HILSTER Testing Framework Demonstrator

Release 2.0.2

HILSTER Testing Solutions GmbH

Oct 09, 2020

CONTENTS

1 Installation **3**

2 Contact **5**

3 Documentation **7**

3.1 Introduction . . . 7
3.2 Installation . . . 7
3.3 Core Functionalities . . . 9
3.4 Data Driven Testing . . . 18
3.5 Interactive Testing . . . 21
3.6 Report Tool . . . 23
3.7 Dashboard . . . 24
3.8 Binary Data and Communication Protocols . . . 28
3.9 Modbus . . . 33
3.10 HART . . . 38
3.11 Continuous Integration . . . 43
3.12 Changelog . . . 44

Welcome to the HILSTER Testing Framework Demonstrator documentation.

HILSTER Testing Framework is a professional Python Testing Framework with a strong focus on reporting, industrial applications and functional safety.

CHAPTER

ONE

INSTALLATION

htf-demo is installed via pip:

```
mkvirtualenv htf-demo
pip install -i https://pypi.hilster.io htf-demo
mkdir htf-demo
cd htf-demo
demo init .
add2virtualenv .
```

htf-demo uses the community license by default to easily get you started.

You might also be interested in the *detailed installation instructions*.

CHAPTER

TWO

CONTACT

If you have any questions or ideas, please don't hesitate and e-mail us via support@hilster.io.

CHAPTER

THREE

DOCUMENTATION

3.1 Introduction

The htf demo is meant to give you a good overview about the capabilities of the HILSTER Testing Framework.

There is a general introduction which should be read by every interested reader, followed by more specific examples. To get you up to speed we suggest to take a look at the examples that are similar to your use case, eg. if you are an embedded engineer you might want to skip the examples regarding web frontends and start with the Modbus examples.

3.2 Installation

First of all you need to install a recent version of Python 3. Please visit the python website download and install Python 3. Python should be added to the `PATH` environment variable.

A great Python IDE is PyCharm. PyCharm offers a community version which can be used for free for non-commercial purposes. Please visit the PyCharm download page, download and install PyCharm.

Alternatively you can install the Jetbrains Toolbox which lets you easily install and update PyCharm and other tools. To get the Toolbox please visit the Jetbrains Toolbox download page.

From commandline install virtualenv and the virtualenv-wrapper.

On Windows, run

```
pip install virtualenv virtualenvwrapper-win
```

and on Linux, run

```
pip install virtualenv virtualenvwrapper
```

To create an virtual python environment for the demo, run

```
mkvirtualenv htf-demo
```

To install and update the htf-demo packet, run

```
pip install -i https://pypi.hilster.io -U htf-demo
```

htf-demo uses the community license by default to easily get you started.

In case you want to create a new directory for the demo files, run the following

```
mkdir htf-demo
cd htf-demo
```

To setup the demo files, run

```
demo init .
add2virtualenv .
```

This will copy all needed python scripts into your project directory. Simply open the folder in PyCharm to start.

The htf demo is now ready to use and you can proceed to the next section.

If you prefer visual assistance, you can take a look at the video below. (The first step, installing virtualenv and virtualenvwrapper, is not shown in the video)

3.2.1 Proxy Server

If you are behind a proxy server, you need to set the "https_proxy" environment variable.

On Windows, run

```
set https_proxy=https://username:password@proxyserver:port
```

and on Linux, run

```
export https_proxy=https://username:password@proxyserver:port
```

before running pip. Please replace username, password, proxy server and port with appropriate values meeting your networking environment.

3.2.2 SSL Verification

In some circumstances it might be necessary to disable SSL verification. To do this simply add the option `--insecure` to `hlm activate` or `hlm renew` or set the environment variable `HLM_INSECURE` to any value.

3.2.3 Demo-license Activation

If you'd like to checkout features not included in the community-edition like Signed-reports, HART and Modbus you have to contact us to let us create a demo-license for you.

To activate the demo-license run

```
hlm purge
hlm activate <your-demo-license-key>
```

3.2.4 Support

If you have any questions please contact us via e-mail to support@hilster.io.

We hope that you will like HILSTER Testing Framework and the demo we provided to you.

We highly appreciate your feedback and we will contact you in about two weeks to talk about HILSTER Testing Framework and your experiences.

3.3 Core Functionalities

This chapter demonstrates the core of the HILSTER Testing Framework.

3.3.1 Foreword

All test reports are best viewed with a modern browser e.g. Google Chrome or Mozilla Firefox. You should expect some restrictions using Microsoft Internet Explorer.

3.3.2 Test Results

The Test Results demo is located in `core/test_results.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf


def test_assert():
    """
    This test shows the easiest way to do assertions.
    """
    assert True


def test_success(assertions):
    """
    This test succeeds.

    Args:
        assertions (htf.fixtures.assertions): the assertions fixture
    """
    assertions.assert_true(True)


def test_failure(assertions):
    """
    This test fails.

    Args:
```

(continues on next page)

(continued from previous page)

```
        assertions (htf.fixtures.assertions): the assertions fixture
    """
    assertions.assert_true(False)


def test_error():
    """
    This test raises an exception

    Args:
        assertions (htf.fixtures.assertions): the assertions fixture
    """
    raise Exception("Error!")


@htf.skip_if(True, "this test is skipped")
def test_skip_if():
    """
    This test is skipped
    """
    pass


if __name__ == '__main__':
    htf.main()
```

To execute the test, run

```
htf -o core/test_results.py
```

from the command line or run it from PyCharm. If you want to change the number of individual tests, you can do so by modifying the variable values at the top of the script.

This will run all tests in the `core/test_results.py` script and create an HTML test report named `testreport.html`, which should be opened by default (because of the `-o` parameter).

Documentation for the HTML test report can be found here.

3.3.3 Docstrings

Tests can be documented using Python's docstrings. htf supports docstrings in reStructured text format. This way the user can add some markup.

More information about reStructured text markup can be found here.

Docstrings are converted into html for the HTML test report, too.

The demo is located in `core/test_docstrings.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf
```

(continues on next page)

(continued from previous page)

```
def test_docstrings():
    """
    Section
    =======

    Docstrings support sections.

    Bullet lists
    ============

    - This is item 1
    - This is item 2

    Tables
    ======

    Grid table:

    +------------+------------+-----------+
    | Header 1   | Header 2   | Header 3  |
    +============+============+===========+
    | body row 1 | column 2   | column 3  |
    +------------+------------+-----------+
    | body row 2 | Cells may span columns.|
    +------------+------------+-----------+
    | body row 3 | Cells may  | - Cells   |
    +------------+ span rows. | - contain |
    | body row 4 |            | - blocks. |
    +------------+------------+-----------+

    Simple table:

    =====  =====  ======
       Inputs     Output
    ------------  ------
      A      B    A or B
    =====  =====  ======
    False  False  False
    True   False  True
    False  True   True
    True   True   True
    =====  =====  ======

    A transition marker is a horizontal line
    of 4 or more repeated punctuation
    characters.

    ------------

    A transition should not begin or end a
    section or document, nor should two
    transitions be immediately adjacent.
"""


if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    htf.main()
```

To execute the docstrings demo from the command line, run

```
htf -o core/test_docstrings.py
```

3.3.4 Steps

Tests can be split up into smaller chunks with a mechanism called steps. The test steps show up in the test reports and help to give tests a better structure and make debugging easier.

The steps demo is located in `core/test_steps.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf


def test_steps(step):
    """
    This test shows how to use steps to structure your tests.

    Args:
         step (htf.fixtures.step): the step fixture
    """
    with step("Initialization"):
        print("init.. ")

    with step("Body"):
        print("body.. ")

        with step("Nested step"):
            print("This is printed in a nested step")

    with step("Cleanup"):
        print("cleanup.. ")


if __name__ == '__main__':
    htf.main()
```

To execute the steps demo from the command line, run

```
htf -o core/test_steps.py
```

3.3.5 Attachments

Tests support file and url attachments. This is a great way to attach test artifact to test reports.

The demo is located in `core/test_attachments.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def test_file_attachment(step, attachments):
    """
    This test plots a graph and attaches it to the test report.
    It also shows the usage of tags.

    Args:
        step (htf.fixtures.step): the step fixture
        attachments (htf.fixtures.attachments): the attachments fixture
    """
    with step("Create plot"):
        def lorenz(x, y, z, s=10, r=28, b=2.667):
            x_dot = s * (y - x)
            y_dot = r * x - y - x * z
            z_dot = x * y - b * z
            return x_dot, y_dot, z_dot

        dt = 0.01
        step_count = 10000

        # Need one more for the initial values
        xs = np.empty((step_count + 1,))
        ys = np.empty((step_count + 1,))
        zs = np.empty((step_count + 1,))

        # Setting initial values
        xs[0], ys[0], zs[0] = (0., 1., 1.05)

        # Stepping through "time".
        for i in range(step_count):
            # Derivatives of the X, Y, Z state
            x_dot, y_dot, z_dot = lorenz(xs[i], ys[i], zs[i])
            xs[i + 1] = xs[i] + (x_dot * dt)
            ys[i + 1] = ys[i] + (y_dot * dt)
            zs[i + 1] = zs[i] + (z_dot * dt)

        fig = plt.figure()
```

(continues on next page)

(continued from previous page)

```
        # ax = fig.gca(projection='3d')
        ax = Axes3D(fig)

        ax.plot(xs, ys, zs, lw=0.5)
        ax.set_xlabel("X Axis")
        ax.set_ylabel("Y Axis")
        ax.set_zlabel("Z Axis")
        ax.set_title("Lorenz Attractor")

        # plt.show()
        plt.savefig("figure.png")

        with step("Attach plot"):
            attachments.attach_file("figure.png", "Lorenz Attractor")


def test_url_attachment(attachments):
    """
    This test attaches a URL to a test results.

    Args:
        attachments (htf.fixtures.attachments): the attachments fixture
    """
    attachments.attach_url("https://hilster.io", "HILSTER")


if __name__ == '__main__':
    htf.main()
```

To execute the attachments demo from the command line, run

```
htf -o core/test_attachments.py
```

3.3.6 Tags

htf supports tags to easily tag tests and test cases.

The tags demo is located in `core/test_tags.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf


@htf.tags("a")
def test_method_a():
    pass


@htf.tags("b")
def test_method_b():
```

(continues on next page)

(continued from previous page)

```
    pass


@htf.tags("c")
def test_method_c():
    pass


if __name__ == '__main__':
    htf.main(tags="a|b")
```

The tags selector is a logical expression.

To execute the tags demo from the command line, run

```
htf -o core/test_tags.py --tags="a|b"
```

for example to select all tests tagged with `a` or `b`. You can also try different tag selectors.

More information on tags can be found in Tagging, Tagging for htf and Tagging for htf.main().

3.3.7 Metadata

Tests can be expanded using metadata. htf supports metadata for test cases and for the test run.

More information about metadata can be found in Metadata for htf and Metadata for htf.main() and in the `Keywords` section in the docs.

The metadata demo is located in `core/test_metadata.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf

Author = htf.Author("$Author: Chuck Norris $")
Source = htf.Source("$Source: /path/to/test_script.py $")
ReviewDate = htf.MetaData("01.01.1970")


def test_with_metadata(metadata):
    """
    This test shows how to use metadata.

    Args:
        metadata (htf.fixtures.metadata): the metadata fixture
    """
    metadata.set('key', 'value')
    metadata['key2'] = 'value2'


if __name__ == '__main__':
    # you can also add test run metadata
```

(continues on next page)

(continued from previous page)

```
    metadata = dict(date="today", time="now", foo="bar")
    htf.main(title="Metadata", metadata=metadata)
```

To execute the test with metadata from the command line, run

```
htf -o core/test_metadata.py -mdate=today,time=now,foo=bar
```

All metadata is included in the test reports.

3.3.8 Fixtures

Fixtures are resources needed by the tests. They can supplement for example communication features, data sources or settings.

Scope: Fixtures do not need to be attached to individual tests. Their lifetime can range from an individual test to an entire test session.

Usage: Fixtures have a name and tests with a parameter of that name will be started with the fixtures. If a fixture depends on another fixture, you can pass the needed fixture as a parameter.

Implementation: Fixtures implemented as Python Generators yielding relevant objects. The generator is decorated with `@htf.fixture(scope)`.

You can find more information in the Fixtures documentation.

The fixture demo is located in `core/test_fixtures.py`.

```
import htf


@htf.fixture('session')
def data_storage():
    data = []
    yield data


@htf.fixture('test')
def logger():
    print("Test started.")
    yield None
    print("Test stopped.")


@htf.test
@htf.data(range(10))
def new_style_test(i, data_storage, logger):
    data_storage.append(i)
    print(data_storage)
    assert True


if __name__ == '__main__':
    htf.main()
```

To execute the fixture demo from the command line, run

```
htf -o core/test_fixtures.py
```

3.3.9 Threads

A common way to provide tests with the means to communicate to a device via a serial port or a network socket is by running the networking code concurrently to the test in a thread. htf offers a wrapper around threads that ties the life time of the thread to the life time of a test.

The threads demo is located in `core/test_threads.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf
import time


def test_background(threads):
    """
    This test shows how to use threads.

    Args:
         threads (htf.fixtures.threads): the threads fixture
    """
    def bg():
        print("Running in background!")

    threads.run_background(bg)
    time.sleep(1.0)


def test_periodic(threads):
    """
    This test shows how to use threads to run a function periodically.

    Args:
         threads (htf.fixtures.threads): the threads fixture
    """
    def p():
        print("Periodically called!")

    threads.run_periodic(p, period=1.0)
    time.sleep(3.0)


if __name__ == '__main__':
    htf.main()
```

To execute the test from command line, run

```
htf -o core/test_threads.py
```

3.3.10 Run All Tests

To execute all tests from `core`, run

```
htf -o core
```

This will also create a test report named `testreport.html` which contains the results of all tests in the core demo.

3.4 Data Driven Testing

Data Driven Testing allows to dynamically generate tests using a data source. The data sources can be Python objects or data files.

Tests are turned into data driven tests by using decorators.

You can find more information in the Data Driven Testing documentation.

3.4.1 Pinging a List of Hosts

You can easily create a test prototype that can ping a host that is supplied as a parameter, and then turn this test into multiple tests trying to ping multiple hosts.

The ping demo is located in `data_driven_testing/test_ping_hosts.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf
import os
import platform


"""
This test case shows how to ping many hosts using data driven testing.
"""


def ping(hostname):
    if platform.platform().startswith("Linux"):
        response = os.system("ping -c 1 " + hostname)
    else:
        response = os.system("ping -n 1 " + hostname)

    htf.assert_equal(response, 0, "the host {} could not be pinged".format(hostname))


@htf.test
def ping_single_host():
    ping("localhost")


@htf.data(["8.8.8.8", "heise.de", "hilster.io"])
```

(continues on next page)

(continued from previous page)

```
def test_ping_hosts(hostname):
    ping(hostname)


@htf.yaml_data("hosts.yml")
def test_ping_hosts_file(hostname):
    ping(hostname)


if __name__ == '__main__':
    htf.main()
```

To run the demo, run

```
cd data_driven_testing
htf -o test_ping_hosts.py
```

3.4.2 External Test Data

It is also possible to put test data into external files that could be generated elsewhere.

The demo includes examples for YAML- and CSV-data.

This test data demo is located in `data_driven_testing/test_data.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf


"""
This test case shows how to run data driven tests with external test data.
"""


@htf.yaml_data("test_data.yml")
def test_data_yaml(hostname, port):
    """
    The dictionaries in the YAML data source must match the parameters.
    A test is dynamically generatged for each item in the test data list.
    """
    print("test", hostname, port)


@htf.csv_data("test_data.csv")
def test_data_csv(hostname, port):
    """
    The headers must match the parameters.
    A test is dynamically generated for each line in the csv data.
    """
    print("test", hostname, port)
```

(continues on next page)

(continued from previous page)

```
@htf.data(range(100))
def test_success(number):
    pass


if __name__ == '__main__':
    htf.main()
```

To run the demo, run

```
cd data_driven_testing
htf -o test_data.py
```

3.4.3 Combinatoric Tests

Let's assume you have a list of hosts and a list of ports of services you'd like to test.

You can dynamically create tests with multiple data files using the combinatoric decorators.

The demo is located in `data_driven_testing/test_combinatoric.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf


@htf.product(htf.YAMLFileIterator("hosts.yml"), htf.YAMLFileIterator("ports.yml"))
def test_hostname_and_service(hostname, port):
    print("test", hostname, port)


if __name__ == '__main__':
    htf.main()
```

To run the demo, run

```
cd data_driven_testing
htf -o test_combinatoric.py
```

3.5 Interactive Testing

In some circumstances tests can not be completely automated.

htf offers interactive testing which allows you to interact with your tests via your web browser.

This feature can be used to script and revision manual tests for example.

You can find more information in the Interactive Testing documentation.

3.5.1 Solve Arithmetic Problems

This demo lets you solve arithmetic problems.

The demo is located in `interaction/test_solve_arithmetic_problems.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf
from operator import add, sub, mul
from random import randint, choice


def test_solve_arithmetic_problems(interaction, assertions):
    """
    This test shows how to interactively solve simple arithmetic problems.

    Args:
         interaction (htf.fixtures.interaction): the interaction fixture
         assertions (htf.fixtures.assertions): the assertions fixture
    """
    try:
        for number_solved in range(1, 6):  # max 5 equations can be solved
            a, b = randint(1, 10), randint(1, 10)  # pick two random numbers
            operator, func = choice([('+', add), ('-', sub), ('*', mul)])  # pick a␣
↪random operator
            answer = interaction.input_dialog(title='Solve this equation',
                                              text="{} {} {} = ?".format(a, operator,␣
↪b),
                                              validator=htf.int_validator)  # only␣
↪accept valid integers as inputs
            assertions.assert_equal(answer, func(a, b))  # check if the answer is␣
↪correct

            if number_solved < 5:
                again = interaction.yes_no_dialog(title='Again',
                                                  text="Would you like to solve␣
↪another one?")
                if again == 'no':  # stop if the user has had enough
                    break
    except AssertionError:
        number_solved -= 1
        raise
```

(continues on next page)

(continued from previous page)

```
    finally:
        interaction.message_dialog(title='Congratulations', text="You solved {}␣
↪equations!".format(number_solved))


if __name__ == '__main__':
    htf.main(interactive=True)
```

To use interactive mode, add `-i` to the htf call.

```
cd interaction
htf -i -o test_solve_arithmetic_problems.py
```

The dialogs will appear in the test report afterwards.

3.5.2 Captcha Solver

This demo lets you solve captchas.

It is located in `interaction/test_captcha_solver.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf
from captcha.image import ImageCaptcha
from random import randint


def test_solve_captchas(interaction, assertions):
    """
    This test shows how to interactively solve captchas.

    Args:
         interaction (htf.fixtures.interaction): the interaction fixture
         assertions (htf.fixtures.assertions): the assertions fixture
    """
    try:
        for number_solved in range(1, 6):  # max 5 captchas can be solved
            image = ImageCaptcha()
            captcha_value = str(randint(1000, 10000))
            image.write(captcha_value, 'captcha.png')

            captcha = """

            .. |captcha| image:: captcha.png
            """

            answer = interaction.input_dialog(title='Captcha',
                                              text="Please solve the following captcha:\
↪n\n|captcha|" + captcha)
            assertions.assert_equal(answer, captcha_value)
```

(continues on next page)

(continued from previous page)

```
            if number_solved < 5:
                again = interaction.yes_no_dialog(title='Again',
                                                  text="Would you like to solve␣
↪another one?")
                if again == 'no':  # stop if the user has had enough
                    break
    except AssertionError:
        number_solved -= 1
        raise
    finally:
        interaction.message_dialog(title='Congratulations', text="You solved {}␣
↪captchas!".format(number_solved))


if __name__ == '__main__':
    htf.main(interactive=True)
```

To use interactive mode add `-i` to the htf call.

```
cd interaction
htf -i -o test_captcha_solver.py
```

The dialogs will appear in the test report afterwards.

3.6 Report Tool

This chapter demonstrates the use of the `report-tool`. Sometimes it is necessary to transform the filetype of a report, or merge reports from multiple test runs together.

You can find more information in the report-tool documentation.

3.6.1 Merge Reports

To merge test reports from multiple sources you only need to supply the original test reports and specify an output type and filename.

You can use the sample reports located in `report_tool/` to test the feature.

To merge both reports to a `html-report`, run

```
cd report_tool
report-tool html_report.html xml_report.xml -H merged_report.html
```

You can find the resulting file in `merged_report.html`.

To merge both reports to a `json-report`, run

```
report-tool html_report.html xml_report.xml -x merged_report.json
```

Once again you can find the resulting file in `merged_report.json`.

3.6.2 Change Report Filetypes

To convert the test report and change the filetype, specify the source file and the output format.

To convert a `html-report` to a `xml-report`, run

```
report-tool html_report.html -x converted_report.xml
```

The resulting file can be found in `converted_report.xml`.

To convert a single file to all possible report types in one command, run

```
report-tool html_report.html -x converted_report.xml -H converted_report.html -j␣
↪converted_report.json -Y converted_report.yml
```

3.7 Dashboard

The HILSTER Dashboard is an interactive analytics tool for test results during development.

Fully integrated into the HILSTER TestBench, test reports can be sent automatically to the Dashboard upon conclusion of a test run.

Additionally, JUnit-XML compatible reports generated by other testing suites can also be uploaded to allow for a comprehensive overview of the project's current test status.

It serves users both in managing and developing roles, showing the current project status as well as changes over time.

Project requirements can be supplied to the Dashboard enabling users to see whether requirements are already covered by tests.

You can find more information in the Dashboard documentation.

3.7.1 Starting the Dashboard

Dashboard requires the use of Docker and Docker Compose to run. It can be installed via your distribution's repositories. As it is shipped as a Linux-image, you will need to run Linux-images on Docker for Windows or use a Linux-machine to run the Dashboard Demo.

To install Docker and Docker Compose please refer to the Docker documentation and Docker Compose documentation.

You also need Git.

First checkout the Docker Compose files from HILSTER's GitHub.

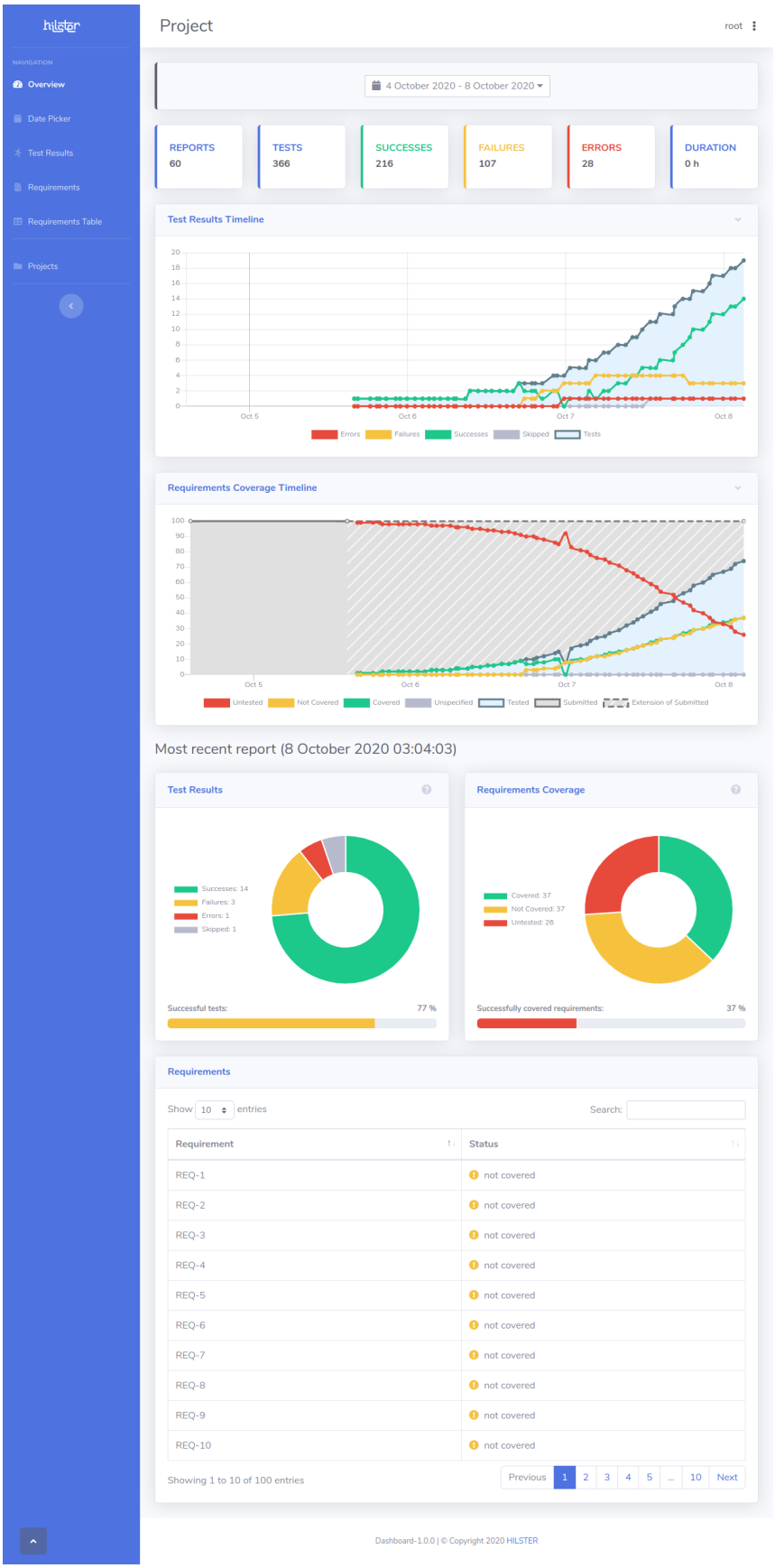
```
git clone https://github.com/hilstertestingsolutions/dashboard-deployment.git
cd dashboard-deployment
```

Edit `docker-compose.yml` and fill in your activation-key.

Then run the dashboard

```
docker-compose up
```

You will now be able to navigate to `http://localhost` in your browser to visit an empty dashboard. The default username is `root`, and the default password is also `root`.


hilster
NAVIGATION
Overview
Date Picker
Test Results
Requirements
Requirements Table
Projects
Project
root
4 October 2020 - 8 October 2020
REPORTS
60
TESTS
366
SUCCESSES
216
FAILURES
107
ERRORS
28
DURATION
0 h
Test Results Timeline
Errors
Failures
Successes
Skipped
Tests
Requirements Coverage Timeline
Untested
Not Covered
Covered
Unspecified
Tested
Submitted
Extension of Submitted
Most recent report (8 October 2020 03:04:03)
Test Results
Successes: 14
Failures: 3
Errors: 1
Skipped: 1
Successful tests:
77 %
Requirements Coverage
Covered: 37
Not Covered: 37
Untested: 26
Successfully covered requirements:
37 %
Requirements
Show 10 entries
Search:
Requirement
Status
REQ-1
not covered
REQ-2
not covered
REQ-3
not covered
REQ-4
not covered
REQ-5
not covered
REQ-6
not covered
REQ-7
not covered
REQ-8
not covered
REQ-9
not covered
REQ-10
not covered
Showing 1 to 10 of 100 entries
Previous
1
2
3
4
5
...
10
Next
Dashboard-1.0.0 | © Copyright 2020 HILSTER

3.7.2 Importing a Project

First you have to login as `root`

```
dashboard login
```

Enter `root` as username and password.

Change directory to `dashboard`

```
cd dashboard
```

Create an empty project

```
dashboard create-project imported_project "Imported Project"
```

Then upload the exported project data by running

```
dashboard import imported_project project_export.zip
```

Now visit `http://localhost/project/imported_project` with your browser. You might need to login with the same credentials used via command line.

3.7.3 Uploading Reports with HILSTER Testing Framework

Create another empty project

```
dashboard create-project example "Example Project"
```

If you reload the browser, you should now see the empty project `Example Project` (http://localhost/project/example). You might need to login with the same credentials used via command line.

Run `test_example.py` multiple times. Play around with the code.

```
htf test_example.py -r http://localhost/project/example
```

```
import htf

successes = 1
failures = 1
errors = 1
skipped = 1

all_requirements = [f"REQ-{i}" for i in range(1, 10 + 1)]
current_requirements = all_requirements[:3]
failing_requirements = current_requirements[:len(current_requirements)//3]
errored_requirements = current_requirements[:-len(current_requirements)//2]


@htf.data(range(successes))
@htf.requirements(*current_requirements)
def test_success(i):
    pass


@htf.data(range(failures))
```

(continues on next page)

(continued from previous page)

```
@htf.requirements(*failing_requirements)
def test_failure(i, assertions):
    assertions.assert_false(True, "This test fails!")


@htf.data(range(errors))
@htf.requirements(*errored_requirements)
def test_error(i):
    raise Exception("This test has an error!")


@htf.data(range(skipped))
def test_skip(i):
    raise htf.SkipTest("This test is skipped!")


if __name__ == '__main__':
    htf.main(report_server="http://localhost/project/example")
```

Upload requirements to the project

```
dashboard upload-requirements example requirements_10.csv
```

Check the project page in the browser. Run the tests again multiple times if you wish.

You can also upload other requirements to see how the project changes.

3.7.4 Requirements Coverage using multiple tools

Assume you have a project which has `20` requirements. The first half is tested with a unit testing framework that generates a JUnit-XML report (`unittests.xml`).

The other half is tested by integration tests using HILSTER Testing Framework.

You'd like to measure the requirements coverage over the whole project.

This is possible with HILSTER Testing Framework and Dashboard.

Create another project and upload the desired requirements.

```
dashboard create-project coverage_project "Coverage Measurement"
dashboard upload-requirements coverage_project requirements_20.csv
```

Create the integration test report

```
htf -H integrationtests.html integrationtests.py
```

Use the `report-tool` to combine both reports and to send them to the Dashboard.

```
report-tool unittests.xml integrationtests.html -r http://localhost/project/coverage_
↪project
```

Now navigate to `http://localhost/project/coverage_project` to check the requirements coverage.

You can vary the scripts to see how the requirements coverage changes.

3.8 Binary Data and Communication Protocols

Oser helps you to work with binary data structures. You can easily build and read binary data files, communication protocols, eeprom contents, etc.

More information can be found in the oser documentation.

Data structures and their dependencies are easily described. You do not need to count bits and bytes anymore.

3.8.1 Communication Protocol

Assume you have a binary communication protocol for a safety relevant system.

Each message consists of a header and one to many data objects. Each data object consists of a type and a type dependent payload. Every header and every data object's consistency is assured with a CRC.

All data is big endian.

The header is specified in the following table.

Byte	Name	Type	Description
0	service_id	UInt8	The service id (Enum)
1..4	timestamp	UInt32	timestamp as unix epoch
5	number_of_data_objects	UInt8	the number of data objects
6..9	crc	UInt32	32 bit checksum with polynomial 0x04C11DB7
10..N	data_objects	Array	The array of data objects

The enumeration for the `service_id` is found in the following table.

Name	Value
Unicast	0x00
Broadcast	0xFF

The data object structure is specified in the following table.

Byte	Name	Type	Description
0	data_object_id	UInt8	The data object id (Enum)
1..N	payload	Object	The specific payload
N+1..N+4	crc	UInt32	32 bit checksum with polynomial 0x04C11DB7

The enumeration for the `data_object_id` can be found in the following table.

Name	Value
U32	0x00
Float[]	0x01
Status	0xFF

The U32 payload is described in the following table.

Byte	Name	Type	Description
1..4	value	UInt32	The value

The Float[] payload is described in the following table.

Byte	Name	Type	Description
1	number_of_values	UInt8	The number of float values
2..N	values	Float list	List of float values

The Status payload is described in the following table.

Byte	Name	Type	Description
0..3	uptime	UInt32	Uptime in seconds
4	status	UInt8	Status bits

The implementation of the protocol is located in `binary_data/protocol.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import oser


class U32Payload(oser.ByteStruct):
    """
    +----------+------------------------+--------+----------------------------+
    | Byte     | Name                   | Type   | Description                |
    +==========+========================+========+============================+
    | 1..4     | value                  | UInt32 | The value                  |
    +----------+------------------------+--------+----------------------------+
    """
    def __init__(self):
        super(U32Payload, self).__init__()
        self.value = oser.UBInt32()


class FloatListPayload(oser.ByteStruct):
    """
    +----------+------------------------+--------+----------------------------+
    | Byte     | Name                   | Type   | Description                |
    +==========+========================+========+============================+
    | 1        | number_of_values       | UInt8  | The number of float values |
    +----------+------------------------+--------+----------------------------+
    | 2..N     | values                 | Float  | List of float values       |
    |          |                        | list   |                            |
    +----------+------------------------+--------+----------------------------+
    """
    def __init__(self):
        super(FloatListPayload, self).__init__()
        self.number_of_values = oser.UBInt8()
        self.values = oser.Array(
            length=lambda ctx: ctx.number_of_values.get(),
            prototype=oser.BFloat
        )
```

(continues on next page)

(continued from previous page)

```
class StatusPayload(oser.ByteStruct):
    """
    +----------+-------------------------+--------+-----------------------------+
    | Byte     | Name                    | Type   | Description                 |
    +==========+=========================+========+=============================+
    | 0..3     | uptime                  | UInt32 | Uptime in seconds           |
    +----------+-------------------------+--------+-----------------------------+
    | 4        | status                  | UInt8  | Status bits                 |
    +----------+-------------------------+--------+-----------------------------+
    """
    def __init__(self):
        super(StatusPayload, self).__init__()
        self.uptime = oser.UBInt32()
        self.status = oser.UBInt8()


PAYLOAD = {
    "U32": (0x00, U32Payload),
    "Float[]": (0x01, FloatListPayload),
    "Status": (0xFF, StatusPayload),
}


def DataObjectId():
    """
    +-----------+-------------------------+
    | Name      | Value                   |
    +===========+=========================+
    | U32       | 0x00                    |
    +-----------+-------------------------+
    | Float[]   | 0x01                    |
    +-----------+-------------------------+
    | Status    | 0xFF                    |
    +-----------+-------------------------+
    """
    return oser.Enum(
        prototype=oser.UBInt8,
        values={key: value[0] for key, value in PAYLOAD.items()}
    )


class DataObject(oser.ByteStruct):
    """
    +----------+-------------------------+--------+-----------------------------+
    | Byte     | Name                    | Type   | Description                 |
    +==========+=========================+========+=============================+
    | 0        | data_object_id          | UInt8  | The data object id (Enum)   |
    +----------+-------------------------+--------+-----------------------------+
    | 1..N     | payload                 | Object | The specific payload        |
    +----------+-------------------------+--------+-----------------------------+
    | N+1..N+4 | crc                     | UInt32 | 32 bit checksum with        |
    |          |                         |        | polynomial 0x04C11DB7       |
    +----------+-------------------------+--------+-----------------------------+
    """
    def __init__(self):
        super(DataObject, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        self.data_object_id = DataObjectId()
        self.payload = oser.Switch(
            condition=lambda ctx: self.data_object_id.get(),
            values={key: value[1]() for key, value in PAYLOAD.items()},
            default=oser.Null()
        )
        self.crc = oser.CRCB32(polynomial=0x04C11DB7, strict=True)


def ServiceId():
    """
    +-----------+-------------------------+
    | Name      | Value                   |
    +===========+=========================+
    | Unicast   | 0x00                    |
    +-----------+-------------------------+
    | Broadcast | 0xFF                    |
    +-----------+-------------------------+
    """
    return oser.Enum(
        prototype=oser.UBInt8,
        values={
            "Unicast": 0x00,
            "Broadcast": 0xFF,
        },
        value="Unicast"
    )


class Message(oser.ByteStruct):
    """
    +---------+-----------------------+--------+---------------------------+
    | Byte    | Name                  | Type   | Description               |
    +=========+=======================+========+===========================+
    | 0       | service_id            | UInt8  | The service id (Enum)     |
    +---------+-----------------------+--------+---------------------------+
    | 1..4    | timestamp             | UInt32 | timestamp as unix epoch   |
    +---------+-----------------------+--------+---------------------------+
    | 5       | number_of_data_objects | UInt8  | the number of data objects |
    +---------+-----------------------+--------+---------------------------+
    | 6..9    | crc                   | UInt32 | 32 bit checksum with      |
    |         |                       |        | polynomial 0x04C11DB7     |
    +---------+-----------------------+--------+---------------------------+
    | 10..N   | data_objects          | Array  | The array of data objects |
    +---------+-----------------------+--------+---------------------------+
    """
    def __init__(self):
        super(Message, self).__init__()

        self.service_id = ServiceId()
        self.timestamp = oser.UBInt32()
        self.number_of_data_objects = oser.UBInt8()
        self.crc = oser.CRCB32(polynomial=0x04C11DB7, strict=True)

        self.data_objects = oser.Array(
            length=lambda ctx: ctx.number_of_data_objects.get(),
            prototype=DataObject
```

(continues on next page)

(continued from previous page)

```
        )
```

An example on how to create a specific message is located in `binary_data/create_message.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import time

import oser
from protocol import Message


if __name__ == '__main__':
    # instantiate a message
    message = Message()

    # set service id
    message.service_id.set("Broadcast")

    # set timestamp
    message.timestamp.set(int(time.time()))

    # add 3 data objects
    message.number_of_data_objects.set(3)

    # set data object 0 to U32
    data_object = message.data_objects[0]
    data_object.data_object_id.set("U32")
    data_object.payload.value.set(0x1234)

    # set data object 1 to Float[] with 4 floats
    data_object = message.data_objects[1]
    data_object.data_object_id.set("Float[]")
    data_object.payload.number_of_values.set(4)
    data_object.payload.values[:] = (1.5**i for i in range(4))

    # set data object 2 to Status
    data_object = message.data_objects[2]
    data_object.data_object_id.set("Status")
    data_object.payload.uptime.set(24*60*60)  # 1 day
    data_object.payload.status.set(0b11110000)

    # encode message to binary
    binary_message = message.encode()

    # print binary message
    print('>>> Message as hex <<<')
    print(oser.to_hex(binary_message))

    # print message
    print('\n>>> Message default print <<<')
    print(message)
```

(continues on next page)

(continued from previous page)

```
    # print message with introspection
    print('\n>>> Message introspection <<<')
    print(message.introspect())
```

An example on how to parse a specific message and extract data is located in `binary_data/parse_message.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

from protocol import Message

binary_message = b"\x00\x5e\xc4\xf1\x91\x0a\xaf\x6b\x65\x56\x00\x00\x00\x00\x00\x00"\
                 b"\x00\x00\x00\x00\x00\x00\x00\x01\x04\xc1\x1d\xb7\x00\x00\x00\x00"\
                 b"\x04\x13\x04\x76\xdc\x00\x00\x00\x00\x09\x22\xc9\xf0\x0f\x00\x00"\
                 b"\x00\x00\x10\x4c\x11\xdb\x70\x00\x00\x00\x00\x19\x6e\xd8\x2b\x7f"\
                 b"\x00\x00\x00\x00\x24\x8b\x27\xc0\x3c\x00\x00\x00\x00\x31\xd0\xf3"\
                 b"\x70\x27\x00\x00\x00\x00\x40\x34\x86\x70\x77\x00\x00\x00\x00\x51"\
                 b"\x7c\x56\xb6\xb0"


if __name__ == '__main__':
    # the above binary_message can be decoded as a Message()
    message = Message()
    message.decode(binary_message)

    # what is the message type?
    print("Message type:", message.service_id.get())

    # how many data objects does the message have?
    print("Number of data objects:", message.number_of_data_objects.get())

    # what is the content of the 3rd data object?
    print("3rd value:", message.data_objects[2].payload.value.get())

    # change the crc at the end and try to decode the message again. What happens?
    broken_binary_message = binary_message[:-1] + b"\xb1"
    # message.decode(broken_binary_message)  # please comment out this line and run␣
↪this script again
```

You will also find a example of a crc checksum error in the last example.

3.9 Modbus

This sections covers htf's modbus features.

You can find more information in the Modbus documentation.

Warning: This Modbus demo is not usable with the htf-community version. Please contact us for a demo license.

3.9.1 Simulating Modbus-Slave

The Modbus demo to simulate a slave device and to query it using a Modbus client is located in `modbus/test_modbus.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf
from htf.communication.modbus import ModbusTCPDevice, ModbusTCPClient

"""
This tests shows how to simulate a modbus device and how to use the client to query
↪it via TCP/IP.
"""

ADDRESS = 'localhost'
PORT = 10502  # must be able to run for non-root users


@htf.fixture('session')
def device():
    """
    Modbus device simulator fixture.
    """
    device = ModbusTCPDevice(ADDRESS, port=PORT)
    yield device
    device.close()


@htf.fixture('test')
def client(device):  # Client requires device to be running already
    """
    Modbus client fixture.
    """
    client = ModbusTCPClient(
        ADDRESS,
        port=PORT,
        references_start_at_one=False,
        auto_disconnect=False,
        debuglevel=0)  # set debuglevel to 1 or 2
    yield client
    client.close()


def test_holding_registers(client, device):
    registers = client.read_multiple_holding_registers(0, 8)
    htf.assert_equal(registers, [0] * 8)

    for address in range(10):
        client.write_multiple_holding_registers(address, [0, 1, 0, 1, 1, 0, 1, 0])
        registers = client.read_multiple_holding_registers(address, 8)
        htf.assert_equal(registers, [0, 1, 0, 1, 1, 0, 1, 0])

        device.write_multiple_holding_registers(address, [1, 1, 0, 0, 1, 0, 1, 0])
```

(continues on next page)

(continued from previous page)

```
        registers = client.read_multiple_holding_registers(address, 8)
        htf.assert_equal(registers, [1, 1, 0, 0, 1, 0, 1, 0])


def test_input_registers(client, device):
    device.write_multiple_input_registers(0, list(range(10)))

    registers = client.read_multiple_input_registers(0, 10)
    htf.assert_equal(registers, list(range(10)))

    registers = device.read_multiple_input_registers(0, 10)
    htf.assert_equal(registers, list(range(10)))

    device.write_multiple_input_registers(10, list(range(20, 31)))

    registers = client.read_multiple_input_registers(5, 15)
    htf.assert_equal(registers, list(range(5, 10)) + list(range(20, 30)))

    for i in range(10):
        r = client.read_input_register(i)
        htf.assert_equal(r, i)

        r = device.read_input_register(i)
        htf.assert_equal(r, i)

    for i in range(10, 20):
        r = client.read_input_register(i)
        htf.assert_equal(r, i+10)

        r = device.read_input_register(i)
        htf.assert_equal(r, i + 10)

    device.write_input_register(2323, 1)
    r = client.read_input_register(2323)
    htf.assert_equal(r, 1)


if __name__ == '__main__':
    htf.main()
```

To execute the demo, run

```
htf -o modbus/test_modbus.py
```

3.9.2 Special Modbus Data Structures

Sometimes you'll need to add special data structures that are not specified by Modbus itself. Using `htf` makes it easy to do so.

Special data types can be described using `oser` and can also be read and written directly with the Modbus convenience functions.

The demo for special Modbus data structures is located in `modbus/test_modbus_data_structure.py`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf
import oser
from datetime import datetime
from htf.communication.modbus import ModbusTCPDevice, ModbusTCPClient


ADDRESS = 'localhost'
PORT = 10502  # must be able to run for non-root users


class DateTime(oser.ByteStruct):
    def __init__(self, value=None):
        """
        ``DateTime`` data type.

        Args:
            value (datetime): a datetime instance to be used
        """
        super(DateTime, self).__init__()

        self.year = oser.UBInt16()
        self.month = oser.UBInt8()
        self.day = oser.UBInt8()

        self.hour = oser.UBInt8()
        self.minute = oser.UBInt8()
        self.second = oser.UBInt8()

        if value is not None:
            self.set(value)

    def set(self, value):
        """
        Set values from a ``datetime`` instance.

        Args:
            value (datetime): a ``datetime`` instance to be used
        """
        if not isinstance(value, datetime):
            raise ValueError("value must be of type datetime")

        self.year.set(value.year)
        self.month.set(value.month)
        self.day.set(value.day)

        self.hour.set(value.hour)
        self.minute.set(value.minute)
        self.second.set(value.second)

    def get(self):
        """
```

(continues on next page)

(continued from previous page)

```
        Get values as datetime.

        Returns:
            datetime: an instance of ``datetime`` containing the current values
        """
        return datetime(year=self.year.get(), month=self.month.get(), day=self.day.
↪get(),
                        hour=self.hour.get(), minute=self.minute.get(), second=self.
↪second.get())


@htf.fixture('test')
def device():
    device = ModbusTCPDevice(ADDRESS, port=PORT)
    yield device
    device.close()


@htf.fixture('test')
def client(device):  # require the simulated device running
    client = ModbusTCPClient(
        ADDRESS,
        port=PORT,
        references_start_at_one=False,
        auto_disconnect=False,
        debuglevel=0)  # set debuglevel to 1 or 2
    yield client
    client.close()


"""
This test shows how to read and write own data structures using Modbus.
"""


def test_datetime_data_structure(client, device):
    """
    A DateTime data type can be read from input registers at address 23.
    """

    now = datetime.now()
    device.write_input_register_data(23, DateTime(now))

    current_date_struct = client.read_input_register_data(23, DateTime)

    htf.assert_equal(current_date_struct.year, now.year)
    htf.assert_equal(current_date_struct.month, now.month)
    htf.assert_equal(current_date_struct.day, now.day)

    htf.assert_equal(current_date_struct.hour, now.hour)
    htf.assert_equal(current_date_struct.minute, now.minute)
    htf.assert_equal(current_date_struct.second, now.second)

    print("device date time is:", current_date_struct.get())


if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    htf.main()
```

To execute the demo, run

```
htf -o modbus/test_modbus_data_structure.py
```

3.10 HART

This section deals with device communication using the HART industrial automation protocol. Familiarity with this protocol and general topology is required to understand the examples given. The usefulness of this section to a general audience will be limited.

You can find more information in the HART documentation.

Warning: This HART demo is not usable with the htf-community version. Please contact us for a demo license.

3.10.1 Basic HART Communication

For a simple example of how sending and receiving HART commands works see `hart/test_basic_hart_communication.py`.

Before running you have to edit the file to fit to your needs. Enter a valid value for `COM_PORT`.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf
from htf.communication.hart import HartDeviceCommunication, HartFrame

COM_PORT = 'ENTER_VALID_COM_PORT'  # Replace this with the COM port your HART modem␣
↪is connected to.


@htf.fixture('test')
def hart_device():
    com = HartDeviceCommunication(COM_PORT)
    com.find_device()
    yield com
    com.close()


"""
These tests show different ways to send and receive HART commands. It is only␣
↪intended to be a high-level
introduction to how communication can be achieved. Please check the documentation if␣
↪lower-level access is needed.
"""
```

(continues on next page)

(continued from previous page)

```
def test_query(hart_device):
    response = hart_device.query(HartFrame(1))  # Command 1 - Read Primary Variable
    print('Response:')
    print(response)


def test_query_context():
    with HartDeviceCommunication(COM_PORT) as com:
        response = com.query(HartFrame(1))  # Command 1 - Read Primary Variable
        print('Response:')
        print(response)


def test_request_payload(hart_device):
    request = HartFrame(18)  # Command 18 - Write Tag, Descriptor, Date
    request.payload.tag.set(b'TESTTAG')
    print('Request:')
    print(request)
    response = hart_device.query(request)
    print('Response:')
    print(response)


if __name__ == '__main__':
    htf.main()
```

To execute the demo, run

```
htf -o hart/test_basic_hart_communication.py
```

3.10.2 Device Specific Commands

In order to use device specific or custom HART commands, a specialized HartFrame is needed. See `hart/test_custom_commands.py` for an example.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#

import htf
import oser
from htf.communication.hart import HartFrame, ResponseCode, FieldDeviceStatus
from htf.communication.hart.data_types import HartResponse

# Let's define a custom command to read random bytes from a devices memory
#
# The command number shall be 123 and have the following payloads:
# Request
#         - one unsigned 8-bit Integer specifying the number of bytes to read
# Response
#         - one unsigned 8-bit Integer to echo the number of requested bytes
#         - a byte string of variable length, depending on the number of requested␣
↪bytes
```

(continues on next page)

(continued from previous page)

```
#
#         Possible error response codes: passed parameter too large (to disallow
↪requests of over 127 bytes)
#                                        too few data bytes received (when the number
↪of bytes parameter is missing)


# Defining request and response data bytes using the oser package


class Command123_ReadRandomBytesRequest(oser.ByteStruct):
    def __init__(self):
        super(Command123_ReadRandomBytesRequest, self).__init__()
        self.number_of_bytes = oser.UBInt8()


class Command123_ReadRandomBytesResponse(HartResponse):
    def __init__(self):
        super(Command123_ReadRandomBytesResponse, self).__init__()
        self.response_code = ResponseCode(error_passed_parameter_too_large=3,
                                          error_too_few_data_bytes_received=5)
        self.field_device_status = FieldDeviceStatus()

        self.number_of_bytes = oser.UBInt8()
        self.read_bytes = oser.Data(length=lambda self: self.number_of_bytes.get())


# To use these payloads a custom HartFrame is required to map request and response
↪data bytes to specific commands.
# Universal and common practice commands are supported by the HartFrame base class,
↪so only additional commands
# need to be added here.

class MyDeviceHartFrame(HartFrame):

    def requests_generator(self):
        """
        This generator function maps request data bytes to their respective command
↪numbers
        """
        yield 123, oser.Lazy(Command123_ReadRandomBytesRequest)  # wrapping the class
↪with oser.Lazy() is not required,
                                                                 # but can improve
↪performance when a large number of
    def responses_generator(self):                               # commands are
↪supported
        """
        This generator function maps response data bytes to their respective command
↪numbers
        """
        yield 123, oser.Lazy(Command123_ReadRandomBytesResponse)


# The new command is now ready to be used and can be passed to HART communication
↪functions as the 'decoder_generator'
# argument.
#
```

(continues on next page)

(continued from previous page)

```
# Example:
#     with HartDeviceCommunication(comport='COM1', decoder_
↪generator=MyDeviceHartFrame) as com:
#         response = com.query(MyDeviceHartFrame(123))
#         print(response)


def test_custom_command():
    request = MyDeviceHartFrame(123)
    print('Request:')
    print(request)
    response = MyDeviceHartFrame(123)
    response.delimiter.frame_type.set('ack')
    print('Response:')
    print(response)


if __name__ == '__main__':
    htf.main()
```

To execute the demo, run

```
htf -o hart/test_custom_commands.py
```

3.10.3 HART Slave Device Simulation

When testing HART master devices it can be difficult to replicate certain test conditions, for example when a slave device is misbehaving or faulty or when a large number of slave devices are required. In these cases it is often easier to use device simulators instead of physical devices. See `hart/test_slave_simulator.py` for an example.

```
# -*- coding: utf-8 -*-
#
# Copyright (c) 2020, HILSTER Testing Solutions GmbH, Germany - https://hilster.io
# All rights reserved.
#
from random import randint

import htf
from htf.communication.hart.slave_simulator import HartSlaveSimulator
from unittest.mock import patch
from hart.test_custom_commands import MyDeviceHartFrame


# This HART slave simulator will respond to the custom command 123 from the previous␣
↪example

class SlaveSimulator(HartSlaveSimulator):
    def __init__(self, comport):
        super(SlaveSimulator, self).__init__(comport=comport,
                                             decoder_generator=MyDeviceHartFrame,
                                             device_type=0x1234,
                                             device_id=0x56)
        self.device_memory = b"This is my memory. There's lots of data in it. " * 50

    def get_random_bytes(self, number_of_bytes):
```

(continues on next page)

(continued from previous page)

```
        start = randint(0, len(self.device_memory) - number_of_bytes)
        end = start + number_of_bytes
        return self.device_memory[start:end+1]

    def handle_command123(self, request):
        response = self._create_response(request)
        number_of_bytes_requested = request.payload.number_of_bytes.get()
        if number_of_bytes_requested > 127:  # only 127 bytes are allowed to be read␣
↪at one time
            response.payload.response_code.set('error_passed_parameter_too_large')
        else:
            response.payload.number_of_bytes.set(number_of_bytes_requested)  # echo␣
↪number of bytes
            random_bytes = self.get_random_bytes(number_of_bytes_requested)  # read␣
↪random bytes from memory
            response.payload.read_bytes.set(random_bytes)
        return response


"""
This test will start the slave simulator with a mocked interface to simulated a␣
↪request.
"""

@patch('htf.communication.hart.slave_simulator.HartInterface')
def test_start_simulator_success(mock_interface_class):
    mock_interface = mock_interface_class.return_value

    request = MyDeviceHartFrame(123)
    request.address.address.set(0x1234000056)  # the slave will only respond to␣
↪commands addressed to it
    request.payload.number_of_bytes.set(25)  # read 25 bytes
    mock_interface.read.return_value = request.encode()  # set the request for the␣
↪slave to read
    print('Request:')
    print(request)

    sim = SlaveSimulator(comport='')
    sim.run(1)  # run until one request has been answered
    response = MyDeviceHartFrame()
    response.decode(mock_interface.write.call_args[0][0])  # decode the response␣
↪written to the mocked interface
    print('Response:')
    print(response)

    htf.assert_equal(response.payload.response_code.get(), 'success')  # expect a␣
↪successful response
    htf.assert_equal(response.payload.number_of_bytes.get(), 25)  # number of␣
↪requested bytes should be echoed
    htf.assert_equal(len(response.payload.read_bytes.get()), 25)  # the actual length␣
↪of read bytes should match

    sim.close()


@patch('htf.communication.hart.slave_simulator.HartInterface')
def test_start_simulator_error_response(mock_interface_class):
```

(continues on next page)

(continued from previous page)

```
    mock_interface = mock_interface_class.return_value
    request = MyDeviceHartFrame(123)
    request.address.address.set(0x1234000056)  # the slave will only respond to␣
↪commands addressed to it
    request.payload.number_of_bytes.set(128)  # read 128 bytes, violating the␣
↪threshold
    mock_interface.read.return_value = request.encode()
    print('Request:')
    print(request)

    sim = SlaveSimulator(comport='')
    sim.run(1)  # run until one request has been answered
    response = MyDeviceHartFrame()
    response.decode(mock_interface.write.call_args[0][0])  # decode the response␣
↪written to the mocked interface
    print('Response:')
    print(response)

    # expect an error response for requesting more than 127 bytes
    htf.assert_equal(response.payload.response_code.get(), 'error_passed_parameter_
↪too_large')

    sim.close()


if __name__ == '__main__':
    htf.main()
```

To execute the demo, run

```
htf -o hart/test_slave_simulator.py
```

3.11 Continuous Integration

3.11.1 Jenkins

The Jenkins demo is located in `continuous_integration/jenkins`.

You need Java to run the demo. You also need Cygwin on Windows and the Cygwin binaries must be in PATH environment variable.

The demo folder contains a simple Jenkins job configured to use htf.

First download jenkins.war into `continuous_integration/jenkins` folder.

Then, on Windows, run

```
set JENKINS_HOME=jenkins_home
java -jar jenkins.war
```

and on Linux run

```
export JENKINS_HOME=`pwd`/jenkins_home
java -jar jenkins.war
```

Now you can access the Jenkins web frontend using your browser by opening this link.

Username and password are `jenkins`.

3.12 Changelog

3.12.1 2.0.2

- install htf-community automatically

3.12.2 htf-demo-20200622

- add binary data demo

3.12.3 htf-demo-20200423

- add demo for HART communication

3.12.4 htf-demo-20200421

- add support for Python-3.8

3.12.5 htf-demo-20190710:

- use `htf-1.3`
- add demo for Data Driven Testing
- add demo for Modbus

3.12.6 htf-demo-20190207:

- initial demo for core and continuous integration