UNIVERSITÀ
DI TORINO

# Design and Development of the Digital Twin of a Greenhouse

947847 Eduard Antonovic Occhipinti

**SUPERVISOR**:
Prof. Ferruccio Damiani

**CO-SUPERVISORS**:
Prof. Einar Broch Jhonsen
Dr. Eduard Kamburjan
Dr. Rudolf Schlatte

# Digital Twins

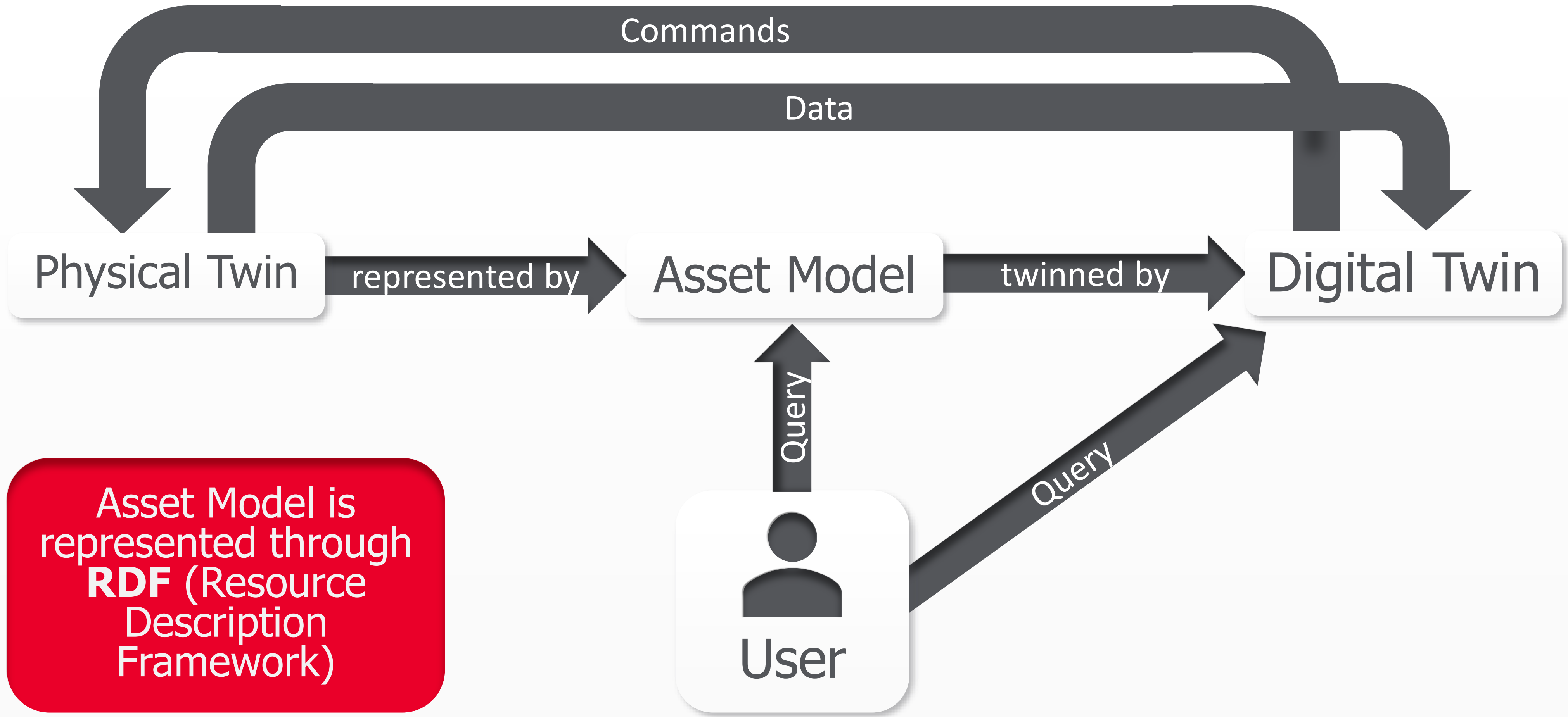
Live replicas of physical systems, generally connected to them in real time

Tools to understand and control **assets** in nature, industry or society at large

Meant to evolve over time

Uses **FMI** as standard, and interface that encapsulates **FMUs**

# High Level Representation of a Typical Digital Twin Architecture




Asset Model is represented through **RDF** (Resource Description Framework)

# SMOL

Programming language in active development at the research lab of the University of Oslo’s computer science department

**Oject-Oriented,** Runs on the JVM

Open Source

Supports **FMU** simulators as primitives

Encapsulates simulators in **FMO**s (Functional Mock-up Objects)

Can be used as a framework for creating digital twins

Natively supports querying of **knowledge bases** with **SPARQL** and **InfluxQL**

Supports **semantic reflection**, making it possible to add axioms on SMOL objects

Supports **semantic lifting** of the program state, enabling us to query it as a knowledge graph

# Setup

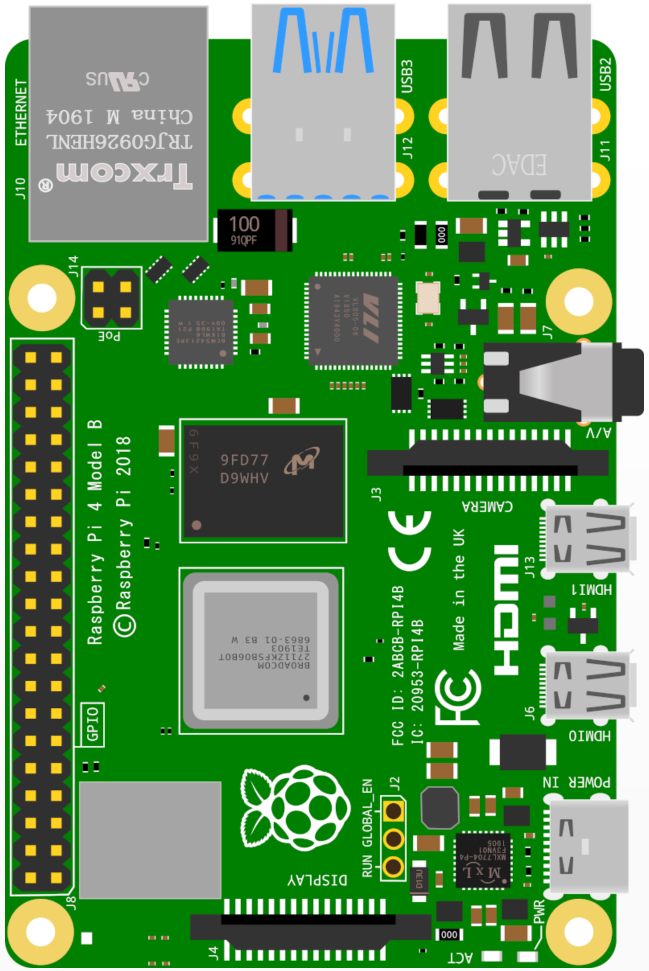
Used a total of 4 Raspberry Pi 4, configured with the help of bash scripts to make it easily repeatable

**Host**

**Actuator**

**Data Collectors**

**Router**



## Host

influxdata®

Hosts the Influx database

Controls the actuators through SSH and schedules the execution of the SMOL program

## Router

Configured with **hostapd** and **dsnmasq** for a local network

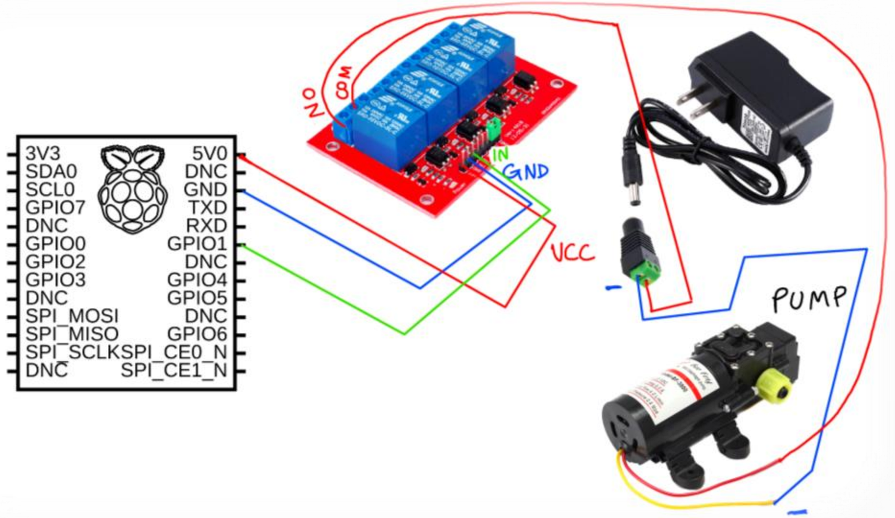
Used to route HTTP requests to the database

Enables communication between the computers through **SSH**

# Actuator

Runs the Python program that controls the actuators

Only one pump connected to a relay but easily expandable

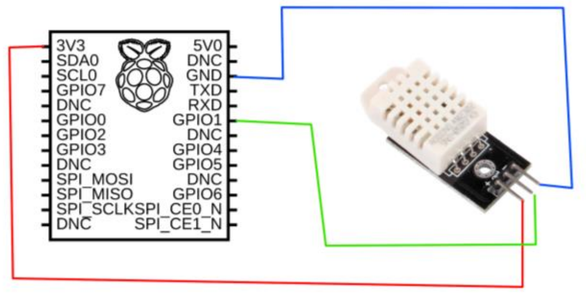


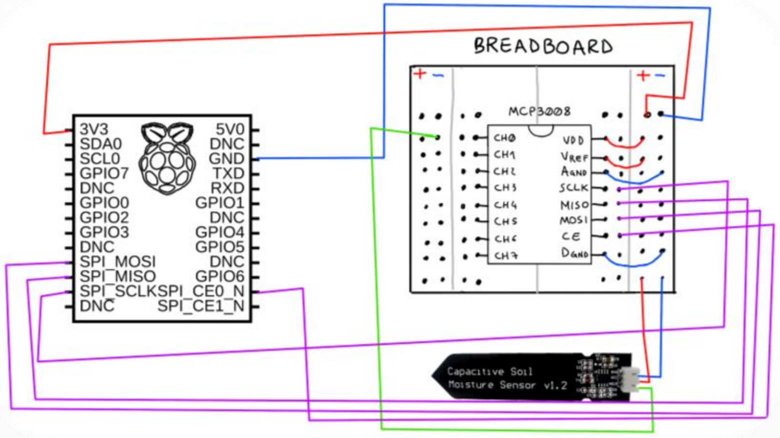
# Data Collectors

**NDVI**: Landstat Normalized Diffusion Index, used to quantify vegetation density and greenness and assess plant health

Connect to the sensors

Continuously send the measurements to the InfluxDB on the host




At the moment log:

- **Temperature**
- **Moisture**
- **Humidity**
- Light Level
- NDVI

SMOL

SMOL Program

SMOL Scheduler

Data Collector

Actuator

# SMOL Program



1. Reads the **asset model** [**OWL** ontology that formally describes the PT, its components, properties and relationship between them]

2. Generates **SMOL Objects** for each **individual** formalized in the asset model

3. Retrieves the sensor data associated with a given asset from the Influx database

4. If a plant needs to be watered it gets added to a list of plants to water with a simple check on the optimal value found in the ontology

# SMOL Scheduler

## **Gradle** project

1. Starts remotely the data collectors

2. **Syncs** the **asset model** so that changes are reflected in real time in the data collectors configuration files

3. Schedules at fixed rate the execution of the SMOL program

4. Lifts the state of the SMOL program with the REPL and sends a command to the actuators that starts the pumps for the plants in the «plantsToWater» list, lifted from SMOL

```
.
├── gradle/wrapper
├── src
│   ├── main
│   │   ├── java
│   │   │   └── no.uio.scheduler
│   │   │       ├── ConfigTypeEnum.java
│   │   │       ├── INIManager.java
│   │   │       ├── ModelReader.java
│   │   │       ├── Main.java
│   │   │       ├── ModelTypeEnum.java
│   │   │       ├── SMOLScheduler.java
│   │   │       ├── SSHSender.java
│   │   │       └── Utils.java
│   │   └── resources
│   │       :
│   │       ├── asset_model.ttl
│   │       └── greenhouse.smol
│   └── tests
├── .gitignore
├── .pre-commit-config.yaml
├── README.md
├── build.gradle
├── gradle.properties
├── gradlew
├── gradlew.bat
└── settings.gradle
```

# Data Collectors

Structured as a Python module

Data collected is trasferred to an **InfluxDB** database on the host computer

Configuration can be changed on-the-fly

Takes advantage of **git hooks** for formatting and static code analysis and **requirements file** to manage dependencies

```
.
├── .github
│   └── workflows
│       └── black.yml
├── collector
│   ├── __init__.py
│   ├── __main__.py
│   ├── assets
│   ├── config.ini.example
│   ├── config.py
│   ├── demo
│   ├── influx
│   ├── sensors
│   └── tests
├── .gitignore
├── .pre-commit-config.yaml
├── README.md
├── requirements.txt
└── scripts
```

```
.
└── assets
    ├── __init__.py
    ├── asset.py
    ├── greenhouse_asset.py
    ├── measurement_type.py
    ├── plant_asset.py
    ├── pot_asset.py
    ├── pump_asset.py
    ├── shelf_asset.py
    └── utils.py
```

# Assets

Mirrors the assets in
the physical greenhouse

```
.
└── influx
    ├── __init__.py
    └── influx_controller.py
```

# Influx

Singleton wrapper for
the **influxdb_client** library

```
.
└── sensors
    ├── __init__.py
    ├── humidity.py
    ├── interpreter.py
    ├── light_level.py
    ├── mcp3008.py
    ├── moisture.py
    ├── ndvi.py
    ├── temperature.py
    └── water_level.py
```

# Sensors

Contains classes that represent
the sensors connected and
enable their readout

```python
from collector.sensors.interpreter import Interpreter
from collector.sensors.mcp3008 import MCP3008


class Moisture:
    def __init__(self, adc: MCP3008, channel: int) -> None:
        """Initializes the Moisture sensor.

        Args:
            adc (MCP3008): the analog to digital converter
            channel (int): the channel of the ADC to which the sensor is connected
        """
        self.interpret = Interpreter("moisture").interpret

        self.adc = adc
        self.channel = channel

    def read(self) -> float:
        return self.interpret(self.adc.read(self.channel))

    def stop(self):
        self.adc.close()
```

```python
import json
import numpy as np

from collector.config import CONFIG_PATH
from configparser import ConfigParser


class Interpreter:
    """
    Class that interprets raw values from sensors to meaningful values. Uses a linear interpolation,
    a variable number of points can be used to define the interpolation function.
    """

    def __init__(self, sensor: str, range: tuple = (0, 100)):
        conf = ConfigParser()
        conf.read(CONFIG_PATH)

        self.XP = json.loads(conf[sensor + "_values"]["XP"])
        self.FP = np.linspace(range[0], range[1], len(self.XP))

        # If the first value is greater than the last one, reverse the arrays
        # numpy.interp does not work with decreasing arrays
        if self.XP[0] > self.XP[-1]:
            self.XP = self.XP[::-1]
            self.FP = self.FP[::-1]

    def interpret(self, value: float) -> float:
        return np.interp(value, self.XP, self.FP)
```

```python
from spidev import SpiDev


class MCP3008:
    """
    Analog to digital converter for the Raspberry Pi. Must operate at 3.3V
    Uses the SPI protocol to communicate with the Raspberry Pi.
    """

    def __init__(self, bus = 0, device = 0) -> None:
        self.bus, self.device = bus, device
        self.spi = SpiDev()
        self.open()
        self.spi.max_speed_hz = 1000000  # 1MHz

    def open(self):
        self.spi.open(self.bus, self.device)
        self.spi.max_speed_hz = 1000000  # 1MHz

    def read(self, channel = 0) -> float:
        adc = self.spi.xfer2([1, (8 + channel) << 4, 0])
        data = ((adc[1] & 3) << 8) + adc[2]
        return data / 1023.0 * 3.3 # convert from 10bit value to voltage reading

    def close(self):
        self.spi.close()
```

# Conclusions

Working prototype that continuously logs data and waters the plant if the moisture goes below the optimal value

FUTURE DEVELOPMENT

- Design an FMU to simulate and predict the behaviour of the system
- Integrate more sensors and actuators
- Design supports for the various components
- Expand the scope to a bigger project

# Thank you for your attention