# Failify Documentation

***Release 0.1.0***

**Armin Balalaie**

**Mar 02, 2019**

# Table of Contents

CHAPTER 1

# Introduction

Failify is a test framework for end-to-end testing of distributed systems. It can be used to deterministically inject failures during a normal test case execution. Currently, node failure, network partition, and clock drift is supported. For a few supported languages, it is possible to enforce a specific order between nodes in order to reproduce a specific time-sensitive scenario and inject failures before or after a specific method is called when a specific stack trace is present.

## 1.1 Prerequisites

To use Failify, you need to install the following dependencies on your machine:

- Java 8+
- Docker 1.13+ (Make sure the user running your test cases is able to run Docker commands. For example, in Linux, you need to add the user to the docker group)

It is also recommended to use a build system like Maven or Gradle to be able to include Failify's dependency.

## 1.2 Quick Start

Failify is a Java-based end-to-end testing framework. So, you will need to write your test cases In Java, or languages that can use Java libraries like the ones that can run on JVM, e.g. Scala. Failify can be used alongside the popular testing frameworks in your programming language of choice e.g. JUnit in Java. Here, we use Java and JUnit . We also use Maven as the build system.

### 1.2.1 Adding dependencies

First, create a simple Maven application and add Failify's dependency to your pom file.

```
<dependency>
    <groupId>io.failify</groupId>
    <artifactId>failify</artifactId>
    <version>0.1.0</version>
</dependency>
```

Also add failsafe plugin to your pom file to be able to run integration tests.

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>3.0.0-M3</version>
        <executions>
          <execution>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

### 1.2.2 Creating a Dockerfile

Next, you need to create a Dockerfile for your application and that Dockerfile should add any dependency that may be needed by your application. In case you want to use the network partition capability of Failify, you need to install `iptables` package as well. Here, we assume the application under test is written in Java. So, we create a Dockerfile in the `docker/Dockerfile` address with the following content:

```
FROM java:8-jre
RUN apt update && apt install -y iptables
```

### 1.2.3 Adding a Test Case

Now, create a JUnit integration test case (ending with IT so failsafe picks it up) in the project's test directory. Here, we are assuming the final distribution of the project is a zipfile in the Maven's `target` directory. Also, we are assuming the zip file contains a `project-[PROJECT_VERSION]` directory and that directory itself contains a `bin` directory which contains a `start.sh` file to start the application.

```
1  public class SampleTestIT {
2      protected static FailifyRunner runner;
3
4      @BeforeClass
5      public static void before() throws RuntimeEngineException {
6          String projectVersion = "0.2.1";
7          Deployment deployment = Deployment.builder("sampleTest")
```

(continues on next page)

(continued from previous page)

```
8               // Service Definition
9               .withService("service1")
10                  .applicationPath("target/project.zip", "/project", PathAttr.
    ↪COMPRESSED)
11                  .startCommand("/project/project-" + projectVersion + "/bin/start.sh -
    ↪conf /config.cfg")
12                  .dockerImage("project/sampleTest:" + projectVersion)
13                  .dockerFileAddress("docker/Dockerfile", false)
14                  .tcpPort(8765)
15                  .serviceType(ServiceType.JAVA).and()
16              // Node Definitions
17              .withNode("n1", "service1")
18                  .applicationPath("config/n1.cfg", "/config.cfg".and()
19              .withNode("n2", "service1")
20                  .applicationPath("config/n2.cfg", "/config.cfg".and()
21              .build();
22
23          FailifyRunner runner = FailifyRunner.run(deployment);
24      }
25
26      @AfterClass
27      public static void after() {
28          if (runner != null) {
29              runner.stop();
30          }
31      }
32
33      public void test1() throws RuntimeEngineException {
34          ProjectClient client = ProjectClient.from(runner.runtime().ip("n1"),
35              runner.runtime().portMapping("n1", 8765, PortType.TCP));
36          ..
37          runner.runtime().clockDrift("n1", 100);
38          ..
39      }
40  }
```

Each Failify test case should start with defining a new `Deployment` object. A deployment definition consists of a a set of service and node definitions. A Service is a node template and defines the docker image for the node, the start bash command, required environment variables, common paths, etc. for a specific type of node. For additional info about available options for a service check ServiceBuilder's JavaDoc.

Line 9-15 defines `service1` service. Line 10 adds the zip file to the service at the `/project` address and also marks it as compressed so Failify decompresses it before adding it to the node (**In Windows and Mac, you should make sure the local path you are using here is shared with the Docker VM**). Line 11 defines the start command for the node, and in this case, it is using the `start.sh` bash file and it feeding it with `-conf /config.cfg` argument. This config file will be provided separately through node definitions later. Line 14 marks tcp port `8765` to be exposed for the service. This is specially important when using Failify in Windows and Mac as the only way to connect to the Docker containers in those platforms is through port forwarding. Line 15 concludes the service definition by marking it as a Java application. If the programming language in use is listed in `ServiceType` enum, make sure to mark your application with the right `ServiceType`.

Lines 17-20 defines two nodes named `n1` and `n2` from `service1` service and is adding a separate local config file to each of them which will be located at the same target address `/config.cfg`. Most of the service configuration can be overriden by nodes. For more information about available options for a node check Node Builder's JavaDoc.

Line 23 starts the defined deployment and line 29 stops the deployment after all tests are executed.

Line 34-35 shows how the `runner` object can be used to get the ip address and port mappings for each node to be

potentially used by a client. Line 37 shows a simple example of how Failify can manipulate the deployed environment by just a method call. In this case, a clock dirft of 100ms will be applied to node `n1`. For more information about available runtime manipulation operations check LimitedRuntimeEngine's JavaDoc.

### 1.2.4 Logger Configuration

Failify uses SLF4J for logging. As such, you can configure your logging tool of choice. A sample configuration with Logback can be like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="Console" class="ch.qos.logback.core.ConsoleAppender">
        <layout class="ch.qos.logback.classic.PatternLayout">
            <Pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</
↪Pattern>
        </layout>
    </appender>

    <logger name="io.failify" level="DEBUG"/>

    <root level="ERROR">
        <appender-ref ref="Console" />
    </root>
</configuration>
```

### 1.2.5 Running the Test Case

Finally, to run the test cases, run the following bash command:

```
$  mvn clean verify
```

## 1.3 Deterministic Failure Injection

Although injecting a failure by calling a method in the middle of a test case is suitable for many of the scenarios, there exists scenarios where it is needed to inject failures in a very specific moment. With Failify, for a few supported languages, it is possible to inject a failure right before or after a method call where a specific stack trace is present. This happens through defining a set of named internal and test case events, ordering those events in a run sequence string, and let the Failify's runtime engine enforce the specified order between the nodes.

### 1.3.1 Internal Events

Internal events are the ones that happen inside a node. Realizing internal events requires binary instrumentation, and as such, is only supported for a few programming languages. You can find more information in *Run Sequence Instrumentation Engine* page. Available internal events are:

- **Scheduling Event**: This event can be of type *BLOCKING* or *UNBLOCKING* and can happen before or after a specific stack trace. The stack trace should come from a stack trace event definition. When defining this kind of events, the definition should be a pair of blocking and unblocking events. Basically, make sure to finally unblock everything that has been blocked. This event is useful when it is needed to block all of the threads for a specific stack trace, do some other stuff or let the other threads make progress, and then, unblock the blocked threads.

```
.withNode("n1", "service1")
    .withSchedulingEvent("bast1")
        .after("st1") // The name of the stack trace event. An example comes later
        .operation(SchedulingOperation.BLOCK)
    .and()
    .withSchedulingEvent("ubast1")
        .after("st1")
        .operation(SchedulingOperation.UNBLOCK)
    .and()
    // The same events using shortcut methods
    .blockAfter("bast1", "st1")
    .unblockAfter("ubast1", "st1")
.and()
```

- **Stack Trace Event**: This event is kind of like a scheduling event except that nothing happens between blocking and unblocking. All of the threads with the defined stack trace will be blocked until the dependencies of the event are satisfied (based on the defined run sequence). The blocking can happen before or after a method. This event can act as an indicator that the program has reached a specific method with a specific stack trace.

```
.withNode("n1", "service1")
    .withStackTraceEvent("st1")
        .trace("io.failify.Hello.worldCaller")
        .trace("io.failify.Hello.world")
        .blockAfter().and()
    // The same event using a shortcut method
    .stackTrace("st1", "io.failify.Hello.worldCaller,io.failify.Hello.world", true)
.and()
```

- **Garbage Collection Event**: This event is for invoking the garbage collector for supported languages e.g. Java.

```
withNode("n1", "service1").
    .withGarbageCollectionEvent("gc1").and()
and()
```

### 1.3.2 Test Case Events

Test case events are the connection point between the test case and the Failify's runtime engine. Internal events's orders are enforced by the runtime engine, but it is the test case responsibility to enforce the test case events if they are included in the run sequence.

```
new Deployment.Builder("sample")
    .testCaseEvents("tc1","tc2")
```

### 1.3.3 The Run Sequence

Finally after defining all the necessary events, you should tie them together in the run sequence by using event names as the operands, `*` and `|` as operators and parenthesis. `*` and `|` indicate sequential and parallel execution respectively.

```
new Deployment.Builder("sample")
    .runSequence("bast1 * w1 * ubast1 * (gc1 | x1)")
```

This run sequence blocks all the threads in node `n1` with the stack trace of event `st1` (`bast1`), waits for the test case to enforce `tc1`, unblcoks the blocked threads in node `n1` (`ubast1`), and finally, in parallel, performs a garbage collection in `n1` (`gc1`) and kills node `n2` (`x1`).

At any point, a test can use the `FailifyRunner` object to enforce the order of a test case event. Enforcement of a test case event in the test case is only needed if something is needed to be done when the event dependencies are satisfied, e.g. injecting a failure.

```
runner.runtime().enforceOrder("tc1", 10, () -> runner.runtime().clockDrift("n1", -
↪100));
```

Here, when the dependencies of event `tc1` are satisified, a clock drift in the amount -100ms will be applied to node `n1`, and `tc1` event will be marked as satisfied. If after 10 seconds the dependencies of `tc1` are not satisfied, a `TimeoutException` will be thrown. If the only thing that the test case needs is to wait for an event or its dependencies to be satisfied the `waitFor` method can be used.

```
runner.runtime().waitFor("st1", 10);
```

Here again, if the event dependecies are not satisfied in 10 seconds, a `TimeoutException` will be thrown.

## 1.4 Run Sequence Instrumentation Engine

Failify's deterministic failure injection requires binary instrumentation. Different programming languages require different instrumentors, and thus, if you are going to use this feature, you need to specify the programming language for involved services.

```
.withService("service1")
    .serviceType(ServiceType.JAVA)
```

Next, for each service, you may need to mark some paths as library or instrumentable paths. Check specific language instructions as this may differ based on the programming language in use.

### 1.4.1 Java

AspectJ is used for Java instrumentation. AspectJ 1.8+ should work perfectly with Failify. You need to install Aspectj on your machine and expose `ASPECTJ_HOME` environment variable pointing to the home directory of AspectJ in your machine. Also, you need to include AspectJ and Failify runtime dependencies to your project. Example dependencies to be added to your pom file with AspectJ 1.8.12 are as follows:

```
<dependency>
    <groupId>io.failify</groupId>
    <artifactId>failifyrt</artifactId>
    <version>0.1.0</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.8.12</version>
</dependency>
```

Finally, you need to mark:

- all of the required jar files or class file directories to run your application as **library path**.
- all of the jar files or class file directories which contain a method included as the last method in one of the stack trace events as **instrumentable path**

```
.withService("service1")
    .applicationPath("./projectFiles", "/project")
    // It is possible to use wildcard paths for marking library paths
    .libraryPath("/project/libs/*.jar") // This is a target path in the node.
    .applicationPath("target/classes", "/project/libs/classes", PathAttr.LIBRARY)
    .applicationPath("./extraLib.jar", "/project/libs/extraLib.jar", PathAttr.LIBRARY)
    .instrumentablePath("/project/libs/main.jar") // This is a target path in the node
    .instrumentablePath("/project/libs/classes")
.and()
```

## 1.5 Creating a Service From JVM Classpath

In case your application is using a JVM-based programming language and is able to include Java libraries, you can use the current JVM classpath to create a service.

```
1 new Deployment.Builder("sample")
2     .withServiceFromJvmClasspath("s1", "target/classes", "**commons-io*.jar")
3         .startCommand("java -cp ${FAILIFY_JVM_CLASSPATH} my.package.Main")
4     .and()
```

This method will create a new service by adding all the paths included in the JVM classpath as library paths to your service. Also, any relative, absolute or wildcard paths that comes after the service name, if exists in the class path, will be added to the service as an instrumentable path. This method will return a `ServiceBuilder` object, as such, all the regular service configurations are available.

As can be seen in line 3, the new classpath based on the new target paths is provided in the `FAILIFY_JVM_CLASSPATH` and can be used in the service's start command.

## 1.6 Running Failify in Docker

### 1.6.1 Why it may be needed?

There could be two reasons that you want to run Failify test cases in a Docker container:

- Your CI nodes are Docker containers and you don't have any other options
- Your client needs to access the nodes using their hostname or on any port number (without exposing them). Either of these cases requires the client to be in the same network namespace as the nodes and that is only possible if you run Failify in a Docker container.

### 1.6.2 How to do this?

1. Create a docker image for running your test cases. That image should at least include Java 8+. You may want to install a build system like Maven as well. Also, install any other packages or libraries which are needed for your test cases to run and are already installed in your machine. In case you need instrumentation for your test cases, install the required packages for your specific instrumentor as well.

```
FROM maven:3.6.0-jdk-8
ADD /path/to/aspectj
ENV ASPECTJ_HOME="/path/to/aspectj"
```

2. Change the current directory to your project's root directory. Start a container from the created image with the following docker run arguments:

- Share your project's root directory with the container (`-v $(pwd):/path/to/my/project`)
- Make the project's root directory mapped path the working directory in the container (`-w /path/to/my/project`)
- Share the docker socket with the container (`-v /var/run/docker.sock:/var/run/docker.sock`)

Your final command to start the container should be something like this:

```
$  docker run --rm -v /var/run/docker.sock:/var/run/docker.sock -v $(pwd):/path/to/my/
↪project
-w /path/to/my/project myImage:1.0 mvn verify
```