase.

Pull request is a type of code review in contexts with lack of trust. So they are great for big open source projects where trust is impossible.

Developement Teams

Development teams are the usual way to change code in companies, they have been created with the idea that if people collaborate together they will achieve better and faster results.

The main focus of a development team is collaboration, we should optimize our ways of working having this in mind.

The idea of teams comes from collaborative sports as football or basketball, and to collaborate, **trust** is a must.

You need to trust in your teammates, you have to accept that you cannot compete alone against the other team, so you cannot run always to get the ball, you just need to trust that your teammates will help you on that.

If trust is important I will argue that a development team needs to see each other with a high frequency, can we build good dev teams in isolation, I don't think so. Can we create development teams in remote?, I think so, we have tools to be in touch all the day.

I think that the baseline to have a team is having all of them in a similar timezone, being able to talk the same language, and collaborate together as much time as we can.

Apart from trust we create development teams to reduce the lead time, we want to be as fast as we can to provide our new features to our customers.

So why is it going to be a good idea to use in development teams a process created for untrusted environments? Why is a good idea to use a process where lead time is not a priority?

Lead Time and Little´s Law

> **In mathematical queueing theory, Little's law (also result, theorem, lemma, or formula) is a theorem by John Little which states that the long-term average number L of customers in a stationary system is equal to the long-term average effective arrival rate λ multiplied by the average time W that a customer spends in the system.**
>
> **Expressed algebraically the law is L = λ W**

In the Kanban world of software development, we use different terms to those in Little's Law. Little's Law using Kanban terminology including Work in Progress (WIP), Throughput and Lead Time could be:

WIP = Throughput * LeadTime

Where
WIP = number of tasks in progress
Throughput = average departure rate, number of stories per week for example
LeadTime = average time a task spends in the system

Lead time doesn't refer to the time we spend on a feature. But rather the time it remains in the system. In our case, the system is our route to production.

Path to production = The steps to put the feature in front of our customers + all the waiting time.

Post Merge Code Reviews

Post merge code happens after the code is in main. We can do post merge codere reviews using a specific a step in our path to production. The code review is just a new state before considering the story finished. Summarizing a lot, the code review needs to happen to consider something as done.

Using the "commit message" will help us to identify commits for a specific feature.

Post merge code reviews are a signal of trust. I mean, you are allowing your teammates to merge their code in main because you trust them. Youo trust on the ways they work, their intentions, their knowledge, etc.

It's important to be sure that the developers care about what they build. So they are the main responsible for their changes.
This is another characteristic of trust, when you trust someone you also make him/her responsible for the things done.
Automatizing simple feedback is usually faster than waiting for people.

Then let's agree on tools to format our code, let's create fitness functions to protect some architectural characteristics, etc.

Post merge code reviews can be sync or async. If we want to reduce our Work In Progress, it is much better use sync reviews.
This means to have reviewer and reviewed in the same computer talking about the changes.
In async mode, reviewer checks the code and writes to the reviewed about changes.
If we agreed on some changes it is better to trust they will be done, not doing another review once they have been implemented.

More about post merge code reviews, here.

Pre-Merge Sync Code Reviews

The idea here is to do the code review with the reviewer and reviewed together. So they can talk about the changes at the time the dev has finished them.

Pull Requests are an option. But we cam also do pre-merge code revies without branches at all, reviewing the code in the computer of the developer before merging to main.

This technique remove some of the delays introduced by the async nature of pull requests, but it requires doing code reviews frequently in the team, to avoid pilling up code reviews. Every day for example you can have a time for code review scheduled in the calendar where everyone is checking the code done by others.

Collaborative Code Reviews

In fact, they are the natural thing that happens when you reduce the batch size to commits. Creating multiple PRs per hour will make reviewer and reviewed to work in the same computer, just because it's cheaper.
Pairing is in fact a method to do continuous code review. It increases trust because working together will make people to find agreements at the right moment.
Remember Pull Requests != Code Reviews.

Mob programming follow the same principle at higher scale. Reducing the WIP to one and making all the team to collaborate.

You can think that pairing or mobbing are going to slow down your team, but it is just the opposite. They are ways of working, based on collaboration, that reduce waste time to the minimum.
Coming back again with the Little's Law and your team:

LeadTime = WIP/Throughput

You can argue that is much more effective to paralellize work, but this is just true when that work doesn't need to integrate.
We tend to minimize the integration cost, it's not like playing lego but to transplat organs.

"**The devil is in the details**" so too much work to integrate when devs work in isolation will be a difficult task.

Let's don't delay integrations. We can integrate small parts frequently even if those parts are incomplete.
We should not underestimate the cost of integrate different parts done by different people during days in isolation.
Integrate small things frequently, will have a high impact in the quality of the product.
Waste time is against lead time, a lot of wast time means unfrequent integrations. This is why pairing or mobbing will help to create a better product sooner.

Code reviews try to improve the quality of the product created. But quality is a property of the system.
It's better to add quality at earlier stages not at the end. Pull Requests != Code Reviews.

> **Cease dependence on inspection to achieve quality. Eliminate the need for inspection on a mass basis by building quality into the product in the first place.**
>
> *- W. Edwards Demming*

> Back to Table of Content

#JAVAPRO #IDE #TOOLS

JBang, the Awesome Java File Runner

Author:

Dr Yassine Benabbas is a DevRel @ Worldline and a teacher. He is a fullstack developer with 20 years of experience, including 10 years of mobile dev (native and cross platform). He loves sharing his passion about programming in different languages through lectures, blog posts and videos. He is a maintainer of the official technical Blog and YouTube channel of Worldline and he is a member of the Lille Android User Group. All his lectures materials are Open Sourced on GitHub and his blog posts are published on Medium, dev.to and blog.worldline.tech.

https://www.linkedin.com/in/benabbasyassine

When developing on the JVM, we may write single file programs. In this situation, we want to build and run them in a very simple way. Classical tools such as Gradle or Maven are inconvenient for such cases. That's what JBang addresses in addition to bringing nice exclusive features. Let's explore them in this post.

Quick Start

JBang CLI (Command Line Interface) can be installed in many ways. The most straightforward one is to run one of these scripts depending on your OS:

```
#Linux/OSX/Windows/AIX Bash
curl -Ls https://sh.jbang.dev | bash -s - app setup
#Windows Powershell
iex "& { $(iwr -useb https://ps.jbang.dev) } app setup"
```

Once the setup is complete, we can open a new terminal session and start using the CLI. We can create a java file named "helloworld.java" and run it as follows:

```
jbang init helloworld.java
jbang helloworld.java # The file will be executed and will print
"Hello world"
```

The generated Java file is a typical one except for the first line, which marks the file as a *JBang* script.

```
///usr/bin/env jbang "$0" "$@" ; exit $?
import static java.lang.System.*;

public class helloworld {
    public static void main(String... args) {
        out.println("Hello world");
    }
}
```

JBang also supports running Kotlin and Groovy files. Here is an example of creating and running a Kotlin file:

```
jbang init -t hello.kt helloworld.kt
jbang helloworld.kt #will run the file and print "hello world"
```

This are some basic features of JBang that unlock many use cases such as teaching, running custom CI scripts and prototyping.

Let's see some more interesting single file apps that we can create with JBang

Using Jbang With Libraries

JBang files can be configured by adding special comments before the comments and any code. Here are some examples:

Description	Syntax
Add a dependency	//DEPS "gradle style dependency"
Add a file	//SOURCHES "relative path to source file"

Use experimental features of Java 23	//JAVA 23+
	//COMPILE_OPTIONS -- enable-preview - source 23
	//RUNTIME_OPTIONS -- enable-preview

Quarkus property	//Q:CONFIG "property"="value"

Let's illustrate next with some concrete examples.

Java File Without A Main Class

This is an experimental feature of Java 23. Thus, we need to use the options that enforce Java 23 as well as the compiler and runtime options that activate experimental features.

```
///usr/bin/env jbang "$0" "$@" ; exit $?
//JAVA 23+
//COMPILE_OPTIONS --enable-preview -source 23
//RUNTIME_OPTIONS --enable-preview

void main(String... args) {
  System.out.println("Hello World");
}
```

Quarkus Rest API with JSON Parsing

JBang supports the BOM feature, allowing to define the version of Quarkus in one line and apply it to all its extensions without having to repeat the version number.

The following example uses quarkus-resteasy-jsonb for JSON parsing because it does not require to setup compiler or build plugins, making it simpler and more relevant for our use case.

```
///usr/bin/env jbang "$0" "$@" ; exit $?
//JAVA 17+
// Update the Quarkus version to what you want here or run jbang with
// `-Dquarkus.version=<version>` to override it.
//DEPS io.quarkus:quarkus-bom:${quarkus.version:3.15.1}@pom
//DEPS io.quarkus:quarkus-resteasy
//DEPS io.quarkus:quarkus-resteasy-jsonb
//DEPS io.quarkus:quarkus-smallrye-openapi
//DEPS io.quarkus:quarkus-swagger-ui

//JAVAC_OPTIONS -parameters
//JAVA_OPTIONS -Djava.util.logging.manager=org.jboss.logmanager.
LogManager

//SOURCES PalindromeService.java

//Q:CONFIG quarkus.banner.enabled=false
//Q:CONFIG quarkus.swagger-ui.always-include=true

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.QueryParam;
import jakarta.ws.rs.core.MediaType;

import java.util.Map;

@Path("/palindrome")
```

```java
@ApplicationScoped
public class palqrest {
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Map<String, String> isPalidrome(@QueryParam("input")
    String input) {
        return Map.of("result",
                PalindromeService.isPalindrome(input) ? "Palindrome"
                : "Not Palindrome");
    }
}
```

The file PalindromeService.java imported by the script is defined as follows:

```java
public class PalindromeService {
    static boolean isPalindrome(String input) {
        int l = input.length();
        for (int i = 0; i < l / 2; i++) {
            if (input.charAt(i) != input.charAt(l - i - 1)) {
                return false;
            }
        }
        return true;
    }
}
```

Isolating the service in a separate file makes it easier to develop a CLI variation with picocli:

```java
///usr/bin/env jbang "$0" "$@" ; exit $?
//DEPS info.picocli:picocli:4.6.3
//SOURCES PalindromeService.java

import picocli.CommandLine;
import picocli.CommandLine.Command;
import picocli.CommandLine.Parameters;
```

```
import java.util.concurrent.Callable;

@Command(name = "palcli", mixinStandardHelpOptions = true, version =
"palcli 0.1", description = "palcli made with jbang")
class palcli implements Callable<Integer> {

    @Parameters(index = "0", description = "The greeting to print",
    defaultValue = "World!")
    private String inputString;

    public static void main(String... args) {
        int exitCode = new CommandLine(new palcli()).execute(args);
        System.exit(exitCode);
    }

    @Override
    public Integer call() throws Exception { // your business logic
    goes here...
        System.out.println(PalindromeService.
        isPalindrome(inputString) ? "Palindrome" : "Not a
        Palindrome");
        return 0;
    }
}
```

Sharing With Catalogs

JBang catalogs provide an easy way to run scripts, also called aliases, and import templates from external sources. Defining a catalog consists of adding a JSON file named 'jbang-catalog.json' in the root folder of any GitHub repository. For example, I host a catalog in github.com/yostane/jbang-catalog/jbang-catalog.json with the following content:

```
{
  "catalogs": {},
  "aliases": {
    "palcli": {
      "script-ref": "scripts/paltools/palcli.java",
```

```
      "description": "Palindrome tester CLI"
       },
       "hellojfx": {
      "script-ref": "scripts/hellojfx.java",
      "description": "Basic JavaFX window that shows Java and JavaFx ⏎
versions"
      }
  },
  "templates": {
    "javafx": {
      "file-refs": {
        "{basename}.java": "templates/javafx.java.qute"
      },
      "description": "A starter JavaFX app"
    }
  }
}
```

My catalog defines two aliases and a template. The basic syntax of calling an alias is "jbang *alias@github_user/repository [args]*". The default value of repository is "jbang-catalog". Thus, we can run the two alises as follows:

```
jbang palcli@yostane madam #will check if madam is a palindrome
jbang hellojfx@yostane #Shows a JavaFX window
```

We can browse through aliases provided by the community in the [JBang AppStore](#) which is a webpage that indexes all JBang catalogs on GitHub.

Coming back to my catalog, we note that is contains a template in addition to the aliases. This template generates a JavaFX application and can be used with this command:

```
jbang init -t javafx@yostane hellojfx #generates a java file named
hellojfx.java
jbang hellojfx.java #run the file generated by the template
```

For more details on how to setup a catalog, please refer to the documentation.

Why Jbang

After exposing the most important features of JBang, we may be challenged about the relevance of this tool compared to what is currently available. Even though technical choices are somewhat influenced by comfort and other subjective preferences, let me try to address some questions in an objective way:

Compared to Maven and Gradle

Let's suppose you want to develop prototypes that fit in one or two Java files or you are a teacher who wants to showcase Java (or Kotlin or Groovy) features in small apps. If we use Maven or Gradle, Imagine how many projects, folders and files would have been created. With JBang, since each Java file is autonomous (it contains the code and the settings), then a single folder that contains only Java file is enough.

In summary, while Maven and Gradle are awesome for medium and large projects, JBang is more relevant for small apps and for teaching. In addition to that, teachers can take advantage of catalogs to share templates and aliases to their students.

Compared to an Interpreted Language

This is a relevant question: why use a compiled language for scripting purposes instead of using an interpreted language such as Bash, JS or Python?

In my opinion, it depends on what we mean by scripting. If we define it as using an interpreted language, than there is no discussion by definition since Java is compiled. Although, we can argue that Java is compiled and interpreted and some scripting languages do some kind of compilation.

Personally, I prefer to define scripting as writing *small apps in a short period of time* that accomplish *very specific tasks*. This definition is bound to the goal and not the technology. This means that whatever

tool fits the criteria can be used. In that case, teams consisting of Java of Kotlin devs can be pragmatic and use JBang instead of learning Bash, JS or any other scripting language.

Conclusion

While being initially loved for it's simplicity, some qualify Java of being heavy and complex. Maybe part of this is related to the structure of Maven and Gradle projects. JBang brings back the simplicity to Java projects while providing powerful and modern features such as catalogs and using dependencies. It makes writing small apps a breeze and teaching Java even more joyful.

> Back to Table of Content

#JAVAPRO #PROJECTMANGEMENT

Agile Nightmares - When Agile Methods Become an Innovation Barrier

Author:

Nina Nitzsche is an Agile Consultant and previous developer based in Berlin, with over 10 years of experience in agile software development. Passionate about fostering team collaboration, driving continuous improvement and empowering teams to achieve their full potential through agile methodologies.

https://www.linkedin.com/in/nina-nitzsche-134382204/

The Scary Figures of Agility

In the dynamic world of software development and beyond, agility has established itself as a
defining approach.

However, behind the promise of flexibility, customer orientation, and continuous improvement, pitfalls sometimes lurk. Calls are increasing to declare agility as ineffective or even 'dead'. The real reasons for this

growing scepticism, however, rarely lie in the agile principles themselves, but rather in their creeping or even deliberate misinterpretation and application.

In the following, we shed light on some of these concise anti-patterns that slow down teams and can counteract the intended benefits of agile ways of working. A special focus is placed on identifying these pitfalls and possible ways to overcome them successfully.

Zombie Scrum – Agility Without Signs of Life

One of the most unsettling phenomena in agile practice is so-called Zombie Scrum. This term describes teams that formally follow the Scrum process but have lost any vitality and the central focus on customer value.

The Scrum rituals are carried out mechanically and without any critical reflection on the added value. Teams merely imitate the external structure of Scrum without internalizing and applying the fundamental principles such as self-organization, transparency, and continuous adaptation to customer needs.

Without these essential principles, however, Scrum loses its core advantages in terms of continuous improvement, adaptability, and a strong customer orientation.

A typical and frequently encountered example of Zombie Scrum manifests itself in the execution of the Daily Scrum. Instead of serving as a synchronizing meeting in which the development team aligns itself with the sprint goal, openly addresses impediments, and identifies topics for further discussion, the Daily often degenerates into a pure status report.

Each team member reports in isolation what they did the previous day, what they will do today, and whether there are any obstacles, without any real discussion or joint planning of further action arising from it.

As a result, valuable information about potential problems and actual progress remains unused, and the team loses its common focus on the sprint goal, as well as the crucial opportunity to improve collaboration

efficiently and react flexibly to changes. The actual goal of the Daily, namely aligning with the sprint goal and identifying blockages, thus becomes obsolete. But we also encounter this pattern away from the Daily Scrum.

Further symptoms of Zombie Scrum are diverse

- Scrum rituals are performed purely mechanically without questioning the benefit.
- The team does not make real decisions and acts more as an executing body.
- The team lacks the necessary autonomy to organize its work independently.
- Retrospectives do not lead to concrete and verifiable improvements in the process or collaboration.
- There is a lack of a clear understanding of goals within the team, and the motivation behind the work is unclear.
- Scrum events become empty ceremonies that are more reminiscent of status meetings than valuable discussions.
- There is little or no stakeholder engagement, so important feedback from users or other stakeholders is missing.
- Sprint Reviews are predictable and merely present status updates instead of genuine innovation or new insights.

As a consequence of the meaninglessness and lack of opportunities for influence, burnout or apathy within the team is not uncommon.

Furthermore, it is problematic insofar as it paralyzes the entire Scrum process. The purely mechanical execution of rituals can significantly reduce the engagement of team members and lead to frustration.

The original goal of Scrum – adaptation and continuous improvement – is neglected, and the noticeable added value of the agile way of working is lost. What remains are time-consuming and empty meetings.

To Combat Zombie Scrum, Consistent Measures are Needed

○ **Build, refresh, and question Scrum knowledge:** In the daily routine of our Scrum team, we mainly focus on the implementation of Scrum. This is natural because the events are intended to support commitments and strengthen responsibilities. However, if their background is unclear, the events degenerate int mere measures – their actual goal is lost. It is therefore helpful for teams to clarify basic questions, for example:

- What responsibilities and accountabilities do we bear?
- Whom should which event help and how can this be achieved?
- What is a 'valuable increment' for us?
- How do we use our events specifically to achieve our commitments (product goal, sprint goal, Definition of Done)?

○ **Retrospectives with concrete measures:** Retrospectives must lead to actual changes – otherwise, they lose their meaning. However, motivation often turns into overcommitment, and too many action items are taken on simultaneously instead of focusing on a few, implementable measures.

Less is often more here: if a team cannot anchor its improvements in practice, they remain mere declarations of intent – just like empty Scrum events. It becomes particularly problematic when measures from the retrospective regularly fizzle out. Then not only the implementation power suffers, but also the trust in the meaning of the retrospective itself. Teams should therefore consciously prioritize:

- Which one measure will take us furthest in the next sprint?
- How do we ensure that it is actually implemented?

This keeps the retrospective a valuable tool for continuous improvement

○ **Focus on customer value:** Teams should regularly reflect on how their work provides added value to the customer. If sprints are only about processing tickets without a clear understanding of customer value, Scrum loses its actual raison d'être.

Teams should therefore consciously reflect on how each developed feature or improvement contributes specifically to solving a customer problem. A simple way is to ask the following question in every Sprint Planning:

- How does this increment help the user?

○ **Sprint Reviews:** Should not only be status reports but actively solicit feedback from stakeholders and users to continuously validate the added value. The already mentioned sprint goal, coupled with the product goal, is again the basis for discussion here. Together with stakeholders, the Review can clarify:

- Have we achieved the sprint goal?
- Were we able to solve the customer's problem a bit with the sprint goal?
- Does the result meet the expectations of our stakeholders?
- Has this sprint brought us closer to our product goal?
- What do we learn from the feedback to work even more specifically on the product goal in the next sprint?

○ **Restoration (and use) of a clear goal for each sprint:** Ensure that each sprint pursues a value-oriented and understandable goal and that this goal is used consistently.

○ **Bring the Scrum ceremonies to life**: Scrum events should serve as a platform for genuine exchange and valuable discussions, not as empty rituals performed "because Scrum." Their real purpose is to create transparency, promote collaboration, and enable continuous improvements.

Bring Scrum Ceremonies to Life

If Scrum events fulfil their actual purpose again, Scrum will not only become more vibrant but also more effective.

1. Sprint Planning

The Planning is more than just distributing tasks for the next X weeks. It should create a common understanding of why this sprint is important and what value it should deliver. Instead of focusing exclusively on story points or individual tasks, the team should actively discuss:

- What is our goal for this sprint – and why is it relevant?
- How do the planned items contribute to achieving this goal?
- Are there dependencies or risks that we should clarify in advance?

2. Daily Scrum

The Daily should be a dynamic coordination for achieving the goal, not an isolated status report. Teams should talk less about what they have done and more about what needs to be done to achieve the sprint goal in the best possible way.

- What do we need to do to achieve our sprint goal in the best possible way?
- What obstacles are in our way – and how can we solve them together?

3. Sprint Review

Reviews must enable genuine dialogues with stakeholders, not pure presentations for managers. Furthermore, such dialogues make these events more tangible for stakeholders. Suddenly, they are no longer passive participants but have the responsibility of co-creation. Instead of presenting processed Jira tickets, it helps to actively ask for feedback:

- Does the increment deliver the desired added value?
- Does it meet the expectations of the stakeholders – or what is still missing?
- What insights do we take with us for the next sprint?

4. Retrospective

Retrospectives should be used as safe spaces for critical reflection and genuine improvements. So that they do not degenerate into mere routine exercises, their output must be concrete, implementable, and verifiable.

Strengthening Teams Through Autonomy

Scrum teams need the authority to make decisions and achieve results. Only when teams have clear decision-making powers – be it in the choice of technical implementation, the prioritization of tasks, or the adaptation of processes – can they unleash their full potential.

To reflect on their own autonomy, teams should ask themselves the following questions:

- What responsibilities are we currently responsible for?
- Can we really do justice to these responsibilities in our role? If not, what do we need to successfully carry this responsibility?
- What overlaps or blockages prevent us from having more autonomy?

Only through an honest examination of these questions can teams regain their decision-making power and actively work towards continuous improvement.

Regular Reflection on Meetings

The purpose and benefit of each Scrum Event should be regularly questioned and adapted if necessary. If meetings become a habit and no one thinks about their benefit anymore, they lose their original power. If meetings do not have the desired effect, they must be adapted – be it through changed formats, such as more frequent but shorter durations or a clearer objective. Scrum thrives on continuous improvement – this should also apply to the process itself.

Therefore, teams should regularly reflect on their own processes:

- Is our Daily really helpful or just a formality?
- Do we use the retrospectives meaningfully?
- Are our Sprint Reviews interactive enough?

Dark Agile – The Dark Side of Agility

Another worrying phenomenon in agile practice is "Dark Agile." This term describes the conscious or unconscious misapplication of agile principles, where control and pressure are exerted instead of trust and self-organization. Dark Agile uses agile methods to strengthen hierarchies and micromanage instead of empowering teams and promoting their self-responsibility. This distorted application leads to a toxic work environment that contradicts the actual values of agility.

A typical example of Dark Agile is the misuse of terms like "Accountability." Instead of promoting the self-responsibility and commitment of team members, this term is used by anagers to exert more control and centralize decisions. Behind this appearance often lies a "Hidden Agenda" that has nothing to do with the real goals of agile methods but rather serves to maintain an old power structure.

The Symptoms of Dark Agile are Diverse and Often Subtle

- **Redefinition of agile terms:** Agile terms such as "Accountability," "Autonomy," or "Transparency" are deliberately redefined to legitimize micromanagement.
- In Dark Agile, **accountability** is used to demand responsibility without giving team members the necessary freedom to act independently. Instead of fostering a culture of trust, it is misused as a tool for micromanagement and control. For example, a team member is constantly asked about the progress of their work without having the freedom to make decisions. Instead of receiving support, the person feels monitored and under pressure, which undermines self-responsibility.
- **Autonomy** means the ability and freedom to make decisions

independently. In an agile environment, autonomy promotes the self-organization of teams, whereby each team member takes responsibility for their own tasks and decisions without being constantly controlled from the outside. Autonomy strengthens trust within the team and helps to foster innovation and initiative. In the case of Dark Agile, however, this state is thwarted. A team is referred to as "autonomous," but all important decisions must be approved by the management level. This creates the impression that the team is working independently, while the actual decision-making is strongly controlled and restricted.

- **Transparency** in an agile environment means that information is made open and accessible to everyone. It is about clearly communicating both successes and challenges so that all team members and stakeholders have the same information base to make informed decisions. Transparency fosters a culture of trust and collaboration. Misguided Interpretation in Dark Agile: In Dark Agile, "transparency" is often misused as a tool for monitoring and control. Instead of openly sharing information or using metrics to support, team members are forced to document and report their work progress. This "transparency" is used for verification and control, not to promote collaboration. This supposed transparency also often leads to relevant information that needs transparency being withheld. Thus, only selected information is shared from all sides, while other information is held back.
- **Responsibility without freedom to decide:** Teams are held accountable for results but have no or insufficient freedom to decide how to influence these results.
- **Continuous pressure:** Instead of focusing on continuous improvement, there is constant pressure to deliver results without room for adjustments and reflection.
- **Lack of a culture of trust:** Agility is applied superficially without creating a culture of trust and openness.
- **Forced introduction of agility:** Agility is forcibly introduced without providing the necessary resources or adequate training.
- **Misuse of metrics:** Metrics are used to control teams instead of supporting their work and identifying potential for improvement.

- **Resistance to change:** The necessary adaptability is lacking, and resistance to change is noticeable while still adhering to outdated processes.

The Consequences of Dark Agile

Dark Agile undermines the fundamental principles of agile methods by promoting control and hierarchy instead of transparency and collaboration. The result is a toxic work environment that stifles innovation and creativity. Teams feel overwhelmed and unable to work independently, which can lead to frustration and even a loss of motivation.

If Scrum Masters and/or managers are misguided, they move away from their task of acting as supporters and enablers and instead become guardians of micromanagement and unnecessary bureaucracy, destroying the agile principle of self-organization.

Measures Against Dark Agile

To successfully combat Dark Agile, organizations must take fundamental measures that refocus on agile values:

- **Establish trust-based leadership:** Managers must trust the teams and transfer responsibility for decisions to them. Trust forms the basis of agile ways of working, and managers play a key role in establishing a culture of respect, openness, and transparency. They play a key role here by acting as role models and actively living agile behavior, which also includes the clear and transparent communication of decisions. Instead of exerting control, they act as enablers and encourage the team to make decisions independently. At the same time, they bear great responsibility in the agile way of working. Through clear orientation and transparency, they can ensure that the teams work in accordance with the overarching corporate goals and values. Furthermore, they are crucial for the development of self-organization. As mentors, they accompany the team through challenges without unnecessary intervention, thus promoting the development of self-responsibility and initiative.

- **Implement transparent feedback loops:** Teams need the opportunity to receive and give feedback regularly. It is important to create an open communication culture in which teams can address their needs and challenges without fear of consequences.
- **Offer comprehensive training:** To ensure that agile principles are correctly understood and applied, training courses and workshops should be mandatory for all team members and managers. The correct understanding of agility is crucial to avoid misuse and implement the principles correctly.
- **Use metrics meaningfully:** Metrics should not be used as instruments of control but as tools to measure progress and identify potential for improvement. They must be designed in such a way that they support the team and do not put pressure on them.
- **Promote adaptability:** Agility requires continuous adjustments. It is important to create an environment in which changes are not only allowed but actively encouraged. Teams should be encouraged to try out new ideas and learn continuously instead of getting stuck in old processes.

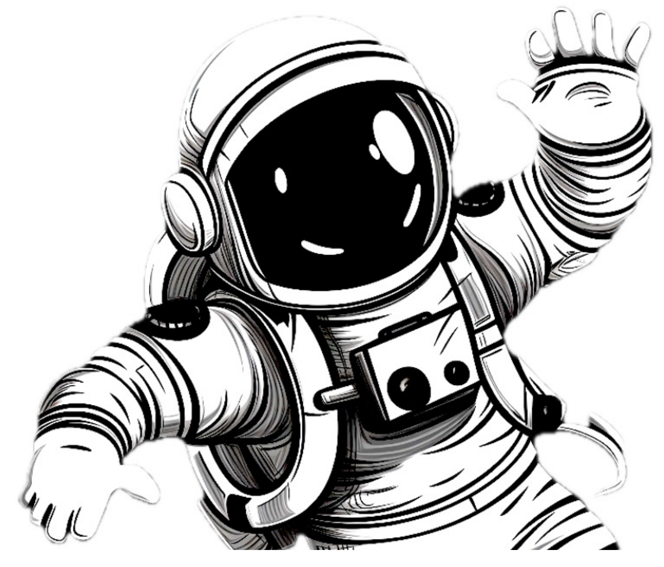
Conclusion: The Return to the True Values of Agility

Dark Agile shows how agile thinking can be misused to build control and pressure. It undermines the purpose of agile methods and leads to a toxic work environment. To overcome Dark Agile, organizations must put the principles of trust, transparency, and continuous improvement back at the center. Managers and Scrum Masters play a key role in this by leading by example and actively supporting the team.

The true strength of agility lies in the self-organization of the teams and the ability to continuously improve. Only when these principles are consistently lived can agility unleash its full potential and lead to real, sustainable change. The misunderstandings and misapplications of agile methods, as seen in the concepts of "Zombie Scrum" and "Dark Agile," illustrate that agility is far more than just a process – it is a culture based on trust, transparency, and self-organization. If these principles are disregarded, agile methods lose their true benefit and lead to a toxic work environment that inhibits innovation and continuous improvement.

To avoid this trap, organizations must revive the core values of agility and anchor them in their daily work processes. Managers and Scrum Masters are crucial here, as they act as role models and must promote the self-responsibility and autonomy of their teams. Agility thrives on continuous reflection, improvement, and the willingness to learn from mistakes. The goal should be to establish an agile culture that not only optimizes processes but also empowers employees to act independently. Only in this way can agility unleash its full potential and promote long-term innovation and sustainable success.

> Back to Table of Content



#JAVAPRO #PROJECTMANGEMENT

Journey to Junior

Author:

Andreas Monschau works as a Senior IT Consultant at Haeger Consulting in Bonn. He has taken on various roles, ranging from software developer and software architect to test manager, team leader, and requirements manager. In addition, he leads Haeger Consulting's extensive trainee program and supports other companies in establishing similar processes.

https://www.linkedin.com/in/andreas-monschau-544aa141/

Alternate Paths Into Software Developement

There’s a real shortage of the right kind of talent out there, and innovative new approaches can be a great way to tackle this problem. Over the past 10 years at Haeger Consulting, we’ve poured time and resources into building a trainee program aimed at bringing lateral entrants, career changers, and newcomers into Java software development – and we’ve seen some amazing success stories along the way. “So, how exactly do you do it?” is one of the many questions we get asked.

So, how do we do it?

A Quick Historical Overview

The trainee program we know today didn't just appear overnight—it evolved through hard work. It all started with me making the switch into the Java ecosystem. I came from a Delphi/.NET background, and what I'd done before could only generously be called "professional software development." So, I wasn't completely out of my element or a lateral entrant in the traditional sense; I was more of a career changer. But when it came to Java, I was as far removed as the Earth is from the sun. One day, I went for an interview with my current boss, Ralf, and got thoroughly grilled by an employee (employee number 1, who's still with us today). After the interview, as we were saying goodbye, Ralf asked, "So, can Andreas do anything?" The answer was as blunt as it was crushing: "Nothing we need." Then Ralf asked, "But do you think he can learn?" "Probably," was the reply—and that was my green light to start teaching myself everything. I managed pretty well, and after a year I was asked, "Andreas, would you like to mentor a trainee for a few months and teach them Java?" Of course, I couldn't say no—and that's how our trainee program was born. In hindsight, it all seems so well planned.

Who We're Looking For

But what kind of lateral entrants, career changers, or newcomers are we after? What qualities should they have? Over the years, we've come to a bold conclusion: about 95% of people just aren't cut out for quickly breaking into IT (a hard lesson learned from trainees who didn't make it). So, we focus on finding that other 5% and equipping them to be ready for their first Java project in a reasonable amount of time. Here are the essential qualities we look for:

CURIOSITY
As a beginner, you need to be curious about the field you're entering. You want to understand how things work—how software and software development tick. It helps to have that "seek out new lifeforms and new civilizations" mindset.

GRIT
You need the determination to really dive into a topic. This isn't about stubbornly sticking with something that isn't working; if you hit a dead

end, you should be able to let go, take a step back, come up with a new plan, and try again.

MOTIVATION

There has to be something that drives you. Ideally, you're excited by the idea of developing software. Just writing a doubly-linked list in Turbo Pascal back in school isn't enough—you should have already played around with a programming language or even a framework.

INDEPENDENCE

Being able to work independently is key. You must be capable of learning new things on your own (of course, with some guidance) and then proactively continuing the work. Waiting around for instructions won't get you far.

TEACHABILITY

It doesn't help if you react badly to criticism. You need to be open to feedback and willing to act on it.

ANALYTICAL AND ABSTRACT THINKING

This is a classic requirement you see in nearly every software developer job posting. For our trainee program, being able to think both analytically and abstractly is essential.

COMMUNICATION SKILLS

It's important to be able to express yourself clearly. You'll always be interacting with colleagues, managers, clients, and so on, so good communication is a must.

COURAGE

Last but not least, courage. Perhaps the most important quality of all: the courage to leave something familiar behind and take a new step. It's challenging to embark on this journey, but it can pay off—both for you and for the company.

The people who fit this bill can come from all walks of life—from former tax clerks to long-time students to even those with PhDs. What matters most is understanding what drives you and knowing that you've got these qualities in spades.

Set A Course...

When we talk about our trainee program, we're referring to a period of about six months. That means we have to clearly define what can be achieved during that time (and what can't). We developed a curriculum that sets the content and the boundaries: our Journey 2 Junior, or J2J for short.

Knowledge has to be built up continuously. Over these six months, we develop skills step by step, chapter by chapter, with each chapter laying the foundation for the next. Essentially, we've defined three J2J phases that cover all the technologies, tools, and methods we consider absolute "must-haves".

Phase 1: Fundamentals
We start by meeting each trainee at their current level and work to bring everyone up to speed. They learn core concepts like object-oriented programming and dive deep into standard Java, databases (and how to access them), testing, and version control with Git, among other things.

Phase 2: Frameworks, Architectures, Interfaces
Once the fundamentals are in place, this chapter usually takes off on its own. Trainees get introduced to key parts of the Spring Framework, start to understand what REST really means, dive deeper into persistence, and explore architectures, design patterns, and more. In the realm of frameworks, those "aha" moments really start to happen—like realizing, "Wow, this automatically handles tasks I used to do by hand, and because I had to do it manually before, I now really understand what's going on. It's not just magic!"

Phase 3: Build and Operations
The final phase focuses on topics such as CI/CD, container technologies, cloud deployment, and similar areas. Here, the "aha" moments might come in the form of, "Oh, this is actually possible!" or "Wow, that's so simple!"

Throughout the journey, the trainees work on a comprehensive real-world project—a continuously evolving application. And they're never

left to fend for themselves. Our skilled trainers are there to guide them, provide constant feedback, answer questions, clarify tasks, review challenges (more on that later), and track each trainee's individual progress.

It's About Getting People Together

We encourage our trainees to engage in lots of discussion—not only to share what they've learned but also to connect with others. They do this through "Mini Skill Factories," which are one-hour presentations on specific technical topics. These sessions are attended by trainees from various disciplines, alumni, and even interested colleagues up to the senior level.

Additionally, several times a year, we hold "Trainee Events." These bring together the current trainee generation, trainers, and alumni (those who have successfully completed the program). It's part fun (laser tag, pizza parties) and part networking between current trainees and alumni. On top of that, we promote interaction between trainees and all employees during company events. We're also thrilled when our trainees attend meetups like Java User Group events to further expand their networks within the local community.

Challenging Challenges

But it's not just about pizza parties, friendly chats, and networking. Periodically, between two phases, we evaluate each trainee's progress within the J2J program using various tools—and one of our key tools is the "Challenge."

At the end of each J2J phase, we set a task with a clear scope that must be solved using at least the technologies, tools, and methods covered in that block. For example, at the end of the first block, the challenge (simplified) requires trainees to implement the behavior of a bank account. Along with an explanatory diagram, we provide a few requirements:

- The customer can deposit money into a bank account (i.e., transfer funds).
- The customer can withdraw money from a bank account.

- The customer can retrieve the current balance.
- The customer can get a list of all transactions on their account.
- The customer can generate account statements that include all transactions between a specified start and end date (inclusive).

From these requirements, the first classes and use cases emerge. To solve the challenge, the trainee must apply what they learned in the first J2J block, such as:

- Object-oriented programming fundamentals
- Standard Java
- Maven
- Data formats
- Git
- Testing with JUnit
- Database access with Java

We intentionally leave out GUI topics here (we don't see much value in building user interfaces with AWT and Swing these days), along with other topics we classify as "out of scope."
This challenge is designed to be completed in two person-days by a single trainee. Once the time is up, the trainee's access to the Git repository is revoked so that there's no chance of working on it after hours. Then, a trainer conducts a thorough review of the solution. During that review period, the trainee can explore bonus topics. After the review, the trainer and trainee discuss the results to pinpoint any gaps or areas needing improvement—or confirm that everything is on track. In any case, the trainee then moves on to the next chapter, with the opportunity to address any identified issues.

Lessons learned

Over the years, we've gathered a lot of experience. Things don't always go smoothly when you're building a trainee program like ours, but many aspects have turned out remarkably well.

More than 30 of our employees have gone through the training program in one form or another, and many are still with us today. Our dropout rate has dropped dramatically—in fact, on average, only one trainee fails to complete the program. It's been so interesting that companies (even competitors!) have reached out asking, "How do you run your trainee program?" With these inquiries piling up, we decided to eventually offer our training program model as a service. We're excited to see the first case studies come out of it!
On the flip side, things have gone wrong in the past when a trainee relationship ended due to mismatched expectations on both sides, which often made long-term collaboration difficult or even impossible. We've learned from these experiences and now emphasize our expectations for trainees as early as possible (ideally during the first meeting, but at the latest just before hiring). We also require prospective trainees to share their expectations of us, or we clearly outline what they can expect from the program.

You keep learning, and in the same way, we continuously expand and improve our trainee program. "Bad" experiences are simply steps on the path to getting better—not setbacks. Inspect and adapt with a human touch.

So, Why Do All This?

Is it just about cheaply producing new developers, or is there more to it? Let me share my personal impressions: over the years, we've witnessed some truly remarkable success stories from our trainees—people who were stuck in personal or professional crises, nearly lost to the job market, or who had been stuck in academia for years without really feeling fulfilled. Together, we recognized opportunities, identified potential, and worked through the trainee program. Now, many of these individuals are in lead roles in their projects—roles that once, not even they believed they could fill. You can make a difference and give people a new perspective. Or you can help them achieve their dream of breaking into IT, especially in Java software development. In most cases, this is met with loyalty—a loyalty we return in kind. Besides, it's also a great way to retain employees in the long run (but that's a story for another time...).

> Back to Table of Content

#JAVAPRO #SECURITY

Move Fast, Break Laws: AI, Open Source and Devs (Part 3)

Author:

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert.
Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.

https://www.linkedin.com/in/noregressions/

The software development landscape is rapidly changing, with legislation emerging as a key driver of industry trends. As our reliance on software and AI grows, so does our vulnerability to cybercrime, which is now a multi-trillion-dollar problem. This has caught the attention of regulators worldwide.

This article series explains the various regulatory efforts in play and summarises actions that developers and executives should consider as they prepare for 2025, the year of software legislation.

Part 1 covered the background, what a software supply chain is and thoughts on AI and open source.

Part 2 explored how governments are working to create legislation and what the current status is.

Part 3 (this article) offers both a Software supply chain and an AI governance & compliance checklists for developers and executives to consider

Part 4 will discuss cybersecurity and incident reporting requirements, examines geopolitical compliance and liability management, and wraps up the series.

There's a lot to take in. I hope you're sitting comfortably.

Accountability Cannot be Outsourced.

I am not a lawyer. This document is a technical view of the legislation and regulations being developed or repurposed. It's imperative to get your own legal assessment when deciding if these elements apply to your situation. Having said that, some aspects are shared. The primary one is accountability. There's no dodging your responsibilities. That means wherever you are in the software supply chain, you have responsibilities to those consuming your software and those using it. Regulations collectively require organisations to assess, monitor, and manage third-party risks, and you'll have to prove that you did the right thing at the right time.

Blaming others without proper due diligence and safeguards is not a valid defence!

Responsibilities and Compliance Checklist

The following sections list commercial tools. They are for example only and do not constitute recommendations or endorsements.

This next section has significant references to keeping documentation and records secure. It's important to realise that this is both evidence

and a critical asset. It's evidence that your software (and AI) follows the rules, but it's also a vital asset because it tells you what's happening and what's in the software you ship. When things go wrong, this data asset will be a primary source for quickly determining why - and you will need the speed.

AI Governance & Compliance

Classify AI risk levels and maintain documentation of training data and testing.

Start by assessing your AI system using a formal risk classification framework. The EU AI Act, for example, categorizes AI applications into *unacceptable, high-risk, limited-risk*, and *minimal-risk* tiers, based on their potential impact on rights and safety. Adopt a similar internal model tailored to your industry, such as using a risk matrix or adapting NIST's AI Risk Management Framework (NIST AI RMF). For each AI model, maintain clear documentation on training data sources, preprocessing steps, data lineage, and test set composition. This supports reproducibility and helps identify blind spots or harmful correlations. Tools like Model Cards or Datasheets for Datasets offer great starting points for structured documentation.

Implement bias testing, transparency controls, and human oversight.

Bias testing should be integrated into both the training and deployment phases. Use libraries like Fairlearn or IBM's AI Fairness 360 to test for statistical parity, disparate impact, and equal opportunity across demographic groups. For transparency, implement model explainability techniques such as SHAP or LIME to interpret decisions and expose reasoning, especially for high-stakes outputs. Consider open-sourcing model summaries or publishing governance reports to build public trust. Establish a human-in-the-loop (HITL) mechanism, especially for decisions impacting individuals (e.g. hiring, lending, medical diagnosis), to ensure ethical judgment and intervene when the AI's decision seems anomalous or harmful.

Log all AI decisions and risk assessments for regulatory audits.

Logging isn't just for debugging. Now, it's a regulatory safeguard. Implement detailed, immutable audit logs that capture every significant AI decision, including input, model version, output, confidence score, and human intervention. Use tools like MLflow or Weights & Biases to track model runs and lineage. For risk assessments, ensure each deployment or model update includes a structured evaluation stored in an internal GRC platform. This record can serve as evidence during regulatory inspections or internal reviews and can be paired with incident response plans to ensure traceability in case of harm or non-compliance.

Ensure external AI vendors comply with required documentation and risk mitigation frameworks.

If you're integrating third-party AI services (e.g., vision APIs, LLMs, and analytics engines), treat them as part of your extended supply chain. Require vendors to provide documentation that matches your internal standards: proof of data provenance, fairness evaluations, security certifications (e.g. ISO/IEC 27001), and compliance with frameworks like the OECD AI Principles or EU AI Act. Build these requirements into procurement policies and conduct regular audits or assessments of vendor practices. If you use SaaS models or APIs, verify if vendors support transparency mechanisms like System Cards or responsible disclosure programs. Establish an exit strategy if a vendor fails to meet ethical or compliance standards.

Software Supply Chain Security

Generate and maintain SBOMs for all software releases.

A Software Bill of Materials (SBOM) is a critical artifact that lists all components, dependencies, and libraries in a software release. SBOMs enable faster vulnerability detection, streamline incident response, and are increasingly becoming a regulatory requirement (e.g., US Executive Order 14028). Use tools that produce SBOMS in Syft, CycloneDX, or SPDX format to automate SBOM generation at build time, integrating into CI/CD pipelines. Maintain versioned SBOMs alongside software artifacts and publish them internally (or externally when required) to support transparency, compliance, and rapid triage during widespread

supply chain vulnerabilities like Log4Shell.

Treat SBOMs like the audit documents they are and be prepared to have many for any particular release. Each stage of the process will need an SBOM, as the eventual comprehensive uber SBOM does not yet exist. You need to keep each one and tie them together.

Use approved repositories and conduct automated security audits.

Restrict all software builds to pull dependencies only from vetted, policy-compliant artifact repositories. This prevents inadvertent use of malicious or outdated libraries. Additionally, implement continuous security audits using tools such as OWASP Dependency-Check, Trivy, or Snyk. These tools should be part of your CI/CD pipelines to flag issues early and enforce fail-fast mechanisms when detecting high-severity vulnerabilities. Use policy-as-code (e.g., Open Policy Agent) to automate enforcement of repository and audit standards across projects.

Ensure you understand the capabilities of the audit tools. You may need more than one to cover the tools , technologies and libraries being used or shipped.

Follow secure development standards (NIST SSDF, OWASP, ISO 27034).

Adhering to formal secure development frameworks builds long-term software assurance. The NIST Secure Software Development Framework (SSDF) provides a high-level guide to integrating security across the SDLC. Similarly, OWASP's Secure Coding Practices and ISO/IEC 27034 offer guidance for aligning software security with enterprise governance. Use these standards to define secure coding checklists, enforce peer review criteria, and guide security training programs. Document your organization's alignment to these standards and conduct regular maturity assessments to identify gaps and drive improvement initiatives.

Document supplier security attestations and vulnerability management processes.

Every third-party software supplier should provide a clear attestation

of their security posture, ideally through standardized forms like the OpenSSF Supplier Declaration of Conformance or SLSA levels. Collect and store these attestations in a centralized, auditable repository, and map each supplier to the products or services they impact. In parallel, maintain an internal vulnerability management policy that defines how vulnerabilities are tracked, triaged, remediated, and disclosed (if applicable). Integrate this with ticketing systems and ensure audit trails exist for every high-severity finding and its resolution.

Implement software composition analysis (SCA) tools to detect and remediate vulnerabilities in dependencies.

Software Composition Analysis (SCA) tools scan your project's dependencies to identify known vulnerabilities and licensing issues. Integrate 3rd party tools or open-source tools like OSS Review Toolkit (ORT) into your development workflow to automate alerts and remediation PRs. Most SCA tools support policy enforcement, allowing teams to block deployments with critical CVEs or disallowed licenses. To maximize impact, integrate SCA findings into developer dashboards and prioritize fixes based on exploitability and usage context. Make remediation progress part of your security OKRs or engineering KPIs to ensure accountability.

Like audit tools, SCA tools can have different scopes and levels of sophistication. Ensure you understand the capabilities and limits of the SCA tools you choose. Picking a less capable one may save money initially but may place you in a problematic situation if vulnerabilities are found externally via a published SBOM analysis by a 3rd party. Shipping code with no known vulnerabilities is a key goal of cybercrime regulations.

Next Time

Read part 4 to conclude this series. There's much left to unpack, including cybersecurity and incident reporting requirements, geopolitical compliance, and liability management.

> Back to Table of Content

#JAVAPRO #SECURITY

Move Fast, Break Laws: AI, Open Source and Devs (Part 4)

Author:

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert.
Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.

https://www.linkedin.com/in/noregressions/

The software development landscape is rapidly changing, with legislation emerging as a key driver of industry trends. As our reliance on software and AI grows, so does our vulnerability to cybercrime, which is now a multi-trillion-dollar problem. This has caught the attention of regulators worldwide.

This article series explains the various regulatory efforts in play and summarises actions that developers and executives should consider as they prepare for 2025, the year of software legislation.

[Part 1](#) covered the background, what a software supply chain is and thoughts on AI and open source.

[Part 2](#) explored how governments are working to create legislation and what the current status is.

[Part 3](#) offered both a Software supply chain and an AI governance & compliance checklists for developers and executives to consider

Part 4 (this article) will discuss cybersecurity and incident reporting requirements, examines geopolitical compliance and liability management, and wraps up the series.

There's a lot to take in. I hope you're sitting comfortably.

Accountability Cannot be Outsourced.

I am not a lawyer. This document is a technical view of the legislation and regulations being developed or repurposed. It's imperative to get your own legal assessment when deciding if these elements apply to your situation. Having said that, some aspects are shared. The primary one is accountability. There's no dodging your responsibilities. That means wherever you are in the software supply chain, you have responsibilities to those consuming your software and those using it. Regulations collectively require organisations to assess, monitor, and manage third-party risks, and you'll have to prove that you did the right thing at the right time.

Blaming others without proper due diligence and safeguards is not a valid defence!

The following sections list commercial tools. They are for example only and do not constitute recommendations or endorsements.

Cybersecurity & Incident Reporting

Implement forensic-grade logging and monitoring.

Your logging and monitoring must meet forensic-grade standards to support post-incident investigations and compliance obligations. Logs should be tamper-evident, timestamped with synchronized time sources (e.g., via NTP), and stored securely with access controls and retention policies. Capture logs from all layers (application, infrastructure, authentication systems, APIs, and user interactions, etc) and enrich them with context (e.g., user ID, IP, session ID). Use centralized logging platforms like Elastic Stack, Datadog, or Splunk and integrate with SIEM tools for real-time threat detection. Protect logs from modification and ensure they are retained per your regulatory obligations (e.g., PCI-DSS, HIPAA, GDPR). Wazuh is a good starting point for building a forensic-grade logging and intrusion detection system for open-source teams.

Establish and test incident response plans.

An incident response plan (IRP) is your playbook for responding to security incidents efficiently and legally. It should define roles (e.g., incident commander, communications lead), escalation paths, forensic evidence handling, containment and recovery procedures, and post-mortem workflows. Base your IRP on established frameworks like NIST SP 800-61 or ISO/IEC 27035. Test your plan regularly to ensure readiness. After each test or real-world incident, perform a structured lessons-learned review and update the plan accordingly. Store the plan in an easily accessible location and ensure key team members can activate it quickly, even outside office hours.

Ensure regulatory compliance with mandatory breach reporting timelines.

Different jurisdictions impose specific timelines for breach notifications, and failure to comply can result in steep penalties. For instance, GDPR mandates breach reporting to authorities within **72 hours**, while U.S. states have their own timelines, and SEC cyber rules may require public disclosure within **four business days**. Maintain a regulatory mapping

document that details breach notification timelines across all applicable jurisdictions where your organization operates. Your legal and compliance teams should be looped into the incident response process early to determine reportability and notification obligations. Use automated triggers in your IRP to initiate legal review the moment a suspected breach crosses severity or data exposure thresholds.

Develop and document processes for rapid regulatory notification and containment of security incidents.

Speed matters during a breach, and regulators expect evidence of preparedness. Create a documented playbook for notifying regulators and affected stakeholders, including pre-approved communication templates, escalation paths, and designated contact points. Map out decision trees for when to notify which regulator, customers, and law enforcement. Include templates for initial notifications, progress updates, and final reports, and ensure they're stored in a secure but easily accessible location (e.g., within your incident response dashboard or GRC tool). Establish a cross-functional breach response team involving legal, PR, IT, and engineering, and rehearse multi-stakeholder simulations. Consider leveraging tooling like Drata, Vanta, or GRC platforms to centralize incident documentation and reduce time-to-report.

Geopolitical Compliance and Liability Management

Screen all customers against export control lists.

Before onboarding customers, especially in international markets, screen them against government export control lists such as the U.S. Department of Commerce Entity List, OFAC sanctions list, and relevant EU or UK restrictions. Use automated tools like Descartes Visual Compliance or LexisNexis Bridger Insight to integrate these checks into your customer onboarding process. Document the outcome of each check to maintain a compliance trail and flag any entities located in embargoed regions or under restricted technology access. Failing to do this can result in severe penalties, export license revocation, or reputational damage.

Implement regional data hosting options to comply with localization laws.

Data localization laws in countries like China, Russia, India, and parts of the EU (e.g., GDPR) require that certain types of data (especially personal or sensitive information) be stored and processed within national borders. Offer customers regional data hosting choices using cloud providers like AWS Local Zones, Azure Regional Services, or Google Cloud's data residency controls. Use GDPR-compliant processors and ensure contracts include standard contractual clauses (SCCs) or binding corporate rules (BCRs) as needed. Maintain a data residency matrix and map workloads accordingly to ensure legal and contractual compliance.

Track and document all software dependencies to ensure compliance.

Managing third-party software usage, both open-source and proprietary, is essential for legal and security compliance. Maintain a real-time, searchable inventory of all components using tools like FOSSA, Snyk, or OWASP Dependency-Track. This helps you comply with licensing terms (e.g., GPL, AGPL), avoid IP infringement, and meet disclosure obligations (e.g., via SBOMs). Include each dependency's version, source, license, and usage context in your records. Ensure license policy enforcement (e.g., banning copyleft licenses in commercial products) and run automated scans during CI to flag violations before code merges.

Ensure compliance with cybersecurity trade restrictions, including bans on foreign software use in critical sectors.

Governments are increasingly restricting the use of foreign-developed software in sensitive sectors such as defence, finance, and energy. For instance, the U.S. bans certain Chinese-developed software and apps from federal systems, and the EU has rules regarding telecom infrastructure and data sovereignty. Regularly audit your software stack, including infrastructure and build tools, for foreign-origin components that may trigger compliance issues. Maintain an internal software classification by origin and criticality and assess against regulations like the U.S. Federal Acquisition Regulation (FAR) or the EU Cybersecurity

Act. Seeking legal counsel and obtaining alternative domestic solutions or certified suppliers where necessary.

Prepare internal risk assessments for emerging regulations on AI liability.

AI liability legislation is rapidly evolving. The EU AI Act introduces obligations for risk categorization, documentation, and human oversight, while U.S. and UK proposals are increasingly focused on accountability for harm. Prepare for these shifts by conducting internal risk assessments of your AI systems, focusing on model use cases, decision impact, and failure modes. Use frameworks like NIST AI RMF and tools like AI Explainability 360 to assess governance and control measures. Maintain a register of high-risk use cases and define policies for auditing, escalation, and responsible deprecation of AI systems that pose liability risks.

Educate engineering teams on evolving software liability laws to reduce legal exposure.

Legal frameworks are starting to hold software producers accountable for insecure or harmful code, especially in safety-critical and AI-driven systems. Host regular legal briefings or brown-bag sessions with your legal team to inform engineers of relevant changes like the EU Cyber Resilience Act or proposed U.S. rules around software liability. Develop training programs or onboarding modules that cover secure coding, licensing compliance, and responsible AI development. Emphasize the concept of engineering defensibility: the idea that your codebase, process, and documentation should hold up to external scrutiny in case of a breach or failure.

Develop compliance roadmaps to keep up with regulatory changes over time.

Staying compliant is not a one-time effort; it's an evolving strategy. Assign responsibility to a cross-functional team (legal, security, DevOps, product) to monitor emerging tech regulations globally and update internal policies accordingly. Use a compliance roadmap to plan upcoming changes, such as adopting new standards (e.g., SBOM formats), implementing localization controls, or adapting to new AI

liability rules. Tools like OneTrust, TrustArc, or even a simple roadmap tracker in Confluence or GitHub can help. Communicate this roadmap across teams so engineering and product planning can anticipate regulatory needs in advance.

Wrap Up

There's a lot to unpack, and we've barely started. The clock is ticking, though, and it's time to start acting. For any significant endeavour, you'll need legal advice. From an open-source project POV, you could do nothing, and apart from potentially losing users, there's little sign that you have any of these obligations. The requirements will continue to grow for everyone else involved in providing software services or products, and the open-source projects that rise to the challenge will gain even more adoption.

However, weaving through the legal considerations is concern over harm. Vulnerable software has often had the potential to cause actual harm to people, and some recent cyber-attacks have played on that fear - or even been capable of inflicting it.

Be alert: as laws evolve, the old assumption that open source == no liability is being challenged, particularly for AI tools that generate content or make decisions. Those using or building on generative OSS tools may soon face new legal expectations, compliance demands, or shared responsibility for harm.

Remember, accountability cannot be outsourced - for commercial organisations and potentially for some OSS tools:

- **Your Responsibility:** You remain liable when using third-party software or AI models.
- **Due Diligence Required:** Regulations mandate a thorough assessment of all components.
- **No Blame Shifting:** You can't claim "not my code" when vulnerabilities impact or harm users.

Using third-party code or models does not transfer legal responsibility. Your organization must ensure that all components meet regulatory requirements.

> Back to Table of Content

#JAVAPRO #DEVOPS

Put Events in the Driver Seat to Manage Your Java Anywhere

Author:

Romain Pelisse works at Red Hat for over a decade. He started as runtimes consultant, building on expertise on JBoss EAP (WildFly), and moved to engineering where he became the lead of the Ansible runtimes initiative, focusing on providing the best integration possible between Red Hat middleware solutions and Ansible Automation Platform.

https://www.linkedin.com/in/rpelisse/

Managing a multi-datacenter VM based or an Edge server executing Java applications is not a simple task and can become even more challenging as the complexity of the hosted app grows. If a pristine installation runs without a hitch, after a while, recurring problems appear and what is called nowadays “Day One” operations take the front seat. Obviously, those are the struggles facing any modern IT infrastructure, and while the target could not be more clear and defined (a functional system with zero or minimum downtime), yet the crucible, how to achieve it, may not be evident.

Regardless of the actual implementation, the key here is automation. Especially in the context of large and complex Java deployments which

often requires the capability to set up software, configure them properly and maintain up to date the underlying infrastructure (operating systems, databases, and so on), but also to be able to effectively interact with the Java Virtual Machines (JVM) executing the applications.

This is the broad and intriguing topic we are going to discuss in this article. How to build an efficient, and modern infrastructure for Java-based systems, in a fully automated fashion, yet also designed to keep running on its own, and thus imbued with the capacity to react to changes and autocorrect, without manual (or minimum) intervention.

Ansible To The Rescue?

Ansible in a Few Words

As we are going to demonstrate in our article, Ansible, an Open Source, Python-based product for automation, is an appropriate fit to address all the concerns and requirements we have just mentioned. It might feel like an odd pick, a non-Java-based ware to manage Java deployments, yet, as we'll see, the software flexibility, features and extended ecosystem makes it a perfect solution.

Before going any further, it's important to note that Ansible does not mandate Python skills and is, as Java, **cross-platform**. Also, Ansible has an **agentless** architecture, which means that, apart from the controller (the machine where Ansible is executed) no other software needs to be installed and managed. And, if *Ansible Automation Platform* is used (see the last section of the article), the appropriate Python environment is provided and administered by the product.

How Does Ansible Fit the Bill?

As we are going to demonstrate in our article, Ansible, an Open Source, Python-based product for automation, is an appropriate fit to address all the concerns and requirements we have just mentioned. It might feel like an odd pick, a non-Java-based ware to manage Java deployments, yet, as we'll see, the software flexibility, features and extended ecosystem makes it a perfect solution.

Before going any further, it's important to note that Ansible does not mandate Python skills and is, as Java, cross-platform. Also, Ansible has an agentless architecture, which means that, apart from the controller (the machine where Ansible is executed) no other software needs to be installed and managed. And, if Ansible Automation Platform is used (see the last section of the article), the appropriate Python environment is provided and administered by the product.

Vocabulary note: With Ansible, the system executing the tool is referred to as the controller, which is often opposed to the targets, the hosts that Ansible manages. Keep in mind this distinction for the rest of the articles. Also note that the targets can be bare metal, VMs or even run in a private or public cloud.

Contrary to other options employing agents, and thus requiring it to run on the target's system, Ansible simply leverages SSH (or a similar authentication mechanism) to communicate with the target hosts. No Python code is ever executed on any system managed by Ansible, nor is any kind of Python setup necessary on the targets (at least, most of the time).

Ansible Playbook

To define the automation needed, Ansible uses YAML descriptors, not code. Those files, called playbooks, are not scripts per se, and utilize a syntax specific to Ansible, to depict the state the target systems are expected to be in along with the steps required to achieve it.

Below is a simple example of a playbook designed to ensure that nginx, a popular Open Source HTTP server, is installed and running on the targets:

```
---
- name: "Ensure nginx is installed and running"
  hosts: webservers
  tasks:
    - name: "Ensure nginx package is installed"
      ansible.builtin.package:
```

```
        name: nginx
        state: present
      notify: "Restart Nginx"

    - name: "Ensure nginx is properly configured"
      ansible.builtin.template:
        src: nginx.conf.j2
        dest: /etc/nginx/nginx.conf
      notify: "Restart Nginx"

  handlers:
    - name: "Restart Nginx"
      ansible.builtin.service:
        name: nginx
        state: restarted
```

As shown by the playbook above, even without knowing the syntax used by Ansible, it's easy to comprehend. The header of the playbook specifies its name, the targets (designated by a group name) and then the tasks to be performed. Each required operation is specified by using an Ansible module (such as `ansible.builtin.package`). The purpose of each of them is easy to understand: package ensures the required software packages are available, service manages the state of the system's services and template deploys a configuration (based on a template, we'll discuss this in a bit more detail below).

Notification and Handlers

The only item that requires some explanation is the use of the attribute notify and the section called `handlers`. In short, Ansible has an internal **notification system** allowing one module to notify a handler. This is mostly used when a change triggers other operations on the target system. Most of the time this mechanism is utilized to ensure that the service associated with the software being deployed is started (or restarted to take a config change into account).

In our example, both the use of `package` and `template` triggers such notification. Indeed, if the package has to be installed, nginx is certainly

not running on the system, so it has to be started. If its configuration file needs to be updated, then the service also needs to be restarted (or at least reloaded) so that the changes are applied.

Overall this playbook ensure that the state of the target is as follow:

1. The nginx package has been installed (prerequisite)

2. The appropriate configuration is deployed (configuration)

3. The nginx service is running (state)

Most of the time, the organization of a playbook will follow the same three high-level steps. Each might be more complex depending on the solutions to manage, but in the end, most playbooks ensure that prerequisites are in place, that configuration is properly deployed and that the appropriate software is running.

Templating with Jinja2

Note that, for the second step, the module used is `ansible.builtin.template.` Indeed, Ansible utilizes a template engine (named Jinja2) that allows his playbook to generate the correct configuration file, based on the specificity of the target system.

Below is an extract of the template employed in the example above:

```
…
   server {
        listen       {{ ansible_all_ipv4_addresses }}:80;
        server_name  {{ ansible_nodename }};
        root         /usr/share/nginx/html;
…
```

A Jinja2 template employs a few simple primitives to replace the dynamic parts of the file by the content of variables managed by Ansible. Here again the syntax is pretty straightforward. In the example above nginx's bind address will be the one of the target (by default, Ansible has such information on any target). Same approach applies for the server name, it is set to match the name used by Ansible for this particular host (by

default, it is its hostname).

There is obviously more to understand about playbooks and their capabilities, but this brief overview is enough for the reader, new to Ansible, to comprehend the rest of this article.

Before we look at why Ansible's features and capacities make it an excellent tool to deploy and maintain Java software, let's talk about its extension mechanism. Like any automation technology, Ansible functionalities can be expanded by adding specific plugins. In Ansible's parlance, those extensions are called collections. We'll see how to install and use such plugins in the next section.

How Can Ansible Manage My Java Deployment?

Extending Ansible Capacities Using Collection

All of the above points made Ansible a compelling solution for automation, yet it does not illustrate how it is a particularly good fit to handle Java products. To demonstrate that, we'll now show how to implement a similar playbook to the one we just did, but this time, with a Java Product: Red Hat JBoss Web Server.

As we just mentioned, Ansible capacities can be extended by adding collections to the ones used by its controller. It's very much akin to introducing a dependency in `pom.xml` of a Java project built by Maven. And like Maven, just one command line does the job:

```
# ansible-galaxy collection install redhat.jws
Starting galaxy collection install process
Process install dependency map
Starting collection install process
Downloading https://console.redhat.com/api/automation-hub/v3/plugin/
ansible/content/published/collections/artifacts/redhat-jws-2.1.1.tar.
gz to /root/.ansible/tmp/ansible-local-17405g5h0ey9/tmpp71vcsnf/
redhat-jws-2.1.1-0l_n4fph
Installing 'redhat.jws:2.1.1' to '/root/.ansible/collections/ansible_
collections/redhat/jws'
Downloading https://console.redhat.com/api/automation-hub/v3/plugin/
```

```
ansible/content/published/collections/artifacts/redhat-runtimes_
common-1.2.2.tar.gz to /root/.ansible/tmp/ansible-local-17405g5h0ey9/
tmpp71vcsnf/redhat-runtimes_common-1.2.2-b2lbzxjg
redhat.jws:2.1.1 was installed successfully
Downloading https://console.redhat.com/api/automation-hub/v3/plugin/
ansible/content/published/collections/artifacts/redhat-rhel_system_
roles-1.88.9.tar.gz to /root/.ansible/tmp/ansible-local-17405g5h0ey9/
tmpp71vcsnf/redhat-rhel_system_roles-1.88.9-7nbobi3i
Installing 'redhat.runtimes_common:1.2.2' to '/root/.ansible/
collections/ansible_collections/redhat/runtimes_common'
redhat.runtimes_common:1.2.2 was installed successfully
Installing 'redhat.rhel_system_roles:1.88.9' to '/root/.ansible/
collections/ansible_collections/redhat/rhel_system_roles'
Downloading https://console.redhat.com/api/automation-hub/v3/plugin/
ansible/content/published/collections/artifacts/ansible-posix-
2.0.0.tar.gz to /root/.ansible/tmp/ansible-local-17405g5h0ey9/
tmpp71vcsnf/ansible-posix-2.0.0-i06rpbpn
redhat.rhel_system_roles:1.88.9 was installed successfully
Installing 'ansible.posix:2.0.0' to '/root/.ansible/collections/
ansible_collections/ansible/posix'
ansible.posix:2.0.0 was installed successfully
```

Again, in a similar fashion to Maven, the ansible-galaxy tool ensures that not only the collection for JWS is installed, but also its dependencies.

Setting Up JWS with Ansible

Now that we have the collection installed, let's look at the implementation. Our goals are the same as with nginx. We want to set up, configure and run as JWS. As a configuration change, we'll have a specific version of Java and, like before, bind the server against the default external IP of the target system.

Because the collection comes with a ready-to-use playbook, we don't even need to write up one. We can simply run the one provided, overriding a few default variables to achieve our goals:

```
$ ansible-playbook -i inventory redhat.jws.playbook -e @service_
account.yml -e jws_java_version=21 -e jws_listen_http_bind_
address="{{ ansible_all_ipv4_addresses[0] }}"
```

The use of @ in front of a filename allows passing all the variables defined in it to Ansible. In this case, we are employing one named `service_account.yml` to provide the tool the credentials required to download the software (JWS) from the Red Hat Customer Portal. Note that, by default, it's always the latest version available (latest version of JWS 6 as of the writing of this article).

Check the Results

At the end of this playbook execution, we can check that JWS is installed and running, simply by leveraging the systemctl command on its service on the target. That can be performed directly by Ansible using a feature called adhoc command:

```
# ansible -i inventory webservers -a "/usr/bin/systemctl status jws6-
tomcat.service"
localhost | CHANGED | rc=0 >>
● jws6-tomcat.service - Jboss Web Server
     Loaded: loaded (/usr/lib/systemd/system/jws6-tomcat.service;
     enabled; preset: disabled)
     Active: active (running) since Wed 2025-03-05 11:17:45 UTC;
     38min ago
    Process: 2697 ExecStart=/opt/jws-6.0/tomcat/bin/systemd-service.
    sh start (code=exited, status=0/SUCCESS)
   Main PID: 2705 (java)
        CPU: 5.128s
     CGroup: /system.slice/jws6-tomcat.service
             └─2705 /etc/alternatives/jre_21/bin/java -Djava.util.
logging.config.file=/opt/jws-6.0/tomcat/conf/
logging.properties -Djava.util.logging.manager=org.apache.juli.
ClassLoaderLogManager -Djdk.tls.ephemeralDHKeySize=2048 -Djava.
protocol.handler.pkgs=org.apache.catalina.webresources -Dorg.apache.
catalina.security.SecurityListener.UMASK=0027 --add-opens=java.base/
java.lang=ALL-UNNAMED --add-opens=java.base/java.io=ALL-UNNAMED
--add-opens=java.base/java.util=ALL-UNNAMED --add-opens=java.base/
```

```
java.util.concurrent=ALL-UNNAMED --add-opens=java.rmi/sun.rmi.
transport=ALL-UNNAMED -classpath /opt/jws-6.0/tomcat/bin/bootstrap.
jar:/opt/jws-6.0/tomcat/bin/tomcat-juli.jar -Dcatalina.base=/opt/
jws-6.0/tomcat -Dcatalina.home=/opt/jws-6.0/tomcat -Djava.io.tmpdir=/
opt/jws-6.0/tomcat/temp org.apache.catalina.startup.Bootstrap start

Mar 05 11:17:45 e9ba2ac1f2b9 systemd[1]: Starting Jboss Web Server...
Mar 05 11:17:45 e9ba2ac1f2b9 systemd-service.sh[2698]: Tomcat
started.
Mar 05 11:17:45 e9ba2ac1f2b9 systemd-service.sh[2697]: Tomcat runs
with PID: 2705
Mar 05 11:17:45 e9ba2ac1f2b9 systemd[1]: Started Jboss Web Server.
```

We can use the same feature to get more information on the version of JWS installed, and thus confirmed that the collection did fetch the latest available version:

```
# ansible -i inventory webservers -a "cat /opt/jws-6.0/version.txt"
localhost | CHANGED | rc=0 >>
Red Hat JBoss Web Server - Version 6.0 GA

Java Components:
Apache Tomcat 10.1.8-4.redhat_00011.1.el8jws
Tomcat Vault 1.1.9-2.Final_redhat_00002.1.el8jws
JBoss mod_cluster 2.0.3-4.Final_redhat_00001.1.el8jws

Native Components:
Apache Tomcat Native 1.2.36-1.redhat_1.el8jws
```

Notes Regarding the edhat.jws Ansible Collection:

The collection we just installed is a certified Ansible content provided by Red Hat for its runtime product Red Hat JBoss Web Server, a servlet engine based on Apache Tomcat, fully supported by the company. To obtain such an extension, and benefit from the software vendor support, one needs a valid subscription for both Ansible Automation Platform and JBoss Web Server. However, as always with Red Hat products, there is an upstream and community version, not supported, that can be installed and used without a subscription:

```
$ ansible-galaxy collection install middleware_automation.jws
```

Content-wise, these two collections are almost identical. One uses by default the community product (Apache Tomcat), the other one the downstream version supported by Red Hat (JWS). Their functionalities differ only where the underlying products do. For instance, the downstream version can update JWS by applying a patch provided by Red Hat. This feature has no equivalent with Apache Tomcat.

What About Automatic Remediation?

Event-Driven Ansible

What we have demonstrated so far is the capability of Ansible to work smoothly with java solutions such as Red Hat JBoss Web Server to ensure that the target systems are in the expected state and from there maintaining this set-up over time (by running the playbook regularly). However, we have yet to tackle the infrastructure's reactiveness we described at the beginning of this article. We want to have automation also able to cope with incidents and execute, consequently, appropriate remediation strategies.

Ansible has a powerful feature, recently introduced, to deal with such use cases. It's called Event-Driven Ansible (EDA). This functionality run playbook if triggered by the detection of some specific events. Under the hood, EDA utilizes an Open Source Java software named Drools. It's a Business Rules Management System (BRMS) including a core Business Rules Engine (BRE), which enables EDA to define rules based on events. Yet another proof that Ansible works smoothly with the Java ecosystem.

EDA in Action

Let's see a concrete example of how to automate a remediation strategy in the context of Java software deployment. For this, we'll describe a simple use case, but also utilize a more elaborated Java solution, the Red Hat JBoss Enterprise Application Platform (EAP) for our demonstration. Here again, to install and run this software we'll leverage the Ansible Collection for EAP provided by the vendor.

```
# ansible-galaxy collection install redhat.eap
Starting galaxy collection install process
[WARNING]: Collection redhat.jws does not support Ansible version
2.15.12
Process install dependency map
Starting collection install process
Downloading https://console.redhat.com/api/automation-hub/v3/plugin/
ansible/content/published/collections/artifacts/redhat-eap-1.5.7.tar.
gz to /root/.ansible/tmp/ansible-local-3004po7xx5f4/tmp3jum19hl/
redhat-eap-1.5.7-moqnbw2b
Installing 'redhat.eap:1.5.7' to '/root/.ansible/collections/ansible_
collections/redhat/eap'
redhat.eap:1.5.7 was installed successfully
'ansible.posix:2.0.0' is already installed, skipping.
'redhat.runtimes_common:1.2.2' is already installed, skipping.
'redhat.rhel_system_roles:1.88.9' is already installed, skipping.
```

Do note that we have installed this new collection on the same Ansible controller, so the dependencies shared with the redhat.jws already available have not been fetched again.

Setting Up The Cluster

Now, let's discuss our use case. Our infrastructure manages with Ansible a large JBoss EAP deployment. Each instance hosts several applications, which synchronize their states thanks to the Java/Jakarta server's feature of clustering. To emulate such an environment, we are going to provision EAP with the `standalone-full-ha.xml` configuration.

Again, the playbook supplied with the Ansible Collection for EAP does the heavy lifting, we just need to specify the configuration we wish to set up all the machines targeted to be part of the cluster:

```
$ ansible-playbook -i inventory redhat.eap.playbook -e eap_
bind_addr="{{ ansible_all_ipv4_addresses[0] }}" -e eap_config_
base=standalone-full-ha.xml -e @service_account.yml
```

Now that we have a functioning array of EAP instances and an active cluster, let's discuss our use case for EDA. This cluster is dedicated to a back end application utilizing a data source. The app has been in production for years, but since the last update, there is a recurring issue. Some connections in the database pool are not being properly returned and remain unusable, in an idle state.

Implementing The Remediation With EDA

The developers managed to create a ReST endpoint to report the problem, but have yet to find a fix. A simple workaround, however, is to leverage JBoss CLI to flush EAP's idle connections. The following rulebook automate this remediation strategy:

```
---
  - name: EDA lab
    hosts: all
    sources:
      - ansible.eda.url_check:
          urls:
            - http://backend.example.com:8080/app/rest/db_healthcheck
          delay: 10
    rules:
      - name: Restart Wildfly if url is not reachable
        condition: event.url_check.status_code == 500
        action:
          run_playbook:
            name: playbooks/flush_idle_connections.yml
```

Here again the syntax employed by Ansible is easy to understand. The rulebook is basically divided in two sections. The first one, sources, as it names implies, lists event sources consumed by EDA. In our case, we are utilizing a built-in primitive, `url_check`, which allows EDA to poll a URL and use the response as an event.

The second portion, titled rules, defines how to react to specific events. In our example, if the rest point returns an internal server error, it means that the number of unemployed connections in the database pool is becoming dangerously high. If this is happening, EDA will then trigger

a playbook called `flush_idle_connections.yml` responsible for flushing the idle connections of all the instances of the cluster:

```
---
- name: "Flush idle connections of {{ ds_name }}"
  hosts: all
  vars:
    ds_name: "BackendDS"
  collections:
    - redhat.eap
  tasks:
    - name: "Flush idle connection {{ ds_name }}"
      ansible.builtin.include_role:
        name: eap_utils
        tasks_from: jboss_cli.yml
      vars:
        jboss_cli_query: "/subsystem=datasources/data-source={{ ds_
        name }}:flush-idle-connection-in-pool()"
```

The playbook above leverages the `jboss_cli.yml` provided by the Ansible Collection for EAP. It takes care of all the required configuration to connect to the remote instance and execute a JBoss CLI query.

Note: A nice benefit of using Ansible is that the remote instance does not need to expose the EAP management port (used by JBoss CLI). Indeed, Ansible will instead execute the query directly on the host (so the EAP server address will be localhost).

Ansible Automation Platform

An Automation Control Center

To define processes and automation, the high-level abstraction provided by Ansible's s is more than appropriate. It's a clean design that works well with a source control system (such as Git), which makes it a perfect tool for DevOps purposes.

However, these, on their own, are not able to give a real overview of a

massive and complex infrastructure. And this is where Red Hat Ansible Automation Platform (AAP) comes into the picture. This product is conceived to be the control center of the automation and also provides much-needed features for enterprise or large organizations (such as SSO integration or ACL).

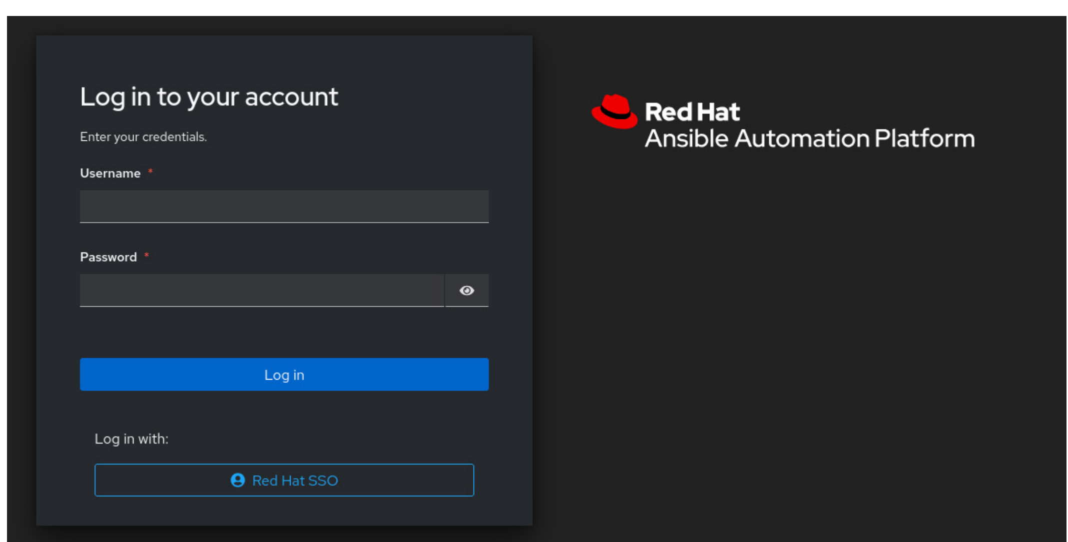
So, what is exactly AAP? It's web UI designed to manage the high-level components of Ansible: organizations, projects (containing playbooks or rulebooks), inventories (dynamic or not) and scheduled jobs (execution of a playbook). It gives a visual overview of the infrastructure, but also enables fine-grained and role-based authorized access to the automation to individuals inside the organization.

For instance, using AAP, a developer may be allowed to trigger a job (a playbook run) only on a specific group of machines, however, they might not be authorized to amend the playbook or the job itself. A contractor working on the database deployment may be permitted to alter the playbooks (by pushing changes into the repository) and execute them, yet only in dry mode (without performing modifications on the target system). And so on.

A Little Tour of AAP

Login Into AAP

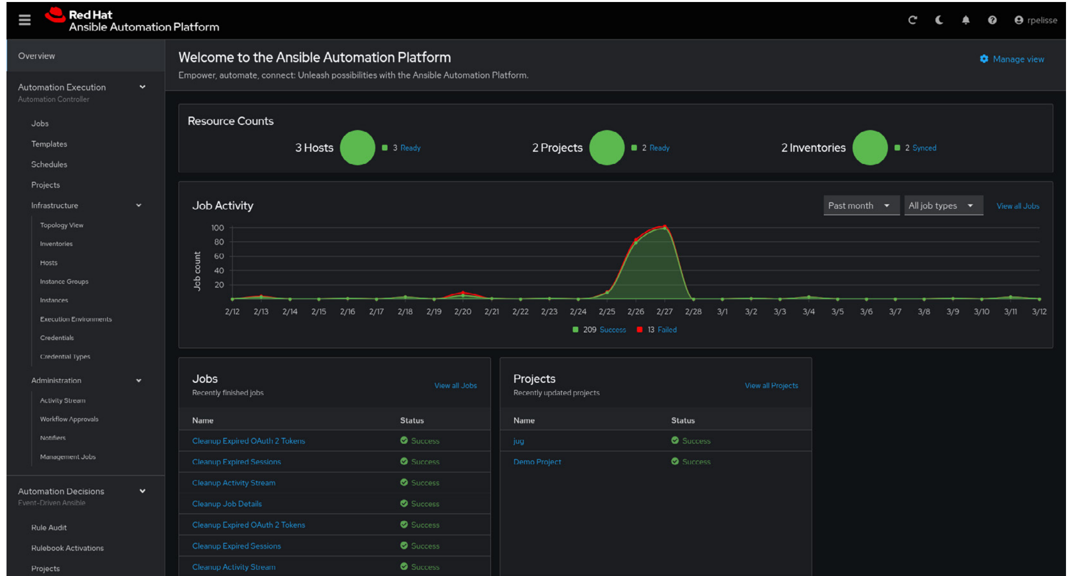
Let's give a quick tour of AAP with a few screenshots to get a better sense of the overview provided by the product. First, we'll begin with the login page:



As shown above, AAP can, of course, integrate with SSO solutions such as Keycloak, and thus the authentication process and the users and roles can be entirely externalized from the tool. If this is needed, AAP can also administer local users and groups.

Landing Page

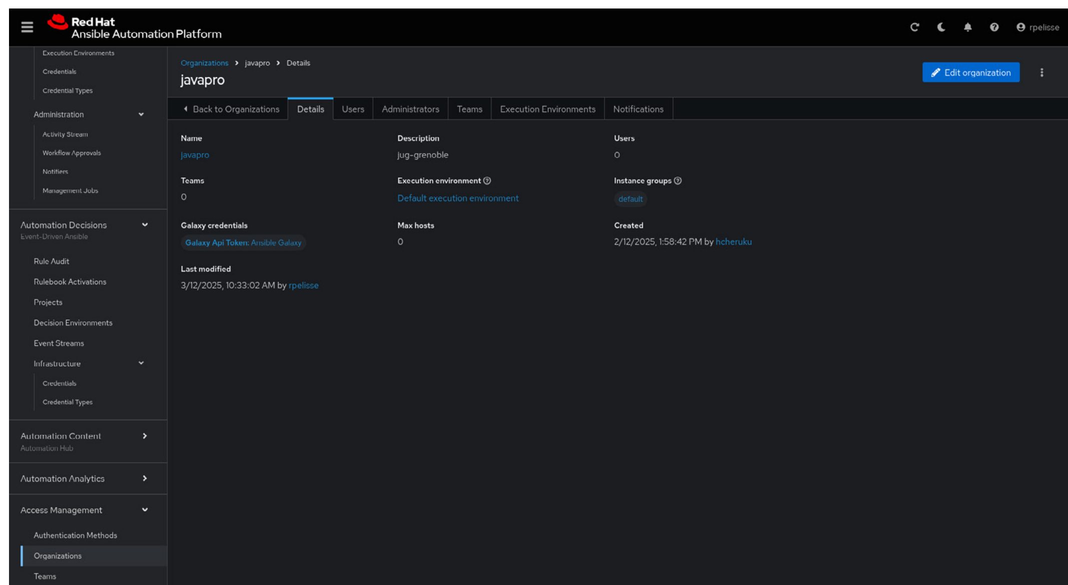
Once logged in, the landing page provides a graphical overview of the infrastructure managed:



Obviously, the screenshot above displays the very low activity of a demo environment. It is not representative of a productive system.

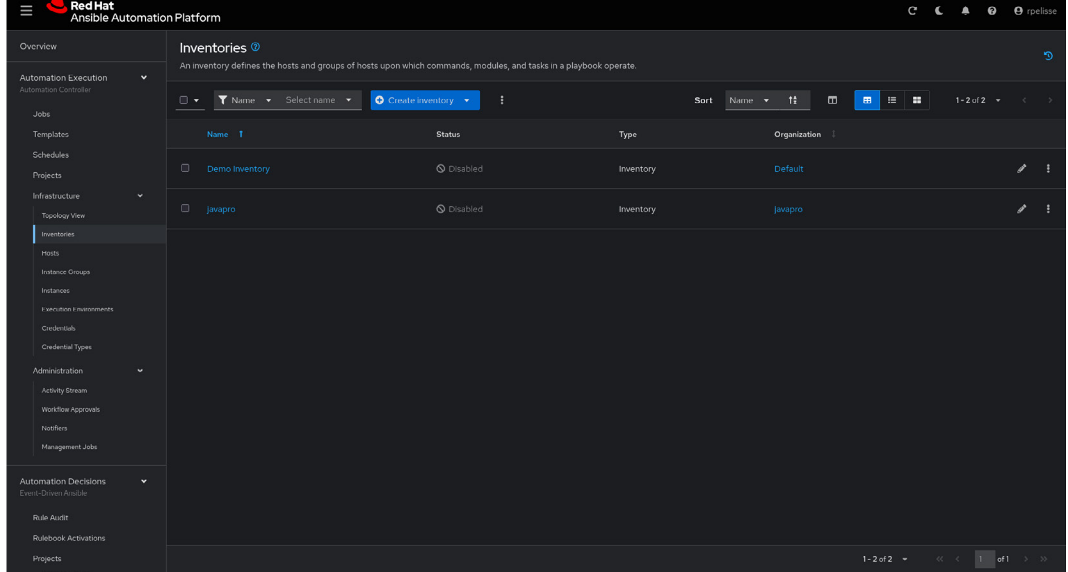
Organizations

As many large groups or enterprises may use the same instance of AAP for different infrastructure, the tool has a notion of “organization.” Anything related to one can be hidden and isolated from the others. This allows AAP to manage, together, completely distinct projects and deployments.

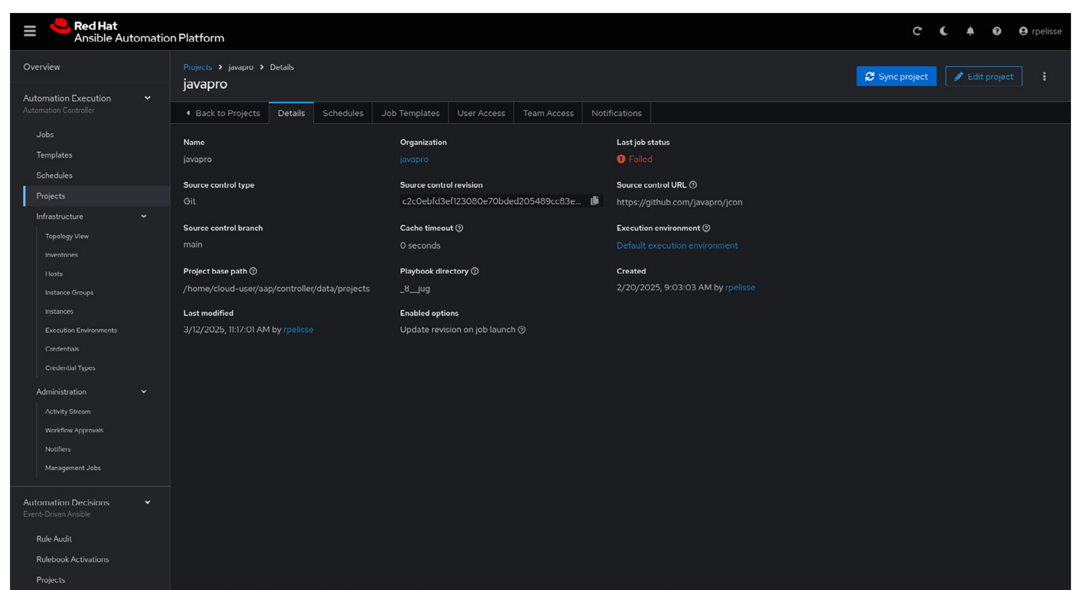


Inventories

For AAP to know which systems it needs to manage, an inventory must be provided:



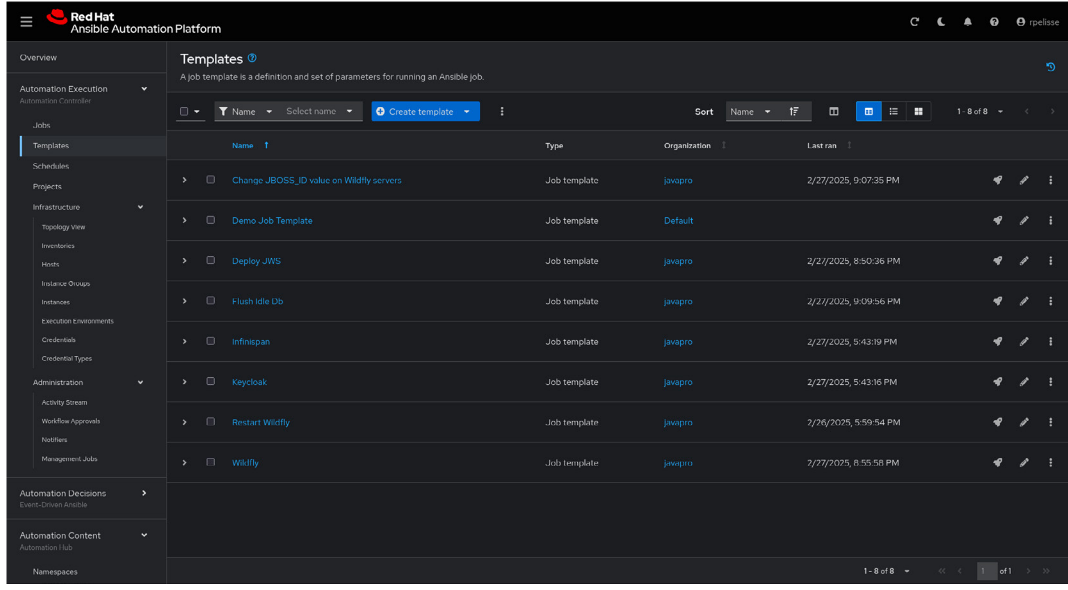
Provisioning the inventory can be done statically or dynamically. In any case, once this is done, the hosts tied to it and listed:



Ansible's playbooks naturally fit into any kind of source control. Therefore, AAP has no real internal mechanism to manage them. Instead it is handling projects checked out from independent source control systems. All that's needed is to provide the connection information to the external repository:

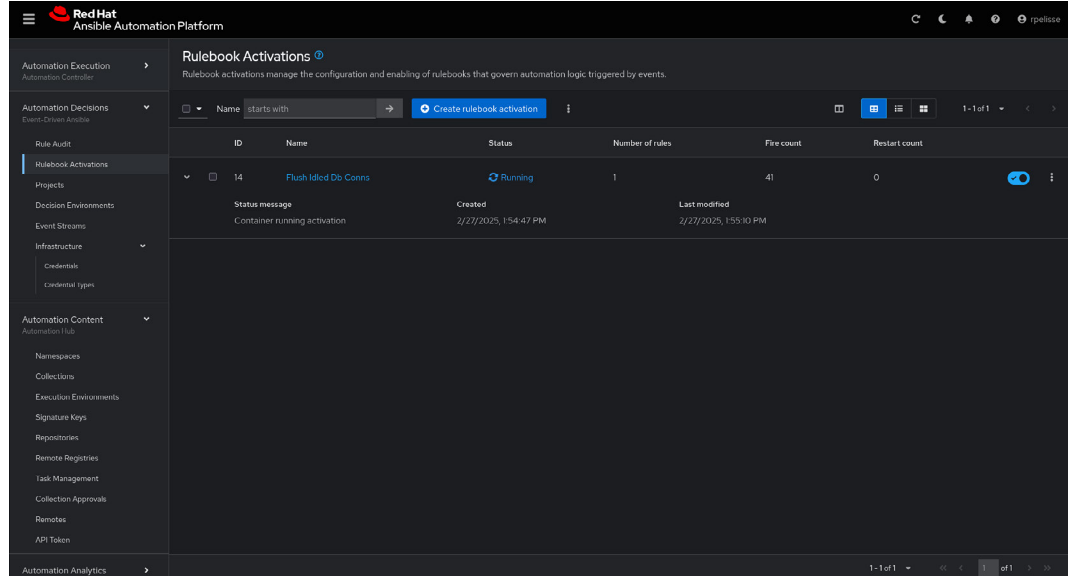
Job Template

Once a project is defined, a job template can be created to run any of the playbooks it contains on the systems of the attached inventory. From there, AAP can directly fire a job (based on the template) or schedule its execution on a regular basis.



Inside AAP, EDA follows a similar general approach. Rulesbooks live in projects that are also attached to organizations and can be associated with inventories. Note that if AAP has two different projects (one for the automation, one for the event-driven remediation), both playbooks and rulebooks can be stored and versioned in the same external repository, if it's more convenient.

Once a rulebook has been properly integrated, AAP will run it on a regular basis and triggers the appropriate job template if needed:



Note that the rulebook needs to be modified to use the job_template primitive rather than the run_playbook one to work properly in the context of AAP.

Summary

Ansible itself is a powerful tool for automation that works well with Java software and requires no extra skills, short of learning the grammar and syntax of its playbooks, rulebooks and templates. With Event-Driven Ansible, it can go even further than automation and management, it can be enhanced to implement remediation of all sorts based on events triggered by incidents.

Adding Ansible Automation Platform on top of these two technologies enables its users to safely oversee massive infrastructure while ensuring

security concerns are fully addressed (access and authorization, ACL, audit trails, ...). Last, but not the least, Ansible has a rich ecosystem and can be easily extended through the utilization of collections, as shown by the ones provided by Red Hat for its (Java) Runtimes products.

All in all, Ansible is an excellent solution to manage one's Java deployment, but also its infrastructure at large and other (non-Java) projects.

> Back to Table of Content

#JAVAPRO #TESTING #QUALITY

Testing Done Right!

Author:

Wouter Bauweaerts is a passionate Java developer at The Beehive. As a full stack developer, he excels at solving complex problems efficiently and always focuses on quality. He is known for his creativity and forward-thinking. He enjoys experimenting with new frameworks and technologies to make the best decisions for future projects. Beyond coding, Wouter is dedicated to coaching and mentoring his colleagues. He helps them improve by discussing project challenges and opportunities. In the past, Wouter has also taught programming courses at Karel de Grote Community College in Antwerp. His practical experience and love for learning make him a valuable asset to his team and the IT community.

https://www.linkedin.com/in/wouter-bauweraerts-938689108/

Oh no! Not another article about testing! You can find a lot of publications about automated testing in software development, but there is a good reason for it. Testing your code is one of the most important aspects in software development!

I'm not going to write yet another article about how you should write your tests. I'm also not covering the details of various types of tests.

I will elaborate on some pitfalls you should be aware of when writing tests and how to solve them. This is not a scientific approach; it is just my opinion. What's covered in this article? Let's start with discussing why testing is so important in software development. Next, I'll elaborate on some risks and how we can avoid them. I will also dive into sociable testing and test data generation.

Will you learn something new from reading this? Maybe, maybe not! But I have one goal in mind while writing this text. I want you to think about how you tackle your testing, so you can grow as a developer!

Over the past few years, I have been teaching courses, coaching colleagues and giving presentations. My focus here? Quality! Quality is not only about testing, but testing is a very important prerequisite to be able to deliver qualitative software. When I was talking to people who attended my presentation, they mostly said "I knew what you were saying, but I didn't know that I was doing it wrong!". So, I hope that you'll benefit from my opinion by thinking about how you can improve your own way of working.

Importance of Testing

As I mentioned in the introduction, testing is a prerequisite to deliver quality. While every developer knows this, I still see that testing is treated very lightly in any programming course. Most courses just slightly touch testing. They introduce testing libraries like Junit, saying that it is helpful to write tests. They give some examples and then just jump into the next topic. There's nothing more to say about testing, right?

When I mentor interns or when I'm coaching a junior developer, I often see them struggling with testing. They often know the basics about how they should be writing their tests, but they don't know when they should write a test or even which test they should be writing.

Why is this a problem? For me that's pretty obvious. My tests are the only way of proving to myself, my team and my customer that the code I've been writing does exactly what I want it to do! It also guarantees that the functionality that I have been writing before still works after adding more features or refactoring the code. That is why we need tests!

Testing Pitfalls

However, I always prefer to have an extensive test suite over having no tests at all, not all tests are equally good. There are some things you should be aware of while writing tests, to ensure that you are writing good tests.

Test First

Let's start easy! You should always try to write your test before you write your implementation. Otherwise, you risk that you're testing the wrong things. We're all human, so we tend to make mistakes.

Let's imagine that we are writing a complex calculation. We started out writing our first test, which should obviously fail. When we start implementing the code, we enjoy the process of implementing the calculation, leading us to writing too much code. At least, more than we should be writing, because we should only write the code we need to make the test pass. Now here's the problem! We make a mistake in our calculation, so without knowing it, we introduce a new bug!

Afterwards, we start adding more tests, in order to cover the entire calculation, including every single edge case. If you're looking at your code to write your tests, you risk testing that the bug exists in the code, as if it should be the way we've implemented it.

It can get even worse! We're developers, so we are lazy! We're also modern developers, so we're using an AI assistant to speed us up. Almost every "self-respecting" AI assistant has a feature to generate unit tests based on a given code snippet, so we ask the AI to write tests for our entire calculation. Why should we write our tests if the AI can do this for us?

Testing The Right Thing!

Another problem is that we're testing the wrong thing. I've had a colleague in the past that wasn't familiar with writing tests. While he was writing a relatively simple feature to keep the clock in the client-side in sync with the server clock, he also had to test this.

He had a simple endpoint retrieving the server time, using LocalDateTime and the Clock class. He was searching the internet how he could solve this, and he ended up just copy-pasting the example test that he found online. The test worked just fine, but he wasn't testing his own code, he was testing that the Java implementation of the fixed clock worked!

There is one more thing I would like to mention. This last issue is caused by how we write our specific tests. Everyone says that a unit test must run in isolation, making it fast and easy to write! This leads to a terrible disease called *Mockitis*. *Mockitis* can be easily diagnosed. When writing a test where you set up doubles for everything, you're mocking too much. *Mockitis* leads to fragile tests, making it hard for us to refactor our code.

We all know that refactoring should be about improving the structure of our code, without changing the behavior. In an ideal setup, we should be able to refactor our code without breaking our tests! Well, welcome to the real world! However, libraries like Mockito can be particularly useful, they can also slow us down when not using them with caution. We need mocks at some points in our tests but be aware of the risk that your tests rely on your specific implementation.

So, if your tests break because you're refactoring, know that your tests were testing the technical implementation instead of the real behavior of the code!

The Solution: Focus Shift

I have read in many publications that test driven development is the solution for most problems with testing. I think this can be a very good place to start, but if you want to skyrocket your quality, I think it's not quite good enough. I'm not preaching that we should all abandon test driven development, not at all! I think we should be adding another layer on top of it!

Test driven development helps us to think about our code before we are writing it, but it does not specify how to write tests and which tests to write. I have seen many introductions to TDD in the past, where the focus is only on writing unit tests. The examples are often too easy, so

easy that you cannot do it wrong. There is no complex problem for you to solve, you don't really need to think about the architecture. No, you just must write the test before you write the code.

I am not saying TDD is bad, because I know it is not. It is particularly important to think about what you should be implementing before you start implementing it. The easiest way to do this is to write a test first.

Then what am I promoting? I think we should start focusing more on testing the behavior of our code instead of the actual implementation. We want to know that our code produces the results we want it to. I do not really care about how the code gets to this result, because this is highly likely to change over time.

Focus on Behavior

Over the past few years, I have become a huge behavior-driven development enthusiast. For many teams, behavior-driven development feels like overkill. But they could not be more wrong. Why do they think about BDD this way? If you do a quick search for BDD online, you will find cucumber listed remarkably high. Cucumber is one of the tools that facilitate the use of BDD in software projects. I am not going to say that you must use Cucumber (or any other tool) to be able to apply some of the best practices of BDD!

BDD is not a library or a framework that you need to depend on, no BDD is a way of working. Where TDD states that software development should be driven by writing tests, BDD prefers that the desired behavior is your primary driver. What are the expectations we need to fulfill with the features we are implementing? However, BDD does not say that it is driven by tests, it uses tests to guarantee that we implement the expected behavior. BDD emphasizes more that we need to implement exactly what our business wants us to implement. We use the business acceptance criteria as a baseline for our (acceptance) tests.

Getting Started with BDD

Applying BDD can simply be done by translating all the acceptance criteria into automated tests. It does not matter whether you use a built-in tool like MockMVC or that you prefer implementing the acceptance tests with a specialized framework like Cucumber. This is a decision that every team should make for itself. If some team members already know how to work with Cucumber, I do not see any reason not to use it!

These libraries are built to facilitate the transition to BDD. They offer a way to make the tests and the test results understandable for non-technical stakeholders because a better understanding of what we are doing as a software development team should increase their interest in our work. As a result, they should be more involved in the development process, giving us the ability to deliver exactly what the business wants from us quickly.

If we transition to a more behavior-driven testing approach, we will find ourselves writing better, more resilient tests that can stand the test of time. Tests focused on behavior do not depend on a specific implementation, so when we start refactoring, our tests should not start breaking too!

Upgrade Your Tests!

But do we really need to introduce BDD to write robust tests? Of course not! The acceptance tests are very often integrated tests that must spin up an application context. This takes time and makes our build slow. Eventually leading to the tests being skipped in early build stages, which is a recipe for disaster!

But how can we focus on behavior instead of implementation with fast tests? Nothing is faster than a unit test and a unit test runs in isolation! That is true, but there is more to say about this.

What Is a Unit?

A unit test is a test for a specific unit, but what is this unit? There is no single definition of a unit. For some developers, a unit is a single class, sometimes even a single method. Personally, I do not agree

with this anymore. A unit can also be as big as an entire flow through the application, where we only avoid depending on integrations with external systems or infrastructure like our database.

Integrating classes to test an entire flow does not mean our tests are no longer isolated. The isolation means that every test runs independently from other tests. They all create their own instance of the unit they are testing, not depending on the state that can be changed by another test.

Solitary Tests

Martin Fowler has written a great post about this on his website[1], where he elaborates on two distinct types of unit tests. The first one is the type of tests that we learn to write when we first start programming. These are the solitary unit tests[2]. A solitary unit test often tests the methods of a single class. All dependencies are replaced by test doubles, like Mockito mocks. This allows us to test the code of a single class, without depending on the implementation of its dependencies.

As I mentioned earlier, the solitary tests can lead to *Mockitis*, something we should always try to avoid. The only known cure for *Mockitis* is abandoning the solitary unit tests in favor of another type of unit test.

Sociable Tests

Fowler described this second type of unit test as sociable unit test. This means the unit under test relies on its dependencies instead of replacing them with a test double. So, if I'm testing a service that relies on another independent service to perform a complex calculation, I do not need to mock out this service. I can simply instantiate my system under test and all its dependencies to be able to test the behavior of the unit I'm testing.

I said all its dependencies, but I should have written this a little more carefully. **Almost** all its dependencies would have been more accurate. I don't want to depend on the implementation of a real external service, or I don't want to depend on the data in a database, even if it is a database test container. I want my tests to be fast, so I do not integrate with components that can slow my test down. These dependencies will still

be replaced by a test double. Otherwise, we would need an application context to be able to connect with the correct database or integrate with an external REST API.

Although making assumptions as a software developer is very often the root cause of problems, while writing sociable unit tests, there is one assumption that we can make! We can assume that all our dependencies behave as expected. Why? All dependencies should have their own tests! This is why we shouldn't be bothering about replacing all dependencies with test doubles, leading to more concise tests with less boilerplate code to mock the behavior of our dependencies.

Testing Behavior, Not Implementation

Instead of mocking, we write our tests with a focus on the behavior. As I mentioned earlier, this leads to less code in our tests. Less is more. When there is less code being written, our understanding of the code should be better.

However, I must mention that we can't entirely abandon test doubles. When using frameworks in our project, like we all do, most of us rely on the framework to work its magic to provide some key components of our applications, like database connectivity for example. When writing (sociable) unit tests, we don't want to spin up an application context to let the framework do its thing. Nevertheless, we will need an implementation for our repository layer. This is something that we will likely replace with a test double.

In my opinion, everything that is within the scope of our application and doesn't directly communicate with an external component, should not be replaced. If the class is responsible for handling interactions with an external component, we will very often use test doubles, just to avoid the need for an application context. Using an application context slows down the test execution too much to benefit from it within a unit test.

I know that setting up sociable tests can be cumbersome, especially in larger projects. I'm currently exploring some possibilities to simplify this, but it's too early to write down all the details here. Are you interested in how this could help you or do you have any ideas? Feel free to reach

out and we'll discuss this in depth! You can find me on LinkedIn, X and Bluesky!

1] https://martinfowler.com/articles/2021-test-shapes.html

[2] Jay Fields came up with the terms "solitary" and "sociable"

Test Data

Another keystone of good tests is the data being used to run the test. Where does the data come from? How is the data being initialized?

I often see developers creating their test data in their tests. This again dirties the test code, especially when you are working with complex nested objects. I also very often see all tests rely on static test objects. This has some critical disadvantages.

Testing with static data limits me to be confident in the tests. I am confident that my code works well with the specific data that is being used in the test, but what if one of the values is changed? Will everything still work? I'm not sure about that!

I prefer to have some more random values in my tests. But how should we use random values in our tests? What are the different approaches, and which is the best choice? Keep reading to find out more about how I think about test data!

Mockaroo

A first step into writing tests with more random data can be taken by using something like Mockaroo. Mockaroo is a platform that can be used to generate random but realistic test data. You can specify what the test data should look like. Mockaroo provides a set of generators that populate the data with realistic values, but you can also create complex formulas that take values from other columns into account to calculate the value of a field.

You can simply define the schema of the data you want to generate and download a set of random values in different formats, like SQL, CSV, JSON,... This data can then be included in your project. This is the easiest way to use Mockaroo. Even though your data is generated randomly, it becomes static data once you start using it. Mockaroo can also be used by doing API calls, but this will again introduce some complexity in your tests. Doing so can also cause test failures when the Mockaroo API is not working correctly. This is again something we want to avoid at all costs, because we only want our tests to fail for one reason, being an issue in our own code.

Data Generation Libraries

Another way to create random data is by using libraries in your code. The first library I encountered to generate realistic random data, was datafaker. Datafaker is a library that can be used to generate realistic pseudo random values, it is shipped with lots of built-in data providers. You also have the possibility to create and register custom providers to generate specific values.

Datafaker

I used datafaker in the past. Back then it was only possible to generate single random values, not entire objects populated with random values. This possibility has been added in 2022. Since then, we can specify a schema where we setup how each value should be populated. This gives us the possibility to create complex nested objects with a single command.

Instancio

Because datafaker didn't provide this possibility in the past, I started searching for another library that could do this. I found a powerful library called Instancio that did just that. The downside of Instancio in comparison to datafaker is that when generating string values, the result is really a random String, while when using datafaker, you can specify which provider it should use to generate the String. However, this is also something that could be done in Instancio by creating a custom generator.

We can also choose to combine both datafaker and Instancio, as they both depend on the default java.util.Random class to generate the values. A big advantage in Instancio is that we can easily regenerate the same value by passing a seed value to our test. This is something that is a little more cumbersome in datafaker.

Which library you choose, doesn't really matter to me. The main thing you should be aware of is that it should be very easy to switch between libraries without having to change all your tests. How can you do this?

Introducing Fixtures

To avoid having to change every test class when you decide to replace your data generation framework, you should always try to avoid using the framework directly in your tests. The cleanest way to do this is by using a fixture class, or what Martin Fowler calls the Object Mother pattern[1]. The creation of the test object can then be moved into the object mother, using a static factory method that is exposed to be used in one or more test cases.

However, as Martin Fowler stated that using Object Mother can cause some typical faults, using Object Mother with more random data will very likely not cause this issue. But why is that? The simplest object mother factory methods create an object with static data. Tests will rely on this static data. When using random data in your factory methods, you won't be able to write your tests with static expectations. You will always have to use the values that are present in the created object to do your assertions.

Reduce Boilerplate

Another issue can occur when you are using object mother in combination with immutable objects (a java record class for instance). If you need a test object with some specific values being set, you must create a factory method with parameters for this specific combination. This could lead to an unmanageable pile of factory methods. This is also something we should try to avoid!

At my current project, where we are using Instancio, we started thinking about how we could solve this issue. We came up with a builder-like construct, where we can set specific values in a fluent way. The fields that we don't care about for that specific test case, will be populated with a random value when the build method is called. It took us some iterations to make it good enough to work for all of our specific use cases, but in the end, we had something that was good enough for us to use.

Open Source

I ended up extracting this setup to a separate project that has been published as an artifact[2], ready to be used by anyone! There is still room for improvement, because the builder still has to be implemented for every class, so there is some boilerplate code required. The plan is to investigate the possibility to generate the entire fixture builder at compile time, just like Lombok generates code based on the annotations we add to a class.

[1] https://martinfowler.com/bliki/ObjectMother.html

[2] Visit https://wouter-bauweraerts.github.io/instancio-fixture-builder/ for more details

Conclusion

Automated tests are a great tool to improve the reliability of our codebase. There are some things that we must be aware of. One of the things that we should avoid is that our tests are written (or AI generated) based on the production code. We can simply do this by using the tests as the entry point for our implementation. To do this, we should adopt a BDD or TDD workflow.

Another thing to keep in mind is that our tests should not rely on a specific implementation. This can be achieved by writing sociable tests, that focus on asserting that the code behaves as expected instead of testing the specific technical implementation with solitary unit tests.

Finally, we can also improve our tests by adopting a data generation library like Instancio or Datafaker to populate our test objects with

random – but realistic – data. There are some patterns that we can use to make an abstraction of this testing library in our tests, so that when we decide to use another testing library, we should not be worried that every test must be changed accordingly.

The Object Mother pattern is the easiest way to achieve this, but we can provide an additional layer of flexibility to this by adding a fluent way to customize the object creation without the risk of cluttering our code by piling up factory methods for every possible parameter combination we need in our tests.

> Back to Table of Content

#JAVAPRO #TESTING #QUALITY

Mastering JUnit: Navigating Between Old and New Versions for a Smarter Test Strategy

Author:

Jean Donato helps companies and developers create software with quality, design, and tests. He has worked in software development for about 14 years, focusing on Java and Agile methods.

https://www.linkedin.com/in/jeandonato/

After the JUnit 5 was released, a lot of developers just added this awesome new library in their projects, because unlike others versions, this new version, is not necessary migrate from JUnit 4 to 5, you just need include the new library in your project, and with all the engine of JUnit 5 you can do your new tests using JUnit 5, and the older one with JUnit 4 or 3, will keep running without problem.

But what can happen in a big project, a project that was builded 10 years ago with two versions of JUnit running in parallel ?

New developers have started to work in the project, some of them with JUnit experience, anothers no. New tests are created using JUnit 5, new tests are created using JUnit 4, at some point a developer without

knowlegde, when they will create a new scenario in a JUnit 5 test that has been already created, they just include a JUnit 4 annotation, and the test became a mix, some @Test of JUnit 4 and some @Test of JUnit 5, and each day is more difficult to remove JUnit 4 library.

So, how to solve this problem?
First of all, you need to show to your team, what is from JUnit 5 and what is from JUnit 4, so that new tests be create using JUnit 5 instead of JUnit 4. After that is necessary follow the boy scout rule, whenever they pass by a JUnit 4 test they must migrate to JUnit 5.

Let's see the mainly changes released in JUnit 5. All starts by the name, in JUnit 5, you don't see packages called org.junit5, but rather org.junit.jupiter. To sum, everything you see with "jupiter", it means that is from JUnit 5. They chose this name because Jupiter starts with "JU", and is the fifth planet from the sun.
Another change is about the @Test , this annotation was moved to a new package : org.junit.jupiter.api and now no one attribute like "expected", "timeout" is used anymore, use extension instead. For example, for timeout, now you have one annotation for this:
@Timeout(value = 100, unit = TimeUnit.MILLISECONDS) . Another change is that neither test methods nor classes need to be public.

Now instead of use @Before and @After in your test configuration, you have to use @BeforeEach and @AfterEach, and you have also @BeforeAll and @AfterAll .

To ignore tests, now you have to use @Disable instead of @Ignore .

A great news that was released in JUnit 5 was the annotation @ParameterizedTest , with that is possible to run one test multiple times with different arguments. For example, if you want to test a method that create some object and you want validade if the fields are filled correctly, you just do:

```java
    @ParameterizedTest
    @MethodSource("getInvalidSources")
    void shouldCheckInvalidFields(String name, String job, String expectedMessage) {
        Throwable exception = catchThrowable(() -> new Client(name, job));

        assertThat(exception)
                .isInstanceOf(IllegalArgumentException.class)
                .hasMessageContaining(expectedMessage);
    }

    static Stream<Arguments> getInvalidSources(){
        return Stream.of(
                Arguments.arguments("Jean Donato", "", "Job is empty."),
                Arguments.arguments("", "Dev", "Name is empty.")
        );
    }
```

There are so nice features in JUnit 5 , I recommend you check it out the JUnit 5 User Guide, to analyze what is useful to your project. https://junit.org/junit5/docs/current/user-guide/

Now that all developers knows what was changed in JUnit 5, you can start the process of removing of JUnit 4 from your project. So, if you are still using JUnit 4 in 2024, and your project is a big project, you will probably have some dependencies using JUnit 4. I recommend you analyze your libraries to check if some of them is using JUnit 4.

In the image bellow I'm using Dependency Analyzer from IntelliJ.

```
jersey-test-framework-provider-simple:2.29.1 (test)
    jersey-test-framework-core:2.29.1 (test)
        junit:4.12 (test) conflict with 4.13.1
    junit:4.12 (test) conflict with 4.13.1
junit-vintage-engine:5.7.0 (test)
    junit:4.12 (test) conflict with 4.13.1
junit:4.13.1 (test)
```

As you can see, jersey-test is using JUnit 4, that is, even if I remove JUnit 4 from my project, JUnit 4 will be available to use because Jersey. The easier way will be bump jersey to 2.35, because JUnit 5 was introduced in jersey-test 2.35, but I can't update jersey-test framework because other libraries will break in my project. So, in this case, what can I do?

I can exclude JUnit from jersey with Dependency Exclusions from maven(like image bellow). That way JUnit 4 will not be use anymore, but rather our JUnit 5.

```
<dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-simple</artifactId>
    <version>${glassfish.jersey.test-framework.providers}</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

After that, when you run some test wich uses Jersey, they will not be loaded, because there are methods in Jersey using JUnit 4 annotations, setUp and tearDown, using @Before and @After . To solve this, you can create one "Configurarion Class" whose extends JerseyTest implementing setUp and tearDown with @BeforeEach and @AfterEach calling super.setUp() and super.TearDown().

```
public class JerseyConfigToJUnit5 extends JerseyTest {

    @BeforeEach
    public void setUp() throws Exception {
        super.setUp();
    }

    @AfterEach
    public void tearDown() throws Exception {
        super.tearDown();
    }
}
```

So, if you have already checked your libraries and no one has more dependency from JUnit 4, you finally can migrate all your tests to JUnit 5, and for this process, there is a good tool that saves you from a lot of work, is OpenRewrite (https://docs.openrewrite.org/) , a automated refactoring ecosystem for source code, they will change all your old packages, the older annotations, and everything to the new one.

That's it folks, now you and your team mates can enjoy JUnit 5 and relax your mind knowing that new test's will be create with JUnit 5 and the project will not became a frankenstein. So, remember, keep your project up-to-date, because if you forget your libraries, each day will be more difficult to update, always use specification, frameworks that follows the specification, and have a good design in your code, this permits you change and move with facility.

> Back to Table of Content

For your IT projects you don't need a know-it-all.

You need a {BUDDY}

Richard Fichter
CEO @ XDEV

Outdated software? Rising maintenance costs? Security risks? We help you to make Java applications fit for the future - with a clear concept and at eye level. In addition to modern tools, we offer premium support for your **Java modernization.**

- **Not a know-it-all - a buddy:** We support your team with pragmatic methods without playing the wise guy.
- **Modernization with strategy:** agile methods & proof of concept for a secure update.
- **Robust solutions:** We rely on proven technologies and practical concepts - without unnecessary complexity.

Let us move your Java project forward together!

trusted by

Arrange a free discovery call here!