hilster

TECHNICAL DOCUMENTATION

HILSTER Testing Framework Demonstrator

Complete documentation.
One clear reference.

Version 4.3.1

HTML · Markdown · PDF

HILSTER

Contents

HILSTER Testing Framework Demonstrator **4**
Installation 4
Contact 4
Documentation 4

Installation **4**
Proxy Server 5
SSL Verification 5
Demo-license Activation 6
Support 6

Core Functionalities **6**
Foreword 6
Test Results 6
Docstrings 8
Steps 10
Attachments 12
Tags 14
Metadata 15
Fixtures 17
Threads 18
Run All Tests 19

Data Driven Testing **20**
Pinging a List of Hosts 20
External Test Data 21
Combinatorial Tests 23

Behavior Driven Development **24**
Setup Webdriver 25
Behavior Driven Development based End-to-End Test 25
Alternative Implementation 35

Interactive Testing **37**
Solve Arithmetic Problems 38
Captcha Solver 39

Report Tool **41**
Merge Reports 41
Change Report Filetypes 42

Dashboard **43**
Starting the Dashboard 45
Importing a Project 46
Uploading Reports with HILSTER Testing Framework 47
Requirements Coverage using multiple tools 49

Binary Data and Communication Protocols **50**
Communication Protocol 50
Modbus **56**
Simulating Modbus-Slave 56
Special Modbus Data Structures 58
HART **60**
Basic HART Communication 60
Device Specific Commands 61
HART Slave Device Simulation 63
HART Simulation/Hardware Tests 65
Continuous Integration **68**
Jenkins 68
Cyber Resilience Act Conformity **68**
Statement of Conformity 69
Security Measures 69
Reporting Vulnerabilities 69
Changelog **70**
htf-demo-4.3.0 70
htf-demo-4.2.4 70
htf-demo-4.2.3 70
htf-demo-4.2.2 70
htf-demo-4.2.1 70
htf-demo-4.2.0 70
htf-demo-4.1.2 70
htf-demo-4.1.1 70
htf-demo-4.1.0 70
htf-demo-4.0.1 70
htf-demo-4.0.0 71
htf-demo-3.0.0 71
htf-demo-2.0.5 71
htf-demo-2.0.4 71
htf-demo-2.0.3 71
htf-demo-20200622 71
htf-demo-20200423 71
htf-demo-20200421 71
htf-demo-20190710: 71
htf-demo-20190207: 71
PDF version **72**

HILSTER Testing Framework Demonstrator

Welcome to the HILSTER Testing Framework Demonstrator documentation.

The htf demo is meant to give you a good overview about the capabilities of the HILSTER Testing Framework, which is a professional Python Testing Framework with a strong focus on reporting, industrial applications and functional safety.

The demo contains specific examples. To get you up to speed we suggest to take a look at those that are similar to your use case, e.g. if you are an embedded engineer you might want to skip the examples regarding web frontends and start with the Modbus examples.

Installation

The `htf-demo` can be installed by following the detailed installation instructions.

Contact

If you have any questions or ideas, please don't hesitate and e-mail us.

Documentation

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 can be used for free **also for professional projects**. 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.

If you use PyCharm, you can create a new project which will automatically create a new virtual environment.

If not, you can use the uv tool which is a great cross-platform tool to initialize and manage a virtual environment. Follow the installation instructions to install uv if you didn't already, then create the new environment.

```
mkdir htf-demo-project
cd htf-demo-project
uv init
```

```
uv venv
source .venv/bin/activate
```

Then, to install and update the htf-demo package, run **inside a virtual environment** the following

```
uv pip install -i https://pypi.hilster.io htf-demo  # When using uv
pip install -i https://pypi.hilster.io htf-demo  # When being already in a venv
```

Note

Got issues with the installation? You might be behind a proxy server, please read Proxy Server and following paragraphs.

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

To create a new directory which will contain the demo files and setup it, run the following

```
mkdir htf-demo
cd htf-demo
demo init .
```

This will copy all demo scripts into your project directory. Simply open the folder in PyCharm (or your IDE of choice) to start, or simply use the command line.

The htf demo is now ready to use and you can proceed to the next section, namely discovering the core functionalities.

Proxy Server

If you are behind a proxy server, you need to set the “https_proxy” environment variable before running pip.

```
set https_proxy=https://username:password@proxyserver:port  # On Windows
export https_proxy=https://username:password@proxyserver:port  # On Linux
```

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

SSL Verification

In some circumstances it might be necessary to disable SSL verification. To do this simply set the environment variable `HLM_INSECURE` to value “1”.

```
set HLM_INSECURE=1  # On Windows
export HLM_INSECURE=1  # On Linux
```

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>
```

In case you got issues with SSL Verification before, add the option `--insecure` to `hlm activate`.

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.

Core Functionalities

This chapter demonstrates the core functionalities of the HILSTER Testing Framework.

Foreword

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

Test Results

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

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import htf
import htf.assertions as assertions


def test_assert() -> None:
```

```
    """
    This test shows the easiest way to do assertions.
    """
    assert True


def test_success() -> None:
    """
    This test succeeds.
    """
    assertions.assert_true(True)


def test_failure() -> None:
    """
    This test fails.
    """
    assertions.assert_true(False)


def test_error() -> None:
    """
    This test raises an exception.
    """
    raise Exception("Error!")


@htf.skip_if(True, "this test is skipped")
def test_skip_if() -> None:
    """
    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.

If this is the first test you're running, you should see that the Community License got activated.

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).

To run the demo with asyncio, run

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

Documentation for the HTML test report can be found here.

Test results and the HTML report

Download recording · Download transcript

```
Test results and the HTML report
================================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_core_testresult.mp4

1. Activate the virtual environment and change to the initialized htf-demo directory.

2. Run the core/test_results.py example with htf and the -o option.

3. The terminal reports successful, failed, erroneous, and skipped test outcomes.
   Failures in this demonstration are intentional examples of result categories.

4. The -o option opens the generated HTML test report. Its navigation groups tests by
   result category and lets the viewer inspect individual entries.
```

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`.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import htf


def test_docstrings() -> None:
    """
    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__":
    htf.main()
```

To execute the docstrings demo from the command line, run

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

To run the demo with asyncio, run

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

Docstrings in test reports

Download recording · Download transcript

```
Docstrings in test reports
==========================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_core_docstrings.mp4

1. Run the test_docstrings.py example from the core directory.

2. Open the generated HTML report and expand the test's documentation.

3. The report displays the test docstring as formatted reStructuredText, including lists
   and tables rather than an unformatted source string.

4. The source listing in the chapter shows how that documentation is written.
```

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`.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import htf


def test_steps(step: htf.fixtures.step) -> None:
    """
    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
```

To run the demo with asyncio, run

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

Hierarchical test steps

Download recording · Download transcript

```
Hierarchical test steps
=======================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_core_steps.mp4

1. Run the test_steps.py example from the core directory.

2. The terminal shows initialization, body, a nested body step, and cleanup, together
   with captured output and results.

3. Open the test in the HTML report. Its step tree displays the same hierarchy,
   including the nested step.

4. The report also retains the captured standard output for the test.
```

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`.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

from typing import Tuple, Any

import htf
import numpy as np
import matplotlib.pyplot as plt  # type: ignore
from mpl_toolkits.mplot3d import Axes3D  # type: ignore


def test_file_attachment(step: htf.fixtures.step, attachments: htf.fixtures.attachments) -> None:
    """
    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(
            xyz: Any, *, s: int = 10, r: int = 28, b: float = 2.667,
        ) -> Any:
            x, y, z = xyz
            x_dot = s * (y - x)
            y_dot = r * x - y - x * z
            z_dot = x * y - b * z
            return np.array([x_dot, y_dot, z_dot])

        dt = 0.01
        num_steps = 10000

        xyzs = np.empty((num_steps + 1, 3))
        xyzs[0] = (0., 1., 1.05)
        for i in range(num_steps):
            xyzs[i + 1] = xyzs[i] + lorenz(xyzs[i]) * dt

        # Plot
        ax = plt.figure().add_subplot(projection='3d')

        ax.plot(*xyzs.T, 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: htf.fixtures.attachments) -> None:
    """
    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
```

To run the demo with asyncio, run

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

File and URL attachments

Download recording · Download transcript

```
File and URL attachments
========================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_core_attachments.mp4

1. Run the test_attachments.py example from the core directory.

2. The tests demonstrate file and URL attachments and complete with successful results.

3. Open the file-attachment test in the HTML report. Its Attachments area includes the
   generated plot and a preview.
```

```
4. The report lets the viewer inspect attached artifacts alongside the test's
   documentation, steps, and output.
```

Tags

htf supports tags to easily tag tests and test cases.

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

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import htf


@htf.tags("a")
def test_method_a() -> None:
    pass


@htf.tags("b")
def test_method_b() -> None:
    pass


@htf.tags("c")
def test_method_c() -> None:
    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"
```

To run the demo with asyncio, run

```
htf -o core_async/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().

Selecting tests by tags

Download recording · Download transcript

```
Selecting tests by tags
=======================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_core_tags.mp4

1. Run the test_tags.py example from the core directory with the tag selector
   --tags="a|b".

2. The selector requests tests carrying tag a or tag b.

3. The terminal lists the selected tests and their results.

4. The generated HTML report contains the selected tests; opening a test displays its
   tags.
```

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`.

```
#
# Copyright (c) 2023, HILSTER - 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: htf.fixtures.metadata) -> None:
    """
```

```
    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
    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
```

To run the demo with asyncio, run

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

All metadata is included in the test reports.

Test and run metadata

Download recording · Download transcript

```
Test and run metadata
=====================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_core_metadata.mp4

1. Run the test_metadata.py example from the core directory.

2. The recording also demonstrates passing metadata on the htf command line.

3. Open the generated test report and inspect the Metadata area of the test.

4. The recorded report shows the metadata values alongside the test's documentation and
   result. The chapter's current source listing explains how metadata is supplied.
```

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`.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

from typing import Generator, Any, List

import htf


@htf.fixture("session")
def data_storage() -> Generator[List[Any], None, None]:
    data: List[Any] = []
    yield data


@htf.fixture("test")
def logger() -> Generator[None, None, None]:
    print("Test started.")
    yield None
    print("Test stopped.")


@htf.test
@htf.data(range(10))
def new_style_test(i: int, data_storage: Any, logger: None) -> None:
    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
```

To run the demo with asyncio, run

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

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`.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import htf
import time


def test_background(threads: htf.fixtures.threads) -> None:
    """
    This test shows how to use threads.

    Args:
         threads (htf.fixtures.threads): the threads fixture
    """

    def bg() -> None:
        print("Running in background!")

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


def test_periodic(threads: htf.fixtures.threads) -> None:
    """
    This test shows how to use threads to run a function periodically.

    Args:
         threads (htf.fixtures.threads): the threads fixture
    """
```

```
    def p() -> None:
        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
```

Background and periodic test activity

Download recording · Download transcript

```
Background and periodic test activity
=====================================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_core_threads.mp4

1. Run the test_threads.py example from the core directory.

2. The terminal shows a background test and a periodic test. The periodic test prints
   its message repeatedly while it runs.

3. Both demonstrations complete and an HTML report is created.

4. Opening the tests in the report shows their results and captured background or
   periodic output.
```

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.

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.

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`.

```
#
# Copyright (c) 2023, HILSTER - 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: str) -> None:
    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() -> None:
    ping("localhost")


@htf.data(["8.8.8.8", "heise.de", "hilster.io"])
def test_ping_hosts(hostname: str) -> None:
    ping(hostname)
```

```
@htf.yaml_data("hosts.yml")
def test_ping_hosts_file(hostname: str) -> None:
    ping(hostname)


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

To run the demo, run

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

To run the demo with asyncio, run

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

Data-driven host tests

Download recording · Download transcript

```
Data-driven host tests
======================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_ddt_pinging.mp4

1. Change to the data_driven_testing directory and execute test_ping_hosts.py.

2. The terminal runs the example for the supplied hosts and reports their results.

3. The generated HTML report lists the individual data-driven test instances, including
   their host parameters.

4. The chapter contains the current host-test source and its execution command.
```

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`.

```
#
# Copyright (c) 2023, HILSTER - 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: str, port: int) -> None:
    """
    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: str, port: int) -> None:
    """
    The headers must match the parameters.
    A test is dynamically generated for each line in the csv data.
    """
    print("test", hostname, port)


@htf.data(range(100))
def test_success(number: int) -> None:
    pass


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

To run the demo, run

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

To run the demo with asyncio, run

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

Tests supplied by external data

Download recording · Download transcript

```
Tests supplied by external data
===============================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_ddt_externaldata.mp4

1. Change to the data_driven_testing directory and execute test_data.py.

2. The recording shows external data files alongside the terminal and the generated HTML
   report.

3. Test instances are generated from the supplied data. Opening an instance in the
   report displays its parameters, such as a host and port.

4. The source listing and input files in the chapter explain how the current example
   consumes external data.
```

Combinatorial 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 combinatorial decorators.

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

```python
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import htf


@htf.product(htf.YAMLFileIterator("hosts.yml"), htf.YAMLFileIterator("ports.yml"))
def test_hostname_and_service(hostname: str, port: int) -> None:
    print("test", hostname, port)


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

To run the demo, run

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

To run the demo with asyncio, run

```
cd data_driven_testing_async
htf -o test_combinatorial.py
```

Combinatorial test data

Download recording · Download transcript

```
Combinatorial test data
=======================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_ddt_combinatoric.mp4

1. Change to the data_driven_testing directory and run the combinatorial example, named
   test_combinatoric.py in the historical recording.

2. htf combines values from hosts.yml and ports.yml to generate host-and-port test
   cases.

3. The terminal reports the generated cases; the HTML report displays their host and
   port parameters.

4. The on-screen explanation lists YAML, JSON, and CSV as supported external-data
   formats. The current chapter names its source example test_combinatorial.py.
```

Behavior Driven Development

Behavior Driven Development (BDD) is an agile software development technique. BDD enables non-technical participants to participate in writing tests.

The HILSTER Testing Framework enables users to write features in a gherkin-like language with add-ons that can be run as tests with seamless integration into the existing ecosystem.

Tests are written in form of Features, Rules, Scenarios and Steps, which are either Given, When, or Then steps. It is also possible to create Data Driven Feature Tests.

Thus every step is implemented in Python based on HILSTER Testing Framework.

HILSTER Testing Framework also offers requirements coverage in combination with the Dashboard.

For more information read the documentation for Behavior Driven Development.

This demo includes end-to-end tests for a web application calculator that runs in a browser.

Setup Webdriver

To be able to run the demo you need a recent webdriver for selenium for your browser. Please check the selenium-python documentation and selenium documentation for more information.

The webdriver shall be placed into `behavior_driven_development` folder or put into the `PATH` environment variable.

Behavior Driven Development based End-to-End Test

The software under test is a calculator running in a browser that can be used interactively.

There are 10 different requirements specified for the calculator.

ID	Description
REQ-1	The calculator shall have buttons for the digits 0 to 9
REQ-2	The calculator shall have a clear button
REQ-3	The calculator shall have an equals button
REQ-4	The calculator shall support the mathematical operation +
REQ-5	The calculator shall support the mathematical operation -
REQ-6	The calculator shall support the mathematical operation *
REQ-7	The calculator shall support the mathematical operation /
REQ-8	The calculator shall support integer inputs
REQ-9	The calculator shall support float inputs
REQ-10	The calculator shall have a decimal point button

The demo consists of multiple tests that are written using features and Python glue code.

The following code blocks show the features covering all requirements.

Digit Buttons Feature

The Digit Buttons feature covers requirement `REQ-1` and checks wheather the calculator offers all digit buttons.

```
Feature: Digit Buttons
  This feature describes the digit buttons.

  Requirements: REQ-1
```

```
  Scenario: Digit Buttons
    Given the calculator is cleared
    When 1234567890 is entered
    Then the result is 1234567890
```

Clear Button Feature

The Clear Button feature covers requirement `REQ-2` and checks wheather the calculator can be cleared.

```
Feature: Clear Button
  This feature describes the clear button.

  Requirements: REQ-2

  Scenario: Clear Button
    Given the calculator is cleared
    When 23 is entered
    And the calculator is cleared
    Then the result is 0
```

Equals Button Feature

The Equals Button feature covers `REQ-3` and checks wheather the calculator is cleared then the clear button is pressed.

```
Feature: Equals Button
  This feature describes the equals button.

  Requirements: REQ-3

  Scenario: Equals Button
    Given the calculator is cleared
    When 1+2 is entered
    And the equals button is pressed
    Then the result is 3
```

Addition Feature

The Addition Feature covers requirement `REQ-4` and checks wheather the calculator supports the addition operation. Is uses a scenario outline.

```
Feature: Addition
  This feature describes the addition.

  Requirements: REQ-4

  Scenario Outline: Addition
    Given the calculator is cleared
    When <a> is added to <b>
    And the equals button is pressed
```

```
    Then the result is <result>

    Examples: Addition
      |   a |   b |
      |   1 |   1 |
      |   2 |  12 |
      | 123 | 456 |
```

Subtraction Feature

The Subtraction Feature covers requirement `REQ-5` and checks wheather the calculator supports the subtraction operation. Is uses a scenario outline.

```
Feature: Subtraction
  This feature describes the subtraction.

  Requirements: REQ-5

  Scenario Outline: Subtraction
    Given the calculator is cleared
    When <a> is subtracted by <b>
    And the equals button is pressed
    Then the result is <result>

    Examples: Subtraction
      |    a |  b |
      |   11 |  1 |
      |   23 |  5 |
      | 1337 | 23 |
```

Multiplication Feature

The Multiplication Feature covers requirement `REQ-6` and checks wheather the calculator supports the multiplication operation. Is uses a scenario outline.

```
Feature: Multiplication
  This feature describes the multiplication.

  Requirements: REQ-6

  Scenario Outline: Multiplication
    Given the calculator is cleared
    When <a> is multiplied by <b>
    And the equals button is pressed
    Then the result is <result>

    Examples: Multiplication
      |    a |  b |
      |   11 |  1 |
      |   23 |  5 |
      | 1337 | 23 |
```

Division Feature

The Division Feature covers requirement `REQ-7` and checks wheather the calculator supports the division operation. Is uses a scenario outline.

```
Feature: Division
  This feature describes the division.

  Requirements: REQ-7

  Scenario Outline: Division
    Given the calculator is cleared
    When <a> is divided by <b>
    And the equals button is pressed
    Then the result is <result>

    Examples: Division
      |   a |  b |
      |  10 |  2 |
      |  12 |  3 |
      | 900 | 30 |
```

Decimal Point Feature

The Decimal Point Feature covert requirements `REQ-8`, `REQ-9` and `REQ-10` and checks wheather the calculator supports integer and float inputs.

```
Feature: Decimal Point
  This feature describes the decimal point.

  Requirements: REQ-8, REQ-9, REQ-10

  Scenario: Decimal Point
    Given the calculator is cleared
    When 10+2.5 is entered
    And the equals button is pressed
    Then the result is 12.5
```

External Data Feature

The External Data Feature covers requirements `REQ-4`, `REQ-5`, `REQ-6` and `REQ-7` to check all mathematical operations using an external data source `input-data.csv`.

```
Feature: External Data
  This feature describes the external data scenario outlines.

  Requirements: REQ-4, REQ-5, REQ-6, REQ-7

  Scenario Outline: External Data
    Given the calculator is cleared
    When <a><operation><b> is entered
    And the equals button is pressed
```

```
    Then the result is <result>

    Examples: Foo
      CSV: ../input-data.csv
```

The content of `input-data.csv` looks like the following:

```
a,operation,b,result
1,+,2,3
3,-,1,2
10,*,100,1000
100,/,10,10
```

Python Generator Data Feature

The Python Generator Data Feature covers requirements `REQ-4`, `REQ-5`, `REQ-6` and `REQ-7` to check all mathematical operations using a Python based data generator.

```
Feature: Python Generator Data
  This feature describes the external data scenario outlines.

  Requirements: REQ-4, REQ-5, REQ-6, REQ-7

  Scenario Outline: Python Generator Data
    Given the calculator is cleared
    When <a><operation><b> is entered
    And the equals button is pressed
    Then the result is <result>

    Examples:
      Generator: Calculations-Generator
```

Fixture

Communicating with the software under test is implemented using a fixture in `calculator.py`:

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

from typing import Generator, Union, Dict, Optional

import htf
import os
import time
from selenium import webdriver  # type: ignore
from selenium.webdriver.common.by import By  # type: ignore
from selenium.webdriver.remote.webelement import WebElement


@htf.fixture("session")
```

```
def driver() -> Generator[webdriver.Chrome, None, None]:  # type: ignore
    """
    Webdriver fixture
    """
    os.environ["PATH"] = os.environ["PATH"] + ":."
    driver = webdriver.Chrome()
    yield driver
    driver.close()


class Calculator:
    """
    Digital twin for web application calculator.
    """

    def __init__(self, driver: webdriver.Chrome, settings: htf.Settings) -> None:  # type: ignore
        self.driver = driver
        self.settings = settings
        self.operations: Dict[str, WebElement] = {}
        self.numbers_elements: Dict[int, WebElement] = {}
        self.decimal_point: Optional[WebElement] = None
        self.clear: Optional[WebElement] = None

        self.delay = settings.delay

        # open webapp
        self.driver.get(f"file://{os.path.join(os.path.dirname(__file__), settings.webapp)}")

        # update elements
        self.get_elements()

    def get_elements(self) -> None:
        """
        Get all elements.
        """

        self.clear = self.driver.find_element(By.CSS_SELECTOR, "#clear")  # Clear
        self.result = self.driver.find_element(By.CSS_SELECTOR, "#viewer")  # Results viewer
        self.equals = self.driver.find_element(By.CSS_SELECTOR, "#equals")  # Equal button

        numbers = self.driver.find_elements(By.CSS_SELECTOR, ".num")  # List of numbers

        for number in numbers:
            try:
                self.numbers_elements[int(number.get_attribute("data-num"))] = number
            except ValueError:
                assert number.get_attribute("data-num") == "."
                self.decimal_point = number

        operations = self.driver.find_elements(By.CSS_SELECTOR, ".ops")  # List of operators

        for operation in operations:
            self.operations[operation.get_attribute("data-ops")] = operation

    def input(self, element: str) -> None:
        """
        Input on an element in the web application.

        Args:
            element: the element to be clicked
```

```
        """
        assert len(element) == 1, "element must contain 1 element"

        if element in ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]:
            # enter a number
            self.numbers_elements[int(element)].click()
        elif element in ["."]:
            # enter decimal point
            self.decimal_point.click()  # type: ignore
        elif element in ["+", "-", "*", "/"]:
            # enter operation
            mapping = {"+": "plus", "-": "minus", "*": "times", "/": "divided by"}
            self.operations[mapping[element]].click()
        elif element in ["="]:
            # equals
            self.equals.click()
        elif element in ["C"]:
            # clear
            self.clear.click()  # type: ignore

        if self.delay:
            time.sleep(self.delay)

    def input_expression(self, expression: str) -> None:
        """
        Input an expression
        """
        if expression:
            for element in expression:
                self.input(element)

    def calculate_result(self) -> None:
        """
        Enter = to calculate the result.
        """
        self.input("=")

    def get_result(self) -> Union[float, str]:
        """
        Enter = and return the result
        """
        # get result
        try:
            return float(self.result.text)
        except ValueError:
            return self.result.text

    def evaluate(self, expression: str) -> Union[float, str]:
        """
        Evaluate expression and return the result.

        Args:
             expression: the expression to be evaluated

        Returns:
            float: the result from the web application
        """
        self.input_expression(expression + "=")
        return self.get_result()
```

```
    def add(self, a: float, b: float) -> Union[float, str]:
        """
        Evaluate a+b
        """
        return self.evaluate(f"{a}+{b}")

    def subtract(self, a: float, b: float) -> Union[float, str]:
        """
        Evaluate a-b
        """
        return self.evaluate(f"{a}-{b}")

    def multiply(self, a: float, b: float) -> Union[float, str]:
        """
        Evaluate a*b
        """
        return self.evaluate(f"{a}*{b}")

    def divide(self, a: float, b: float) -> Union[float, str]:
        """
        Evaluate a/b
        """
        return self.evaluate(f"{a}/{b}")


@htf.fixture("test")
def calculator(driver: webdriver.Chrome, settings: htf.Settings) -> Generator[Calculator, None,
None]:
    calculator = Calculator(driver, settings)
    calculator.clear.click()  # type: ignore
    yield calculator
```

Steps

In order to be able to run features as tests, glue code is required. The Python glue code is implemented in `steps.py`:

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

from typing import Dict, Any, Union, Generator

import htf
import htf.assertions as assertions
from behavior_driven_development.calculator import Calculator


@htf.given("the calculator is cleared")
@htf.when("the calculator is cleared")
def clear_calculator(calculator: Calculator) -> None:
    calculator.clear.click()  # type: ignore


@htf.when("{expression} is entered")
def enter_expression(expression: str, calculator: Calculator) -> None:
```

```
    calculator.input_expression(expression)


@htf.when("{a} is added to {b:g}")
def add(a: float, b: float, calculator: Calculator) -> None:
    calculator.input_expression(f"{a}+{b}")


@htf.when("{a} is subtracted by {b}")
def subtract(a: float, b: float, calculator: Calculator) -> None:
    calculator.input_expression(f"{a}-{b}")


@htf.when("{a} is multiplied by {b}")
def multiply(a: float, b: float, calculator: Calculator) -> None:
    calculator.input_expression(f"{a}*{b}")


@htf.when("{a} is divided by {b}")
def divide(a: float, b: float, calculator: Calculator) -> None:
    calculator.input_expression(f"{a}/{b}")


@htf.when("the equals button is pressed")
def equals_button_pressed(calculator: Calculator) -> None:
    calculator.calculate_result()


@htf.then("the result is {expected_result:g}")
def check_result(expected_result: float, calculator: Calculator) -> None:
    assertions.assert_equal(expected_result, calculator.get_result())


def _clean_up_data(data: Dict[str, str]) -> Dict[str, float]:
    a: Any = data["a"]
    b: Any = data["b"]
    try:
        if "." in a:
            a = float(a)
        else:
            a = int(a)
    except TypeError:
        pass
    try:
        if "." in b:
            b = float(b)
        else:
            b = int(b)
    except TypeError:
        pass

    return_data: Dict[str, float] = {}
    return_data["a"] = float(a)
    return_data["b"] = float(b)
    return return_data


@htf.transformation("Addition")
def expected_addition_result(data: Dict[str, str]) -> Dict[str, float]:
    clean_data = _clean_up_data(data)
```

```
    clean_data["result"] = clean_data["a"] + clean_data["b"]
    return clean_data


@htf.transformation("Subtraction")
def expected_subtraction_result(data: Dict[str, str]) -> Dict[str, float]:
    clean_data = _clean_up_data(data)
    clean_data["result"] = clean_data["a"] - clean_data["b"]
    return clean_data


@htf.transformation("Multiplication")
def expected_multiplication_result(data: Dict[str, str]) -> Dict[str, float]:
    clean_data = _clean_up_data(data)
    clean_data["result"] = clean_data["a"] * clean_data["b"]
    return clean_data


@htf.transformation("Division")
def expected_division_result(data: Dict[str, Union[float, str]]) -> Dict[str, Union[float, str]]:
    data["result"] = float(data["a"]) / float(data["b"])
    return data


@htf.data_generator("Calculations-Generator")
def generate_calculations() -> Generator[Dict[str, Union[float, str]], None, None]:
    yield dict(a=1, operation="+", b=2, result=3)
    yield dict(a=3, operation="-", b=1, result=2)
    yield dict(a=10, operation="*", b=100, result=1000)
    yield dict(a=100, operation="/", b=10, result=10)
```

Run Tests

To run the demo, run

```
cd behavior_driven_development
htf -o calculator.py steps.py features/*.feature
```

To run the demo with asyncio, run

```
cd behavior_driven_development_async
htf -o calculator.py steps.py features/*.feature
```

Behavior-driven calculator tests

Download recording · Download transcript

```
Behavior-driven calculator tests
================================
```

```
Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_web_bdd.mp4

1. The recording places the htf terminal beside the JavaScript calculator under test.

2. Feature scenarios supply calculator inputs and expected results. The terminal
   displays Given, When, and Then steps while the browser presses calculator buttons.

3. Addition and subtraction scenarios include examples supplied by external data and a
   Python generator. The terminal reports the result of each scenario.

4. The accompanying chapter contains the feature files, step definitions, calculator
   page, and command needed to reproduce the example.
```

Alternative Implementation

There is an alternative implementation for the same tests without using Behavior Driven Development but plain Python code:

The tests are implemented in `test_web_application.py`:

```python
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

# example how to test a web application calculator

import htf
from behavior_driven_development.calculator import Calculator


@htf.requirements("REQ-1")
def test_digit_button(calculator: Calculator) -> None:
    """
    This test asserts that all digits input buttons exists.
    """
    htf.assert_almost_equal(calculator.evaluate("1234567890"), 1234567890)  # type: ignore


@htf.requirements("REQ-2")
def test_clear_button(calculator: Calculator) -> None:
    """
    This test asserts that the clear button clears the result to zero
    """
    htf.assert_almost_equal(float(calculator.evaluate("23")), 23)  # type: ignore
    calculator.clear.click()  # type: ignore
    htf.assert_almost_equal(float(calculator.get_result()), 0)  # type: ignore
```

```python
@htf.requirements("REQ-3")
def test_equals_button(calculator: Calculator) -> None:
    """
    This test asserts that the equals button evaluates the result of the input.
    """
    htf.assert_almost_equal(float(calculator.evaluate("1+2")), 3)


@htf.requirements("REQ-4")
@htf.data([[1, 1], [2, 12], [123, 456]], unpack=True)
def test_addition(a: int, b: int, calculator: Calculator) -> None:
    expected_result = a + b
    htf.assert_almost_equal(float(calculator.add(a, b)), expected_result)


@htf.requirements("REQ-5")
@htf.data([[11, 1], [23, 5], [1337, 23]], unpack=True)
def test_subtraction(a: int, b: int, calculator: Calculator) -> None:
    expected_result = a - b
    htf.assert_almost_equal(float(calculator.subtract(a, b)), expected_result)


@htf.requirements("REQ-6")
@htf.data([[11, 1], [23, 5], [1337, 23]], unpack=True)
def test_multiplication(a: int, b: int, calculator: Calculator) -> None:
    expected_result = a * b
    htf.assert_almost_equal(float(calculator.multiply(a, b)), expected_result)


@htf.requirements("REQ-7")
@htf.data([[10, 2], [12, 3], [900, 30]], unpack=True)
def test_division(a: int, b: int, calculator: Calculator) -> None:
    expected_result = a / b
    htf.assert_almost_equal(float(calculator.divide(a, b)), expected_result)


@htf.requirements("REQ-4", "REQ-5", "REQ-6", "REQ-7")
@htf.csv_data("input-data.csv")
def test_external_data(a: int, operation: str, b: int, result: int) -> None:
    print(a, operation, b, result)


@htf.requirements("REQ-8", "REQ-9", "REQ-10")
def test_integer_and_float_inputs(calculator: Calculator) -> None:
    """
    This test asserts that integer as well as float inputs are supported.
    """
    htf.assert_almost_equal(float(calculator.evaluate("10+2.5")), 12.5)


if __name__ == "__main__":
    htf.main(open_report=True)
```

To run the demo, run

```
cd behavior_driven_development
htf -o calculator.py test_web_application.py
```

To run the demo with asyncio, run

```
cd behavior_driven_development_async
htf -o calculator.py test_web_application.py
```

Python calculator browser tests

Download recording · Download transcript

```
Python calculator browser tests
===============================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_web.mp4

1. The terminal runs the Python web-application example while an automated browser
   operates the JavaScript calculator.

2. The collected tests cover calculator buttons and arithmetic examples. Browser input
   and displayed calculator results change as individual tests execute.

3. The terminal shows the current test and its success result, followed by a running
   test count.

4. The documented Python test_web_application.py example provides the current source and
   execution instructions.
```

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.

Solve Arithmetic Problems

This demo lets you solve arithmetic problems.

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

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import htf
import htf.assertions as assertions
from operator import add, sub, mul
from random import randint, choice


def test_solve_arithmetic_problems(
    interaction: htf.fixtures.interaction,
) -> None:
    """
    This test shows how to interactively solve simple arithmetic problems.

    Args:
         interaction (htf.fixtures.interaction): the interaction 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
    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
```

Note

If the default TCP/IP port `8080` is not usable on your system you can change it by appending `--interactive-port <port>` to the command, e.g.:

```
htf -i -o --interactive-port 12345 test_solve_arithmetic_problems.py
```

The dialogs will appear in the test report afterwards.

Interactive arithmetic prompts

Download recording · Download transcript

```
Interactive arithmetic prompts
==============================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_interactive_arithmetic.mp4

1. Change to the interaction directory and start the arithmetic example with htf's -i
   interactive option.

2. The terminal displays an interactive-session address and QR code. Open the displayed
   session in a browser.

3. Answer the example arithmetic prompts and complete the interaction.

4. The generated HTML report retains the prompts, responses, and final test result.
```

Captcha Solver

This demo lets you solve captchas.

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

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import htf
import htf.assertions as assertions
from captcha.image import ImageCaptcha  # type: ignore
from random import randint


def test_solve_captchas(
    interaction: htf.fixtures.interaction,
) -> None:
    """
    This test shows how to interactively solve captchas.

    Args:
         interaction (htf.fixtures.interaction): the interaction 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)
            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.

Interactive CAPTCHA demonstration

Download recording · Download transcript

```
Interactive CAPTCHA demonstration
=================================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_interactive_captcha.mp4

1. Change to the interaction directory and start the CAPTCHA example with htf's -i
   interactive option.

2. Open the interactive-session address displayed by the example.

3. The demonstration asks the operator to enter the characters shown in its example
   prompt.

4. After completing the example, open the generated HTML report to inspect the recorded
   interaction and test result.
```

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.

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`.

Merging test reports

Download recording · Download transcript

```
Merging test reports
====================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_reporttool_merging.mp4

1. Change to the report_tool example directory.

2. Run report-tool with the provided HTML and XML test reports and a new output
   filename.

3. The terminal confirms that the merged report has been written.

4. Open the new HTML report to inspect test entries brought together from the input
   reports. The chapter provides the current command syntax.
```

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
```

Converting a test-report format

Download recording · Download transcript

```
Converting a test-report format
===============================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_reporttool_converting.mp4

1. Change to the report_tool example directory.

2. Run report-tool with the existing HTML report, the desired XML format, and the output
   filename.

3. The terminal confirms creation of the converted XML report, which also appears in the
   directory view.

4. The chapter provides the current command syntax for selecting input, output, and
   format.
```

Dashboard

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

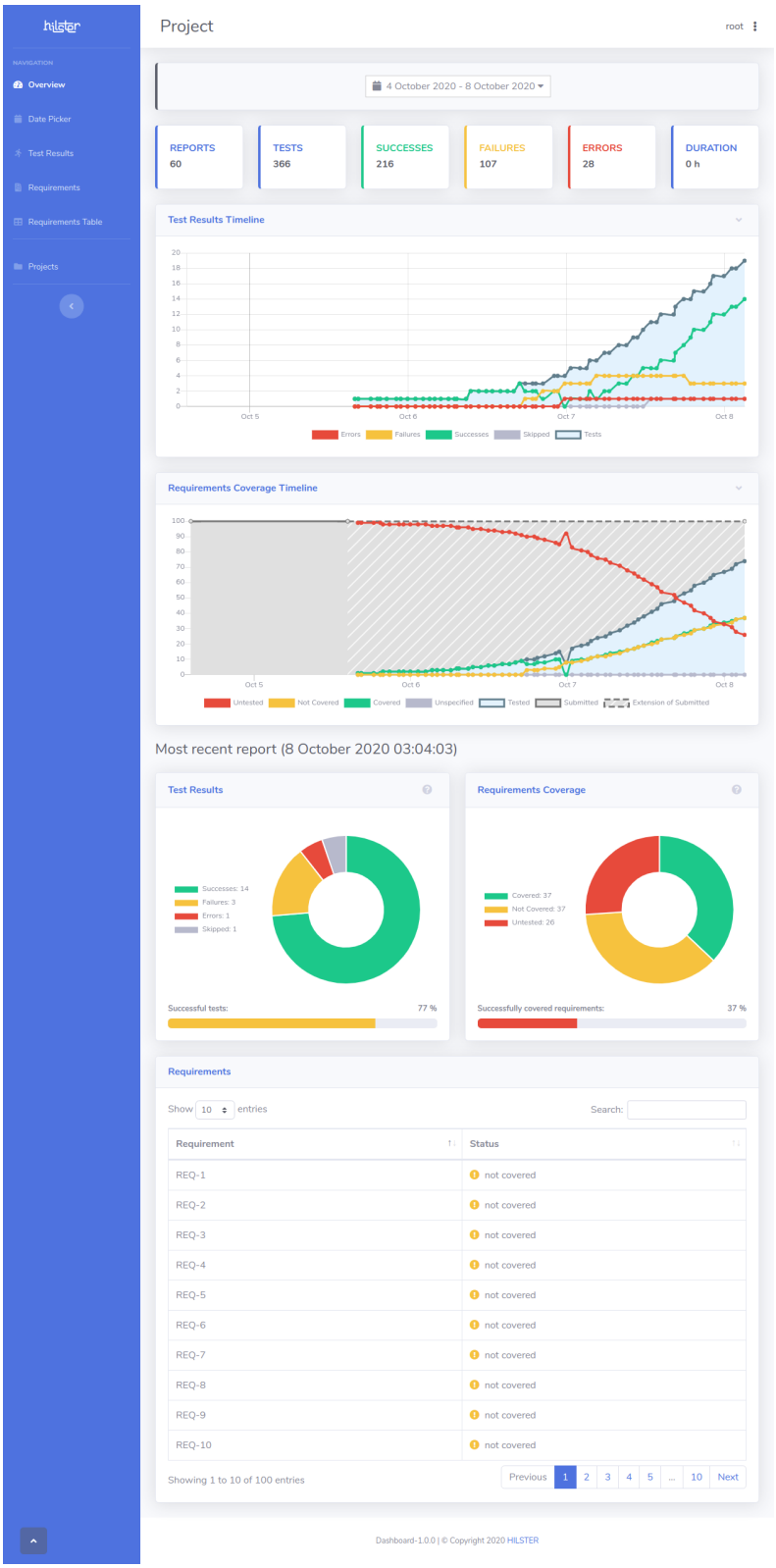
Fully integrated into QABench, test reports can be sent automatically to the Dashboard upon conclusion of a test run.

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.



Dashboard browser overview

Download recording · Download transcript

```
Dashboard browser overview
==========================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: dashboard_overview.mp4

1. The recording starts with the Dashboard project list and its report counts, test-
   result summaries, requirements coverage, and last-modified column.

2. Opening a project displays test-result and requirements-coverage timelines.

3. Recent-report doughnut charts summarize test outcomes and requirements coverage.
   Pointing at chart segments exposes their values.

4. The project page also lists individual requirements and their coverage status. The
   associated Dashboard documentation explains these views in detail.
```

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`.

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-project 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.

Importing a Dashboard project

Download recording · Download transcript

```
Importing a Dashboard project
=============================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_dashboard_1.mp4
```

1. Log in to the local demonstration Dashboard with its configured account and change to the dashboard example directory.

2. Create the imported_project project using the Dashboard command-line interface.

3. Import the provided project_export.zip data into that project. The terminal displays progress while reports and requirements are imported.

4. Log in through the browser, open the imported project, and inspect its populated statistics and timelines.

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
```

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import htf
import htf.assertions as assertions

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: int) -> None:
    pass
```

```
@htf.data(range(failures))
@htf.requirements(*failing_requirements)
def test_failure(i: int) -> None:
    assertions.assert_false(True, "This test fails!")


@htf.data(range(errors))
@htf.requirements(*errored_requirements)
def test_error(i: int) -> None:
    raise Exception("This test has an error!")


@htf.data(range(skipped))
def test_skip(i: int) -> None:
    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.

Uploading reports and requirements

Download recording · Download transcript

```
Uploading reports and requirements
==================================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.

Original recording: demo_dashboard_2.mp4

1. Create a new empty example project in the local Dashboard.

2. Upload the example requirements to that project.

3. Run test_example.py with the Dashboard project URL as the report target. The
   demonstration includes intentional failed tests so the result categories can be
   inspected.
```

```
4. Repeat uploads as desired, then inspect the project's test-result and requirements-
   coverage charts and its latest report.
```

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.

Combining requirements coverage from multiple tools

Download recording · Download transcript

```
Combining requirements coverage from multiple tools
==================================================

Visual transcript and walkthrough of the silent recording.
Based on reviewed screen captions/interface actions and the associated written example.
```

```
Original recording: demo_dashboard_3.mp4

1. Create the coverage measurement project and upload its requirements.

2. Run the integrationtests.py example to create an htf HTML report.

3. Use report-tool to combine that report with the provided JUnit XML report and send
   the combined result to the Dashboard project.

4. Open the project in the browser and inspect the combined test results and
   requirements coverage.
```

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.

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`.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import oser  # type: ignore


class U32Payload(oser.ByteStruct):
    """
    +----------+------------------------+--------+----------------------------+
    | Byte     | Name                   | Type   | Description                |
    +==========+========================+========+============================+
    | 1..4     | value                  | UInt32 | The value                  |
    +----------+------------------------+--------+----------------------------+
    """

    def __init__(self) -> None:
        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) -> None:
        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)


class StatusPayload(oser.ByteStruct):
    """
    +----------+------------------------+--------+----------------------------+
    | Byte     | Name                   | Type   | Description                |
    +==========+========================+========+============================+
    | 0..3     | uptime                 | UInt32 | Uptime in seconds          |
    +----------+------------------------+--------+----------------------------+
    | 4        | status                 | UInt8  | Status bits                |
    +----------+------------------------+--------+----------------------------+
    """

    def __init__(self) -> None:
        super(StatusPayload, self).__init__()
        self.uptime = oser.UBInt32()
        self.status = oser.UBInt8()


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


def DataObjectId() -> oser.Enum:  # noqa: N802
    """
    +-----------+------------------------+
    | 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) -> None:
        super(DataObject, self).__init__()

        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() -> oser.Enum:  # noqa: N802
    """
    +-----------+-------------------------+
    | 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) -> None:
        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)
```

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

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

import time

import oser  # type: ignore
from protocol import Message  # type: ignore


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)

    # 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`.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

from protocol import Message  # type: ignore

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.

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.

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`.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

from typing import Generator

import htf
from htf.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() -> Generator[ModbusTCPDevice, None, None]:
    """
    Modbus device simulator fixture.
    """
    device = ModbusTCPDevice(ADDRESS, port=PORT)
    yield device
    device.close()


@htf.fixture("test")
def client(
    device: ModbusTCPDevice,
) -> Generator[ModbusTCPClient, None, None]:  # 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: ModbusTCPDevice, device: ModbusTCPDevice) -> None:
    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])
        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: ModbusTCPDevice, device: ModbusTCPDevice) -> None:
    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
```

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`.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

from typing import Optional, Generator

import htf
import oser  # type: ignore
from datetime import datetime
from htf.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: Optional[datetime] = None) -> None:
        """
        ``DateTime`` data type.

        Args:
            value: 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: datetime) -> None:
        """
        Set values from a ``datetime`` instance.

        Args:
            value: 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) -> datetime:
        """
        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() -> Generator[ModbusTCPDevice, None, None]:
    device = ModbusTCPDevice(ADDRESS, port=PORT)
    yield device
    device.close()


@htf.fixture("test")
def client(device: ModbusTCPDevice) -> Generator[ModbusTCPClient, None, None]:  # 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: ModbusTCPClient, device: ModbusTCPDevice) -> None:
    """
    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: DateTime = client.read_input_register_data(23, DateTime)  # type: ignore

    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__":
    htf.main()
```

To execute the demo, run

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

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.

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`.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

from typing import Generator

import htf
from htf.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() -> Generator[HartDeviceCommunication, None, None]:
    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.
"""


def test_query(hart_device: HartDeviceCommunication) -> None:
    response = hart_device.query(HartFrame(1))  # Command 1 - Read Primary Variable
    print("Response:")
    print(response)


def test_query_context() -> None:
    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: HartDeviceCommunication) -> None:
    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
```

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.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
```

```
#

from typing import Generator, Tuple

import htf
import oser  # type: ignore
from htf.hart import HartFrame, ResponseCode, FieldDeviceStatus
from htf.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
#
#         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):  # noqa: N801
    def __init__(self) -> None:
        super(Command123_ReadRandomBytesRequest, self).__init__()
        self.number_of_bytes = oser.UBInt8()


class Command123_ReadRandomBytesResponse(HartResponse):  # noqa: N801
    def __init__(self) -> None:
        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) -> Generator[Tuple[int, oser.Lazy], None, None]:
        """
        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) -> Generator[Tuple[int, oser.Lazy], None, None]:
        """
        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.
#
# Example:
#     with HartDeviceCommunication(comport='COM1', decoder_generator=MyDeviceHartFrame) as com:
#         response = com.query(MyDeviceHartFrame(123))
#         print(response)


def test_custom_command() -> None:
    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
```

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.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

from typing import Optional

import htf
from htf.hart import HartFrame
from htf.hart.slave_simulator import HartSlaveSimulator
from hart.test_custom_commands import MyDeviceHartFrame
from random import randint
```

```
from unittest.mock import patch

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


class SlaveSimulator(HartSlaveSimulator):
    def __init__(self, comport: str) -> None:
        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: int) -> bytes:
        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: HartFrame) -> Optional[HartFrame]:
        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.
"""


def test_start_simulator_success() -> None:
    with patch("htf.hart.slave_simulator.HartInterface") as 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()


def test_start_simulator_error_response() -> None:
    with patch("htf.hart.slave_simulator.HartInterface") as 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(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
```

HART Simulation/Hardware Tests

When writing tests to check device behavior you may want to run them against a simulator and your device without having to develop the same test twice. This can be accomplished by using a combination of fixtures and tags. Once the test is written you can select the target by passing the desired fixture tag to the runner.

See `hart/test_read_primary_variable.py` for an example.

```
#
# Copyright (c) 2023, HILSTER - https://hilster.io
# All rights reserved.
#

from typing import Generator, Any, Optional
```

```
import htf
from unittest.mock import patch
from htf.hart import HartFrame, HartDeviceCommunication
from htf.hart.slave_simulator import HartSlaveSimulator


COM_PORT = "/dev/ttyUSB0"  # !!! Change this to the port your device is connected to


class PrimaryVariableSimulator(HartSlaveSimulator):
    def __init__(self, comport: str) -> None:
        super(PrimaryVariableSimulator, self).__init__(
            comport=comport, decoder_generator=HartFrame, device_type=0x1234, device_id=0x56
        )
        self.broken = False

    def handle_command1(self, request: HartFrame) -> Optional[HartFrame]:
        response = self._create_response(request)
        response.payload.primary_variable.set(123.45)
        if self.broken:
            response.payload.response_code.set("error_device_specific_command_error")
        return response


@htf.tags("simulator")
@htf.fixture(scope="test", name="device")
def device_simulator() -> Generator[PrimaryVariableSimulator, None, None]:
    patch("htf.hart.slave_simulator.HartInterface")
    sim = PrimaryVariableSimulator("/dev/ttyUSB0")
    sim.run(2)
    yield sim


@htf.tags("hardware")
@htf.fixture(scope="test", name="device")
def device_hardware() -> Generator[Any, None, None]:
    yield object()  # Replace this with an object controlling your device


@htf.tags("simulator")
@htf.fixture(scope="test", name="interface")
def interface_simulator(device: Any) -> Generator["SimulatorInterface", None, None]:  # type:
ignore  # noqa: F821
    class SimulatorInterface:
        def query(self, request: HartFrame) -> Optional[HartFrame]:
            return device._get_response_from_handler(request)  # type: ignore

    yield SimulatorInterface()


@htf.tags("hardware")
@htf.fixture(scope="test", name="interface")
def interface_hardware(device: Any) -> Generator[HartDeviceCommunication, None, None]:
    yield HartDeviceCommunication(COM_PORT)


def test_read_primary_variable(
    device: Any,
    interface: "SimulatorInterface",  # type: ignore  # noqa: F821
    step: htf.fixtures.step,
```

```
) -> None:
    with step("Read primary variable"):
        request = HartFrame(1)
        request.address.address.set(0x1234000056)  # the slave will only respond to commands
addressed to it
        print("Request:")
        print(request)

        response = interface.query(request)
        print("Response:")
        print(response)

    with step("Validate primary variable and response code"):
        htf.assert_equal(response.payload.response_code.get(), "success")  # expect a successful
response
        htf.assert_almost_equal(
            response.payload.primary_variable.get(), 123.45
        )  # number of requested bytes should be echoed

    with step("Break device"):
        device.broken = True

    with step("Read primary variable again"):
        print("Request:")
        print(request)

        response = interface.query(request)
        print("Response:")
        print(response)

    with step("Validate response code indicates device specific error"):
        htf.assert_equal(response.payload.response_code.get(),
"error_device_specific_command_error")  # expect an error
```

To execute the demo against the simulator, run

```
htf -o -F simulator hart/test_read_primary_variable.py
```

likewise, to use a real device as the target run

```
htf -o -F hardware hart/test_read_primary_variable.py
```

Attention: You will need to change the ``COM_PORT` variable at the top of the test module to the port your device is connected to.

Continuous Integration

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`.

The existing job `htf_demo` contains the pipeline script that runs `htf` and stores and evaulates the reports.

```
node("master") {
    stage("Run tests using htf") {
        try {
            sh "htf -H results.html -x results.xml ../../../../../core/*.py || true"
        }
        finally {
            archiveArtifacts artifacts: 'results.html', fingerprint: true
            junit 'results.xml'
        }
    }
}
```

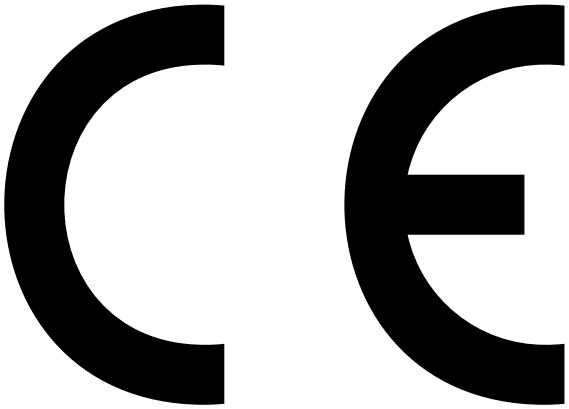
Cyber Resilience Act Conformity

This software is distributed by

> HILSTER Testing Solutions GmbH

Amtsstraße 10
22143 Hamburg
Germany

Statement of Conformity



This software is placed on the market as a product with digital elements in the sense of Regulation (EU) 2024/2847 (Cyber Resilience Act, “CRA”). It is therefore subject to the conformity assessment obligations of the CRA.

This software is classified as “standard” risk class under the CRA.

Security Measures

As required by the CRA, HILSTER Testing Solutions GmbH takes the following measures to ensure a high level of security:

- Regular review of the source code and the dependencies used
- Prompt provision of updates for known security vulnerabilities
- Minimised collection and storage of user data
- No telemetry or automatic transmission of data to third parties

Reporting Vulnerabilities

Security vulnerabilities can be reported at any time by sending an e-mail to security@hilster.io.

Note

This statement is for information purposes only and does not constitute a declaration of conformity within the meaning of the Cyber Resilience Act.

Changelog

htf-demo-4.3.0

- update dependencies

htf-demo-4.2.4

- update dependencies

htf-demo-4.2.3

- implement Cyber Resilience Act Conformity (CRA)
- update dependencies

htf-demo-4.2.2

- update documentation

htf-demo-4.2.1

- fix installation

htf-demo-4.2.0

- use `htf-4.2`

htf-demo-4.1.2

- add instructions how to change the TCP/IP port when running interactive tests

htf-demo-4.1.1

- fix command to import a project using the Dashboard

htf-demo-4.1.0

- use `htf-4.1`

htf-demo-4.0.1

- add support for Python 3.12

htf-demo-4.0.0

- use `htf-4.0`
- add demos for `asyncio`

htf-demo-3.0.0

- add demo for Behavior Driven Development
- use `htf-3.0`
- update dependencies

htf-demo-2.0.5

- update dependencies

htf-demo-2.0.4

- update docs

htf-demo-2.0.3

- install htf-community automatically

htf-demo-20200622

- add binary data demo

htf-demo-20200423

- add demo for HART communication

htf-demo-20200421

- add support for Python-3.8

htf-demo-20190710:

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

htf-demo-20190207:

- initial demo for core and continuous integration

PDF version

Click here for the PDF version of the documentation.