PENNY ROHR CURICH

TESTING WITH MARTINI

Enterprise-Level Behavior Driven Development Testing in Java

Table of Contents

Advanced Testing

Martini offers numerous advanced concepts that address various common problems encountered with the testing of complex software systems.

Filtering

Scenario filtering with nested boolean expressions is supported by levering Spring Expression Language (SpEL).

For more information on SpEL, reference Spring documentation at https://docs.spring.io/spring/docs/5.1.2.RELEASE/spring-framework-reference/core.html#expressions.

Features can be included or excluded from a test run based on resource location. This is accomplished using the isResource('…') expression which utilizes Spring's ApplicationContext as a ResourcePatternResolver.

For example, consider the following Maven classpath structure.

```
/resources
   /features
      / subsystemA
         subsystemA.feature
         /subsystemA1
            subsystemA1-1.feature
      / subsystemB
         subsystemB.feature
      / subsystemC
```

If only subsystemA should be tested, then we can use the following runtime argument:

```
-spelFilter "isResource('/features/subsystemA/**/*.feature')"
```

This would result in scenarios contained in resources `subsystemA.feature` and `subsystemA1-1.feature` being executed.

If only scenarios in resource `subsystemA.feature` should be executed, then a more succinct filter can be applied:

```
-spelFilter "isResource('/features/subsystemA/subsystemA.feature')"
```

If subsystemA and subsystemB scenarios should be run with the exception of subsystemA1, then either of the following filter could be applied:

```
-spelFilter "(isResource('/features/subsystemA') AND !
isResource('features/subsystemA1')) OR isResource('/features/
subsystemB/**.feature')"
```

```
-spelFilter "isResource('/features/subsystemA/*.feature') ||
isResource('/features/subsystemB/**/*.feature')
```

Reference Spring API document https://docs.spring.io/spring/docs/5.1.2.RELEASE/javadoc-api/ for more information on the application of ResourcePatternResolver.

Gating

Test suites with a large number of scenarios that execute in parallel against a particular subsystem may potentially result in application resource contention.

For example, imagine an application that creates a report, but for technical reasons can only run one report at a time. This means the application must either refuse, block or queue subsequent report requests when already running a report.

The effect on parallelized tests for this application will potentially be one of

- failures based on refused requests
- failures based on applied timeouts

- differing run times across test executions

Martini answers this with a gating concept, managing the parallelization of scenarios with shared gates by evaluating available gate permits. The standalone engine executes scenarios with the largest number of unique gates first, interleaving other non-gated scenarios and scenarios with unshared gates.

A gate is defined by applying one or more @Gated annotation to a step method.

For example, examine the following two scenarios:

```
Scenario: One
Given condition one
When something happens
Then something happened

Scenario: Two
Given condition two
When something else happens
Then verify something happened
```

And the following step implementations:

```
@Steps
public class One {
   @Gated(“auditing”)
   @Gated(“reporting”)
   @When(“something happens”)
   public void doSomething() {
      …
   }
}


@Steps
public class Two {
   @Gated(“reporting”)
   @Then(“verify something happened”)
   public void assertSomething() {
```

```
        …
    }
}
```

Both scenarios share the “reporting” gate, which by default has a single permit.

With these as the only two scenarios, Scenario One will be executed first as it has more unique gates.

Executed in parallel with other scenarios that also have the “reporting” gate, the standalone engine will skip execution of these tests in favor of other available tests while until a reporting permit is available. If the remaining tests all have “reporting” gates, then the standalone engine will execute the remaining tests serially.

By default, all gates are assigned one permit.

Gate permits are configured via Spring properties. This can be accomplished via the Java command-line as a -D argument, or via properties loaded by the application.

Given the following Spring configuration file:

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:martini="http://qas.guru/schema/martini"

    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd

       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-
context.xsd">
```

```
    <import resource="classpath*:**/martiniContext.xml"/>

    <context:property-placeholder
location="classpath*:gates.properties"/>
    <context:component-scan base-package="com.mycompany.spring"/>
  </beans>
```

For the previous example, the "reporting" permits could be defined by adding an entry file gates.properties:

```
martini.gate.permits.reporting=3
```

This would allow up to three scenarios with the "reporting" gate to run simultaneously.