X-Road test automation

**Authors**: Sten Luhtoja, Märt Tibar, Arto Vaas

**Version**: 0.1

**Last updated**: 04.04.2017



Tallinn 2017

Table of Contents

[**Introduction**](#_bdcfi5j73cvb) **4**

[**Used Technologies**](#_bj4y07yx6h4h) **5**

[Programming languages](#_30j0zll) 5

[Automation tools](#_1fob9te) 5

[Build management](#_3znysh7) 5

[Used Ubuntu Packages](#_u4tv41thj346) 5

[Used Libraries, frameworks and APIs:](#_2et92p0) 5

[Used Plugins](#_tyjcwt) 5

[Recording and publishing test results](#_nnmnseeki7h0) 6

[**Framework description**](#_8enmo8sfju3n) **6**

[Framework structure](#_pxptgwgq9s5d) 6

[List of framework packages (directories)](#_mzy4kr55mwao) 7

[Helpers Package](#_oqfi9b9a5wps) 8

[ConfReader class (confreader.py)](#_pxfbvzkf0itr) 8

[INI file (ini\_path)](#_szi6wpw66276) 9

[Text file (config\_file)](#_anf6utbw6mii) 10

[JSON file](#_taakbah1w2mk) 10

[MockRunner class (mockrunner.py)](#_hqf76ewi709) 11

[Starting the service](#_dc6r481ctm02) 12

[Stopping the service](#_f2prhkekbh0t) 12

[SoapTestClient class (soaptestclient.py)](#_y5nmdij905t7) 14

[SSHClient class (ssh\_client.py)](#_re0pdcr3sdwc) 16

[Main Package](#_6vacwf2mhmmc) 17

[Mock Directory](#_k9obpgarkr9y) 20

[Temp Directory](#_sgszq447pgf1) 21

[Tests package](#_d5eyk9l45q5s) 21

[Mock Service](#_z2qnx9ls94ux) 22

[WSDL files](#_iywmsw8motnv) 24

[**Executing tests in Jenkins CI**](#_4i7ojhp) **25**

[Project configuration](#_ue4pbm35jz9j) 25

[Starting the project](#_2bn6wsx) 27

[Test results](#_15jomnahxasl) 27

[**Performance testing Technologies**](#_5kiejm6ibwza) **28**

[Programming languages](#_me8g9j4qv14p) 28

[Automation tools](#_y957atm1hufr) 28

[Build management](#_3619cstye3gx) 28

[Used Packages](#_dewlp1r1it1u) 28

[Used Plugins](#_w5j8utu375bn) 28

[Mock](#_3b4na7hgshul) 28

[Installing performance test](#_kdbz0ty8qklb) 28

[Command line parameters](#_vfi6tfkyd8fk) 28

[Installing performance test](#_dfx6bso78opv) 29

[Setting up Jenkins to run performance test](#_hcsekdy48c0j) 29

[Running performance test](#_uxcsneqt6xj1) 31

# 

# 

# Introduction

This document holds the information and instructions for the X-ROAD automated tests. The document introduces the project architecture and describes packages and more important classes and methods.

Firstly, the document introduces technologies which were used in the creation of the project. It also describes the plugins, packages and libraries needed for executing the project was agreed upon.

Secondly, the architecture of the project is described. Specifically, packages, classes and methods are described in detail.

Finally Jenkins project configuration is brought out and how to execute the test. In similar manner it is possible to run other tests too.

# Used Technologies

## Programming languages

Automatic tests are created using Python (version 2.7)

## Automation tools

Firefox browser version 47.0.2 and Selenium WebDriver 2.53.6 were used when tests were created.   
It is also possible to use Firefox browser version 52.0 and Selenium WebDriver version 3.0.2 when executing tests.

## Build management

Used libraries and other needed software are meant to be installed according to the manual or according to the documentation of the library.

### Used Ubuntu Packages

* Nginx
* Jenkins
* Openjdk-7-jdk
* Firefox
* Geckodriver
* Firefox xvfb

### Used Libraries, frameworks and APIs:

* update
* requests
* selenium
* sudo pip install cffi
* cryptography
* paramiko
* nose2

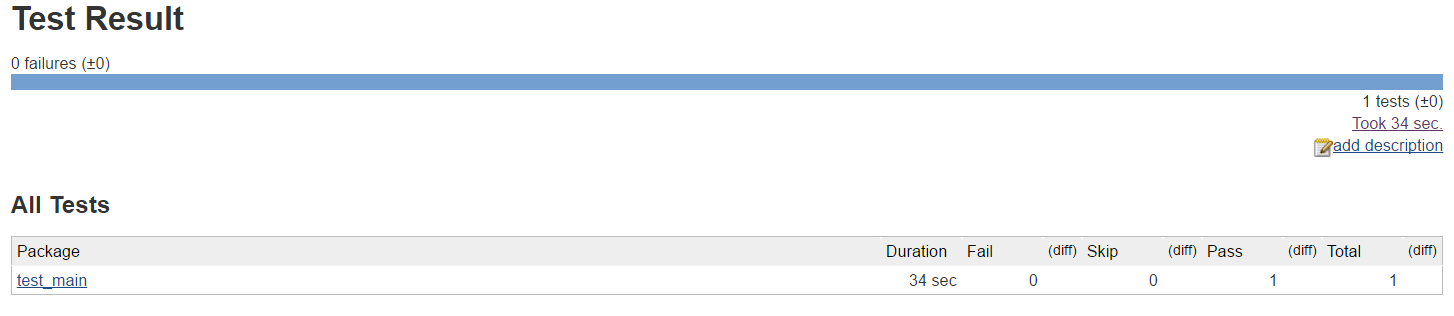
### Used Plugins

* ShiningPanda Plugin
* Junit Plugin

## Recording and publishing test results

Test results are saved in the filesystem until the next test run, it will then be overwritten. Tests history is visible in Jenkins environment, it is possible to see logs and test results.

Python library Nose2 will save the the result in XML format, Jenkins plugin Junit plugin converts it to a html format.

*Figure 1. Example of a test result*

# Framework description

The main class in the test framework is **main.MainController**. This communicates with the browser, handles exceptions and provides configuration parameters to the tests using a **helpers.ConfReader** instance.

## Framework structure

The framework consists of Python packages that contain classes, view models and helper files. Helper files contain functions used by different classes.

The framework also requires a mock service to be running and accessible from security servers. This is not part of the testing framework but as it is a required service, it is also described in this chapter.

The framework also requires an HTTP server that serves WSDL files (service descriptions) to the security server. This is not part of the testing framework but as it is a required service, the WSDL files are also described in this chapter.

#### 

#### 

## List of framework packages (directories)

The list of packages and directories in the framework is shown in Table 1.

*Table 1. List of framework packages.*

|  |  |
| --- | --- |
| **Package** | **Description** |
| helpers | Other classes and helper functions. |
| main | Main framework controller classes. |
| mock | Certificates and queries for mocking services |
| temp | Temporary data created from tests |
| tests | Tests. |
| view\_models | Page or object related functions and constants. |

## 

## 

## Helpers Package

This package holds different helper classes and methods for easier access to resources.

*Table 2. List of files in helpers package*

|  |  |
| --- | --- |
| **Filename** | **Description** |
| confreader.py | ConfReader class, described later. |
| mockrunner.py | MockRunner class, described later. |
| soaptestclient.py | SoapTestClient class, described later. |
| ssh\_client.py | SSHClient class, described later. |
| login.py | Holds login, logout and login-checking methods. |
| ssh\_server\_actions.py | Helper functions for checking XRoad logs and other server specific methods. |
| ssh\_user\_actions.py | Helper functions for manipulating Linux users. |
| webdriver\_init.py | Helper functions for creating browser instances. |
| xroad.py | Helper functions for XRoad datatypes. |

## ConfReader class (confreader.py)

ConfReader is a class for storing and reading parameters from different sources. It supports reading parameters from command-line arguments, INI files, JSON files and text files containing one key=value pair on each line (ignoring empty lines).

**Configuration parsing**

INI files, config files and command-line arguments are all parsed to detect types (Boolean, None, string, integer, float) automatically. Values are initially loaded as strings and type detection is necessary for some methods that need integers or floats as input. The values are automatically checked for text “True”, “False”, “None”, and numeric values. Because keeping a numeric value as string may sometimes be necessary, type detection can be avoided by wrapping the value inside single quotes (‘), example: age=’14’.

**Configuration examples setting the following values:**

|  |
| --- |
| additional\_info=None (NoneType)  person.name=”John Doe” (string)  person.age=24 (integer)  person.tv\_show=”24” (string)  person.married=True (Boolean) |

**Command-line arguments**

Each argument needs to be prefixed with two dashes (--). Long values should be wrapped inside double quotes (“), values without spaces do not need to be wrapped.

Example assumes script filename of *test.py*

|  |
| --- |
| python test.py --additional\_info=None --person.name=”John Doe” --person.age=24 --person.tv\_show=”’24’” --person.married=True |

#### 

## INI file (ini\_path)

INI files ignore empty lines and comment lines starting with a semicolon (;). ConfReader parameter names are constructed from section and key names, concatenated with a dot (.). Section [MAIN] (capital letters) is a special section that will not be used as a prefix.

Sections can be used for grouping parameters, but everything can be written under MAIN.

If command-line arguments are parsed (*init\_command\_line=True* when initializing the class), INI file can be specified with argument ***--ini=/path/to/file.ini***.

|  |  |
| --- | --- |
| ; Grouped parameters  [MAIN]  additional\_info=None  [person]  name=John Doe  age=24  tv\_show=’24’  married=True | ; Everything grouped under MAIN  [MAIN]  additional\_info=None  person.name=John Doe  person.age=24  person.tv\_show=’24’  person.married=True |

## Text file (config\_file)

Each line consists of key=value pair. Empty lines are ignored. Lines without an equals sign (=) are ignored and can be used as comments.

If command-line arguments are parsed (*init\_command\_line=True* when initializing the class), configuration file can be specified with argument ***--config=/path/to/file***.

|  |
| --- |
| This is an example config file  additional\_info=None  person.name=John Doe  person.age=24  person.tv\_show=’24’  person.married=True |

## JSON file

The advantage of using JSON (JavaScript Object Notation) is that values are converted to dictionaries without parsing them.

|  |
| --- |
| {“additional\_info”: “None”,  “person.name”: “John Doe”,  “person.age”: 24,  “person.tv\_show”: “24”,  “person.married”: true} |

If command-line arguments are parsed (*init\_command\_line=True* when initializing the class), JSON file can be specified with argument ***--json=/path/to/file***.

### 

### 

*Table 3. ConfReader class methods*

|  |  |
| --- | --- |
| Method name | Method description |
| clear\_config() | Clear all parameters. |
| set\_config(conf) | Update parameters with dictionary *conf*, overwriting the existing values but not clearing everything. |
| set(key, value) | Set parameter *key* value to *value* |
| get(key) | Get parameter *key* value. |
| get\_string(key, default=’’) | Get parameter *key* value. If *key* does not exist, return default value. |
| get\_int(key, default=0) | Get parameter *key* integer value. If *key* does not exist or is not an integer, return default value. |
| get\_floatkey, default=0.0) | Get parameter *key* float value. If *key* does not exist or is not a float, return default value. |
| get\_bool(key, default=False) | Get parameter *key* Boolean value. If *key* does not exist or is not a Boolean, return default value. |
| read\_command\_line\_arguments() | Reads parameters from command-line arguments. |
| read\_ini(ini\_file) | Reads parameters from an INI file. |
| read\_key\_value\_pairs(file) | Reads parameters from a configuration text file (key=value pairs). |
| read\_json(file) | Reads parameters from a JSON file. |

## 

## MockRunner class (mockrunner.py)

MockRunner is a simple class that controls the mock service script (SoapUI MockRunner) over an SSH connection.

Uses ssh\_helper.SSHClient component. Connects to SSH server, sends a one-liner command and then waits until a specified regex matches output or a timeout occurs. To stop the service, sends a single keycode (Ctrl-C by default).

## Starting the service

When MockRunner tries to start the service, it goes through the following steps.

1. Connect to service server over SSH
2. Send username and password
3. Execute *command*
4. Check standard output (stdout) on the server and look for *ready\_regex* match.
5. If match is found, service has been started. If timeout occurs first, service start failed.
6. Control is given back to the calling program.

If any of these steps fail, the service will not be started, start command returns False and MockRunner.error variable will contain an error message.

## Stopping the service

Stopping the service is done by sending *stop\_keycode* to standard input (stdin) over the SSH link. No other checks are made.

#### 

#### 

*Table 4. MockRunner class methods*

|  |  |
| --- | --- |
| **Method name** | **Method description** |
| start() | Start the service. Returns True if service start was detected, False otherwise. |
| stop() | Stop the service by sending *stop\_keycode* key. |
| restart() | Restart the service by executing stop() and start() in sequence. |
| get\_error() | Returns last start() error or None if no error. |

## 

## 

## SoapTestClient class (soaptestclient.py)

SoapTestClient is the test client class for sending SOAP requests to XRoad security servers. The component is used for testing if the services respond, succeed, return an error, and if the return data matches the expected data.

SoapTestClient uses the Python Requests library to send SOAP queries over HTTP(S) and parses response XML to extract XRoad service parameters.

*Table 5. SoapTestClient class methods*

|  |  |
| --- | --- |
| **Method name** | **Method description** |
| query(url=None, body=None, params=None, timeout=None) | Sends a query to the service.  *url* and *body* are required parameters.  All parameters are optional when calling the method. If not set, they are replaced with default ones that were supplied to the init method. |
| check\_query\_success(url=None, body=None, params=None, query\_timeout=None, faults=None) | Checks if a query succeeds.  Sends a query to the service using query() with the same parameters and check if a the response contains a Fault element and if the fault code matches a supplied list of faults.  All parameters are optional when calling the method. If not set, they are replaced with default ones that were supplied to the init method. |
| check\_query\_loop(url=None, body=None, params=None, query\_timeout=None, faults=None, fail\_timeout=None, retry\_interval=None, verify\_service=None, check\_success=True) | Checks if a query succeeds or fails (depending on *check\_success* value) before *query\_timeout* occurs.  Sends a query to the service using check\_query\_success() every *retry\_interval* seconds until it returns the same value as *check\_success*. If *verify\_service* is supplied, it then checks if it is a subset of response service parameters. Returns *True* if all conditions are *True* before *fail\_timeout* seconds pass, *False* otherwise.  All parameters are optional when calling the method. If not set, they are replaced with default ones that were supplied to the init method. |
| check\_success(url=None, body=None, params=None, query\_timeout=None, faults=None, fail\_timeout=None, retry\_interval=None, verify\_service=None) | Checks if the query succeeds before *query\_timeout* occurs.  Sends a query to the service using check\_query\_loop() using *check\_success=True* every *retry\_interval* seconds until it returns *True*. If a timeout of *query\_timeout* seconds occurs returns *False*.  All parameters are optional when calling the method. If not set, they are replaced with default ones that were supplied to the init method. |
| check\_fail(url=None, body=None, params=None, query\_timeout=None, faults=None, fail\_timeout=None, retry\_interval=None, verify\_service=None) | Checks if a query fails before *query\_timeout* occurs.  Sends a query to the service using check\_query\_loop() using *check\_success=False* every *retry\_interval* seconds until it returns *True*. If a timeout of *query\_timeout* seconds occurs returns *False*.  All parameters are optional when calling the method. If not set, they are replaced with default ones that were supplied to the init method. |
| get\_service() | Internal method to extract service parameters as a dictionary from latest query response XML. |
| verify\_service(service) | Method to check if *service* is a subset of last service parameters returned by *get\_service()* |
| log(str) | Internal default debug logging method. Prints a string to standard output. |

## SSHClient class (ssh\_client.py)

SSHClient is a simple SSH Client class using the Paramiko library. It connects to an SSH server and allows to execute commands, use standard input, output and receive errors (stdin, stdout, stderr). Connection is made during class initialization and remains open until *close()* method is called.

*Table 6. SSHClient class methods*

|  |  |
| --- | --- |
| **Method name** | **Description** |
| get\_client() | Returns the Paramiko internal SSHClient instance. |
| exit\_status() | Returns the last command’s exit status code. |
| write(str, flush=False) | Writes to remote standard input (stdin). If *flush=True*, flushes the buffer after writing. |
| write\_flush(str) | Writes to remote standard input (stdin) and flushes the buffer. |
| writeline(line) | Writes a line to remote server, automatically adds a linefeed character. |
| readline() | Reads a line from remote standard output (stdout) and returns it without linefeed characters. |
| exec\_command(command, sudo=False, timeout=None) | Executes a command on the remote server. If *sudo=True*, tries to execute the command with *sudo* prefix (root rights, user must have *sudo* access). If *timeout* in seconds is supplied, sets the Paramiko internal channel timeout. |
| open(host, username, password) | Connects to host over SSH using specified credentials. |
| close() | Closes the connection. |

## Main Package

Main package contains two classes: MainController and AssertHelper.

**AssertHelper** is a base class for MainController adding assertion methods that can be used during tests.

**MainController** class is the test framework main controller. It handles and interacts with the web browser, provides helper functions for finding elements, logging in and out of XRoad, basic logging and main configuration parameters.

*Table 7. MainController class methods*

|  |  |
| --- | --- |
| Method name | Method description |
| setUp() | Setup method for starting a test. Starts a browser (WebDriver) and Mock Service if they are configured to start automatically. |
| tearDown() | Test teardown method, used for closing the test environment after successful or failed tests. Stops mock service if it was started, and closes the browser window. If an exception has been raised during the test, and screenshots and traceback saving has been enabled, takes a screenshot and saves a traceback to a file. |
| remove\_files(file\_list, remove\_directories=True) | Removes files and, if *remove\_directories=True*, directories given in *file\_list* as a list of strings. Single filename may also be supplied as a string for *file\_list* parameter. Returns *True* if everything was successfully deleted, *False* otherwise. |
| empty\_directory(path) | Empties a directory without deleting the directory itself. |
| get\_path(path=‘’) | Gets the current base path (one level up from maincontroller.py location). If *path* parameter is supplied, then: if it is an absolute path, returns *path*, if a relative path, concatenates it with the base path. |
| get\_temp\_path(path=’’) | Returns the configured temporary path. Otherwise works like *get\_path()* |
| get\_download\_path(path=’’) | Returns the configured download path. Otherwise works like *get\_path()* |
| get\_cert\_path(path=’’) | Returns the configured mock certificate path. Otherwise works like *get\_path()* |
| get\_query\_path(path=’’) | Returns the configured test query path. Otherwise works like *get\_path()* |
| get\_xml\_query(filename) | Returns XML query data from a specified file using *get\_query\_path()* to compute the full path. |
| reset\_webdriver(url, username=None, password=None, close\_previous=None, init\_new\_webdriver=True) | Resets the browser (WebDriver) to a specified URL. If the browser window is not open or *init\_new\_webdriver=True*, starts a new browser instance. If *close\_previous=True* and a browser window was open, closes it. Is *username* is not *None*, tries to log in to XRoad GUI with *username* and *password*. |
| reload\_webdriver(url, username=None, password=None) | Sets the browser to a new URL. |
| reset\_page() | Reloads the current browser page. |
| start\_mock\_service() | If starting the Mock Service is not disabled in the configuration, tries to start it. Creates a new *MockRunner* instance if it does not exist. |
| save\_screenshot(filename) | Takes a screenshot of the current browser window and saves it to configured temporary directory. |
| save\_text\_data(filename, data) | Saves text data to a file in the configured temporary directory. |
| logout() | Logs out from XRoad GUI. |
| login(username, password) | Tries to log in to XRoad GUI with the specified credentials. |
| log(message) | Prints the current timestamp and message to standard output. |
| **Web element methods** |  |
| by\_id(element) | Gets an element from the current page in the browser using the *id* attribute. |
| by\_xpath(element, multiple=False) | Gets an element from the current page in the browser using an XPath locator string. If *multiple=True*, returns multiple elements. |
| by\_css(element, multiple=False) | Gets an element from the current page in the browser using a CSS selector string. If *multiple=True*, returns multiple elements. |
| wait(condition, timeout=120) | Waits until the browser reports the specified condition to be true, or a timeout of *timeout* seconds occurs. |
| wait\_until\_visible(element, type=None, timeout=10, multiple=False) | Waits until specified element (or specified elements, depending on *multiple* value) is visible or a timeout of *timeout* seconds occurs. If *type* is specified, *element* can be a locator string. |
| js(script, \*args) | Executes a JavaScript with optional arguments in the browser and returns the result. |
| async\_js(script, \*args) | Executes a JavaScript asynchronously (non-blocking) in the browser. |
| wait\_jquery(timeout=120) | Waits until jQuery object is not active or a timeout of *timeout* seconds occurs . Used when checking if jQuery-based AJAX queries have finished. |
| get\_classes(element) | Returns a list of CSS classes associated with the specified *element*. |
| input(element, text, click=True, clear=True) | Types *text* into an HTML input/textarea element. If *click=True*, first clicks on the element. If *clear=True*, clears the field before typing. |
| **Methods inherited from AssertHelper** |  |
| is\_true(con1, test\_name=None, msg=’Failed’, log\_message=None) | If con1 is not True, raises an *AssertionException* with message *msg*. Logs the assertion with *test\_name* and *log\_message* (if specified). |
| is\_false(con1, test\_name=None, msg=’Failed’, log\_message=None) | If con1 is not False, raises an *AssertionException* with message *msg*. Logs the assertion with *test\_name* and *log\_message* (if specified). |
| is\_equal(con1, con2, test\_name=None, msg=’Failed’, log\_message=None) | If con1 and con2 are not equal, raises an *AssertionException* with message *msg*. Logs the assertion with *test\_name* and *log\_message* (if specified). |
| not\_equal(con1, con2, test\_name=None, msg=’Failed’, log\_message=None) | If con1 and con2 are equal, raises an *AssertionException* with message *msg*. Logs the assertion with *test\_name* and *log\_message* (if specified). |
| is\_none(con1, test\_name=None, msg=’Failed’, log\_message=None) | If con1 is not *None*, raises an *AssertionException* with message *msg*. Logs the assertion with *test\_name* and *log\_message* (if specified). |
| is\_not\_none(con1, test\_name=None, msg=’Failed’, log\_message=None) | If con1 is *None*, raises an *AssertionException* with message *msg*. Logs the assertion with *test\_name* and *log\_message* (if specified). |

## Mock Directory

Mock directory contains the certificates and XML query bodies that are used by SoapTestClient mock client. It contains two subdirectories.

The “cert” subdirectory contains the certificate (mock.crt) and private key (mock.key) of the mock client, and the certificate of the Mock Service (mockservice.crt, used for verifying the server’s certificate).

The “queries” subdirectory contains queries that are sent by the mock client to the server when testing access lists and services themselves. These are basically static\* SOAP XML files that are sent to the server by the client. The list of queries is shown in the following table.

\* A dynamic variable of “{uuid}” is allowed in the files. This is replaced with a random UUID to make the queries unique.

*Table 8. Mock directory files*

|  |  |
| --- | --- |
| Filename | Description |
| testservice\_bodyMassIndex\_CLIENT1\_sub.xml | Test query from client *KS1 : COM : CLIENT1 : sub* to service *KS1 : COM : CLIENT1 : testservice : bodyMassIndex.v1*. |
| testservice\_bodyMassIndex\_TS2OWNER\_sub.xml | Test query from client *KS1 : GOV : TS2OWNER : sub* to service *KS1 : COM : CLIENT1 : testservice : bodyMassIndex.v1*. |
| testservice\_*xroadGetRandom*\_CLIENT1\_sub.xml | Test query from client *KS1 : COM : CLIENT1 : sub* to service *KS1 : COM : CLIENT1 : testservice : xroadGetRandom.v1*. |
| testservice\_*xroadGetRandom*\_TS2OWNER\_sub.xml | Test query from client *KS1 : GOV : TS2OWNER : sub* to service *KS1 : COM : CLIENT1 : testservice : xroadGetRandom.v1*. |
| centralservice\_random\_CLIENT1\_sub.xml | Test query from client *KS1 : COM : CLIENT1 : sub* to central service *random* (*KS1 : random*) |

## Temp Directory

This directory holds temporary data and test failures(text files and screenshots) created by tests.

There is also a sub-directory “download” where downloaded files are stored, eg. certificates and signing requests.

## Tests package

This package contains all the tests. Tests have their own packages.The test structure contains mostly three major files: \_\_init\_\_.py, test\_main.py and the test itself(usually in format test\_\*.py)

*Table 9. Test example architecture*

|  |  |
| --- | --- |
| \_\_init\_\_.py | Must be file for python package. Makes the test visible to others |
| test\_main.py | Extends unittest.TestCase class, calls the test with needed data. |
| test\_\*.py | Test itself, takes care of methods and calls needed for the test.. |

*Table 10. Test dependencies and requirements*

|  |  |  |
| --- | --- | --- |
| Test number | Depends on finishing other test(s) | Requires helper scenarios |
| 2.1.3 | None |  |
| 2.1.8 | None |  |
| 2.1.9 | None |  |
| 2.2.1 | None | 2.1.3 |
| 2.2.2 | 2.2.1 | 2.1.8 |
| 2.2.5 | 2.2.2 |  |
| 2.2.6 | 2.2.2 |  |
| 2.2.7 | 2.2.2 |  |
| 2.2.8 | 2.2.2 | 2.1.8 |
| 2.2.9 | 2.2.2 | 2.1.8 |
| 2.9.1 | None |  |
| 2.10.1 | None |  |
| 2.11.1 | None |  |
| 2.11.2 | None | 2.1.3, 2.1.8 |

## Mock Service

The mock service serves SOAP requests from users proxied by the security servers. It is used for testing service parameters and access lists.

The mock service has been made in SoapUI and requires SoapUI to run. The mock service is not directly a part of the framework as it can be configured to be running all the time in a different server or the framework may be configured to use other (or non-SoapUI) services that may already be running.

The mock service location is set in the WSDL files (under *<wsdl:service>* subelement *<soap:address>* attribute *location*) and framework configuration parameters *services.test\_service\_url*, *services.test\_service\_2\_url*, *services.test\_service\_url\_ssl* (HTTPS), *services.test\_service\_2\_url\_ssl* (HTTPS). The service must listen on two ports, one using regular HTTP protocol and the other using HTTPS.

|  |
| --- |
| Examples of the part of a WSDL file that points to two services that should already be running on localhost port 8086 under different URLs.  <wsdl:service name="xroadGetRandom">  <wsdl:port name="xroadGetRandomPortSoap11"  binding="tns:xroadGetRandomPortSoap11">  <soap:address location="http://localhost:8086/xroadGetRandom" />  </wsdl:port>  </wsdl:service>  <wsdl:service name="bodyMassIndex">  <wsdl:port name="bodyMassIndexPortSoap11"  binding="tns:bodyMassIndexPortSoap11">  <soap:address location="http://localhost:8086/bodyMassIndex" />  </wsdl:port>  </wsdl:service> |

The Mock Service itself is defined in a single XML file (SoapUI project) but as it has to support certificates, it also requires a SoapUI settings file (for supporting SSL), a keystore and a certificate file. The list of files required to run the service is in the following table.

*Table 11. Mock service directory files*

|  |  |
| --- | --- |
| Filename | Description |
| soapui-settings.xml | SoapUI settings file (XML). Defines SSL port for services to be 18086, keystore file as “mockservice.keystore” and keystore password as “password”. These values can be edited in SoapUI or directly in XML under elements *<con:setting>* with attributes “id” starting with “SSLSettings”. |
| testservice-soapui-project.xml | Mock service SoapUI project (XML). Defines two services: *xroadGetRandom* and *bodyMassIndex*, both are run from port 8086 (if SoapUI settings define an SSL port, they are run from the SSL port at the same time as well). Can be edited from SoapUI. |
| mockservice.keystore | Keystore for the mock service. Default keystore password used is “password”. |

## WSDL files

The WSDL files are used for providing information about services to the security servers. They describe the service (name and version) and the URL from which the security server can access it. Multiple services can be described in one WSDL.

The test automation scenarios require different WSDL files to test different outcomes - a working WSDL, a WSDL that removes a previously defined service, a WSDL that gives a parser error and a WSDL that gives a parser warning.

The WSDL files have to be served from an HTTP server that is accessible from security servers and accessible over SSH from the test framework (to edit the files remotely). WSDL files base URL is defined in the main configuration with key *wsdl.remote\_path* and other WSDL parameters are also under *wsdl* section.

List of WSDL files defined in the default configuration is in the following table.

*Table 12. WSDL files in default configuration*

|  |  |
| --- | --- |
| Filename | Description |
| testservice.wsdl | The WSDL that is added to the security servers. As the tests require the same file to be overwritten with others (working, erroneous, warning, removed service), it contains different data at different times. |
| testservice\_original.wsdl | This is the base WSDL file that is working correctly and defines two services: *bodyMassIndex.v1* and *xroadGetRandom.v1* |
| testservice\_xroadGetRandom\_only.wsdl | Derived from “testservice\_original.wsdl” but defines only one service: *xroadGetRandom.v1* |
| error.wsdl | A file that results in a WSDL parser error. |
| warning.wsdl | A file that results in a WSDL parser warning. |
| gatling.wsdl | Used for performance testing. |

# Executing tests in Jenkins CI

## Project configuration

1. Choose a freestyle project type
2. Choose project name
3. Add Build step “Custom Python Builder”
   1. Home = Python 2.7 location (/usr/bin/python2.7)
   2. Nature = Shell
   3. Command:

repo\_root\_dir=/opt/riajenk/x-road-tests/

test\_dir=xroad\_everything

test\_name=test\_main

project\_name=testing\_all\_automated\_tests

cd $repo\_root\_dir/tests/$test\_dir

export DISPLAY=:10

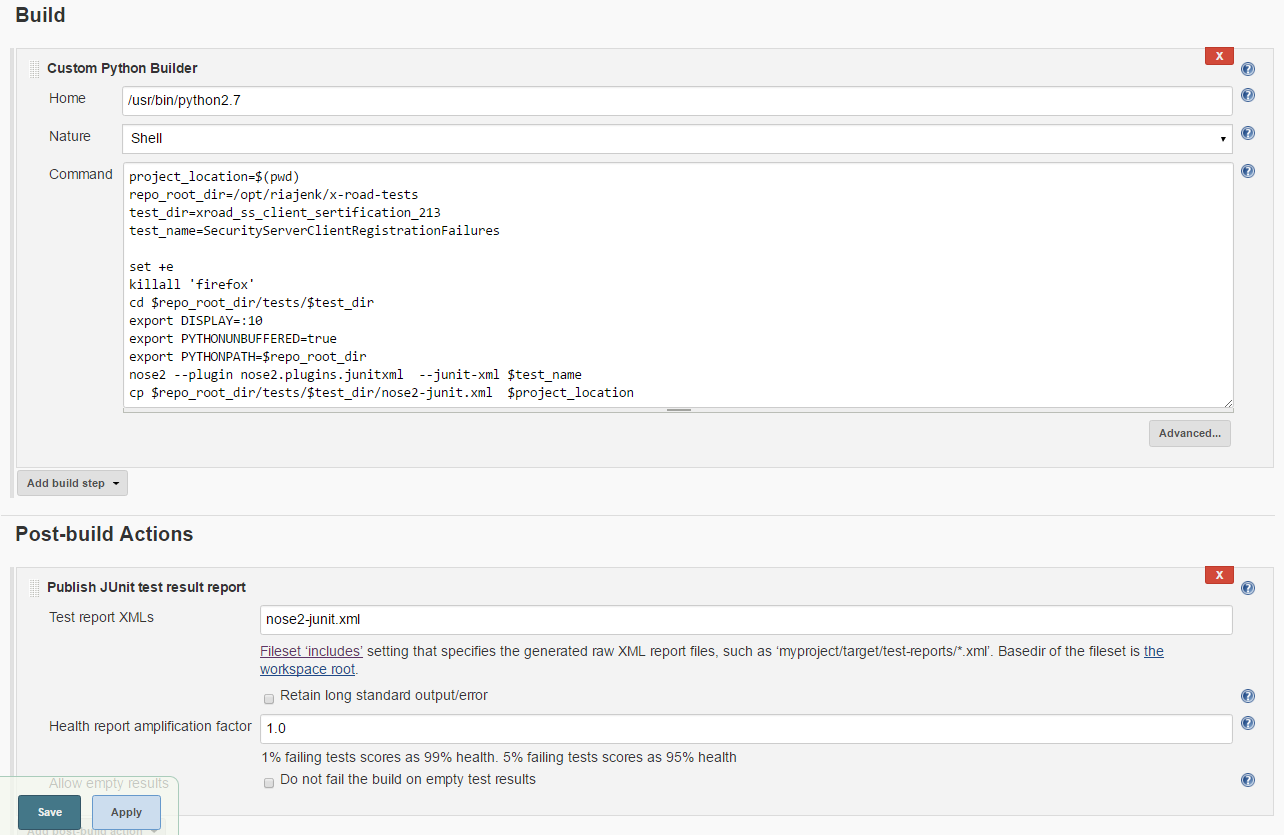
export PYTHONUNBUFFERED=true

export PYTHONPATH=$repo\_root\_dir

nose2 --plugin nose2.plugins.junitxml --junit-xml $test\_name

cp $repo\_root\_dir/tests/$test\_name/nose2-junit.xml /var/lib/jenkins/workspace/$project\_name/

1. Add post build action
   1. Publish Junit test result report
   2. Add the file name which was previously copied.



*Figure 2. Example Jenkins project configuration*

## Starting the project

Open project or from the projects list open project dropdown and click “Build now”. It is also possible to add build triggers.

## Test results

When the test is done, you can see test result in Jenkins. Choose the test which result is wanted. Click on “Latest Test Result”.

The Junit report contains three sections: Result bar, All Failed Tests, and All Tests.

Result bar, show the percentage of failed, skipped and passed test on a line:

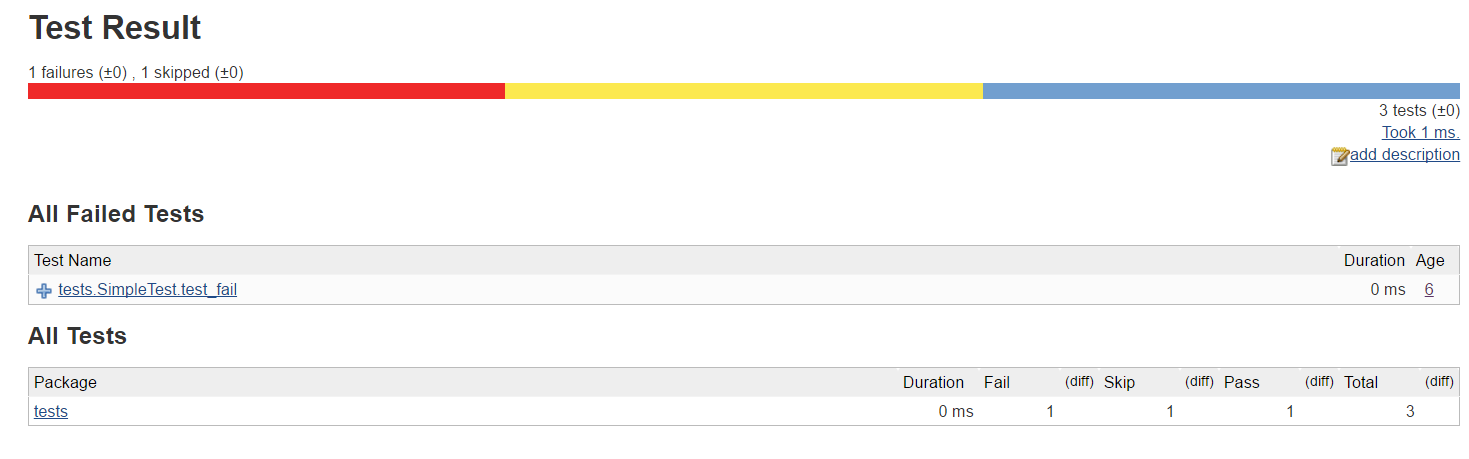
Red - failed tests

Yelllow - skipped tests

Blue - passed tests

All Failed Tests table: shows the list of all failed tests.

All Tests table: shows the list of all tests, if test contains multiple tests, one can see them when clicking on the test link..

*Figure 3. Example of test results*

# Performance testing Technologies

## Programming languages

Performance tests are created using scala programming language.

## Automation tools

Performance test is developed to work with Gatling 2.2.3 version.

## Build management

Used libraries and other needed software are meant to be installed according to the manual or according to the documentation of the library.

### Used Packages

* Jenkins
* Gatling

### Used Plugins

* Junit Plugin

## Mock

Mock sends out the messages with the size what gatling script says. Sends them against the xtee9.ci.kit where is the WSDL registred.

Mock service will be started at the jnekins job. Mock service runs cd /opt/riajenk/xrd-soapui-mock

../SoapUI-5.3.0/bin/mockservicerunner.sh

## Installing performance test

Performance test file “xroad.scala” should be placed in the gatling/user-files/simulations folder.

## Command line parameters

Table will show command line parameters how you can modify the performance test. Variable should be used on the command line as -Dvariable=value.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Default value** | **Value information** |
| xRoadURL | http://127.0.0.1:8080/ | X-road security server |
| memberCode | 11045744 | X-road client and provider code |
| rpsTargets | 1,5,10,15,20,25,30,35,40,45,50 | Target request per second (req/s) |
| warmUpHoldPeriod | 30 | Warm-up lenght (sek) |
| mainHoldPeriod | 600 | Period of holding threads (sek) |
| userBumpInterval | 3 | Ramp-up period(sek) |
| weight2kB | 30.0 | The proportion of messages 2kB (%) |
| weight10kB | 60.0 | The proportion of messages 10kB (%) |
| weight100kB | 9.0 | The proportion of messages 100kB (%) |
| weight2MB | 0.9 | The proportion of messages 2MB(%) |
| weight10MB | 0.1 | The proportion of messages 10MB (%) |

Gatling performance test is not sending out messages in different size, it will send “desiredResponseSize” parameter, what indicates to mock service on whitch size should the response be. If users change the percentage of the sizes it is important to control that all size percentages summ will be 100.

## Installing performance test

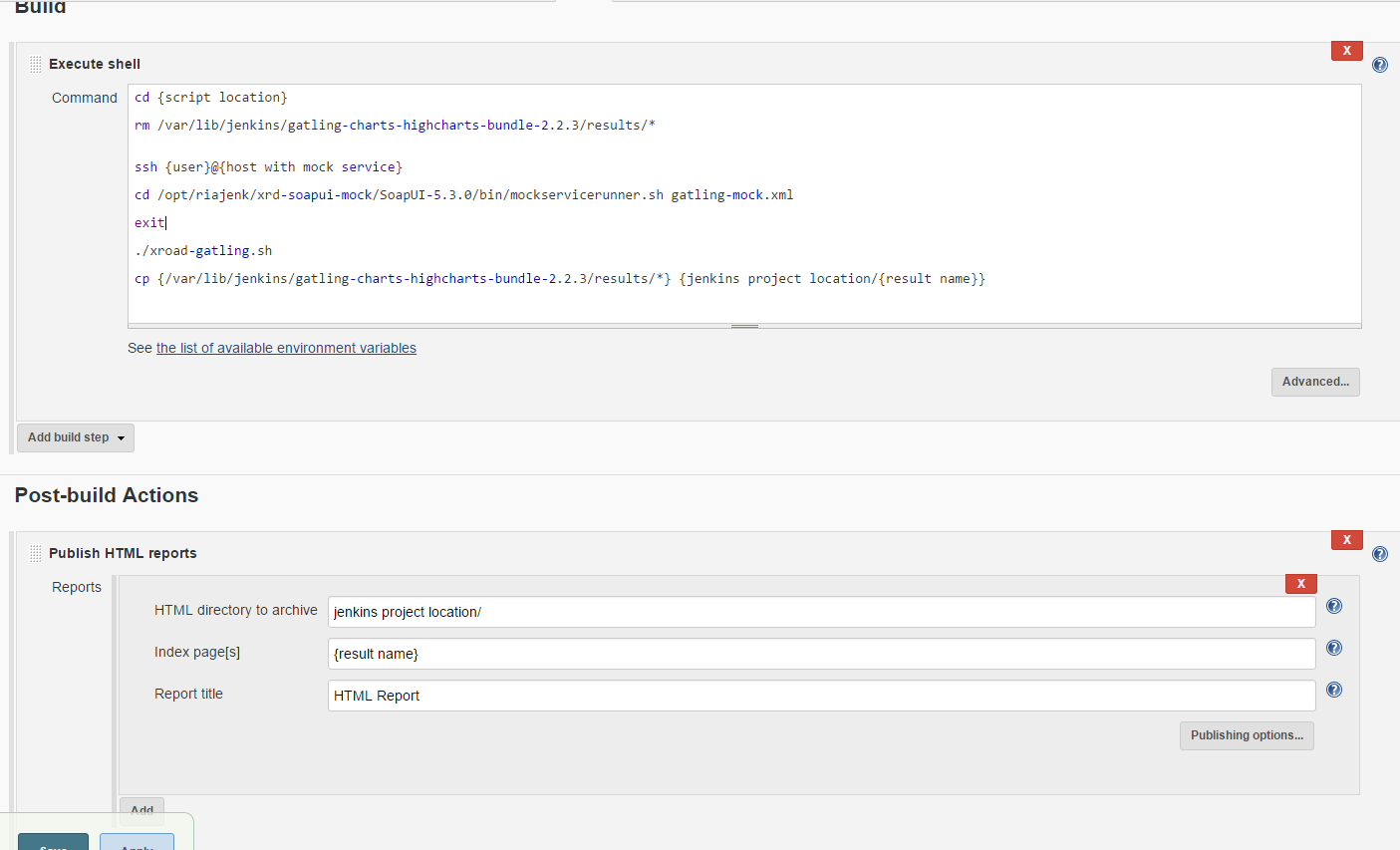
Performance test file “xroad.scala” should be placed in the /var/lib/jenkins/gatling-charts-highcharts-bundle-2.2.3/user-files/simulations folder.

## Setting up Jenkins to run performance test

Create new project for performance test - Freestyle project

Add new build step “shell”

Add new post-build actions “Publish HTML report”



*Figure 4. Jenkins setup example*

Shell script:

ssh [riajenk@xtee2.ci](mailto:riajenk@xtee2.ci).kit

cd /opt/riajenk/xrd-sopatui-moc/SoapUi-5.3.0/bin/mocservicerunner.sh gatling-mock.xml

exit

ssh riajenkins@xtee-jenkins.ci.kit

cd ./var/lib/jenkins/gatling-charts-highcharts-bundle-2.2.3/user-files/simulations

rm /var/lib/jenkins/galting-charts-highcarts-bundle-2.2.3/results/\*

cd /opt/riajenk/xroad-meter

./xroad-gatling-sh

cp {/var/lib/jenkins/galting-charts-highcarts-bundle-2.2.3/results/\*} { /var/lib/jenkins/workspace/xroad-gatling/result}

exit

Publish HTML info:

HTML directory: /var/lib/jenkins/workspace/xroad-gatling/result

Index page: Result name

Rport title: HTML report

## Running performance test

User can change build step “shell” script within the paragraph 4.6 mentioned parameters using them like this : **./xroad-gatling.sh -DrpsTargets=10,20 -DmainHoldPeriod=300** (10 threads run 300 seconds and 20 threads run 300 seconds).

These parameters should be added on the shell script on the ./xroad-gatling.sh line.