CYDEO

1- Cucumber Rerun
2- Maven Cucumber Reporting
3 – Parallel Execution-

After this session

- You will be able to run only failed tests (RERUN)
- You will be able to generate Maven Cucumber Reporting
- You will be able to run tests in parallel

CUCUMBER RERUN PLUGIN

- Cucumber rerun plugin allows us to store failed scenarios in a file
- We can run those scenarios using another runner class
- Why do we need another runner class?
- Because we need to point

```
plugin = {
        "html:target/cucumber-report.html",
        "rerun:target/rerun.txt"
},
```

FailedTestRunner

```
@RunWith(Cucumber.class)
@CucumberOptions(
        features = "@target/rerun.txt",
        glue = "com/cydeo/step_definitions"
)
public class FailedTestRunner {
}
```

Maven Cucumber Reporting

- This dependency is created as a wrapper to already existing plugin.
- There is a new dependency we need to add to be able to generate a new type of report.
- Me.jvt.cucumber → Let's see from Gitlab!

Implement in 2 easy steps:

1. Add dependency in pom.xml

```
<dependency>
    <groupId>me.jvt.cucumber</groupId>
    <artifactId>reporting-plugin</artifactId>
    <version>5.3.0</version>
</dependency>
```

2. Add plugin in the runner class

```
plugin = {
        "html:target/cucumber-report.html",
        "rerun:target/rerun.txt",
        "me.jvt.cucumber.report.PrettyReports:target/cucumber"
},
```

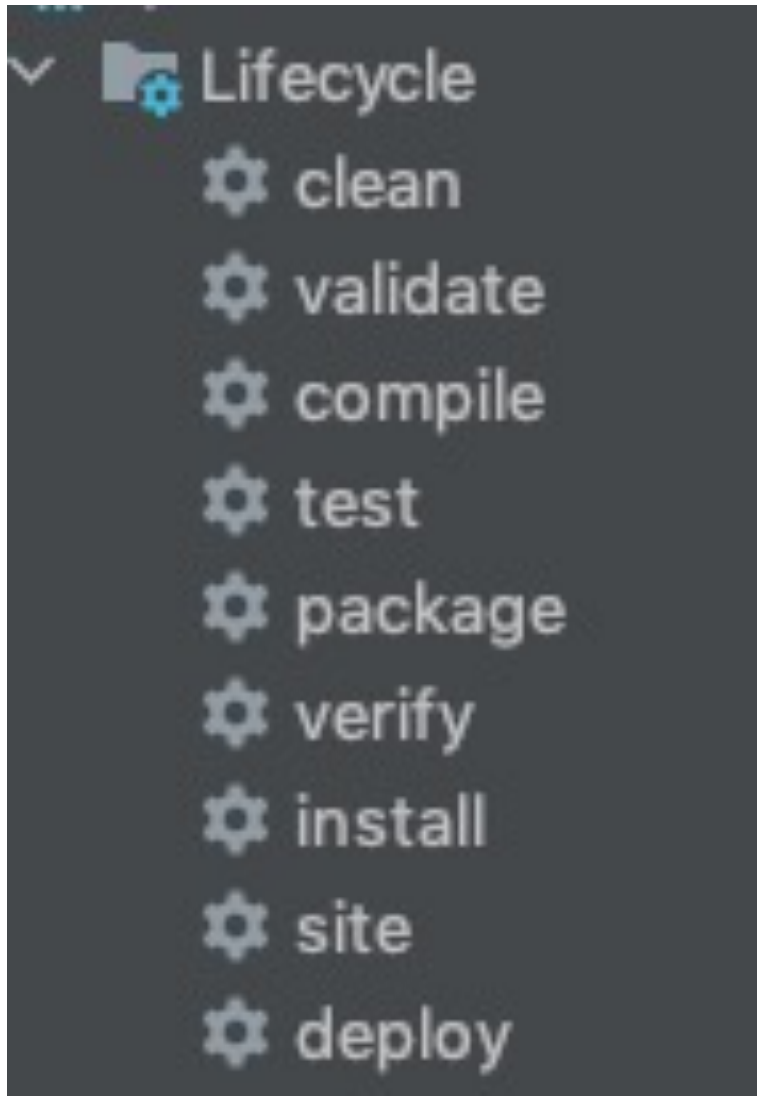
Why do we use parallel testing?

- Save time
- Cucumber supports parallel testing out of the box since version 4

- https://cucumber.io/docs/guides/parallel-execution/

What are maven lifecycles?

- Maven is based around the central concept of a build lifecycle.
- What this means is that the process for building and distributing a particular artifact (project) is clearly defined.
- For the person building a project, this means that it is only necessary to learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired.



- https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

A little bit more into the lifecycles

- `validate` - validate the project is correct and all necessary information is available
- `compile` - compile the source code of the project
- `test` - test the compiled source code using a suitable unit testing framework.

These tests should not require the code be packaged or deployed

- `package` - take the compiled code and package it in its distributable format, such as a JAR.
- `verify` - run any checks on results of integration tests to ensure quality criteria are met
- `install` - install the package into the local repository, for use as a dependency in other projects locally
- `deploy` - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

- Bunch of jar files: java classes, libraries ready to use
- So, we add to our project and readily use them
- Dependencies are not involved in Maven lifecycles
- They are just ready jar files, we add and use

```
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>6.9.1</version>
    <scope>test</scope>
</dependency>
```

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
</dependency>
```

```
<dependency>
    <groupId>io.github.bonigarcia</groupId>
    <artifactId>webdrivermanager</artifactId>
    <version>4.2.2</version>
</dependency>
```

```
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>6.9.1</version>
</dependency>
```

What is a plugin?

- Plugins are also jar files just like dependencies
- They are very similar, but plugins are involved in the maven lifecycles
- Most of the work in the maven lifecycle are done by plugins
- Whereas dependencies are only jar files that is not involved in maven lifecycles.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M5</version>
    <configuration>
        <testFailureIgnore>true</testFailureIgnore>
        <parallel>classes</parallel>
        <threadCount>2</threadCount>
        <forkCount>2C</forkCount>
        <perCoreThreadCount>false</perCoreThreadCount>
        <includes>
            <include>**/CukesRunner*.java</include>
        </includes>
    </configuration>
</plugin>
```

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.6.1</version>
    <configuration>
        <source>8</source>
        <target>8</target>
    </configuration>
</plugin>
```

Plugins we are going to be using today

- Maven compiler plugin
- Maven cucumber reporting
- Maven surefire plugin

#1: Maven Surefire Plugin

- This plugin allows us to run maven lifecycle commands
- Maven life cycles : compile > test > package > verify > install > deploy
- We can also pass additional configuration to allow parallel testing

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M5</version>
    <configuration>
        <testFailureIgnore>true</testFailureIgnore>
        <parallel>classes</parallel>
        <threadCount>2</threadCount>
        <forkCount>2C</forkCount>
        <perCoreThreadCount>false</perCoreThreadCount>
        <includes>
            <include>**/CukesRunner*.java</include>
        </includes>
    </configuration>
</plugin>
```

Understanding the maven surefire plugin's similar lines as dependencies

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <configuration>
                <parallel>methods</parallel>
                <threadCount>4</threadCount>
                <perCoreThreadCount>false</perCoreThreadCount>
                <testFailureIgnore>true</testFailureIgnore>
                <includes>
                    <include>**/CukesRunner*.java</include>
                </includes>
            </configuration>
        </plugin>
    </plugins>
</build>
```

#2: Maven Cucumber Reporting (plugin)

- Custom plugin that is allowing us to create a prettier and more detailed report compared to what cucumber have by default
- Maven cucumber reporting uses cucumber json reporting to generate its own report
- We already have this reporting so we won't be adding it now again, but if you were to be using this as a plugin rather than dependency, this plugin would require you to generate a json type of report.

```
plugin = {
        "json:target/cucumber.json",
```

- Google → maven cucumber reporting
- https://github.com/damianszczepanik/maven-cucumber-reporting

Let’s add plugins to pom.xml and CukesRunner

- And generate report.

Is adding these plugins good enough to run our tests parallel?

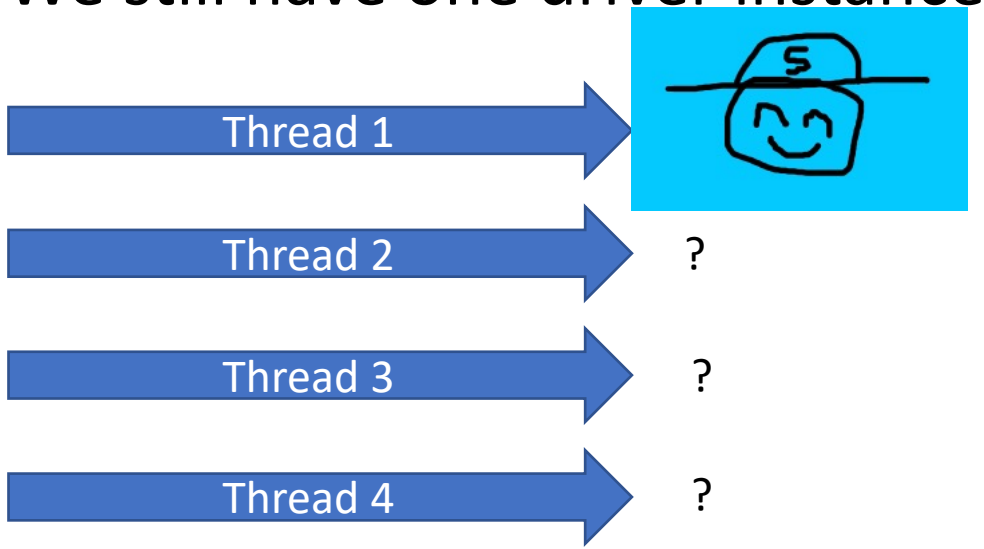
- No.

Singleton design pattern creating an issue:

Nope

Is adding these plugins good enough to run our tests parallel?

- No.
- Let’s say we created multiple Threads for our execution.
- We still have one driver instance



What problem we need to solve after adding plugin

- Driver utility, singleton design pattern.
- Normally it is a good solution to our previous problems
- But now it limits our code so that we cannot run with multi threads with our current setup.

What is the solution?

- We will use ThreadLocal class from Java to create driverPool
- ThreadLocal is a class that creates the object of the given class **PER THREAD.**
- You can think of it as this class will provide us with a pool of drivers, where our code will be able to go and use as many as needed.

We will adjust our Driver utils class accordingly

- driverPool.remove() → will remove particular object for given thread
  - We will use this instead of driver.quit()
- driverPool.set() → set/add object
  - For setting, creating webdriver
- driverPool.get() → return object

#1- We wrap our driver with ThreadLocal

```
private static InheritableThreadLocal<WebDriver> driverPool = new InheritableThreadLocal<>();
```

#2 - driverPool.get();

- Instead of using this:

```
if (driver == null) {
```

- We will use this:

```
if (driverPool.get() == null) {
```

#3- driverPool.set()

- Instead of using this:

```
driver = new ChromeDriver();
```

- We will use this:

```
driverPool.set(new ChromeDriver());
```

#4- return driver

- Instead of returning driver
- We return driverPool.get();

We also need to change our closeDriver() method

```
public static void closeDriver() {
    if (driverPool.get() != null) {
        driverPool.get().quit();
        driverPool.remove();
    }
}
```

Interview question

- If you used Singleton in your Driver, how did you handle parallel execution?

- #1- I wrapped my WebDriver object with ThreadLocal that creates copy of driver object per thread.
- #2- Instead of using driver directly I used driverPool.get() method to get a driver instance from a pool of drivers objects
- #3- This will provide me as many drivers as the number of threads I am running

Cucumber io parallel execution

- JUNIT can execute Feature files in parallel
- TestNG can execute Scenarios in parallel

How to run our tests without touching to our CukesRunner

- mvn test -Dcucumber.filter.tags=@smoke

- https://cucumber.io/docs/cucumber/api/