Parallel Tests - Best Practices

This article builds on the article, Parallel Tests Execution and focuses more on the best practices to execute tests in parallel.

In the original article, there are mentions of these practices, we will discuss those with more detail.

The example provided in this article is based on our Sample code. Below are the steps to get the sample code.

1. Find the sample test in our git repository.
2. Fork the repository.

Listed below are good practices that will go a long way for a successful implementation of parallel execution of tests.

- Include parallelism at test design/development stage
- Independent Test methods
- Parallelism and Logging
- Avoid usage of static variables in tests
- Parallelism and Test Drivers

Include parallelism at test design/development stage

As discussed in the original article, running parallel tests strategy cannot be employed at the final stage of test development and by just running tests in parallel. Parallelism needs to be thought at an early stage of the design and development of the Test Suite.

Although most of the Test framework and their language provides a way to run a test in parallel, it has observed that this is not enough. Significant changes need to be done in Tests to make this work.

Independent test methods/Tests

Tests should have a specific object and should not depend on another test. This is a very important consideration and every test developer needs to be educated about this.

Let's discuss this with an example.
Dependent test methods example

```java
@Test
public void eriBankLogin(@Optional("company") String userName, @Optional("company") String password) {
    LOGGER.info("Enter eriBankLogin - userName = " + userName + " password = " + password);
    // Find Element commands for Find Login elements.
    driver.findElement(ELEMENTS.LOGIN_USER.getBy(TYPE.ANDROID)).sendKeys(userName);
    driver.findElement(ELEMENTS.LOGIN_PASS.getBy(TYPE.ANDROID)).sendKeys(password);
    driver.findElement(ELEMENTS.LOGIN_BUTTON.getBy(TYPE.ANDROID)).click();
    LOGGER.info("Exit eriBankLogin");
}

@Test (dataProvider = "makePaymentsData")
public void makePaymentTest(String phone, String name, String amount, String country) {
    LOGGER.info("Enter makePaymentTest - Phone = " + phone + " name = " + name + " amount = " + amount + " country = " + country);
    driver.findElement(ELEMENTS.PAYMENT_BUTTON.getBy(TYPE.ANDROID)).click();
    driver.findElement(ELEMENTS.PHONE.getBy(TYPE.ANDROID)).sendKeys(phone);
    driver.findElement(ELEMENTS.NAME.getBy(TYPE.ANDROID)).sendKeys(name);
    driver.findElement(ELEMENTS.AMOUNT.getBy(TYPE.ANDROID)).sendKeys(amount);
    driver.findElement(ELEMENTS.COUNTRY.getBy(TYPE.ANDROID)).sendKeys(country);
    driver.findElement(ELEMENTS.SEND_PAYMENT_BUTTON.getBy(TYPE.ANDROID)).click();
    driver.findElement(ELEMENTS.YES_BUTTON.getBy(TYPE.ANDROID)).click();
    LOGGER.info("Exit makePaymentTest");
}
```

Two test methods i.e `makePaymentsData` and `eriBankLogin` are dependent on each other.

The test `makePaymentsData` assumes, that `eriBankLogin` will be called before and hence the state of the user will be logged in. However, when tests are run in parallel or the order of the execution changes then `makePaymentTests` will fail.

The sample code performs login operation `@BeforeMethod`, which ensures it will be always called before executing any test method.

To conclude, It is of utmost importance to develop a method which is independent of each other.

**Parallelism and Logging**

When tests are run in parallel, it's not easy to figure out what exactly is happening at execution time. The tester needs to know the sequence of the test suite and individual tests.

Generally, logs provide this information. Most of the popular languages achieve parallelism using "Threads". Hence it is a matter of utmost importance that Thread Identifier should find a place in the logs.

**Log generated when test executed in parallel WITHOUT thread identifier.**

<table>
<thead>
<tr>
<th>Time</th>
<th>Message</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>28-11-2018 17:13:34,140</td>
<td>INFO [TestNG] i.a.t.AndroidTestNGExampleTest: Enter initDefaultDesiredCapabilities</td>
<td>i.a.t.AndroidTestNGExampleTest</td>
</tr>
<tr>
<td>28-11-2018 17:13:34,140</td>
<td>INFO [TestNG] i.a.t.AndroidTestNGExampleTest: Setting up Desired Capabilities</td>
<td>i.a.t.AndroidTestNGExampleTest</td>
</tr>
<tr>
<td>28-11-2018 17:13:34,139</td>
<td>INFO [TestNG] i.a.t.iOSTestNGExampleTest: Enter initDefaultDesiredCapabilities</td>
<td>i.a.t.iOSTestNGExampleTest</td>
</tr>
<tr>
<td>28-11-2018 17:13:34,139</td>
<td>INFO [TestNG] i.a.t.iOSTestNGExampleTest: Setting up Desired Capabilities</td>
<td>i.a.t.iOSTestNGExampleTest</td>
</tr>
<tr>
<td>28-11-2018 17:13:34,139</td>
<td>INFO [TestNG] i.a.t.iOSTestNGExampleTest: Device Query = @os='ios'</td>
<td>i.a.t.iOSTestNGExampleTest</td>
</tr>
</tbody>
</table>

**Log generated when test executed in parallel WITH thread identifier**

<table>
<thead>
<tr>
<th>Time</th>
<th>Message</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>28-11-2018 17:13:34,140</td>
<td>INFO [TestNG] i.a.t.AndroidTestNGExampleTest: Enter initDefaultDesiredCapabilities</td>
<td>i.a.t.AndroidTestNGExampleTest</td>
</tr>
<tr>
<td>28-11-2018 17:13:34,140</td>
<td>INFO [TestNG] i.a.t.AndroidTestNGExampleTest: Setting up Desired Capabilities</td>
<td>i.a.t.AndroidTestNGExampleTest</td>
</tr>
<tr>
<td>28-11-2018 17:13:34,139</td>
<td>INFO [TestNG] i.a.t.iOSTestNGExampleTest: Enter initDefaultDesiredCapabilities</td>
<td>i.a.t.iOSTestNGExampleTest</td>
</tr>
<tr>
<td>28-11-2018 17:13:34,139</td>
<td>INFO [TestNG] i.a.t.iOSTestNGExampleTest: Setting up Desired Capabilities</td>
<td>i.a.t.iOSTestNGExampleTest</td>
</tr>
<tr>
<td>28-11-2018 17:13:34,139</td>
<td>INFO [TestNG] i.a.t.iOSTestNGExampleTest: Device Query = @os='ios'</td>
<td>i.a.t.iOSTestNGExampleTest</td>
</tr>
</tbody>
</table>
As we can see from above, the second snippet of the log is much clearer to a user as it clearly gives an indication that "AndroidTestNGExampleTest" is executed in thread "11". There is also more clarity to the user how the tests were executed by the run-time engine.

**Avoid usage of static variables in tests**

Using static variables should be avoided because in a parallel execution environment this will invariably result in conflicts.

Let us consider the following example.

The code snippet is a Factory class that returns a driver which is static i.e one per class.

**A Factory class to get appium driver**

```java
public class DriverFactory {
    private static AppiumDriver driver;

    public static AppiumDriver getDriver(DesiredCapabilities dc) {
        if (driver == null) {
            driver = new AndroidDriver("<seetest_cloud_url>", dc);
        }
        return driver;
    }
}
```

Assume two different tests try to make use of it.
Let's consider Class A and Class B belong to Test1 and Test2 respectively and both tests are run in parallel. This would mean test methods, testA and testB both will grab the driver simultaneously and will lead to conflicts.

Many such examples can be given which shows that using static variables is not a good idea for parallel execution of tests and hence should be avoided.

**Parallelism and Test drivers**

Automation tests for Mobile and Web applications are generally achieved using test drivers provided by libraries like Appium/Selenium.

**Test driver for a Mobile application and browser**

Parallelism can be achieved at different levels, Let us consider a test suite whose composition looks like the diagram below.
1. Individual test methods like test-1-A and test-1-B can be run parallel.
2. Tests, Test-1 and Test-2 can be run parallel. In this case, individual test methods will be run in the parent thread.

Choosing the first strategy will mean that the test driver needs to be created in the test methods to avoid thread conflicts. This is becoming counterproductive because this will lead to frequent acquisition and release of devices affecting the performance of the test.

It is hence recommended from SeeTest.IO perspective to keep parallel execution in between tests i.e. multiple tests running in parallel which means test drivers to be setup per Test.

**Appium driver setup at class/test level using TestNG framework**

```java
@BeforeClass
public void setUp(@Optional("android") String os, ITestContext testContext) {
    DesiredCapabilities dc = DesiredCapabilities();
    driver = os.equals("android") ?
            new AndroidDriver(SeeTestProperties.SEETEST_IO_APPIUM_URL, dc) :
            new IOSDriver(SeeTestProperties.SEETEST_IO_APPIUM_URL, dc);
    ...
}
```

**Test driver for Browser only application**

SeeTest.IO cloud also provides browser only instances for testing.

Since browsers are less resource-intensive, it is fine to develop test suites with a view to run individual test methods in parallel.

In such scenarios language concept such as Java's ThreadLocal is a good choice which ensures returned drivers are per thread. Even if testA and testB are executed in parallel they will use a webdriver per thread thus avoiding conflicts.
public class WebTestExample {
  ...
  private ThreadLocal<WebDriver> webDriver = new ThreadLocal<WebDriver>();

  protected void createWebDriver() {
    // Launch and set for the current thread.
    webDriver.set(new RemoteWebDriver(new URL("https://cloud.seetest.io:443/wd/hub"), capabilities));
  }

  @Test
  public void testA() {
    // Get the driver for current thread.
    WebDriver driver = createWebDriver().get();
    ....
  }

  @Test
  public void testA() {
    // Get the driver for current thread
    WebDriver driver = createWebDriver().get();
    ....
  }
}