TUTORIAL FOR RECHECK-WEB

This tutorial is a step-by-step guide on of how to:

- Install and start using recheck-web and recheck.cli
- Create a first executable test case
- Make it repeatable by ignoring irrelevant changes

Furthermore, this tutorial will show some typical usage scenarios in development and test maintenance, thereby exemplifying the capabilities of recheck-web and recheck.cli.

INSTALLATION AND SETUP USING MAVEN

This tutorial assumes you have Java (https://www.java.com/) and Maven (https://maven.apache.org/) readily installed on your system. You can verify that by opening a terminal / CMD and running

```
java -version
mvn --version
```

The output should contain no error and show a Java version of 8 or above. Now you can create a new folder (e.g. `recheck-web-tutorial`) and a simple `pom.xml` file with the following content:

```
<project>
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.mycompany</groupId>
	<artifactId>recheck-web-tutorial</artifactId>
	<version>0.1-SNAPSHOT</version>

	<properties>
		<maven.compiler.source>1.8</maven.compiler.source>
		<maven.compiler.target>1.8</maven.compiler.target>
	</properties>

	<dependencies>
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>4.12</version>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>de.retest</groupId>
			<artifactId>recheck-web</artifactId>
			<version>1.0.1</version>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>org.seleniumhq.selenium</groupId>
			<artifactId>selenium-java</artifactId>
			<version>3.141.59</version>
			<scope>test</scope>
```

```
            </dependency>
        </dependencies>
</project>
```

With this, you can now turn to your favorite IDE (e.g. `mvn eclipse:eclipse`) and create your first test class. Before a Selenium test can be executed correctly, you first need to download a driver/browser executable according to your liking and operating system, e.g. Chrome (http://chromedriver.storage.googleapis.com/index.html).

Extract the archive to your hard drive. Note that for the chrome driver to work, you need a matching version of chrome installed on your system. Now we should be all set up.

CREATING YOUR FIRST RECHECK-WEB TEST CASE

A very basic test with Selenium and recheck-web could look like this:

```
package com.mycompany;

import org.junit.*;
import org.openqa.selenium.*;
import de.retest.recheck.*;

public class MyFirstTest {

  private WebDriver driver;
  private Recheck re;

  @Before
  public void setup() {
    re = new RecheckImpl();
    System.setProperty("webdriver.chrome.driver", "C:\\pathto\\chromedriver.exe");
    driver = new ChromeDriver();
  }

  @Test
  public void google() throws Exception {
    re.startTest();

    driver.get("http://google.com");
    re.check(driver, "open");

    re.capTest();
  }

  @After
  public void tearDown() {
    driver.quit();
    re.cap();
  }
}
```

The `@Before` annotated method creates both the `Recheck` instance to use, as well as the `ChromeDriver`. The `@Test` annotated method first tells recheck to start the test (calling `startTest`), then load the Google start page into chrome. Then it will recheck the current version of the page against the *Golden Master* (explained later) by calling `check` and giving it a semantic and unique identifier.

During a typical, more elaborate test, you would call `check` multiple times, each time with a unique identifier. Since differences are not that uncommon, we do not want our test to fail immediately. So the calls to the `check` method will gather all differences, but not immediately make the test fail. To make the test fail in case of differences, the `capTest` method is called at the end of the test. Should you forget to do so, then a message in the log will tell you so. After the test finishes, the `@After` method shuts down chrome by calling `driver.quit()` and makes recheck create a summary report file of all encountered changes by calling `cap`.

EXECUTING YOUR FIRST TEST CASE LOCALLY

Now you should be able to execute this test with `mvn test`. If everything was setup correctly, you should see the following error message the first time you run this test:

```
java.lang.AssertionError: 'com.mycompany.MyFirstTest':
No recheck file found. First time test was run? Created recheck file now, don't
forget to commit...
        at de.retest.recheck.RecheckImpl.capTest(RecheckImpl.java:137)
        at com.mycompany.MyFirstTest.google(MyFirstTest.java:30)
        at ... some more stack trace ...
```

As mentioned above, recheck works by comparing the current state of the software (i.e. the website) against a baseline called Golden Master from an earlier state of the software. If no such Golden Master can be found, recheck throws an error. This is the expected and desired behavior—e.g. imagine you forget to commit your Golden Master into your version control system, or a path changes and the Golden Master cannot be found anymore. In that case you would want your test to fail, not pass.

The test has to fail the first time it is executed. But during that first execution, recheck created the Golden Master at `src\test\resources\retest\recheck\com.mycompany.MyFirstTest\google.open.recheck`. In that folder you can now find an XML file containing every non-default attribute of every HTML-DOM element after rendering the website, together with a screenshot of the website. Now if you run your test again, all of those elements and attributes of the Golden Master are compared against the current HTML-DOM elements. Any non-ignored difference of any element makes the test fail. Doing so (again with `mvn test`) probably creates an output similar to this:

```
java.lang.AssertionError:
A detailed report will be created at 'target\test-
classes\retest\recheck\com.mycompany.MyFirstTest.report'. You can review the
details by using our GUI from https://retest.de/review/.

The following differences have been found in 'com.mycompany.MyFirstTest'(with 1
check(s)):
Test 'google' has 37 differences in 1 states:
open resulted in:
        A [About Google] at 'HTML[1]/BODY[1]/DIV[1]/DIV[3]/DIV[1]/DIV[1]/A[1]':
                ping: expected="some gibberish"
        ... many more differences ...
```

As you can see, recheck really shows you all differences of all attributes. This is comparable to using Git (https://git-scm.com) without configuration on an existing project: It shows you all

differences, including log files, binaries and many other files that are typically not of interest. Luckily, Git allows to easily ignore those differences using `.gitignore`.

Recheck works in a very comparable manner. You can simply ignore all of those volatile and non-relevant differences. Conveniently, recheck created a `.retest` folder in the project root. In there you can find an example `recheck.ignore` file. To ignore all those volatile elements and make the given test pass, you simply need to edit this plain text file. Putting the following content into the file should make the test pass:

```
attribute=ping
attribute=jsdata
attribute=data-.*
attribute=class
attribute=outline
attribute=transform
```

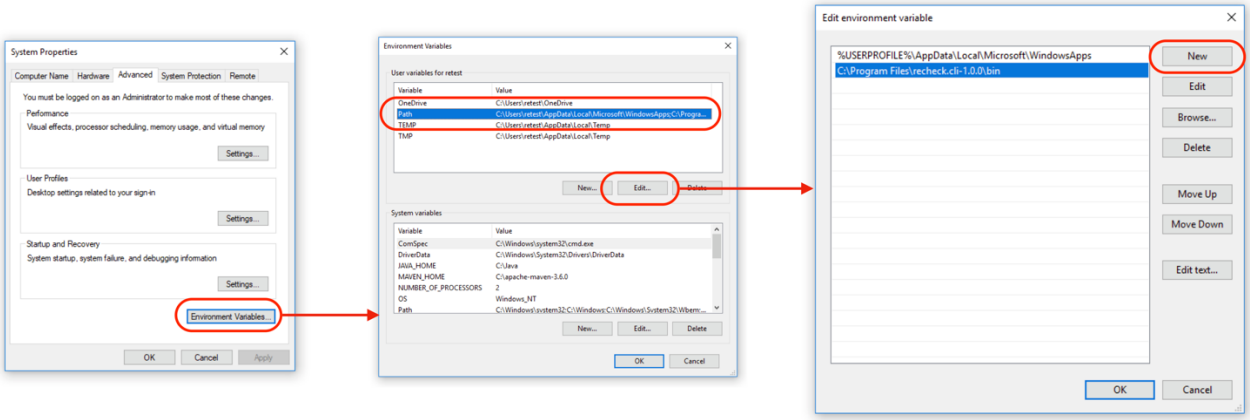
Note that Google is constantly changing its site, so you might need to add some more attributes. But as you can see, this is not difficult, and even wild-card ignore of attributes (using the Java Pattern mechanism: https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html) is possible. The default `recheck.ignore` file contains examples on how to e.g. ignore whole sub-trees of the DOM or specific attributes of specific elements. More information can also be found in the documentation at https://retest.github.io/docs/recheck/how-ignore-works-in-recheck/. This mechanism is very powerful and e.g. allows you to ignore the font of a text, but not the text itself. The semantic ignore mechanism is one of the core features of recheck, and we will explore it in more depth later.

Depending on what you specify in the ignore file, different testing scenarios can be realized. The general mechanism of recheck allows you to perform functional testing, cross-browser and cross-device testing, as well as visual regression testing. For these testing purposes, be careful with what you ignore. For pure functional testing, many CSS attributes can easily be ignored.

MAINTAINING THE GOLDEN MASTER

When you create an automated test, you want to ensure that it passes if nothing changes, i.e. that it is not flaky. But the real benefit of an automated test is to show you the effects of changes, especially unintended side-effects. However, during normal software development, many changes will be intended. With a version control system, you can simply approve of the changes, i.e. commit them. With recheck, we need the same functionality to maintain the Golden Master.

To be able to do so, you need to install e.g. the recheck CLI. You can download the latest release from GitHub: https://github.com/retest/recheck.cli/releases. To install it, simply unzip it e.g. to `C:\Program Files\recheck.cli-1.0.0`. Then you only need to add the `recheck.cmd` or `recheck` file to your path. This heavily depends on your operating system and version. E.g. for Windows 10, it works like this: Open Settings. In the search box enter “env” and select “Edit the system environment variables”. Then click on tab “Advanced” -> “Environment Variables” -> “Path” -> “Edit” -> “New”. Then add the path to the `recheck/bin` folder.



Now you can verify whether this worked correctly by typing `recheck --version` into a newly started CMD. The output should show the current version of recheck, i.e. `recheck CLI version 1.0.0`.

To easily generate changes to check for, open a browser and go to http://scratchpad.io. Opening the page will forward you to a unique URL (e.g. http://scratchpad.io/recheck-45678). Now we will edit our test from the previous section and replace the method name `google` with `scratchpad` and adjust the URL to load your newly created unique URL. The method body should then look similar to this:

```
  @Test
  public void scratchpad() throws Exception {
    re.startTest();
    driver.get("http://scratchpad.io/recheck-45678");
    re.check(driver, "open");

    re.capTest();
  }
```

Now you can run your test again using `mvn test`. As expected, it will fail the first time, as recheck cannot find a Golden Master for the test `scratchpad`, but it will create one under `src/test/resources/...`. Now running this test the second time will also fail, as the site contains a volatile URL. We will later see how you can treat that in a more sophisticated way, but for now we want to use our newly installed recheck CLI. In your CMD, go to the root folder of the project. Then type `recheck` to see all available commands. It will output something like:

```
C:\Users\retest\Desktop\recheck-web-tutorial>recheck
Usage: recheck [--help] [--version] [COMMAND]
Command-line interface for recheck.
      --help      Display this help message.
      --version   Display version info.
Commands:
  version     Display version info.
  diff        Display given differences.
  commit      Accept given differences.
  ignore      Ignore given differences.
  completion  Generate and display auto completion script.
  help        Displays help information about the specified command
```

Now we want to automatically ignore all irrelevant changes. To do that, simply type `recheck ignore --all target\test-classes\retest\recheck\com.mycompany.MyFirstTest.report`. This will automatically add the following line to your `recheck.ignore` file.

```
matcher: xpath=HTML[1]/BODY[1]/DIV[1]/P[3]/IFRAME[1], attribute: src
```

This makes recheck ignore just one attribute of one element, a Twitter-API related IFrame. Now rerunning your test should show a successful build and a passing test. So now you can use your regular browser to go to the URL you open in your test (e.g. http://scratchpad.io/recheck-45678) and edit the displayed content. For instance, let's replace `<h1>Welcome to <span>scratchpad.io</span></h1><br>` on the left-hand side of the website with `<h1>Welcome to <span>recheck</span></h1><br>`. Doing so and re-running the test should result in the following output:

```
The following differences have been found in 'com.mycompany.MyFirstTest'(with 1
check(s)):
Test 'scratchpad' has 7 differences in 1 states:
open resulted in:
        textnode [scratchpad.io] at
'HTML[1]/BODY[1]/DIV[3]/DIV[2]/DIV[1]/DIV[3]/DIV[23]/textnode[2]':
                text: expected="scratchpad.io", actual="recheck"
        DIV at 'HTML[1]/BODY[1]/DIV[3]/DIV[2]/DIV[1]/DIV[5]/DIV[1]':
                left: expected="210.125px", actual="173.75px"
                right: expected="252.813px", actual="289.188px"
                style: expected="left: 210.125px; top: 286px; width: 6.0625px;
height: 13px;", actual="left: 173.75px; top: 286px; width: 6.0625px; height: 13px;"
        TEXTAREA at 'HTML[1]/BODY[1]/DIV[3]/TEXTAREA[1]':
                left: expected="255.25px", actual="218.875px"
                right: expected="148.297px", actual="184.672px"
                style: expected="top: 285px; height: 13px; width: 6.0625px; left:
255.25px;", actual="top: 285px; height: 13px; width: 6.0625px; left: 218.875px;"
        at de.retest.recheck.RecheckImpl.capTest(RecheckImpl.java:137)
```

Now we see that scratchpad is generating and adapting a style attribute. Interesting enough, since all relevant style information is rendered and thus represented by individual CSS attributes, we can just add `style` to the ignored attributes. Rerunning the test now gives us the expected differences in `text` and as a result of that change also in `left` and `right`.

Now let's assume this is an intended change, and we now want to update our Golden Master. For that, we can now open a CMD in the project folder and run the following command:

```
recheck commit --all \target\test-
classes\retest\recheck\com.mycompany.MyFirstTest.report
```

The result of that call should be something like:

```
Updated SUT state file C:\Users\retest\Desktop\recheck-web-
tutorial\src\test\resources\retest\recheck\com.mycompany.MyFirstTest\scratchpad.ope
n.recheck
```

If there were more than one Golden Master, all of them would now be updated. If you had your Golden Master files in a version control system, they would now show as changed, and you would need to also commit the changes within e.g. Git. Now we can rerun the test (`mvn test`) and see whether our update worked—now the test should check whether the site contains "welcome to recheck" and pass accordingly.

To further show the functionality of the recheck CLI, let's adapt the content of the scratchpad again. Open your browser, and change the welcome message to `recheck-web`. Rerunning the test should again show the difference and produce a report file. You can use recheck CLI to display the contents of that file by running :

```
recheck diff target\test-classes\retest\recheck\com.mycompany.MyFirstTest.report
```

Doing so should result in an output similar to the following:

```
Checking test report in path 'C:\Users\retest\Desktop\recheck-web-
tutorial\target\test-classes\retest\recheck\com.mycompany.MyFirstTest.report'.
Reading JS ignore rules file from C:\Users\retest\Desktop\recheck-web-
tutorial\.retest\recheck.ignore.js.
Specified JS ignore file has no 'shouldIgnoreAttributeDifference' function.
Specified JS ignore file has no 'shouldIgnoreElement' function.

Test 'scratchpad' has 5 differences in 1 states:
open resulted in:
        textnode [recheck] at
'HTML[1]/BODY[1]/DIV[3]/DIV[2]/DIV[1]/DIV[3]/DIV[23]/textnode[2]':
                text: expected="recheck", actual="scratchpad.io"
        DIV at 'HTML[1]/BODY[1]/DIV[3]/DIV[2]/DIV[1]/DIV[5]/DIV[1]':
                left: expected="173.75px", actual="210.125px"
                right: expected="289.188px", actual="252.813px"
        TEXTAREA at 'HTML[1]/BODY[1]/DIV[3]/TEXTAREA[1]':
                left: expected="218.875px", actual="255.25px"
                right: expected="184.672px", actual="148.297px"
```

CONCLUSION

This tutorial showed you how to setup recheck-web and CLI and create and maintain a difference testing based automated test. This testing approach has many advantages over assertion-based testing. It is easier to set up: no identification of individual elements e.g. via XPath or id anymore. And it is easier to maintain—instead of manually copying and pasting adapted values to be the new expected, simply approve them via a single command. But the greatest benefit is that these tests are much more complete than their assertion-based counterparts: You cannot create an assertion for an unexpected change. But with difference testing, no change goes unnoticed.

A semantic comparison is also beneficial in comparison to the many pixel-based tools that implement a Golden Master-based testing approach. Today's websites are very dynamic. With the CSS `transition` attribute, you want to visually verify more than just the looks. Checking the CSS attributes directly points you to changes, that you then can verify manually. Also, instead of ignoring whole areas crudely, you can be very specific in what you want to ignore. You can ignore font type and font size but not the text. You can even implement a rule-based ignoring of changes (e.g. ignore changes with a difference of less than 5 pixel).

But this existing rule-based ignore will be part of another more in-depth tutorial of the advanced features of recheck. Such a future tutorial will also cover how to make your tests almost "unbreakable", by using the Golden Master to look up what the old state before a "breaking change" looked like. And it will show how to migrate existing tests and integrate with other tools, such as CI/CD or third-part testing frameworks, such as cucumber.

If you liked this tutorial, consider subscribing to our mailing-list at http://retest.dev, and get notified about follow-up tutorials.