8 Extending Open Source Frameworks for Advanced Functional Testing

THIS CHAPTER WAS CONTRIBUTED BY WIM SELLES, TEST AUTOMATION ENGINEER AT RABOBANK, NETHERLANDS

WIM SELLES, Test Automation Engineer contracted by Rabobank since 2010, lives in the Netherlands and works for a small company called deTesters. Wim has been using Perfecto since 2014. In March 2015 he started with the implementation of an automation framework with Protractor + CucumberJS to automate a hybrid app based on Angular. Before Perfecto introduced support for Protractor in October 2015, Wim and a colleague of his managed to set up an on-site device lab for Rabobank.

In his spare time Wim likes to make websites and tries to learn more and more about test automation. He is fond of sharing and contributes to a few open-source projects on GitHub.

INTRODUCTION

One of the key benefits of using open source testing tools is having the flexibility of customizing them for unique use cases and complementing them to meet special app testing requirements. There are plenty of examples on the market where organizations positioned test frameworks, such as Selenium WebDriver/ Grid and Appium, as technology foundation for custom tailored test solutions satisfying special requirements.

Within this chapter, a similar example of extending the Selenium WebDriver test framework will be provided. The value of this approach will be illustrated through a powerful open source test framework named Protractor.[1]

As part of mobile web, responsive web or hybrid app testing, it is necessary to deal with either unique objects or dynamic objects changing upon an event or other triggers. Sometimes Selenium cannot fully detect and provide the means for addressing such state changes and transitions. For this reason, the community developed Protractor which runs on top of Node.js and has been specially designed for testing AngularJS applications.

Protractor uses Selenium WebDriver and supports BDD frameworks like Jasmine[2], Mocha[3] or Cucumber[4] to execute tests on real browsers and devices.

FIRST STEPS WITH PROTRACTOR

What would a typical test solution based on Protractor look like? It merely consists of a small number of plain text files that need to be understood and authored accordingly.

The Spec.js file is the place for test automation engineers to develop their scripts or test scenarios. Conf.js is the file where configuration of the underlying Selenium WebDriver/ Grid is done. The Spec.js describes the targets for testing, e.g. mobile devices, browsers.

1 Protractor project page — http://www.protractortest.org/#/

2 Jasmine test framework — http://jasmine.github.io/

3 Mocha test framework — http://mochajs.org/

4 Cucumber framework — https://cucumber.io/

Figure 42 provides the example source code (**spec.js**) of a test that uses Protractor.

```
// spec.js
describe('Protractor Demo App', function() {
  var firstNumber = element(by.model('first'));
  var secondNumber = element(by.model('second'));
  var goButton = element(by.id('gobutton'));
  var latestResult = element(by.binding('latest'));

  beforeEach(function() {
    browser.get('http://juliemr.github.io/protractor-demo/');
  });

  it('should have a title', function() {
    expect(browser.getTitle()).toEqual('Super Calculator');
  });

  it('should add one and two', function() {
    firstNumber.sendKeys(1);
    secondNumber.sendKeys(2);

    goButton.click();

    expect(latestResult.getText()).toEqual('3');
  });

  it('should add four and six', function() {
    // Fill this in.
    expect(latestResult.getText()).toEqual('10');
  });
});
```

Figure 42: A Sample "spec.js" Protractor File[5]

Last but not least, the test execution environment needs to be configured. This is done by modifying the **conf.js** file. With this particular example in **Figure 43** a typical execution environment for web apps is defined and configured to use both Chrome and Firefox browsers.

Once a Protractor test solution has beens developed and the environment has beens setup, a test execution could be initiated through Jenkins CI or by manually typing ***"protractor conf.js"*** on the command line.

5 Source — Protractor Project Website

```
// conf.js
exports.config = {
  framework: 'jasmine',
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['spec.js'],
  multiCapabilities: [{
    browserName: 'firefox'
  }, {
    browserName: 'chrome'
  }]
}
```

Figure 43: A Sample "conf.js" File for Protractor

HOW RABOBANK WENT HYBRID WITH PROTRACTOR

Rabobank develops and maintains a hybrid[6] banking application. More than 75% of its implementation is based on WebView. The rest is native. The main reason for creating a hybrid banking application is to have a common platform powering up all use cases related to mobile, web, and mobile web.

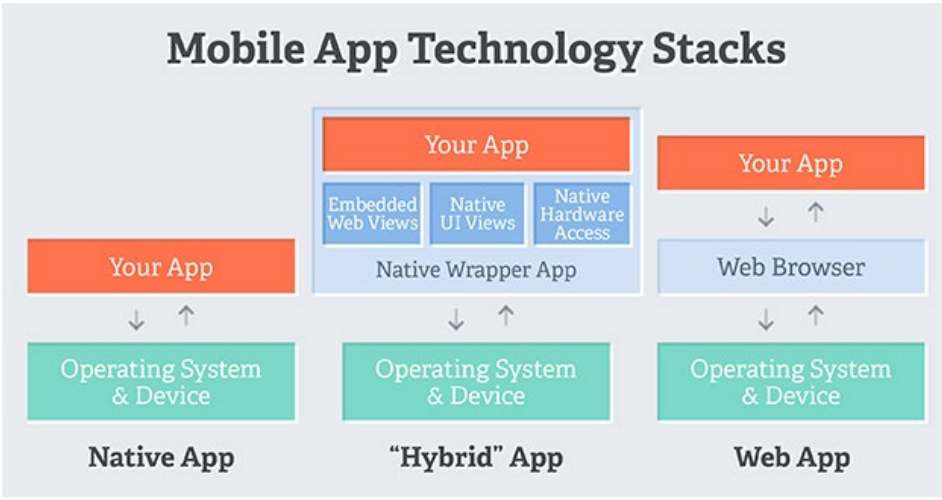


Figure 44: "Hybrid" Apps as Part of the Mobile App Technology Stacks

6 Example of Hybrid app architecture — https://myshadesofgray.wordpress.com/2014/04/15/hybrid-applications-and-android-native-browser/

WHY PROTRACTOR

A couple of years ago Rabobank tried to solve a set of problems. Keep in mind that in 2014 the market was way more immature in regards to offering the proper means for testing hybrid mobile applications. This said, after a careful analysis of the pain points the team got to the list of high level requirements towards the soon-to-come technical foundation for test automation.

1. The test automation solution needed to support the following capabilities:
   a. Test the app in parallel natively, in web and in mobile web modes
   b. Efficiently serve 15 different teams that are involved in the SDLC of this app
2. The solution should have easily provided mocked app data..
3. By supporting multiple teams, it needed to be accessible and shared among all of the aforementioned teams — both testers and developers.
4. The maintenance of the test environment should have been very easy and seamless for all of the teams.
5. Tests would have to be written in Gherkin.[7]

Of course, there were also other challenges, but taking the above into consideration, Rabobank decided to turn to Protractor as the shell framework for their solution.

SPECIAL REQUIREMENTS OR EXTENDING PROTRACTOR

A framework, all by itself, is not enough, at least not enough to fully support teams with automating the features they build. Take the following as an example.

The test toolset used by Rabobank included Protractor and CucumberJS. In order to test any given feature for (mobile) web, one needed to login to a secure environment and navigate to this feature. Testing the same feature within a native app actually imposed a number of extra steps, such as install and open the app, navigate to an entry point for the secure environment there, login, and navigate to the feature of interest.

7 Gherkin: https://github.com/cucumber/cucumber/wiki/Gherkin

If there are 15 teams, the wheel should not be reinvented 15 times (and even more, taking into account how creative developers could be). Especially when the goal for each team is to create, deliver, and contribute **their feature** instead of "wasting time" with test tools/ authentication methods. This was the starting point for laying the foundations of an utility library called "protractor-utils".

Rabobank wanted a common place for dependency management of their test tools. It should have also boosted sharing test automation knowledge throughout the organization. Thus, each team should have used the very same version of the test toolset to prevent technical tool depths. From one hand side, Protractor was evolving really fast. On the other, Selenium WebDriver was aggressively striving to support the fast release cycles of modern web browsers.

By using "protractor-utils" the authentication mechanism required for testing resulted in a single method called "logon()", as illustrated in **Figure 45.** Based on the capabilities passed to the Selenium WebDriver, the method itself is smart enough to determine if it needs to logon to (a mobile) web or a native version of the app. Well, in fact "protractor-utils" takes care of the heavy lifting part and automation engineers just need to provide credentials. Note that the documentation of the method is generated automatically.

Methods

logon()

Exposes a promise to logon as a customer to the teamserver. When the baseUrl contains 'localhost' logon is not needed, the promise is resolved and the given baseUrl is loaded.

Returns

promise — The promise is resolved as soon as the login has been executed

Example

```
// Example of the params
{
    customerID: 110800000000,
    redirectUrl: '/path/to/your/feature',
    securityLevel: 3,
    showTour: false,
    siteContext: '/secure/environment'
}
 //Feature-file
 Given I logon with valid credentials

 //Stepfile:
 this.Given(/^I logon with valid credentials$/, givenILogon);
 function givenILogon() {
    var params = {
        customerID: '110800000000',
        redirectUrl: '/path/to/your/feature',
        securityLevel: 3,
        showTour: false,
        siteContext: '/secure/environemt'
    };
    return protractorutils.logon(params);
 }
```

Figure 45: Login Functionality Based on "protractor-utils"

From that point on, this useful library evolved into a npm module for Protractor that has being used by multiple teams for the past two years. They found it useful for developing Cucumber-based test automation and for performing parallel testing on both mobile and web platforms. Last but not least, the library also helps with producing standard test execution reports for all teams.

"protractor-utils" now contains:

- All of the dependencies (the underlying test frameworks), such as Protractor, CucumberJS, etc.
- "Default" configurations for Protractor and CucumberJS
- Utility methods to ease the testing of frontend projects
- Means facilitating reporting (hooks and basic test execution report generation)
- Configuration files
- Examples

In addition, as part of "protractor-utils" extra npm-modules were developed to bring in value added capabilities for:

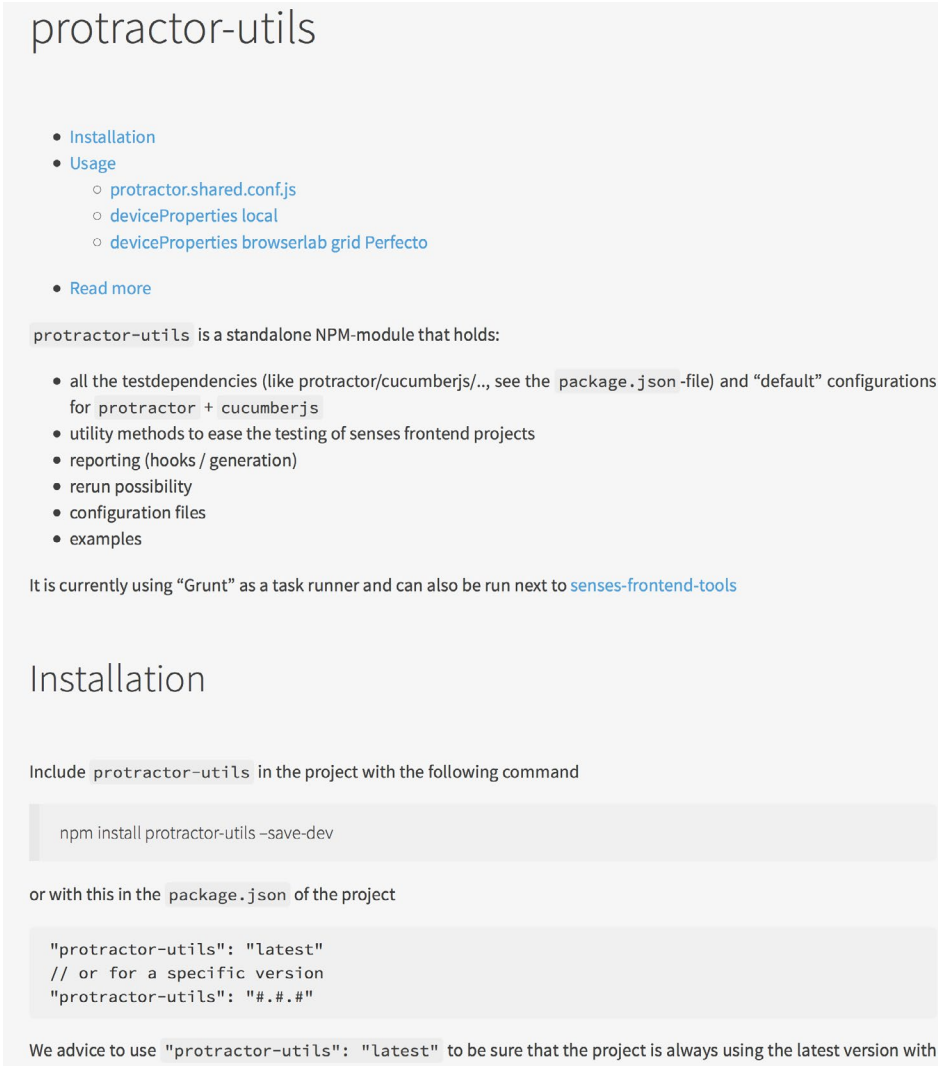
- Re-running of flaky tests with "protractor-flake"
- Filtering in the device lab to retrieve the correct Selenium WebDriver capabilities
- Image comparisons (screenshots/ elements) created with (mobile) web or native apps (based on a fork of "pix-diff").

Support for Cucumber, Firefox, Internet Explorer, Microsoft Edge and Safari was introduced gradually

As you could see in **Figure 46,** "protractor-utils" is fully documented. Each method has its own automatically generated and well-structured description.

Figure 46: An Extract from the Documentation of "protractor-utils"

Furthermore, the usage of "protractor-utils" is explained in multiple markdown README.md files provided as part of the module. The contributors maintain them as clear and understandable guides for getting started with the basics really quickly, as also evident in **Figure 47.**



protractor-utils

- Installation
- Usage
  - protractor.shared.conf.js
  - deviceProperties local
  - deviceProperties browserlab grid Perfecto
- Read more

`protractor-utils` is a standalone NPM-module that holds:

- all the testdependencies (like protractor/cucumberjs/.., see the `package.json`-file) and "default" configurations for `protractor` + `cucumberjs`
- utility methods to ease the testing of senses frontend projects
- reporting (hooks / generation)
- rerun possibility
- configuration files
- examples

It is currently using "Grunt" as a task runner and can also be run next to senses-frontend-tools

Installation

Include `protractor-utils` in the project with the following command

```
npm install protractor-utils –save-dev
```

or with this in the `package.json` of the project

```
"protractor-utils": "latest"
// or for a specific version
"protractor-utils": "#.#.#"
```

We advice to use `"protractor-utils": "latest"` to be sure that the project is always using the latest version with

Figure 47: Installation Procedure for "protractor-utils"

PROTRACTOR AND FLAKY TESTS[8]

Whoever is in the business of assuring the quality of mobile and web applications understands that testing against such dynamic platforms often returns false negatives or failures. While some of the reasons might be due to incorrectly

8 Protractor flaky tests mechanism repo — https://github.com/NickTomlin/protractor-flake/blob/master/README.md

implemented tests, others directly relate to changes of the target environment or its availability, more precisely, to its lack of availability.

For example, a target device may malfunction or the installation of a native app may fail, or probably a browser session could not be initiated as desired. Keep on using your imagination and you will get to a long list of potential glitches that may (and will) cause test execution failures at some point. Thus, it is hard to think of mobile target environments in very friendly terms. What is even worse, one needs to count on them to get the job done. Tricky, isn't it?

This is why Nick Tomlin developed a NPM-module that would re-run potentially flaky Protractor tests before they would be announced as "failures". Initially "protractor-flake" did not meet the requirements of Rabobank, because it only supported Jasmine and Mocha tests. Therefore, CucumberJS parser and CucumberJS documentation[9] were introduced next, so that the NPM module could be adopted and used by Rabobank as well.

By bringing in this into "protractor-utils" flakiness could be reduced significantly by just appending a single command line option.

```
protractor:subtask --rerun-flake --attemps=amount
```

Apparently, this feature enhances efficiency, saves debugging time, and even more so, if you consider the large number of Rabobank teams working on this project.

FILTER ON THE DEVICE LAB TO RETRIEVE CORRECT CAPABILITIES (PERFECTO CLOUD)[10]

This NPM module has two clear goals:

- Always request devices with a standard/ controlled set of capabilities.
- Target specific device capabilities, such as device model, OS, browser (name), etc.

An internal NPM-module for introspecting and filtering the device Cloud of

9 https://github.com/NickTomlin/protractor-flake/blob/master/docs/cucumber.md

10 Querying a Cloud for specific devices — https://community.perfectomobile.com/posts/1072813

Rabobank was developed. Thus, the module returns the desired capabilities to be used by the test code.

With “protractor-utils” one could select all iOS devices by appending a single command line option, just like this.:

```
protractor:subtask –capabilityFilter=’[{“platform-
Name”:”iOS”}]’
```

What if selecting more than one specific devices would be necessary? No issues here, since the capability filter is able to accept any number of filtering criteria.

```
protractor:subtask –capabilityFilter=’[{“model”:”i-
Phone-6S”}, {“model”:”Galaxy S6”}]’
```

Running some of the commands above will produce a JSON-formatted file describing all of the desired capabilities needed by Rabobank in order to start a device in the Cloud. It is based on a predefined template. One could refer to the extract in **Figure 48** to get the full picture. This way multiple implementations for starting devices will be prevented and there is no need for all of the feature teams to keep and maintain this kind of knowledge.

The above NPM module is the central place for:

- Maintaining and evolving the knowledge for managing all of the devices used
- Ensuring the same device conditions for every test, i.e. a more stable target environment
- Updating the version of the tested app without notifying each team in advance

```
deviceTemplate = {
    applicationName: '<# var applicationName = environment +\'_\'+ manufacturer +\'_\'+ model; #><#= applicationName.toLowerCase().replace(\/ \/g, \'_\') #>',
    appPackage: '',
    app: '',
    autoInstrument: true,
    available: '<#= available #>',
    browserName: '',
    bundleId: '',
    deviceProperties: {
        appType: '<#= appType #>',
        deviceId: '<#= deviceId #>',
        deviceType: '<#= deviceType #>',
        environment: 'pm',
        inScope: '<#= inScope #>',
        os: {
            name: '<#= os.toLowerCase() #>',
            version: '<#= osVersion #>'
        },
        resolution: '<#= resolution #>'
    },
    fullReset: true,
    inUse: '<#= (status === \'Not Connected\' || status === \'Ready to connect\') ? false : inUse #>',
    logName: '<#= manufacturer #> <#= model #> app',
    manufacturer: '<#= manufacturer #>',
    model: '<#= model #>',
    password: '',
    platformName: '<#= os #>',
    platformVersion: '<#= osVersion #>',
    reserved: '<#= reserved #>',
    screenshotOnError: true,
    status: '<#= status #>',
    user: '',
    useHybridAppAsBrowser: true,
    windTunnelPersona: 'Empty'
};
```

Figure 48: An Example of a Device Template File

IMAGE COMPARISON (SCREENSHOTS/ UI ELEMENT SCREENSHOTS)

Rabobank also needed the ability to compare screens/ UI elements with each other. Researching the topic on the Internet in 2015 they found the library Pix-Diff. The core of Pix-Diff[11] could not be used for testing the hybrid app of Rabobank with browsers different from Chrome. Hence, they forked Pix-Diff and added Cucumber, hybrid app and additional browser support. Screens/ UI elements could now be compared during the "normal" test runs with just a single line of code.

Compare a screen:

```
expect(browser.imageComparison.checkScreen('exampleP-
age')).toEqual(0);
```

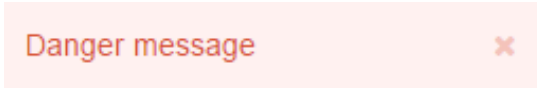
Compare an UI element:

```
expect(browser.imageComparison.checkElement(element(By.
id('title')), 'examplePageTitle')).toEqual(0);
```

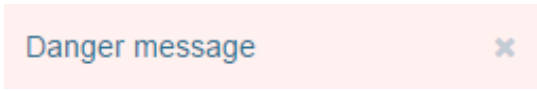
11 Pix-Diff repository — https://github.com/koola/pix-diff

For example, one could compare a baseline UI element image with the actual UI element image as it would be faced by anyone exploring this version of the app manually. In the particular example depicted in **Figure 49,** the actual UI element screenshot deviates from the mandatory color for alerting messages. This is detected and reported by the tool as a substantial difference.

- **Baseline:**



- **Actual screenshot:**



- **Difference:**

Figure 49: An Example of Detecting Deviations against Baseline Requirements

As a consequence, in 2016 the following framework features were contributed to the Pix-Diff open source project.

- CucumberJS
- Multiple browser sessions
- Appium as the backbone for image comparison on mobile devices

At the end of 2016 Wim announced and published his own open source NPM module "protractor-image-comparison"[12] with even more functionality that Rabobank and others could benefit from.

THE TEST AUTOMATION STRATEGY OF RABOBANK

The way the app has been developed within Rabobank requires a well-defined

12 NPM: https://www.npmjs.com/package/protractor-image-comparison, GIT: https://github.com/wswebcreation/protractor-image-comparison

process, constant alignment and discipline amongst all of the involved teams. Since each of them would contribute a specific component to the app while utilizing a common framework and infrastructure, it is important for each team to take care of its own tasks without neglecting dependencies that would be the foundation for other teams to perform their near future activities.

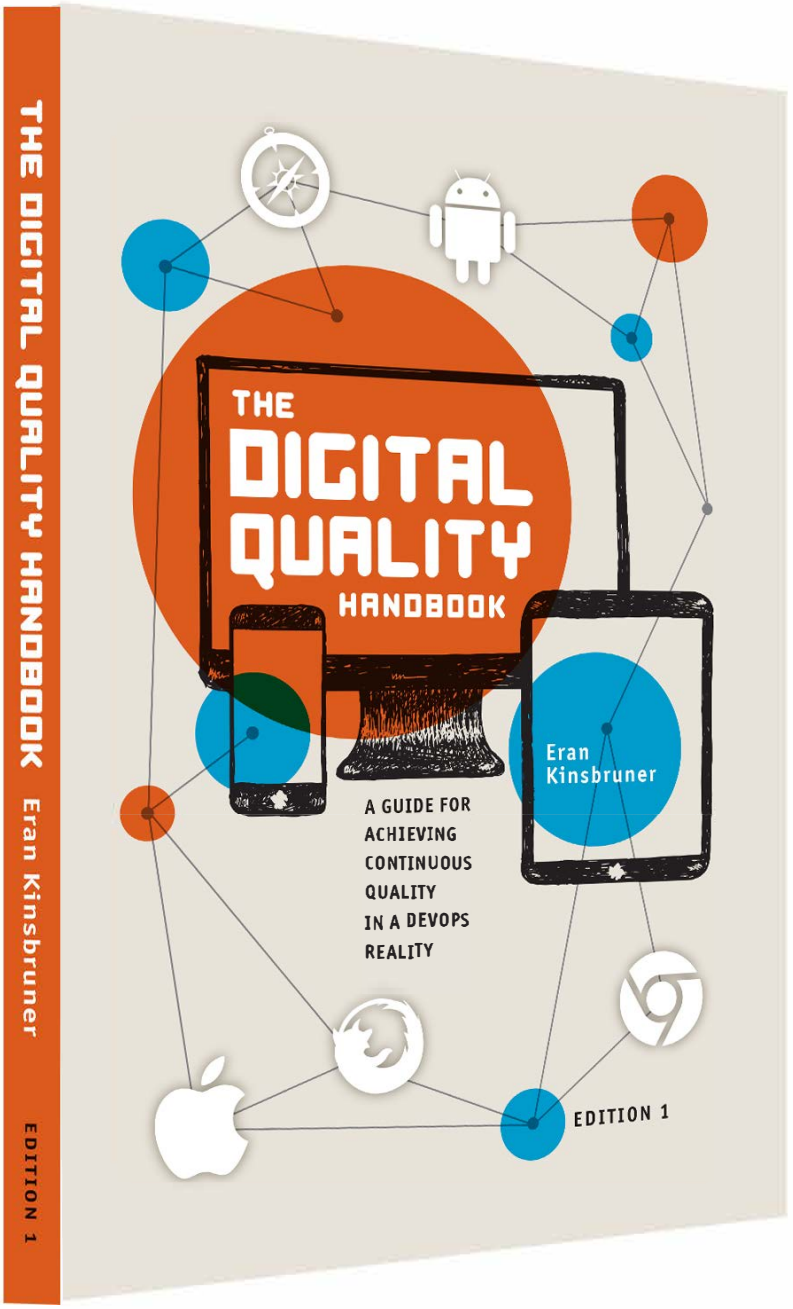
A prominent example for such a dependency would be the "constantly" and rapidly changing HTML code. Within the timeframe of a week the HTML code for a date field might change from three input fields (denoting day, month, and year) to three select options. Such a change would definitely break all of the tests relying on this specific date field.

In order to mitigate and manage this risk, Rabobank agreed that each team would create PageObjects with a consistent API that other teams could also reuse for testing their own features. As long as the API would not change, dependent tests would not break when the new version of the aforementioned date field would be introduced.

SUMMARY

At the end of the day, by extending the Protractor framework Wim managed to offer a robust framework satisfying the needs of 15 feature teams consisting of both developers and testers. It was designed from ground zero to seamlessly integrate within the target environment and IDEs used on daily basis by the developers at Rabobank. It was also easily connected with the Jenkins CI workflow as well.

Open source, best practices, and industry standards (such as PageObjects, Cucumber, and a number of open source modules) allow testing teams to develop and deliver test automation code and enhancements in a friendly and easy, yet controlled, manner without any worries about the underlying test engine.



PRACTICAL WAYS OF DELIVERING FLAWLESS APP EXPERIENCES

INSIDE

THE DIGITAL QUALITY HANDBOOK

A 360-DEGREE VIEW OF DIGITAL QUALITY WITH RECOMMENDATIONS FOR AGILE & DEVOPS PRACTITIONERS

WRITTEN BY A CONSORTIUM OF INDUSTRY EXPERTS

HTTP://BOOK.PERFECTO.IO/