Protractor E2E Tests for the NRP Frontend

Yann Fabel

August 10, 2015

Contents

1 Preface

The objective of this document is to give an overview of end-to-end testing of the Neurorobotics Platform using Protractor. The NRP is a subproject of the Human Brain Project, providing a web-based platform, where simulated experiments can be performed. In such experiments a brain model is coupled to a simulated roboter, interacting within a simulated environment. End-to-end testing involves ensuring that that integrated components of an application function as expected. In this context, the use of contiunous automated testing of the platform's frontend has a high value, as minor changes may lead to unexpected errors.

2 Protractor

2.1 Introduction to Protractor

Protractor is a JavaScript end-to-end test framework for web pages, primarily AngularJS applications, built on top of WebDriverJS. Thus it works together with Selenium, which is a software testing framework for web applications, supporting several programming languages, such as Python, Ruby, Java and C#. Protractor uses a real browser to run automated tests against the webpage/application, simulating user interaction. When working with Protractor, it is necessary to specify how to connect to the browser drivers, which will start up and control the browsers you are testing on. It is recommended to use a Selenium Server, otherwise you can also connect directly to the browser drivers[1](only with Chrome and Firefox). As a Node.js program Protractor supports Jasmine, Mocha and Cucumber test frameworks (by default is uses the Jasmine framework). Moreover, the common operating systems like Linux, MacOS or Windows are supported.

2.2 The Selenium Server

The Selenium server is part of the browser automation framework called Selenium. Apart from the server, Selenium also includes the WebDriver APIs and the WebDriver browser drivers. While executing Protractor the Selenium server acts as a proxy between the test scripts and the browser driver. It forwards commands from your script to the browser and returns the responses. There are two possibilities for working with a Selenium Server, either by using the standalone Selenium Server or by using a remote Selenium server. The standalone server can be installed and started directly with Protractor on a local machine. A suitable option for using a remote server would be Sauce Labs[1] as it is already supported by Protractor.

[1] `https://angular.github.io/protractor/#/server-setup`

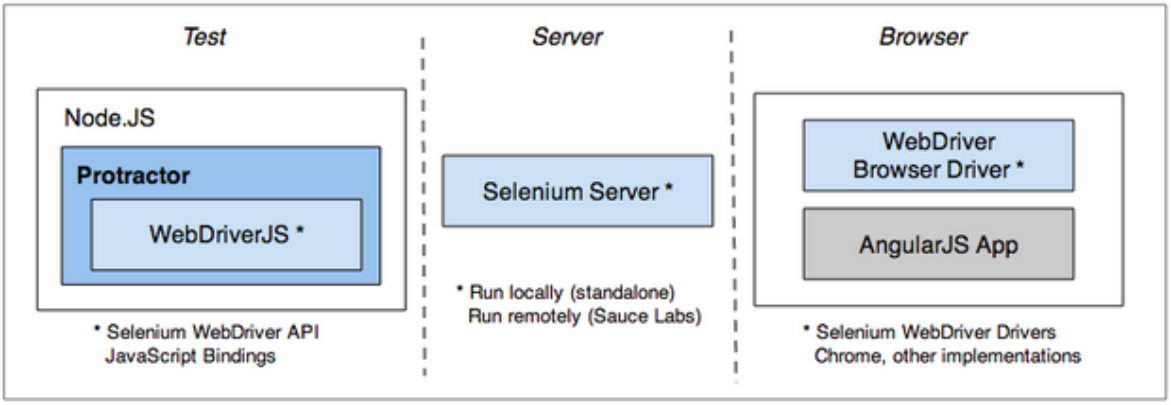


Figure 1: `https://angular.github.io/protractor/#/infrastructure`

2.3 Setup of Protractor

To run Protractor, Node.js has to be installed.[2]. Node.js is a server-side platform for network applications, built on Chrome's JavaScript runtime environment. With the provided paketmanager npm Protractor can be globally installed by using the command:

```
$ npm install -g protractor
```

Afterwards the command line tools `protractor` and `webdriver-manager` can be utilized. For running a standalone Selenium server on a local machine, JDK is required. To install and start the server you use the following commands:

```
$ webdriver-manager update
$ webdriver-manager start
```

2.4 The conf.js File

The configuration file `conf.js` is used to run the test and specify its properties. For instance, it contains the address of the testfile `spec.js` and the applied Selenium server protractor is talking to. Detailed information for the configuration file can be found here: `https://github.com/angular/protractor/blob/master/docs/referenceConf.js`. An example with the minimum configuration could look like this:

```
// conf.js
exports.config = {
        seleniumAddress: 'http://localhost:4444/wd/hub', //Default
        specs: ['spec.js']}
```

[2] `https://nodejs.org/`

2.5 The spec.js File

The spec file includes at least one 'describe'-function, which creates a test. This test can contain various test cases, that are initialized by an 'it'-function. These two functions are part of the Jasmine framework. Usually there is an expect function at the end (or in between), where you define an assertion you want to check. An example for a spec file is given below, where the title of a webpage is requested:

```
// spec.js
describe('HBP homepage', function() {
    it('should have the right title', function() {
        browser.driver.get('https://www.humanbrainproject.eu/');
        var title ='The Human Brain Project - Human Brain Project';
        expect(browser.driver.getTitle()).toEqual(title);
    });
});
```

To execute the test you switch to the directory where the conf.js file is stored and enter the following command into the console:

```
$ protractor conf.js
```

The console output would be:

```
$ protractor conf.js
Using the selenium server at http://localhost:4444/wd/hub
[launcher] Running 1 instances of WebDriver
.

Finished in 4.036 seconds
1 test, 1 assertion, 0 failures

[launcher] 0 instance(s) of WebDriver still running
[launcher] chrome #1 passed
```

2.6 Functionality of Protractor

2.6.1 Promises

Since Protractor is wrapped around WebDriverJS and WebDriver commands are asynchronous, you have to consider a control flow, where functions return promises instead of primitive values.[3] Hence, the control flow is a queue of pending promises that manages the order of execution. For Protractor/WebDriver functions there is no need of an explicit catenation of promises to assure the right order of execution, but the control flow schedules these commands top down, always waiting for the callback of the previous promise. However, this does not

[3] https://angular.github.io/protractor/#/control-flow

hold for other functions. Thus, the order of execution does not have to match with the order of commands in the code, since the callback of the promises may take some time. To avoid a wrong order, you have to pyramid these commands by using 'then'-functions.[4] As you can see in the following example, the output messages 2 and 3, which are not dependent of a promise get logged at first:

```
describe('HBP homepage', function() {
    it('should have the right title', function() {
        browser.driver.get('https://www.humanbrainproject.eu/')
                .then(function() {
                        console.log('Message1');
                });
        console.log('Message2');
        var title ='The Human Brain Project - Human Brain Project';
        expect(browser.driver.getTitle()).toEqual(title);
        console.log('Message3');
    });
});
```

Console output:

```
$ protractor conf.js
Using the selenium server at http://localhost:4444/wd/hub
[launcher] Running 1 instances of WebDriver
Message2
Message3
Message1
.

Finished in 4.732 seconds
1 test, 1 assertion, 0 failures

[launcher] 0 instance(s) of WebDriver still running
[launcher] chrome #1 passed
```

2.6.2 Protractor for Non-Angular pages

As Protractor was implemented mainly for AngularJS application there are some characteristics you have to consider. For instance, the command `browser.driver` instead of `browser` is used for interacting with non-Angular pages, as you access WebDriver directly. Furthermore, experience showed that Protractor's automatic waiting for pending tasks to finish, did not work at all times for non-Angular pages.

[4] `http://tritarget.org/blog/2012/11/28/the-pyramid-of-doom-a-javascript-style-trap/`

2.6.3 Timeouts

There are lots of reason why a timeout error may occur, for a first reference see: `https://angular.github.io/protractor/#/timeouts`. A quick solution might be to change the default value to a greater value. The use of `\$timeout()` in AngularJS controllern may also lead to problems, see: 'Waiting for Page Synchronisation' at `https://github.com/angular/protractor/blob/master/docs/timeouts.md`.

3 Using Protractor for the NRP Frontend

3.1 Login Process

For devoloping our tests of the Neurorobotics Platform we regularly pulled updates from the main development branch to a local machine and accessed it by creating a local HTTP server using `grunt serve`. Besides we were working with a standalone Selenium Server. The complete code, i.e. the files `conf.js` and `spec.js` can be found in the git repository of the NRP.
By visiting the NRP start page (for local tests using grunt it would be `http://localhost:9000/#/`) you automatically get redirected to the non-Angular login page, thus we had to make sure that Protractor waits until the redirection had finished, before sending the authentication data. Afterwards, the test checks, if the browser returns to the start page and accesses the Experiment Viewer, where simulations can be performed.
To commit valid authentication data without storing it in public code fragments, you add username and password as parameters to the `protractor` command:

```
$ protractor conf.js --params.login.user='fabel'
--params.login.password=$PW
```

As soon as the NRP is publicly available, the login process will not be obligatory.

3.2 Creating new Simulations

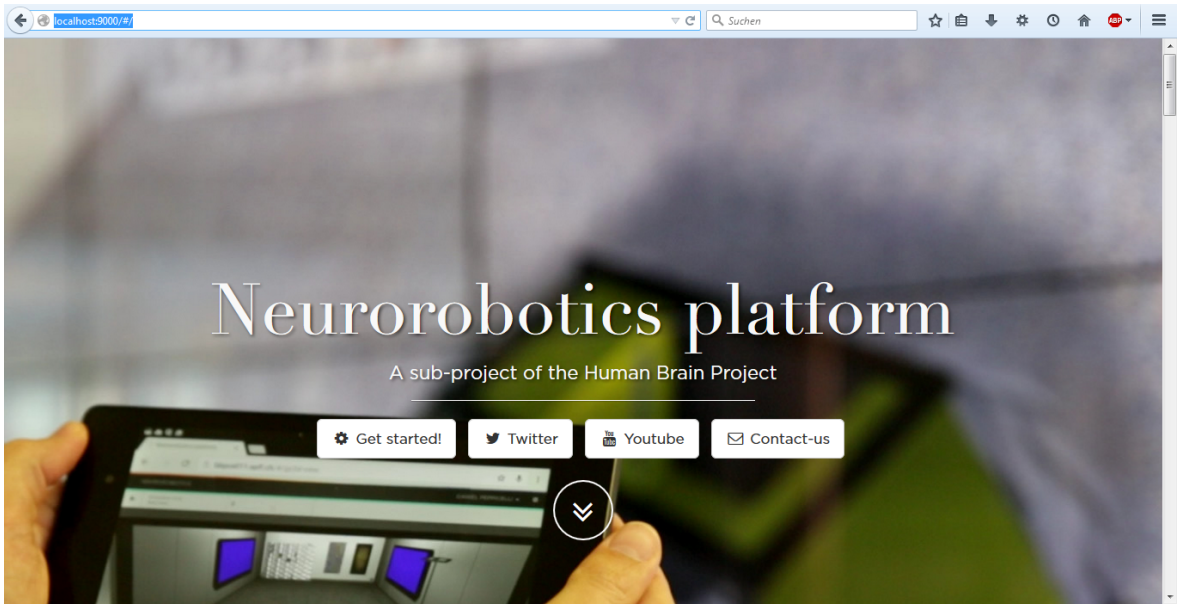
Arbitrarly, we chose the 'Husky Braitenberg experiment' for testing. Hence, Protractor starts a simulation by clicking the button 'Create new simulation'. As the start of a simulation involves various components, like the brain model, the simulated robot and the simulated world, which are in general not computed locally but at different servers, the loading of the experiment takes some time. To prevent Protractor from failing the test because of timeouts, some default values had to be changed and we set a timeout manually, to also check that the loading does not exceed a maximum time of 45s.

3.3 Start and Stop Experiment

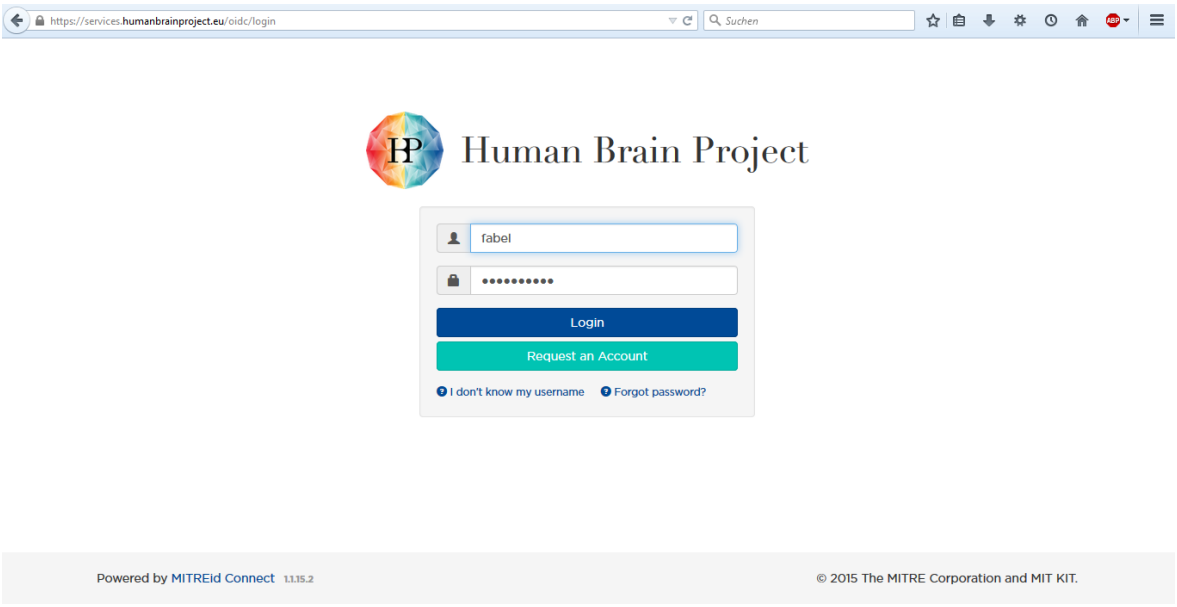
After the experiment has loaded totally, the experiment can be started by clicking the 'play' button and the robot begins interacting with its environment. Another test case of Protractor is to check that the time is running while the experiment is running, whereas the time stands still while the experiment is paused. At the end, the simulation should be terminated correctly by clicking the 'stop' button and the browser should return to the Experiment Simulation Viewer.

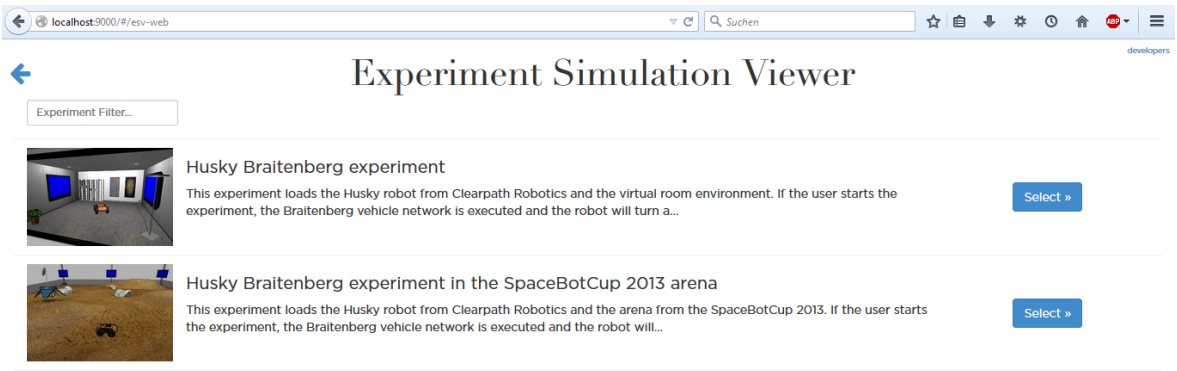
3.4 Screenshots and Console Output of Testing the NRP

In the following pages you can have a look at how Protractor controls the browser. The current output of the console is inserted below the screenshots.

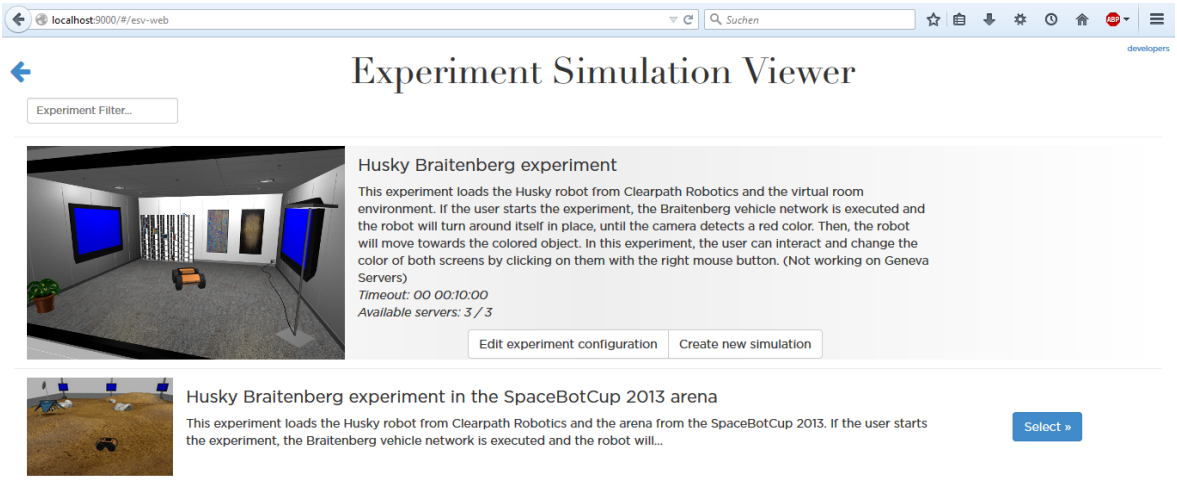



```
Using the selenium server at http://localhost:4444/wd/hub
[launcher] Running 1 instances of WebDriver
Started
2015-08-03 11:03:16.046: [TC] Starting login testcase
```

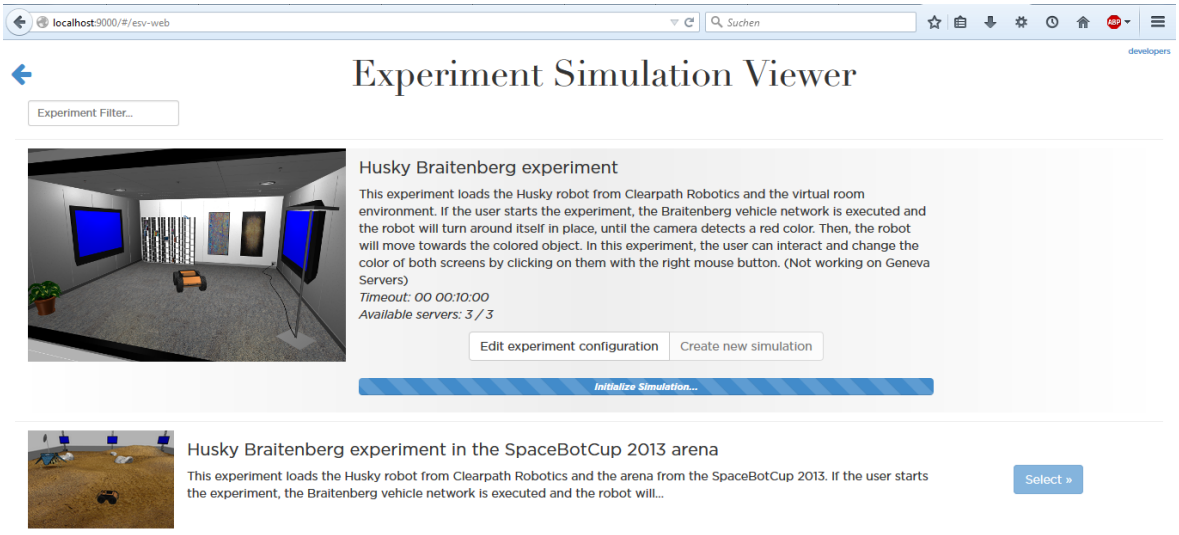




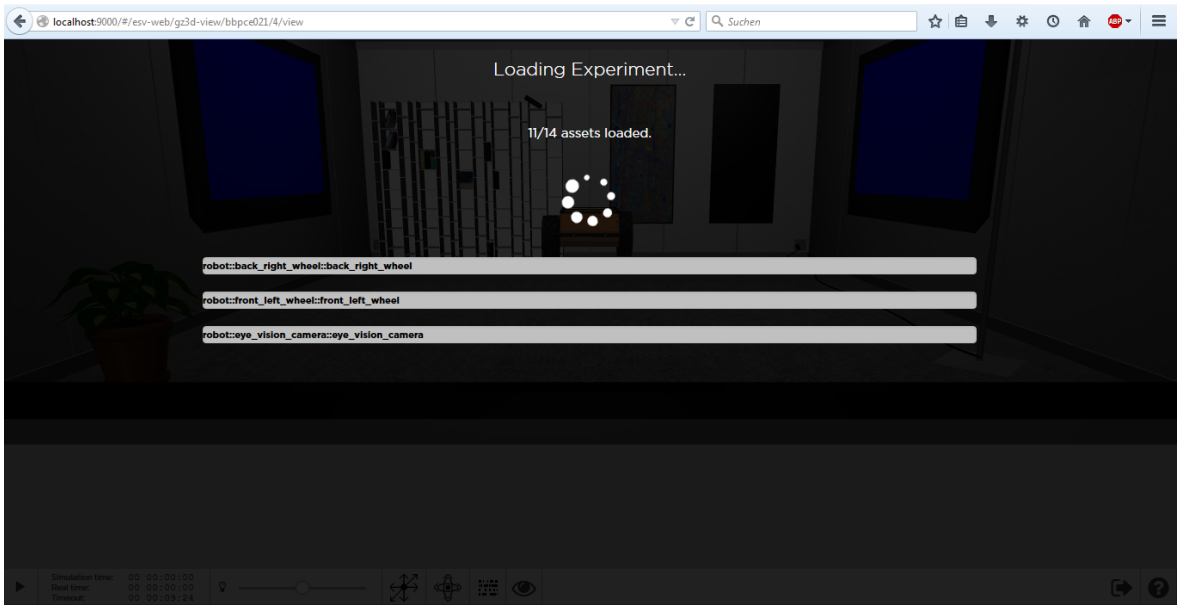

```
2015-08-03 11:03:41.996: Login successful
2015-08-03 11:03:42.280: Now we are on http://localhost:9000/#/esv-web
.
1 spec, 0 failures
Finished in 26.368 seconds
2015-08-03 11:03:44.037: [launcher]
2015-08-03 11:03:44.038: [launcher]
```



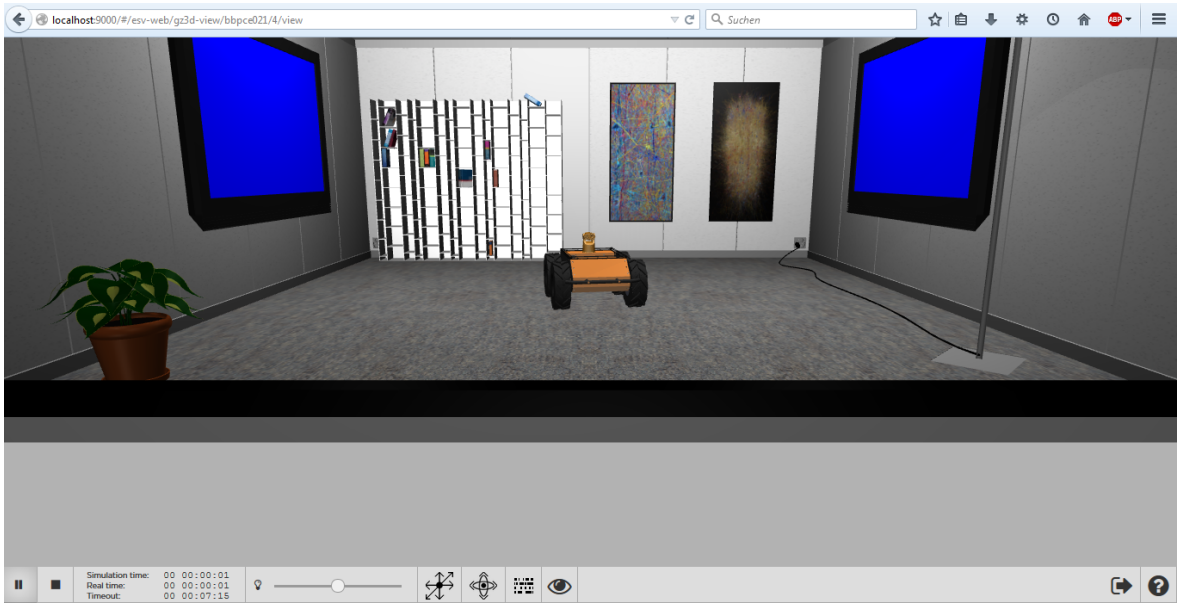

```
.2015-08-05 15:41:44.653: [TC] Starting Husky start-stop testcase
2015-08-05 15:41:54.659: Waited for 10s now to be sure page is completely loaded
2015-08-05 15:41:55.202: Clicked on the (first) experiment containing text: "Hus
ky Braitenberg experiment"
```



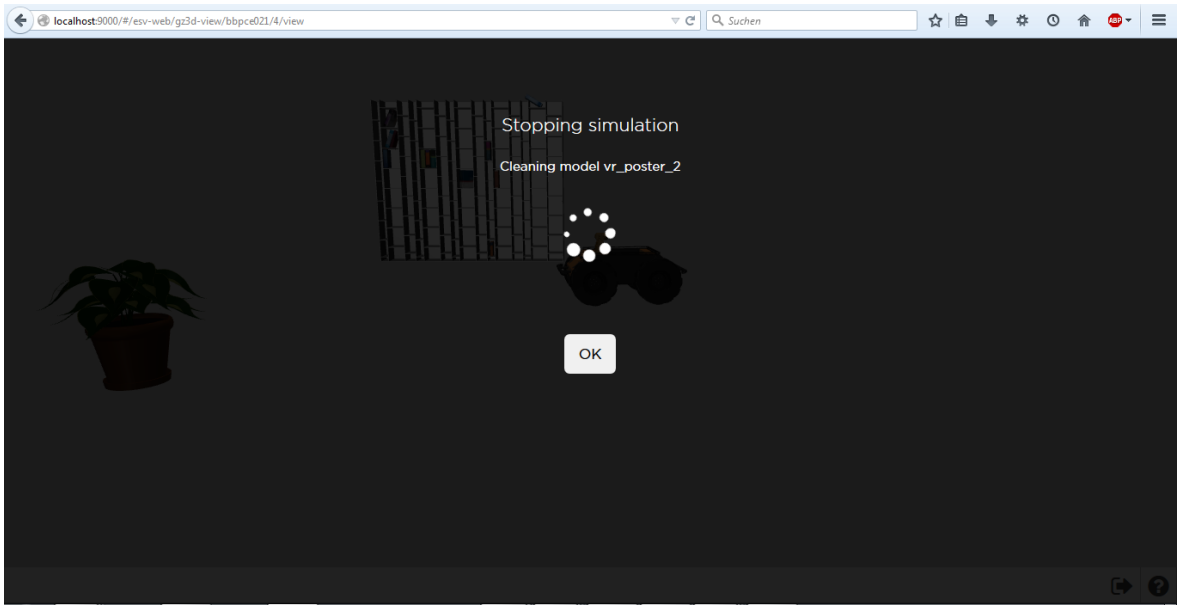

```
2015-08-05 15:41:55.923: Clicked on button: "Create new simulation"
```



2015-08-05 15:42:40.928: Waited for 45s



2015-08-05 15:42:48.585: Simulation started (time should change)
2015-08-05 15:42:52.067: Simulation Time: 00 00:00:00
2015-08-05 15:42:59.880: Simulation Time: 00 00:00:07
2015-08-05 15:43:02.665: Simulation paused (time should not change more than 1s)
2015-08-05 15:43:05.266: Simulation time: 00 00:00:11
2015-08-05 15:43:12.254: Simulation time: 00 00:00:11
2015-08-05 15:43:12.255: Deviation = 0




```
2015-08-05 15:43:14.582: Simulation stopped
2015-08-05 15:43:17.247: Waited for 10s now for splash screen to vanish
2 specs, 0 failure
Finished in 108.254 seconds
2015-08-05 15:43:17.869: [launcher]
2015-08-05 15:43:17.869: [launcher]
```

4 Unsolved Problems

4.1 Integration into Jenkins

The mayor problem of including the automated testing with Protractor into a continuous-integration server, i.e. Jenkins, is that there is no graphical interface and therefore also no browser installed. Thus, the tests have to run in headless mode. Furthermore, the requirements of the NRP are relatively high, regarding the complexity of simulation, precisely the representation of the 3-dimensional environment using the API WebGL.
A first attempt was to use a headless WebKit called PhantomJS, which is theoretically possible,[5] but did not work for our case, as it does not support WebGL. The Jenkins Plugin Xvfb, that emulates a dumb framebuffer using virtual memory led to the same result. Another problem regarding Jenkins is that we do not want to store personal login data on a collaborative server, as the tests should run automatically on a regular basis without someone having to enter his personal data everytime. A possible solution, which has already been requested, would be a limited dummy account, only for testing the frontend.
We also created a ticket regarding the need of Chrome for our testing: `https://bbpteam.epfl.ch/project/issues/servicedesk/customer/portal/3/HELP-3807`

[5] `https://github.com/angular/protractor/blob/master/docs/browser-setup.md`

4.2 Setting up a new Server

Instead of using the Jenkins server, another idea was to set up a new server with chromedriver installed, where the tests could be executed with Selenium. However, there still was the problem with WebGL, since Chromedriver does not support it either and we would also need a graphic board using a 'complete' browser like Chrome.

4.3 Using Sauce Lab

A possible solution for running the tests automatically on a regular basis might be by using Sauce Labs.[6] Via a Sauce Lab account, Protractor can connect to a remote Selenium server using its capabilities. You have to set the following options in your configuration file:

```
    sauceUser: 'SauceLabUsername',
    sauceKey: 'SauceLabKey'
```

[6] https://saucelabs.com/