Digital Twin as a Service (DTaaS)

DTaaS Development Team

Table of contents

1. What is DTaaS?

The Digital Twin as a Service (DTaaS) software platform is useful to **Build, Use and Share** digital twins (DTs).

💪🏻 **Build**: DTs are built on DTaaS using reusable DT assets available on the platform.

👩‍🔬👨‍🔧 **Use**: Run your DTs on DTaaS.

🤝 **Share**: Share ready-to-use DTs with other users. It is also possible to share the services offered by one DT with other users.

There is an overview of DTaaS available in the form of slides, video, and feature walkthrough.

1.1 License

This software is owned by The INTO-CPS Association and is available under the INTO-CPS License.

DTaaS software platform uses third-party open-source software. These software components have their own licenses.

2. User

2.1 DTaaS for Users

2.1.1 User Guide

This guide is for users of the DTaaS platform. Please make sure that you have access to a live installation of the DTaaS platform. The easiest is localhost installation scenario.

Please see user-specific Slides and Video to get the conceptual framework behind composable digital twins in the DTaaS platform.

2.1.2 Motivation

How can DT software platforms enable users collaborate to:

- Build digital twins (DTs)
- Use DTs themselves
- Share DTs with other users
- Provide the existing DTs as Service to other users

In addition, how can the DT software platforms:

- Support DT lifecycle
- Scale up rather than scale down (flexible convention over configuration)

2.1.3 Existing Approaches

There are quite a few solutions proposed in the recent past to solve this problem. Some of them are:

- Focus on data from Physical Twins (PTs) to perform analysis, diagnosis, planning etc...
- Share DT assets across the upstream, downstream etc....
- Evaluate different models of PT
- DevOps for Cyber Physical Systems (CPS)
- Scale DT / execution of DT / ensemble of related DTs
- Support for PT product lifecycle

2.1.4 Our Approach

- Support for transition from existing workflows to DT frameworks
- Create DTs from reusable assets
- Enable users to share DT assets
- Offer DTs as a Service
- Integrate the DTs with external software systems
- Separate configurations of independent DT components

2.2 Overview

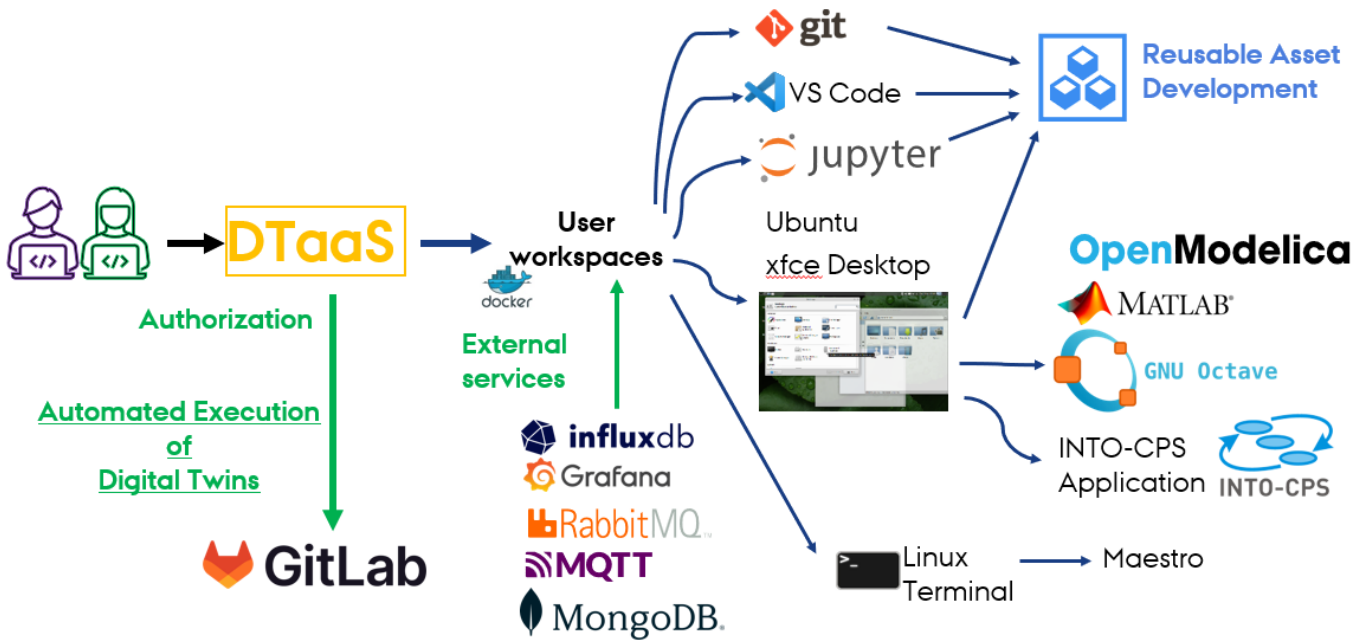
2.2.1 Advantages

The DTaaS software platform provides certain advantages to users:

- Support for different kinds of Digital Twins
- CFD, Simulink, co-simulation, FEM, ROM, ML etc.
- Integrates with other Digital Twin frameworks
- Facilitate availability of Digital Twin as a Service
- Collaboration and reuse
- Private workspaces for verification of reusable assets, trial run DTs
- Cost effectiveness

2.2.2 Software Features

Each installation of DTaaS platform comes with the features highlighted in the following picture.



All the users have dedicated workspaces. These workspaces are dockerized versions of Linux Desktops. The user desktops are isolated so the installations and customizations done in one user workspace do not effect the other user workspaces.

Each user workspace comes with some development tools pre-installed. These tools are directly accessible from web browser. The following tools are available at present:

Tool	Advantage
Jupyter Lab	Provides flexible creation and use of digital twins and their components from web browser. All the native Jupyterlab usecases are supported here.
Jupyter Notebook	Useful for web-based management of their files (library assets)
VS Code in the browser	A popular IDE for software development. Users can develop their digital twin-related assets here.
ungit	An interactive git client. Users can work with git repositories from web browser

In addition, users have access to xfce-based remote desktop via VNC client. The VNC client is available right in the web browser. The xfce supported desktop software can also be run in their workspace.

The DTaaS software platform has some pre-installed services available. The currently available services are:

Service	Advantage
InfluxDB	Time-series database primarly for storing time-series data from physical twins. The digital twins can use an already existing data. Users can also create visualization dashboards for their digital twins.
RabbitMQ	Communication broker for communication between physical and digital twins
Grafana	Visualization dashboards for their digital twins.
MQTT	Lightweight data transfer broker for IoT devices / physical twins feeding data into digital twins.
MongoDB	NoSQL document database for storing metadata of data from physical twins

In addition, the workspaces are connected to the Internet so all the Digital Twins running in the workspace can interact with both the internal and external services.

The users can publish and reuse the digital twin assets available on the platform. In addition, users can run their digital twins and make these live digital twins available as services to their clients. The clients need not be users of the DTaaS software installation.

2.3 DTaaS Website Screenshots

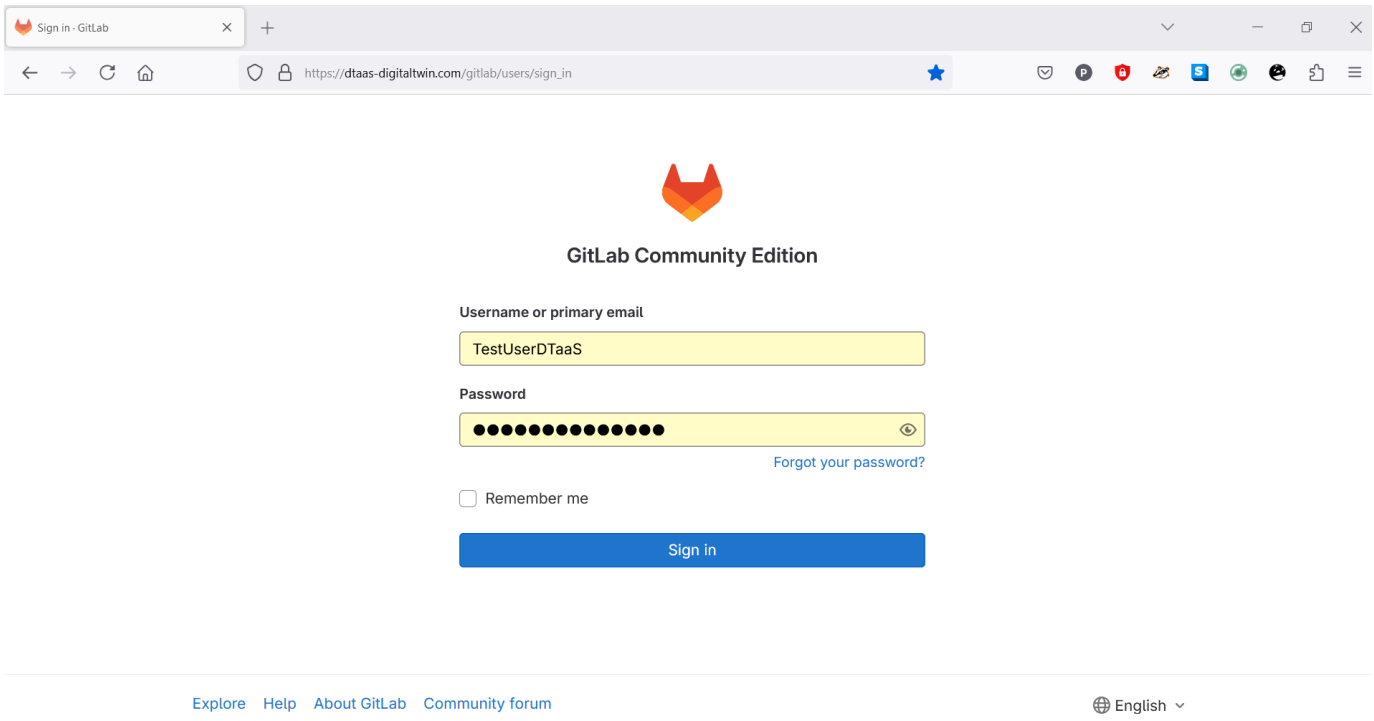
This page contains a screenshot driven preview of the website serving the DTaaS software platform.

2.3.1 Visit the DTaaS Application

Start off by simply visiting the website of the DTaaS instance for which you are a user.

2.3.2 Redirected to Authorization Provider

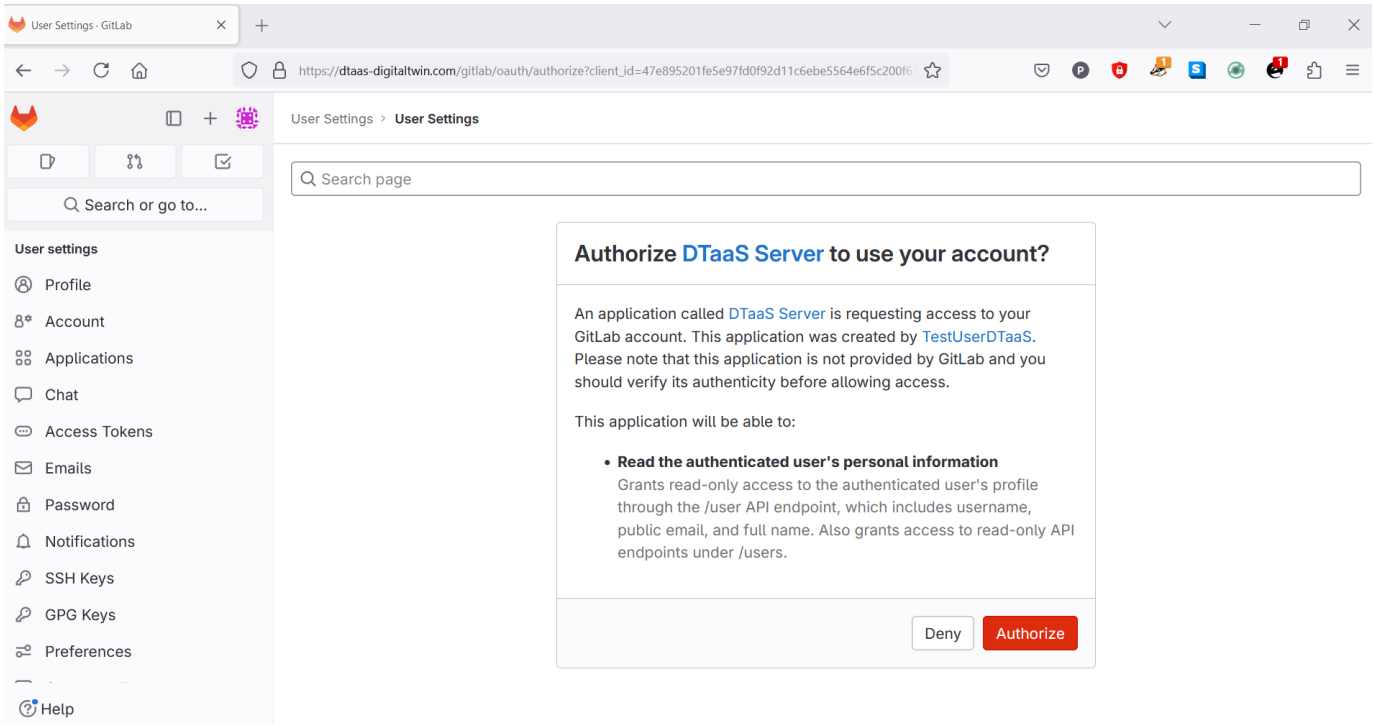
You will be redirected to the Gitlab Authorization for DTaaS.



Enter your email/username and password. If the email ID registered with DTaaS, is the same as your Google Login email ID, you can also opt to sign in using Google.

You will be redirected to the OAuth Application page.

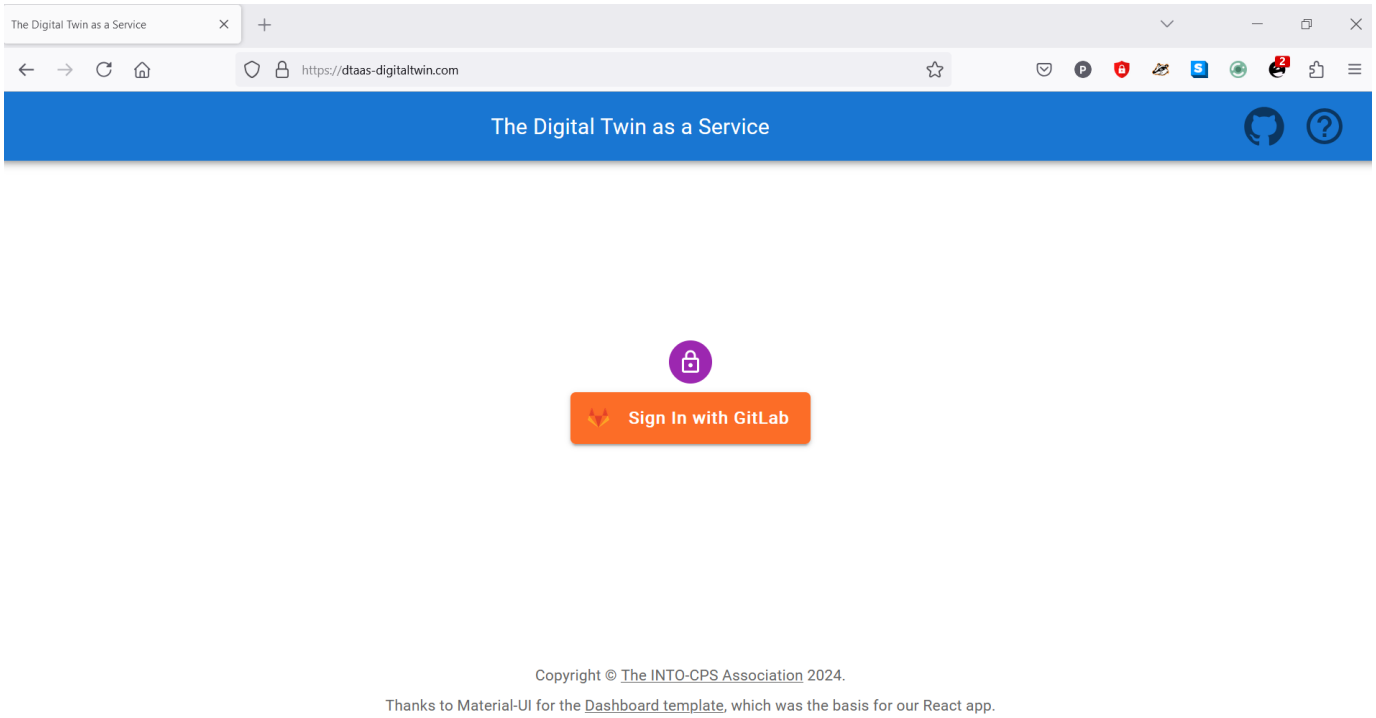
2.3.3 Permit DTaaS Server to Use Gitlab



Click on Authorize to allow the OAuth application to access the information connected to your Gitlab account. This is a necessary step.

You are now logged into the DTaaS server. You will be redirected to the login page of the DTaaS website.

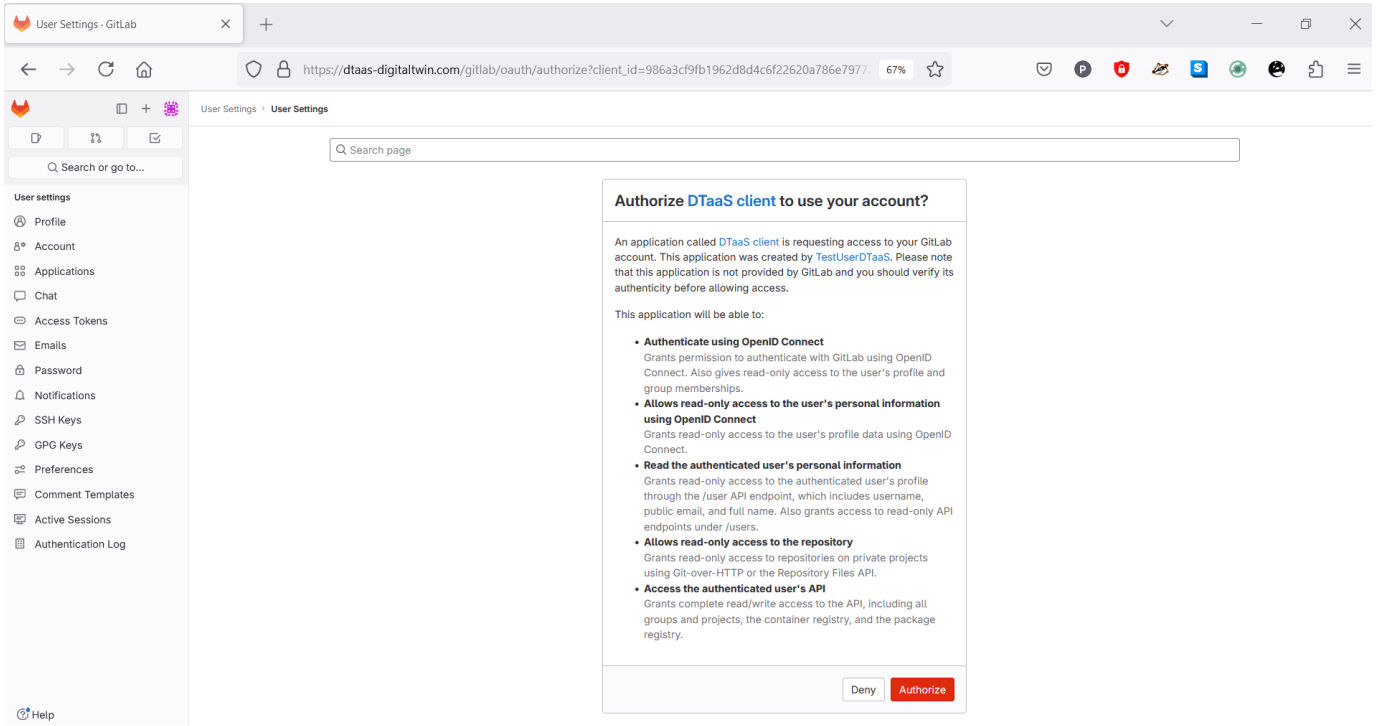
The DTaaS website uses an additional layer of security - the third-party authorization protocol known as OAuth. This protocol provides secure access to a DTaaS installation if users have a working active accounts at the selected OAuth service provider. This also uses Gitlab as OAuth provider.



You can see the Gitlab signin button. A click on this button takes you to Gitlab instance providing authorization for DTaaS. You will not need to sign in to Gitlab again, unless you have explicitly logged out of your Gitlab account.

2.3.4 Permit DTaaS Website to Use Gitlab

The DTaaS website needs your permission to use your Gitlab account for authorization. Click on **Authorize** button.

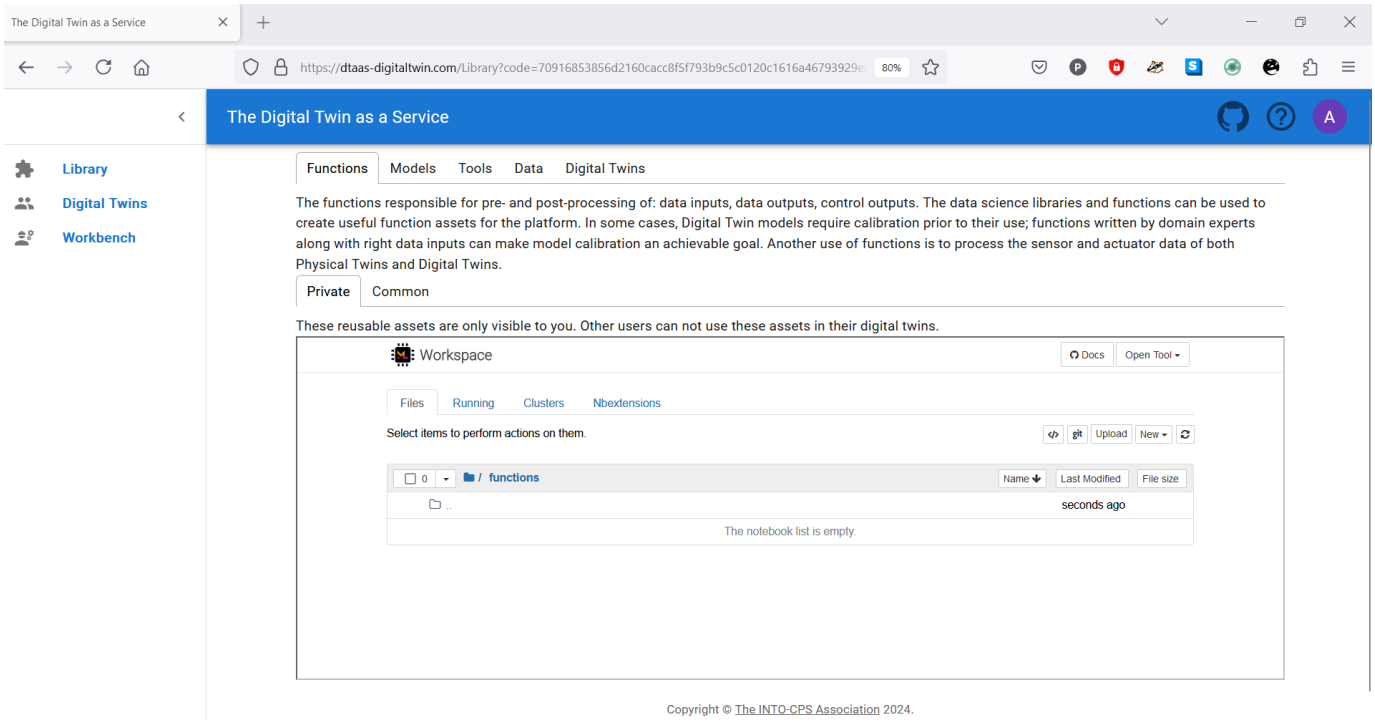


After successful authorization, you will be redirected to the **Library** page of the DTaaS website.

There are two icons on the top-right of the webpage. The hyperlink on **question mark icon** redirects to help page while the hyperlink on **github icon** redirects to github code repository.

2.3.5 Overview of menu items

The menu is hidden by default. Only the icons of menu items are visible. You can click on the ☰ icon in the top-left corner of the page to see the menu.



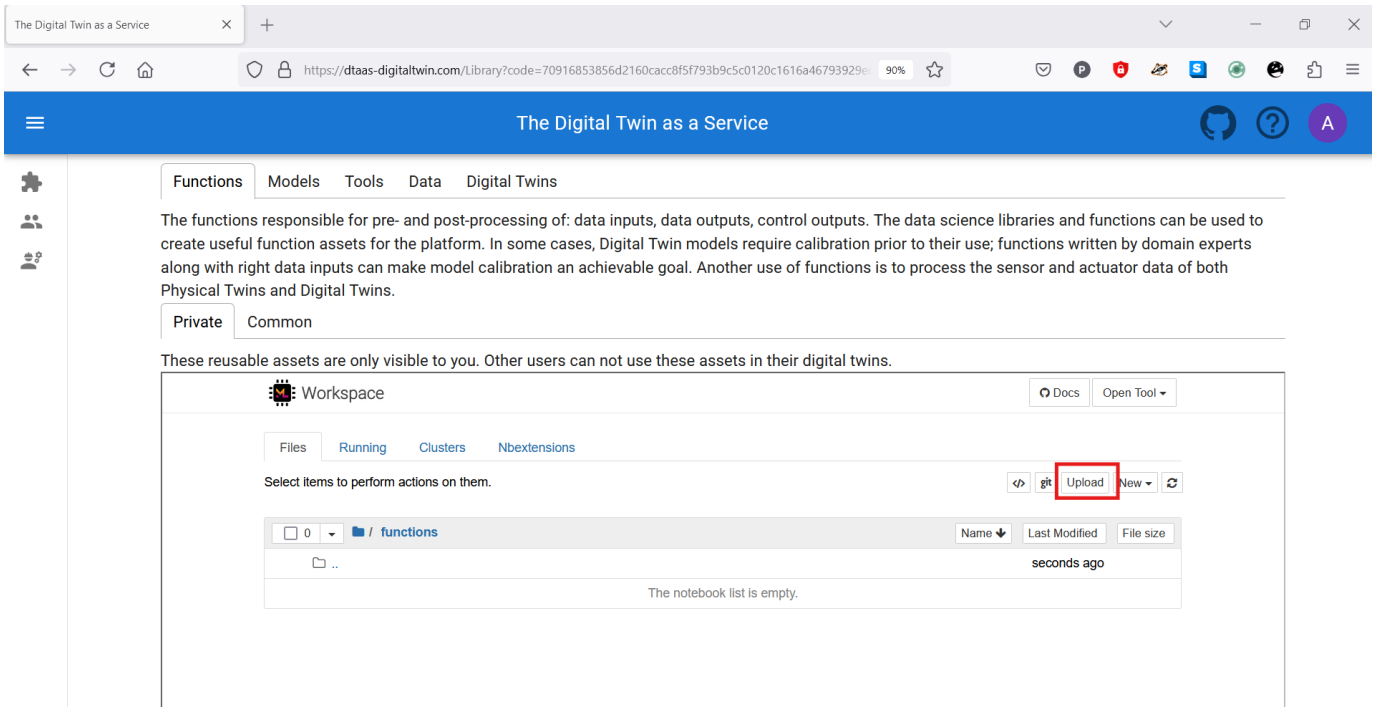
There are three menu items:

Library: for management of reusable library assets. You can upload, download, create and modify new files on this page.

Digital Twins: for management of digital twins. You are presented with the Jupyter Lab page from which you can run the digital twins.

Workbench: Not all digital twins can be managed within Jupyter Lab. You have more tools at your disposal on this page.

2.3.6 Library tabs and their help text



You can see five tabs each corresponding to one type of digital twin assets. Each tab provides help text to guide users on the type of an asset.

Functions

The functions responsible for pre- and post-processing of: data inputs, data outputs, control outputs. The data science libraries and functions can be used to create useful function assets for the platform. In some cases, Digital Twin models require calibration prior to their use; functions written by domain experts along with right data inputs can make model calibration an achievable goal. Another use of functions is to process the sensor and actuator data of both Physical Twins and Digital Twins.

Data

The data sources and sinks available to a digital twins. Typical examples of data sources are sensor measurements from Physical Twins, and test data provided by manufacturers for calibration of models. Typical examples of data sinks are visualization software, external users and data storage services. There exist special outputs such as events, and commands which are akin to control outputs from a Digital Twin. These control outputs usually go to Physical Twins, but they can also go to another Digital Twin.

Models

The model assets are used to describe different aspects of Physical Twins and their environment, at different levels of abstraction. Therefore, it is possible to have multiple models for the same Physical Twin. For example, a flexible robot used in a car production plant may have structural model(s) which will be useful in tracking the wear and tear of parts. The same robot can have a behavioural model(s) describing the safety guarantees provided by the robot manufacturer. The same robot can also have a functional model(s) describing the part manufacturing capabilities of the robot.

Tools

The software tool assets are software used to create, evaluate and analyze models. These tools are executed on top of a computing platforms, i.e., an operating system, or virtual machines like Java virtual machine, or inside docker containers. The tools tend to be platform specific, making them less reusable than models. A tool can be packaged to run on a local or distributed virtual machine environments thus allowing selection of most suitable execution environment for a Digital Twin. Most models require tools to evaluate them in the context of data inputs. There exist cases where executable packages are run as binaries in a computing environment. Each of these packages are a pre-packaged combination of models and tools put together to create a ready to use Digital Twins.

Digital Twins

These are ready to use digital twins created by one or more users. These digital twins can be reconfigured later for specific use cases.

There are two sub-tabs, namely **private** and **common**. The library assets in private category are visible only to the logged in user while the library assets in common category are available to all the users.

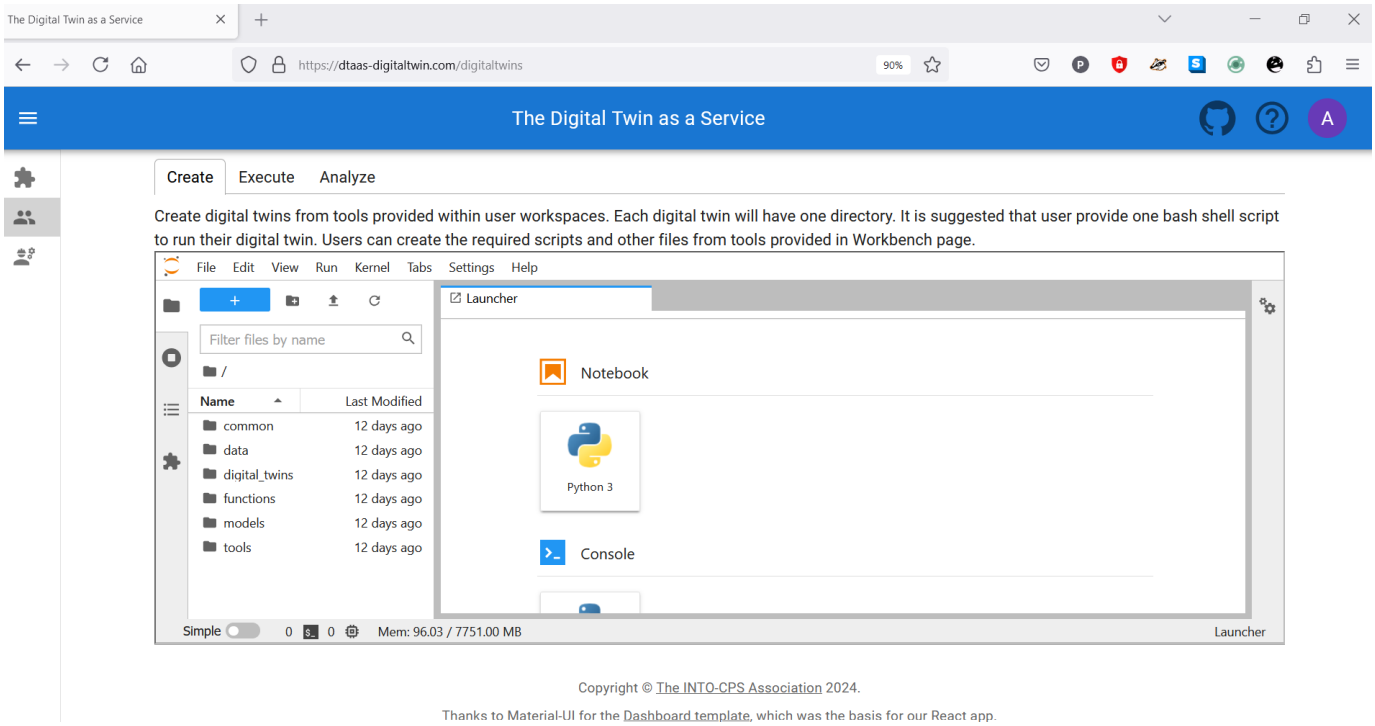
Further explanation on the placement of reusable assets within each type and the underlying directory structure on the server is in the assets page

Note

You can upload assets (files) using the **upload** button.

The file manager is based on Jupyter notebook and all the tasks you can perform in the Jupyter Notebook can be undertaken here.

2.3.7 Digital Twins page



The digital twins page has three tabs and the central pane opens Jupyter lab. There are three tabs with helpful instructions on the suggested tasks you can undertake in the **Create - Execute - Analyze** life cycle phases of digital twin. You can see more explanation on the life cycle phases of digital twin.

Create

Create digital twins from tools provided within user workspaces. Each digital twin will have one directory. It is suggested that user provide one bash shell script to run their digital twin. Users can create the required scripts and other files from tools provided in Workbench page.

Execute

Digital twins are executed from within user workspaces. The given bash script gets executed from digital twin directory. Terminal-based digital twins can be executed from VSCode and graphical digital twins can be executed from VNC GUI. The results of execution can be placed in the data directory.

Analyze

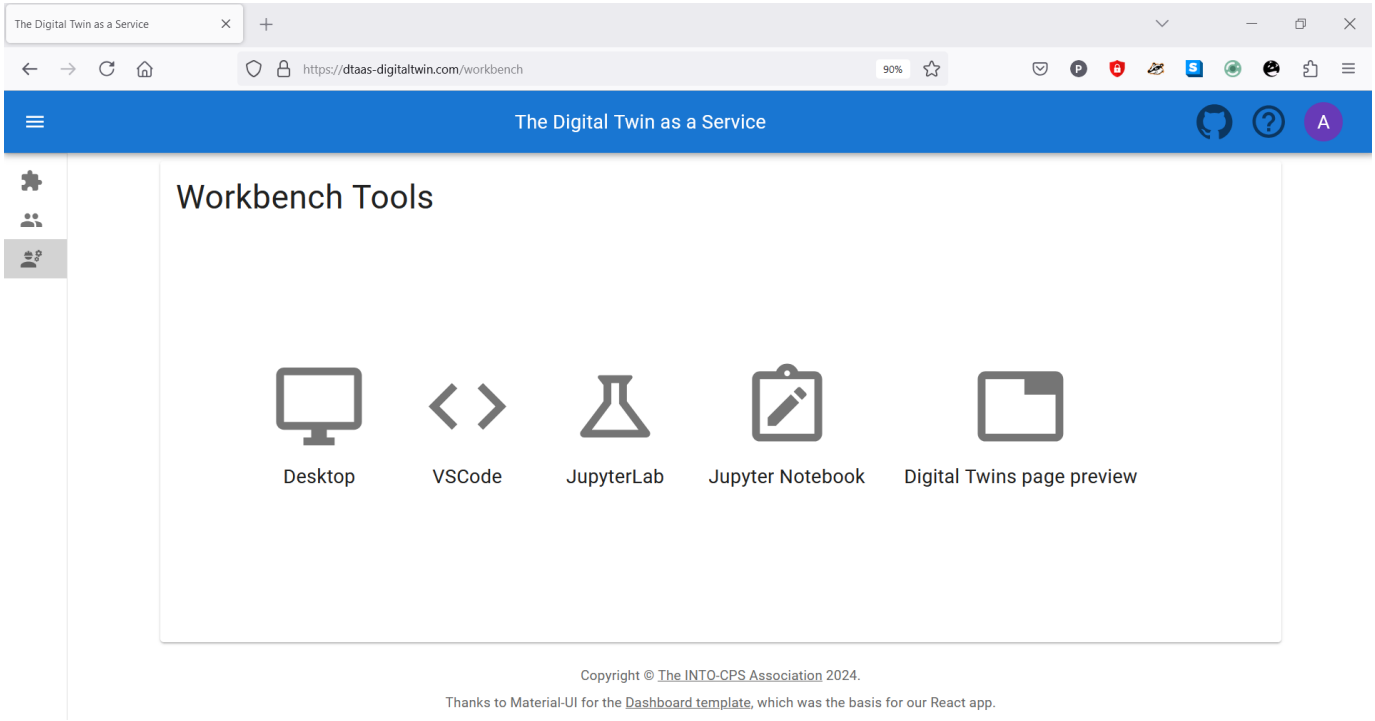
The analysis of digital twins requires running of digital twin script from user workspace. The execution results placed within data directory are processed by analysis scripts and results are placed back in the data directory. These scripts can either be executed from VSCode and graphical results or can be executed from VNC GUI. The analysis of digital twins requires running of digital twin script from user workspace. The execution results placed within data directory are processed by analysis scripts and results are placed back in the data directory. These scripts can either be executed from VSCode and graphical results or can be executed from VNC GUI.

The reusable assets (files) seen in the file manager are available in the Jupyter Lab. In addition, there is a git plugin installed in the Jupyter Lab using which you can link your files with the external git repositories.

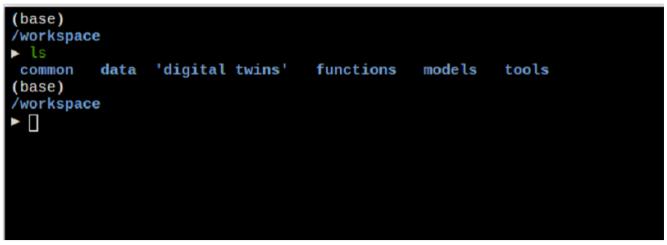
2.3.8 Workbench

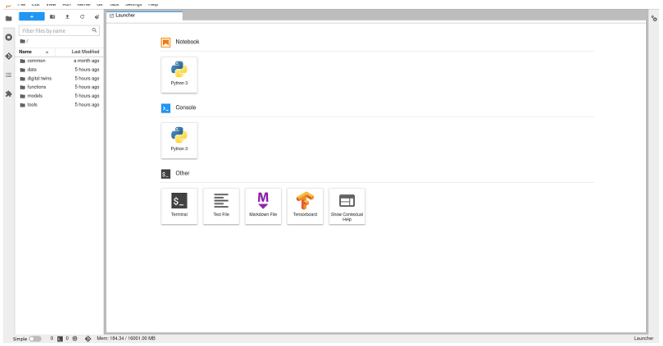
The **workbench** page provides links to four integrated tools.

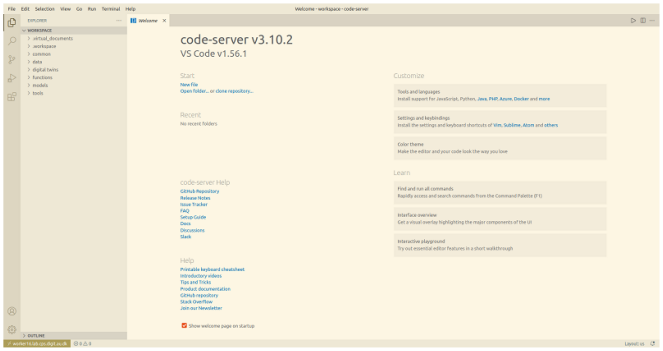
- Desktop
- VS Code
- Jupyter Lab
- Jupyter Notebook

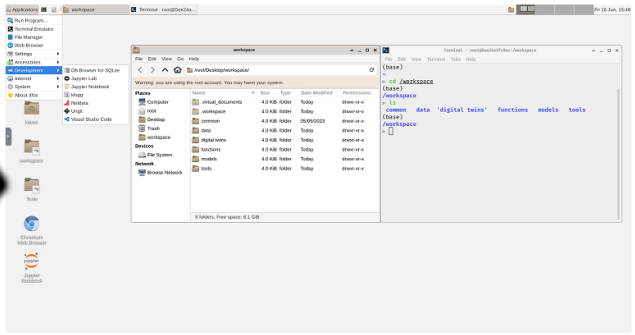


The hyperlinks open in new browser tab. The screenshots of pages opened in new browser are:



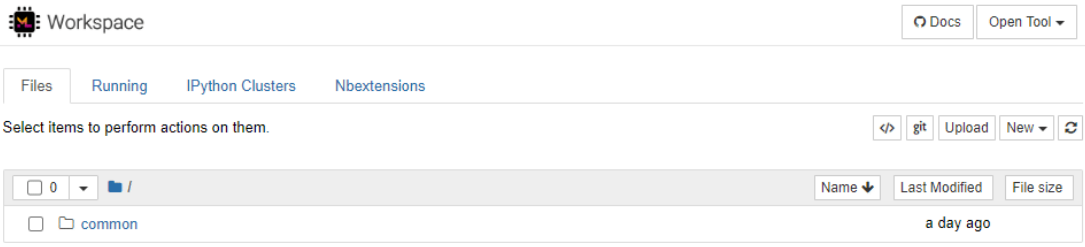






Terminal

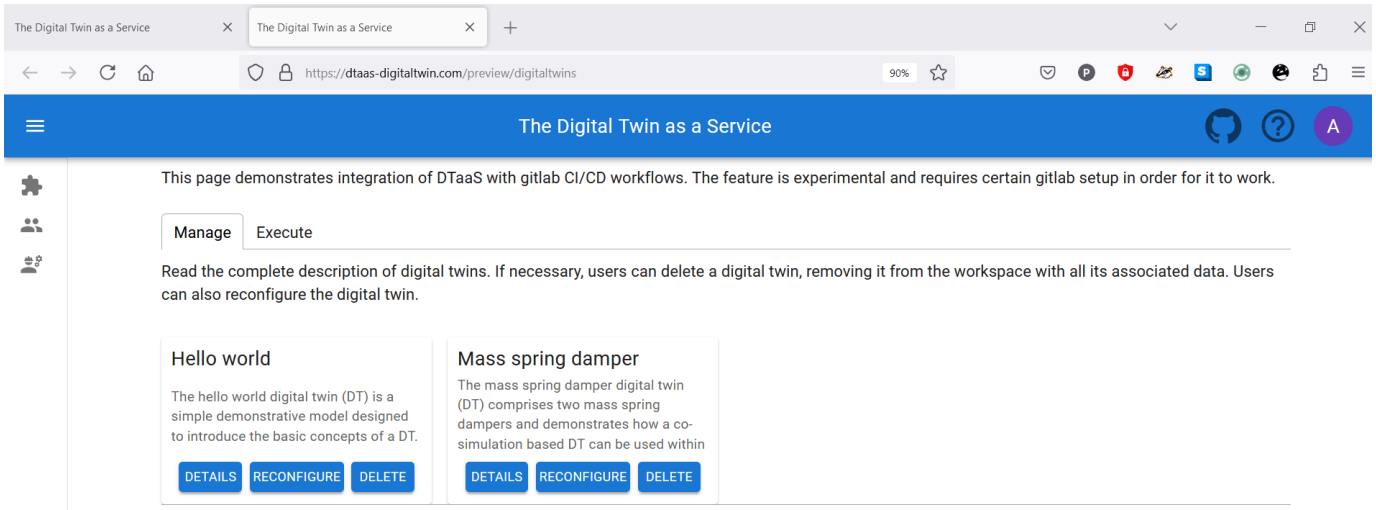
The Terminal hyperlink does not exist on workbench page. If you want terminal. Please use the tools dropdown in the Jupyter Notebook.

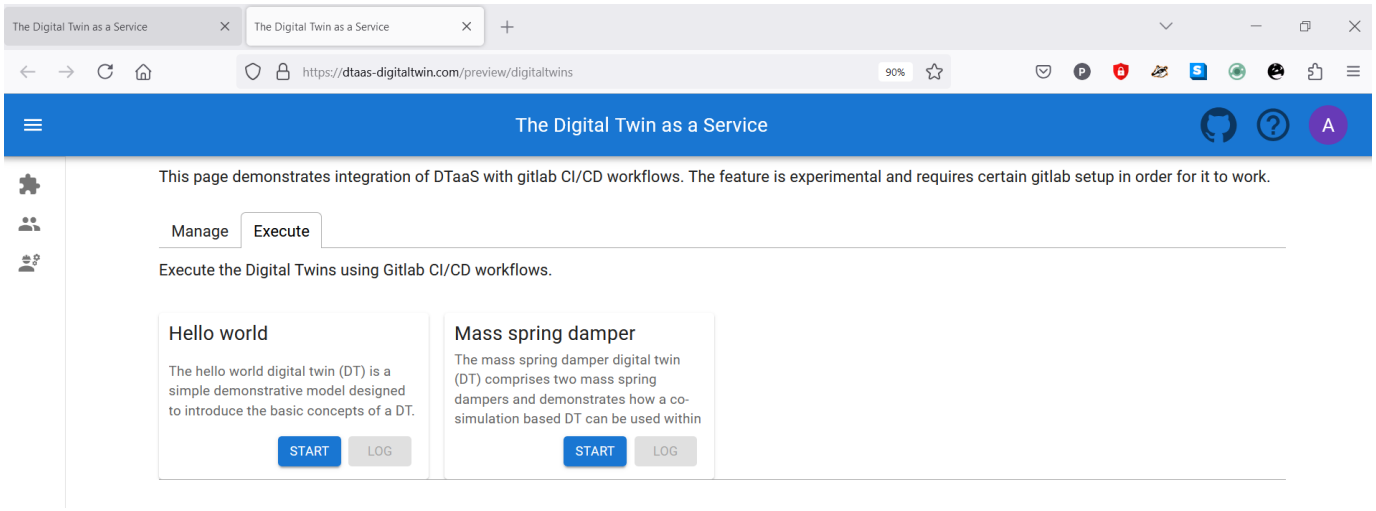


2.3.9 Digital Twins Preview Page

There is a fifth link on the Workbench page. It is **Digital Twins Preview Page**. It is an experimental feature at the moment. Clicking on the link opens a new page that is similar to the Digital Twins page.

There are two tabs, namely **Manage** and **Execute**.





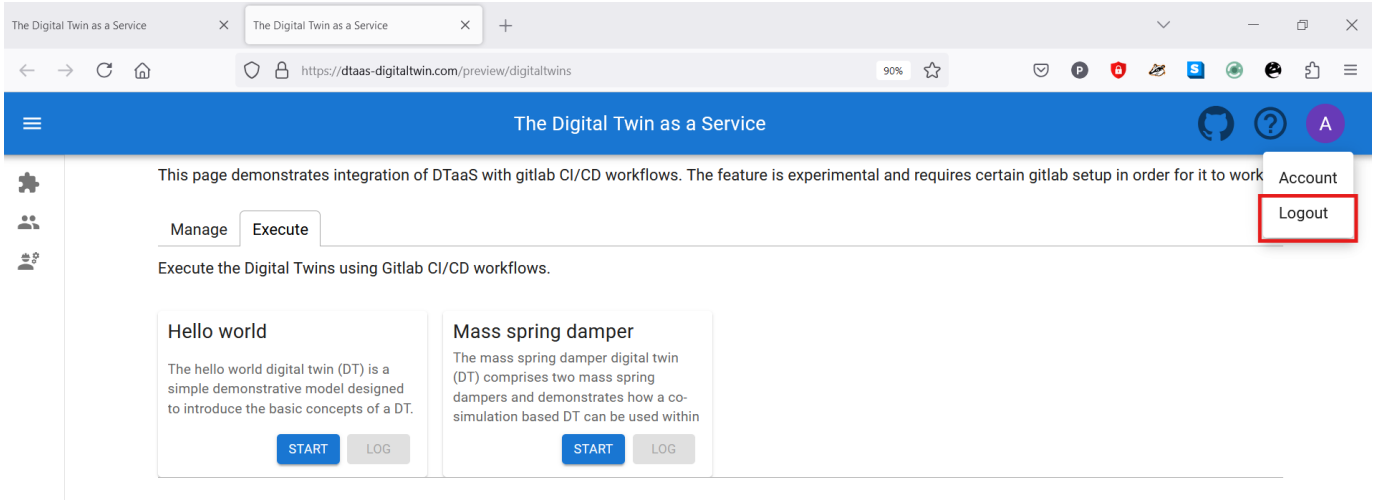
Manage

Read the complete description of digital twins. If necessary, users can delete a digital twin, removing it from the workspace with all its associated data. Users can also reconfigure the digital twin.

Execute

Execute the Digital Twins using Gitlab CI/CD workflows.

2.3.10 Finally logout



You have to close the browser in order to completely exit the DTaaS software platform.

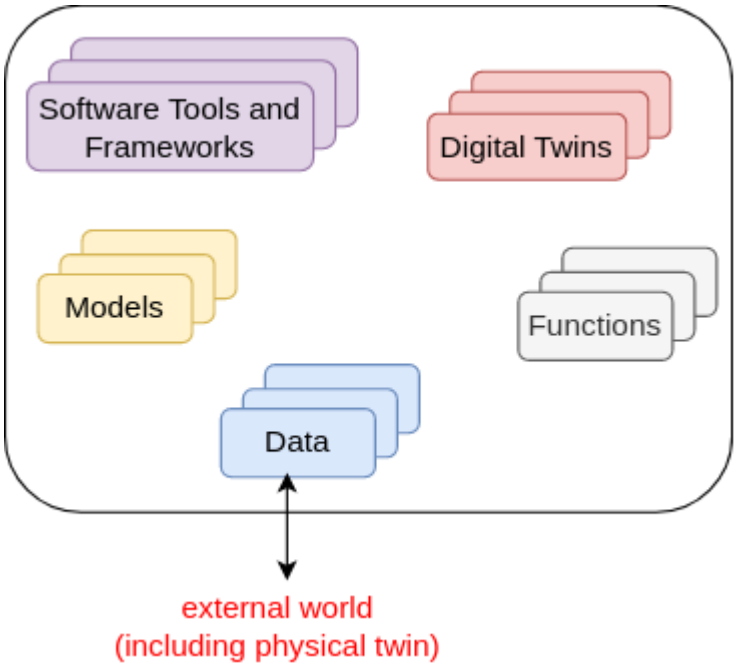
2.4 Reusable Assets

2.4.1 Reusable Assets

The reusability of digital twin assets makes it easy for users to work with the digital twins. The reusability of assets is a fundamental feature of the platform.

Kinds of Reusable Assets

The DTaaS software categorizes all the reusable library assets into six categories:



DATA

The data sources and sinks available to a digital twins. Typical examples of data sources are sensor measurements from Physical Twins, and test data provided by manufacturers for calibration of models. Typical examples of data sinks are visualization software, external users and data storage services. There exist special outputs such as events, and commands which are akin to control outputs from a Digital Twin. These control outputs usually go to Physical Twins, but they can also go to another Digital Twin.

MODELS

The model assets are used to describe different aspects of Physical Twins and their environment, at different levels of abstraction. Therefore, it is possible to have multiple models for the same Physical Twin. For example, a flexible robot used in a car production plant may have structural model(s) which will be useful in tracking the wear and tear of parts. The same robot can have a behavioural model(s) describing the safety guarantees provided by the robot manufacturer. The same robot can also have a functional model(s) describing the part manufacturing capabilities of the robot.

TOOLS

The software tool assets are software used to create, evaluate and analyze models. These tools are executed on top of a computing platforms, i.e., an operating system, or virtual machines like Java virtual machine, or inside docker containers. The tools tend to be platform specific, making them less reusable than models. A tool can be packaged to run on a local or distributed virtual machine environments thus allowing selection of most suitable execution environment for a Digital Twin. Most models require tools to evaluate them in the context of data inputs. There exist cases where executable packages are run as binaries in a computing environment. Each of these packages are a pre-packaged combination of models and tools put together to create a ready to use Digital Twins.

FUNCTIONS

The functions responsible for pre- and post-processing of: data inputs, data outputs, control outputs. The data science libraries and functions can be used to create useful function assets for the platform. In some cases, Digital Twin models require calibration prior to their use; functions written by domain experts along with right data inputs can make model calibration an achievable goal. Another use of functions is to process the sensor and actuator data of both Physical Twins and Digital Twins.

DIGITAL TWINS

These are ready to use digital twins created by one or more users. These digital twins can be reconfigured later for specific use cases.

File System Structure

Each user has their assets put into five different directories named above. In addition, there will also be common library assets that all users have access to. A simplified example of the structure is as follows:

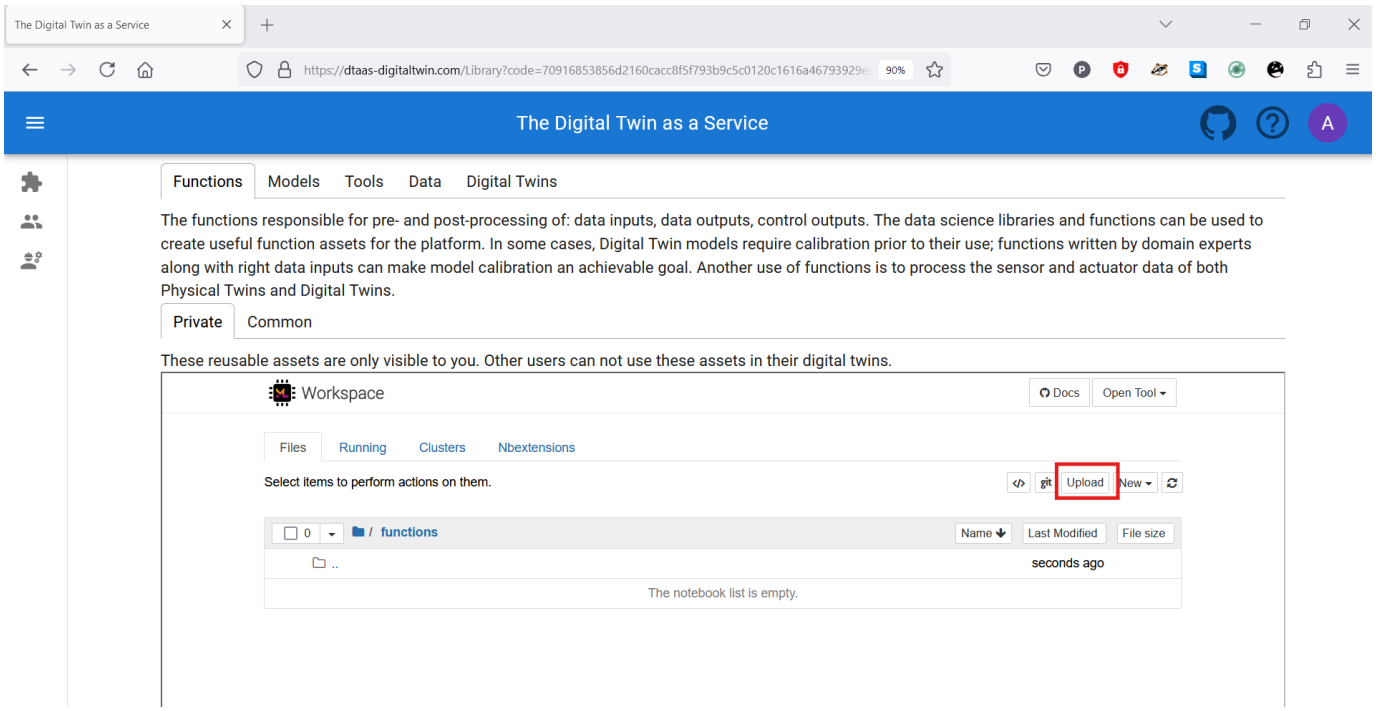
```
workspace/
  data/
    data1/ (ex: sensor)
      filename (ex: sensor.csv)
      README.md
    data2/ (ex: turbine)
      README.md (remote source; no local file)
    ...
  digital_twins/
    digital_twin-1/ (ex: incubator)
      config (yaml and json)
      README.md (usage instructions)
      description.md (short summary of digital twin)
      lifecycle/ (directory containing lifecycle scripts)
    digital_twin-2/ (ex: mass spring damper)
      config (yaml and json)
      README.md (usage instructions)
      description.md (short summary of digital twin)
      lifecycle/ (directory containing lifecycle scripts)
    digital_twin-3/ (ex: model swap)
      config (yaml and json)
      README.md (usage instructions)
      description.md (short summary of digital twin)
      lifecycle/ (directory containing lifecycle scripts)
    ...
  functions/
    function1/ (ex: graphs)
      filename (ex: graphs.py)
      README.md
    function2/ (ex: statistics)
      filename (ex: statistics.py)
      README.md
    ...
  models/
    model1/ (ex: spring)
      filename (ex: spring.fmu)
      README.md
    model2/ (ex: building)
      filename (ex: building.skp)
      README.md
    model3/ (ex: rabbitmq)
      filename (ex: rabbitmq.fmu)
      README.md
    ...
  tools/
    tool1/ (ex: maestro)
      filename (ex: maestro.jar)
      README.md
    ...
  common/
    data/
    functions/
    models/
    tools/
```

The DTaaS is agnostic to the format of your assets. The only requirement is that they are files which can be uploaded on the Library page. Any directories can be compressed as one file and uploaded. You can decompress the file into a directory from a Terminal or xfce Desktop available on the Workbench page.

A recommended file system structure for storing assets is also available in DTaaS examples.

Upload Assets

Users can upload assets into their workspace using Library page of the website.



You can go into a directory and click on the **upload** button to upload a file or a directory into your workspace. This asset is then available in all the workbench tools you can use. You can also create new assets on the page by clicking on **new** drop down menu. This is a simple web interface which allows you to create text-based files. You need to upload other files using **upload** button.

The user workbench has the following services:

- Jupyter Notebook and Lab
- VS Code
- XFCE Desktop Environment available via VNC
- Terminal

Users can also bring their DT assets into user workspaces from outside using any of the above mentioned services. The developers using *git* repositories can clone from and push to remote git servers. Users can also use widely used file transfer protocols such as FTP, and SCP to bring the required DT assets into their workspaces.

2.4.2 Library Microservice

The lib microservice is responsible for handling and serving the contents of library assets of the DTaaS platform. It provides API endpoints for clients to query, and fetch these assets.

This document provides instructions for using the library microservice.

Please see assets for a suggested storage conventions of your library assets.

Once the assets are stored in the library, they become available in user workspace.

Application Programming Interface (API)

The lib microservice application provides services at two end points:

GraphQL API Endpoint: `http://foo.com/lib`

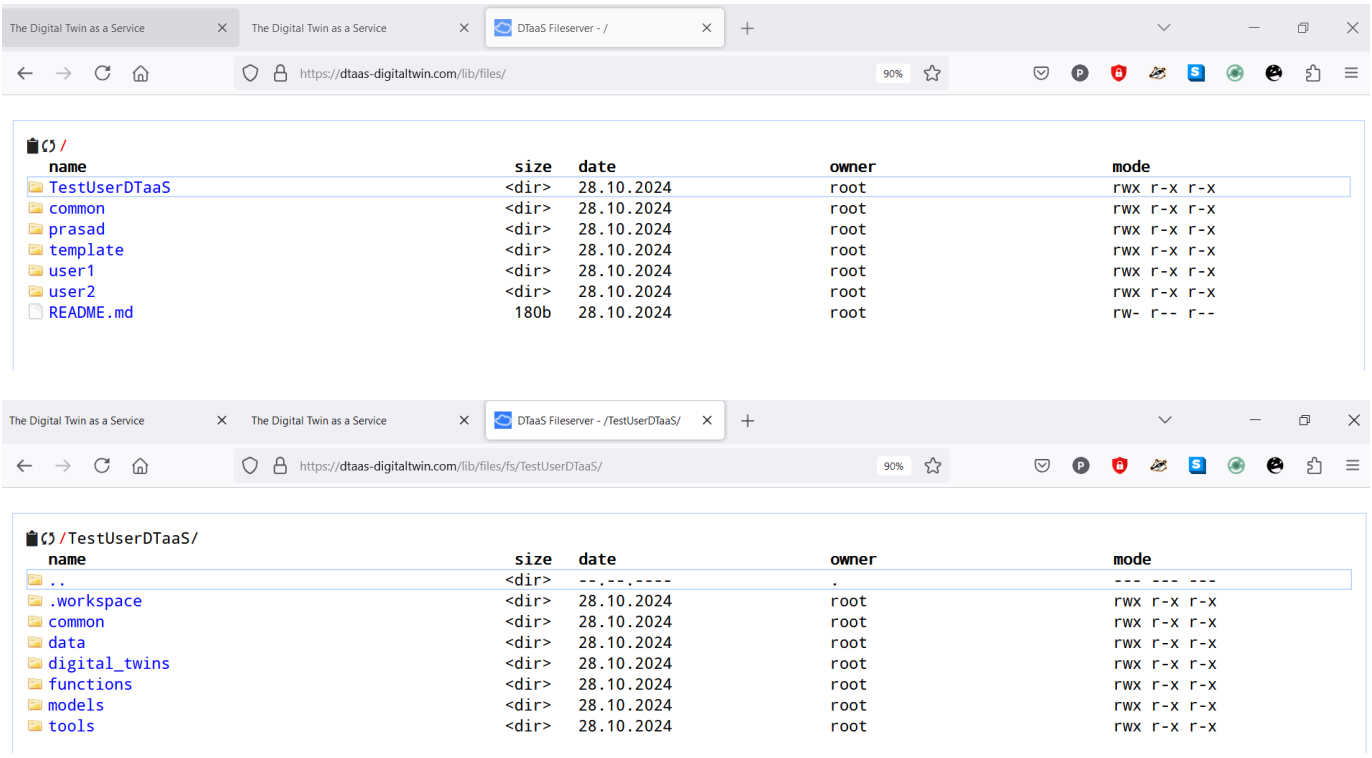
HTTP Endpoint: `http://foo.com/lib/files`

HTTP PROTOCOL

Endpoint: `localhost:PORT/lib/files`

This option needs to be enabled with `-H http.json` flag. The regular file upload and download options become available.

Here are sample screenshots.



GRAPHQL PROTOCOL

Endpoint: `localhost:PORT/lib`

The `http://foo.com/lib` URL opens a graphql playground.

You can check the query schema and try sample queries here. The graphql queries need to be sent as HTTP POST requests and get responses.

The library microservice services two API calls:

- Provide a list of contents for a directory
- Fetch a file from the available files

The API calls are accepted over GraphQL and HTTP API end points. The format of the accepted queries are:

PROVIDE LIST OF CONTENTS FOR A DIRECTORY

To retrieve a list of files in a directory, use the following GraphQL query.

Replace `path` with the desired directory path.

send requests to: https://foo.com/lib

GraphQL query for list of contents

```
query {
  listDirectory(path: "user1") {
    repository {
      tree {
        blobs {
          edges {
            node {
              name
              type
            }
          }
        }
        trees {
          edges {
            node {
              name
              type
            }
          }
        }
      }
    }
  }
}
```

GraphQL response for list of contents


```
{
  "data": {
    "listDirectory": {
      "repository": {
        "tree": {
          "blobs": {
            "edges": []
          },
          "trees": {
            "edges": [
              {
                "node": {
                  "name": "common",
                  "type": "tree"
                }
              },
              {
                "node": {
                  "name": "data",
                  "type": "tree"
                }
              },
              {
                "node": {
                  "name": "digital twins",
                  "type": "tree"
                }
              },
              {
                "node": {
                  "name": "functions",
                  "type": "tree"
                }
              },
              {
                "node": {
                  "name": "models",
                  "type": "tree"
                }
              },
              {
                "node": {
                  "name": "tools",
                  "type": "tree"
                }
              }
            ]
          }
        }
      }
    }
  }
}
```


HTTP request for list of contents


```
POST /lib HTTP/1.1
Host: foo.com
Content-Type: application/json
Content-Length: 388

{
  "query":"query {\n  listDirectory(path: \"user1\") {\n    repository {\n      tree {\n        blobs {\n          edges {\n            node {\n              name\n              type\n            }\n          }\n        }\n        trees {\n          edges {\n            node {\n              name\n              type\n            }\n          }\n        }\n      }\n    }\n  }\n}"
}
```


HTTP response for list of contents


```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Connection: close
Content-Length: 306
Content-Type: application/json; charset=utf-8
Date: Tue, 26 Sep 2023 20:26:49 GMT
X-Powered-By: Express
{"data":{"listDirectory":{"repository":{"tree":{"blobs":{"edges":[]},"trees":{"edges":[{"node":{"name":"data","type":"tree"}},{"node":{"name":"digital twins","type":"tree"}},{"node":{"name":"functions","type":"tree"}},{"node":{"name":"models","type":"tree"}},{"node":{"name":"tools","type":"tree"}}]}}}}}}
```


FETCH A FILE FROM THE AVAILABLE FILES

This query receives directory path and send the file contents to user in response.

To check this query, create a file `files/user2/data/welcome.txt` with content of `hello world`.

GraphQL query for fetch a file

```
query {
  readFile(path: "user2/data/sample.txt") {
    repository {
      blobs {
        nodes {
          name
          rawBlob
          rawTextBlob
        }
      }
    }
  }
}
```

GraphQL response for fetch a file

```
{
  "data": {
    "readFile": {
      "repository": {
        "blobs": {
          "nodes": [
            {
              "name": "sample.txt",
              "rawBlob": "hello world",
              "rawTextBlob": "hello world"
            }
          ]
        }
      }
    }
  }
}
```

HTTP request for fetch a file

```
POST /lib HTTP/1.1
Host: foo.com
Content-Type: application/json
Content-Length: 217
{
  "query":"query {\n  readFile(path: \"user2/data/welcome.txt\") {\n    repository {\n      blobs {\n        nodes {\n          name\n          rawBlob\n
rawTextBlob\n        }\n      }\n    }\n  }\n}"
}
```

HTTP response for fetch a file

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Connection: close
Content-Length: 134
Content-Type: application/json; charset=utf-8
Date: Wed, 27 Sep 2023 09:17:18 GMT
X-Powered-By: Express
{"data":{"readFile":{"repository":{"blobs":{"nodes":[{"name":"welcome.txt","rawBlob":"hello world","rawTextBlob":"hello world"}]}}}}}
```

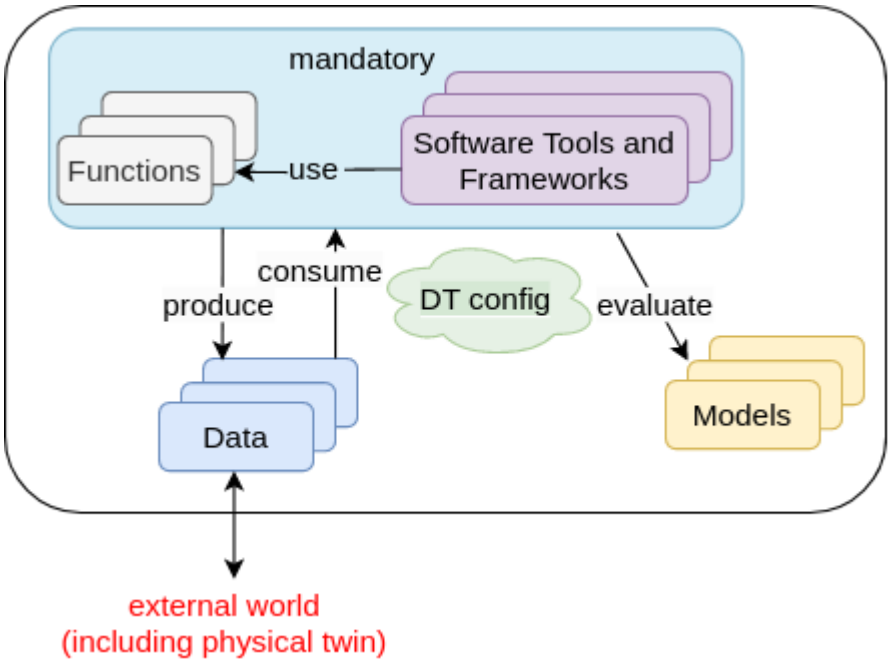
The *path* refers to the file path to look at: For example, *user1* looks at files of **user1**; *user1/functions* looks at contents of *functions/* directory.

2.5 Digital Twins

2.5.1 Create a Digital Twin

The first step in digital twin creation is to use the available assets in your workspace. If you have assets / files in your computer that need to be available in the DTaaS workspace, then please follow the instructions provided in library assets.

There are dependencies among the library assets. These dependencies are shown below.



A digital twin can only be created by linking the assets in a meaningful way. This relationship can be expressed using a mathematical equation:

where D denotes data, M denotes models, F denotes functions, T denotes tools, denotes DT configuration and is a symbolic notation for a digital twin itself. The expression denotes composition of DT from D,M,T and F assets. The indicates zero or one more instances of an asset and indicates one or more instances of an asset.

The DT configuration specifies the relevant assets to use, the potential parameters to be set for these assets. If a DT needs to use RabbitMQ, InfluxDB like services supported by the platform, the DT configuration needs to have access credentials for these services.

This kind of generic DT definition is based on the DT examples seen in the wild. You are at liberty to deviate from this definition of DT. The only requirement is the ability to run the DT from either commandline or desktop.

> **Tip**
>
> If you are stepping into the world of Digital Twins, you might not have distinct digital twin assets. You are likely to have one directory of everything in which you run your digital twin. In such a case we recommend that you upload this monolithic digital twin into **digital_twin/your_digital_twin_name** directory.

Example

The Examples repository contains a co-simulation setup for mass spring damper. This example illustrates the potential of using co-simulation for digital twins.

The file system contents for this example are:

```
workspace/
  data/
    mass-spring-damper
        input/
        output/

  digital_twins/
    mass-spring-damper/
      cosim.json
      time.json
      lifecycle/
        analyze
        clean
        evolve
        execute
        save
        terminate
      README.md

  functions/
  models/
    MassSpringDamper1.fmu
    MassSpringDamper2.fmu

  tools/
  common/
    data/
    functions/
    models/
    tools/
        maestro-2.3.0-jar-with-dependencies.jar
```

The `workspace/data/mass-spring-damper/` contains `input` and `output` data for the mass-spring-damper digital twin.

The two FMU models needed for this digital twin are in `models/` directory.

The co-simulation digital twin needs Maestro co-simulation orchestrator. Since this is a reusable asset for all the co-simulation based DTs, the tool has been placed in `common/tools/` directory.

The actual digital twin configuration is specified in `digital twins/mass-spring-damper` directory. The co-simulation configuration is specified in two json files, namely `cosim.json` and `time.json`. A small explanation of digital twin for its users can be placed in `digital twins/mass-spring-damper/README.md`.

The launch program for this digital twin is in `digital twins/mass-spring-damper/lifecycle/execute`. This launch program runs the co-simulation digital twin. The co-simulation runs till completion and then ends. The programs in `digital twins/mass-spring-damper/lifecycle` are responsible for lifecycle management of this digital twin. The lifecycle page provides more explanation on these programs.

> **Execution of a Digital Twin**
>
> A frequent question arises on the run time characteristics of a digital twin. The natural intuition is to say that a digital twin must operate as long as its physical twin is in operation. **If a digital twin runs for a finite time and then ends, can it be called a digital twin? The answer is a resounding YES**. The Industry 4.0 usecases seen among SMEs have digital twins that run for a finite time. These digital twins are often run at the discretion of the user.

You can run this digital twin by,

1. Go to Workbench tools page of the DTaaS website and open VNC Desktop. This opens a new tab in your browser
2. A page with VNC Desktop and a connect button comes up. Click on Connect. You are now connected to the Linux Desktop of your workspace.
3. Open a Terminal (black rectangular icon in the top left region of your tab) and type the following commands.
4. Download the example files by following the instructions given on examples overview.
5. Go to the digital twin directory and run

```
cd /workspace/examples/digital_twins/mass-spring-damper
lifecycle/execute
```

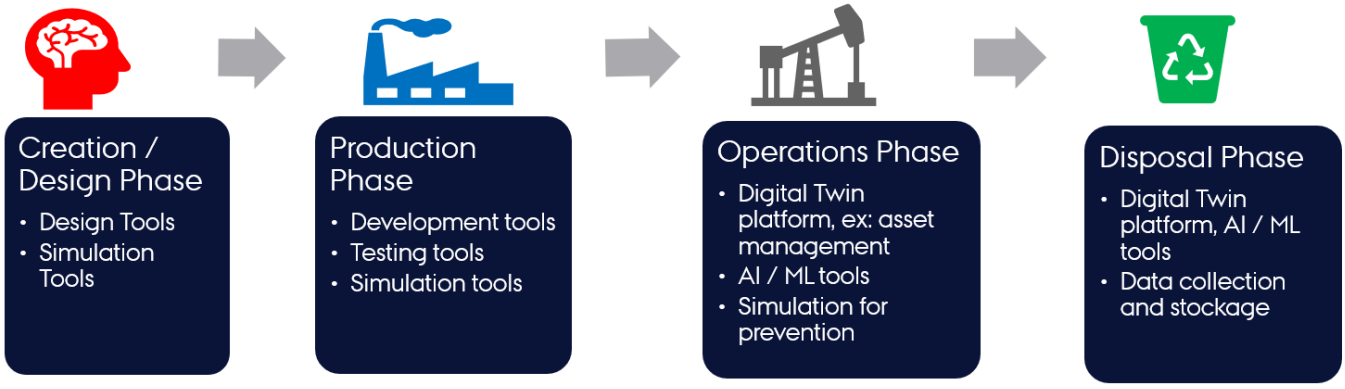
The last command executes the mass-spring-damper digital twin and stores the co-simulation output in `data/mass-spring-damper/output`.

2.5.2 ♻ Digital Twin Lifecycle

The physical products in the real world have product lifecycle. A simplified four-stage product life is illustrated here.

A digital twin tracking the physical products (twins) need to track and evolve in conjunction with the corresponding physical twin.

The possible activities undertaken in each lifecycle phases are illustrated in the figure.



(Ref: Minerva, R, Lee, GM and Crespi, N (2020) Digital Twin in the IoT context: a survey on technical features, scenarios and architectural models. Proceedings of the IEEE, 108 (10). pp. 1785-1824. ISSN 0018-9219.)

Lifecycle Phases

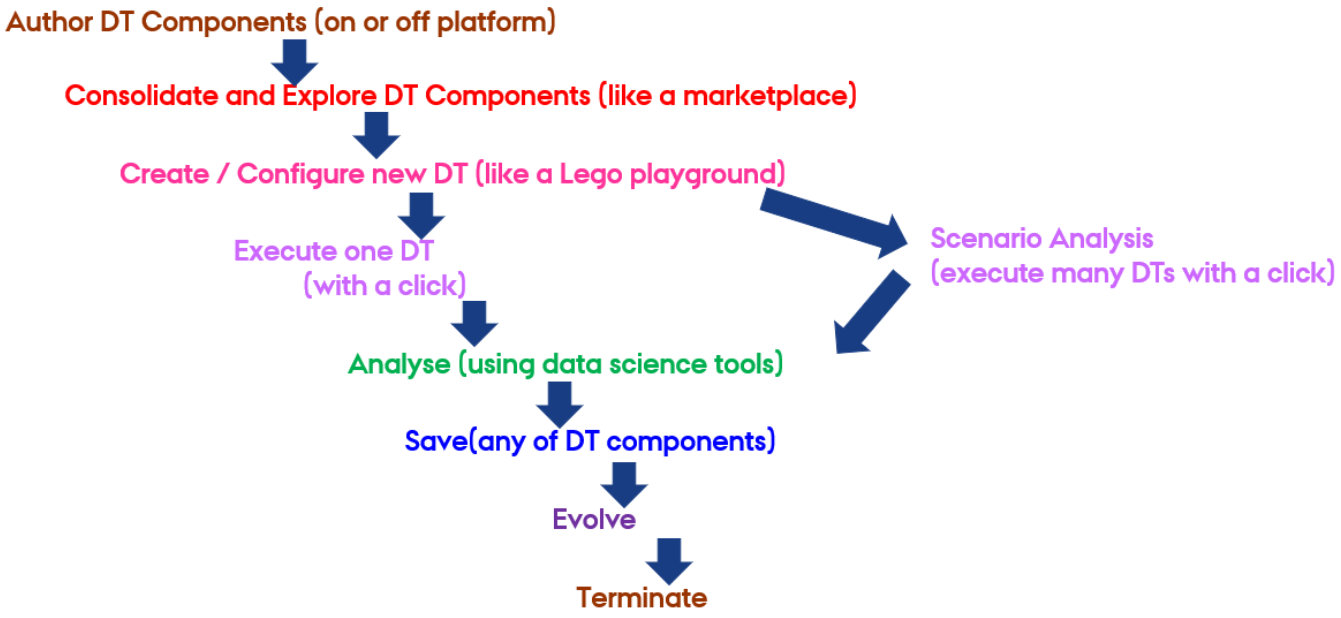
The four phase lifecycle has been extended to a lifecycle with eight phases. The new phase names and the typical activities undertaken in each phase are outlined in this section.

A DT lifecycle consists of **explore, create, execute, save, analyse, evolve** and **terminate** phases.

Phase	Main Activities
explore	selection of suitable assets based on the user needs and checking their compatibility for the purposes of creating a DT.
create	specification of DT configuration. If DT already exists, there is no creation phase at the time of reuse.
execute	automated / manual execution of a DT based on its configuration. The DT configuration must checked before starting the execution phase.
analyse	checking the outputs of a DT and making a decision. The outputs can be text files, or visual dashboards.
evolve	reconfigure DT primarily based on analysis.
save	involves saving the state of DT to enable future recovery.
terminate	stop the execution of DT.

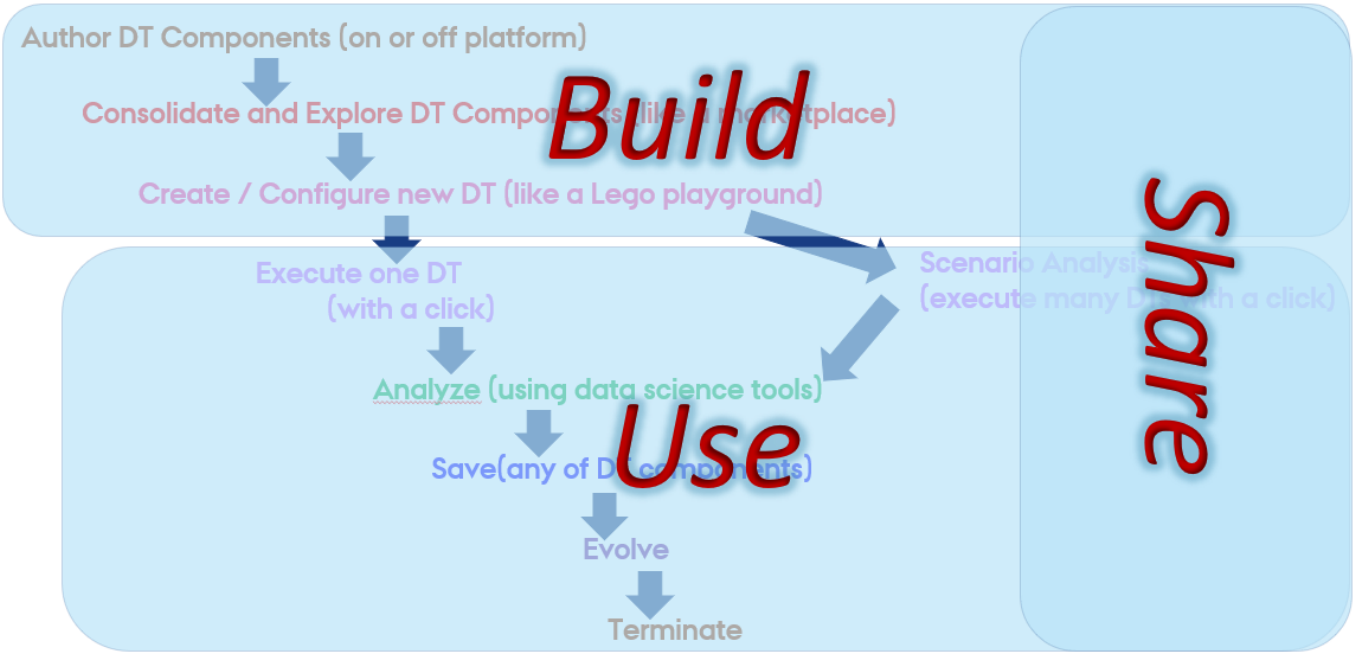
A digital twin faithfully tracking the physical twin lifecycle will have to support all the phases. It is also possible for digital twin engineers to add more phases to digital they are developing. Thus it is important for the DTaaS software platform needs to accommodate needs of different DTs.

A potential linear representation of the tasks undertaken in a digital twin lifecycle are shown here.



Again this is only a one possible pathway. Users are at liberty to alter the sequence of steps.

It is possible to map the lifecycle phases identified so far with the **Build-Use-Share** approach of the DTaaS software platform.



Even though not mandatory, having a matching coding structure makes it easy to for users to create and manage their DTs within the DTaaS. It is recommended to have the following structure:

```
workspace/
  digital_twins/
    digital-twin-1/
      lifecycle/
        analyze
        clean
        evolve
        execute
        save
        terminate
```

A dedicated program exists for each phase of DT lifecycle. Each program can be as simple as a script that launches other programs or sends messages to a live digital twin.

The recommended way to implement lifecycle phases within DTaaS is to create scripts. These scripts can be as simple as shell scripts.

Example Lifecycle Scripts

Here are the example programs / scripts to manage three phases in the lifecycle of **mass-spring-damper DT**.

```
#!/bin/bash
mkdir -p /workspace/data/mass-spring-damper/output
#cd ..
java -jar /workspace/common/tools/maestro-2.3.0-jar-with-dependencies.jar \
    import -output /workspace/data/mass-spring-damper/output \
    --dump-intermediate sg1 cosim.json time.json -i -vi FMI2 \
    output-dir>debug.log 2>&1
```

The execute phases uses the DT configuration, FMU models and Maestro tool to execute the digital twin. The script also stores the output of cosimulation in `/workspace/data/mass-spring-damper/output`.

It is possible for a DT not to support a specific lifecycle phase. This intention can be specified with an empty script and a helpful message if deemed necessary.

```
#!/bin/bash
printf "operation is not supported on this digital twin"
```

The lifecycle programs can call other programs in the code base. In the case of `lifecycle/terminate` program, it is calling another script to do the necessary job.

```
#!/bin/bash
lifecycle/clean
```

2.5.3 DevOps Preview

Digital Twin File Structure in Gitlab

We use GitLab as a file store for performing DevOps on Digital Twins. The user interface page is a front-end for this gitlab-backed file storage.

Each DTaaS installation comes with an integrated Gitlab. There must be a gitlab group named **dtaas** and a gitlab repository for each user where repository name matches the username. For example, if there are two users, namely *user1* and *user2* on a DTaaS installation, then the following repositories must exist on the linked Gitlab installation.

```
https://foo.com/gitlab/dtaas/common.git
https://foo.com/gitlab/dtaas/user1.git
https://foo.com/gitlab/dtaas/user2.git
```

Each user repository must also have a specific structure. The required structure is as follows.

```
<username>/
├── common/
├── data/
├── digital_twins/
├── functions/
├── models/
├── tools/
├── .gitlab-ci.yml
└── README.md
```

This file structure follows the same pattern user sees on the existing **Library** page.

DIGITAL TWIN STRUCTURE

The `digital_twins` folder contains DTs that have been pre-built by one or more users. The intention is that they should be sufficiently flexible to be reconfigured as required for specific use cases.

Let us look at an example of such a configuration. The dtaas/user1 repository on gitlab.com contains the `digital_twins` directory with a `hello_world` example. Its file structure looks like this:

```
hello_world/
├── lifecycle/ (at least one lifecycle script)
│   ├── clean
│   ├── create
│   ├── execute
│   └── terminate
├── .gitlab-ci.yml (Gitlab DevOps config for executing lifecycle scripts)
└── description.md (optional but is recommended)
└── README.md (optional but is recommended)
```

The `lifecycle` directory here contains four files - `clean`, `create`, `execute` and `terminate`, which are simple BASH scripts. These correspond to stages in a digital twin's lifecycle. Further explanation of digital twin is available on lifecycle stages.

Digital Twins and DevOps

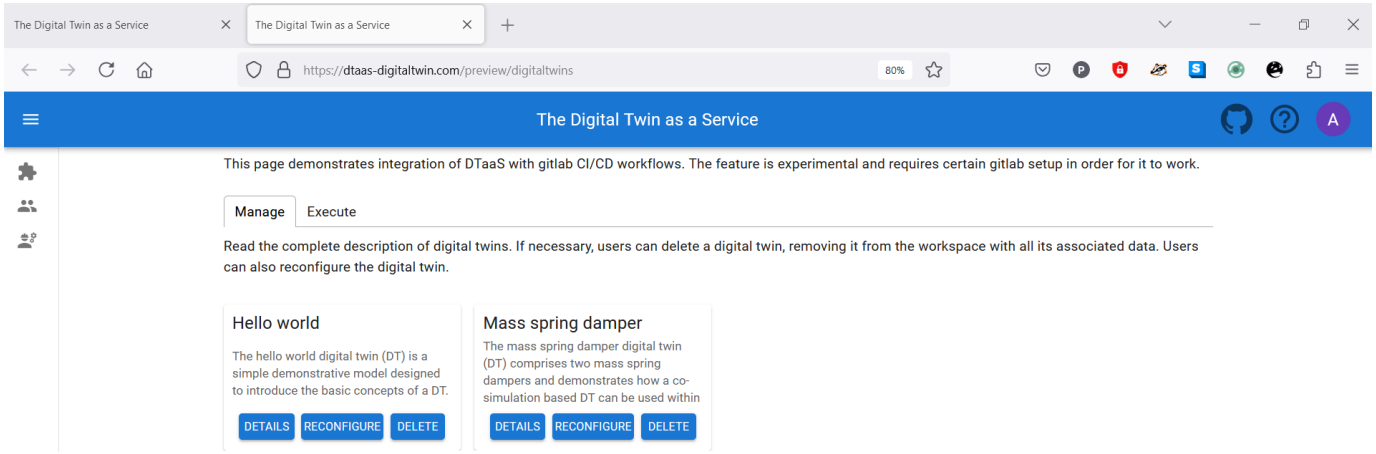
DevOps has been a well established software development practice. We are bringing out an experimental feature of integration DevOps in the DTaaS.

> **Warning**
>
> This feature is under documented. Please expect some instability in this release. However, we are working to mature the feature and improve the documentation.

This feature requires specific installation setup.

1. Integrated gitlab installation
2. A valid gitlab repository for the logged in user. Please see an example repository. You can clone this repository and customize to your needs.
3. A linked Gitlab Runner to the user gitlab repository.

MANAGE TAB

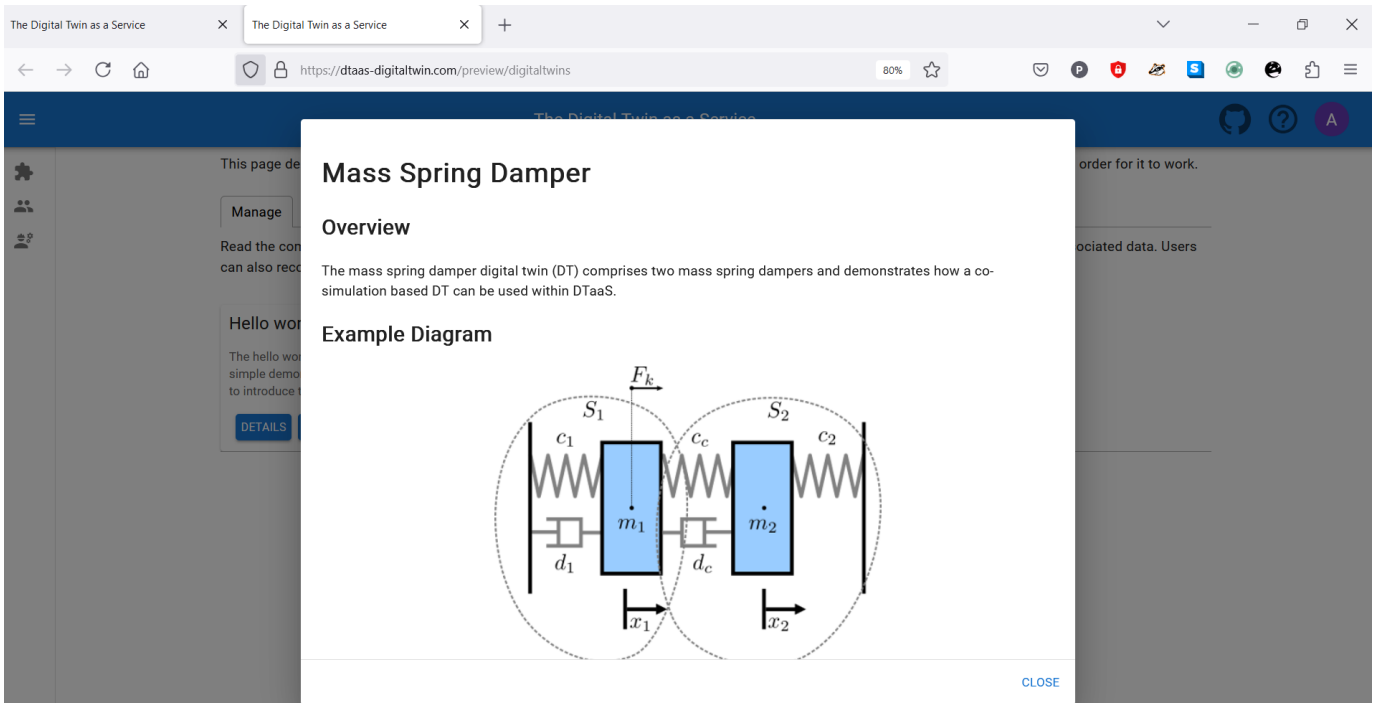


The manage tab allows for different operations on a digital twin:

- Checking the details (**Details** button)
- Delete (**Delete** button)
- Modify / Reconfigure (**Reconfigure** button)

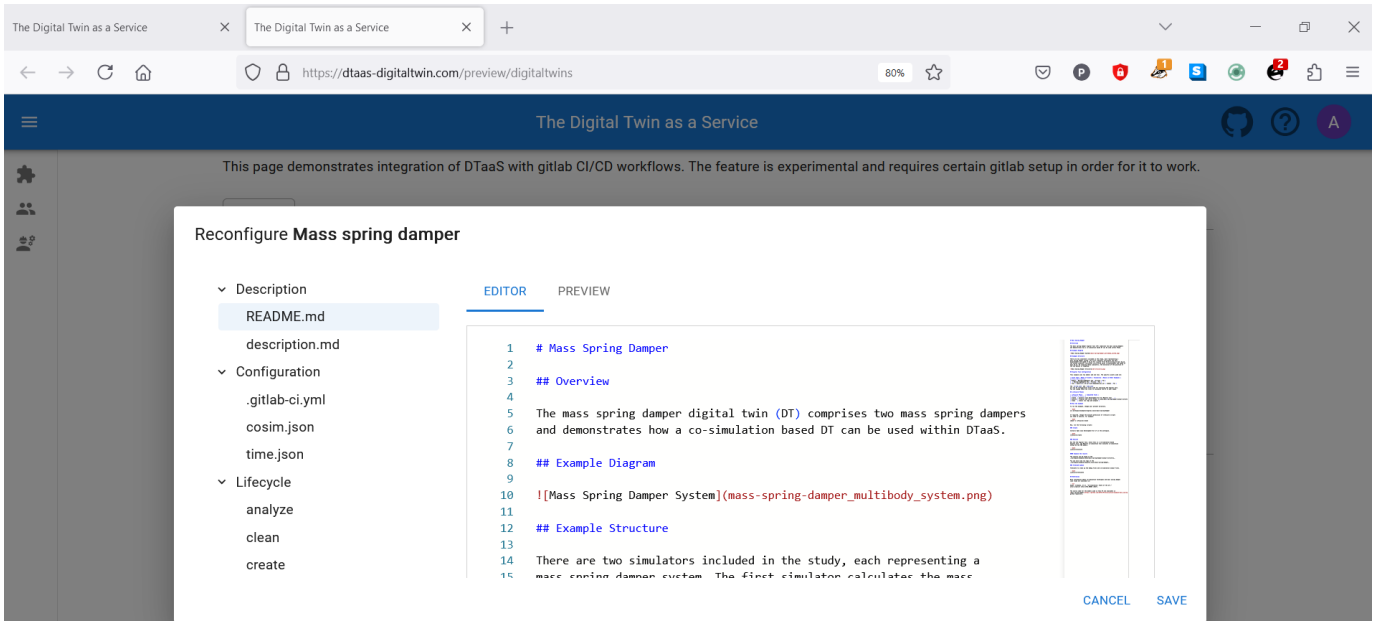
A digital twin placed in the DTaaS has a certain recommended structure. Please see the assets pag for an explanation and this example.

The information page shown using the Details button, shows the README.md information stored inside the digital twin directory.

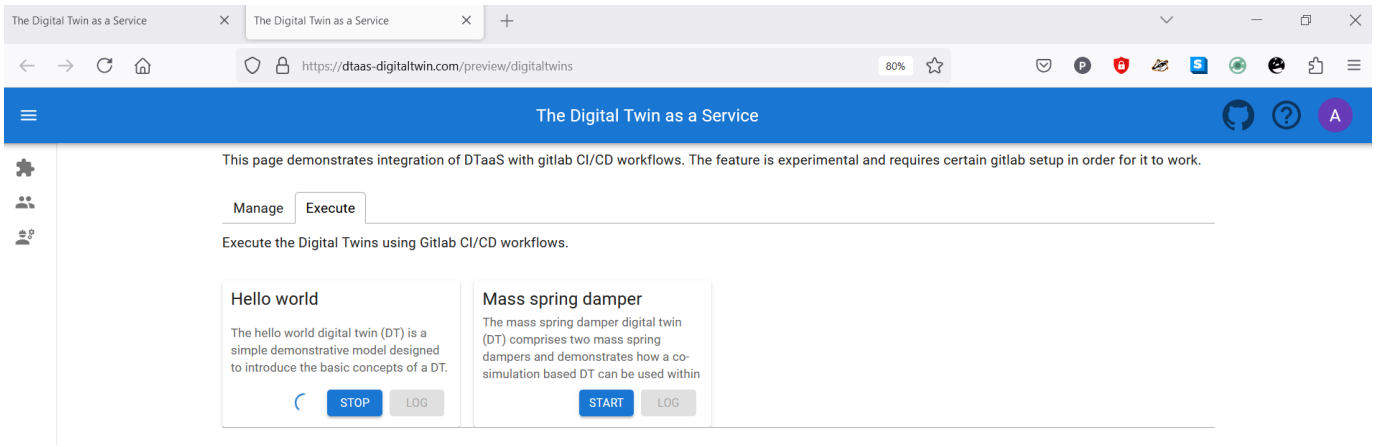


A reconfigure button opens an editor and shows all the files corresponding to a digital twin. All of these files can be updated. These files correspond to three categories.

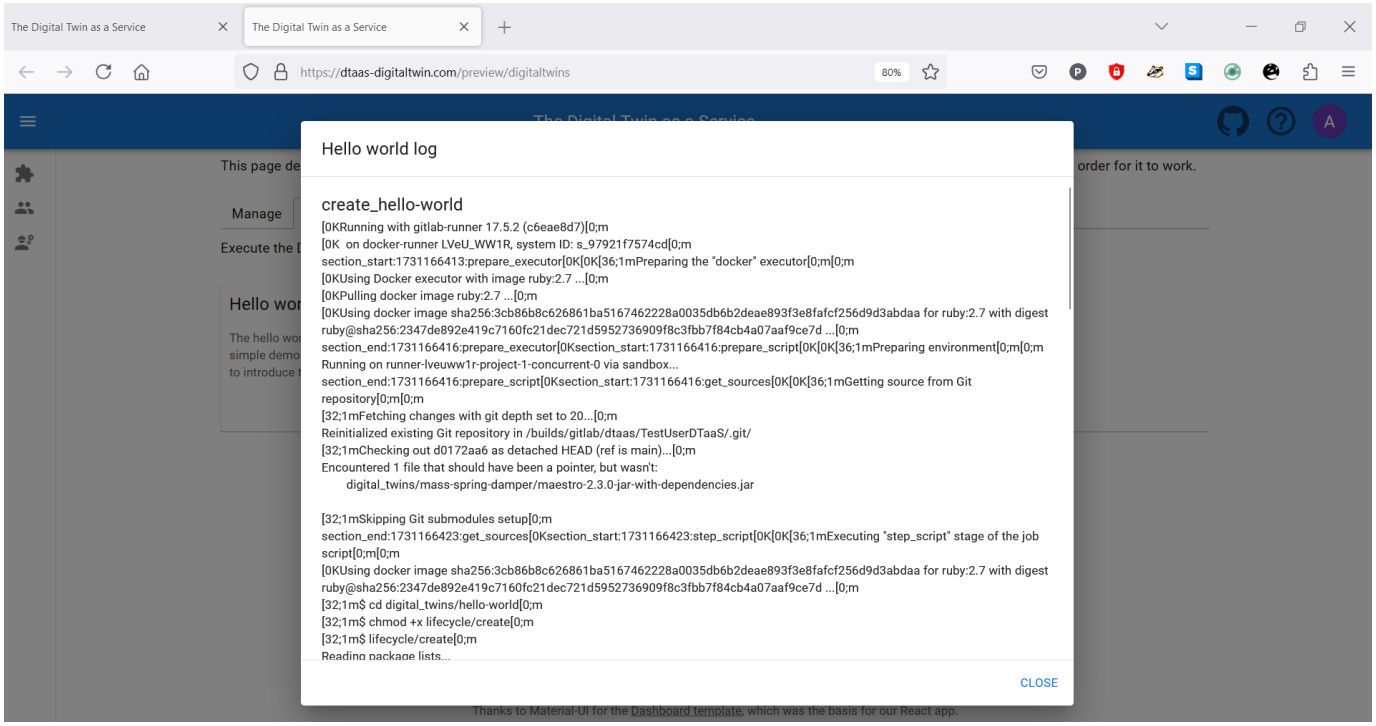
- **Description**
- **Configuration**
- **Lifecycle**



EXECUTE TAB



The execute tabs shows the possibility of executing multiple digital twins. Once an execution of digital twin is complete, you can see the execution log as well.



2.6 Working with Gitlab

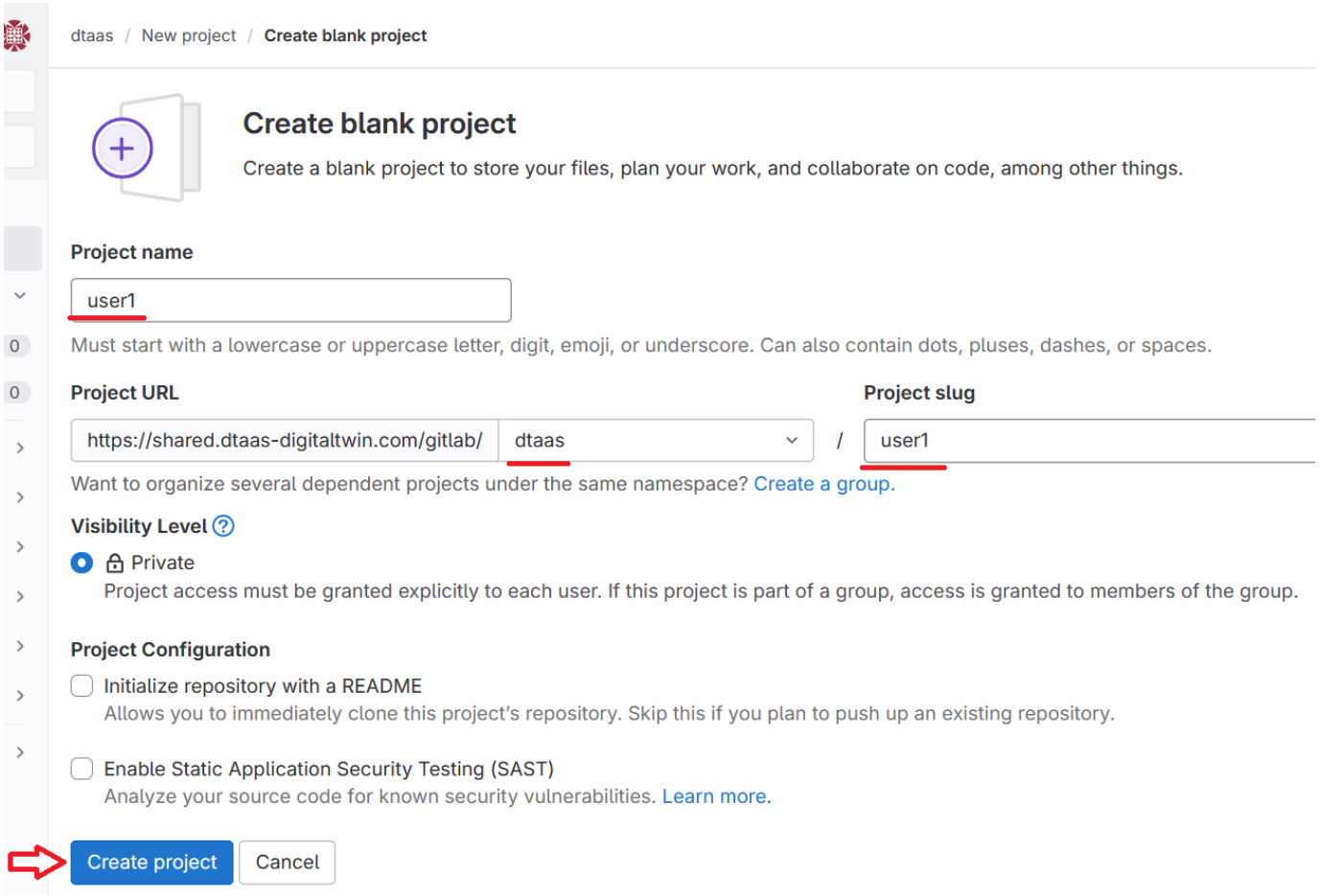
The DTaaS relies on Gitlab for two purposes.

1. OAuth2 authorisation service
2. DevOps service

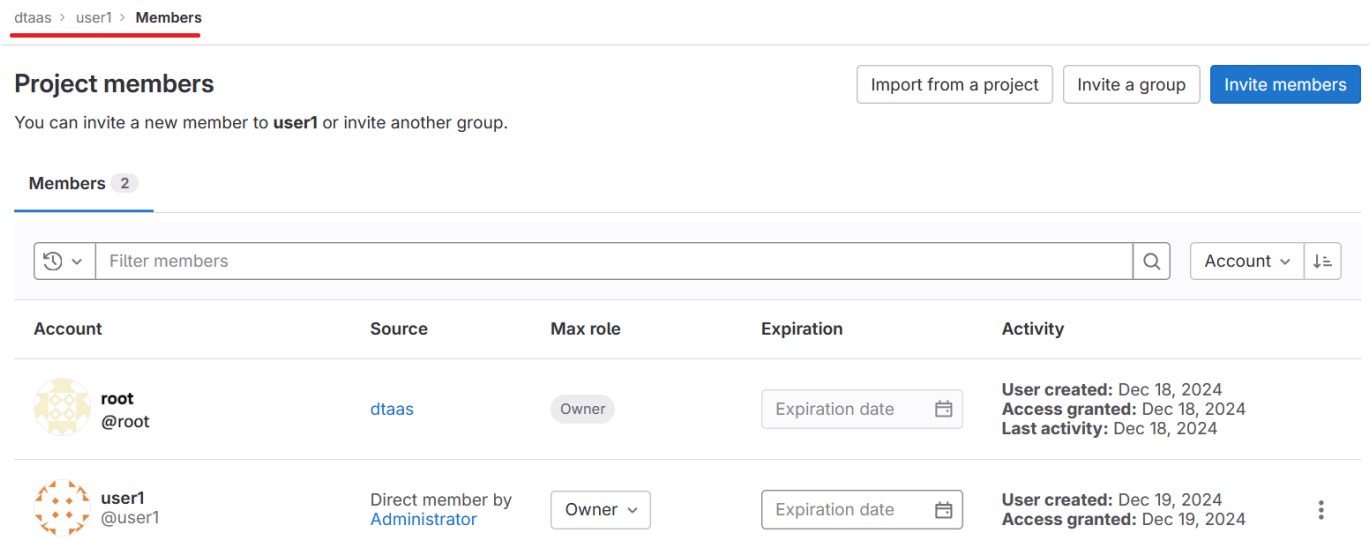
The admin documentation covers the OAuth2 authorisation configuration. This guide covers the use of git commands and project structure for Gitlab DevOps service inside the DTaaS.

2.6.1 Preparation

The first step is to create a gitlab project with *username* in gitlab user group named *dtaas*.



This user needs to have ownership permissions over the project.



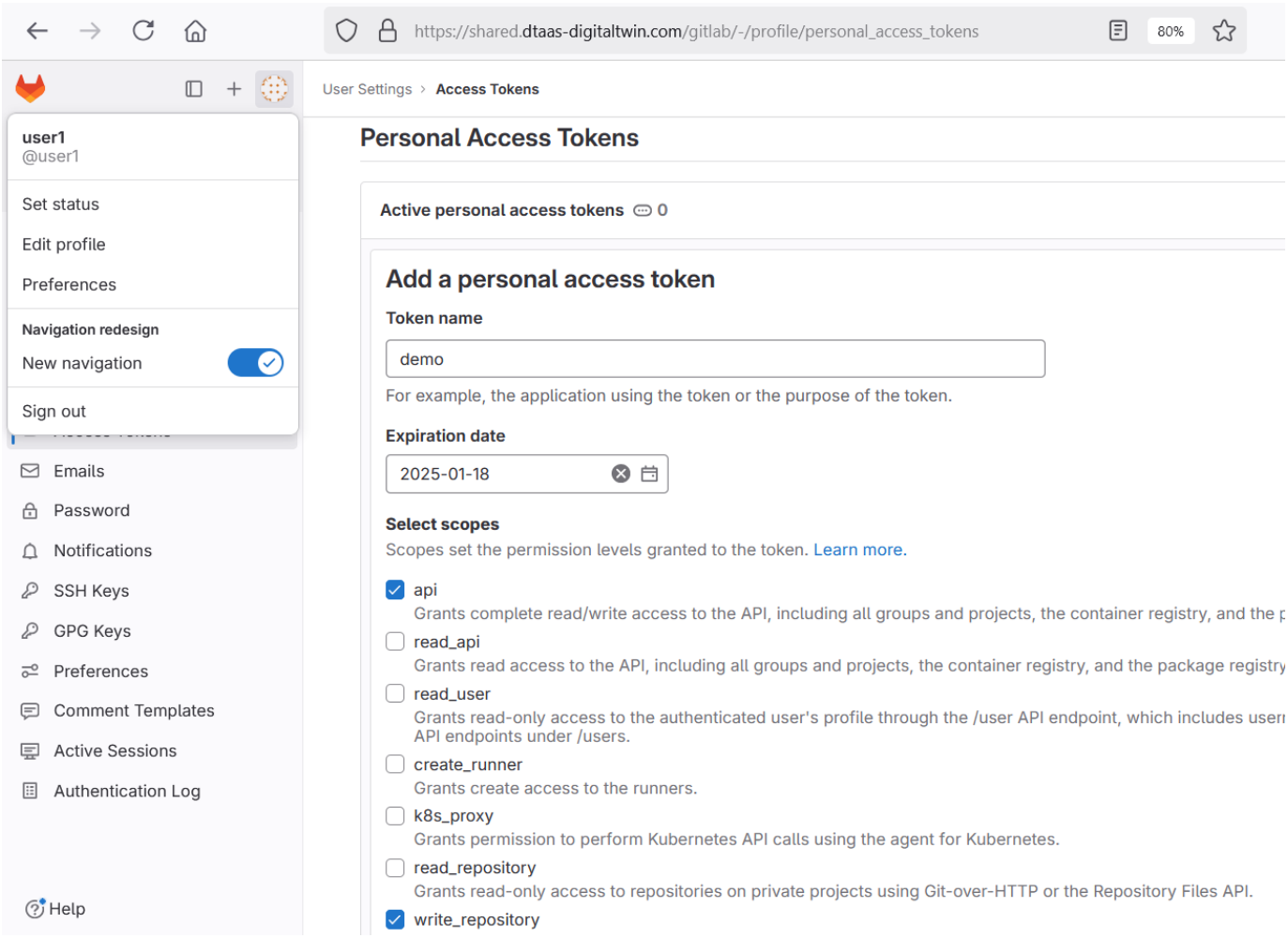
2.6.2 Git commands

The usual git commands and workflows should be used. There are two ways to use Gitlab project as a remote git server.

1. Over SSH using personal SSH key
2. Over HTTPS using personal access tokens (PAT)

This tutorial shows use of PAT for working with Gitlab server.

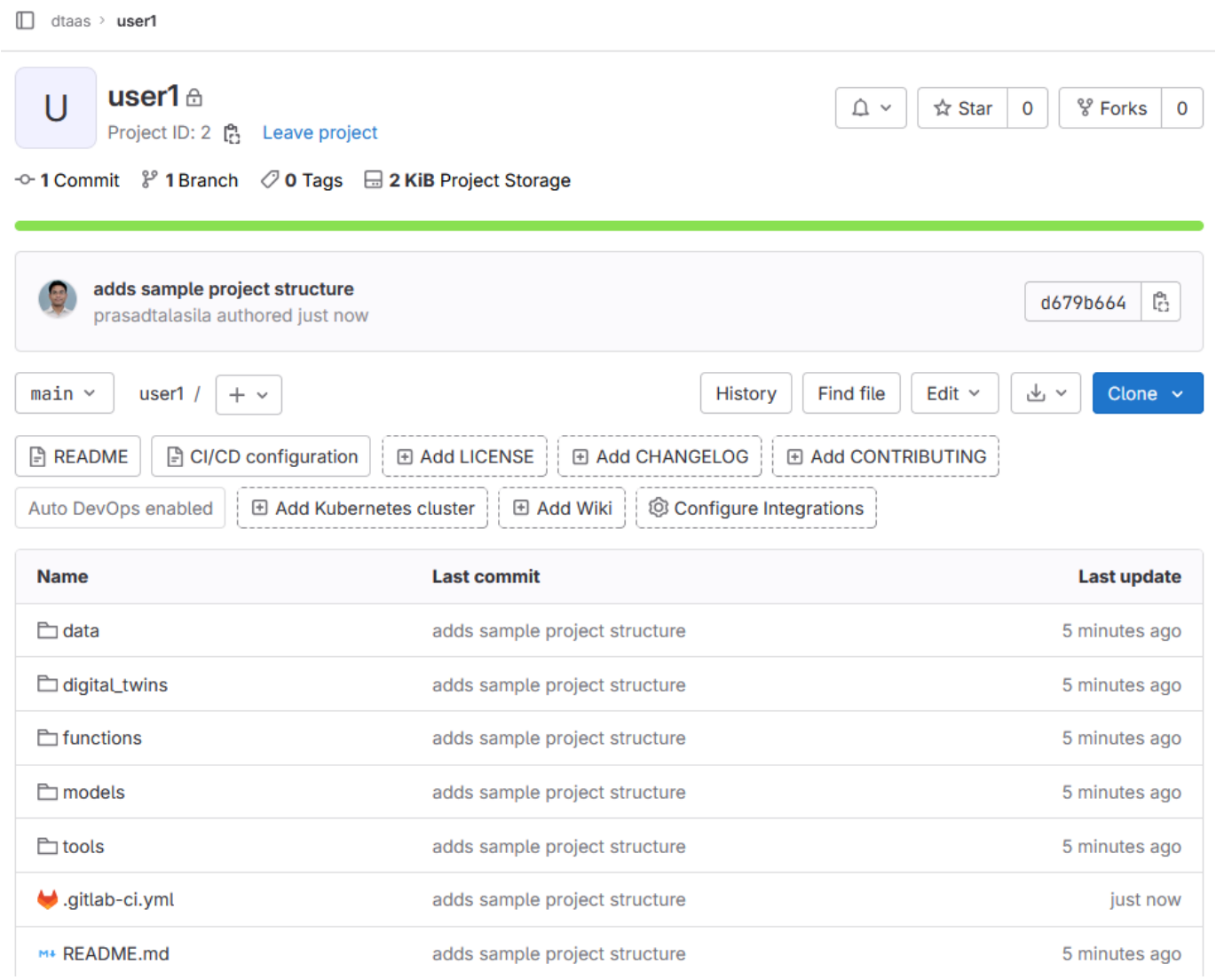
First step is to create PAT.



Copy this token and use it to clone the git repository.

2.6.3 Library Assets

The Gitlab is used to store the reusable **Library** assets of all users. There is a mandatory structure for storing and using Library assets including digital twins. A properly initialised gitlab project should have the following structure.



Please pay special attention to `.gitlab-ci.yml`. It must be a valid Gitlab DevOps configuration. You can also check example repo for a sample structure.

For example, with `PAT1` as PAT of `dtaas/user1` repository, the command to clone the repository is

```
$git clone https://user1:PAT1@shared.dtaas-digitaltwin.com/gitlab/dtaas/user1.git
$cd user1
```

Add the required Library assets and then

```
$git push origin
```

2.6.4 Next Steps

Remember to have a gitlab runner integrated with your project repository. There might already be some runners installed with your DTaaS application. You can check them on the runners page. In addition, you can install your own runners integrated into your repository.

Now, the Digital Twins Preview can be used to access the DevOps features of the DTaaS platform.

2.7 Runner

A utility service to manage safe execution of remote scripts / commands. User launches this from commandline and let the utility manage the commands to be executed.

The runner utility runs as a service and provides REST API interface to safely execute remote commands. Multiple runners can be active simultaneously on one computer. The commands are sent via the REST API and are executed on the computer with active runner.

2.7.1 Install

NPM Registry

The package is available on npmjs.

Install the package with the following command:

```
sudo npm install -g @into-cps-association/runner
```

Github Registry

The package is available in Github packages registry.

Set the registry and install the package with the following commands

```
sudo npm config set @into-cps-association:registry https://npm.pkg.github.com
sudo npm install -g @into-cps-association/runner
```

The *npm install* command asks for username and password. The username is your Github username and the password is your Github personal access token. In order for the npm to download the package, your personal access token needs to have *read:packages* scope.

2.7.2 Configure

The utility requires config specified in YAML format. The template configuration file is:

```
port: 5000
location: 'script' #directory location of scripts
commands: #list of permitted scripts
  - create
  - execute
  - terminate
```

It is suggested that the configuration file be named as *runner.yaml* and placed in the directory in which the *runner* microservice is run.

The `location` refers to the relative location of the scripts directory with respect to the location of *runner.yaml* file.

However, there is no limitation on either the configuration filename or the `location`. The path to *runner.yaml* can either be relative or absolute path. However, the `location` path is always relative path with respect to the path of *runner.yaml* file.

The commands must be executable. Please make sure that the commands have execute permission on Linux platforms.

2.7.3 Create Commands

The runner requires commands / scripts to be run. These need to be placed in the `location` specified in *runner.yaml* file. The location must be relative to the directory in which the **runner** microservice is being run.

2.7.4 🚀 Use

Display help.

```
$runner -h
Usage: runner [options]

Remote code execution for humans

Options:
  -v --version          package version
  -c --config <string>  runner config file specified in yaml format (default: "runner.yaml")
  -h --help             display help
```

The config option is not mandatory. If it is not used, **runner** looks for *runner.yaml* in the directory from which it is being run. Once launched, the utility runs at the port specified in *runner.yaml* file.

```
runner  #use runner.yaml of the present working directory
runner -c FILE-PATH       #absolute or relative path to config file
runner --config FILE-PATH #absolute or relative path to config file
```

If launched on one computer, you can access the same at `http://localhost:<port>`.

Access to the service on network is available at `http://<ip or hostname>:<port>/`.

Application Programming Interface (API)

Three REST API methods are active. The route paths and the responses given for these two sources are:

REST API Route	HTTP Method	Return Value	Comment
localhost:port	POST	Returns the execution status of command	Executes the command provided. Each invocation appends to *array* of commands executed so far.
localhost:port	GET	Returns the execution status of the last command sent via POST request.	
localhost:port/history	GET	Returns the array of POST requests received so far.	

POST REQUEST TO /

Executes a command. The command name given here must exist in *location* directory.

Valid HTTP Request **HTTP Response - Valid Command** **HTTP Response - Inalid Command**

```
POST / HTTP/1.1
Host: foo.com
Content-Type: application/json
Content-Length: 388

{
  "name": "<command-name>"
}
```

```
Connection: close
Content-Length: 134
Content-Type: application/json; charset=utf-8
Date: Tue, 09 Apr 2024 08:51:11 GMT
Etag: W/"86-ja15r8P5HJu72JcROfBTV4sAn2I"
X-Powered-By: Express

{
  "status": "success"
}
```

```
Connection: close
Content-Length: 28
Content-Type: application/json; charset=utf-8
Date: Tue, 09 Apr 2024 08:51:11 GMT
Etag: W/"86-ja15r8P5HJu72JcROfBTV4sAn2I"
X-Powered-By: Express

{
  "status": "invalid command"
}
```

GET REQUEST TO /

Shows the status of the command last executed.

Valid HTTP Request **HTTP Response - Valid Command** **HTTP Response - Inalid Command**

```
GET / HTTP/1.1
Host: foo.com
Content-Type: application/json
Content-Length: 388

{
  "name": "<command-name>"
}
```

```
Connection: close
Content-Length: 134
Content-Type: application/json; charset=utf-8
Date: Tue, 09 Apr 2024 08:51:11 GMT
Etag: W/"86-ja15r8P5HJu72JcROfBTV4sAn2I"
X-Powered-By: Express

{
  "name": "<command-name>",
  "status": "valid",
  "logs": {
    "stdout": "<output log of command>",
    "stderr": "<error log of command>"
  }
}
```

```
Connection: close
Content-Length: 70
Content-Type: application/json; charset=utf-8
Date: Tue, 09 Apr 2024 08:51:11 GMT
Etag: W/"86-ja15r8P5HJu72JcROfBTV4sAn2I"
X-Powered-By: Express

{
  "name": "<command-name",
  "status": "invalid",
  "logs": {
    "stdout": "",
    "stderr": ""
  }
}
```

GET REQUEST TO /HISTORY

Returns the array of POST requests received so far. Both valid and invalid commands are recorded in the history.

```
[
  {
    "name": "valid command"
  },
  {
    "name": "valid command"
  },
  {
    "name": "invalid command"
  }
]
```

2.8 Examples

2.8.1 DTaaS Examples

There are some example digital twins created for the DTaaS software. You can peruse these examples and follow the steps given in this **Examples** section to experience features of the DTaaS software platform and understand best practices for managing digital twins within the platform. Please see these slides and video to get an overview of these examples.

There are two demo vides available: CP-SENS project (slides and video) and Incubator (video). These two videos have been recorded using DTaaS v0.5.0.

Copy Examples

The first step is to copy all the example code into your user workspace within the DTaaS. Use the given shell script to copy all the examples into `/workspace/examples` directory.

```
wget https://raw.githubusercontent.com/INTO-CPS-Association/DTaaS-examples/main/getExamples.sh
bash getExamples.sh
```

Example List

The digital twins provided in examples vary in their complexity. It is best to use the examples in the following order.

1. Mass Spring Damper
2. Water Tank Fault Injection
3. Water Tank Model Swap
4. Desktop Robotti and RabbitMQ
5. Water Treatment Plant and OPC-UA
6. Three Water Tanks with DT Manager Framework
7. Flex Cell with Two Industrial Robots
8. Incubator
9. Firefighters in Emergency Environments
10. Mass Spring Damper with NuRV Runtime Monitor FMU
11. Water Tank Fault Injection with NuRV Runtime Monitor FMU
12. Incubator Co-Simulation with NuRV Runtime Monitor FMU
13. Incubator with NuRV Runtime Monitor as Service
14. Incubator with NuRV Runtime Monitor FMU as Service

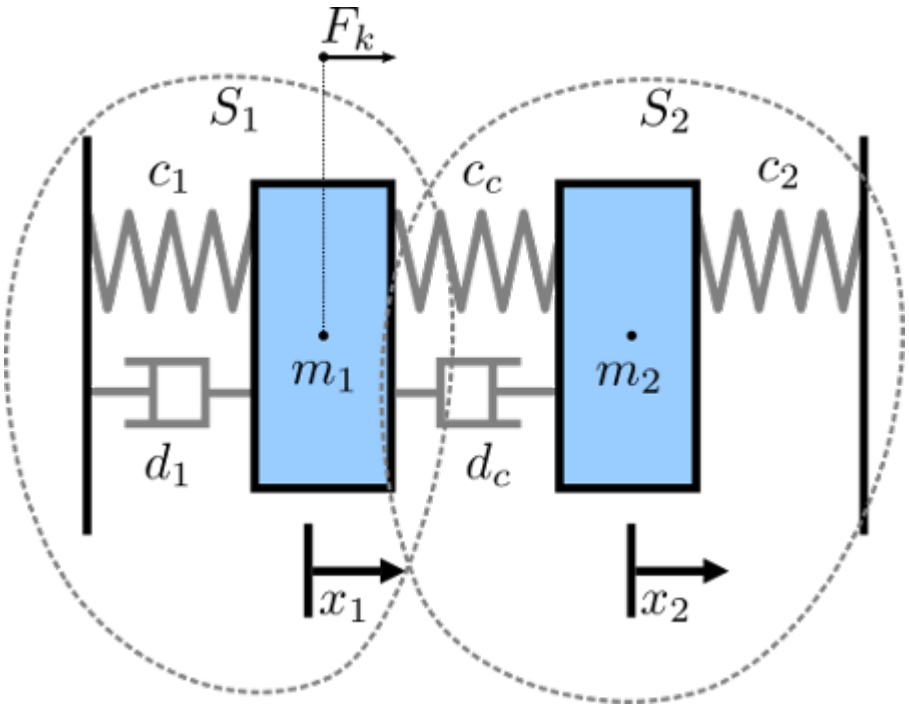
DTaaS examples

2.8.2 Mass Spring Damper

Overview

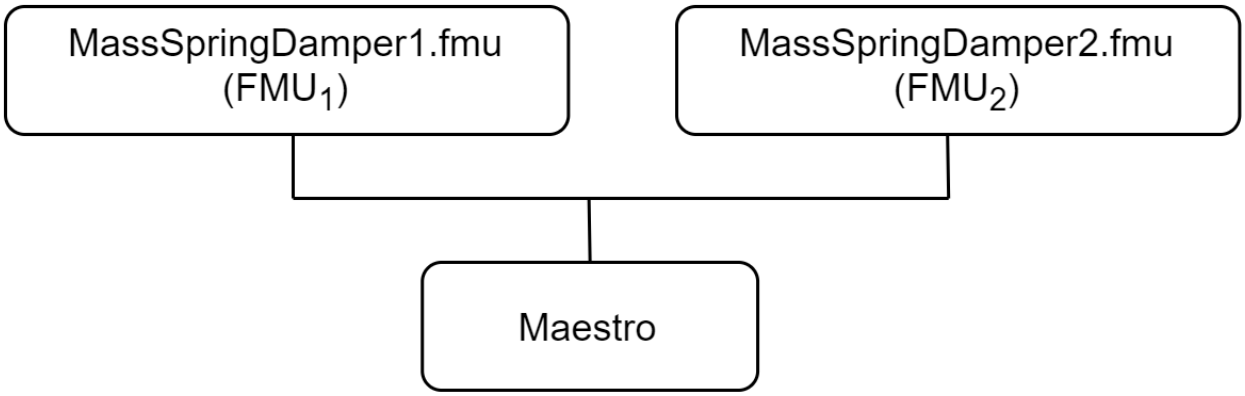
The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within DTaaS.

Example Diagram



Example Structure

There are two simulators included in the study, each representing a mass spring damper system. The first simulator calculates the mass displacement and speed of for a given force acting on mass . The second simulator calculates force given a displacement and speed of mass . By coupling these simulators, the evolution of the position of the two masses is computed.



Digital Twin Configuration

This example uses two models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	MassSpringDamper1.fmu	Private	Yes
	MassSpringDamper2.fmu	Private	Yes
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The `co-sim.json` and `time.json` are two DT configuration files used for executing the digital twin. You can change these two files to customize the DT to your needs.

Lifecycle Phases

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool
Execute	Produces and stores output in data/mass-spring-damper/output directory
Clean	Clears run logs and outputs

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital_twins/mass-spring-damper
```

If required, change the execute permission of lifecycle scripts you need to execute, for example:

```
chmod +x lifecycle/create
```

Now, run the following scripts:

CREATE

Installs Open Java Development Kit 17 in the workspace.

```
lifecycle/create
```

EXECUTE

Run the the Digital Twin. Since this is a co-simulation based digital twin, the Maestro co-simulation tool executes co-simulation using the two FMU models.

```
lifecycle/execute
```

Examine the results

The results can be found in the */workspace/examples/data/mass-spring-damper/output directory*.

You can also view run logs in the */workspace/examples/digital_twins/mass-spring-damper*.

TERMINATE PHASE

Terminate to clean up the debug files and co-simulation output files.

```
lifecycle/terminate
```

References

More information about co-simulation techniques and mass spring damper case study are available in:

```
Gomes, Cláudio, et al. "Co-simulation: State of the art."
arXiv preprint arXiv:1702.00686 (2017).
```

The source code for the models used in this DT are available in mass spring damper github repository.

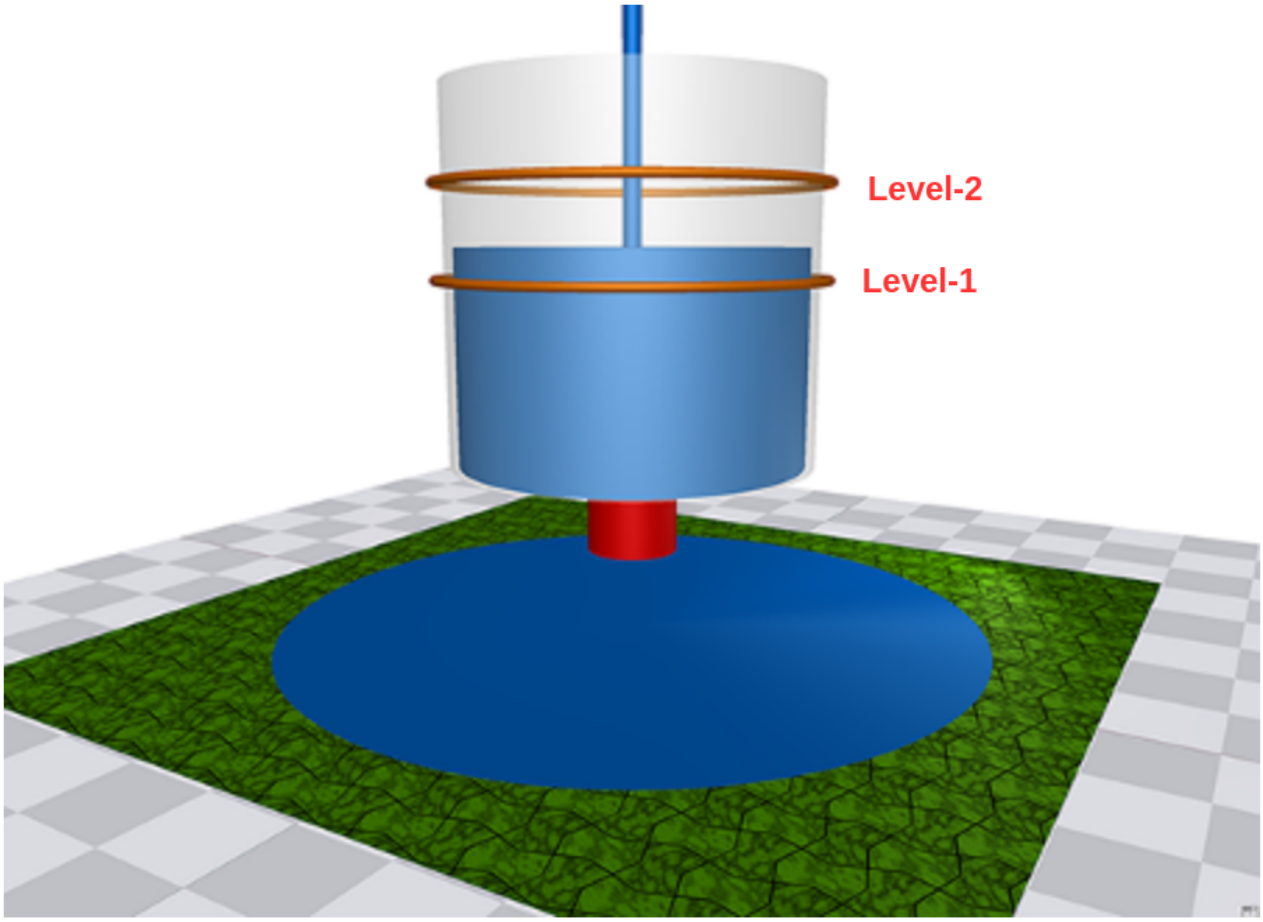
2.8.3 Water Tank Fault Injection

Overview

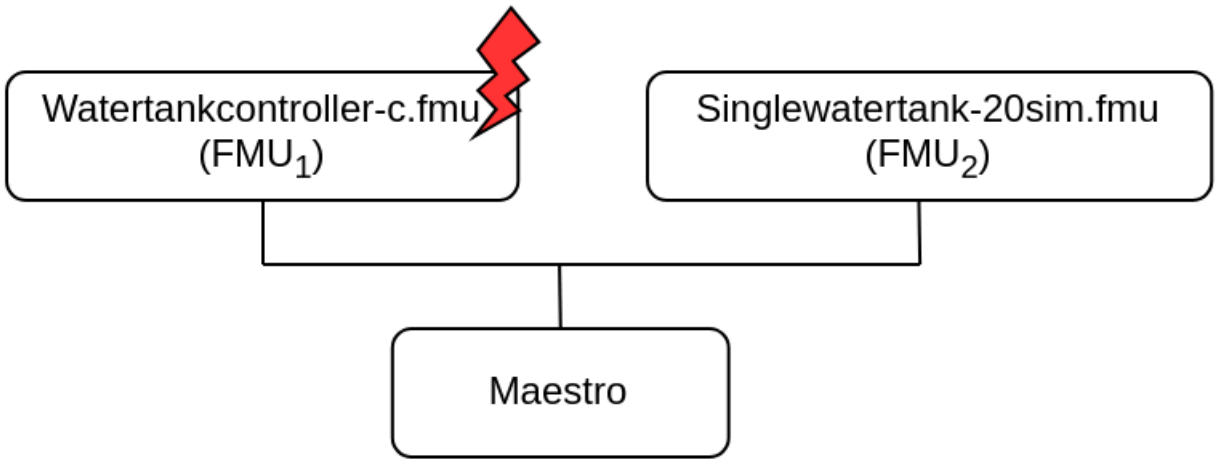
This example shows a fault injection (FI) enabled digital twin (DT). A live DT is subjected to simulated faults received from the environment. The simulated faults is specified as part of DT configuration and can be changed for new instances of DTs.

In this co-simulation based DT, a watertank case-study is used; co-simulation consists of a tank and controller. The goal of which is to keep the level of water in the tank between `Level-1` and `Level-2`. The faults are injected into output of the water tank controller (**Watertankcontroller-c.fmu**) from 12 to 20 time units, such that the tank output is closed for a period of time, leading to the water level increasing in the tank beyond the desired level (`Level-2`).

Example Diagram



Example Structure



Digital Twin Configuration

This example uses two models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	watertankcontroller-c.fmu	Private	Yes
	singlewatertank-20sim.fmu	Private	Yes
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The `multimodelFI.json` and `simulation-config.json` are two DT configuration files used for executing the digital twin. You can change these two files to customize the DT to your needs.

The faults are defined in **wt_fault.xml**.

Lifecycle Phases

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool
Execute	Produces and stores output in data/water_tank_FI/output directory
Clean	Clears run logs and outputs

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital_twins/water_tank_FI
```

If required, change the execute permission of lifecycle scripts you need to execute, for example:

```
chmod +x lifecycle/create
```

Now, run the following scripts:

CREATE

Installs Open Java Development Kit 17 and pip dependencies. The pandas and matplotlib are the pip dependencies installated.

```
lifecycle/create
```

EXECUTE

Run the co-simulation. Generates the co-simulation output.csv file at `/workspace/examples/data/water_tank_FI/output`.

```
lifecycle/execute
```

ANALYZE PHASE

Process the output of co-simulation to produce a plot at: `/workspace/examples/data/water_tank_FI/output/plots/`.

```
lifecycle/analyze
```

Examine the results

The results can be found in the */workspace/examples/data/water_tank_FI/output directory*.

You can also view run logs in the */workspace/examples/digital_twins/water_tank_FI*.

TERMINATE PHASE

Clean up the temporary files and delete output plot

```
lifecycle/terminate
```

References

More details on this case-study can be found in the paper:

```
M. Frasheri, C. Thule, H. D. Macedo, K. Lausdahl, P. G. Larsen and
L. Esterle, "Fault Injecting Co-simulations for Safety,"
2021 5th International Conference on System Reliability and Safety (ICSRS),
Palermo, Italy, 2021.
```

The fault-injection plugin is an extension to the Maestro co-orchestration engine that enables injecting inputs and outputs of FMUs in an FMI-based co-simulation with tampered values. More details on the plugin can be found in fault injection git repository. The source code for this example is also in the same github repository in a example directory.

2.8.4 Water Tank Model Swap

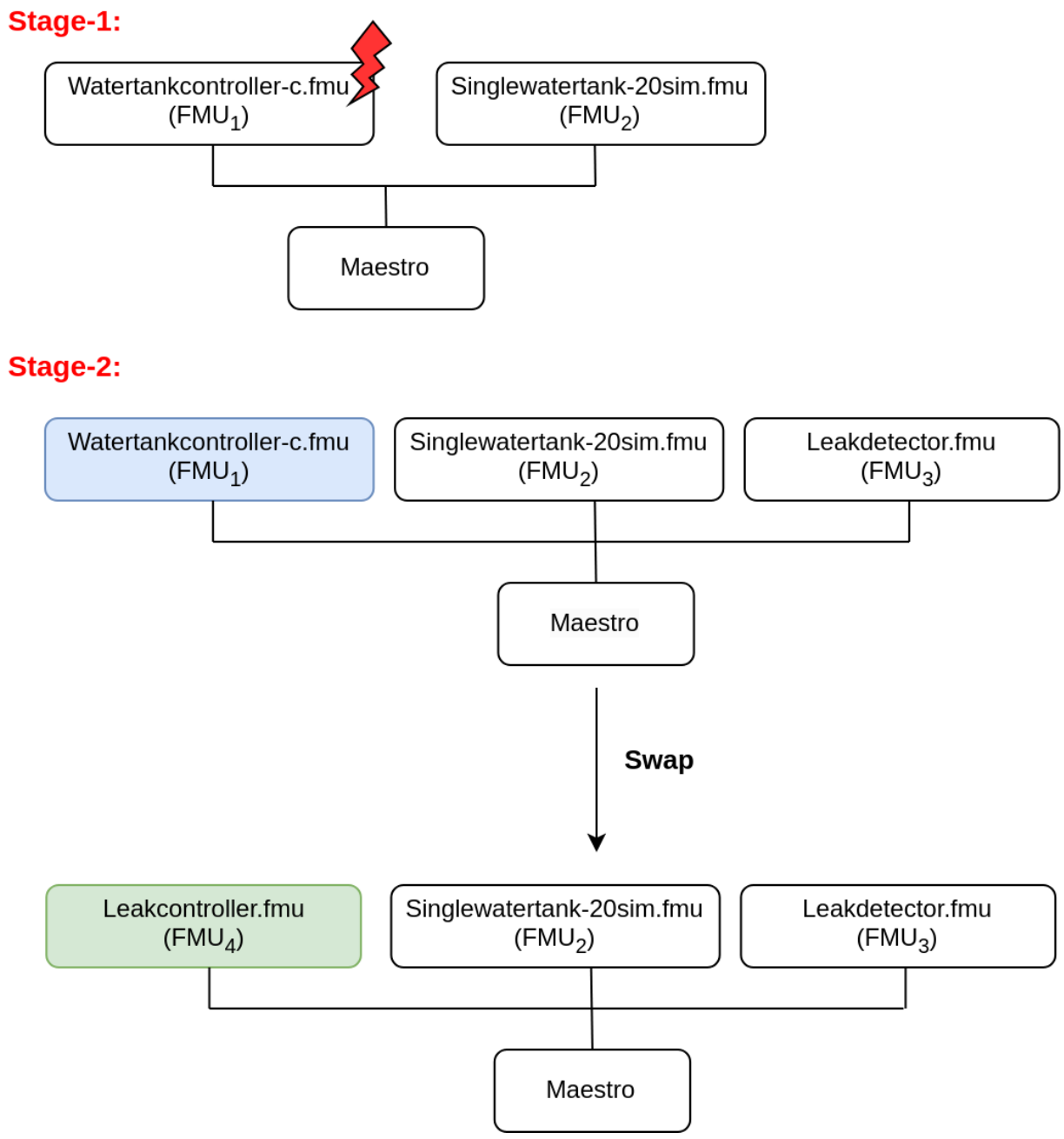
Overview

This example shows multi-stage execution and dynamic reconfiguration of a digital twin (DT). Two features of DTs are demonstrated here:

- Fault injection into live DT
- Dynamic auto-reconfiguration of live DT

The co-simulation methodology is used to construct this DT.

Example Structure



Configuration of assets

This example uses four models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	Watertankcontroller-c.fmu	Private	Yes
	Singlewatertank-20sim.fmu	Private	Yes
	Leak_detector.fmu	Private	No
	Leak_controller.fmu	Private	No
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

This DT has many configuration files. The DT is executed in two stages. There exist separate DT configuration files for each stage. The following table shows the configuration files and their purpose.

Configuration file name	Execution Stage	Purpose
mm1. json	stage-1	DT configuration
wt_fault.xml, FaultInject.mabl	stage-1	faults injected into DT during stage-1
mm2.json	stage-2	DT configuration
simulation-config.json	Both stages	Configuration for specifying DT execution time and output logs

Lifecycle Phases

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool
Execute	Produces and stores output in data/water_tank_swap/output directory
Analyze	Process the co-simulation output and produce plots
Clean	Clears run logs, outputs and plots

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital_twins/water_tank_swap
```

If required, change the permission of files you need to execute, for example:

```
chmod +x lifecycle/create
```

Now, run the following scripts:

CREATE

Installs Open Java Development Kit 17 and pip dependencies. The matplotlib pip package is also installated.

```
lifecycle/create
```

EXECUTE

This DT has two-stage execution. In the first-stage, a co-simulation is executed. The Watertankcontroller-c.fmu and Singlewatertank-20sim.fmu models are used to execute the DT. During this stage, faults are injected into one of the models (Watertankcontroller-c.fmu) and the system performance is checked.

In the second-stage, another co-simulation is run in which three FMUs are used. The FMUs used are: watertankcontroller, singlewatertank-20sim, and leak_detector. There is an in-built monitor in the Maestro tool. This monitor is enabled during the stage and a swap condition is set at the beginning of the second-stage. When the swap condition is satisfied, the Maestro swaps out Watertankcontroller-c.fmu model and swaps in Leakcontroller.fmu model. This swapping of FMU models demonstrates the dynamic reconfiguration of a DT.

The end of execution phase generates the co-simulation output.csv file at `/workspace/examples/data/water_tank_swap/output`.

```
lifecycle/execute
```

ANALYZE PHASE

Process the output of co-simulation to produce a plot at: `/workspace/examples/data/water_tank_FI/output/plots/`.

```
lifecycle/analyze
```

Examine the results

The results can be found in the *workspace/examples/data/water_tank_swap/output directory*.

You can also view run logs in the *workspace/examples/digital_twins/water_tank_swap*.

TERMINATE PHASE

Clean up the temporary files and delete output plot

```
lifecycle/terminate
```

References

The complete source of this example is available on model swap github repository.

The runtime model (FMU) swap mechanism demonstrated by the experiment is detailed in the paper:

```
Ejersbo, Henrik, et al. "fmiSwap: Run-time Swapping of Models for
Co-simulation and Digital Twins." arXiv preprint arXiv:2304.07328 (2023).
```

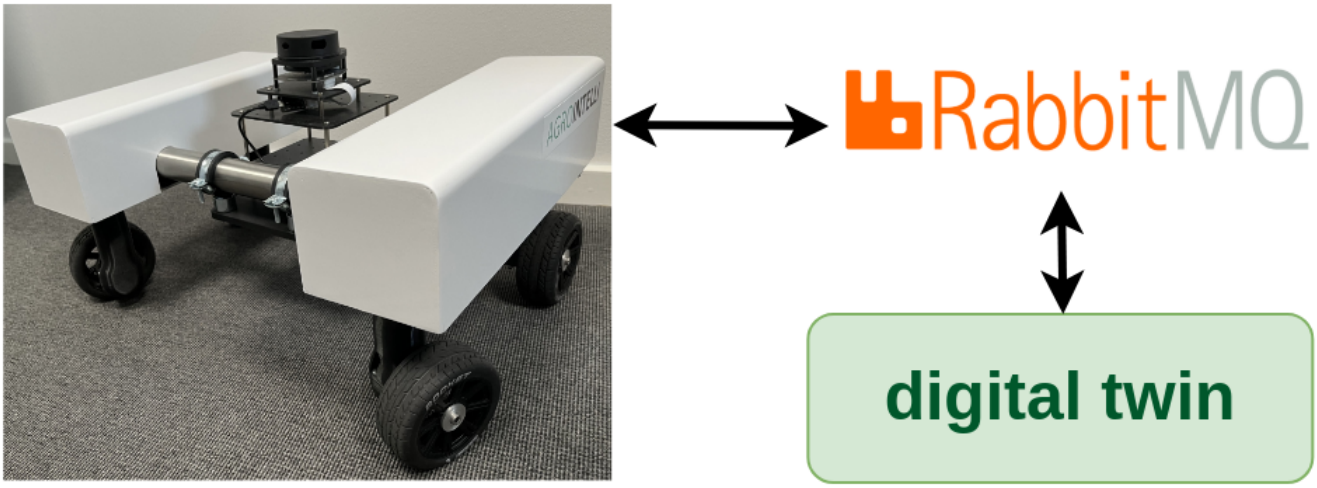
The runtime reconfiguration of co-simulation by modifying the Functional Mockup Units (FMUs) used is further detailed in the paper:

```
Ejersbo, Henrik, et al. "Dynamic Runtime Integration of
New Models in Digital Twins." 2023 IEEE/ACM 18th Symposium on
Software Engineering for Adaptive and Self-Managing Systems
(SEAMS). IEEE, 2023.
```

2.8.5 Desktop Robotti with RabbitMQ

Overview

This example demonstrates bidirectional communication between a mock physical twin and a digital twin of a mobile robot (Desktop Robotti). The communication is enabled by RabbitMQ Broker.

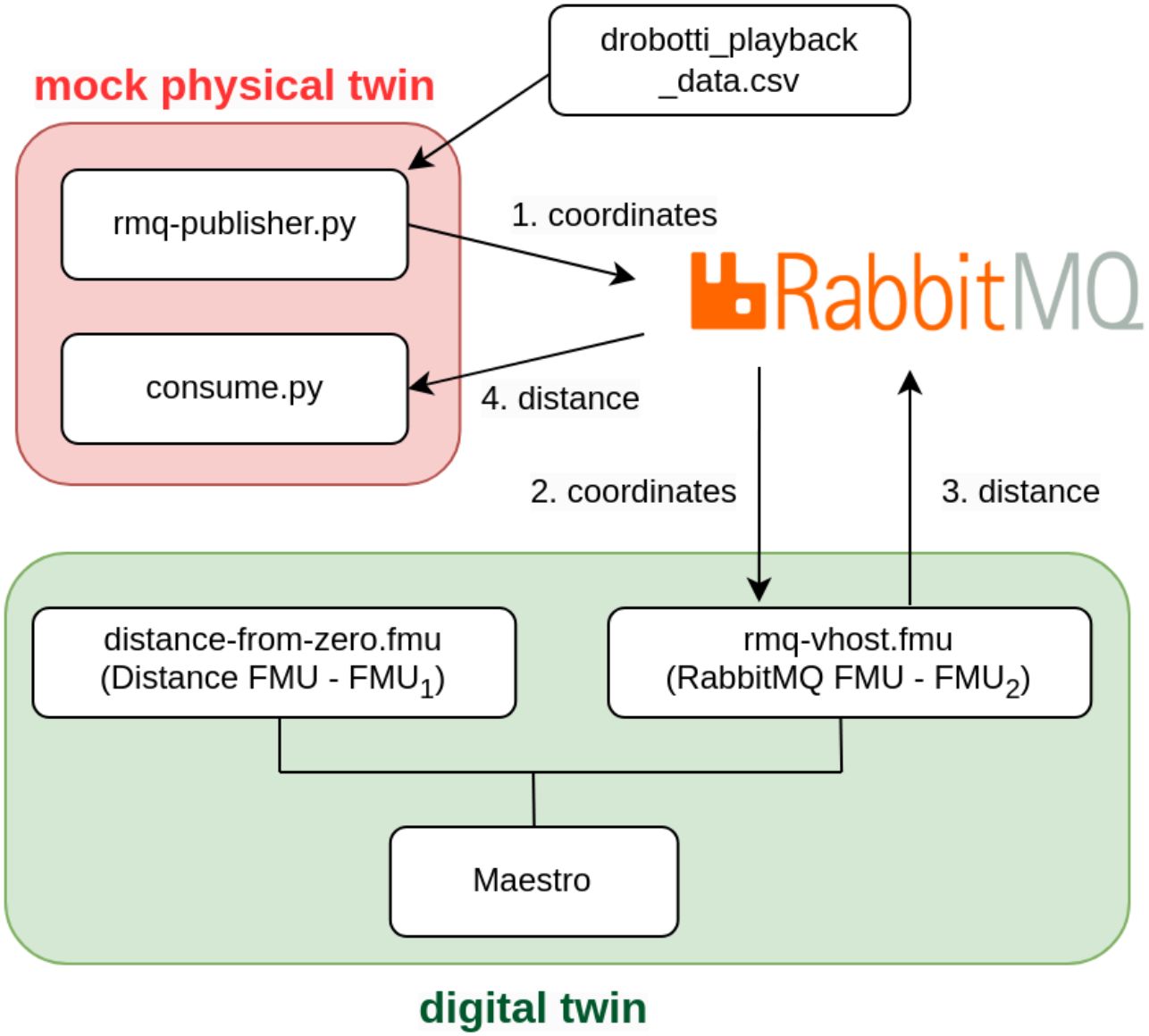


Example Structure

The mock physical twin of mobile robot is created using two python scripts

1. data/drobotti_rmqfmu/rmq-publisher.py
2. data/drobotti_rmqfmu/consume.py

The mock physical twin sends its physical location in *(x,y)* coordinates and expects a cartesian distance calculated from digital twin.



The *rmq-publisher.py* reads the recorded *(x,y)* physical coordinates of mobile robot. The recorded values are stored in a data file. These *(x,y)* values are published to RabbitMQ Broker. The published *(x,y)* values are consumed by the digital twin.

The *consume.py* subscribes to RabbitMQ Broker and waits for the calculated distance value from the digital twin.

The digital twin consists of a FMI-based co-simulation, where Maestro is used as co-orchestration engine. In this case, the co-simulation is created by using two FMUs - RMQ FMU (rabbitmq-vhost.fmu) and distance FMU (distance-from-zero.fmu). The RMQ FMU receives the *(x,y)* coordinates from *rmq-publisher.py* and sends calculated distance value to *consume.py*. The RMQ FMU uses RabbitMQ broker for communication with the mock mobile robot, i.e., *rmq-publisher.py* and *consume.py*. The distance FMU is responsible for calculating the distance between *(0,0)* and *(x,y)*. The RMQ FMU and distance FMU exchange values during co-simulation.

Digital Twin Configuration

This example uses two models, one tool, one data, and two scripts to create mock physical twin. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	distance-from-zero.fmu	Private	No
	rmq-vhost.fmu	Private	Yes
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes
Data	drobotti_playback_data.csv	private	No
Mock PT	rmq-publisher.py	Private	No
	consume.py	Private	No

This DT has many configuration files. The `coe.json` and `multimodel.json` are two DT configuration files used for executing the digital twin. You can change these two files to customize the DT to your needs.

The RabbitMQ access credentials need to be provided in `multimodel.json`. The `rabbitMQ-credentials.json` provides RabbitMQ access credentials for mock PT python scripts. Please add your credentials in both these files.

Lifecycle Phases

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool and pip packages for python scripts
Execute	Runs both DT and mock PT
Clean	Clears run logs and outputs

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital_twins/drobotti_rmqfmu
```

If required, change the execute permission of lifecycle scripts you need to execute, for example:

```
chmod +x lifecycle/create
```

Now, run the following scripts:

CREATE

Installs Open Java Development Kit 17 in the workspace. Also install the required python pip packages for *rmq-publisher.py* and *consume.py* scripts.

```
lifecycle/create
```

EXECUTE

Run the python scripts to start mock physical twin. Also run the the Digital Twin. Since this is a co-simulation based digital twin, the Maestro co-simulation tool executes co-simulation using the two FMU models.

```
lifecycle/execute
```

Examine the results

The results can be found in the */workspace/examples/digital_twins/drobotti_rmqfmu directory*.

Executing the DT will generate and launch a co-simulation (RMQFMU and distance FMU), and two python scripts. One to publish data that is read from a file. And one to consume what is sent by the distance FMU.

In this examples the DT will run for 10 seconds, with a stepsize of 100ms. Thereafter it is possible to examine the logs produce in `/workspace/examples/digital_twins/drobotti_rmqfmu/target`. The outputs for each FMU, xpos and ypos for the RMQFMU, and the distance for the distance FMU are recorded in the `outputs.csv` file. Other logs can be examined for each FMU and the publisher scripts. Note that, the RMQFMU only sends data, if the current input is different to the previous one.

TERMINATE PHASE

Terminate to clean up the debug files and co-simulation output files.

```
lifecycle/terminate
```

References

The RabbitMQ FMU github repository contains complete documentation and source code of the rmq-vhost.fmu.

More information about the case study is available in:

```
Frasheri, Mirgita, et al. "Addressing time discrepancy between digital
and physical twins." Robotics and Autonomous Systems 161 (2023): 104347.
```

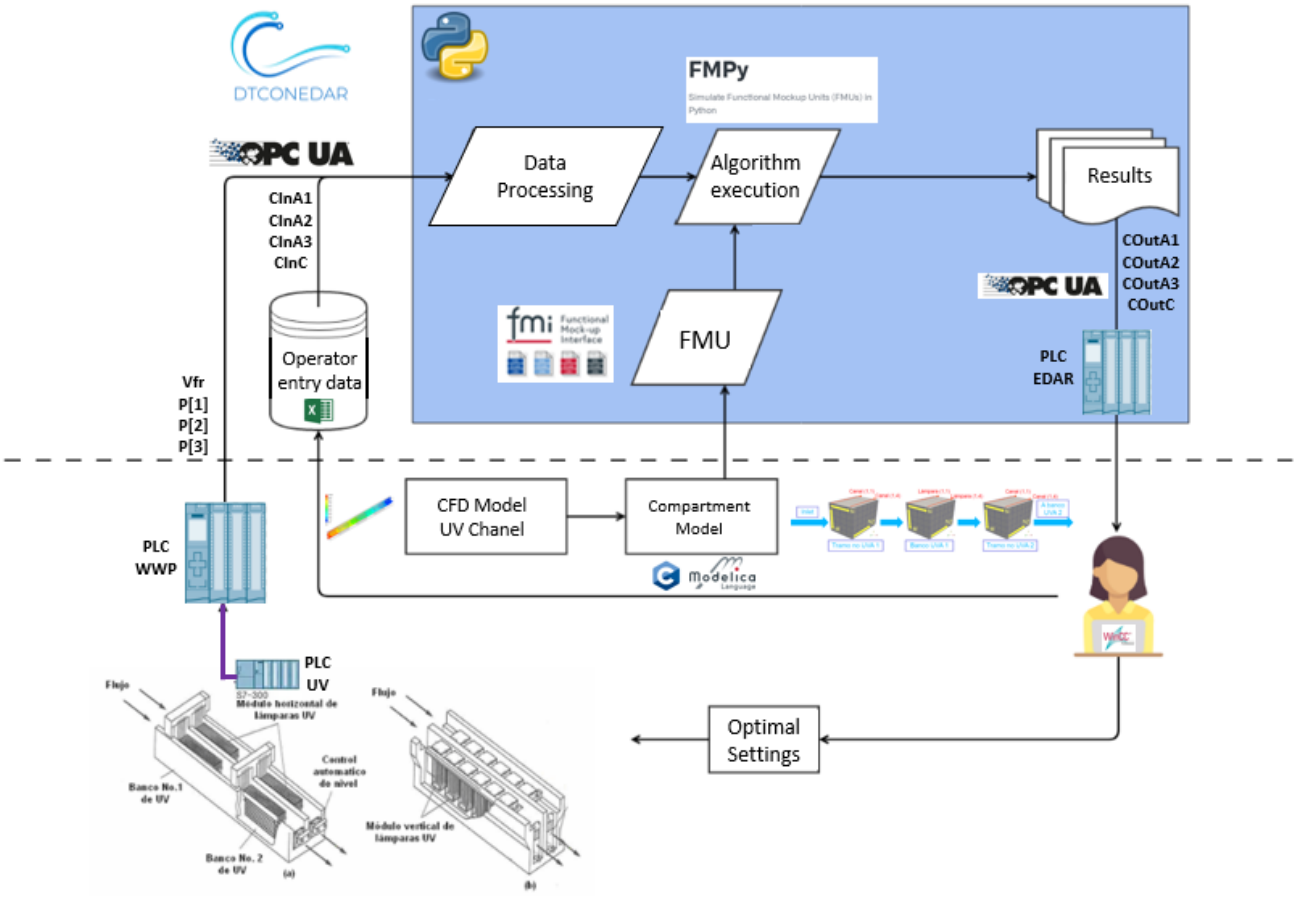
2.8.6 Waste Water Plant with OPC-UA

Introduction

Waste water treatment (WWT) plants must comply with substance and species concentration limits established by regulation in order to ensure the good quality of the water. This is usually done taking periodic samples that are analyzed in the laboratory. This means that plant operators do not have continuous information for making decisions and, therefore, operation setpoints are set to higher values than needed to guarantee water quality. Some of the processes involved in WWT plants consume a lot of power, thus adjusting setpoints could significantly reduce energy consumption.

Physical Twin Overview

This example demonstrates the communication between a physical ultraviolet (UV) disinfection process (the tertiary treatment of a WWT plant) and its digital twin, which is based on Computational Fluid Dynamics (CFD) and compartment models. The aim of this digital twin is to develop "virtual sensors" that provide continuous information that facilitates the decision making process for the plant operator.



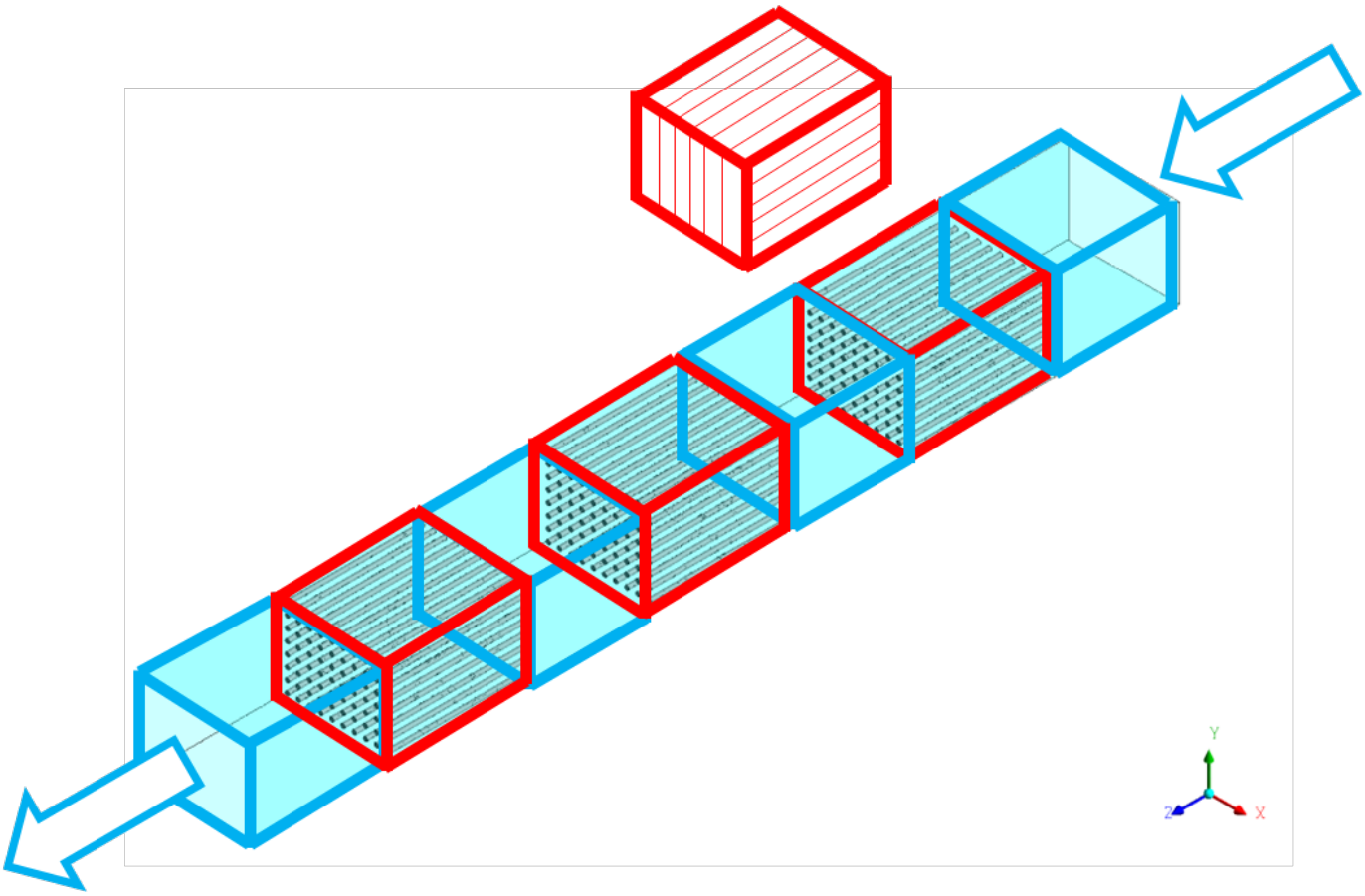
The physical twin of the waste water plant is composed of an ultraviolet channel controlled by a PLC that controls the power of the UV lamps needed to kill all the pathogens of the flow. The channel has 3 groups of UV lamps, therefore the real channel (and is mathematical model) is subdivided into 7 zones: 4 correspond to zones without UV lamps (2 for the entrance and exit of the channel + 2 zones between UV lamps) and the 3 reamaining for the UV lamps.

The dose to be applied (related with the power) changes according to the residence time (computed from the measure of the volume flow) and the UV intensity (measured by the intensity sensor).

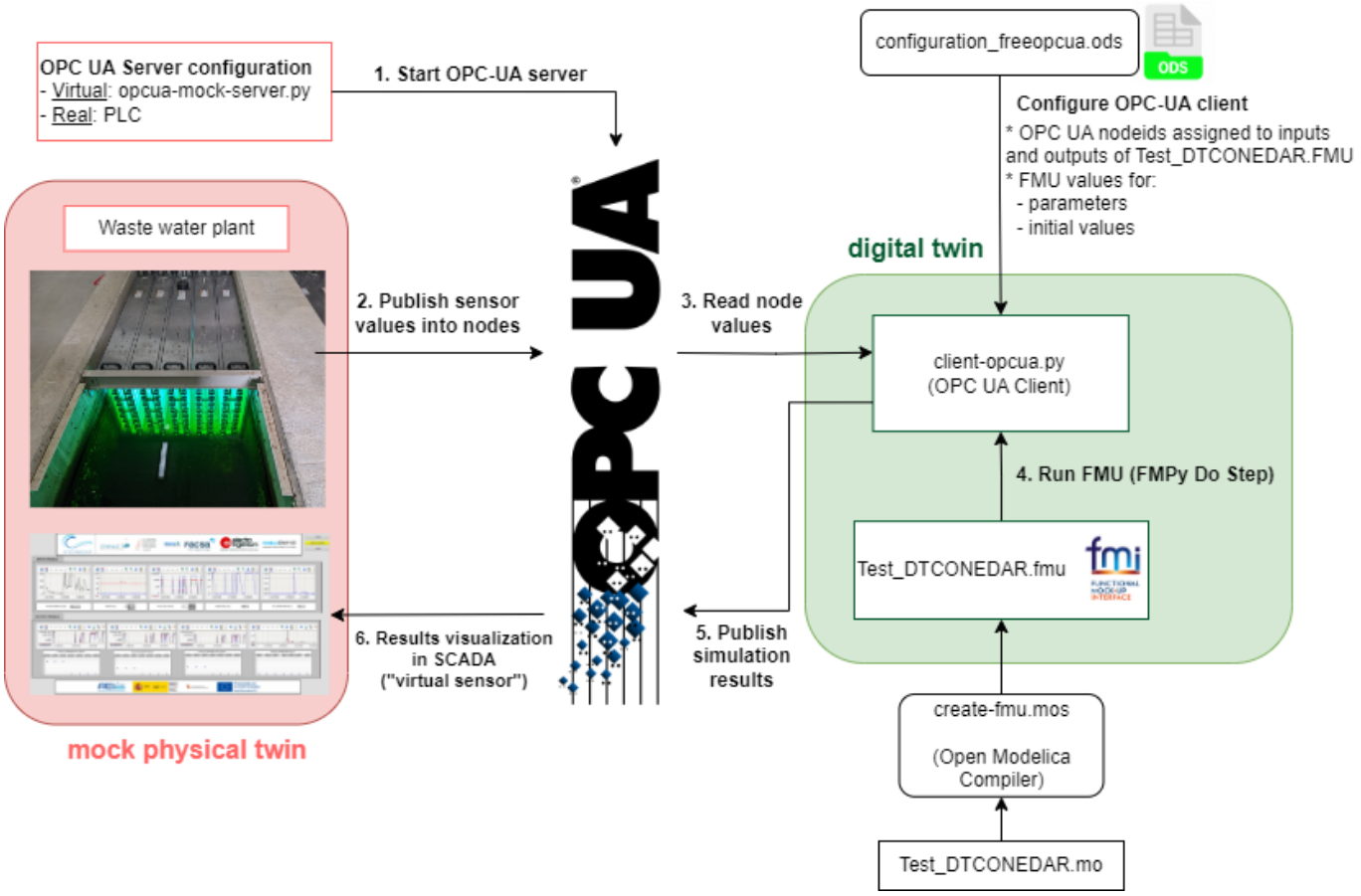
The information of the volumetric flow and power (in the three parts of the channel) is transmitted to the PLC of the plant. Furthermore, the PLC is working as OPC UA Server to send and receive data to and from an OPC UA Client. Additionally, some sizing parameters and initial values are read from a spreadsheet filled in by the plant operator. In this case, the spreadsheet is an Open Office file (.ods) due to the software installed in the SCADA PC. Some of the variables like initial concentration of disinfectant and pathogens are included, among

others. Some values defined by the plant operator correspond to input signals that are not currently being measured, but are expected to be measured in the future.

Digital Twin Overview



The digital twin is a reduced model (developed in C) that solves physical conservation laws (mass, energy and momentum), but simplifies details (geometry, mainly) to ensure real-time calculations and accurate results. The results are compared to the ones obtained by the CFD. C solver developed is used by the OpenModelica model. OpenModelica converts it into the FMI standard, to be integrated in the OPC UA Client (*client-opcua.py*).



Digital Twin Configuration

Asset Type	Name of Asset	Visibility	Reuse in Other Examples
Model	Test_DTCONEDAR.mo	private	No
Data	configuration_freopcua.ods	private	No
	model_description.csv (generated by client-asyncua.py)	private	No
	Mock OPC UA Server: opcua-mock-server.py	private	No
Tool	**OPC UA Client**: client-opcua.py	private	No
	FMU builder: create-fmu.mos	private	No

In this example, a dummy model representation of the plant is used, instead of the real model. The simplified model (with not the real equations) is developed in **Open Modelica (Test_DTCONEDAR.mo)**. The FMU is generated from the Open Modelica interface to obtain the needed binaries to run the FMU. It is possible to run an FMU previously generated, however, to ensure that we are using the right binaries it is recommended to install Open Modelica Compiler and run `script.mos` to build the FMU from the Modelica file `Test_DTCONEDAR.mo`.

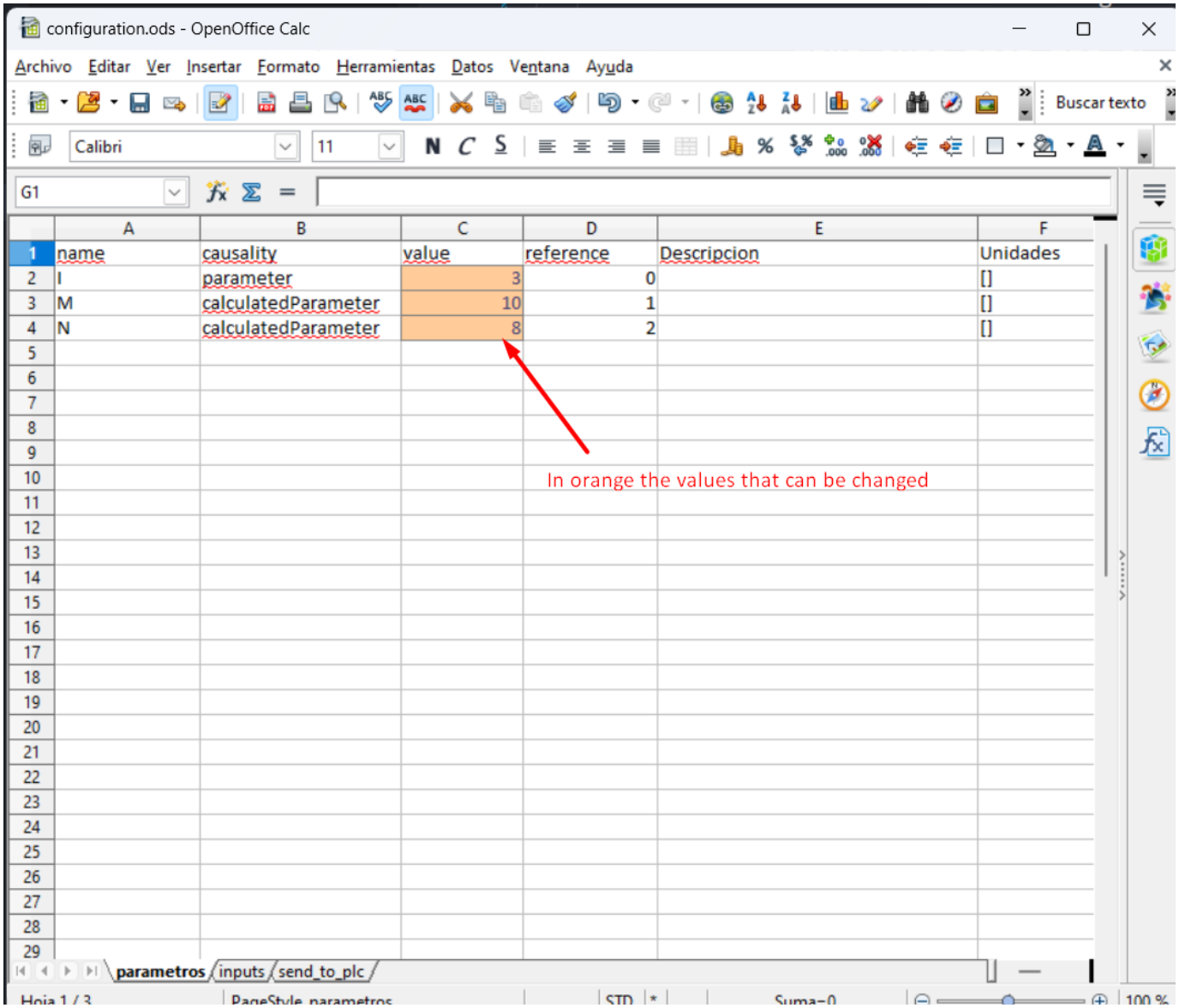
The FMU model description file (`modelDescription.xml` file inside `Test_DTCONEDAR.fmu`) has the information of the value references configuration_freeopcua.ods has the information of the OPC-UA node IDs And both have in common the variable name

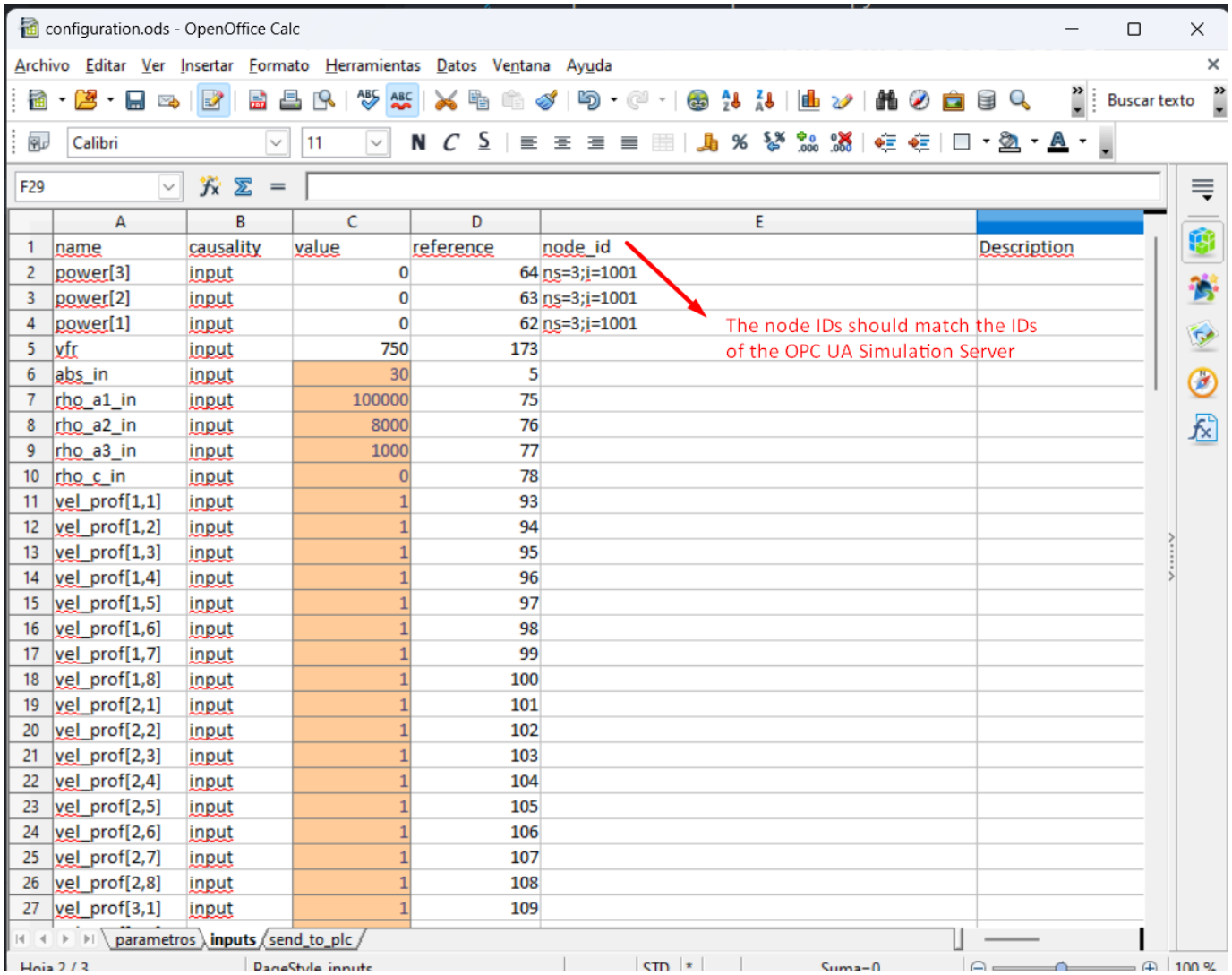
The client python script (`client-opcua.py`) does the following actions:

- Reads the variable names and the variable value references from the model description file of the Test_DTCONEDAR.FMU.
- Reads *configuration_freeopcua.ods* to obtain opcua node IDs and assigns those node IDs to the variables read from the FMU
- Read *configuration_freeopcua.ods* to fix initial values, parameters and some inputs (those inputs that are not being measured, a reasonable value is assumed).
- Read values from PLC using a client OPC.
- Execute the algorithm with the FMPy library using the .fmu created from the compartment model (based on CFD)
- Obtain results.
- Send by OPC UA protocol the result values to the PLC, to visualize them in the SCADA and with the aim to improve the decision-making process of the plant operator.

INPUT DATA VARIABLES

The *configuration_freeopcua.ods* date file is used for customizing the initial input data values used by the server.





	A	B	C	D	E	
1	name	causality	value	reference	node_id	Description
2	power[3]	input	0	64	ns=3;i=1001	
3	power[2]	input	0	63	ns=3;i=1001	
4	power[1]	input	0	62	ns=3;i=1001	
5	vfr	input	750	173		
6	abs_in	input	30	5		
7	rho_a1_in	input	100000	75		
8	rho_a2_in	input	8000	76		
9	rho_a3_in	input	1000	77		
10	rho_c_in	input	0	78		
11	vel_prof[1,1]	input	1	93		
12	vel_prof[1,2]	input	1	94		
13	vel_prof[1,3]	input	1	95		
14	vel_prof[1,4]	input	1	96		
15	vel_prof[1,5]	input	1	97		
16	vel_prof[1,6]	input	1	98		
17	vel_prof[1,7]	input	1	99		
18	vel_prof[1,8]	input	1	100		
19	vel_prof[2,1]	input	1	101		
20	vel_prof[2,2]	input	1	102		
21	vel_prof[2,3]	input	1	103		
22	vel_prof[2,4]	input	1	104		
23	vel_prof[2,5]	input	1	105		
24	vel_prof[2,6]	input	1	106		
25	vel_prof[2,7]	input	1	107		
26	vel_prof[2,8]	input	1	108		
27	vel_prof[3,1]	input	1	109		

DT CONFIG

The *config.json* specifies the configuration parameters for the OPC UA client.

config.json M

```
{
    "url"             : "opc.tcp://0.0.0.0:4840",
    "config_ods"      : "configuration_freeopcua.ods",
    "fmu_filename"    : "Test_DTCONEDAR_linux.fmu",
    "stop_time"       : 10.0,
    "step_size"       : 0.5,
    "record"          : false,
    "record_interval"    : 5.0,
    "record_variables"   : [],
    "enable_send"        : true
}
```

Optional parameters can be modified:

- stop_time
- step_size
- record = True, if we want to save the results of the simulation
- record_interval. Sometimes the simulation step_size is small and a the size of the results file can be too big. For instance, if the simulation step_size is 0.01 seconds, we can increase the record_interval so as to reduce the result file size.
- record_variables: we can specify the list of variables that we want to record.
- enable_send = True, if we want to send results to the OPC UA Server.

Lifecycle Phases

The lifecycles that are covered include:

Lifecycle Phase	Completed Tasks
Install	Installs Open Modelica, Python 3.10 and the required pip dependencies
Create	Create FMU from Open Modelica file
Execute	Run OPC UA mock server and normal OPC-UA client
Clean	Delete the temporary files

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital twins/opc-ua-waterplant
```

If required, change the execute permission of lifecycle scripts.

```
chmod +x lifecycle/*
```

Now, run the following scripts:

INSTALL

Installs Open Modelica, Python 3.10 and the required pip dependencies

```
lifecycle/install
```

CREATE

Create `Test_DTCONEDAR.fmu` co-simulation model from `Test_DTCONEDAR.mo open modelica file.

```
lifecycle/create
```

EXECUTE

Start the mock OPC UA server in the background. Run the OPC UA client.

```
lifecycle/execute
```

CLEAN

Remove the temporary files created by Open Modelica and output files generated by OPC UA client.

```
lifecycle/clean
```

References

More explanation about this example is available at:


```
Royo, L., Labarías, A., Arnau, R., Gómez, A., Ezquerra, A., Cilla, I., &
Díez-Antoñazas, L. (2023). Improving energy efficiency in the tertiary
treatment of Alguazas WWTP (Spain) by means of digital twin development
based on Physics modelling . Cambridge Open Engage.
[doi:10.33774/coe-2023-1vjcw](https://doi.org/10.33774/coe-2023-1vjcw)
```

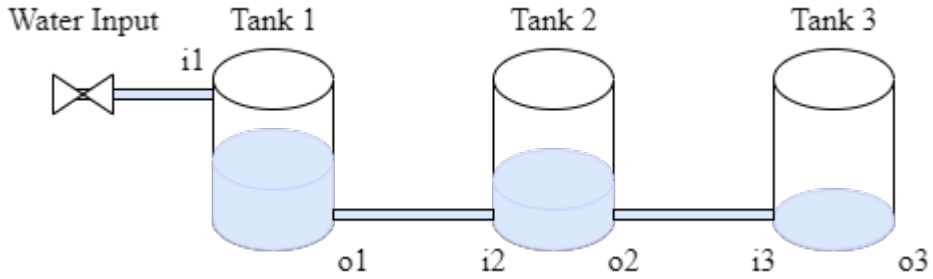

Acknowledgements

The work on this example was done in a project subsidised thanks to the support grants for Agrupación Empresarial Innovadora (AEI) of the Ministry of Industry, Trade and Tourism (MINCOTUR) with the aim of improving the competitiveness of small and medium-sized enterprises within the Recovery, Transformation and Resilience Plan (PRTR) financed by the Next Generation EU (NGEU) funds, with Grant Number: AEI-010500-2022b-196.

2.8.7 Three-Tank System Digital Twin

Overview

The three-tank system is a simple case study allows us to represent a system that is composed of three individual components that are coupled in a cascade as follows: The first tank is connected to the input of the second tank, and the output of the second tank is connected to the input of the third tank.



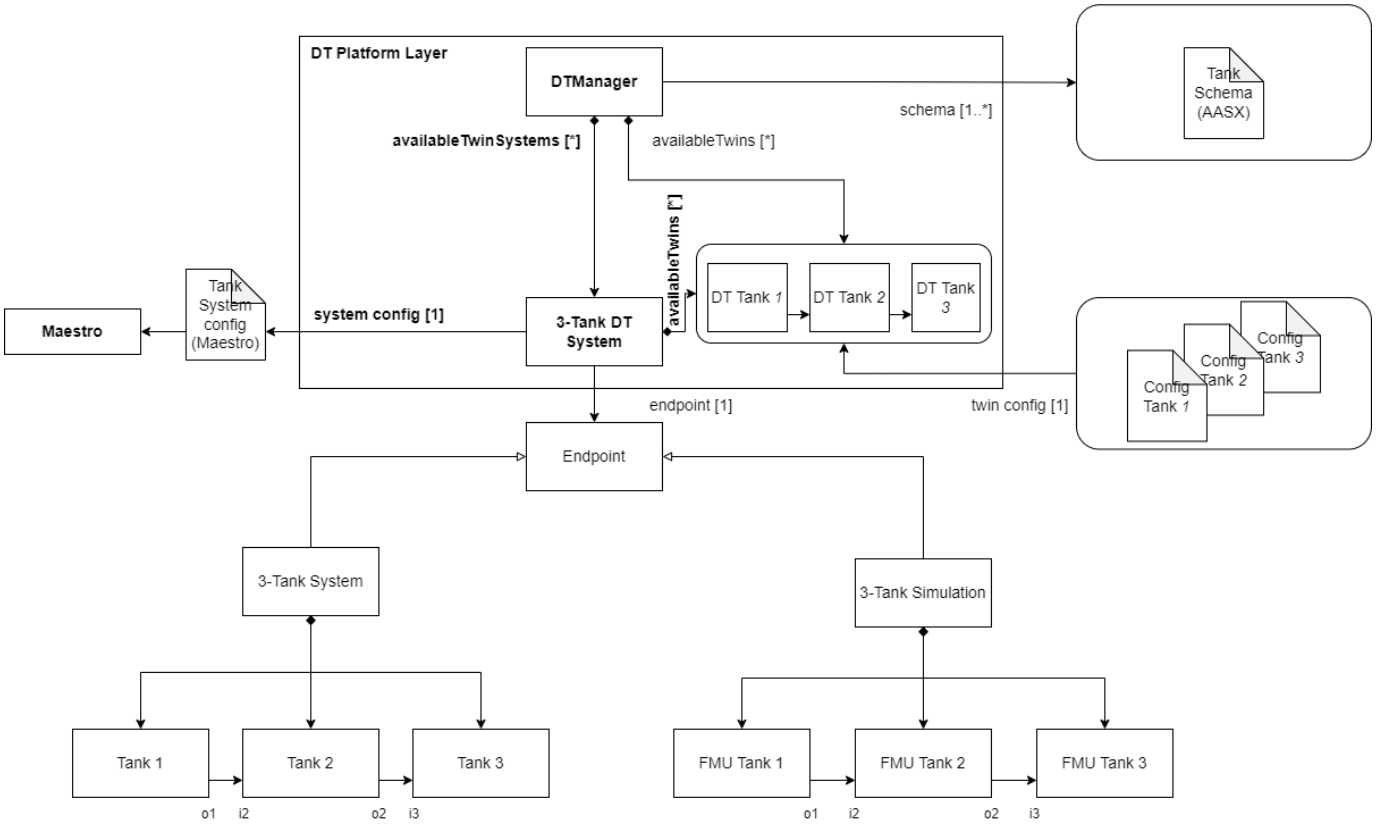
This example contains only the simulated components for demonstration purposes; therefore, there is no configuration for the connection with the physical system.

The three-tank system case study is managed using the `DTManager`, which is packed as a jar library in the tools, and run from a java main file. The `DTManager` uses Maestro as a slave for co-simulation, so it generates the output of the co-simulation.

The main file can be changed according to the application scope, i.e., the `/workspace/examples/tools/three-tank/TankMain.java` can be manipulated to get a different result.

The `/workspace/examples/models/three-tank/` folder contains the `Linear.fmu` file, which is a non-realistic model for a tank with input and output and the `TankSystem.aasx` file for the schema representation with Asset Administration Shell. The three instances use the same `.fmu` file and the same schema due to being of the same object class. The `DTManager` is in charge of reading the values from the co-simulation output.

Example Structure



Digital Twin Configuration

This example uses two models, two tools, one data, and one script. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Model	Linear.fmu	Private	No
	TankSystem.aasx	Private	No
Tool	DTManager-0.0.1-Maestro.jar (wraps Maestro)	Common	Yes
	maestro-2.3.0-jar-with-dependencies.jar (used by DTManager)	Common	Yes
	TankMain.java (main script)	Private	No
Data	outputs.csv	Private	No

This DT has multiple configuration files. The *coe.json* and *multimodel.json* are used by Maestro tool. The *tank1.conf*, *tank2.conf* and *tank3.conf* are the config files for three different instances of one model (Linear.fmu).

Lifecycle Phases

The lifecycles that are covered include:

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool
Execute	The DT Manager executes the three-tank digital twin and produces output in `data/three-tank/output` directory
Terminate	Terminating the background processes and cleaning up the output

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital twins/three-tank
```

If required, change the execute permission of lifecycle scripts you need to execute, for example:

```
chmod +x lifecycle/create
```

Now, run the following scripts:

CREATE

Installs Open Java Development Kit 11 and pip dependencies. Also creates `DTManager` tool (DTManager-0.0.1-Maestro.jar) from source code.

```
lifecycle/create
```

EXECUTE

Execute the three-tank digital twin using DTManager. DTManager in-turn runs the co-simulation using Maestro. Generates the co-simulation output.csv file at `/workspace/examples/data/three-tank/output`.

```
lifecycle/execute
```

TERMINATE

Stops the Maestro running in the background. Also stops any other jvm process started during **execute** phase.

```
lifecycle/terminate
```

CLEAN

Removes the output generated during execute phase.

```
lifecycle/terminate
```

Examining the results

Executing this Digital Twin will generate a co-simulation output, but the results can also be monitored from updating the `/workspace/examples/tools/three-tank/TankMain.java` with a specific set of `getAttributeValue` commands, such as shown in the code.

That main file enables the online execution of the Digital Twin and its internal components.

The output of the co-simulation is generated to the `/workspace/examples/data/three-tank/output` folder.

In the default example, the co-simulation is run for 10 seconds in steps of 0.5 seconds. This can be modified for a longer period and different step size. The output stored in `outputs.csv` contains the level, in/out flow, and leak values.

No data from the physical twin are generated/used.

References

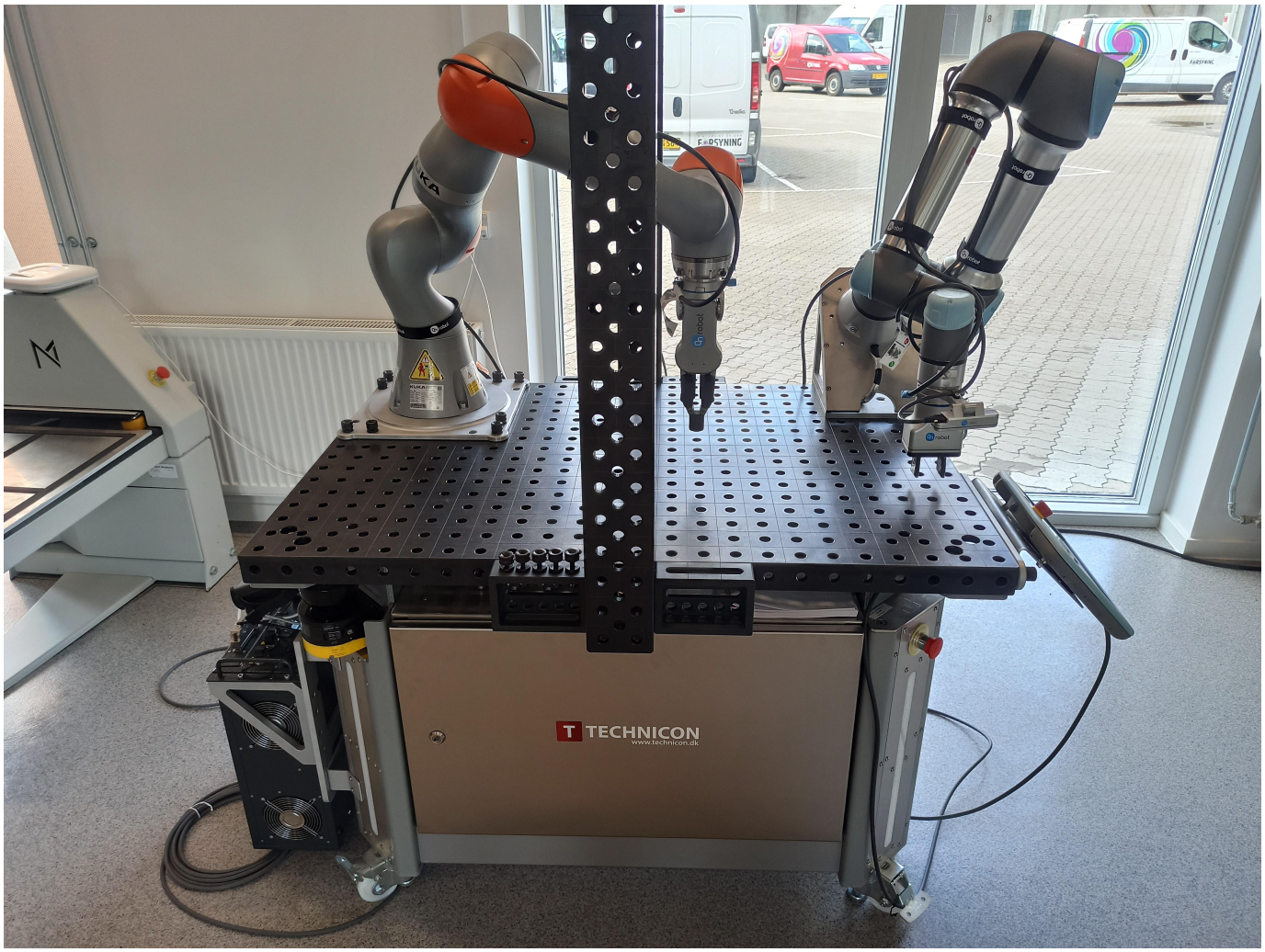
More information about the DT Manager is available at:

```
D. Lehner, S. Gil, P. H. Mikkelsen, P. G. Larsen and M. Wimmer,
"An Architectural Extension for Digital Twin Platforms to Leverage
Behavioral Models," 2023 IEEE 19th International Conference on
Automation Science and Engineering (CASE), Auckland, New Zealand,
2023, pp. 1-8, doi: 10.1109/CASE56687.2023.10260417.
```

2.8.8 Flex Cell Digital Twin with Two Industrial Robots

Overview

The flex-cell Digital Twin is a case study with two industrial robotic arms, a UR5e and a Kuka LBR iiwa 7, working in a cooperative setting on a manufacturing cell.



The case study focuses on the robot positioning in the discrete cartesian space of the flex-cell working space. Therefore, it is possible to send (X,Y,Z) commands to both robots, which refer to the target hole and height they want should move to.

The flex-cell case study is managed using the `TwinManager` (formerly `DT Manager`), which is packed as a jar library in the tools, and run from a java main file.

The `TwinManager` uses Maestro as a slave for co-simulation, so it generates the output of the co-simulation and can interact with the real robots at the same time (with the proper configuration and setup). The mainfile can be changed according to the application scope, i.e., the `/workspace/examples/tools/flex-cell/FlexCellDTaaS.java` can be manipulated to get a different result.

The `/workspace/examples/models/flex-cell/` folder contains the `.fmu` files for the kinematic models of the robotic arms, the `.urdf` files for visualization (including the grippers), and the `.aasx` files for the schema representation with Asset Administration Shell.

The case study also uses RabbitMQFMU to inject values into the co-simulation, therefore, there is the rabbitmqfmu in the models folder as well. Right now, RabbitMQFMU is only used for injecting values into the co-simulation, but not the other way around. The `TwinManager` is in charge of reading the values from the co-simulation output and the current state of the physical twins.

Example Structure

The example structure represents the components of the flex-cell DT implementation using the `TwinManager` architecture.

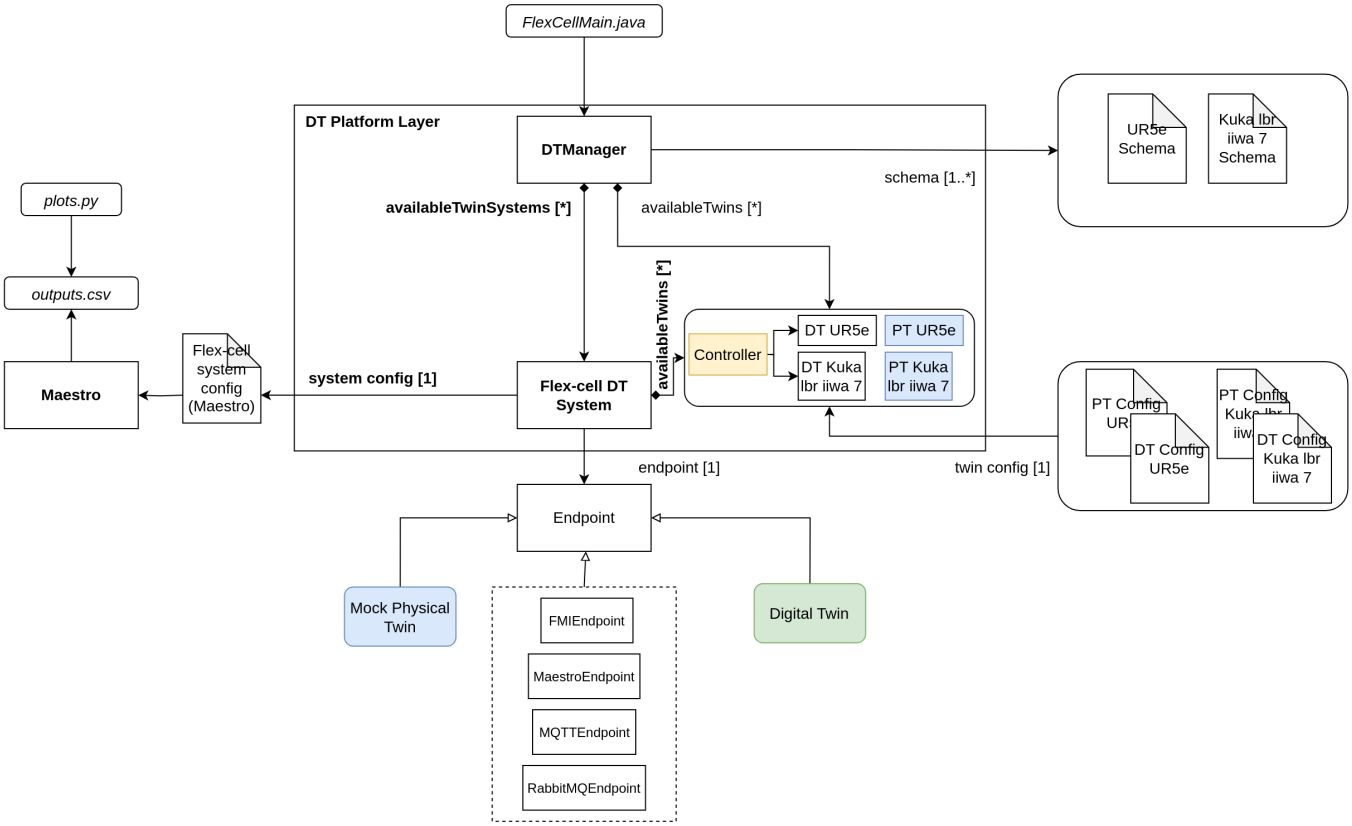
The TwinManager orchestrates the flex-cell DT via the *Flex-cell DT System*, which is composed of 2 smaller Digital Twins, namely, the *DT UR5e* and the *DT Kuka lbr iiwa 7*. The TwinManager also provides the interface for the Physical Twins, namely, *PT UR5e* and *PT Kuka lbr iiwa 7*. Each Physical Twin and Digital Twin System has a particular endpoint (with a different specialization), which is initialized from configuration files and data model (twin schema).

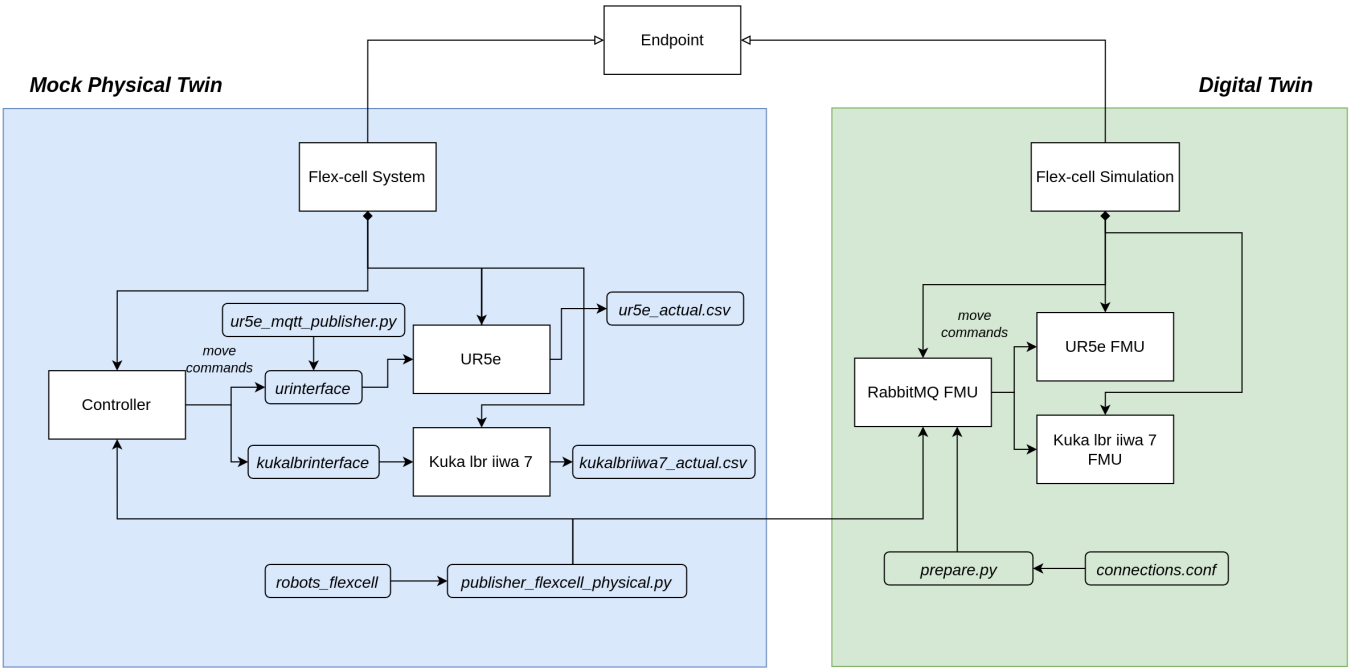
The current endpoints used in this implementation are:

Digital or Physical Twin	Endpoint
Flex-cell DT System	MaestroEndpoint
DT UR5e	FMIEndpoint
DT Kuka lbr iiwa 7	FMIEndpoint
PT UR5e	MQTTEndpoint and RabbitMQEndpoint
PT Kuka lbr iiwa 7	MQTTEndpoint and RabbitMQEndpoint

The Flex-cell DT System uses another configuration to be integrated with the Maestro co-simulation engine.

In the lower part, the Flex-cell System represents the composed physical twin, including the two robotic arms and controller and the Flex-cell Simulation is the mock-up representation for the real system, which is implemented by FMU blocks and their connections.





Digital Twin Configuration

This example uses seven models, five tools, six data files, two functions, and one script. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Model	kukalbriiwa_model.fmu	Private	No
	kuka_irw_gripper_rg6.urdf	Private	No
	kuka.aasx	Private	No
	ur5e_model.fmu	Private	No
	ur5e_gripper_2fg7.urdf	Private	No
	ur5e.aasx	Private	No
	rmq-vhost.fmu	Private	Yes
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes
	TwinManagerFramework-0.0.2.jar	Private	Yes
	urinterface (installed with pip)	Private	No
	kukalbrinterface	Private	No
	robots_flexcell	Private	No
	FlexCellDTaaS.java (main script)	Private	No
Data	publisher-flexcell-physical.py	Private	No
	ur5e_mqtt_publisher.py	Private	No
	connections.conf	Private	No
	outputs.csv	Private	No
	kukalbriiwa7_actual.csv	Private	No
	ur5e_actual.csv	Private	No
Function	plots.py	Private	No
	prepare.py	Private	No

Lifecycle Phases

The lifecycles that are covered include:

1. Installation of dependencies in the create phase.
2. Preparing the credentials for connections in the prepare phase.
3. Execution of the experiment in the execution phase.
4. Saving experiments in the save phase.
5. Plotting the results of the co-simulation and the real data coming from the robots in the analyze phase.
6. Terminating the background processes and cleaning up the outputs in the termination phase.

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool, Compiles source code of TwinManager to create a usable jar package (used as tool)
Prepare	Takes the RabbitMQ and MQTT credentials in connections.conf file and configures different assets of DT.
Execute	The TwinManager executes the flex-cell DT and produces output in `data/flex-cell/output` directory
Save	Save the experimental results
Analyze	Uses plotting functions to generate plots of co-simulation results
Terminate	Terminating the background processes
Clean	Cleans up the output data

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital_twins/flex-cell
```

If required, change the execute permission of lifecycle scripts you need to execute, for example:

```
chmod +x lifecycle/create
```

This example requires Java 11. The **create** script installs Java 11; however if you have already installed other Java versions, your default *java* might be pointing to another version. You can check and modify the default version using the following commands.

```
java -version
update-alternatives --config java
```

Now, run the following scripts:

CREATE

Installs Open Java Development Kit 11 and a python virtual environment with pip dependencies. Also builds the `TwinManager` tool (TwinManagerFramework-0.0.2.jar) from source code.

```
lifecycle/create
```

PREPARE

Configure different assets of DT with these credentials. The `functions/flex-cell/prepare.py` script is used for this purpose. The only thing needed to set up the connection is to update the file `/workspace/examples/data/flex-cell/input/connections.conf` with the connection parameters for MQTT and RabbitMQ and then execute the `prepare` script.

```
lifecycle/prepare
```

The following files are updated with the configuration information:

1. `/workspace/examples/digital_twins/flex-cell/kuka_actual.conf`
2. `/workspace/examples/digital_twins/flex-cell/ur5e_actual.conf`
3. `/workspace/examples/data/flex-cell/input/publisher-flexcell-physical.py`
4. `modelDescription.xml` for the RabbitMQFMU require special credentials to connect to the RabbitMQ and the MQTT brokers.

EXECUTE

Execute the flex-cell digital twin using TwinManager. TwinManager in-turn runs the co-simulation using Maestro. Generates the co-simulation output.csv file at `/workspace/examples/data/flex-cell/output`. The execution needs to be stopped with `control + c` since the `TwinManager` runs the application in a non-stopping loop.

```
lifecycle/execute
```

SAVE

Each execution of the DT is treated as a single run. The results of one execution are saved as time-stamped co-simulation output file in The TwinManager executes the flex-cell digital twin and produces output in `data/flex-cell/output/saved_experiments` directory.

```
lifecycle/save
```

The `execute` and `save` scripts can be executed in that order any number of times. A new file `data/flex-cell/output/saved_experiments` directory with each iteration.

ANALYZE

There are dedicated plotting functions in `functions/flex-cell/plots.py`. This script plots the co-simulation results against the recorded values from the two robots.

```
lifecycle/analyze
```

TERMINATE

Stops the Maestro running in the background. Also stops any other jvm process started during **execute** phase.

```
lifecycle/terminate
```

CLEAN

Removes the output generated during execute phase.

```
lifecycle/clean
```

Examining the results

Executing this Digital Twin will generate a co-simulation output, but the results can also be monitored from updating the `/workspace/examples/tools/flex-cell/FlexCellDTaaS.java` with a specific set of `getAttributeValue` commands, such as shown in the code. That main file enables the online execution and comparison on Digital Twin and Physical Twin at the same time and at the same abstraction level.

The output is generated to the `/workspace/examples/data/flex-cell/output` folder. In case a specific experiments is to be saved, the **save** lifecycle script stores the co-simulation results into the `/workspace/examples/data/flex-cell/output/saved_experiments` folder.

In the default example, the co-simulation is run for 11 seconds in steps of 0.2 seconds. This can be modified for a longer period and different step size. The output stored in `outputs.csv` contains the joint position of both robotic arms and the current discrete (X,Y,Z) position of the TCP of the robot. Additional variables can be added, such as the discrete (X,Y,Z) position of the other joints.

When connected to the real robots, the tools `urinterface` and `kukalbrinterface` log their data at a higher sampling rate.

References

The RabbitMQ FMU github repository contains complete documentation and source code of the **rmq-vhost.fmu**.

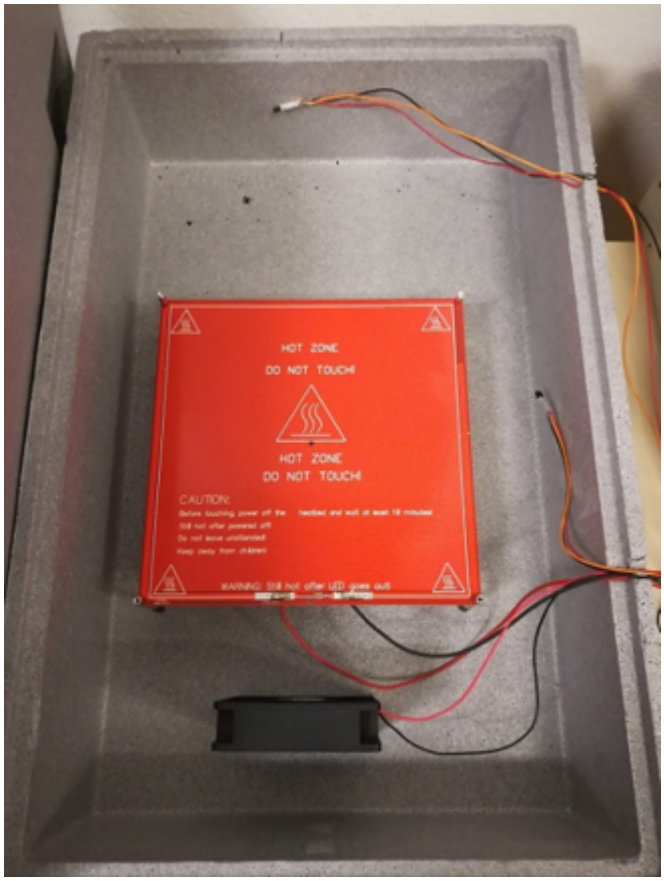
More information about the TwinManager (formerly DT Manager) and the case study is available in:

1. D. Lehner, S. Gil, P. H. Mikkelsen, P. G. Larsen and M. Wimmer, "An Architectural Extension for Digital Twin Platforms to Leverage Behavioral Models," 2023 IEEE 19th International Conference on Automation Science and Engineering (CASE), Auckland, New Zealand, 2023, pp. 1-8, doi: 10.1109/CASE56687.2023.10260417.
2. S. Gil, P. H. Mikkelsen, D. Tola, C. Schou and P. G. Larsen, "A Modeling Approach for Composed Digital Twins in Cooperative Systems," 2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA), Sinaia, Romania, 2023, pp. 1-8, doi: 10.1109/ETFA54631.2023.10275601.
3. S. Gil, C. Schou, P. H. Mikkelsen, and P. G. Larsen, “Integrating Skills into Digital Twins in Cooperative Systems,” in 2024 IEEE/SICE International Symposium on System Integration (SII), 2024, pp. 1124–1131, doi: 10.1109/SII58957.2024.10417610.

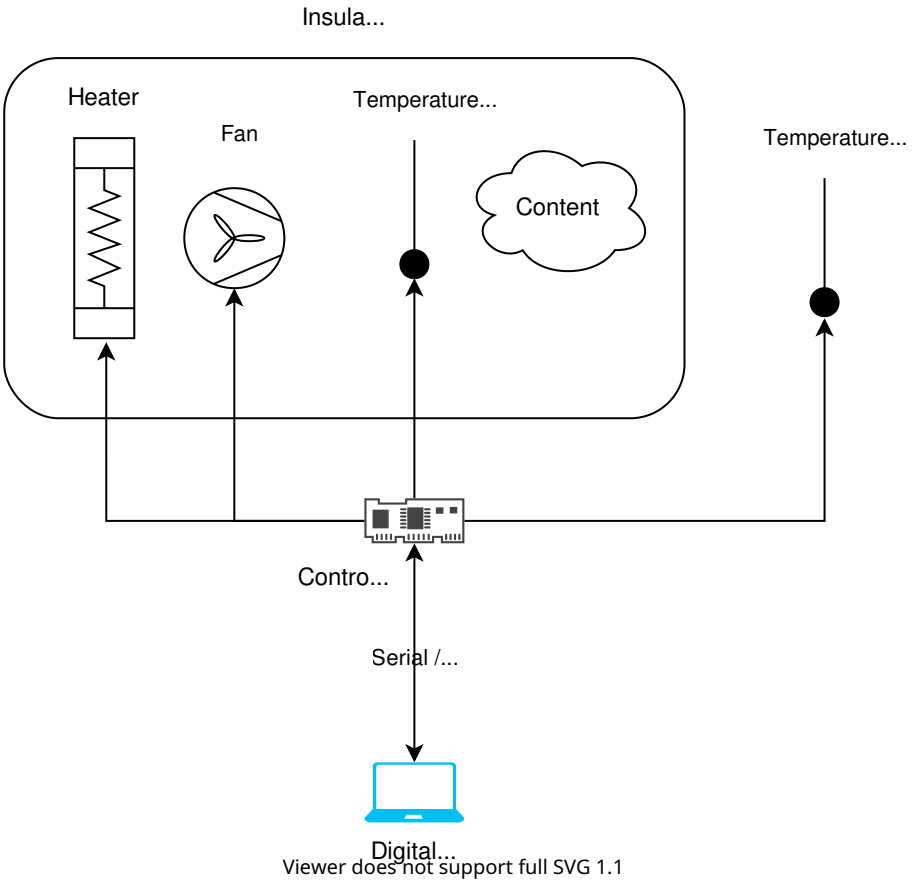
2.8.9 Incubator Digital Twin

Overview

This is a case study of an Incubator with the purpose of understanding the steps and processes involved in developing a digital twin system. This incubator is an insulated container with the ability to keep a temperature and heat, but not cool. A picture of the incubator is given below.



The overall purpose of the system is to reach a certain temperature within a box and keep the temperature regardless of content. An overview of the system can be seen below:



The system consists of:

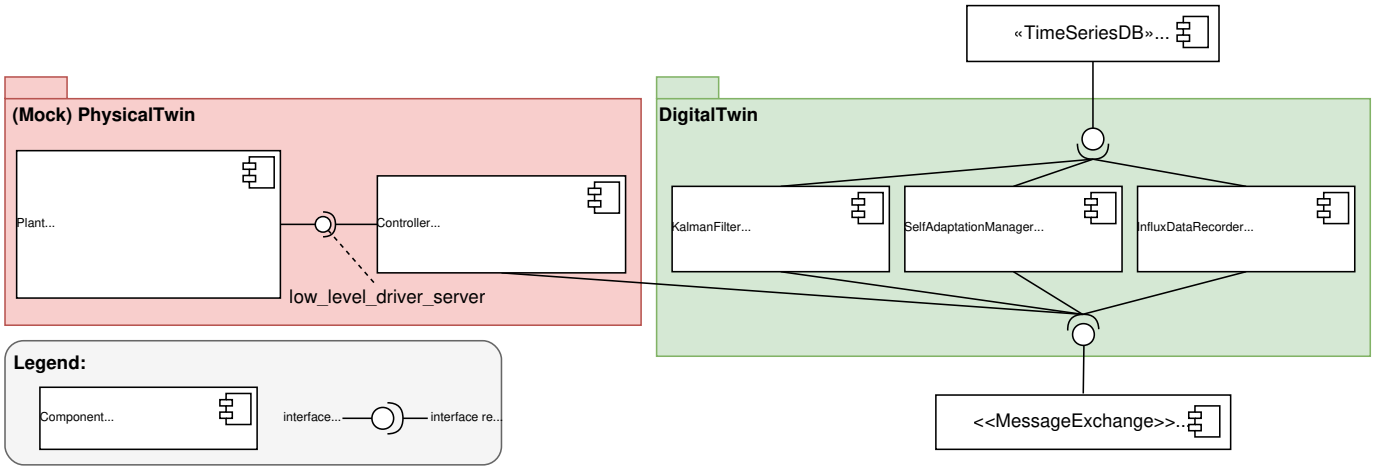
- 1x styrofoam box in order to have an insulated container
- 1x heat source to heat up the content within the Styrofoam box.
- 1x fan to distribute the heating within the box
- 2x temperature sensor to monitor the temperature within the box
- 1x temperature Sensor to monitor the temperature outside the box
- 1x controller to actuate the heat source and the fan and read sensory information from the temperature sensors, and communicate with the digital twin.

The original repository for the example can be found: Original repository. This trimmed version of the codebase does not have the following:

- docker support
- tests
- datasets

The original repository contains the complete documentation of the example, including the full system architecture, instructions for running with a physical twin, and instructions for running a 3D visualization of the incubator.

Digital Twin Structure



This diagrams shows the main components and the interfaces they use to communicate. All components communicate via the RabbitMQ message exchange, and the data is stored in a time series database. The RabbitMQ and InfluxDB are platform services hosted by the DTaaS.

The Incubator digital twin is a pre-packaged digital twin. It can be used as is or integrated with other digital twins.

The mock physical twin is executed from `incubator/mock_plant/real_time_model_solver.py` script.

Digital Twin Configuration

This example uses a plethora of Python scripts to run the digital twin. By default it is configured to run with a mock physical twin. Furthermore, it depends on a RabbitMQ and an InfluxDB instances.

There is one configuration file: `simulation.conf`. The RabbitMQ and InfluxDB configuration parameters need to be updated.

Lifecycle Phases

The lifecycles that are covered include:

Lifecycle Phase	Completed Tasks
Create	Potentially updates the system and installs Python dependencies
Execute	Executes the Incubator digital twin and produces output in the terminal and in *incubator/log.log*.
Clean	Removes the log file.

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital_twins/incubator
```

If required, change the execute permission of lifecycle scripts you need to execute, for example:

```
chmod +x lifecycle/create
```

Now, run the following scripts:

CREATE

Potentially updates the system and installs Python dependencies.

```
lifecycle/create
```

EXECUTE

Executes the Incubator digital twin with a mock physical twin. Pushes the results in the terminal, *incubator/log.log*, and in InfluxDB.

```
lifecycle/execute
```

CLEAN

Removes the output log file.

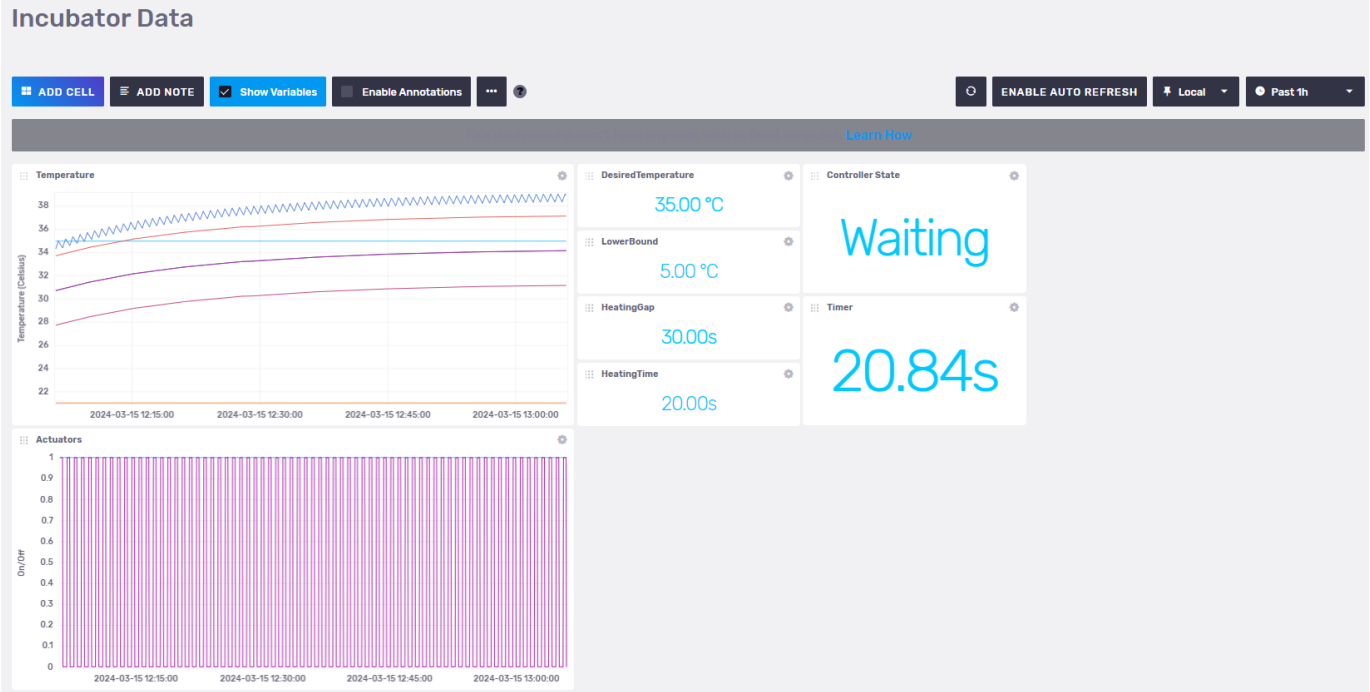
```
lifecycle/clean
```

Examining the results

After starting all services successfully, the controller service will start producing output that looks like the following:

```
time           execution_interval  elapsed  heater_on  fan_on   room   box_air_temperature  state
19/11 16:17:59  3.00                0.01     True       False    10.70  19.68                Heating
19/11 16:18:02  3.00                0.03     True       True     10.70  19.57                Heating
19/11 16:18:05  3.00                0.01     True       True     10.70  19.57                Heating
19/11 16:18:08  3.00                0.01     True       True     10.69  19.47                Heating
19/11 16:18:11  3.00                0.01     True       True     10.69  19.41                Heating
```

An InfluxDB dashboard can be setup based on `incubator/digital_twin/data_access/influxdbserver/dashboards/incubator_data.json`. If the dashboard on the InfluxDB is setup properly, the following visualization can be seen:



References

📋 Forked from: Incubator repository with commit ID: 989ccf5909a684ad26a9c3ec16be2390667643aa

To understand what a digital twin is, we recommend you read/watch one or more of the following resources:

1. Feng, Hao, Cláudio Gomes, Casper Thule, Kenneth Lausdahl, Alexandros Iosifidis, and Peter Gorm Larsen. “Introduction to Digital Twin Engineering.” In 2021 Annual Modeling and Simulation Conference (ANNSIM), 1–12. Fairfax, VA, USA: IEEE, 2021. https://doi.org/10.23919/ANNSIM52504.2021.9552135.
2. Video recording of presentation by Claudio Gomes

2.8.10 Firefighter Mission in a Burning Building

In an firefighter mission it is important to monitor the oxygen levels of each firefighters Self Contained Breating Aparatus (SCBA) in context of their mission.

Physical Twin Overview

Image: Schematic overview of a firefighter mission. Note the mission commander on the lower left documenting the air supply pressure levels provided by radio communication from the firefighters inside and around the burning building. This image was created with the assistance of DALL·E.

We assume the following scenario:

- a set of firefighters work to extinguish a burning building
- they each use an SCBA with pressurised oxygen to breath
- a mission commander on the outside coordinates the efforts and surveills the oxygen levels

Digital Twin Overview

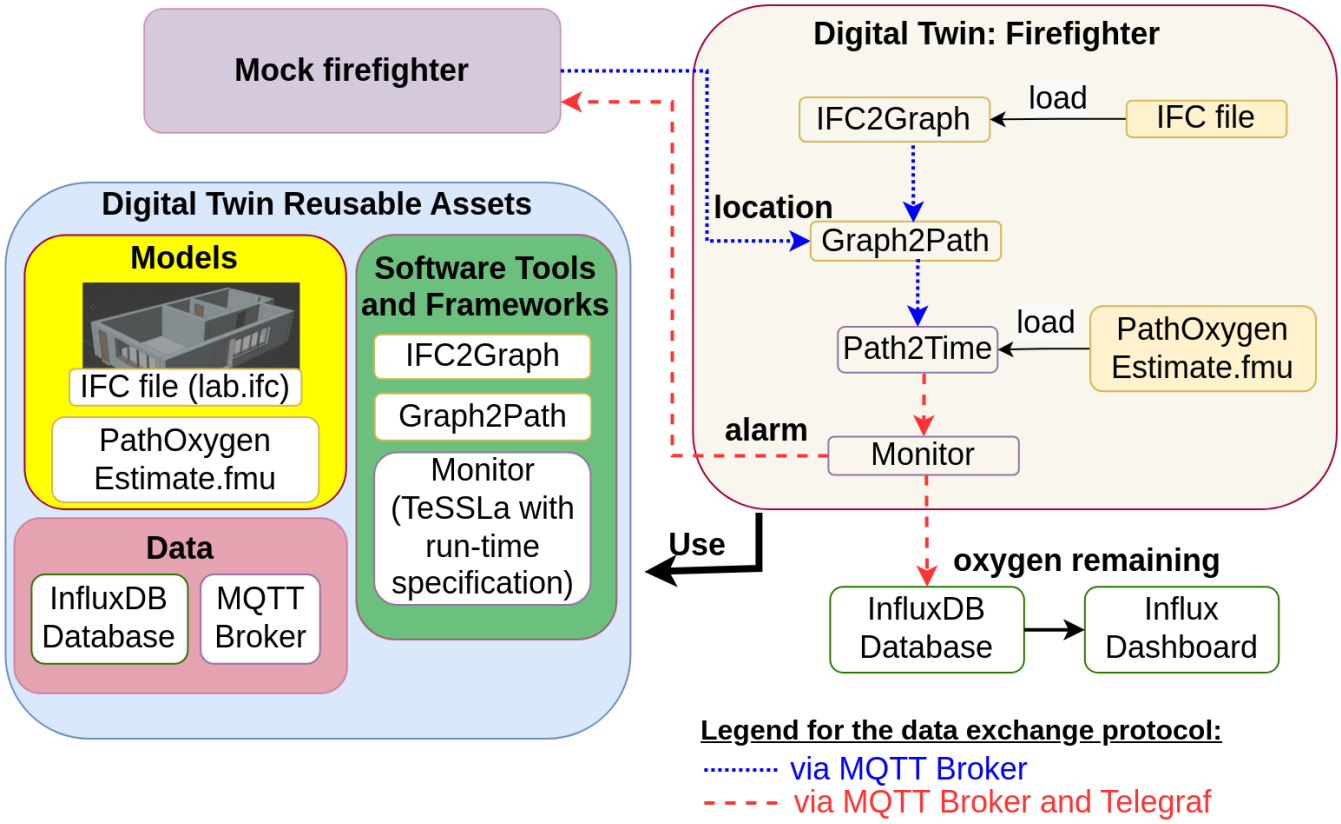
In this example a monitor is implemented, that calculates how much time the firefighers have left, until they need to leave the building. To that end, the inputs used are:

- 3D-model of the building in which the mission takes place,
- pressure data of a firefighters SCBA and
- firefighters location inside of the building

are used to estimate:

- the shortest way out,
- how much time this will need and
- how much time is left until all oxygen from the SCBA is used up.

The remaining mission time is monitored and the firefighter receive a warning if it drops under a certain threshold.



This example is an implementation of the the paper *Digital Twin for Rescue Missions--a Case Study* by Leucker et al.

QUICK CHECK

Before runnnig this example please make sure the following files are at the correct locations:

```
/workspace/examples/
   data/o5g/input/
      runTessla.sh
      sensorSimulation.py
      telegraf.conf

   models/
    lab.ifc
    makefmu.mos
      PathOxygenEstimate.mo

  tools/
    graphToPath.py
    ifc_to_graph
    pathToTime.py
    tessla-telegraf-connector/
    tessla-telegraf-connector/
      tessla.jar
      specification.tessla (run-time specification)

   digital_twins/o5g/
      main.py
      config
      lifecycle/ (scripts)
```

DIGITAL TWIN CONFIGURATION

All configuration for this example is contained in `digital_twins/o5g/config`.

To use the MQTT-Server, account information needs to be provided. The topics are set to their default values, which allow the DT to access the mock physical twins sensor metrics and to send back alerts.

```
export O5G_MQTT_SERVER=
export O5G_MQTT_PORT=
export O5G_MQTT_USER=
export O5G_MQTT_PASS=

export O5G_MQTT_TOPIC_SENSOR='vgiot/ue/metric'
export O5G_MQTT_TOPIC_AIR_PREDICTION='vgiot/dt/prediction'
export O5g_MQTT_TOPIC_ALERT='vgiot/dt/alerts'
```

This example uses InfuxDB as a data storage, which will need to be configured to use your Access data. The following configuration steps are needed:

- Log into the InfluxDB Web UI
- Obtain **org** name (is below your *username* in the sidebar)
- Create a data bucket if you don't have one already in `Load Data -> Buckets`
- Create an API access token in `Load Data -> API Tokens`, Copy and save this token somewhere immediately, you can not access it later!

```
export O5G_INFLUX_SERVER=
export O5G_INFLUX_PORT=
export O5G_INFLUX_TOKEN=
export O5G_INFLUX_ORG=
export O5G_INFLUX_BUCKET=
```

Lifecycle Phases

The lifecycles that are covered include:

Lifecycle Phase	Completed Tasks
Install	Installs Open Modelica, Rust, Telegraf and the required pip dependencies
Create	Create FMU from Open Modelica file
Execute	Execute the example in the background tmux terminal session
Terminate	Terminate the tmux terminal session running in the background
Clean	Delete the temporary files

Run the example

INSTALL

Run the install script by executing

```
lifecycle/install
```

This will install all the required dependencies from apt and pip, as well as Open Modelica, Rust, Telegraf and the required pip dependencies from their respective repos.

Create

Run the create script by executing

```
lifecycle/create
```

This will compile the modelica model to an Functional Mockup Unit (FMU) for the correct platform.

Exceute

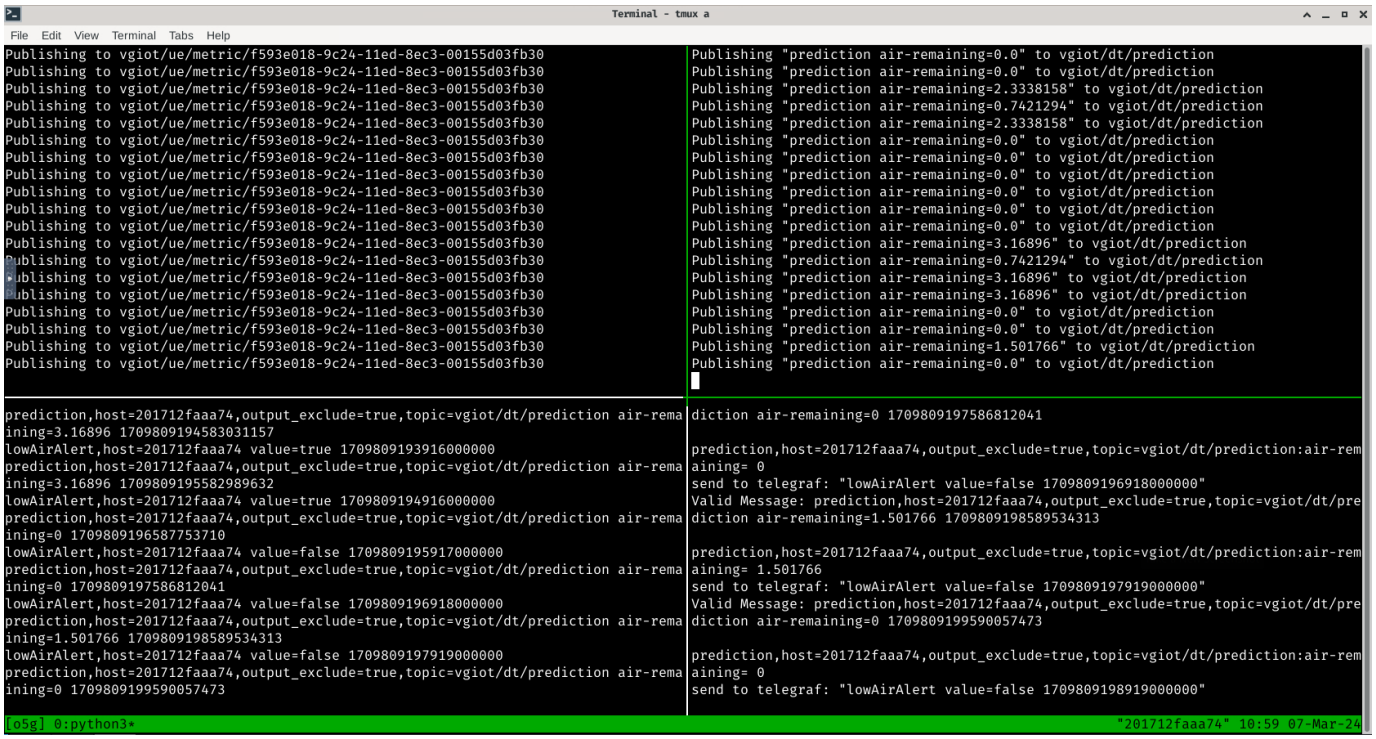
To run the Digital Twin execute

```
lifecycle/execute
```

This will start all the required components in a single tmux session called `o5g` in the background. To view the running Digital Twin attatch to this tmux session by executing

```
tmux a -t o5g
```

To detatch press `Ctrl-b` followed by `d`.



The *tmux* session contains 4 components of the digital twin:

Panel location	Purpose
Top Left	Sensor simulator generating random location and O2-level data
Top Right	Main Digital Twin receives the sensor data and calculates an estimate of how many minutes of air remain
Bottom Left	Telegraf to convert between different message formats, also displays all messages between components
Bottom Right	TeSSLa monitor raises an alarm, if the remaining time is to low.

Examine the Results

For additional mission awareness, we recommend utilising the Influx data visualisation. We provide a dashboard configuration in the file *influx-dashoard.json*. Log in to your Influx Server to import (usually port 8086). A screenshot of the dashboard is given here.

The data gets stored in `o5g->prediction->air-remaining->37ae3e4fb3ea->true->vgiot/dt/prediction` variable of the InfluxDB. In addition to importing dashboard configuration given above, it is possible to create your custom dashboards using the stored data.

Terminate

To stop the all components and close the *tmux* session execute

```
lifecycle/terminate
```

Clean

To remove temoporary files created during execution

```
lifecycle/clean
```

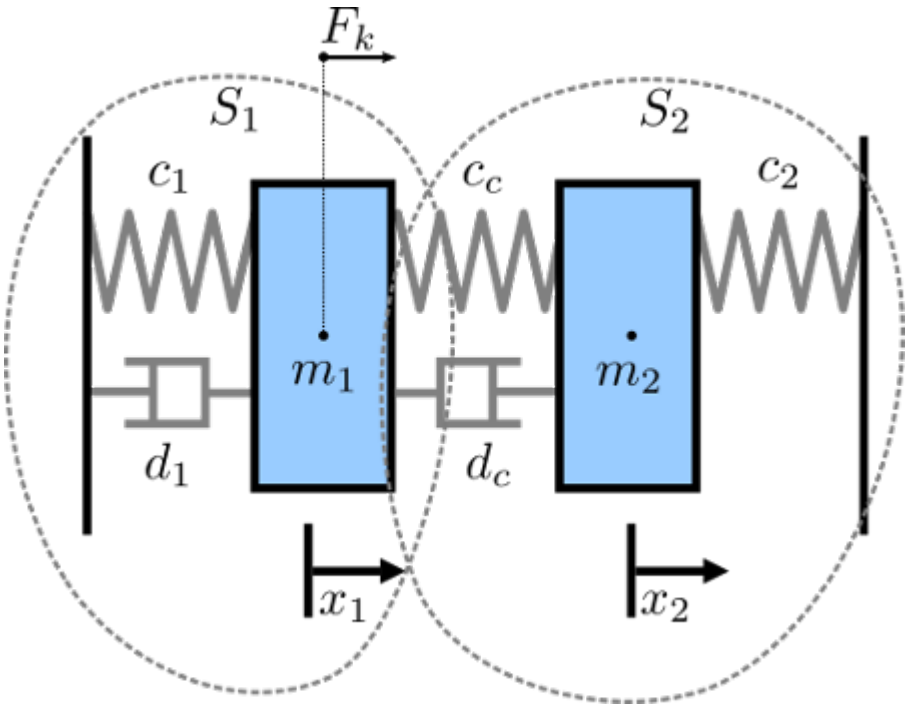
2.8.11 Mass Spring Damper with NuRV Runtime Monitor

Overview

This digital twin is derives from te Mass String Damper digital twin.

The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within DTaaS. This version of the example is expanded with a monitor generated by NuRV. More information about NuRV is available here.

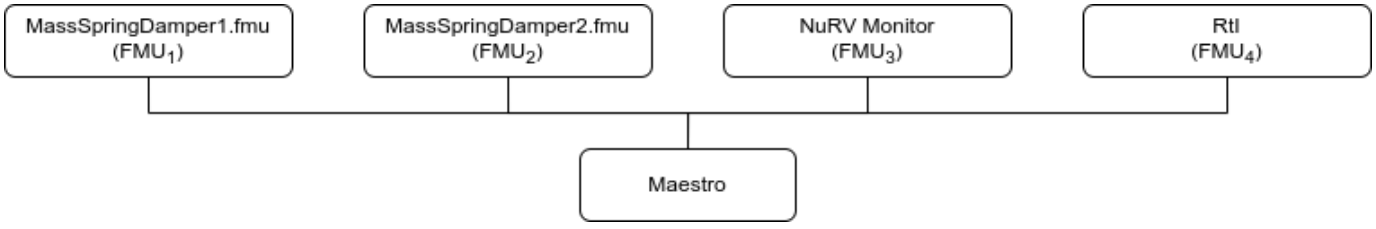
Example Diagram



Example Structure

There are two simulators included in the study, each representing a mass spring damper system. The first simulator calculates the mass displacement and speed of for a given force acting on mass . The second simulator calculates force given a displacement and speed of mass . By coupling these simulators, the evolution of the position of the two masses is computed.

Additionally, a monitor is inserted in the simulation to check at runtime whether the displacement of the two masses stays below a maximum threshold.



Digital Twin Configuration

This example uses two models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	MassSpringDamper1.fmu	Private	Yes
	MassSpringDamper2.fmu	Private	Yes
	m2.fmu	Private	No
	RtI.fmu	Private	Yes
Specification	m2.smv	Private	No
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The `co-sim.json` and `time.json` are two DT configuration files used for executing the digital twin. You can change these two files to customize the DT to your needs.

Lifecycle Phases

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool Generates and compiles the monitor FMU
Execute	Produces and stores output in data/mass-spring-damper-monitor/output directory
Clean	Clears run logs and outputs

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital_twins/mass-spring-damper-monitor
```

If required, change the execute permission of lifecycle scripts you need to execute, for example:

```
chmod +x lifecycle/create
```

Now, run the following scripts:

CREATE

- Installs Open Java Development Kit 17 in the workspace.
- Generates and compiles the monitor FMU from the NuRV specification

```
lifecycle/create
```

EXECUTE

Run the the Digital Twin. Since this is a co-simulation based digital twin, the Maestro co-simulation tool executes co-simulation using the two FMU models.

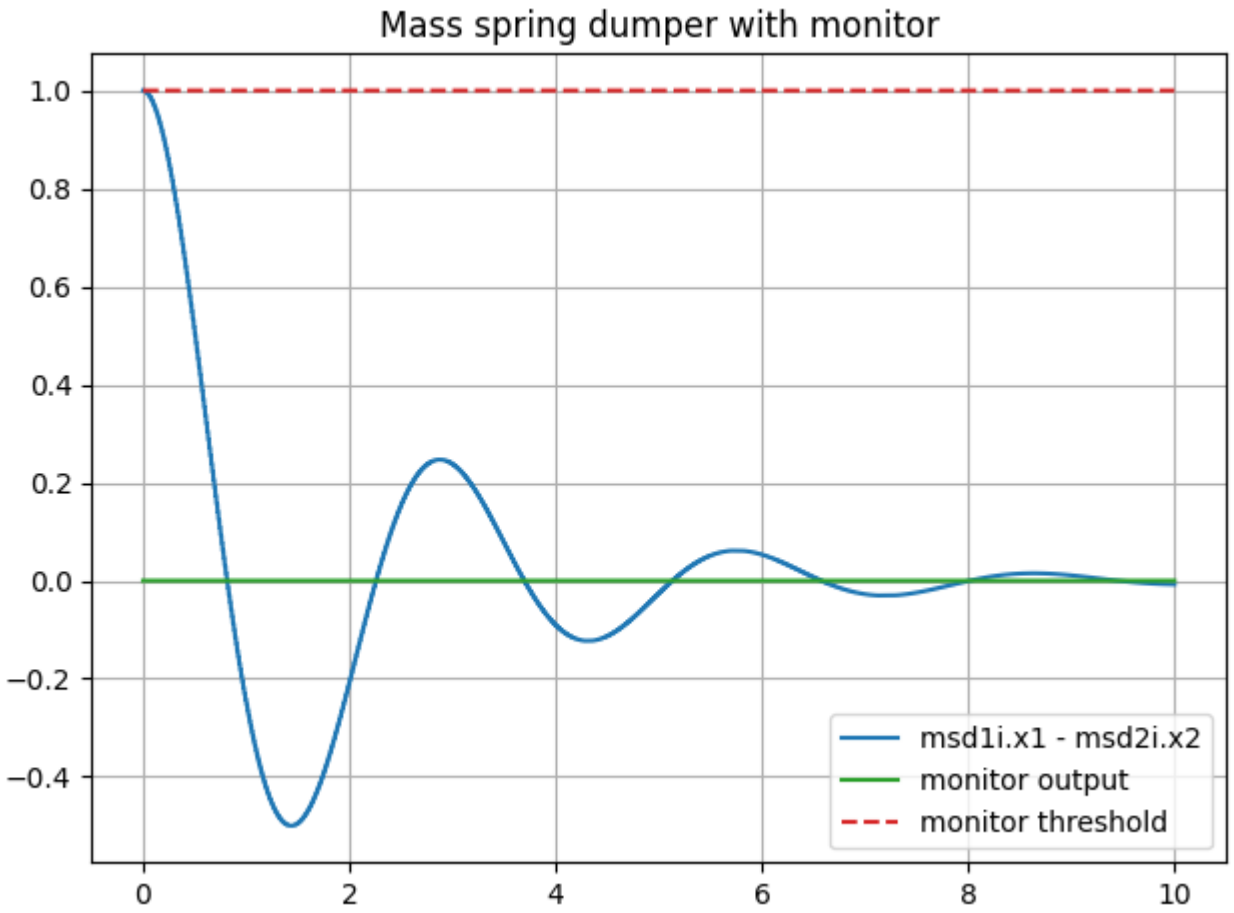
```
lifecycle/execute
```

ANALYZE PHASE

Process the output of co-simulation to produce a plot at: `/workspace/examples/data/mass-spring-damper-monitor/output/plots`.

```
lifecycle/analyze
```

A sample plot is given here.



In the plot, three color-coded indicators are used to represent different values. The blue line shows the distance between the two masses, while the green indicates the monitor's verdict. A red dashed line serves as a reference point, marking the distance checked by the monitor. Since the distance of the masses is always below the threshold, the output of the monitor is fixed to `unknown` (`0`).

Examine the results

The results can be found in the */workspace/examples/data/mass-spring-damper-monitor/output directory*.

You can also view run logs in the */workspace/examples/digital_twins/mass-spring-damper-monitor*.

TERMINATE PHASE

Terminate to clean up the debug files and co-simulation output files.

```
lifecycle/terminate
```

References

More information about co-simulation techniques and mass spring damper case study are available in:

```
Gomes, Cláudio, et al. "Co-simulation: State of the art."
arXiv preprint arXiv:1702.00686 (2017).
```

The source code for the models used in this DT are available in mass spring damper github repository.

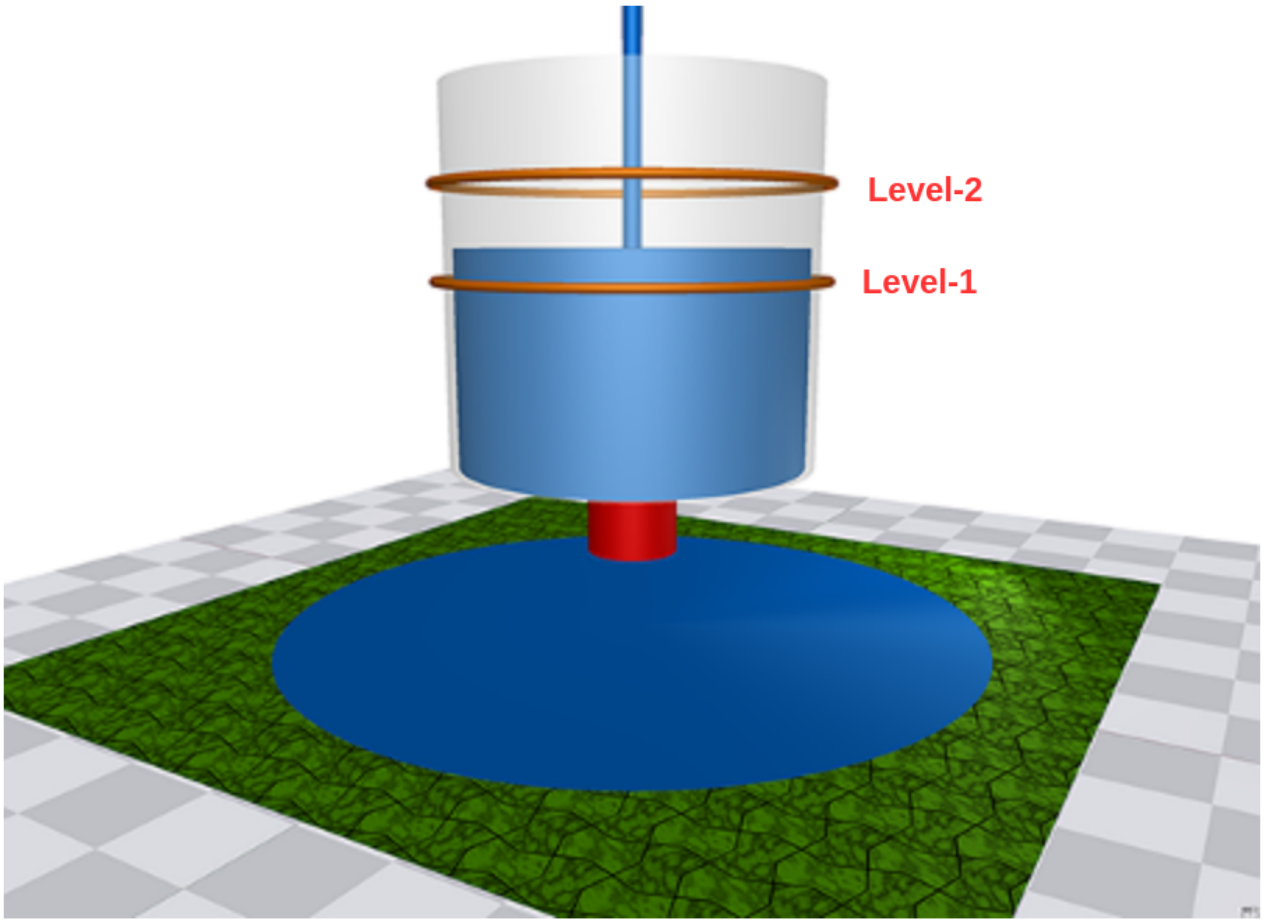
2.8.12 Water Tank Fault Injection with NuRV Runtime Monitor

Overview

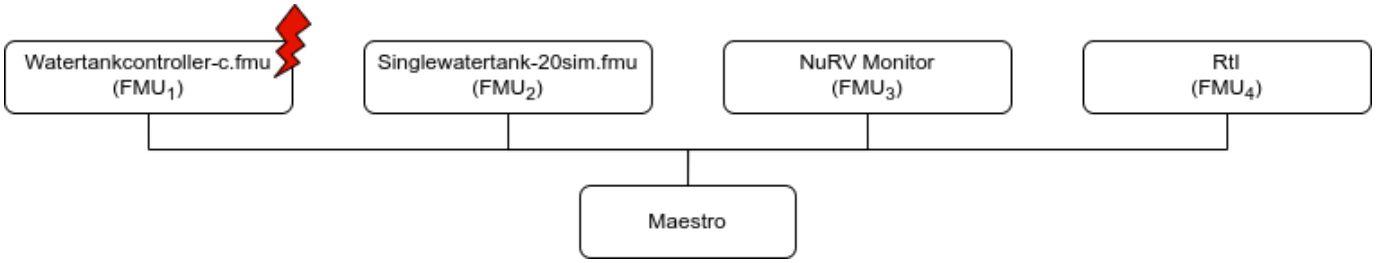
This example shows a fault injection (FI) enabled digital twin (DT). A live DT is subjected to simulated faults received from the environment. The simulated faults is specified as part of DT configuration and can be changed for new instances of DTs. This version of the example is expanded with a monitor generated by NuRV. More information about NuRV is available here.

In this co-simulation based DT, a watertank case-study is used; co-simulation consists of a tank and controller. The goal of which is to keep the level of water in the tank between `Level-1` and `Level-2`. The faults are injected into output of the water tank controller (**Watertankcontroller-c.fmu**) from 12 to 20 time units, such that the tank output is closed for a period of time, leading to the water level increasing in the tank beyond the desired level (`Level-2`). Additionally, a monitor is inserted in the simulation to check at runtime whether the level of the water stays below a maximum threshold.

Example Diagram



Example Structure



Digital Twin Configuration

This example uses two models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	watertankcontroller-c.fmu	Private	Yes
	singlewatertank-20sim.fmu	Private	Yes
	m1.fmu	Private	No
	RtI.fmu	Private	Yes
Specification	m1.smv	Private	No
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The `multimodelFI.json` and `simulation-config.json` are two DT configuration files used for executing the digital twin. You can change these two files to customize the DT to your needs.

The faults are defined in **wt_fault.xml**.

Lifecycle Phases

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool Generates and compiles the monitor FMU
Execute	Produces and stores output in data/water_tank_FI_monitor/output directory
Clean	Clears run logs and outputs

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital_twins/water_tank_FI_monitor
```

If required, change the execute permission of lifecycle scripts you need to execute, for example:

```
chmod +x lifecycle/create
```

Now, run the following scripts:

CREATE

Installs Open Java Development Kit 17 and pip dependencies. The pandas and matplotlib are the pip dependencies installated. The monitor FMU from the NuRV specification is generated and compiled.

```
lifecycle/create
```

EXECUTE

Run the co-simulation. Generates the co-simulation output.csv file at `/workspace/examples/data/water_tank_FI_monitor/output`.

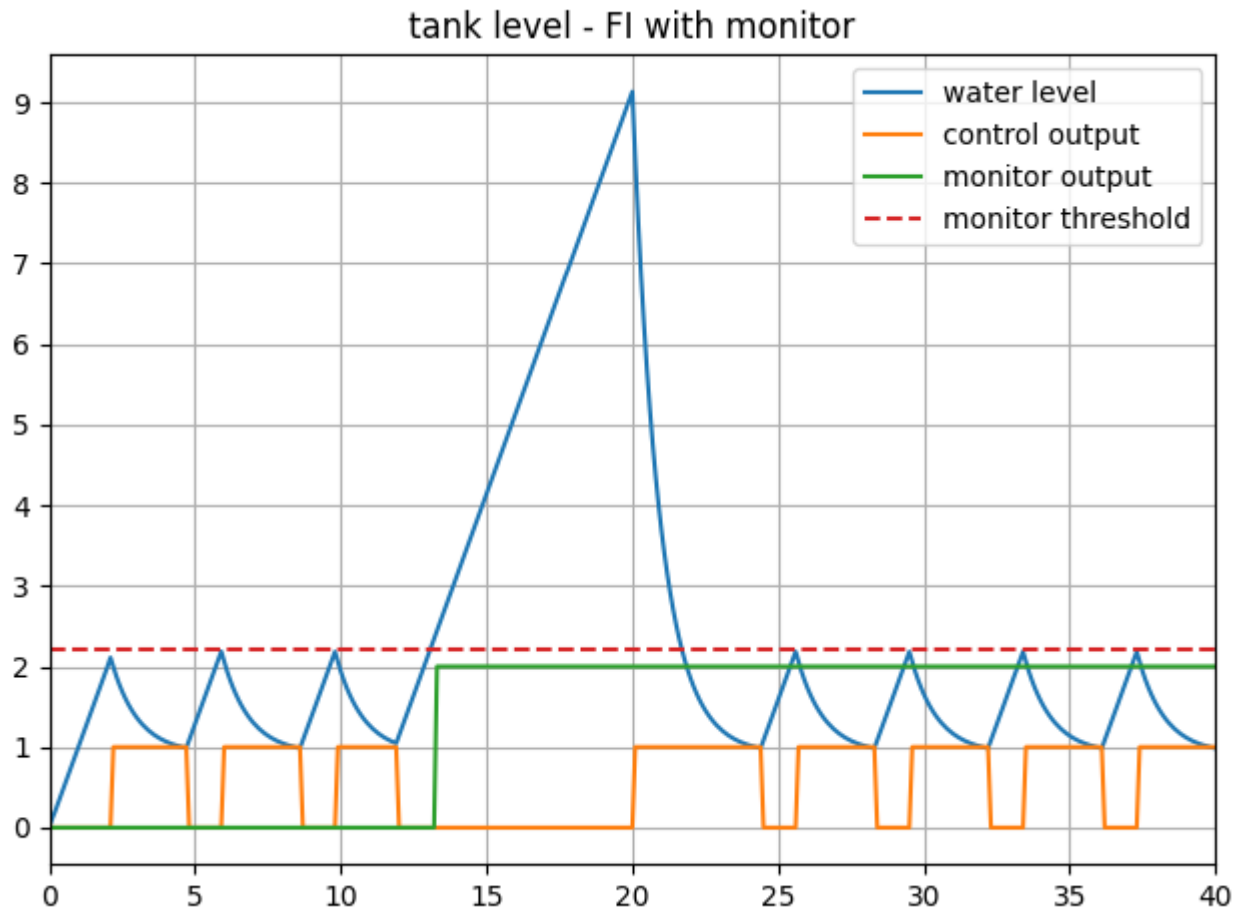
```
lifecycle/execute
```

ANALYZE PHASE

Process the output of co-simulation to produce a plot at: `/workspace/examples/data/water_tank_FI_monitor/output/plots/`.

```
lifecycle/analyze
```

A sample plot is given here.



In the plot, four color-coded indicators are used to represent different values. The blue line shows the water tank level, while orange represents the control output and green indicates the monitor's verdict. A red dashed line serves as a reference point, marking the level checked by the monitor. As the water level exceeds this threshold, the monitor's verdict changes from `unknown` (`0`) to `false` (`2`).

Examine the results

The results can be found in the */workspace/examples/data/water_tank_FI_monitor/output directory*.

You can also view run logs in the */workspace/examples/digital_twins/water_tank_FI_monitor*.

TERMINATE PHASE

Clean up the temporary files and delete output plot

```
lifecycle/terminate
```

References

More details on this case-study can be found in the paper:

```
M. Frasheri, C. Thule, H. D. Macedo, K. Lausdahl, P. G. Larsen and
L. Esterle, "Fault Injecting Co-simulations for Safety,"
2021 5th International Conference on System Reliability and Safety (ICSRS),
Palermo, Italy, 2021.
```

The fault-injection plugin is an extension to the Maestro co-orchestration engine that enables injecting inputs and outputs of FMUs in an FMI-based co-simulation with tampered values. More details on the plugin can be found in fault injection git repository. The source code for this example is also in the same github repository in a example directory.

2.8.13 Incubator Co-Simulation Digital Twin validation with NuRV Monitor

Overview

This example demonstrates how to validate some digital twin components using FMU monitors (in this example, the monitors are generated with NuRV[1]).

Simulated scenario

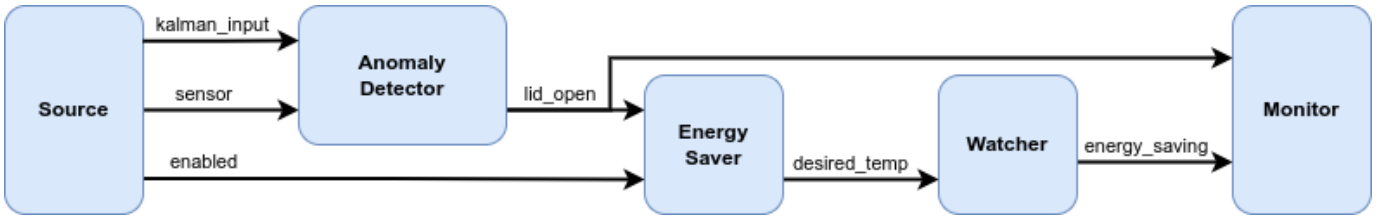
This example validates some components of the Incubator digital twin, by performing a simulation in which the commponents are wrapped inside FMUs, and are then inspected at runtime by some a FMU monitor generated by NuRV. Please note that the link to Incubator digital twin is only provided to know the details of the incubator physical twin. The digital twin (DT) presented here is a co-simulation DT of the Incubator.

The input data for the simulation is generated by a purpose-built FMU component named *source*, which supplies testing data to the *anomaly detector*, simulating an anomaly occurring at time t=60s. An additional component, *watcher*, is employed to verify whether the *energy saver* activates in response to an anomaly reported by the *anomaly detector*.

The output of the watcher is the passed to the monitor, which ensures that when an anomaly is detected, the *energy saver* activates within a maximum of three simulation cycles.

Example structure

A diagram depicting the logical software structure of the example can be seen below.



Digital Twin configuration

The example uses the following assets:

Asset Type	Names of Assets	Visibility	Reuse in other Examples
Models	anomaly_detection.fmu	Private	No
	energy_saver.fmu	Private	No
	Source.fmu	Private	No
	Watcher.fmu	Private	No
Specification	safe-operation.smv	Private	No
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The *safe-operation.smv* file contains the default monitored specification as described in the Simulated scenario section.

Lifecycle phases

The lifecycle phases for this example include:

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool Generates and compiles the monitor FMU
Execute	Produces and stores output in data/incubator-NuRV-monitor-validation/output directory
Clean	Clears run logs and outputs

If required, change the execute permissions of lifecycle scripts you need to execute. This can be done using the following command

```
chmod +x lifecycle/{script}
```

where {script} is the name of the script, e.g. *create*, *execute* etc.

Run the example

To run the example, change your present directory.

```
cd /workspace/examples/digital_twins/incubator-NuRV-monitor-validation
```

If required, change the execute permission of lifecycle scripts you need to execute, for example:

```
chmod +x lifecycle/create
```

Now, run the following scripts:

CREATE

- Installs Open Java Development Kit 17 in the workspace.
- Generates and compiles the monitor FMU from the NuRV specification

```
lifecycle/create
```

EXECUTE

Run the the Digital Twin. Since this is a co-simulation based digital twin, the Maestro co-simulation tool executes co-simulation using the FMU models.

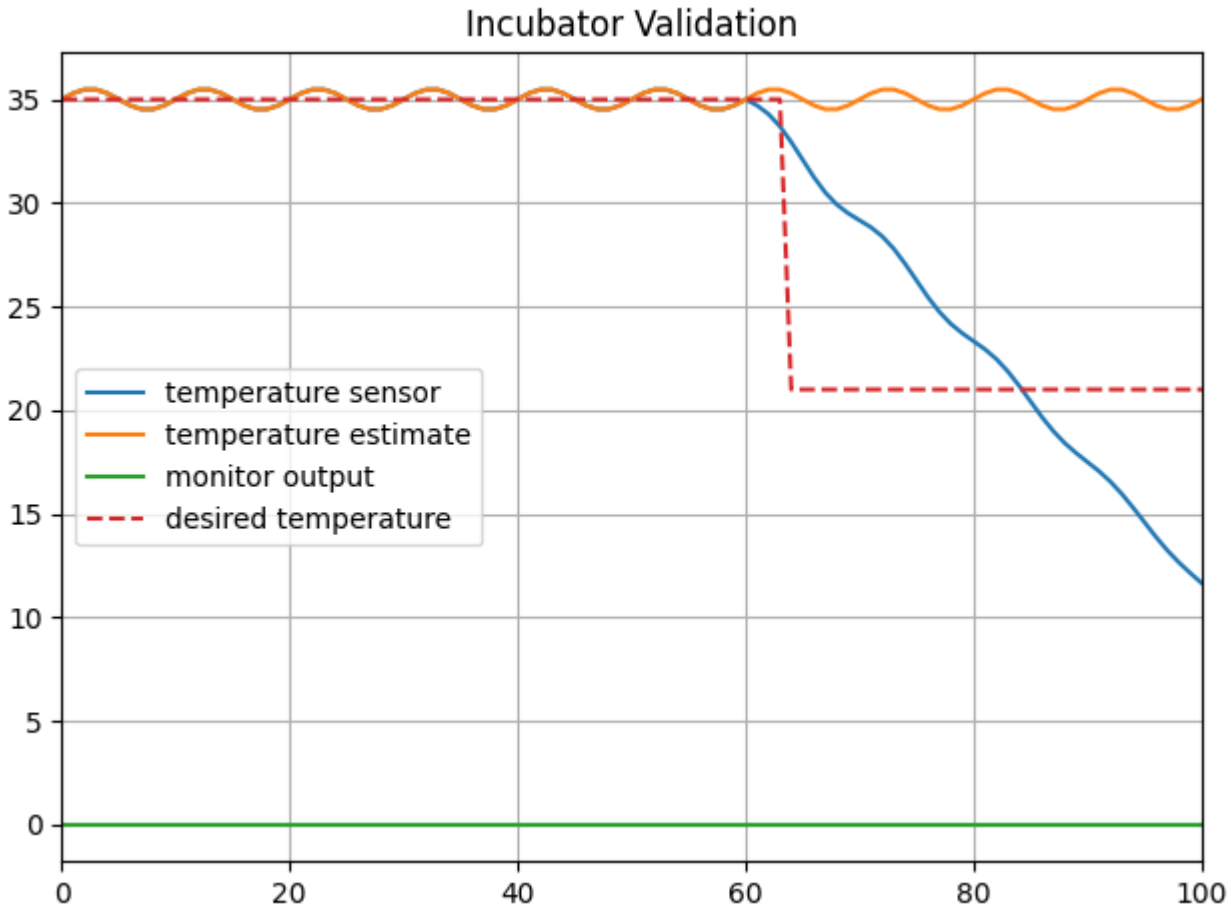
```
lifecycle/execute
```

ANALYZE PHASE

Process the output of co-simulation to produce a plot at: `/workspace/examples/data/incubator-NuRV-monitor-validation/output/plots`.

```
lifecycle/analyze
```

A sample plot is given here.



In the plot, four color-coded indicators provide a visual representation of distinct values. The blue line depicts the simulated temperature, while orange one represents the temperature estimate. A red dashed line indicates the target temperature set by the energy saver component. The green line shows the monitor's output verdict. As observed, when there is a disparity between the estimated and actual temperatures, the energy saver adjusts the target temperature downward, indicating that the component is working properly. Thus, the output of the monitor is fixed at `unknown` (`0`), signifying that the monitoring property is not violated.

Examine the results

The results can be found in the */workspace/examples/data/incubator-NuRV-monitor-validation/output* directory where the logs are also included.

Figures of the output results can be found in the */workspace/examples/data/incubator-NuRV-monitor-validation/output* directory.

TERMINATE PHASE

Terminate to clean up the debug files and co-simulation output files.

```
lifecycle/terminate
```

References

1. More information about NuRV is available here.

2.8.14 Incubator Digital Twin with NuRV monitoring service

Overview

This example demonstrates how a runtime monitoring service (in this example NuRV[1]) can be connected with the Incubator digital twin to verify runtime behavior of the Incubator.

Simulated scenario

This example simulates a scenario where the lid of the Incubator is removed and later put back on. The Incubator is equipped with anomaly detection capabilities, which can detect anomalous behavior (i.e. the removal of the lid). When an anomaly is detected, the Incubator triggers an energy saving mode where the heater is turned off.

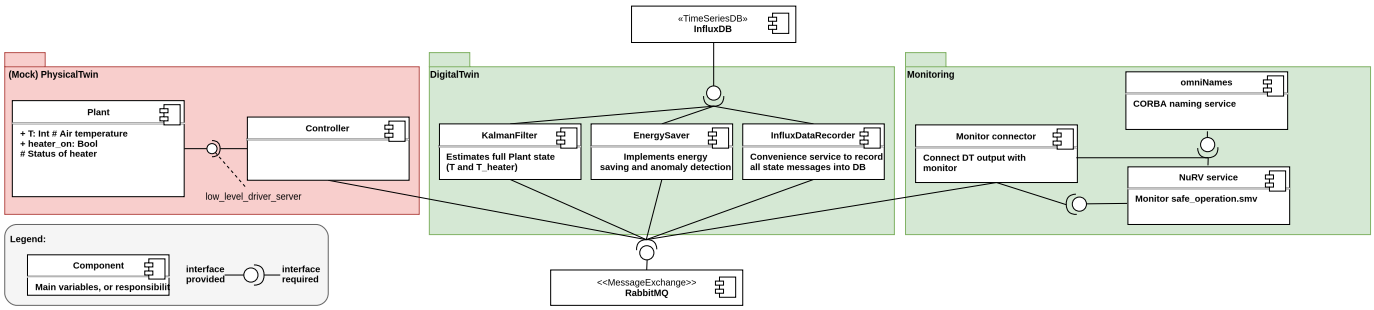
From a monitoring perspective, we wish to verify that within 3 simulation steps of an anomaly detection, the energy saving mode is turned on. To verify this behavior, we construct the property: . Whenever a True or False verdict is produced by the monitor, it is reset, allowing for the detection of repeated satisfaction/violation detections of the property.

The simulated scenario progresses as follows:

- *Initialization*: The services are initialized and the Kalman filter in the Incubator is given 2 minutes to stabilize. Sometimes, the anomaly detection algorithm will detect an anomaly at startup even though the lid is still on. It will disappear after approx 15 seconds.
- *After 2 minutes*: The lid is lifted and an anomaly is detected. The energy saver is turned on shortly after
- *After another 30 seconds*: The energy saver is manually disabled producing a False verdict.
- *After another 30 seconds*: The lid is put back on and the anomaly detection is given time to detect that the lid is back on. The simulation then ends.

Example structure

A diagram depicting the logical software structure of the example can be seen below.



The *execute.py* script is responsible for orchestrating and starting all the relevant services in this example. This includes the Incubator DT, CORBA naming service (omniNames) and the NuRV monitor server as well as implementing the *Monitor connector* component that connects the DT output to the NuRV monitor server.

The NuRV monitor server utilizes a CORBA naming service where it registers under a specific name. A user can then query the naming service for the specific name, to obtain a reference to the monitor server. For more information on how the NuRV monitor server works, please refer to [1].

After establishing connection with the NuRV monitor server, the Incubator DT is started and a RabbitMQ client is created that subscribes to changes in the *anomaly* and *energy_saving* states of the DT. Each time an update is received of either state, the full state (the new updated state and the previous other state) is pushed to the NuRV monitor server whereafter the verdict is printed to the console.

Digital Twin configuration

Before running the example, please configure the *simulation.conf* file with your RabbitMQ credentials.

The example uses the following assets:

Asset Type	Names of Assets	Visibility	Reuse in other Examples
Service	common/services/NuRV_orbit	Common	Yes
DT	common/digital_twins/incubator	Common	Yes
Specification	safe-operation.smv	Private	No
Script	execute.py	Private	No

The *safe-operation.smv* file contains the default monitored specification as described in the Simulated scenario section. These can be configured as desired.

Lifecycle phases

The lifecycle phases for this example include:

Lifecycle phase	Completed tasks
create	Downloads the necessary tools and creates a virtual python environment with the necessary dependencies
execute	Runs a python script that starts up the necessary services as well as the Incubator simulation. Various status messages are printed to the console, including the monitored system states and monitor verdict.
clean	Removes created *data* directory and incubator log files.

If required, change the execute permissions of lifecycle scripts you need to execute. This can be done using the following command

```
chmod +x lifecycle/{script}
```

where {script} is the name of the script, e.g. *create*, *execute* etc.

Running the example

To run the example, first run the following command in a terminal:

```
cd /workspace/examples/digital_twins/incubator-monitor-server/
```

Then, first execute the *create* script (this can take a few mins depending on your network connection) followed by the *execute* script using the following command:

```
lifecycle/{script}
```

The *execute* script will then start outputting system states and the monitor verdict approx every 3 seconds. The output is printed as follows "**State: {anomaly state} & {energy_saving state}, verdict: {Verdict}**" where "*anomaly*" indicates that an anomaly is detected and "!anomaly" indicates that an anomaly is not currently detected. The same format is used for the energy_saving state.

The monitor verdict can be True, False or Unknown, where the latter indicates that the monitor does not yet have sufficient information to determine the satisfaction of the property.

An example output trace is provided below:

```
....
Running scenario with initial state: lid closed and energy saver on
Setting energy saver mode: enable
Setting G_box to: 0.5763498
State: !anomaly & !energy_saving, verdict: True
State: !anomaly & !energy_saving, verdict: True
....
State: anomaly & !energy_saving, verdict: Unknown
State: anomaly & energy_saving, verdict: True
State: anomaly & energy_saving, verdict: True
```

There is currently some startup issues with connecting to the NuRV server, and it will likely take a few tries before the connection is established. This is however handled automatically.

References

1. Information on the NuRV monitor can be found on FBK website.

2.8.15 Incubator Digital Twin with NuRV FMU Monitoring Service

Overview

This example demonstrates how an FMU can be used as a runtime monitoring service (in this example NuRV[1]) and connected with the Incubator digital twin to verify runtime behavior of the Incubator.

Simulated scenario

This example simulates a scenario where the lid of the Incubator is removed and later put back on. The Incubator is equipped with anomaly detection capabilities, which can detect anomalous behavior (i.e. the removal of the lid). When an anomaly is detected, the Incubator triggers an energy saving mode where the heater is turned off.

From a monitoring perspective, we wish to verify that within 3 messages of an anomaly detection, the energy saving mode is turned on. To verify this behavior, we construct the property:

.

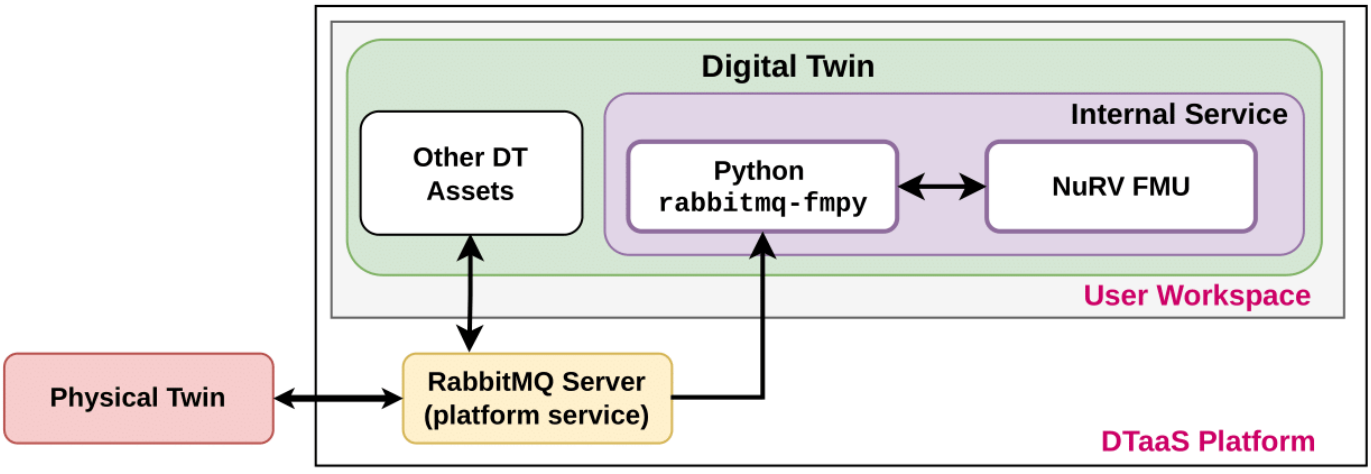
The monitor will output the *unknown* state as long as the property is satisfied and will transition to the *false* state once a violation is detected.

The simulated scenario progresses as follows:

- *Initialization*: The services are initialized and the Kalman filter in the Incubator is given 2 minutes to stabilize. Sometimes, the anomaly detection algorithm will detect an anomaly at startup even though the lid is still on. It will disappear after approx 15 seconds.
- *After 2 minutes*: The lid is lifted and an anomaly is detected. The energy saver is turned on shortly after.
- *After another 30 seconds*: The energy saver is manually disabled producing a *false* verdict.
- *After another 30 seconds*: The lid is put back on and the anomaly detection is given time to detect that the lid is back on. The monitor is then reset producing an Unknown verdict again. The simulation then ends.

Example structure

A diagram depicting the logical software structure of the example can be seen below.



The *execute* script is responsible for starting the NuRV service and running the Python script that controls the scenario (*execute.py*).

The *execute.py* script starts the Incubator services and runs the example scenario. Once the Incubator DT is started, a RabbitMQ client is created that subscribes to changes in the *anomaly* and *energy_saving* states of the DT, as well as the verdicts produced by the NuRV service. Each time an update is received, the full state and verdict is printed to the console.

Digital Twin configuration

Before running the example, please configure the *simulation.conf* file with your RabbitMQ credentials.

The example uses the following assets:

Asset Type	Names of Assets	Visibility	Reuse in other Examples
Tools	common/tool/NuRV/NuRV	Common	Yes
Other	common/fmi2_headers	Common	Yes
DT	common/digital_twins/incubator	Common	Yes
Specification	safe-operation.smv	Private	No
Script	execute.py	Private	No

The *safe-operation.smv* file contains the default monitored specification as described in the Simulated scenario section. These can be configured as desired.

Lifecycle phases

The lifecycle phases for this example include:

Lifecycle phase	Completed tasks
create	Downloads the necessary tools and creates a virtual python environment with the necessary dependencies
execute	Runs a python script that starts up the necessary services as well as the Incubator simulation. Various status messages are printed to the console, including the monitored system states and monitor verdict.

If required, change the execute permissions of lifecycle scripts you need to execute. This can be done using the following command.

```
chmod +x lifecycle/{script}
```

where {script} is the name of the script, e.g. *create*, *execute* etc.

Running the example

To run the example, first run the following command in a terminal:

```
cd /workspace/examples/digital_twins/incubator-NuRV-fmu-monitor-service/
```

Then, first execute the *create* script followed by the *execute* script using the following command:

```
lifecycle/{script}
```

The *execute* script will then start outputting system states and the monitor verdict approx every 3 seconds. The output is printed as follows.

"**State: {anomaly state} & {energy_saving state}**"

where "*anomaly*" indicates that an anomaly is detected and "!anomaly" indicates that an anomaly is not currently detected. The same format is used for the energy_saving state. NuRV verdicts are printed as follows

"**Verdict from NuRV: {verdict}**".

The monitor verdict can be false or unknown, where the latter indicates that the monitor does not yet have sufficient information to determine the satisfaction of the property. The monitor will never produce a true verdict as the entire trace must be verified to ensure satisfaction due to the G operator. Thus the unknown state can be viewed as a tentative true verdict.

An example output trace is provided below:

```
....
Using LIFECYCLE_PATH: /workspace/examples/digital_twins/incubator-NuRV-fmu-monitor-service/lifecycle
Using INCUBATOR_PATH: /workspace/examples/digital_twins/incubator-NuRV-fmu-monitor-service/lifecycle/../../../common/digital_twins/incubator
Starting NuRV FMU Monitor Service, see output at /tmp/nurv-fmu-service.log
NuRVService.py PID: 13496
Starting incubator
Connected to rabbitmq server.
Running scenario with initial state: lid closed and energy saver on
Setting energy saver mode: enable
Setting G_box to: 0.5763498
State: !anomaly & !energy_saving
State: !anomaly & !energy_saving
Verdict from NuRV: unknown
State: !anomaly & !energy_saving
State: !anomaly & !energy_saving
Verdict from NuRV: unknown
State: !anomaly & !energy_saving
State: !anomaly & !energy_saving
Verdict from NuRV: unknown
State: !anomaly & !energy_saving
State: !anomaly & !energy_saving
Verdict from NuRV: unknown
State: !anomaly & !energy_saving
State: !anomaly & !energy_saving
Verdict from NuRV: unknown
```

References

1. Information on the NuRV monitor can be found on FBK website.

3. Admin

3.1 Install

3.1.1 Overview

Install

The goal is to install and administer the DTaaS application for users.

The DTaaS can be installed in different ways. Each version serves a different purpose.

> **Easy Setup on Localhost**
>
> The localhost installation is easy for first time users. Please give it a try.

Otherwise, use the installation setup that fits your needs.

Installation Setup	Purpose
localhost	Install DTaaS on your computer for a single user; does not need a web server. *This setup does not require domain name.*
secure localhost	Install DTaaS on your computer for a single user over HTTPS with integrated gitlab installation; does not need a web server. *This setup does not require domain name.*
Server	Install DTaaS on server for multiple users. Please check the requirements. It is also possible to host the application over HTTPS with integrated gitlab installation
One vagrant machine	Install DTaaS on a virtual machine; can be used for single or multiple users.
Two vagrant machines	Install DTaaS on two virtual machines; can be used for single or multiple users.
	The core DTaaS application is installed on the first virtual machine and all the services (RabbitMQ, MQTT, InfluxDB, Grafana and MongoDB) are installed on second virtual machine.
Independent Packages	Can be used independently; do not need full installation of DTaaS.

The installation steps is a good place to start the installation process.

Administer

There is a CLI to add and delete users of a running application.

3.1.2 Installation Steps

Complete DTaaS Application

DTaaS application is available in two flavors. One is **localhost** which is suitable for single-user, local usage. Another is **production server** which is suitable for multi-user setup.

In both cases, the installation is a three step process.

SETUP AUTHORIZATION

DTaaS provides security using OAuth authorization for both react client frontend and backend services.

There is a default frontend authorization application setup for all localhost and backend authorization is not required for localhost installation.

The production server installation requires both react client frontend and backend services applications.

CONFIGURE COMPONENTS

DTaaS is available as docker compose application. There are four docker compose files

1. `compose.local.yml` for localhost installation served over HTTP connection.
2. `compose.local.secure.yml` for secure localhost installation served over HTTPS connection.
3. `compose.server.yml` for production server installation served over HTTP connection.
4. `compose.server.secure.yml` for production server installation served over HTTPS connection.

These four compose files require environment configuration files. The explanation of this configuration file is available directly on the installation pages.

In addition, react client frontend requires configuration which is explained on this page.

INSTALL

Follow installation instructions given on either localhost or production server pages.

Independent Packages

Each release of DTaaS also comes with release of three reusable packages. These packages have dedicated documentation.

3.1.3 Requirements

Tip

These optional requirements are not needed for **localhost** installation. They are only required for installation of the DTaaS on a production web server.

There are two optional requirements for installing the DTaaS.

OAuth Provider

The DTaaS software is uses OAuth for user authorization. It is possible to use either *gitlab.com* or your own OAuth service provider.

Domain name

The DTaaS software is a web application and is preferably hosted on a server with a domain name like *foo.com*. However, it is possible to install the software on your computer and use access it at *localhost*.

3.1.4 Authorization

OAuth for React Client

To enable user authorization on DTaaS React client website, you will use the OAuth authorization protocol, specifically the PKCE authorization flow. Here are the steps to get started:

1. Choose Your GitLab Server:

- You need to set up OAuth authorization on a GitLab server. The commercial gitlab.com is not suitable for multi-user authorization (DTaaS requires this), so you'll need an on-premise GitLab instance.
- You can use GitLab Omnibus Docker for this purpose.
- Configure the OAuth application as an instance-wide authorization type.

2. Determine Your Website's Hostname:

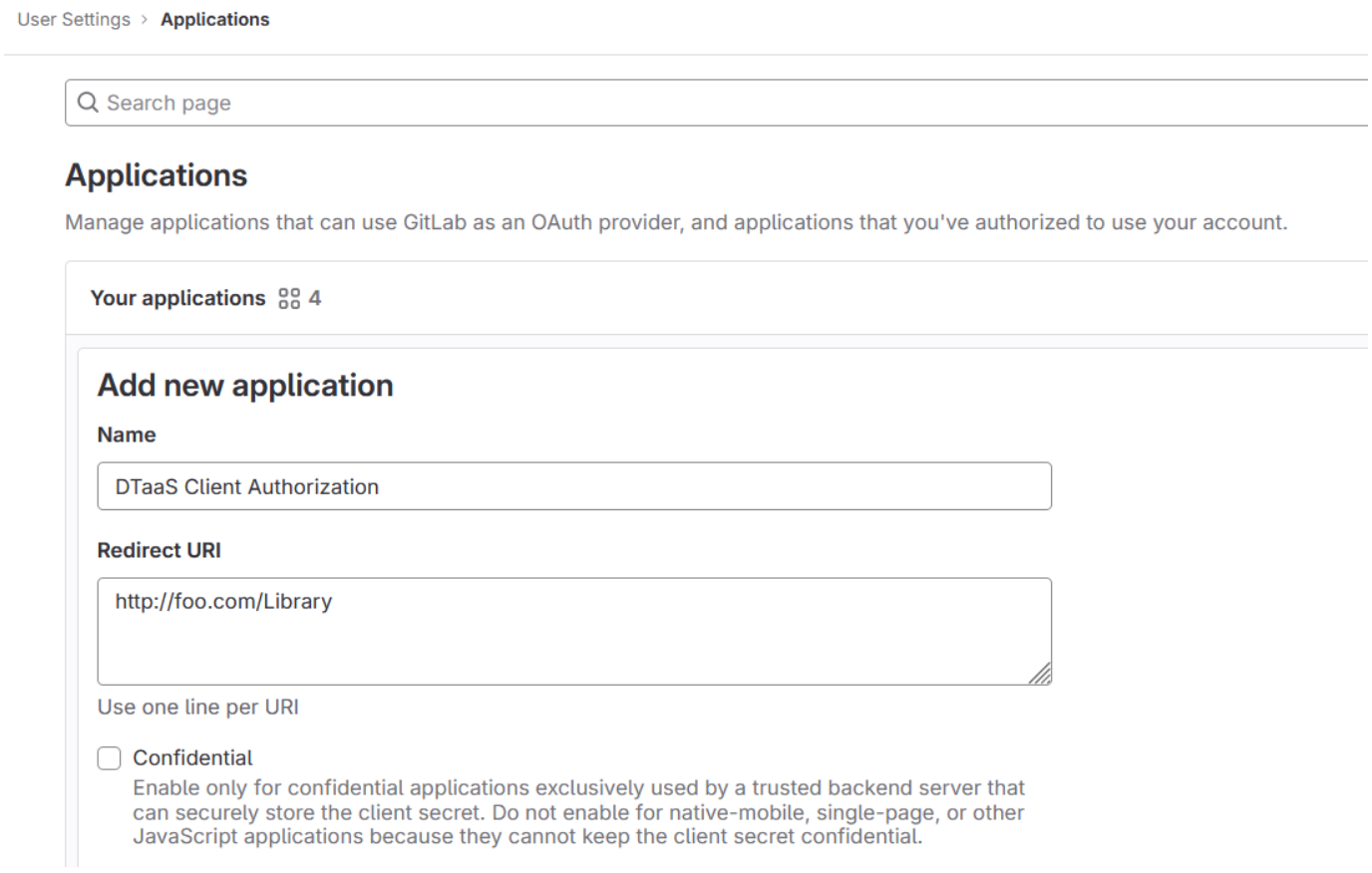
- Before setting up OAuth on GitLab, decide on the hostname for your website. It's recommended to use a self-hosted GitLab instance, which you will use in other parts of the DTaaS application.

3. Define Callback and Logout URLs:

- For the PKCE authorization flow to function correctly, you need two URLs: a callback URL and a logout URL.
- The callback URL informs the OAuth provider of the page where signed-in users should be redirected. It's different from the landing homepage of the DTaaS application.
- The logout URL is where users will be directed after logging out.

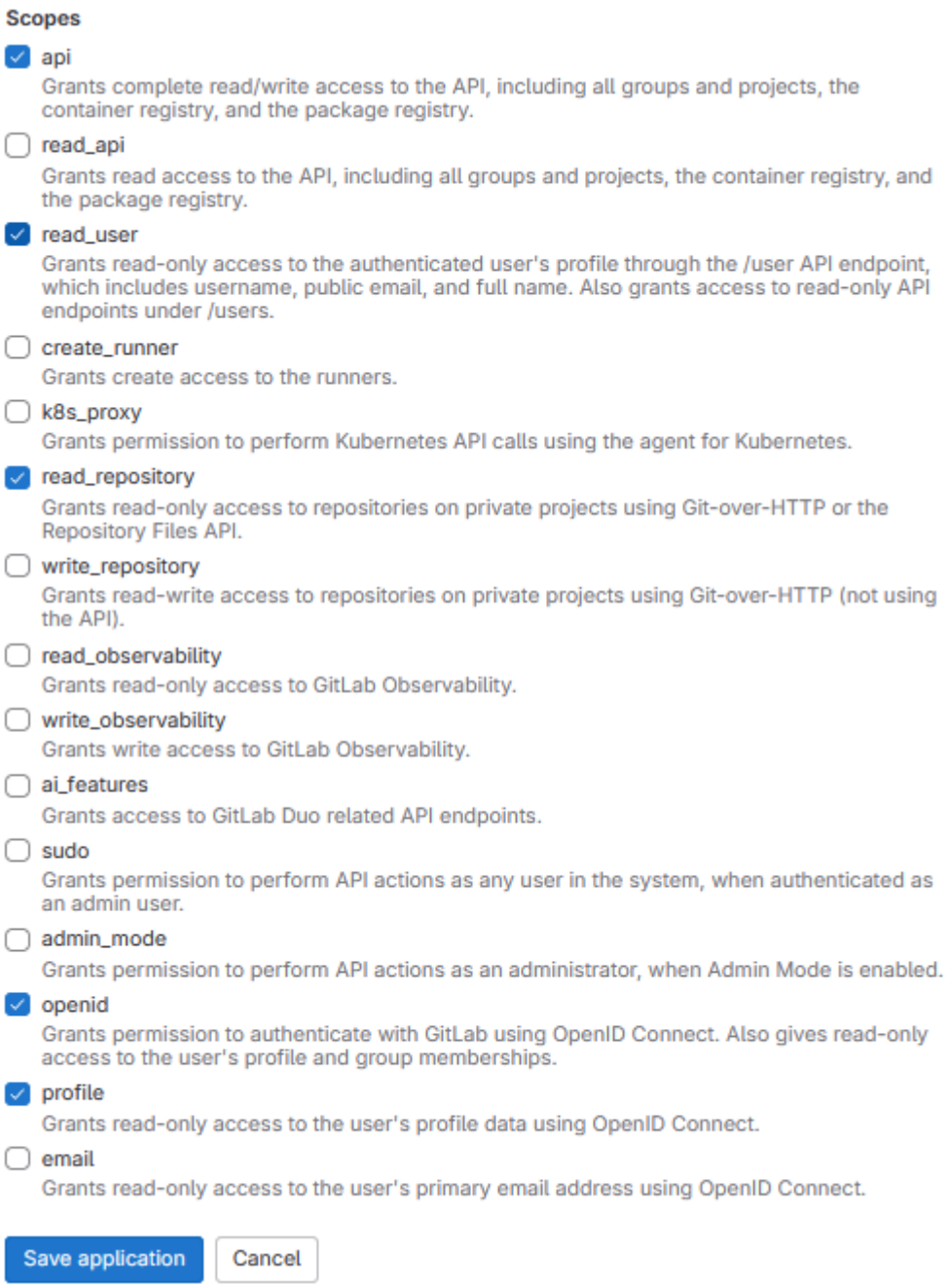
4. OAuth Application Creation:

- During the creation of the OAuth application on GitLab, you need to specify the scope. Choose openid, profile, read_user, read_repository, and api scopes.



5. Application ID:

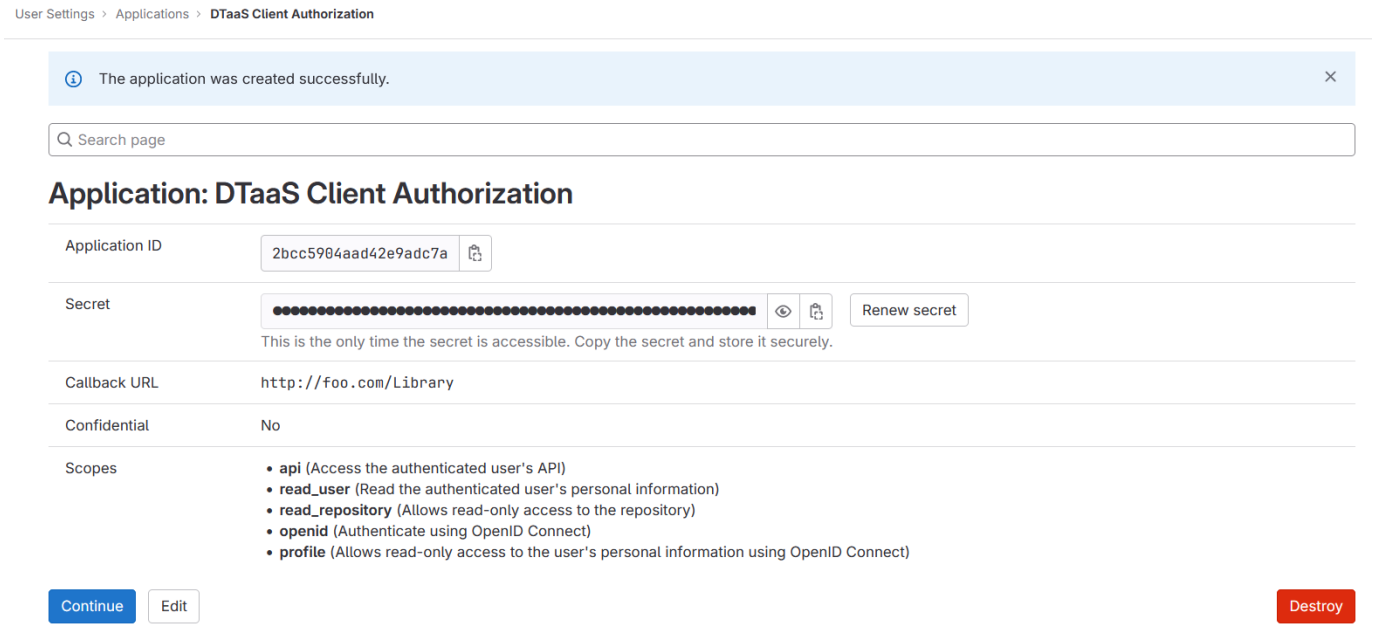
- After successfully creating the OAuth application, GitLab generates an application ID. This is a long string of HEX values that you will need for your configuration files.



6. Required Information from OAuth Application:

- You will need the following information from the OAuth application registered on GitLab:

GitLab Variable Name	Variable Name in Client env.js	Default Value
OAuth Provider	REACT_APP_AUTH_AUTHORITY	https://gitlab.foo.com/
Application ID	REACT_APP_CLIENT_ID	
Callback URL	REACT_APP_REDIRECT_URI	https://foo.com/Library
Scopes	REACT_APP_GITLAB_SCOPES	openid, profile, read_user, read_repository, api



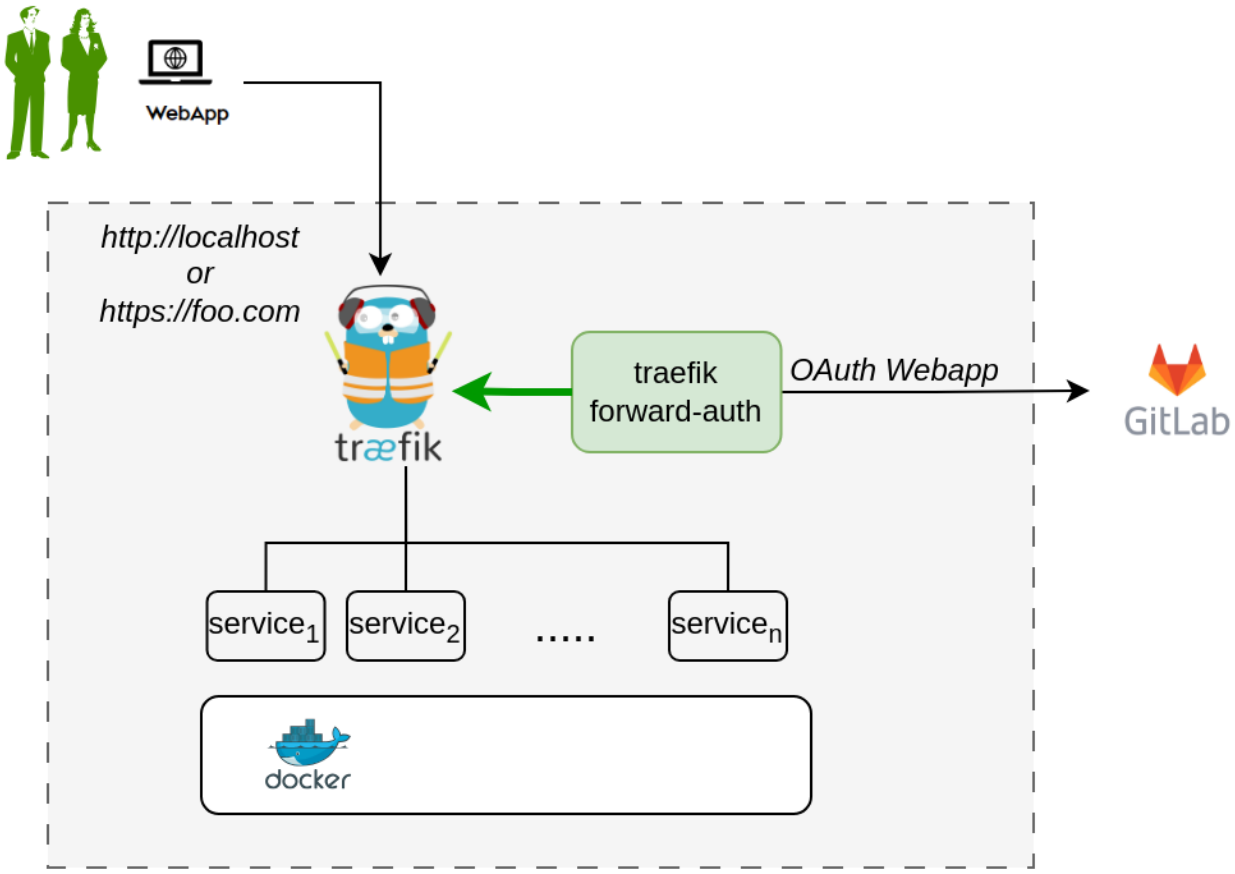
7. Create User Accounts:

Create user accounts in gitlab for all the usernames chosen during installation. The *trial* installation script comes with two default usernames - *user1* and *user2*. For all other installation scenarios, accounts with specific usernames need to be created on gitlab.

OAuth for Traefik Gateway

The traefik gateway is used to serve the DTaaS. All the services provided as part of the application are secured at the traefik gateway. The security is based on Traefik forward-auth.

An illustration of the docker containers used and the authorization setup is shown here.



The **traefik forward-auth** can use any OAuth2 provider, but within the DTaaS gitlab is used as authorization provider. You will use the OAuth the web / server application authorization flow.

Here are the steps to get started:

1. Choose GitLab Server:

- You need to set up OAuth authorization on a GitLab server. The commercial gitlab.com is not suitable for multi-user authorization (DTaaS requires this), so you'll need an on-premise GitLab instance.
- You can use GitLab Omnibus Docker for this purpose.
- Configure the OAuth application as an instance-wide authorization type. Select option to generate client secret and also selection option for trusted application.

2. Determine Website Hostname:

Before setting up OAuth on GitLab, decide on the hostname for your website. It's recommended to use a self-hosted GitLab instance, which you will use in other parts of the DTaaS application.

3. Determine Callback and Logout URLs:

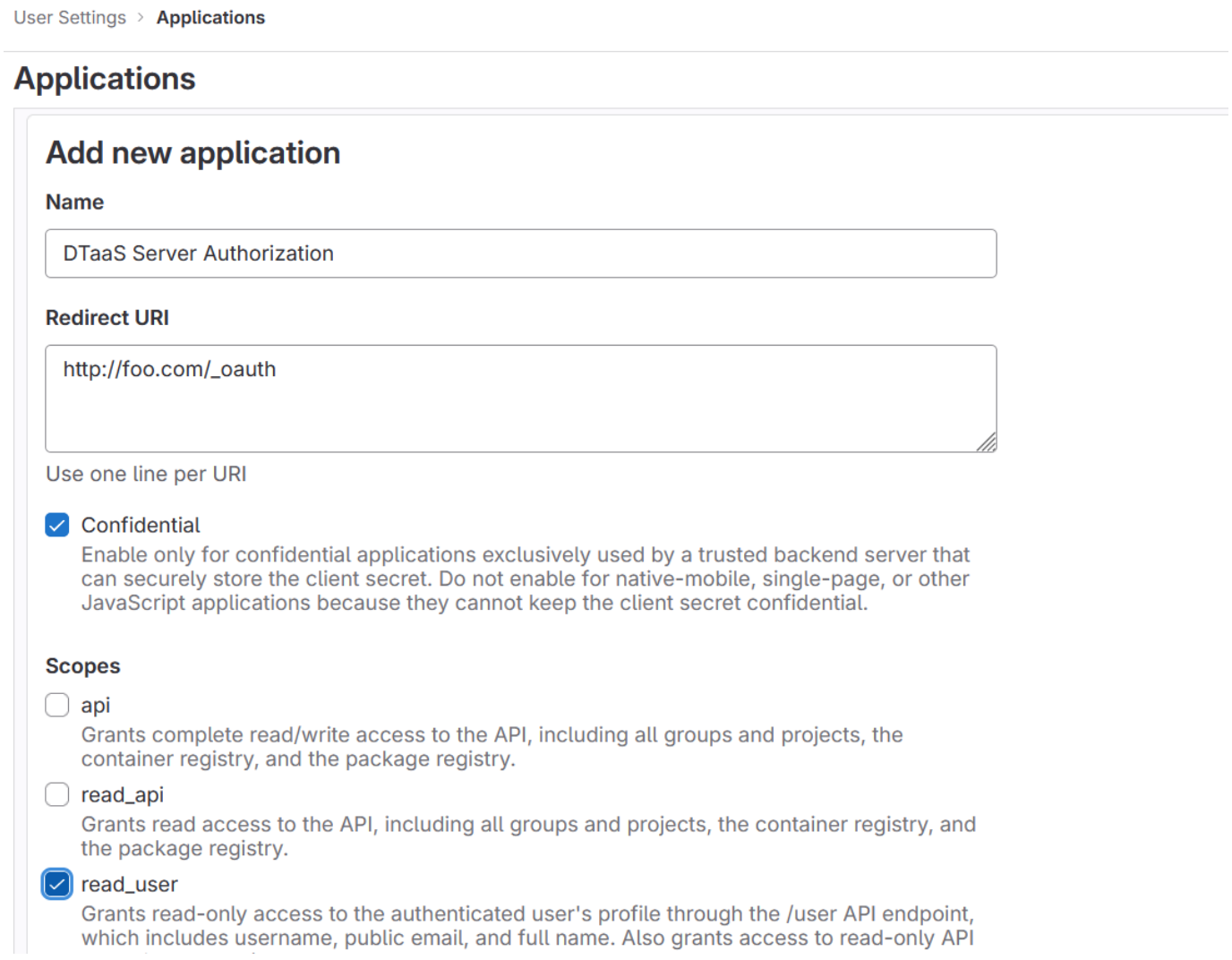
For the web / server authorization flow to function correctly, you need two URLs: a *callback URL* and a *logout URL*.

- The callback URL informs the OAuth provider of the page where signed-in users should be redirected. It is the landing homepage of the DTaaS application. (either http://foo.com/_oauth/ or http://localhost/_oauth/)
- The logout URL is the URL for signout of gitlab and clear authorization within traefik-forward auth. (either http://foo.com/_oauth/logout or http://localhost/_oauth/logout). The logout URL is to help users logout of traefik forward-auth. The logout URL should not be entered into Gitlab OAuth application setup.

4. Create OAuth Application:

Oauth application setup on GitLab can be located on Edit Profile -> Application https://gitlab.foo.com/-/profile/applications.

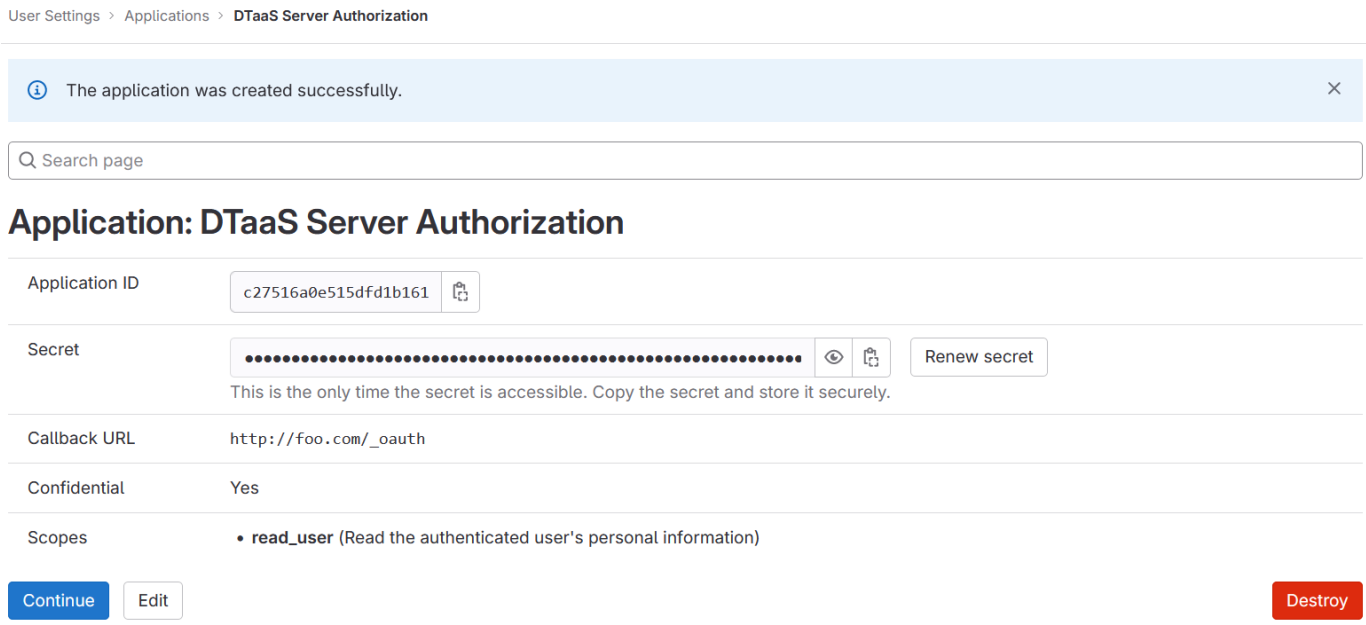
During the creation of the OAuth application on GitLab, you need to specify the scope. Choose ***read_user*** scope.



5. Copy Application Credentials:

After successfully creating the OAuth application, GitLab generates an *application ID* and *client secret*.

Both these values are long string of HEX values that you will need for your configuration files.



6. Checklist: Required Information from OAuth Application:

You will need the following information from the OAuth application registered on GitLab:

GitLab Variable Name	Variable Name in .env of docker compose file	Default Value
OAuth Provider	OAUTH_URL	https://gitlab.foo.com/
Application ID	OAUTH_CLIENT_ID	*xx*
Application Secret	OAUTH_CLIENT_SECRET	*xx*
Callback URL	(to be directly entered in Gitlab OAuth registration)	
Forward-auth secret	OAUTH_SECRET	*random-secret-string* (password for forward-auth, can be changed to your preferred string)
Scopes	read_user	

DEVELOPMENT ENVIRONMENT

The development environment and server installation scenarios requires traefik forward-auth.

CONFIGURE AUTHORIZATION RULES FOR TRAEFIK FORWARD-AUTH

The Traefik forward-auth microservices requires configuration rules to manage authorization for different URL paths. The *conf.server* file can be used to configure the specific rules. There are broadly three kinds of URLs:

Public Path Without Authorization

To setup a public page, an example is shown below.

```
rule.noauth.action=allow
rule.noauth.rule=Path(`/public`)
```

Here, 'noauth' is the rule name, and should be changed to suit rule use. Rule names should be unique for each rule. The 'action' property is set to "allow" to make the resource public. The 'rule' property defines the path/route to reach the resource.

Common to All Users

To setup a common page that requires Gitlab OAuth, but is available to all users of the Gitlab instance:

```
rule.all.action=auth
rule.all.rule=Path(`/common`)
```

The 'action' property is set to "auth", to enable Gitlab OAuth before the resource can be accessed.

Selective Access

Selective Access refers to the scenario of allowing access to a URL path for a few users. To setup selective access to a page:

```
rule.onlyu1.action=auth
rule.onlyu1.rule=Path(`/user1`)
rule.onlyu1.whitelist = user1@localhost
```

The 'whitelist' property of a rule defines a comma separated list of email IDs that are allowed to access the resource. While signing in users can sign in with either their username or email ID as usual, but the email ID corresponding to the account should be included in the whitelist.

This restricts access of the resource, allowing only users mentioned in the whitelist.

USER MANAGEMENT

DTaaS provides an easy way to add and remove additional users from your DTaaS instance.

All such user management can be done via the DTaaS CLI

LIMITATION

The rules in _conf._ *file are not dynamically loaded into the traefik-forward-auth microservice. Any change in the conf file requires restart of traefik-forward-auth for the changes to take effect. All the existing user sessions get invalidated when the traefik-forward-auth** restarts.

Use a simple command on the terminal.

- For a local instance:

```
docker compose -f compose.server.yml --env-file .env up -d --force-recreate traefik-forward-auth
```

- For a server instance:

```
docker compose -f compose.server.yml --env-file .env.server up -d --force-recreate traefik-forward-auth
```

3.1.5 Configuration

Configure Client Website

This page describes various configuration options for react website.

```
if (typeof window !== 'undefined') {
  window.env = {
    REACT_APP_ENVIRONMENT: "prod | dev | local | test",
    REACT_APP_URL: "URL for the gateway",
    REACT_APP_URL_BASENAME: "Base URL for the client website"(optional, can be null),
    REACT_APP_URL_DTLINK: "Endpoint for the Digital Twin",
    REACT_APP_URL_LIBLINK: "Endpoint for the Library Assets",
    REACT_APP_WORKBENCHLINK_VNCDESKTOP: "Endpoint for the VNC Desktop link",
    REACT_APP_WORKBENCHLINK_VSCODE: "Endpoint for the VS Code link",
    REACT_APP_WORKBENCHLINK_JUPYTERLAB: "Endpoint for the Jupyter Lab link",
    REACT_APP_WORKBENCHLINK_JUPYTERNOTEBOOK:
      "Endpoint for the Jupyter Notebook link",
    REACT_APP_WORKBENCHLINK_DT_PREVIEW: "Endpoint for the Digital Twins page preview",
    REACT_APP_CLIENT_ID: 'AppID genereated by the gitlab OAuth provider',
    REACT_APP_AUTH_AUTHORITY: 'URL of the private gitlab instance',
    REACT_APP_REDIRECT_URI: 'URL of the homepage for the logged in users of the website',
    REACT_APP_LOGOUT_REDIRECT_URI: 'URL of the homepage for the anonymous users of the website',
    REACT_APP_GITLAB_SCOPES: 'OAuth scopes. These should match with the scopes set in gitlab OAuth provider',
  };
};

// Example values with no base URL. Trailing and ending slashes are optional.
if (typeof window !== 'undefined') {
  window.env = {
    REACT_APP_ENVIRONMENT: 'prod',
    REACT_APP_URL: 'https://foo.com/',
    REACT_APP_URL_BASENAME: '',
    REACT_APP_URL_DTLINK: '/lab',
    REACT_APP_URL_LIBLINK: '',
    REACT_APP_WORKBENCHLINK_VNCDESKTOP: '/tools/vnc/?password=vncpassword',
    REACT_APP_WORKBENCHLINK_VSCODE: '/tools/vscode/',
    REACT_APP_WORKBENCHLINK_JUPYTERLAB: '/lab',
    REACT_APP_WORKBENCHLINK_JUPYTERNOTEBOOK: '',
    REACT_APP_WORKBENCHLINK_DT_PREVIEW: '/preview/digitaltwins',
    REACT_APP_CLIENT_ID: '1be55736756190b3ace4c2c4fb19bde386d1dcc748d20b47ea8cfb5935b8446c',
    REACT_APP_AUTH_AUTHORITY: 'https://gitlab.foo.com/',
    REACT_APP_REDIRECT_URI: 'https://foo.com/Library',
    REACT_APP_LOGOUT_REDIRECT_URI: 'https://foo.com/',
    REACT_APP_GITLAB_SCOPES: 'openid profile read_user read_repository api',
  };
};


// Example values with "bar" as basename URL.
//Trailing and ending slashes are optional.
if (typeof window !== 'undefined') {
  window.env = {
    REACT_APP_ENVIRONMENT: "dev",
    REACT_APP_URL: 'http://localhost:4000/',
    REACT_APP_URL_BASENAME: 'bar',
    REACT_APP_URL_DTLINK: '/lab',
    REACT_APP_URL_LIBLINK: '',
    REACT_APP_WORKBENCHLINK_VNCDESKTOP: '/tools/vnc/?password=vncpassword',
    REACT_APP_WORKBENCHLINK_VSCODE: '/tools/vscode/',
    REACT_APP_WORKBENCHLINK_JUPYTERLAB: '/lab',
    REACT_APP_WORKBENCHLINK_JUPYTERNOTEBOOK: '',
    REACT_APP_WORKBENCHLINK_DT_PREVIEW: '/preview/digitaltwins',
    REACT_APP_CLIENT_ID: '1be55736756190b3ace4c2c4fb19bde386d1dcc748d20b47ea8cfb5935b8446c',
    REACT_APP_AUTH_AUTHORITY: 'https://gitlab.foo.com/',
    REACT_APP_REDIRECT_URI: 'http://localhost:4000/bar/Library',
    REACT_APP_LOGOUT_REDIRECT_URI: 'http://localhost:4000/bar',
    REACT_APP_GITLAB_SCOPES: 'openid profile read_user read_repository api',
  };
};
```

MULTIPLE DTAAS APPLICATIONS

> **Warning**
>
> This is not a regular installation scenario and is only recommended for experts. This installation setup requires significant modifications to the docker compose files.

The DTaaS is a regular web application. It is possible to host multiple DTaaS applications on the same server. The only requirement is to have a distinct URLs. You can have three DTaaS applications running at the following URLs.

```
https://foo.com/au
https://foo.com/acme
https://foo.com/bar
```

All of these instances can use the same gitlab instance for authorization.

DTaaS application URL	Gitlab Instance URL	Callback URL	Logout URL	Application ID
https://foo.com/au	https://foo.gitlab.com	https://foo.com/au/Library	https://foo.com/au	autogenerated by gitlab
https://foo.com/acme	https://foo.gitlab.com	https://foo.com/au/Library	https://foo.com/au	autogenerated by gitlab
https://foo.com/bar	https://foo.gitlab.com	https://foo.com/au/Library	https://foo.com/au	autogenerated by gitlab

If you are hosting multiple DTaaS instances on the same server, do not install DTaaS with a null basename on the same server. Even though it works well, the setup is confusing to setup and may lead to maintenance issues.

If you choose to host your DTaaS application with a basename (say bar), then the URLs in `env.js` change to:

```
DTaaS application URL: https://foo.com/bar
Gitlab instance URL: https://gitlab.foo.com
Callback URL: https://foo.com/bar/Library
Logout URL: https://foo.com/bar
```

Configure Library Microservice

The microservices requires config specified in yaml format. The template configuration file is:

```
port: '4001'
mode: 'local' or 'git'
local-path: '/Users/<Username>/DTaaS/files'
log-level: 'debug'
apollo-path: '/lib' or ''
graphql-playground: 'false' or 'true'

#Only needed if git mode
git-repos:
  - <username>:
      repo-url: '<git repo url>'
  ...
  - <username>:
      repo-url: '<git repo url>'
```

The `LOCAL_PATH` variable is the absolute filepath to the location of the local directory which will be served to users by the Library microservice.

The `MODE` variable sets the mode for which and how the files should be served. If `git` mode is chosen, the following is required.

Variable	Description
`username`	Username in which folder the repos will be cloned.
`git repo url`	HTTP URL of the git repository to clone. Optional to add `.git` to the end of the URL.

Replace the default values the appropriate values for your setup.

The **libms** looks for `libms.yaml` file in the working directory from which it is run. If you want to run **libms** without explicitly specifying the configuration file, run with `-c <path-to-file>`.

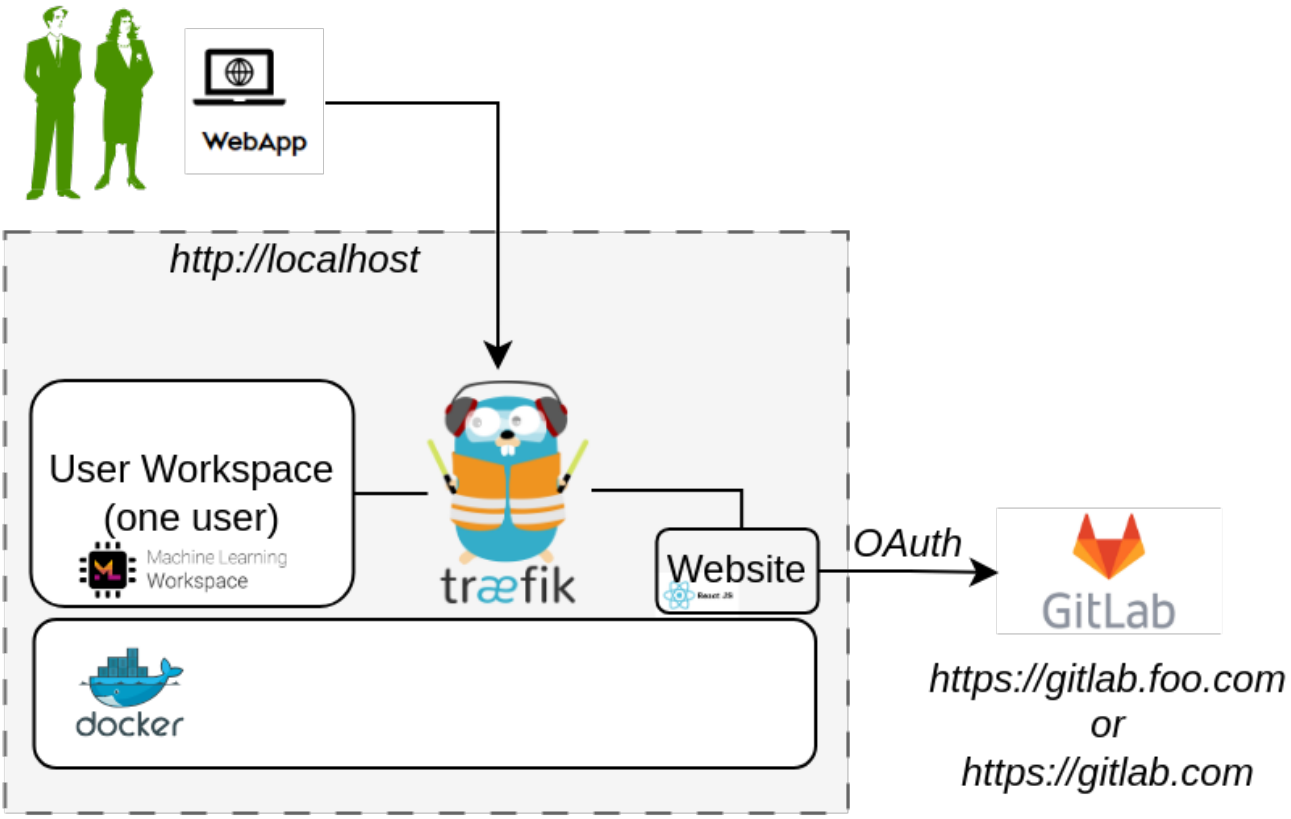
3.1.6 Docker

Install DTaaS on localhost

The installation instructions provided in this README are ideal for running the DTaaS on both localhost. This installation is ideal for single users intending to use DTaaS on their own computers.

DESIGN

An illustration of the docker containers used and the authorization setup is shown here.



REQUIREMENTS

The installation requirements to run this docker version of the DTaaS are:

- docker desktop / docker CLI with compose plugin
- User account on *gitlab.com*

> **Tip**
>
> The frontend website requires authorization. The default authorization configuration works for *gitlab.com*. If you desire to use locally hosted gitlab instance, please see the client docs.

CLONE CODEBASE

```
git clone https://github.com/INTO-CPS-Association/DTaaS.git
cd DTaaS
```

Tip

1. The filepaths shown here follow POSIX convention. The installation procedures also work with Windows paths.
2. The description below refers to filenames. All the file paths mentioned below are relatively to the top-level **DTaaS** directory.

CONFIGURATION

Docker Compose

The docker compose configuration is in `deploy/docker/.env.local`; it is a sample file. It contains environment variables that are used by the docker compose files. It can be updated to suit your local installation scenario. It contains the following environment variables.

Edit all the fields according to your specific case.

URL Path	Example Value	Explanation
DTAAS_DIR	'/home/Desktop/DTaaS'	Full path to the DTaaS directory. This is an absolute path with no trailing slash.
username1	'user1'	Your gitlab username

Tip

Important points to note:

1. The path examples given here are for Linux OS. These paths can be Windows OS compatible paths as well.
2. The client configuration file is located at `deploy/config/client/env.local.js`. If you are following the guide to use HTTPS on localhost, edit the URLs in this file by replacing `http` with `https`. Beyond this, it is not necessary to modify this file.

Create User Workspace

The existing filesystem for installation is setup for `user1`. A new filesystem directory needs to be created for the selected user.

Please execute the following commands from the top-level directory of the DTaaS project.

```
cp -R files/user1 files/username
```

where *username* is the selected username registered on https://gitlab.com.

RUN

The commands to start and stop the appliation are:

```
docker compose -f compose.local.yml --env-file .env.local up -d
docker compose -f compose.local.yml --env-file .env.local down
```

To restart only a specific container, for example `client``

```
docker compose -f compose.local.yml --env-file .env.local up -d --force-recreate client
```

USE

The application will be accessible at: http://localhost from web browser. Sign in using your https://gitlab.com account.

All the functionality of DTaaS should be available to you through the single page client now.

LIMITATIONS

The library microservice is not included in the localhost installation scenario.

REFERENCES

Image sources: Traefik logo, ml-workspace, reactjs, gitlab

Install DTaaS on localhost with Gitlab Integration

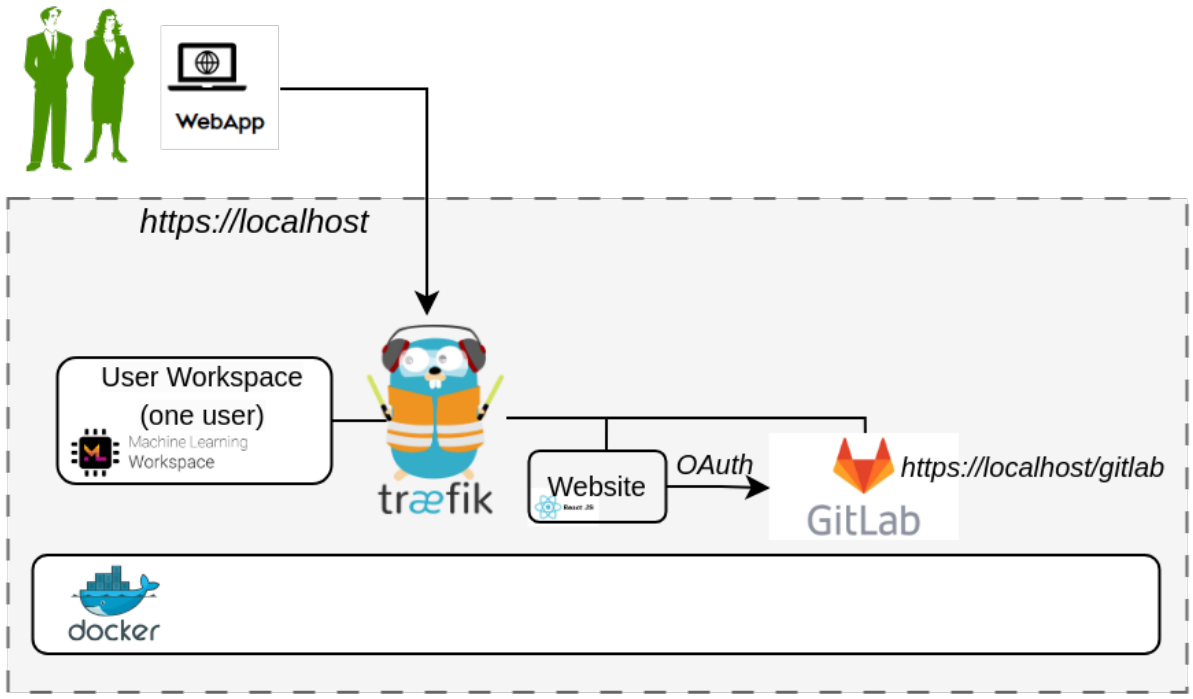
This installation is ideal for single users intending to use DTaaS on their own computers.

The installation instructions provided in this README are ideal for running the **DTaaS on localhost served over HTTPS connection**. **The intention is to integrate Gitlab into DTaaS so that both are running on localhost.**

If you do not need gitlab running on localhost, please use the simpler localhost setup.

DESIGN

An illustration of the docker containers used and the authorization setup is shown here.



REQUIREMENTS

The installation requirements to run this docker version of the DTaaS are:

- docker desktop / docker CLI with compose plugin
- mkcert

CLONE CODEBASE

```
git clone https://github.com/INTO-CPS-Association/DTaaS.git
cd DTaaS
```

📋 file pathnames

1. The filepaths shown here follow POSIX convention. The installation procedures also work with Windows paths.
2. The description below refers to filenames. All the file paths mentioned below are relatively to the top-level **DTaaS** directory.

CONFIGURE AND RUN

Create User Workspace

The existing filesystem for installation is setup for `user1`. A new filesystem directory needs to be created for the selected user.

Please execute the following commands from the top-level directory of the DTaaS project.

```
cp -R files/user1 files/username
```

where *username* is the selected username to be created (in next steps) on Gitlab running at https://localhost/gitlab.

Obtain TLS / HTTPS Certificate

Use mkcert to generate TLS certificates using this guide. The certificates need to be generated for `localhost`.

The names of the certificates must be `fullchain.pem` and `privkey.pem`. The `fullchain.pem` corresponds to public certificate and the `privkey.pem` corresponds to private key.

Add TLS Certificates to Traefik

Copy the two certificate files into:

- `deploy/docker/certs/localhost/fullchain.pem`
- `deploy/docker/certs/localhost/privkey.pem`

Traefik will run with self-issued certificates if the above two certificates are either not found or found invalid.

Configure Docker Compose

The docker compose configuration is in `deploy/docker/.env.local`; it is a sample file. It contains environment variables that are used by the docker compose files. It can be updated to suit your local installation scenario. It contains the following environment variables.

Edit all the fields according to your specific case.

URL Path	Example Value	Explanation
DTAAS_DIR	'/home/Desktop/DTaaS'	Full path to the DTaaS directory. This is an absolute path with no trailing slash.
username1	'user1'	Your gitlab username

📋 Important points to note:

1. The path examples given here are for Linux OS. These paths can be Windows OS compatible paths as well.
2. The client configuration file is located at `deploy/config/client/env.local.js`. Edit the URLs in this file by replacing `http` with `https`. Beyond this, it is not necessary to modify this file.

Start DTaaS to Integrate Gitlab

Start the appliation with

```
docker compose -f compose.local.secure.yml --env-file .env.local up -d
```

Start Gitlab

Use the instructions provided in gitlab integration to bring up Gitlab on localhost and the Gitlab service will be available at https://localhost/gitlab

Register OAuth2 Application

The frontend website requires OAuth2 application registration on the integrated Gitlab. The details of Oauth2 app for the frontend website are in client docs.

The default OAuth client application provided in `env.local.js` works. However, if you intend to run an integrated gitlab instance, then this application needs to be created on gitlab running at https://localhost/gitlab.

Remember to use https://localhost/Library as the Callback URL (`REACT_APP_REDIRECT_URI`).

Please see gitlab oauth provider documentation for further help with creating this OAuth application.

Update Client Website Configuration

Replace the contents of `deploy/config/client/env.local.js` with the following.

```
if (typeof window !== 'undefined') {
  window.env = {
    REACT_APP_ENVIRONMENT: 'local',
    REACT_APP_URL: 'https://localhost/',
    REACT_APP_URL_BASENAME: '',
    REACT_APP_URL_DTLINK: '/lab',
    REACT_APP_URL_LIBLINK: '',
    REACT_APP_WORKBENCHLINK_VNCDESKTOP: '/tools/vnc/?password=vncpassword',
    REACT_APP_WORKBENCHLINK_VSCODE: '/tools/vscode/',
    REACT_APP_WORKBENCHLINK_JUPYTERLAB: '/lab',
    REACT_APP_WORKBENCHLINK_JUPYTERNOTEBOOK: '',

    REACT_APP_CLIENT_ID: 'xxxxxx',
    REACT_APP_AUTH_AUTHORITY: 'https://localhost/gitlab/',
    REACT_APP_REDIRECT_URI: 'https://localhost/Library',
    REACT_APP_LOGOUT_REDIRECT_URI: 'https://localhost/',
    REACT_APP_GITLAB_SCOPES: 'openid profile read_user read_repository api',
  };
};
```

And then update OAuth2 client application ID (`REACT_APP_CLIENT_ID`) with that of the newly registered OAuth2 application.

Restart DTaaS Client Website

To update the client website configuration, run

```
docker compose -f compose.local.secure.yml --env-file .env.local up -d --force-recreate client
```

USE

The application will be accessible at: https://localhost from web browser. Sign in using your https://localhost/gitlab account.

All the functionality of DTaaS should be available to you through the single page client now.

LIMITATIONS

The library microservice is not included in the localhost installation scenario.

DOCKER HELP

The commands to start and stop the appliation are:

```
docker compose -f compose.local.secure.yml --env-file .env.local up -d
docker compose -f compose.local.secure.yml --env-file .env.local down
```

To restart only a specific container, for example `client`

```
docker compose -f compose.local.secure.yml --env-file .env.local up -d --force-recreate client
```

REFERENCES

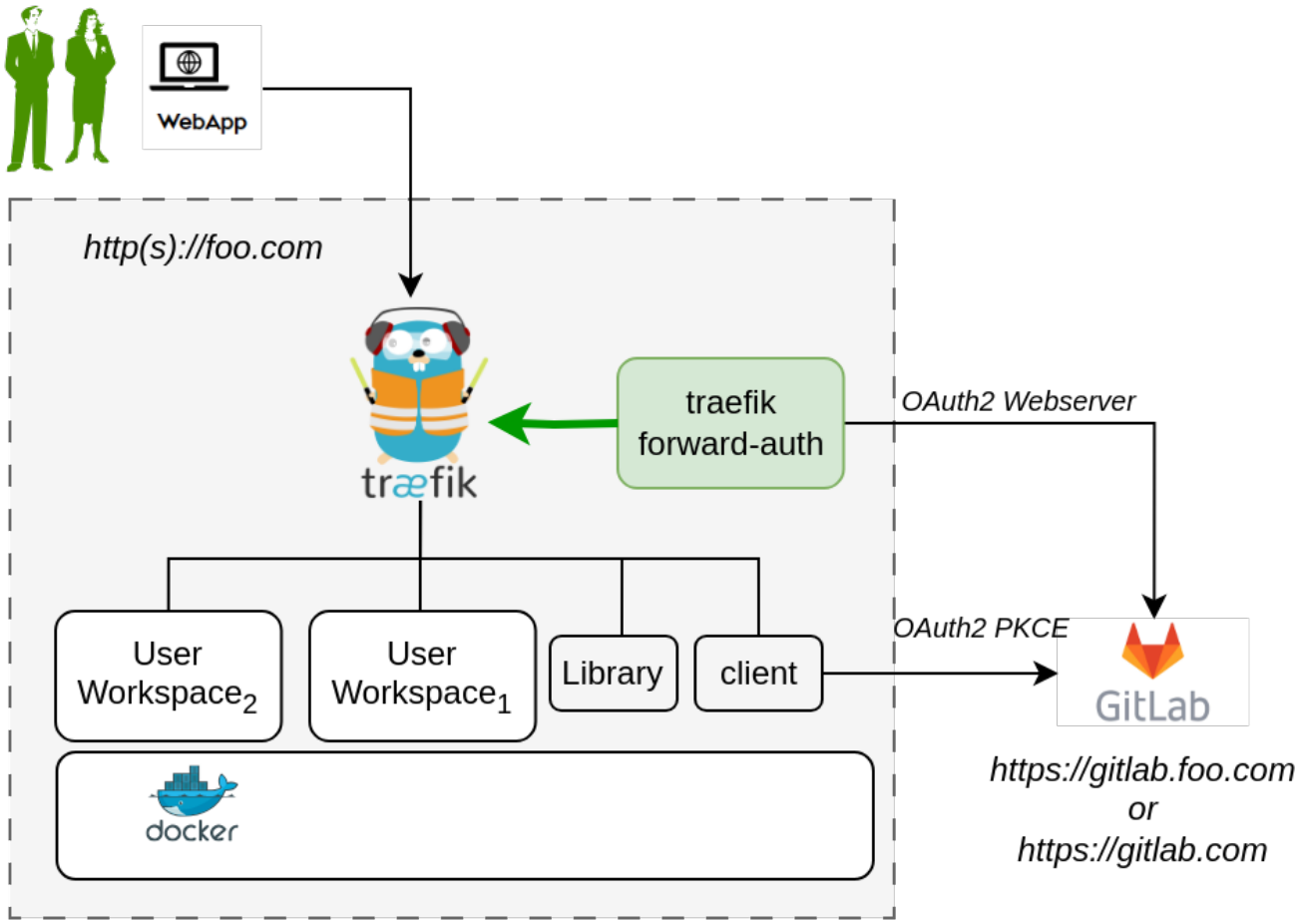
Image sources: Traefik logo, ml-workspace, reactjs, gitlab

Install DTaaS on a Production Server

The installation instructions provided in this README are ideal for hosting the DTaaS as web application for multiple users.

DESIGN

An illustration of the docker containers used and the authorization setup is shown here.



In the new application configuration, there are two OAuth2 applications.

REQUIREMENTS

The installation requirements to run this docker version of the DTaaS are:

Docker with Compose Plugin

It is mandatory to have Docker installed on your computer. We highly recommend using Docker Desktop.

Domain name

The DTaaS software is a web application and is preferably hosted on a server with a domain name like *foo.com*. It is also possible to use an IP address in place of domain name.

TLS / HTTPS Certificate (Optional)

It is possible to add HTTPS option to the DTaaS software installation. Creation of the required TLS certificates is possible through certbot.

OAuth Provider

Gitlab Instance - The DTaaS uses Gitlab OAuth2.0 authorization for user authorization. You can either have an on-premise instance of gitlab, or use gitlab.com itself.

User Accounts

Create user accounts in a linked gitlab instance for all the users.

The default docker compose file contains two - *user1* and *user2*. These names need to be changed to suitable usernames.

OAuth2 Application Registration

The multi-user installation setup requires dedicated authorization setup for both frontend website and backend services. Both these authorization requirements are satisfied using OAuth2 protocol.

- The frontend website is a React single page application (SPA).
- The details of Oauth2 app for the frontend website are in client docs.
- The Oauth2 authorization for backend services is managed by Traefik forward-auth. The details of this authorization setup are in server docs.

It is possible to use https://gitlab.com or a local installation of Gitlab can be used for this purpose. Based on your selection of gitlab instance, it is necessary to register these two OAuth2 applications and link them to your intended DTaaS installation.

Please see gitlab oauth provider documentation for further help with creating these two OAuth applications.

CLONE CODEBASE

```
git clone https://github.com/INTO-CPS-Association/DTaaS.git
cd DTaaS
```

Tip

1. The filepaths shown here follow Linux OS. The installation procedures also work with Windows OS.
2. The description below refers to filenames. All the file paths mentioned below are relatively to the top-level **DTaaS** directory.

CONFIGURATION

Three following configuration files need to be updated.

Docker Compose

The docker compose configuration is in `deploy/docker/.env.server` . it is a sample file. It contains environment variables that are used by the docker compose files. It can be updated to suit your local installation scenario. It contains the following environment variables.

Edit all the fields according to your specific case.

URL Path	Example Value	Explanation
DTAAS_DIR	'/Users/username/DTaaS'	Full path to the DTaaS directory. This is an absolute path with no trailing slash.
SERVER_DNS	*foo.com*	The server DNS, if you are deploying with a dedicated server. Remember not use http(s) at the beginning of the DNS string
OAUTH_URL	*gitlab.foo.com*	The URL of your Gitlab instance. It can be *gitlab.com* if you are planning to use it for authorization.
OAUTH_CLIENT_ID	'xx'	The ID of your server OAuth application
OAUTH_CLIENT_SECRET	'xx'	The Secret of your server OAuth application
OAUTH_SECRET	'random-secret-string'	Any private random string. This is a password you choose for local installation.
username1	'user1'	The gitlab instance username of a user of DTaaS
username2	'user2'	The gitlab instance username of a user of DTaaS

Tip

Important points to note:

1. The path examples given here are for Linux OS. These paths can be Windows OS compatible paths as well.
2. The client configuration file is located at `deploy/config/client/env.js`.
3. The Server DNS can also be an IP address. However, for proper working it is neccessary to use the same convention (IP/DNS) in the client configuration file as well.

Website Client

The frontend React website requires configuration which is specified in the client configuration file (`deploy/config/client/env.js`).

Further explanation on the client configuration is available in client config.

Tip

There is a default OAuth application registered on https://gitlab.com for client. The corresponding OAuth application details are:

```
REACT_APP_CLIENT_ID: '1be55736756190b3ace4c2c4fb19bde386d1dcc748d20b47ea8cfb5935b8446c',
REACT_APP_AUTH_AUTHORITY: 'https://gitlab.com/',
```

This can be used for test purposes. Please use your own OAuth application for secure production deployments.

Create User Workspace

The existing filesystem for installation is setup for `files/user1`. A new filesystem directory needs to be created for the selected user.

Please execute the following commands from the top-level directory of the DTaaS project.

```
cp -R files/user1 files/username
```

where *username* is one of the selected usernames. This command needs to be repeated for all the selected users.

Configure Authorization Rules for Backend Authorization

The Traefik forward-auth microservices requires configuration rules to manage authorization for different URL paths. The `deploy/docker/conf.server` file can be used to configure the authorization for user workspaces.

```
rule.onlyu1.action=auth
rule.onlyu1.rule=Path(`/user1`)
rule.onlyu1.whitelist = user1@localhost

rule.onlyu1.action=auth
rule.onlyu1.rule=Path(`/user2`)
rule.onlyu1.whitelist = user2@localhost
```

Please change the usernames and email addresses to the matching user accounts on the OAuth provider (either https://gitlab.foo.com or https://gitlab.com).

Caveat

The usernames in the `deploy/docker/.env.server` file need to match those in the `deploy/docker/conf.server` file.

Traefik routes are controlled by the `deploy/docker/.env.server` file. Authorization on these routes is controlled by the `deploy/docker/conf.server` file. If a route is not specified in `deploy/docker/conf.server` file but an authorisation is requested by traefik for this unknown route, the default behavior of traefik forward-auth kicks in. This default behavior is to enable endpoint being available to any signed in user.

If there are extra routes in `deploy/docker/conf.server` file but these are not in `deploy/docker/.env.server` file, such routes are not served by traefik; it will give **404 server response**.

ACCESS RIGHTS OVER FILES

> **Warning**
>
> The default setting in docker compose file exposes all user files at http://foo.com/lib/files. All files of all the users are readable-writable by all logged in users. The `compose.server.yml` / `compose.server.secure.yml` file needs to be updated to expose another directory like common assets directory.

If you wish to reduce this scope to only **common assets**, please change,

```
  libms:
    image: intocps/libms:latest
    restart: unless-stopped
    volumes:
      - ${DTAAS_DIR}/deploy/config/lib.env:/dtaas/libms/.env
      - ${DTAAS_DIR}/files/common:/dtaas/libms/files
```

The change in the last line. The `${DTAAS_DIR}/files` got replaced by `${DTAAS_DIR}/files/common`. With this change, only common files are readable-writable by all logged in users.

Add TLS Certificates (Optional)

The application can be served on HTTPS connection for which TLS certificates are needed. The certificates need to be issued for `foo.com` or `*.foo.com`. The names of the certificates must be `fullchain.pem` and `privkey.pem`. Copy these two certificate files into:

- `certs/foo.com/fullchain.pem`
- `certs/foo.com/privkey.pem`

Traefik will run with self-issued certificates if the above two certificates are either not found or found invalid.

Remember to update `dynamic/tls.yml` with correct path matching your DNS name. For example, if your DNS name is `www.foo.com`, then copy the TLS certificates of `www.foo.com` to `certs/` directory and update `dynamic/tls.yml` as follows.

```
tls:
  certificates:
    - certFile: /etc/traefik-certs/www.foo.com/fullchain.pem
      keyFile: /etc/traefik-certs/www.foo.com/privkey.pem
      stores:
        - default
```

RUN

Over HTTP

This docker compose file serves application over HTTP.

The commands to start and stop the appliation are:

```
docker compose -f compose.server.yml --env-file .env.server up -d
docker compose -f compose.server.yml --env-file .env.server down
```

To restart only a specific container, for example `client``

```
docker compose -f compose.server.yml --env-file .env.server up -d --force-recreate client
```

Over HTTPS

This docker compose file serves application over HTTP.

The commands to start and stop the appliation are:

```
docker compose -f compose.server.secure.yml --env-file .env.server up -d
docker compose -f compose.server.secure.yml --env-file .env.server down
```

To restart only a specific container, for example `client``

```
docker compose -f compose.server.secure.yml --env-file .env.server up -d --force-recreate client
```

USE

The application will be accessible at: from web browser. Sign in using your account linked to either *gitlab.com* or your local gitlab instance.

All the functionality of DTaaS should be available to your users through the single page client now.

You may have to click Sign in to Gitlab on the Client page and authorize access to the shown application.

Adding a new user

Please see the add new user to add new users.

REFERENCES

Image sources: Traefik logo, ml-workspace, reactjs, gitlab

3.1.7 Vagrant

DTaaS Vagrant Box

This README provides instructions on creating a custom Operating System virtual disk for running the DTaaS software. The virtual disk is managed by **vagrant**. The purpose is two fold:

- Provide cross-platform installation of the DTaaS application. Any operating system supporting use of vagrant software utility can support installation of the DTaaS software.
- Create a ready to use development environment for code contributors.

There are two scripts in this directory:

Script name	Purpose	Default
`user.sh`	user installation	✅
`developer.sh`	developer installation	❌

If you are installing the DTaaS for developers, the default installation caters to your needs. You can skip the next step and continue with the creation of vagrant box.

If you are a developer and would like additional software installed, you need to modify `Vagrantfile`. The existing `Vagrantfile` has two lines:

```
1       config.vm.provision "shell", path: "user.sh"
2       #config.vm.provision "shell", path: "developer.sh"
```

Uncomment the second line to have more software components installed. If you are not a developer, no changes are required to the `Vagrantfile`.

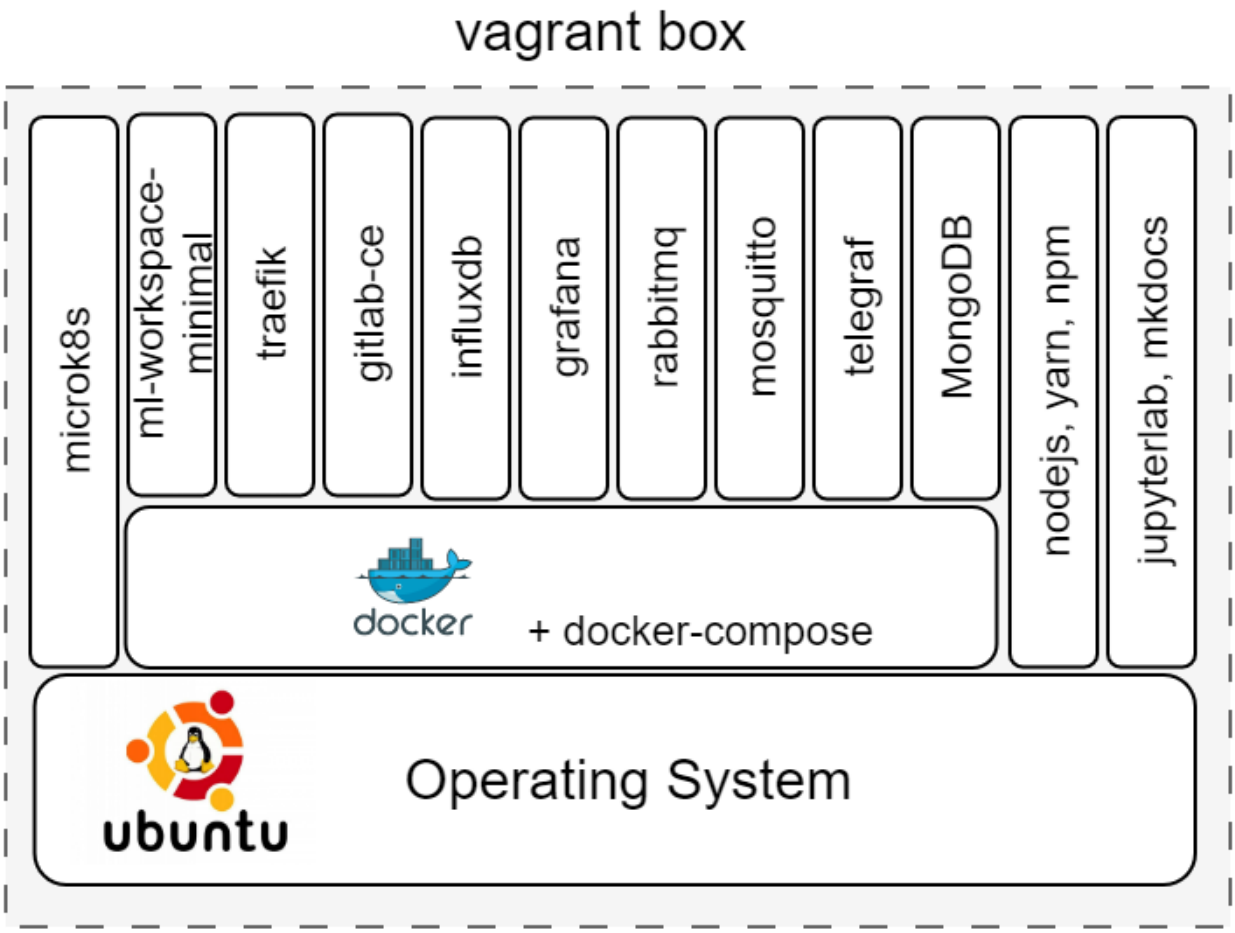
This vagrant box installed for users will have the following items:

1. docker v24.0
2. nodejs v20.10
3. yarn v1.22
4. npm v10.2
5. containers - ml-workspace-minimal v0.13, traefik v2.10, gitlab-ce v16.4, influxdb v2.7, grafana v10.1, rabbitmq v3-management, eclipse-mosquitto (mqtt) v2, mongodb v7.0

This vagrant box installed for developers will have the following items additional items:

- docker-compose v2.20
- microk8s v1.27
- jupyterlab
- mkdocs
- container - telegraf v1.28

At the end of installation, the software stack created in vagrant box can be visualised as shown in the following figure.



The upcoming instructions will help with the creation of base vagrant box.

```
#create a key pair
ssh-keygen -b 4096 -t rsa -f vagrant -q -N ""

vagrant up

# let the provisioning be complete
# replace the vagrant ssh key-pair with personal one
vagrant ssh

# install the oh-my-zsh
sh -c "$(curl -fsSL https://raw.github.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"
# install plugins: history, autosuggestions,
git clone https://github.com/zsh-users/zsh-autosuggestions ${ZSH_CUSTOM:-~/.oh-my-zsh/custom}/plugins/zsh-autosuggestions

# inside ~/.zshrc, modify the following line
plugins=(git zsh-autosuggestions history cp tmux)

# to replace the default vagrant ssh key-pair with
# the generated private key into authorized keys
cp /vagrant/vagrant.pub /home/vagrant/.ssh/authorized_keys

# exit vagrant guest machine and then
# copy own private key to vagrant private key location
cp vagrant .vagrant/machines/default/virtualbox/private_key

# check
vagrant ssh #should work

# exit vagrant guest machine and then
vagrant halt

vagrant package --base dtaas \
--info "info.json" --output dtaas.vagrant

# Add box to the vagrant cache in ~/.vagrant.d/boxes directory
vagrant box add --name dtaas ./dtaas.vagrant

# You can use this box in other vagrant boxes using
#config.vm.box = "dtaas"
```

REFERENCES

Image sources: Ubuntu logo

DTaaS on Single Vagrant Machine

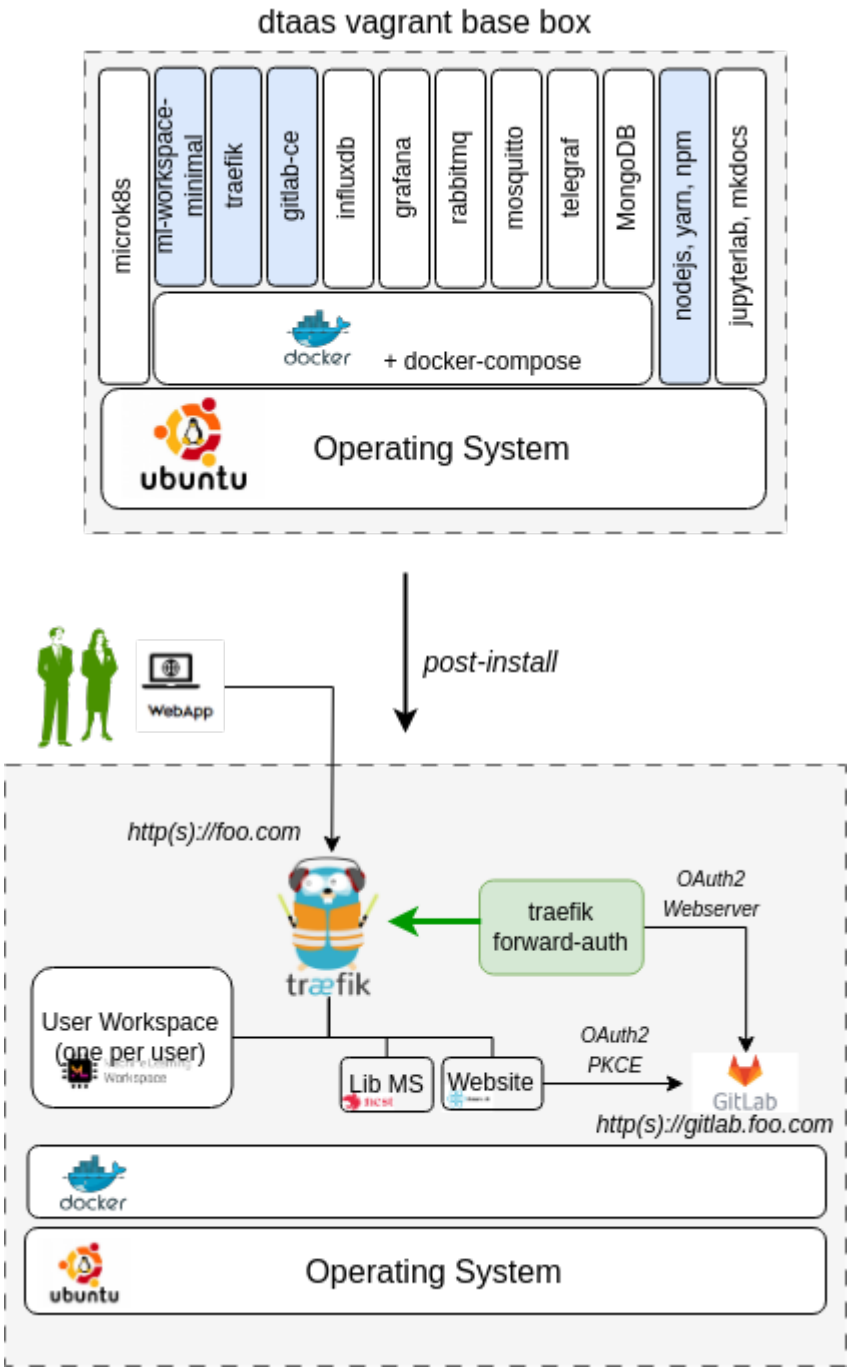
These are installation instructions for running DTaaS software inside one vagrant Virtual Machine. The setup requires a machine which can spare 16GB RAM, 8 vCPUs and 50GB Hard Disk space to the vagrant box.

CREATE BASE VAGRANT BOX

Create **dtaas** Vagrant box. You would have created an SSH key pair - *vagrant* and *vagrant.pub*. The *vagrant* is the private SSH key and is needed for the next steps. Copy *vagrant* SSH private key into the current directory (`deploy/vagrant/single-machine`). This shall be useful for logging into the vagrant machines created for two-machine deployment.

TARGET INSTALLATION SETUP

The goal is to use the **dtaas** Vagrant box to install the DTaaS software on one single vagrant machine. A graphical illustration of a successful installation can be seen here.



There are many unused software packages/docker containers within the dtaas base box. The used packages/docker containers are highlighed in blue color.

Tip

The illustration shows hosting of gitlab on the same vagrant machine with *http(s)://gitlab.foo.com* The gitlab setup is outside the scope this installation guide. Please refer to gitlab docker install for gitlab installation.

CONFIGURE SERVER SETTINGS

A dummy **foo.com** URL has been used for illustration. Please change this to your unique website URL.

Please follow the next steps to make this installation work in your local environment.

Update the **Vagrantfile**. Fields to update are:

1. Hostname (`node.vm.hostname = "foo.com"`)
2. MAC address (`:mac => "xxxxxxxx"`). This change is required if you have a DHCP server assigning domain names based on MAC address. Otherwise, you can leave this field unchanged.
3. Other adjustments are optional.

INSTALLATION STEPS

Execute the following commands from terminal

```
vagrant up
vagrant ssh
```

Set a cronjob inside the vagrant virtual machine to remote the conflicting default route. Download the route script and run the following command.

```
sudo bash route.sh
```

Please follow the instructions of regular server installation setup to complete the installation.

REFERENCES

Image sources: Ubuntu logo, Traefik logo, ml-workspace, nodejs, reactjs, nestjs

DTaaS on Two Vagrant Machines

These are installation instructions for running DTaaS application in two vagrant virtual machines (VMs). In this setup, all the user workspaces shall be run on server1 while all the platform services will be run on server2.

The setup requires two server VMs with the following hardware configuration:

server1: 16GB RAM, 8 x64 vCPUs and 50GB Hard Disk space

server2: 6GB RAM, 3 x64 vCPUs and 50GB Hard Disk space

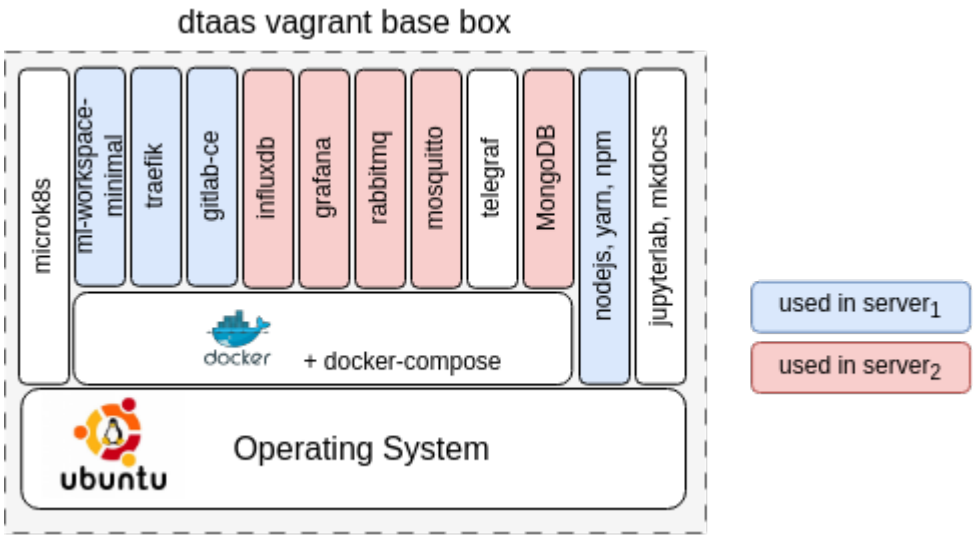
Under the default configuration, two user workspaces are provisioned on server1. The default installation setup also installs InfluxDB, Grafana, RabbitMQ and MQTT services on server2. If you would like to install more services, you can create shell scripts to install the same on server2.

CREATE BASE VAGRANT BOX

Create **dtaas** Vagrant box. You would have created an SSH key pair - *vagrant* and *vagrant.pub*. The *vagrant* is the private SSH key and is needed for the next steps. Copy *vagrant* SSH private key into the current directory (`deploy/vagrant/two-machine`). This shall be useful for logging into the vagrant machines created for two-machine deployment.

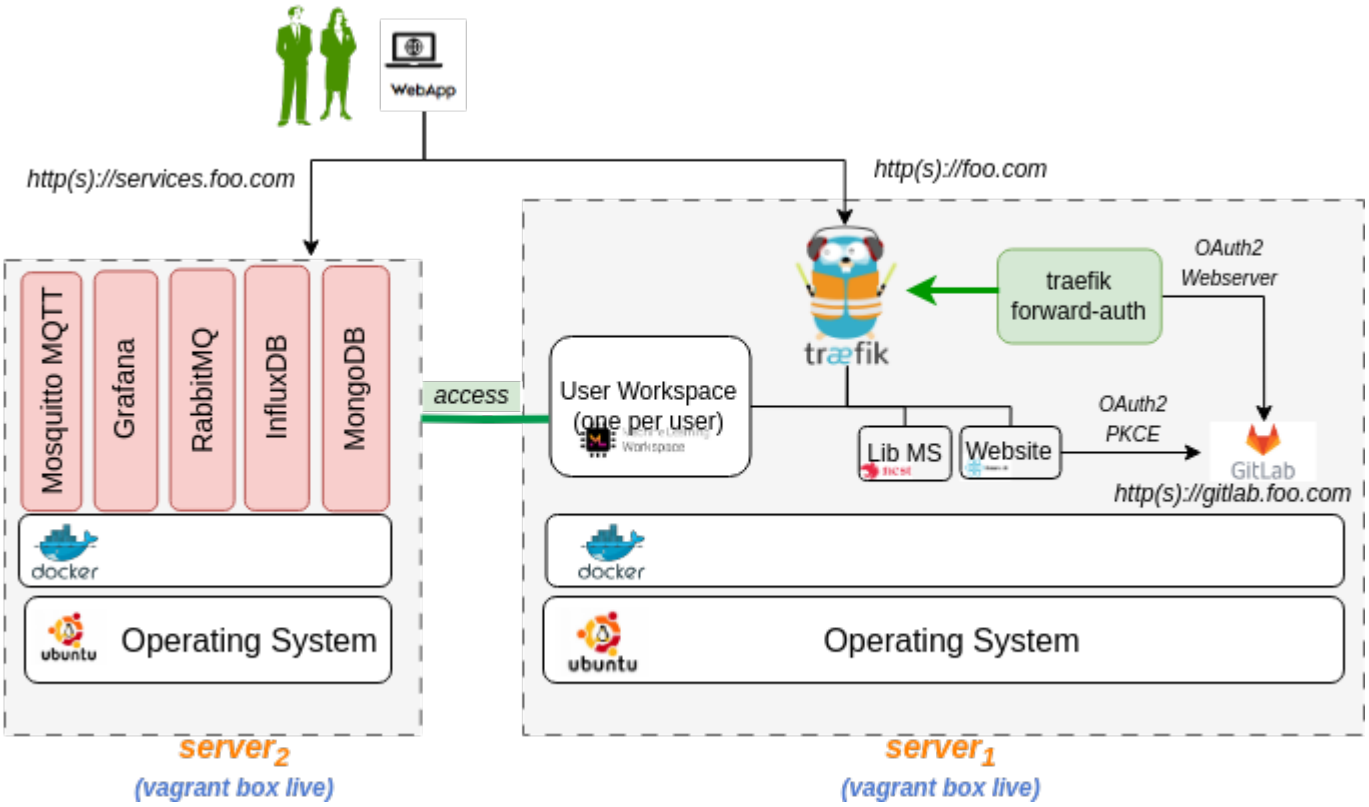
TARGET INSTALLATION SETUP

The goal is to use this **dtaas** vagrant box to install the DTaaS software on server1 and the default platform services on server2. Both the servers are vagrant machines.



There are many unused software packages/docker containers within the dtaas base box. The used packages/docker containers are highlighed in blue and red color.

A graphical illustration of a successful installation can be seen here.



In this case, both the vagrant boxes are spawed on one server using two vagrant configuration files, namely *boxes.json* and *Vagrantfile*.

> **Tip**
>
> The illustration shows hosting of gitlab on the same vagrant machine with *http(s)://gitlab.foo.com* The gitlab setup is outside the scope this installation guide. Please refer to gitlab docker install for gitlab installation.

CONFIGURE SERVER SETTINGS

📋 A dummy **foo.com** and **services.foo.com** URLs has been used for illustration. Please change these to your unique website URLs.

The first step is to define the network identity of the two VMs. For that, you need *server name*, *hostname* and *MAC address*. The hostname is the network URL at which the server can be accessed on the web. Please follow these steps to make this work in your local environment.

Update the **boxes.json**. There are entries one for each server. The fields to update are:

1. `name` - name of server1 (`"name" = "dtaas-two"`)
2. `hostname` - hostname of server1 (`"name" = "foo.com"`)
3. MAC address (`:mac => "xxxxxxxx"`). This change is required if you have a DHCP server assigning domain names based on MAC address. Otherwise, you can leave this field unchanged.
4. `name` - name of server2 (`"name" = "services"`)
5. `hostname` - hostname of server2 (`"name" = "services.foo.com"`)
6. MAC address (`:mac => "xxxxxxxx"`). This change is required if you have a DHCP server assigning domain names based on MAC address. Otherwise, you can leave this field unchanged.
7. Other adjustments are optional.

INSTALLATION STEPS

The installation instructions are given separately for each vagrant machine.

Launch DTaaS Platform Default Services

Follow the installation guide for services to install the DTaaS platform services.

After the services are up and running, you can see the following services active within server2 (*services.foo.com*).

service	external url
InfluxDB database	services.foo.com
Grafana visualization service	services.foo.com:3000
MQTT Broker	services.foo.com:1883
RabbitMQ Broker	services.foo.com:5672
RabbitMQ Broker management website	services.foo.com:15672
MongoDB database	services.foo.com:27017

Install DTaaS Application

Execute the following commands from terminal

```
vagrant up
vagrant ssh
```

Set a cronjob inside the vagrant virtual machine to remote the conflicting default route. Download the route script and run the following command.

```
sudo bash route.sh
```

Please follow the instructions of regular server installation setup to complete the installation.

REFERENCES

Image sources: Ubuntu logo, Traefik logo, ml-workspace, nodejs, reactjs, nestjs

3.1.8 Third-party Services

The DTaaS software platform uses third-party software services to provide enhanced value to users.

InfluxDB, Grafana, RabbitMQ and Mosquitto are default services integrated into the DTaaS software platform.

Pre-requisites

All these services run on raw TCP/UDP ports. Thus a direct network access to these services is required for both the DTs running inside the DTaaS software and the PT located outside the DTaaS software.

There are two possible choices here:

- Configure Traefik gateway to permit TCP/UDP traffic
- Bypass Traefik altogether

Unless you are an informed user of Traefik, we recommend bypassing traefik and provide raw TCP/UDP access to these services from the Internet.

The InfluxDB service requires a dedicated hostname. The management interface of RabbitMQ service requires a dedicated hostname as well.

Grafana service can run well behind Traefik gateway. The default Traefik configuration makes permits access to Grafana at URL: http(s): *foo.com/vis*.

Configure and Install

If you have not cloned the DTaaS git repository, cloning would be the first step. In case you already have the codebase, you can skip the cloning step. To clone, do:

```
git clone https://github.com/into-cps-association/DTaaS.git
cd DTaaS/deploy/services
```

The next step in installation is to specify the config of the services. There are two configuration files. The **services.yml** contains most of configuration settings. The **mqtt-default.conf** file contains the MQTT listening port. Update these two config files before proceeding with the installation of the services.

Now continue with the installation of services.

```
yarn install
node services.js
```

Use

After the installation is complete, you can see the following services active at the following ports / URLs.

service	external url
Influx	services.foo.com
Grafana	services.foo.com:3000
RabbitMQ Broker	services.foo.com:5672
RabbitMQ Broker Management Website	services.foo.com:15672
MQTT Broker	services.foo.com:1883
MongoDB database	services.foo.com:27017

The firewall and network access settings of corporate / cloud network need to be configured to allow external access to the services. Otherwise the users of DTaaS will not be able to utilize these services from their user workspaces.

3.2 Integrated Gitlab

3.2.1 Install Gitlab

This guide helps with installation of a dedicated Gitlab service. This Gitlab installation can be used as OAuth2 authorization provider to the DTaaS software. In addition, it is also possible to use the integrated Gitlab for enabling the digital twin DevOps experimental features of the DTaaS.

There are two possible ways you can install Gitlab:

- At dedicated domain name (ex: *gitlab.foo.com*)
- At a URL path on existing WWW server (ex: foo.com/gitlab)

This guide illustrates the installation of Gitlab at: foo.com/gitlab. But the instructions can be adapted to install Gitlab at a dedicated domain name.

Configure and Install

If you have not cloned the DTaaS git repository, cloning would be the first step. In case you already have the codebase, you can skip the cloning step. To clone, do:

```
git clone https://github.com/into-cps-association/DTaaS.git
cd DTaaS/deploy/services/gitlab
```

This directory contains files needed to set up the docker container containing the local GitLab instance.

1. `./data`, `./config`, `./logs` are the directories that will contain data for the GitLab instance
2. `compose.gitlab.yml` and `.env` are the Docker compose and environment files to manage the containerized instance of gitlab

If the DTaaS application and gitlab are to be hosted at https://foo.com, then the client config file (`deploy/config/client/env.js`) needs to use the https://foo.com/gitlab as `REACT_APP_AUTH_AUTHORITY`. In addition, this hosting at https://foo.com also requires changes to config file (`.env.server`).

If the DTaaS application and gitlab are to be hosted at https://localhost, then the client config file (`deploy/config/client/env.local.js`) needs to use the https://localhost/gitlab as `REACT_APP_AUTH_AUTHORITY`. If the application and the integrated gitlab are to be hosted at `https://localhost/gitlab`, then `.env.server` need not be modified.

Edit the `.env` file available in this directory to contain the following variables:

Variable	Example Value	Explanation
GITLAB_HOME	'/home/Desktop/DTaaS/deploy/services/gitlab'	Full path to the DTaaS gitlab directory. This is an absolute path with no trailing slash.
SERVER_DNS	either `foo.com` or `localhost`	The server DNS, if you are deploying with a dedicated server. Remember not use *http(s)* at the beginning of the DNS string.

NOTE: The DTaaS client uses the `react-oidc-context` node package, which incorrectly causes redirects to use the `HTTPS` URL scheme. This is a known issue with the package, and forces us to use `HTTPS` for the DTaaS server. If you are hosting the DTaaS locally, your GitLab instance should be available at https://localhost/gitlab. If you are hosting the DTaaS at https://foo.com, then you Gitlab instance should be available at https://foo.com/gitlab.

Run

NOTE: The GitLab instance operates with the `dtaas-frontend` network, which requires the DTaaS server to be running before you start it. You may refer to secure installation scenarios for the same.

The commands to start and stop the instance are:

```
# (cd deploy/services/gitlab)
docker compose -f compose.gitlab.yml up -d
docker compose -f compose.gitlab.yml down
```

Each time you start the container, it may take a few minutes. You can monitor the progress with `watch docker ps` and check if the gitlab container is `healthy`.

POST-INSTALL CONFIGURATION

Gitlab also requires post-installation configuration.

> **Information**
>
> This configuration needs to be done from within the running container.

```
docker exec -it gitlab bash
```

The configuration file to change is */etc/gitlab/gitlab.rb*. The variables to change are:

```
external_url 'http(s)://foo.com/gitlab'
nginx['listen_port'] = 80
nginx['enable'] = true

nginx['listen_https'] = false
nginx['redirect_http_to_https'] = false
letsencrypt['enable'] = false
```

The `external_url` mentioned about indicates hosting of gitlab at https://foo.com/gitlab. If the gitlab needs to be available at https://localhost/gitlab, then the `external_url` should be https://localhost/gitlab.

Save the changes and reconfigure gitlab by running:

```
# inside the gitlab docker container
gitlab-ctl reconfigure
exit
```

The administrator username for GitLab is: `root`. The password for this user account will be available in: */etc/gitlab/initial_root_password*. Be sure to save this password somewhere, as **this file will be deleted after 24 hours** from the first time you start the local instance.

After running the container, your local GitLab instance will be available at `external_url` specified in *gitlab.rb*, i.e., either at https://foo.com/gitlab or at https://localhost/gitlab.

CREATE USERS

The newly installed gitlab only contains `root` user. More users need to be created for use with DTaaS. Please see the Gitlab docs for further help.

Pending Tasks

This README helps with installation of Gitlab along side DTaaS application. But the OAuth2 integration between Gitlab and DTaaS will still be pending. Follow the integration guide and the runner setup guide to setup the Gitlab integration.

3.2.2 GitLab Integration Guide

This guide helps with integration of a local GitLab instance with a DTaaS server installation, and integrating the OAuth Authorization feature with the DTaaS installation.

After following this guide, the GitLab instance will be integrated as OAuth provider for both DTaaS client application and Traefik Forward Auth backend authorization.

> **Note**
>
> The DTaaS client uses the `react-oidc-context` node package, which incorrectly causes authorization redirects to use the `HTTPS` URL scheme. This is a known issue with the package, and forces us to use `HTTPS` for the DTaaS server. This means your server should be set up to use either https://localhost or https://foo.com. This guide will henceforth use `foo.com` to represent either localhost or a custom domain.

Integration Steps

1. SET UP THE DTAAS SERVER OVER HTTPS

Follow the existing guides to set up the DTaaS web application over HTTPS connection on either localhost (https://localhost) or a custom domain (https://foo.com).

> **Note**
>
> You may ignore steps related to configuring OAuth application tokens at https://gitlab.com. We will be using the initial installation to host the local GitLab instance, on which we will later create the OAuth application tokens.

2. SET UP THE GITLAB INSTANCE

Follow the guide to set up a GitLab instance.

After this step, and once you run `gitlab-ctl reconfigure`, you will have a functioning GitLab instance (at either https://localhost/gitlab or https://foo.com/gitlab). Login credentials of the root user.

3. CREATE OAUTH TOKENS IN GITLAB

Follow these guides to create OAuth Application Tokens for - backend and client. Please note that backend is not required for https://localhost installation.

After this step you will have credentials for the application tokens titled "DTaaS Server Authorization" and "DTaaS Client Authorization", which we will use in the next step.

4. USE VALID OAUTH APPLICATION TOKENS

We can now use the OAuth tokens generated on the GitLab instance to enable authorization.

If the DTaaS application is hosted at https://localhost, then configure the following files:

1. **DTaaS Client Authorization** token in *deploy/config/client/env.local.js*.
2. *deploy/docker/.env.local* Add localpath and username.

If the DTaaS application is hosted at https://foo.com, then configure the following files:

1. **DTaaS Client Authorization** token in *deploy/config/client/env.js*.
2. *deploy/docker/.env.server* - Add localpath and username, OAuth client ID and client secret from the **DTaaS Server Authorization** token

Restart Services

LOCALHOST INSTALLATION

The updated OAuth application configuration needs to be loaded into the **client website** service.

```
cd deploy/docker
docker compose -f compose.local.yml --env-file .env.local up -d --force-recreate client
```

PRODUCTION SERVER INSTALLATION

The updated OAuth application configuration needs to be loaded into the **client website** and the **forward-auth** services.

The production server can be installed with either **http** or **https** option. If it is installed with **http** option, run the following commands.

```
cd deploy/docker
docker compose -f compose.server.yml --env-file .env.server up -d --force-recreate client
docker compose -f compose.server.yml --env-file .env.server up -d --force-recreate traefik-forward-auth
```

If the production server is installed with **https** option, run the following commands.

```
cd deploy/docker
docker compose -f compose.server.secure.yml --env-file .env.server up -d --force-recreate client
docker compose -f compose.server.secure.yml --env-file .env.server up -d --force-recreate traefik-forward-auth
```

Post Setup Usage

If you have set up everything correctly:

1. You will have a functioning path-prefixed GitLab instance available at `https://foo.com/gitlab` that you may use in a similar manner to https://gitlab.com.
2. Data, configuration settings and logs pertaining to the GitLab installation will be available on the DTaaS server within the directory: *deploy/services/gitlab*.
3. Traefik Forward Auth will use the path-prefixed GitLab instance for authorization on the multi-user installation scenario i.e. `foo.com` (but not on `localhost`).

3.2.3 GitLab Runner Integration

This document outlines the steps needed to create a Docker container named `gitlab-runner` which will contain a single runner that will be responsible for the execution of Digital Twins. There are two installation scenarios:

1. **Localhost Installation** - You are using the integrated runner locally with a GitLab instance hosted at `https://localhost/gitlab`.
2. **Server Installation** - You are using the integrated runner with a GitLab instance hosted on a production server. This server may be a remote server and not necessarily your own, and may have TLS enabled with a self-signed certificate.

Following the steps below sets up the integrated runner which can be used to execute digital twins from the Digital Twins Preview Page.

Prerequisites

A GitLab Runner picks up CI/CD jobs by communicating with a GitLab instance. For an explanation of how to set up a GitLab instance that integrates with a DTaaS application, refer to our GitLab instance document and our GitLab integration guide.

The rest of this document assumes you have a running DTaaS application with a GitLab instance running.

Runner Scopes

A GitLab Runner can be configured for three different scopes:

Runner Scope	Description
Instance Runner	Available to all groups and projects in a GitLab instance.
Group Runner	Available to all projects and subgroups in a group.
Project Runner	Associated with one specific project.

We suggest creating **instance runners** as they are the most straightforward, but any type will work. More about these three types can be found on the official GitLab documentation page.

Obtaining A Registration Token

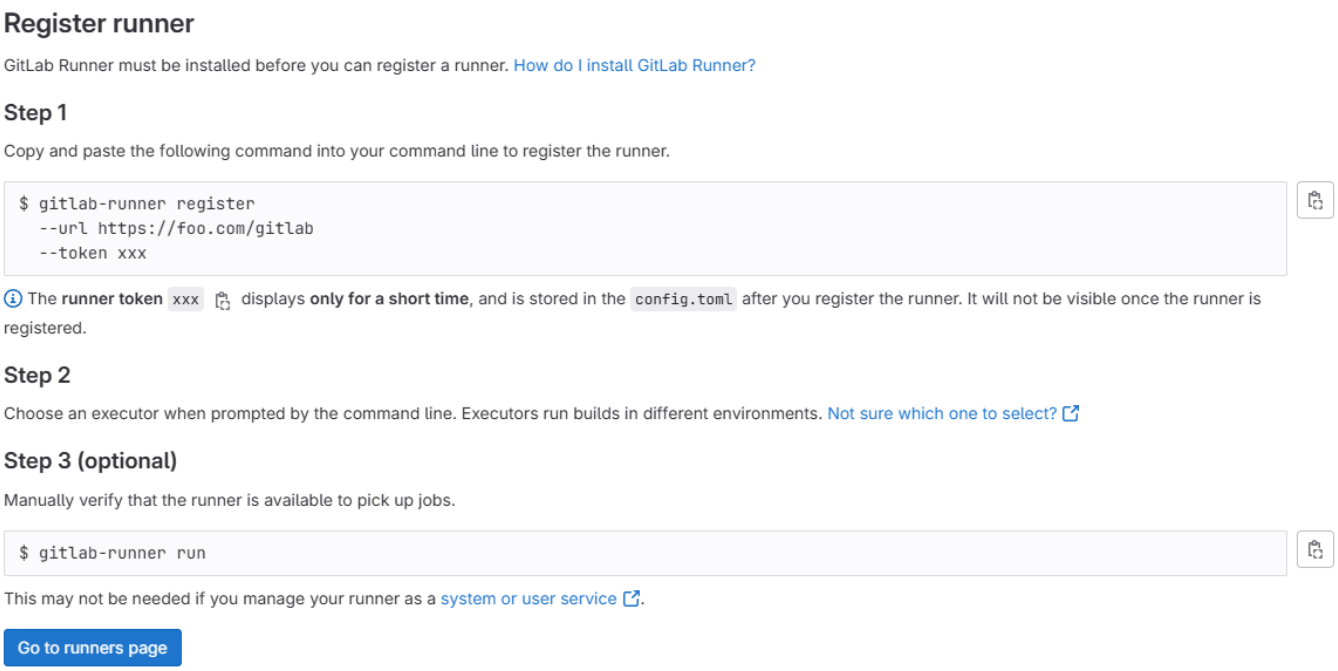
First, we will obtain the token necessary to register the runner for the GitLab instance. Open your GitLab instance (remote or local) and depending on your choice of runner scope, follow the steps given below:

Runner Scope	Steps
Instance Runner	1. On the **Admin** dashboard, navigate to **CI/CD** > **Runners**. 2. Select **New instance runner**.
Group Runner	1. On the **DTaaS** group page, navigate to **Settings** > **CI/CD** > **Runners**. 2. Ensure the **Enable shared runners for this group** option is enabled. 3. On the **DTaaS** group page, navigate to **Build** > **Runners**. 4. Select **New group runner**.
Project Runner	1. On the **DTaaS** group page, select the project named after your GitLab username. 2. Navigate to **Settings** > **CI/CD** > **Runners**. 3. Select **New project runner**.

For any scope you have chosen, you will be directed to a page to create a runner:

1. Under **Platform**, select the Linux operating system.
2. Under **Tags**, add a `linux` tag.
3. Select **Create runner**.

You should then see the following screen:



Be sure to save the generated runner authentication token.

Configuring the Runner

Depending on your installation scenario, the runner setup reads certain configurations settings:

1. **Localhost Installation** - uses `deploy/docker/.env.local`
2. **Server Installation** - uses `deploy/docker/.env.server`

These files are integral to running the DTaaS application, so it will be assumed that you have already configured these.

We need to register the runner with the GitLab instance so that they may communicate with each other. `deploy/services/runner/runner-config.toml` has the following template:

```
[[runners]]
  name = "dtaas-runner-1"
  url = "https://foo.com/gitlab/" # Edit this
  token = "xxx" # Edit this
  executor = "docker"
  [runners.docker]
    tls_verify = false
    image = "ruby:2.7"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    volumes = ["/cache"]
    network_mode = "host" # Disable this in secure contexts
```

1. Set the `url` variable to the URL of your GitLab instance.
2. Set the `token` variable to the runner registration token you obtained earlier.
3. If you are following the server installation scenario, remove the line `network_mode = "host"`.

A list of advanced configuration options is provided on the GitLab documentation page.

Start the GitLab Runner

You may use the following commands to start and stop the `gitlab-runner` container respectively, depending on your installation scenario:

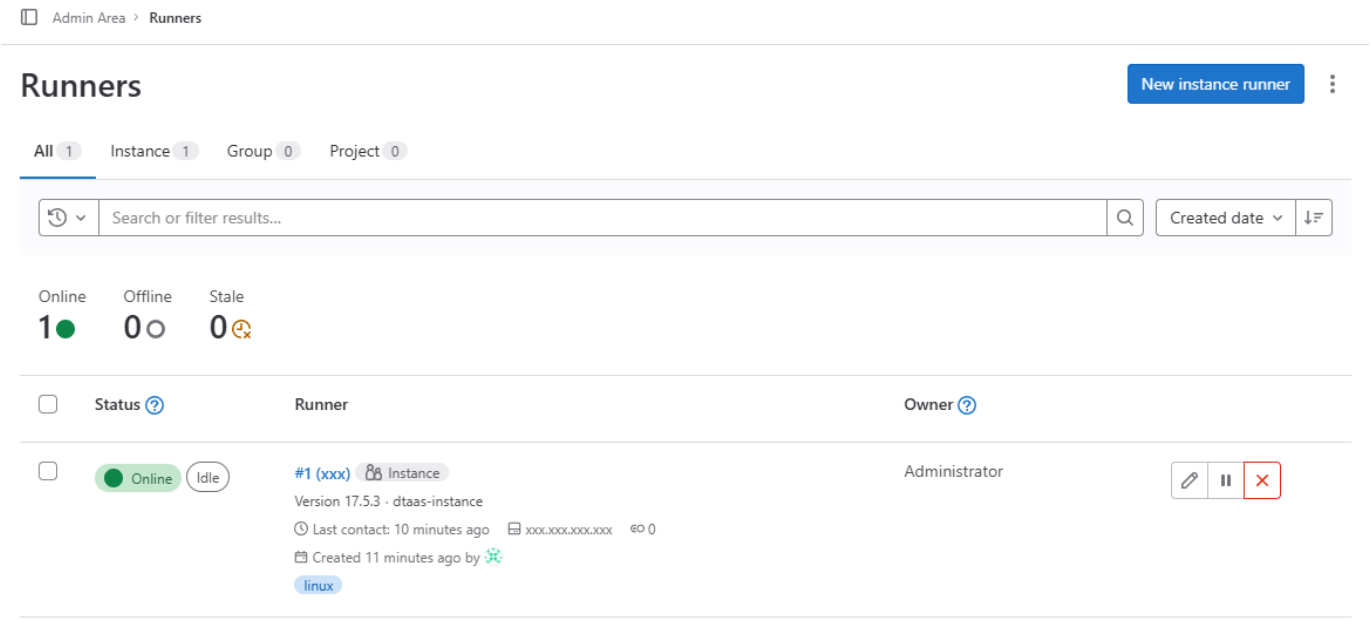
1. Go to the DTaaS home directory (`DTaaS_DIR`) and execute one of the following commands.
2. Localhost Installation

```
docker compose -f deploy/services/runner/compose.runner.local.yml --env-file deploy/docker/.env.local up -d
docker compose -f deploy/services/runner/compose.runner.local.yml --env-file deploy/docker/.env.local down
```

3. Server Installation

```
docker compose -f deploy/services/runner/compose.runner.server.yml --env-file deploy/docker/.env.server up -d
docker compose -f deploy/services/runner/compose.runner.server.yml --env-file deploy/docker/.env.server down
```

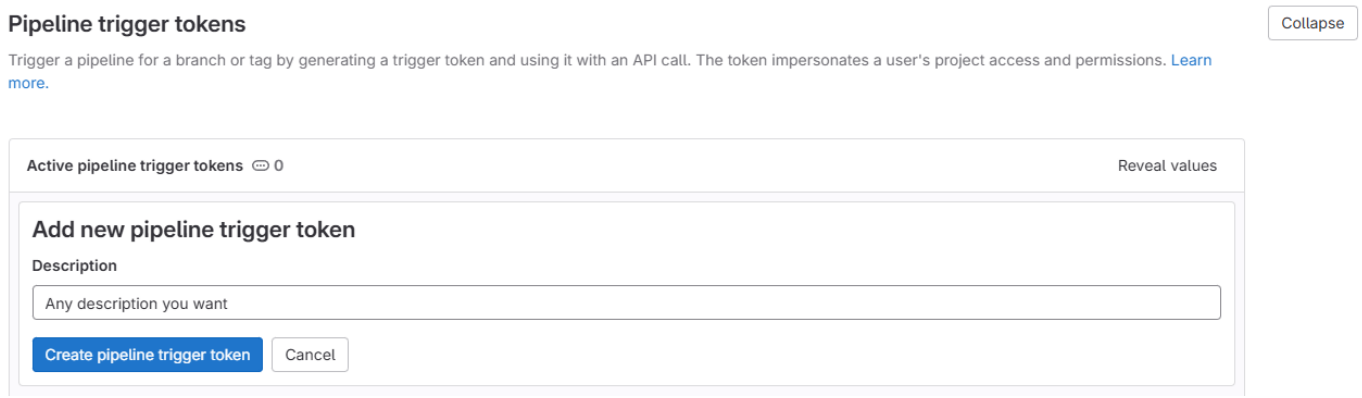
Once the container starts, the runner within it will run automatically. You can tell if the runner is up and running by navigating to the page where you created the runner. For example, an Instance Runner would look like this:



You will now have a GitLab runner ready to accept jobs for the GitLab instance.

Pipeline Trigger Token

The Digital Twins Preview Page uses the GitLab API which requires a Pipeline Trigger Token. Go to your project in the **DTaaS** group and navigate to **Settings** > **CI/CD** > **Pipeline trigger tokens**. Add a new token with any description of your choice.



You can now use the Digital Twins Preview Page to manage and execute your digital twins.

3.3 DTaaS Command Line Interface

This is a command line tool for the INTO-CPS-Association Digital Twins as a Service.

3.3.1 Prerequisite

The DTaaS application with base users and essential containers should be up and running before using the CLI.

3.3.2 Installation

Simply install using:

We recommend installing this in a virutal environment.

Steps to install:

- Change the working folder:

```
cd <DTaaS-directory>/cli
```

- We recommend installing this in a virtual environment. Create and activate a virtual environment.
- To install, simply:

```
pip install dtaas
```

3.3.3 Usage

Setup

The base DTaaS system should be up and running before adding/deleting users with the CLI.

Additionally, Setup the *dtaas.toml* file in the *cli* directory:

- Set *common.server-dns* to domain name of your server. If you want to bring up the server locally, please set this to *"localhost"*.
- Set the *path* to the full system path of the DTaaS directory.

```
[common]
# absolute path to the DTaaS application directory
server-dns = "localhost"
path = "/home/Desktop/DTaaS"
```

Select Template

The *cli* uses YAML templates provided in this directory to create new user workspaces. The templates are:

1. *user.local.yml*: localhost installation
2. *User.server.yml*: multi-user web application application over HTTP
3. *user.server.secure.yml*: multi-user web application application over HTTPS

Please note that the *cli* is not capable of detecting the difference between HTTP and HTTPS modes of web application. Thus if you are serving web application over HTTPS, please do one extra step.

```
cp user.server.secure.yml user.server.yml
```

This will change the user template from insecure to secure.

Add users

To add new users using the CLI, fill in the *users.add* list in *dtaas.toml* with the Gitlab instance usernames of the users to be added

```
[users]
# matching user info must present in this config file
add = ["username1","username2", "username3"]
```

Make sure you are in the *cli* directory.

Then simply:

```
dtaas admin user add
```

The command checks for the existence of `files/<username>` directory. If it does not exist, a new directory with correct file structure is created. The directory, if it exists, must be owned by the user executing **dtaas** command on the host operating system. If the files do not have the expected ownership rights, the command fails.

CAVEATS

This brings up the containers, without the AuthMS authentication.

- Currently the *email* fields for each user in *dtaas.toml* are not in use, and are not necessary to fill in. These emails must be configured manually for each user in the deploy/docker/conf.server files and the *traefik-forward-auth* container must be restarted. This is done as follows:
- Go to the *docker* directory

```
cd <DTaaS>/deploy/docker
```

- Add three lines to the `conf.server` file

```
rule.onlyu3.action=auth
rule.onlyu3.rule=PathPrefix(`/user3`)
rule.onlyu3.whitelist = user3@emailservice.com
```

- Run the command for these changes to take effect:

```
docker compose -f compose.server.yml --env-file .env up -d --force-recreate traefik-forward-auth
```

The new users are now added to the DTaaS instance, with authorization enabled.

Delete users

- To delete existing users, fill in the *users.delete* list in *dtaas.toml* with the Gitlab instance usernames of the users to be deleted.

```
[users]
# matching user info must present in this config file
delete = ["username1","username2", "username3"]
```

- Make sure you are in the *cli* directory.

Then simply:

```
dtaas admin user delete
```

- Remember to remove the rules for deleted users in *conf.server*.

Additional Points to Remember

- The *user add* CLI will add and start a container for a new user. It can also start a container for an existing user if that container was somehow stopped. It shows a *Running* status for existing user containers that are already up and running, it doesn't restart them.

- *user add* and *user delete* CLIs return an error if the *add* and *delete* lists in *dtaas.toml* are empty, respectively.
- '.' is a special character. Currently, usernames which have '.'s in them cannot be added properly through the CLI. This is an active issue that will be resolved in future releases.

3.4 Independent Packages

3.4.1 Independent Packages

The DTaaS development team publishes reusable packages which are then put together to form the complete DTaaS application.

The packages are published on github, npmjs, and docker hub repositories.

The packages on github are published more frequently but are not user tested. The packages on npmjs and docker hub are published at least once per release. The regular users are encouraged to use the packages from npm and docker hub.

A brief explanation of the packages is given below.

Package Name	Description	Documentation for	Availability
dtaas-web	React web application	Not useful as standalone package	docker hub and github
libms	Library microservice	npm package	npmjs and github
		container image	docker hub and github
runner	REST API wrapper for multiple scripts/programs	npm package	npmjs and github
ml-workspace-minimal (fork of ml-workspace)	User workspace	not available	docker hub. Please note that this package is **highly experimental** and only v0.15.0-b2 is usable now.

3.4.2 Library Microservice

Host Library Microservice

The **lib microservice** is a simplified file manager serving files over graphQL and HTTP API.

It has two features:

- provide a listing of directory contents.
- upload and download files

This document provides instructions for installing npm package of library microservice and running the same as a standalone service.

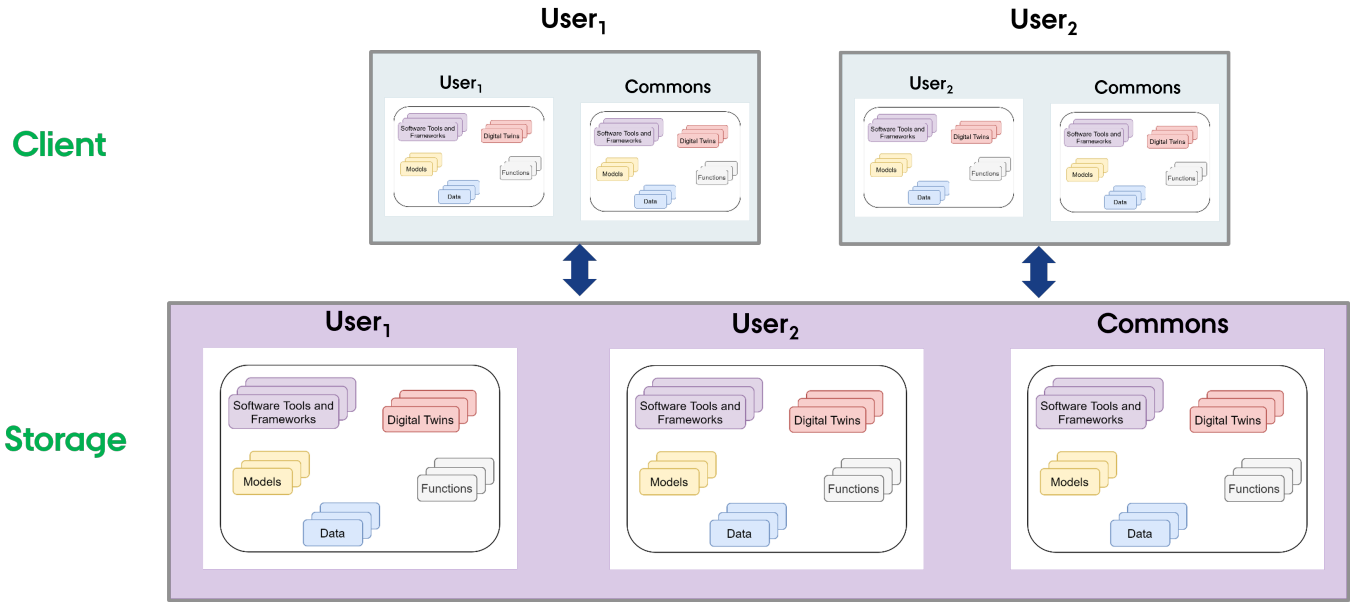
SETUP THE FILE SYSTEM

Outside DTaaS

The package can be used independently of DTaaS. If this is your use case, you do not need any specific file structure. Any valid file directory is sufficient.

Inside DTaaS

The users of DTaaS expect the following file system structure for their reusable assets.



There is a skeleton file structure in DTaaS codebase. You can copy and create file system for your users.

INSTALL

The npm package is available in Github packages registry and on npmjs. **Prefer the package on npmjs over Github**.

Set the registry and install the package with the one of the two following commands

npmjs

```
sudo npm install -g @into-cps-association/libms  # requires no login
```

Github

```
# requires login
sudo npm config set @into-cps-association:registry https://npm.pkg.github.com
```

The *github package registry* asks for username and password. The username is your Github username and the password is your Github personal access token. In order for the npm to download the package, your personal access token needs to have *read:packages* scope.

🚀 USE

Display help.

```
libms -h
```

Please see configuration for explanation of configuration conventions. To use `.env` as configuration file, run

```
libms
```

To run **libms** with a custom config file,

```
libms -c FILE-PATH
libms --config FILE-PATH
```

If the environment file is named something other than `.env`, for example as `.env.libms`, you can run

```
libms -c ".env.libms"
```

You can press `Ctl+C` to halt the application. If you wish to run the microservice in the background, use

```
nohup libms [-c FILE-PATH] & disown
```

The lib microservice is now running and ready to serve files.

This microservice can also serve files in a browser with files transferred over HTTP protocol.

This option needs to be enabled with `-H http.json` flag. A sample http config provided here can be used.

```
nohup libms [-H http.json] & disown
```

The regular file upload and download options become available.

SERVICE ENDPOINTS

The GraphQL URL: `localhost:PORT/lib`

The HTTP URL: `localhost:PORT/lib/files`

The service API documentation is available on user page.

Host Library Microservice

The **lib microservice** is a simplified file manager serving files over graphQL and HTTP API.

It has two features:

- provide a listing of directory contents.
- transfer a file to user.

This document provides instructions for running docker container to provide a stand alone library microservice.

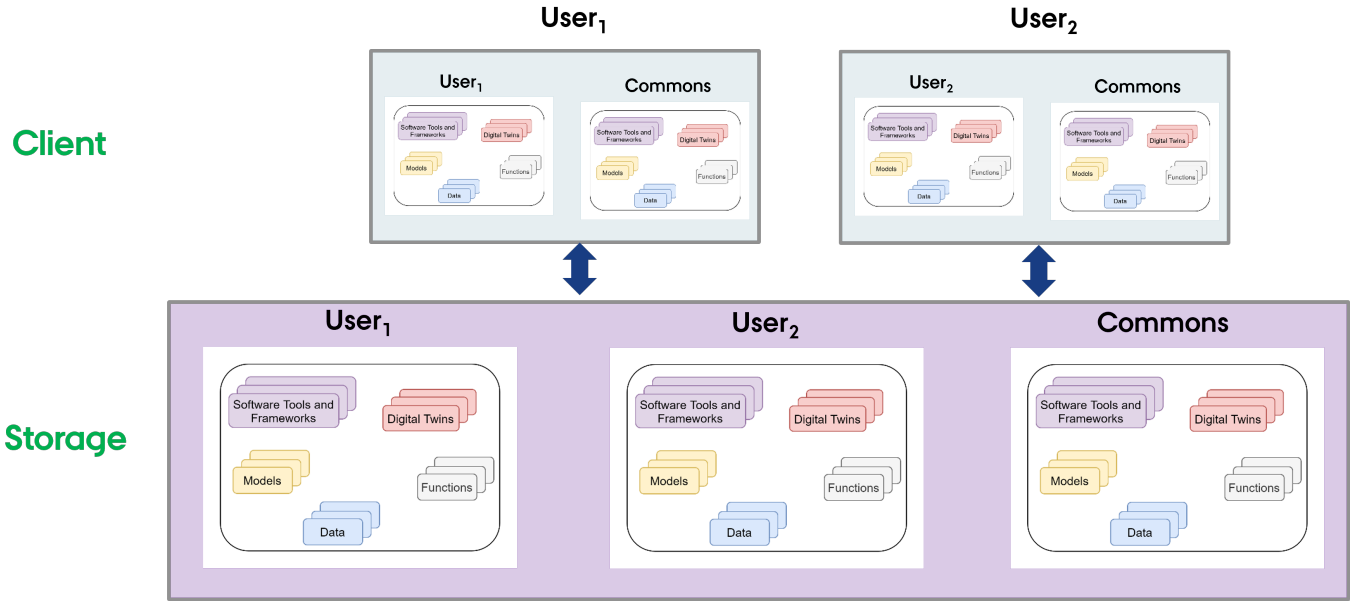
SETUP THE FILE SYSTEM

Outside DTaaS

The package can be used independently of DTaaS. If this is your use case, you do not need any specific file structure. A valid file directory named `files` is sufficient and it should be placed in the directory from which the `compose.lib.yml` will be run.

Inside DTaaS

The users of DTaaS expect the following file system structure for their reusable assets.



There is a skeleton file structure in DTaaS codebase. You can copy and create file system for your users. Remember to name the directory containing the file structure as `files` and place it in the directory from which the `compose.lib.yml` will be run.

🚀 USE

Use the docker compose file to start the service.

```
# To bring up the container
docker compose -f compose.lib.yml up -d
# To bring down the container
docker compose -f compose.lib.yml down
```

SERVICE ENDPOINTS

The GraphQL URL: `localhost:4001/lib`

The HTTP URL: `localhost:4001/lib/files`

The service API documentation is available on user page.

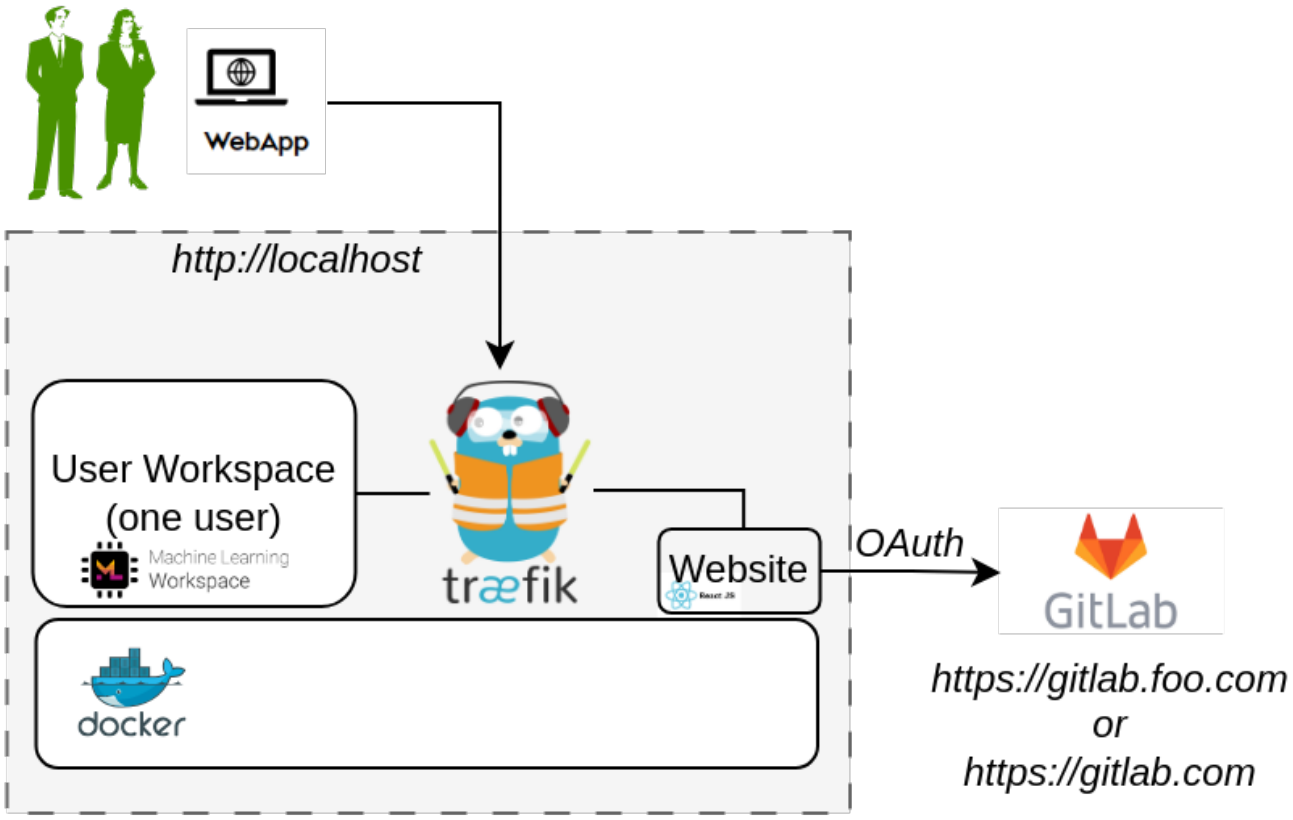
3.5 Guides

3.5.1 Install DTaaS on localhost (GUI)

The installation instructions provided in this README are ideal for running the DTaaS on localhost via a Graphical User Interface (GUI). This installation is ideal for single users intending to use DTaaS on their own computers.

Design

An illustration of the docker containers used and the authorization setup is shown here.



Requirements

The installation requirements to run this docker version of the DTaaS are:

- docker desktop / docker CLI with compose plugin
- User account on *gitlab.com*

> **Tip**
>
> The frontend website requires authorization. The default authorization configuration works for *gitlab.com*. If you desire to use locally hosted gitlab instance, please see the client docs.

Clone Codebase

If you have not cloned the DTaaS git repository, cloning would be the first step. In case you already have the codebase, you can skip the cloning step. To clone, do:

```
git clone https://github.com/into-cps-association/DTaaS.git
cd DTaaS/deploy/services/gitlab
```

In this guide we will assume the contents of the zip file have been extracted to the directory: `/home/DTaaS`.

The path given here is for Linux OS. It can be Windows compatible as well, for example: `C:\\DTaaS`. Make sure to use this path and format in place of `/home/DTaaS` in this guide.

Starting Portainer

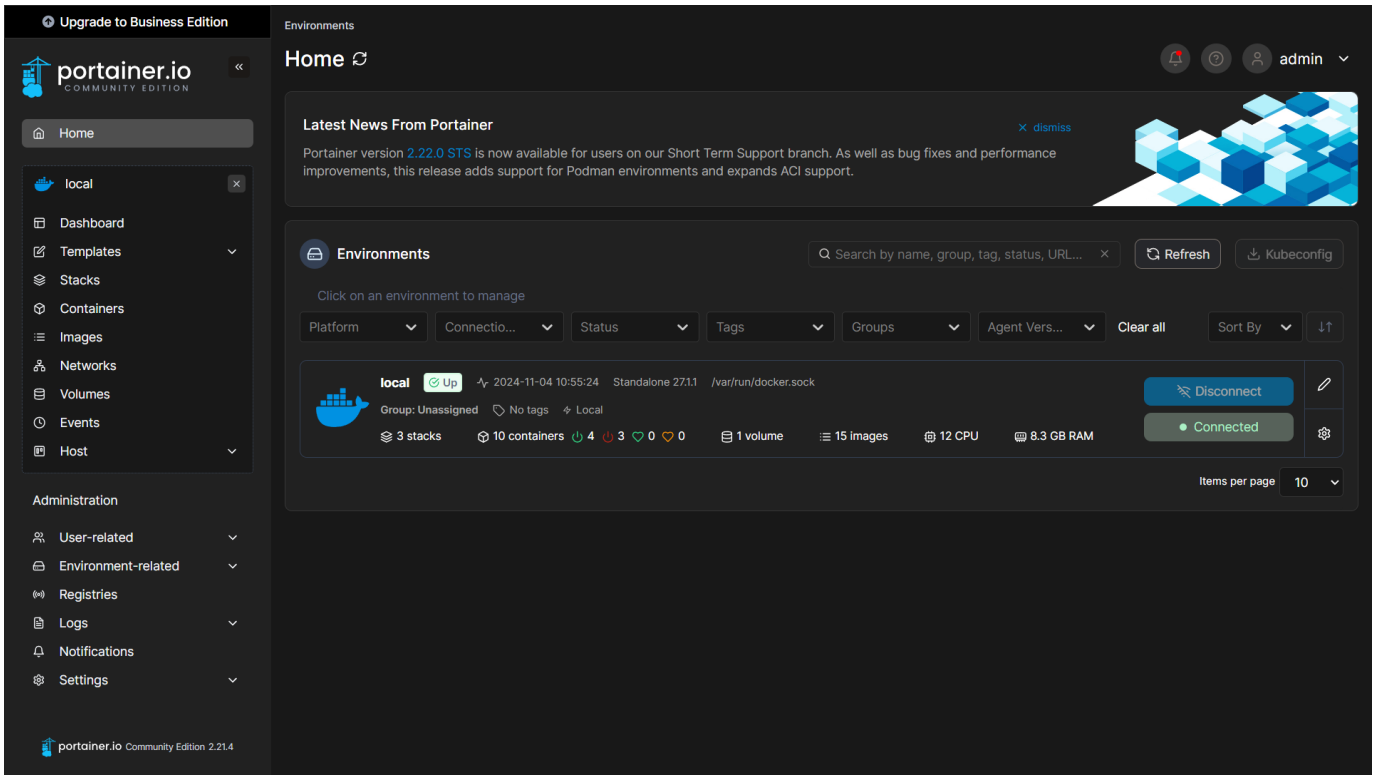
The GUI used to run the application and docker containers will be provided by Portainer Community Edition. It is itself a Docker container that will create a website at `https://localhost:9443`, which will present a graphical interface for starting and stopping the application.

You may follow the official documentation for setting up a Portainer CE Server. Alternatively, open a terminal on your system (Terminal on Linux / MacOS, Powershell on Windows, etc) and copy the following commands into it:

```
docker volume create portainer_data
docker run -d -p 8000:8000 -p 9443:9443 --name portainer --restart=always -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer-ce:2.21.4
```

This will start the Portainer server on your system, which will host its dashboard at `https://localhost:9443`. Follow the Initial Setup Guide to set up an administrator account for Portainer on your system.

Portainer should now be set up on your system, and you can access the dashboard:



The next time you wish to start the Portainer server, run `docker start portainer`.

Configuration

CREATE USER WORKSPACE

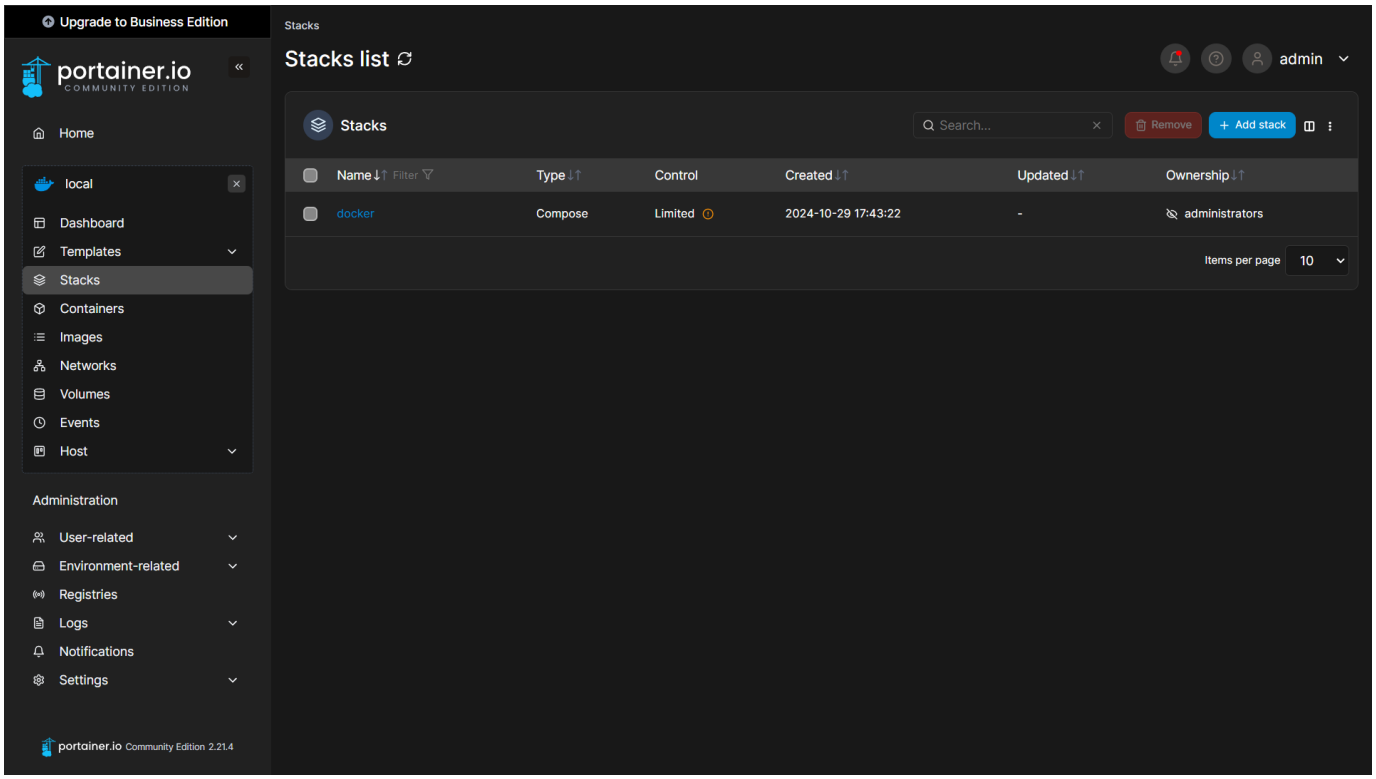
The existing filesystem for installation is setup for `user1`. A new filesystem directory needs to be created for the selected user.

You may use your file explorer or an equivalent application to duplicate the `files/user1` directory and rename it as `files/username` where *username* is the selected username registered on https://gitlab.com.

ALternatively, you may execute the following commands from the top-level directory of the DTaaS project.

```
cp -R files/user1 files/username
```

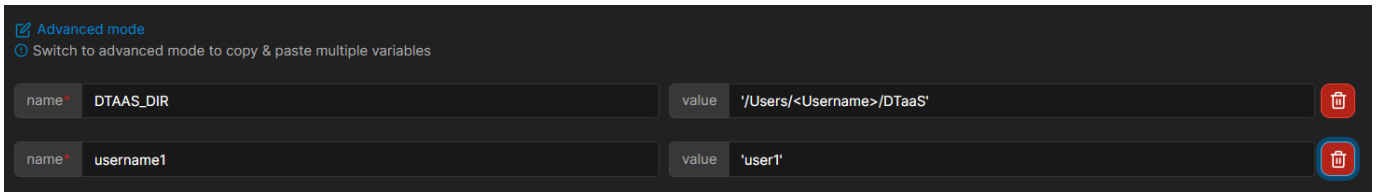
CREATING THE PORTAINER STACK



Portainer Stacks are equivalent to using `docker compose` commands to manage containers.

1. Navigate to the *Stacks* tab on the side panel, and click on the *Add Stack* button.
2. Name the Stack anything descriptive, for example: `dtaas-localhost`.
3. Select the *Upload* build method.
4. Upload the compose file located at `deploy/docker/compose.local.yml`.
5. Select the option to load variables from a .env file, and upload the file `deploy/docker/.env.local`.

Sometimes the `.env.local` file does not show up in the file explorer. You may fix this by selecting the option to show *All Files* rather than those with the extension *.env*.



The `.env.local` file contains environment variables that are used by the compose file. Portainer allows you to modify them as shown in the screenshot above, here is a summary:

URL Path	Example Value	Explanation
DTAAS_DIR	'/home/Desktop/DTaaS'	Full path to the DTaaS directory. This is an absolute path with no trailing slash.
username1	'user1'	Your gitlab username

Tip

Important points to note:

1. The path examples given here are for Linux OS. These paths can be Windows OS compatible paths as well.
2. The client configuration file is located at `deploy/config/client/env.local.js`. If you are following the guide to use HTTPS on localhost, edit the URLs in this file by replacing `http` with `https`.

Once you have configured the environment variables, click on the button *Deploy the stack*.

Use

The application will be accessible at: http://localhost from web browser. Sign in using your https://gitlab.com account.

All the functionality of DTaaS should be available to you through the single page client now.

Limitations

The library microservice is not included in the localhost installation scenario.

References

Image sources: Traefik logo, ml-workspace, reactjs, gitlab

3.5.2 Add User

This page provides steps to adding a user from a DTaaS installation. The username **alice** is used here to illustrate the steps involved in removing a user account.

Please do the following:

1. Add user to Gitlab instance:

Add a new account for the new user on the Gitlab instance. Note the username and email of the new account.

2. Create User Workspace:

Use the DTaaS CLI to bring up the workspaces for new users. This brings up the containers, without the backend authorization.

3. Add backend authorization for the user:

- Go to the *docker* directory

```
cd <DTaaS>/docker
```

- Add three lines to the `conf.server` file

```
rule.onlyu3.action=auth
rule.onlyu3.rule=PathPrefix(`/alice`)
rule.onlyu3.whitelist = alice@foo.com
```

4. Restart the docker container responsible for backend authorization.

```
docker compose -f compose.server.yml --env-file .env up -d --force-recreate traefik-forward-auth
```

5. The new users are now added to the DTaaS instance, with authorization enabled.

3.5.3 Remove User

This page provides steps to remove a user from a DTaaS installation. The username **alice** is used here to illustrate the steps involved in removing a user account.

Please do the following:

1. Remove an existing user with the DTaaS CLI.

2. Remove backend authorization for the user:

- Go to the *docker* directory

```
cd <DTaaS>/docker
```

- Remove these three lines from the `conf.server` file

```
rule.onlyu3.action=auth
rule.onlyu3.rule=PathPrefix(`/alice`)
rule.onlyu3.whitelist = alice@foo.com
```

- Run the command for these changes to take effect:

```
docker compose -f compose.server.yml --env-file .env up -d --force-recreate traefik-forward-auth
```

The extra users now have no backend authorization.

3. Remove users to Gitlab instance (optional):

Please see gitlab docs for more help.

4. The user account is now deleted.

Caveat

You cannot delete the two base users that the DTaaS software is installed with. You can only delete the extra users that have been added to the software.

3.5.4 Add other services

Pre-requisite

You should read the documentation about the already available services

This guide will show you how to add more services. In the following example we will be adding **MongoDB** as a service, but these steps could be modified to install other services as well.

Adding other services requires more RAM and CPU power. Please make sure the host machine meets the hardware requirements for running all the services.

1. Add the configuration:

Select configuration parameters for the MongoDB service.

Configuration Variable Name	Description
username	the username of the root user in the MongoDB
password	the password of the root user in the MongoDB
port	the mapped port on the host machine (default is 27017)
datapath	path on host machine to mount the data from the MongoDB container

Open the file `/deploy/services/services.yml` and add the configuration for MongoDB:

```
services:
    rabbitmq:
        username: "dtaas"
        password: "dtaas"
        vhost: "/"
        ports:
            main: 5672
            management: 15672
    ...
    mongodb:
        username: <username>
        password: <password>
        port: <port>
        datapath: <datapath>
    ...
```

2. Add the script:

The next step is to add the script that sets up the MongoDB container with the configuraiton.

Create new file named `/deploy/services/mongodb.js` and add the following code:

```
#!/usr/bin/node
/* Install the optional platform services for DTaaS */
import { $ } from "execa";
import chalk from "chalk";
import fs from "fs";
import yaml from "js-yaml";

const $$ = $({ stdio: "inherit" });
const log = console.log;
let config;

try {
  log(chalk.blue("Load services configuration"));
  config = await yaml.load(fs.readFileSync("services.yml", "utf8"));
  log(
    chalk.green(
      "configuration loading is successful and config is a valid yaml file"
    )
  );
} catch (e) {
  log(chalk.red("configuration is invalid. Please rectify services.yml file"));
  process.exit(1);
}

log(chalk.blue("Start MongoDB server"));
const mongodbConfig = config.services.mongodb;

try {
  log(
    chalk.green(
      "Attempt to delete any existing MongoDB server docker container"
    )
  );
  await $$`docker stop mongodb`;
  await $$`docker rm mongodb`;
} catch (e) {}

log(chalk.green("Start new Mongodb server docker container"));
await $$`docker run -d -p ${mongodbConfig.port}:27017 \
  --name mongodb \
  -v ${mongodbConfig.datapath}:/data/db \
  -e MONGO_INITDB_ROOT_USERNAME=${mongodbConfig.username} \
  -e MONGO_INITDB_ROOT_PASSWORD=${mongodbConfig.password} \
  --restart always \
  mongo:7.0.3`;
log(chalk.green("MongoDB server docker container started successfully"));
```

3. Run the script:

Go to the directory `/deploy/services/` and run services script with the following commands:

```
yarn install
node mongodb.js
```

The MongoDB should now be available on **services.foo.com:<port>**.

3.5.5 Link services to local ports

Requirements

- User needs to have an account on server2.
- SSH server must be running on server2

To link a port from the service machine (server2) to the local port on the user workspace. You can use ssh local port forwarding technique.

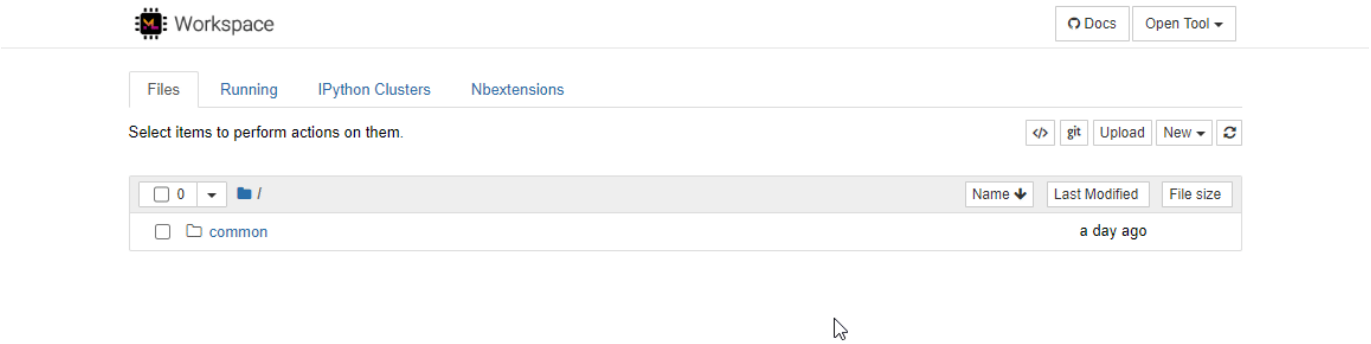
1. Step:

Go to the user workspace, on which you want to map from localhost to the services machine

- e.g. `foo.com/user1`

2. Step:

Open a terminal in your user workspace.



3. Step:

Run the following command to map a port:

```
ssh -fNT -L <local_port>:<destination>:<destination_port> <user>@<services.server.com>
```

Here's an example mapping the RabbitMQ broker service available at 5672 of *services.foo.com* to localhost port 5672.

```
ssh -fNT -L 5672:localhost:5672 vagrant@services.foo.com
```

Now the programs in user workspace can treat the RabbitMQ broker service as a local service running within user workspace.

3.5.6 Make Common Assets Read Only

Why

In some cases you might want to restrict the access rights of some users to the common assets. In order to make the common area read only, you have to change the install script section performing the creation of user workspaces.

> **Note**
>
> These step needs to be performed before installation of the application.

How

To make the common assets read-only for a user, the following changes need to be made to the `compose.server.yml` file.

```
...
user1:
  ....
  volumes:
    - ${DTAAS_DIR}/files/common:/workspace/common:ro
  ....

user2:
  ....
  volumes:
    - ${DTAAS_DIR}/files/common:/workspace/common:ro
  ....
```

Please note the `:ro` at the end of the line. This suffix makes the common assets read only.

If you want to have the same kind of read only restriction for new users as well, please make a similar change in `cli/users.server.yml`.

4. Frequently Asked Questions

4.1 Abreviations

Term	Full Form
DT	Digital Twin
DTaaS	Digital Twin as a Service
PT	Physical Twin

4.2 General Questions

What is DTaaS?

DTaaS is software platform on which you can create and run digital twins. Please see the features page to get a sense of the things you can do in DaaS.

What is the scope and current capabilities of DTaaS?

1. DTaaS is a web based interface to allow you to invoke various tools related to work you want to perform with one or more DTs.
2. DTaaS permits users to run DTs in their private workspaces. These user workspaces are based on Ubuntu 20.04 Operating system.
3. DTaaS can help you create reusable DT assets only if DT asset authoring tools can work in Ubuntu 20.04 xfce desktop environment.
4. DTs are just executables, as far as DTaaS is concerned. Users are not constrained to work with DTs in a certain way. The DTaaS suggests creation of DTs from reusable assets and provides a suggestive structure for DTs. The examples provide more insight into DTaaS way of working with DTs. But this suggested workflow is not binding on the users.
5. DTs can be run as services with REST API from within user workspace which can help with service-level DT composition.

What can not be done inside DTaaS?

1. DTaaS as such won't help you to install DTs that you get from elsewhere.
2. The current user interface of DTaaS web application is heavily reliant on the use of Jupyter lab and notebook. The **Digital Twins** page has Create / Execute / Analyze sections but all point to Jupyter lab. Web interface. The functionality of these pages is still under development.
3. DTaaS does not yet have DevOps like automation capabilities.

Is there any fundamental difference between commercial solutions like Ansys Twin Builder and DTaaS?

Commercial DT platforms like *Ansys Twin Builder* provide tight integration between models, simulation and sensors. This leads to fewer choices in DT design and implementation. In addition, there is a limitation of vendor lockin. On the other hand, DTaas lets users separate DT into reusable assets and combine these assets in a flexible way.

Do you provide licensed software like Matlab?

Proprietary and commercially licensed software is not available by default on the software platform. But users have private workspaces which are based on Linux-based xfce Desktop environment. Users can install proprietary and commercially licensed software in their workspaces. Please see a screencast of using Matlab Simulink within the DTaaS software. The licensed software installed by one user is not available to another user.

4.3 Digital Twin Assets

Can DTaaS be used to create new DT assets?

The core feature of DTaaS software is to help users create DTs from assets already available in the library. Create Library Assets However, it is possible for users to take advantage of services available in their workspace to install asset authoring tools in their own workspace. These authoring tools can then be used to create and publish new assets. User workspaces are private and are not shared with other users. Thus any licensed software tools installed in their workspace is only available to them.

4.4 Digital Twin Models

Can DTaaS create new DT models?

DTaaS is not a model creation tool. You can put model creation tool inside DTaaS and create new models. The DTaaS itself does not create digital twin models but it can help users create digital twin models. You can run Linux desktop / terminal tools inside the DTaaS. So you can create models inside DTaaS and run them using tools that can run in Linux. The Windows only tools can not run in DTaaS.

How can DTaaS help to design geometric model? Does it support 3D modeling and simulation?

Well, DTaaS by itself does not produce any models. DTaaS only provides a platform and an ecosystem of services to facilitate digital twins to be run as services. Since each user has a Linux OS at their disposal, they can also run digital twins that have graphical interface. In summary, DTaaS is neither a modeling nor simulation tool. If you need these kinds of tools, you need to bring them onto the platform. For example, if you need Matlab for your work, you need to bring he licensed Matlab software.

Can DTaaS support only the information models (or behavioral models) or some other kind of models?

The DTaaS as such is agnostic to the kind of models you use. DTaaS can run all kinds of models. This includes behavioral and data models. As long as you have models and the matching solvers that can run in Linux OS, you are good to go in DTaaS. In some cases, models and solvers (tools) are bundled together to form monolithic DTs. The DTaaS does not limit you from running such DTs as well. DTaaS does not provide dedicated solvers. But if you can install a solver in your workspace, then you don't need the platform to provide one.

Does it support XML-based representation and ontology representation?

Currently No. **We are looking for users needing this capability. If you have concrete requirements and an example, we can discuss a way of realizing it in DTaaS**.

4.5 Communication Between Physical Twin and Digital Twin

How can DTaaS control the physical entity? Which technologies it uses for controlling the physical world?

At a very abstract level, there is a communication from physical entity to digital entity and back to physical entity. How this communication should happen is decided by the person designing the digital entity. The DTaaS can provide communication services that can help you do this communication with relative ease. You can use InfluxDB, RabbitMQ and Mosquitto services hosted on DTaaS for two communication between digital and physical entities.

How would you measure a physical entity like shape, size, weight, structure, chemical attributes etc. using DTaaS? Any specific technology used in this case?

The real measurements are done at physical twin which are then communicated to the digital twin. Any digital twin platform like DTaaS can only facilitate this communication of these measurements from physical twin. The DTaaS provides InfluxDB, RabbitMQ and Mosquitto services for this purpose. These three are probably most widely used services for digital twin communication. Having said that, DTaaS allows you to utilize other communication technologies and services hosted elsewhere on the Internet.

How can real-time data differ from static data and what is the procedure to identify dynamic data? Is there any UI or specific tool used here?

DTaaS can not understand the static or dynamic nature of data. It can facilitate storing names, units and any other text description of interesting quantities (weight of batter, voltage output etc). It can also store the data being sent by the physical twin. The distinction between static and dynamic data needs to be made by the user. Only metadata of the data can reveal such more information about the nature of data. A tool can probably help in very specific cases, but you need metadata. If there is a human being making this distinction, then the need for metadata goes down but does not completely go away. In some of the DT platforms supported by manufacturers, there is a tight integration between data and model. In this case, the tool itself is taking care of the metadata. The DTaaS is a generic platform which can support execution of digital twins. If a tool can be executed on a Linux desktop / commandline, the tool can be supported within DTaaS. The tool (ex. Matlab) itself can take care of the metadata requirements.

4.6 Data Management

Can DTaaS collect data directly from sensors?

Yes via platform services.

Does DTaaS support data collection from different sources like hardware, software and network? Is there any user interface or any tracking instruments used for data collection?

The DTaaS provids InfluxDB, RabbitMQ, MQTT and MongoDB services. Both the physical twin and digital twin can utilize these protocols for communication. The IoT (time-series) data can be collected using InfluxDB and MQTT broker services. There is a user interface for InfluxDB which can be used to analyze the data collected. Users can also manually upload their data files into DTaaS.

Is DTaaS able to transmit data to cloud in real time?

Yes via platform services.

Which transmission protocol does DTaaS allow?

InfluxDB, RabbitMQ, MQTT and anything else that can be used from Cloud service providers.

Does DTaaS support multisource information and combined multi sensor input data? Can it provide analysis and decision-supporting inferences?

You can store information from multiple sources. The existing InfluxDB services hosted on DTaaS already has a dedicated Influx / Flux query language for doing sensor fusion, analysis and inferences.

Which kinds of visualization technologies DTaaS can support (e.g. graphical, geometry, image, VR/AR representation)?

Graphical, geometric and images. If you need specific licensed software for the visualization, you will have to bring the license for it. DTaaS does not support AR/VR.

4.7 Platform Native Services on DTaaS Platform

Is DTaaS able to detect the anomalies about-to-fail components and prescribe solutions?

This is the job of a digital twin. If you have a ready to use digital twin that does the job, DTaaS allows others to use your solution.

4.8 Comparison with other DT Platforms

All the DT platforms seem to provide different features. Is there a comparison chart?

Here is a qualitative comparison of different DT integration platforms:

Legend: high performance (**H**), mid performance (**M**) and low performance (**L**)

DT Platforms	License	DT Development Process	Connectivity	Security	Processing power, performance and Scalability	Data
Microsoft Azure DT	Commercial Cloud	H	H	H	M	H
AWS IOT Greengrass	Open source commercial	H	H	H	M	H
Eclipse Ditto	Open source	M	H	M	H	H
Asset Administration Shell	Open source	H	H	L	H	M
PTC Thingworx	Commercial	H	H	H	H	H
GE Predix	Commercial	M	H	H	M	L
AU's DTaaS	Open source	H	H	L	L	M

Adopted by Tanusree Roy from Table 4 and 5 of the following paper.

Ref: Naseri, F., Gil, S., Barbu, C., Cetkin, E., Yarimca, G., Jensen, A. C., ... & Gomes, C. (2023). Digital twin of electric vehicle battery systems: Comprehensive review of the use cases, requirements, and platforms. Renewable and Sustainable Energy Reviews, 179, 113280.

All the comparisons between DT platforms seems so confusing. Why?

The fundamental confusion comes from the fact that different DT platforms (Azure DT, GE Predix) provide different kind of DT capabilities. You can run all kinds of models natively in GE Predix. In fact you can run models even next to (on) PTs using GE Predix. But you cannot natively do that in Azure DT service. You have to do the leg work of integrating with other Azure services or third-party services to get the kind of capabilities that GE Predix natively provides in one interface. The takeaway is that we pick horses for the courses.

4.9 GDPR Concerns

Does your platform adhere to GDPR compliance standards? If so, how?

The DTaaS software platform does not store any personal information of users. It only stores username to identify users and these usernames do not contain enough information to deduce the true identify of users.

Which security measures are deployed? How is data encrypted (if exists)?

The default installation requires a HTTPS terminating reverse proxy server from user to the DTaaS software installation. The administrators of DTaaS software can also install HTTPS certificates into the application. The codebase can generate HTTPS application and the users also have the option of installing their own certificates obtained from certification agencies such as LetsEncrypt.

What security measures does your cloud provider offer?

The current installation of DTaaS software runs on Aarhus University servers. The university network offers firewall access control to servers so that only permitted user groups have access to the network and physical access to the server.

How is user access controlled and authenticated?

There is a two-level authorization mechanism in place in each default installation of DTaaS. The first-level is HTTP basic authorization over secure HTTPS connection. The second-level is the OAuth PKCE authorization flow for each user. The OAuth authorization is provider by a Gitlab instance. The DTaaS does not store the account and authorization information of users.

Does you platform manage personal data? How is data classified and tagged based on the sensitivity? Who has access to the critical data?

The platform does not store personal data of users.

How are identities and roles managed within the platform?

There are two roles for users on the platform. One is the administrator and the other one is user. The user roles are managed by the administrator.

5. Developer

5.1 Contributors Guide

Welcome to the Digital Twin as a Service (DTaaS) contributing guide

Thank you for investing your time in contributing to our project!

Read our Code of Conduct to keep our community approachable and respectable.

In this guide you will get an overview of the contribution workflow from opening an issue, creating a PR, reviewing, and merging the PR.

5.1.1 Project Goals

It helps development team members get familiar with the DTaaS project software design, and development processes. Please see developer-specific Slides, Video, and Research paper.

5.1.2 💻 Development Environment

Please use the steps given here to have suitable development environment.

DevContainers

There is a devcontainer configuration (`.devcontainer/devcontainer.json`) for the project. Please use it to get a dockerized development environment. DevContainer is the easiest way to get started.

Ubuntu/Linux

The code base has been developed for most part on Ubuntu/Linux Operating System.Thus certain parts of the code base might have bugs when run on Windows. At the moment, only runner has problems running on non-Linux OS.

The development environment can be installed by using the following scripts.

```
bash script/env.sh
bash script/docker.sh
```

⚠️ The docker images are large and are likely to consume about 5GB of bandwidth and 15GB of space. You will have to download the docker images on a really good network.

Windows

The development environment scripts for Windows are still buggy. Any help in improving them is greatly appreciated. Given that, caveat, please use the following installation steps for Windows.

Two powershell installation scripts, namely `base.ps1` and `env.ps1` are available to install the required software packages. But errors might crop up due to missing environment variables. The potential errors are:

1. `npm is not recognized.....` in `base.ps1`.
2. `gem is not recognized.....` in `env.ps1`

If you encounter these errors, remember to include *node* and *ruby* installation locations in **PATH** environment variable (`Settings --> search for "system environment variables" --> Advanced --> Environment Variables --> PATH`).

The `base.ps1` and `env.ps1` scripts can be run again after setting the correct **PATH** environment variable.

PRE-INSTALL NODEJS AND RUBY SOFTWARE

Another way to solve the **PATH** environment problem is to install Nodejs and Ruby software packages before running the powershell scripts.

1. Install the latest stable version of NodeJS from the official NodeJS website.
2. Install Ruby from official Ruby website and follow all the defaults during the installation.

RUN SCRIPTS

Then, open powershell with **administrative** priviledges and run the following commands in the given order:

```
powershell -F script/base.ps1
powershell -F script/env.ps1
powershell -F script/docker.ps1
```

⚠️ The docker images are large and are likely to consume about 5GB of bandwidth and 15GB of space. You will have to download the docker images on a really good network.

5.1.3 🏗 Development Workflow

To manage collaboration by multiple developers on the software, a development workflow is in place. Each developer should follow these steps:

1. Fork of the main repository into your github account.
2. Setup Code Climate and Codecov for your fork. The codecov does not require secret token for public repositories.
3. Install git-hooks for the project.
4. Use Fork, Branch, PR workflow.
5. Work in your fork and open a PR from your working branch to your `feature/distributed-demo` branch. The PR will run all the github actions, code climate and codecov checks.
6. Resolve all the issues identified in the previous step.
7. Once changes are verified, a PR should be made to the `feature/distributed-demo` branch of the upstream DTaaS repository.
8. The PR will be merged after checks by either the project administrators or the maintainers.

Remember that every PR should be meaningful and satisfies a well-defined user story or improve the code quality.

5.1.4 👁 Code Quality

The project code qualities are measured based on:

- Linting issues identified by Code Climate
- Test coverage report collected by Codecov
- Successful github actions

Code Climate

Code Climate performs static analysis, linting and style checks. Quality checks are performed by codeclimate are to ensure the best possible quality of code to add to our project.

While any new issues introduced in your code would be shown in the PR page itself, to address any specific issue, you can visit the issues or code section of the codeclimate page.

It is highly recommended that any code you add does not introduce new quality issues. If they are introduced, they should be fixed immediately using the appropriate suggestions from Code Climate, or in worst case, adding a ignore flag (To be used with caution).

Codecov

Codecov keeps track of the test coverage for the entire project. For information about testing and workflow related to that, please see the testing page.

Github Actions

The project has multiple github actions defined. All PRs and direct code commits must have successful status on github actions.

5.2 Contributor Covenant Code of Conduct

5.2.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

5.2.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

5.2.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

5.2.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

5.2.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at Open new issue. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

5.2.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

5.2.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

5.3 Secrets for Github Action

The Github actions require the following secrets to be obtained from docker hub:

Secret Name	Explanation
`DOCKERHUB_SCOPE`	Username or organization name on docker hub
`DOCKERHUB_USERNAME`	Username on docker hub
`DOCKERHUB_TOKEN`	API token to publish images to docker hub, with `Read`, `Write` and `Delete` permissions
`NPM_TOKEN`	Token to publish npm packages to the default npm registry.

Remember to add these secrets to Github Secrets Setting of your fork.

5.4 System

5.4.1 System Overview

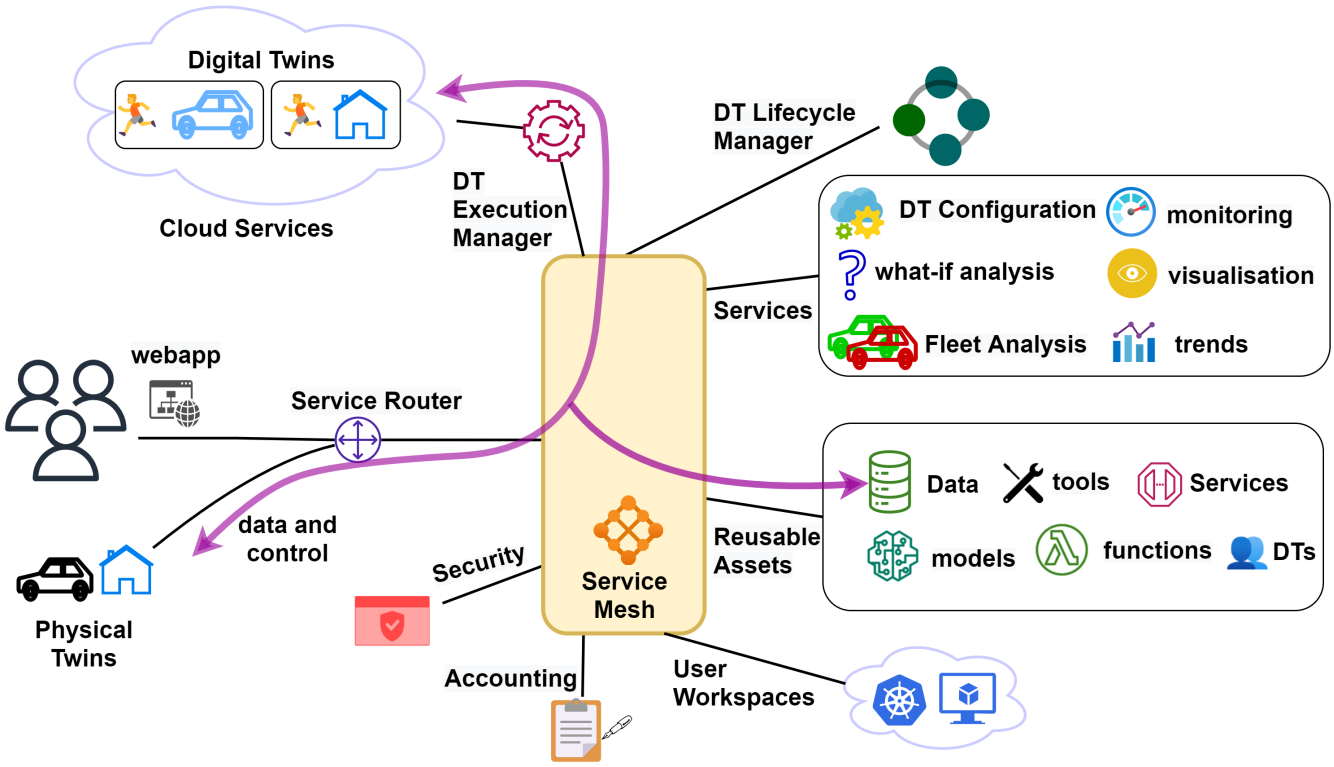
User Requirements

The DTaaS software platform users expect a single platform to support the complete DT lifecycle. To be more precise, the platform users expect the following features:

1. **Author** – create different assets of the DT on the platform itself. This step requires use of some software frameworks and tools whose sole purpose is to author DT assets.
2. **Consolidate** – consolidate the list of available DT assets and authoring tools so that user can navigate the library of reusable assets. This functionality requires support for discovery of available assets.
3. **Configure** – support selection and configuration of DTs. This functionality also requires support for validation of a given configuration.
4. **Execute** – provision computing infrastructure on demand to support execution of a DT.
5. **Explore** – interact with a DT and explore the results stored both inside and outside the platform. Exploration may lead to analytical insights.
6. **Save** – save the state of a DT that's already in the execution phase. This functionality is required for on demand saving and re-spawning of DTs.
7. **Services** – integrate DTs with on-platform or external services with which users can interact with.
8. **Share** – share a DT with other users of their organisation.

System Architecture

The figure shows the system architecture of the the DTaaS software platform.



SYSTEM COMPONENTS

The users interact with the software platform using a webapp. The service router is a single point of entry for direct access to the platform services. The service router is responsible for controlling user access to the microservice components. The service mesh enables discovery of microservices, load balancing and authorization functionalities.

In addition, there are microservices for catering to managing DT reusable assets, platform services, DT lifecycle manager, DT execution manager, accouting and security. The microservices are complementary and composable; they fulfil core requirements of the system.

The microservices responsible for satisfying the user requirements are:

1. **The security microservice** implements role-based access control (RBAC) in the platform.
2. **The accounting microservice** is responsible for keeping track of the live status of platform, DT asset and infrastructure usage. Any licensing, usage restrictions need to be enforced by the accounting microservice. Accounting is a pre-requisite to commercialisation of the platform. Due to significant use of external infrastructure and resources via the platform, the accounting microservice needs to interface with accounting systems of the external services.
3. **User Workspaces** are virtual environments in which users can perform lifecycle operations on DTs. These virtual environments are either docker containers or virtual machines which provide desktop interface to users.
4. **Reusable Assets** are assets / parts from which DTs are created. Further explation is available on the assets page
5. **Services** are dedicated services available to all the DTs and users of the DTaaS platform. Services build upon DTs and provide user interfaces to users.
6. **DT Execution Manager** provides virtual and isolated execution environments for DTs. The execution manager is also responsible for dynamic resource provisioning of cloud resources.
7. **DT Lifecycle Manager** manages the lifecycle operations on all DTs. It also directs *DT Execution Manager* to perform execute, save and terminate operations on DTs.

If you are interested, please take a look at the C4 architectural diagram.

A mapping of the architectural components to related pages in the documentation is available in the table.

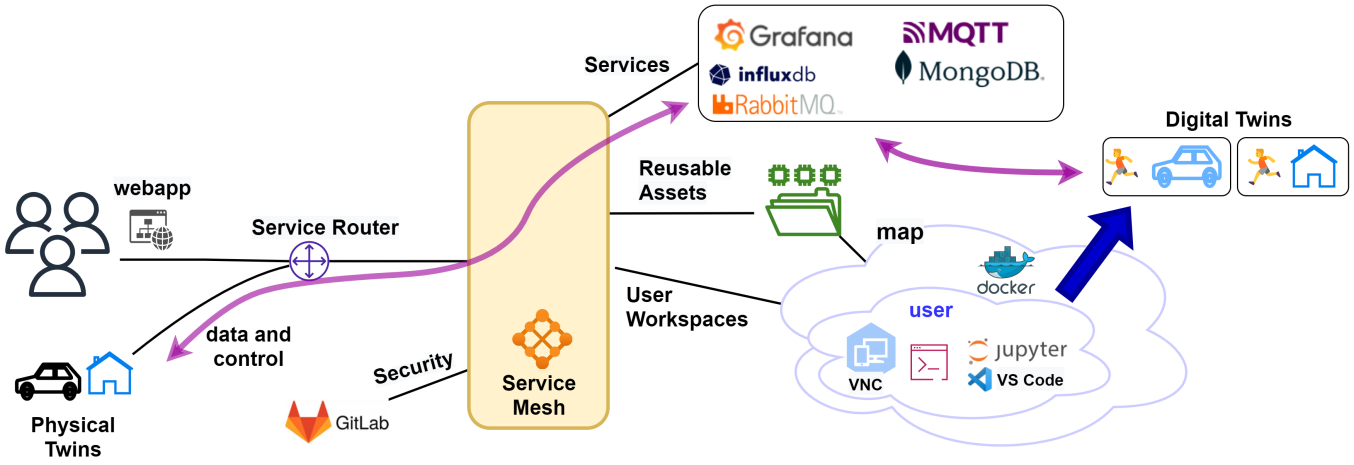
System Component	Doc Page(s)
Service Router	Traefik Gateway
Web Application	React Webapplication
Reusable Assets	Library Microservice
Digital Twins and DevOps	Integrated Gitlab
Services	Third-party Services (MQTT, InfluxDB, RabbitMQ, Grafana and MongoDB)
DT Lifecycle	Not available yet
Security	Gitlab client OAuth and server OAuth
Digital Twins as Services	DT Runner
Accounting	Not available yet
Execution Manager	Not available yet

References

Font sources: fileformat

5.4.2 Current Status

The DTaaS software platform is currently under development. Crucial system components are in place with ongoing development work focusing on increased automation and feature enhancement. The figure below shows the current status of the development work.



A C4 representation of the same diagram is also available.

User Security

There is a two-level authorization mechanisms in place for the react website and the Traefik gateway.

The react website component uses Gitlab for user authorization using OAuth protocol.

GATEWAY AUTHORIZATION

The Traefik gateway has OAuth2 web server authorization provided by Traefik-forward-auth microservice. This authorization protects all the microservices and workspaces running in the backend.

User Workspaces

All users have dedicated dockerized-workspaces. These docker-images are based on container images published by mltooling group.

Thus DT experts can develop DTs from existing DT components and share them with other users. A file server has been setup to act as a DT asset repository. Each user gets space to store private DT assets and also gets access to shared DT assets. Users can synchronize their private DT assets with external git repositories. In addition, the asset repository transparently gets mapped to user workspaces within which users can perform DT lifecycle operations. There is also a library microservice which in the long-run will replace the file server.

Users can run DTs in their workspaces and also permit remote access to other users. There is already shared access to internal and external services. With these two provisions, users can treat live DTs as service components in their own software systems.

Platform Services

There are four external services integrated with the DTaaS software platform. They are: InfluxDB, Grafana, RabbitMQ, MQTT, and MongoDB.

These services can be used by DTs and PTs for communication, storing and visualization of data. There can also be monitoring services setup based on these services.

Development Priorities

The development priorities for the DTaaS software development team are:

- DevOps from React frontend website
- Upgrade software stack of user workspaces
- Increased automation of installation procedures
- DT Runner (API Interface to DT)
- DT Configuration DSL ín the form of YAML schema

Your contributions are highly welcome.

References

Font sources: fileformat

5.5 OAuth2 Authorization

5.5.1 OAuth 2.0 Summary

The Auth MS works on the OAuth 2.0 RFC. This document provides a brief summary of the working of the OAtuh 2.0 technology.

Entities

OAuth2, as used for user identity verification, has 3 main entities:

- **The User:** This is the entity whose identity we are trying to verify/know. In our case, this is the same as the user of the DTaaS software.
- **The Client:** This is the entity that wishes to know/verify the identity of a user. In our case, this is the Auth MS (initialised with a Gitlab application). This shouldn't be confused with the frontend website of DTaaS (referred to as Client in the previous section).
- **The OAuth2 Identity Provider:** This is the entity that allows the client to know the identity of the user. In our case, this is GitLab. Most commonly, users have an existing, protected account with this entity. The account is registered using a unique key, like an email ID or username and is usually password protected so that only that specific user can login using that account. After the user has logged in, they will be asked to approve sharing their profile information with the client. If they approve, the client will have access to the user's email id, username, and other profile information. This information can be used to know/verify the identity of the user.

Note: In general, it is possible for the Authorization server (which asks user for approval) and the Resource (User Identity) provider to be 2 different servers. However, in our case the Gitlab instance itself handles both the functions, through different API endpoints. The concepts remain the same. Thus, we only discuss the 3 main entities, the User, the OAuth2 Client and the Gitlab instance in our discussion.

THE OAUTH2 CLIENT

Many sites allow you to initialise an OAuth2 client. For our purposes, we will use Gitlab itself, by making an "application" in Gitlab. However, it is not necessary to initialise a client using the same website as the identity provider. These are separate things. Our OAuth2 client is initialized by creating and configuring a Gitlab instance-wide application. There are two main things in this configuration:

- **Redirect URI**: It is the URI where the users are redirected to after they approve sharing information with the client.
- **Scopes:** These are the types and levels of access that the client can have over the user's profile. For our purposes, we only require the read user scope, which allows us to access the user's profile information for knowing the identity.

After the GitLab application is successfully created, we are provided a Client ID and Client Secret. This means our initialization is complete. This Client ID and Client Secret can be used in any application, essentially making that application the OAuth2 Client. This is why the Client secret should never be shared. We will use this Client ID and Client secret in our Auth MS, making it an OAuth2 Client application. It will now be able to follow the OAuth2 workflow to verify the identity of users.

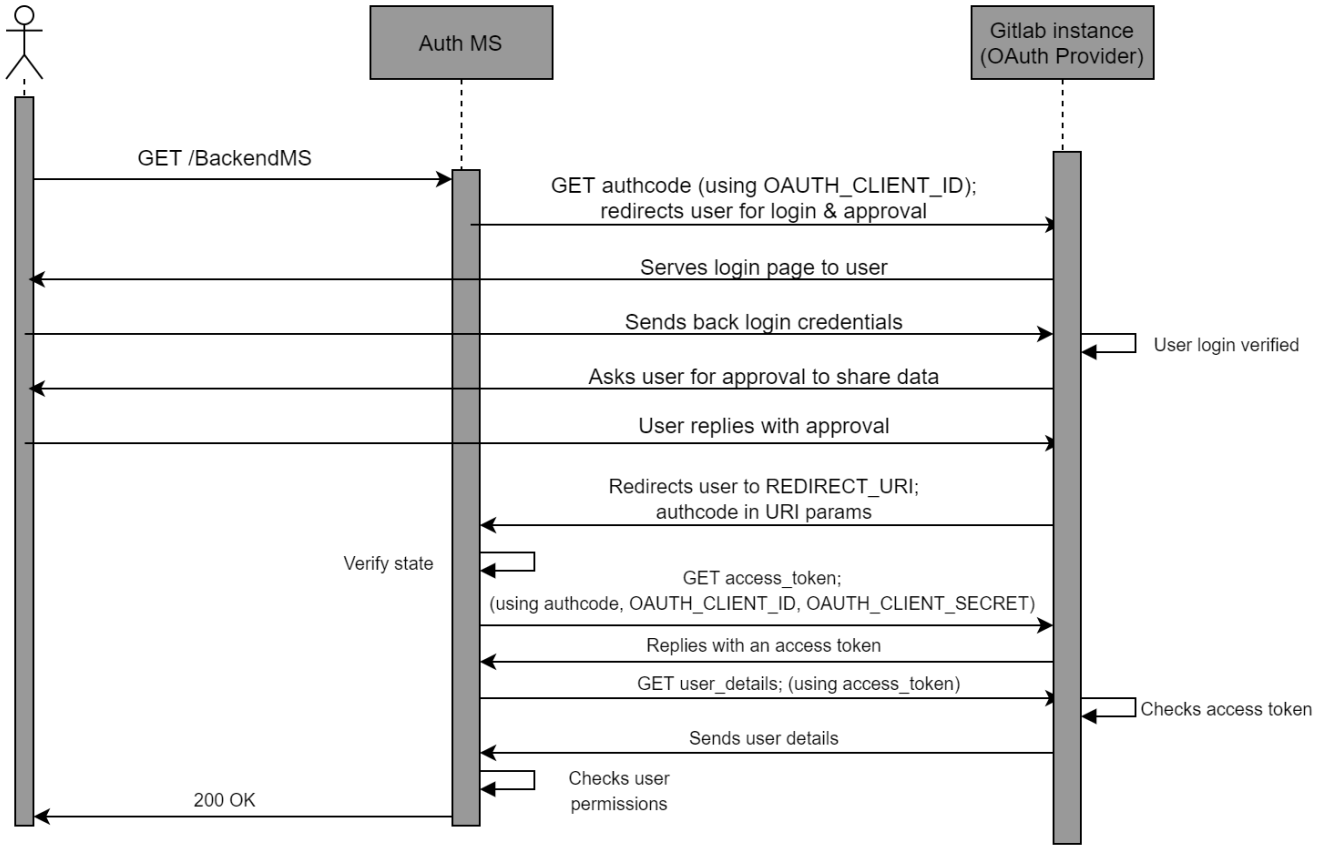
OAuth 2.0 Workflows

Two major different OAuth2.0 flows are used in DTaaS.

OAUTH2 AUTHORIZATION CODE FLOW

This flow involves several steps and the exchange of an authorization code for an access tokens to ensure secure authorization. This flow is used for the DTaaS AuthMS, which is responsible for securing all backend DTaaS services

The OAuth2 workflow is initiated by the Client (Auth MS) whenever it requires knowing the identity of the user. Briefly, the flow starts when the Auth MS sends an authorization request to Gitlab. The Auth MS tries to obtain an access token, using which it can gather user information. Once it has user information, it can know the identity of the user and check whether the user has permission to access the requested resource.



The requests made by the Auth MS to the OAuth2 provider are abbreviated. A detailed explanation of the workflow for DTaaS specifically can be found in the AuthMS implementation docs

OAUTH2 PKCE (PROOF KEY FOR CODE EXCHANGE) FLOW

This is an extension to the OAuth2 Authorization Code Flow designed to provide an additional layer of security, particularly for public clients that cannot securely store client secrets. PKCE mitigates certain attack vectors, like authorization code interception.

The DTaaS client website login works based on the PKCE OAuth2.0 flow. More information about the details of this flow can be found here

5.5.2 System Design of DTaaS Authorization Microservice

DTaaS requires backend authorization to protect its backend services and user workspaces. This document details the system design of the DTaaS Auth Microservice which is responsible for the same.

Requirements

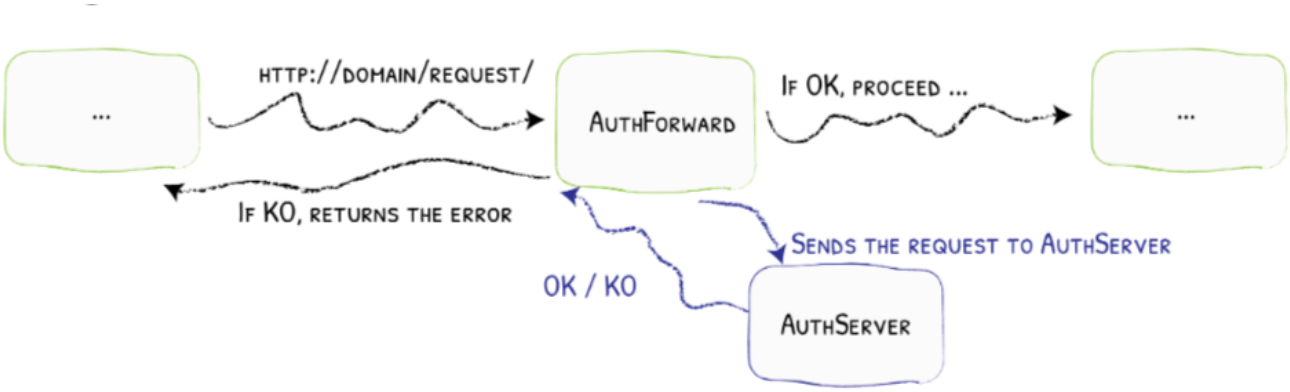
For our purpose, we require the Auth MS to be able to handle only requests of the general form ”Is User X allowed to access /BackendMS/example?”.

If the user’s identity is correctly verified though the GitLab OAuth2 provider AND this user is allowed to access the requested microservice/action, then the Auth MS should respond with a 200 (OK) code and let the request pass through the gateway to the required microservice/server.

If the user’s identity verification through GitLab OAuth2 fails OR this user is not permitted to access the request resource, then the Auth MS should respond with a 40X (NOT OK) code, and restrict the request from going forward.

Forward Auth Middleware in Traefik

Traefik allows middlewares to be set for the routes configured into it. These middlewares intercept the route path requests, and perform analysis/modifications before sending the requests ahead to the services. Traefik has a ForwardAuth middleware that delegates authentication to an external service. If the external authentication server responds to the middleware with a 2XX response codes, the middleware acts as a proxy, letting the request pass through to the desired service. However, if the external server responds with any other response code, the request is dropped, and the response code returned by the external auth server is returned to the user

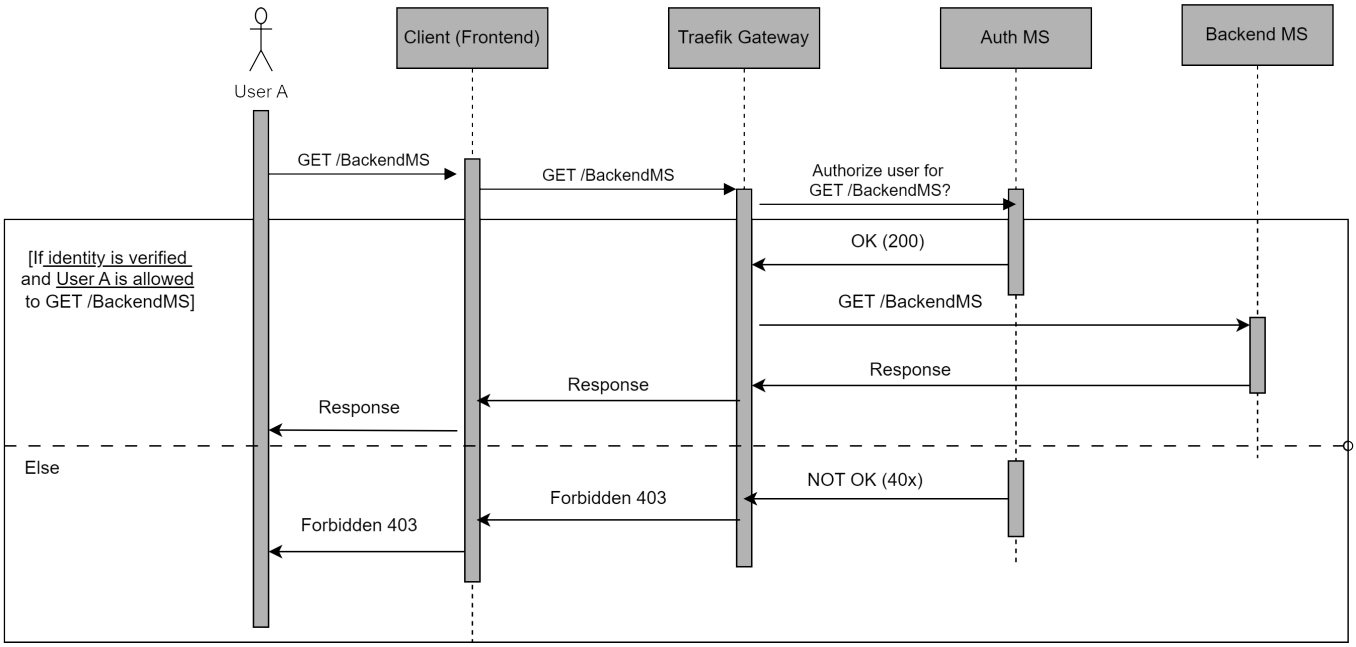


(source: Treafik documentation)

Thus, an Auth Microservice can be integrated into the existing gateway and DTaaS system structure easily by adding it as the external authentication server for ForwardAuth middlewares. These middlewares can be added on whichever routes/requests require authentication. For our specific purpose, this will be added to all routes since we impose atleast identity verification of users for any request through the gateway

Auth MS Design

The integrated Auth MS should thus work as described in the sequence diagram.



- Any request made by the user is made on the React website, i.e. the frontend of the DTaaS software.
- This request then goes through the Traefik gateway. Here it should be interrupted by the respective ForwardAuth middleware.
- The middleware asks the Auth MS if this request for the given user should be allowed.
- The Auth MS, i.e. the Auth server verifies the identity of the user using OAuth2 with GitLab, and checks if this user should be allowed to make this request.
- If the user is verified and allowed to make the request, the Auth server responds with a 200 OK to Traefik Gateway (more specifically to the middleware in Traefik)
- Traefik then forwards this request to the respective service. A response by the service, if any, will be passed through the chain back to the user.
- However, If the user is not verified or not allowed to make this request, the Auth server responds with a 40x to Traefik gateway.
- Traefik will then drop the request and respond to the Client informing that the request was forbidden. It will also pass the Auth servers response code

5.5.3 Auth Microservice

This document details the workflow and implementation of the DTaaS Auth Microservice. Please go through the System Design and the summary of the OAuth2.0 technology to be able to understand the content here better.

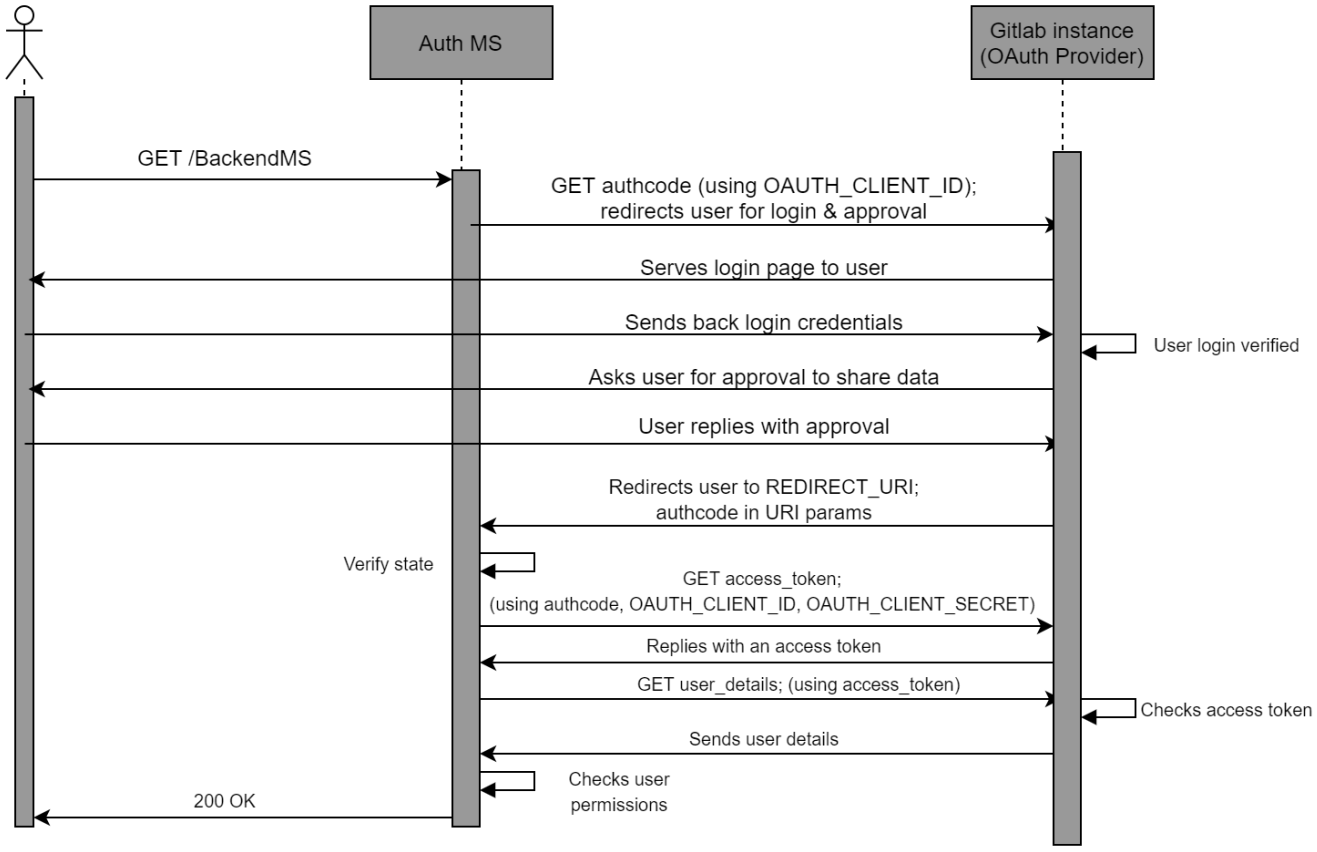
Workflow

USER IDENTITY USING OAUTH2.0

We define some constants that will help with the following discussion:

- CLIENT ID: The OAuth2 Client ID of the Auth MS
- CLIENT SECRET: The OAuth2 Client Secret of Auth MS
- REDIRECT URI: The URI where the user is redirected to after the user has approved sharing of information with the client.
- STATE: A random string used as an identifier for the specific "GET authcode" request (Figure 3.3)
- AUTHCODE: The one-use-only Authorization code returned by the OAuth2 provider (GitLab instance) in response to "GET authcode" after user approval.

Additionally, let's say DTaaS uses a dedicated gitlab instance hosted at the URL https://gitlab.foo.com (instead of https://foo.com)



A successful OAuth2 workflow (Figure 3.3) has the following steps:

- The user requests a resource, say *GET/BackendMS*
- The Auth MS intercepts this request, and starts the OAuth2 process.
- The Auth MS sends a authorization request to the Gitlab instance.

This is written in shorthand as *GET/authcode*. The actual request (a user redirect) looks like:

```
https ://gitlab.foo.com/oauth/
authorize?
response_type=code&
client_id=OAUTH_CLIENT_ID&
redirect_uri=REDIRECT_URI&
scope=read_user&state = STATE
```

Here the gitlab.foo.com/oauth/authorize is the specific endpoint of the Gitlab instance that handles authorisation code requests.

The query parameters in the request include the expected response type, which is fixed as ”code”, meaning that we expect an Authorization code. Other query parameters are the client id, the redirect uri, the scope which is set to read user for our purpose, and the state (the random string to identify the specific request).

- The OAuth2 provider redirects the user to the login page. Here the user logs into their protected account with their username/email ID and password.
- The OAuth2 provider then asks the user to approve/deny sharing the requested information with the Auth MS. The user should approve this for successful authentication.
- After approval, the user is redirected by the GitLab instance to the REDIRECT URI. This URI has the following form:

```
REDIRECT_URI?code=AUTHCODE&state=STATE
```

The REDIRECT URI is as defined previously, during the OAuth2 Client initialisation, i.e. the same as the one provided in the ”GET authcode” request by the Auth MS. The query parameters are provided by the Gitlab instance. These include the AUTHCODE which is the authoriation code that the Auth MS had requested, and the STATE which is the same random string in the ”GET authcode” request.

- The Auth MS retrieves these query parameters. It verifies that the STATE is the same as the random string it provided during the "GET authcode" request. This confirms that the AUTHCODE it has received is in response to the specific request it had made.
- The Auth MS uses this one-use-only AUTHCODE to exchange it for a general access token. This access token wouldn’t be one-use-only, although it would expire after a specified duration of time. To perform this exchange, the Auth MS makes another request to the GitLab instance. This request is written in shorthand as *GET/access_token* in the sequence diagram. The true form of the request is:

```
POST https://gitlab.foo.com/oauth/token,
parameters = 'client_id=OAUTH_CLIENT_ID&
client_secret=OAUTH_CLIENT_SECRET&
code=AUTHCODE&
grant_type=authorization_code&
redirect_uri=REDIRECT_URI'
```

The request to get a token by exchanging an authorization code, is actually a POST request (for most OAuth2 providers). The https://gitlab.foo.com/oauth/token API endpoint handles the token exchange requests. The parameters sent with the POST request are the client ID, the client secret, the AUTHCODE and the redirect uri. The grant type parameter is always set to the string ”authorization code”, which conveys that we will be exchanging an authentication code for an access token.

- The Gitlab instance exchanges a valid AUTHCODE for an Access Token. This is sent as a response to the Auth MS. An example response is of the following form:

```
{
  "access_token": "d8aed28aa506f9dd350e54",
  "token_type": "bearer",
  "expires_in": 7200,
  "refresh_token":"825f3bffb2544b976633a1",
  "created_at": 1607635748
}
```

The access token field provides the string that can be used as an access token in the headers of requests tryng to access user information. The token type field is usually ”bearer”, the expires in field specifies the time in seconds for which the access token will be valid, and the

created at field is the Epoch timestamp at which the token was created. The refresh token field has a string that can be used to refresh the access token, increasing it's lifetime. However we do not make use of the refresh token field. If an access token expires, the Auth MS simply asks for a new one. TOKEN is the access token string returned in the response.

- The Auth MS has finally obtained an access token that it can use to retrieve the user's information. Note that if the Auth MS already had an existing valid access token for information about this user, the steps above wouldn't be necessary, and thus wouldn't be performed by the Auth MS. The steps till now in the sequence diagram are simply to get a valid access token for the user information.
- The Auth MS makes a final request to the Gitlab instance, shorthanded as *GET user_details* in the sequence diagram. The actual request is of the form:

```
GET https ://gitlab.foo.com/api/v4/user
"Authorization": Bearer <TOKEN>
```

Here, https://gitlab.foo.com/api/v4/user is the API endpoint that responds with user information. An authorization header is required on the request, with a valid access token. The required header is added here, and TOKEN is the access token that the Auth MS holds.

- The Gitlab instance verifies the access token, and if it is valid, responds with the required user information. This includes username, email ID, etc. An example response looks like:

```
{
  "id": 8,
  "username": "UserX",
  "name": "XX",
  "state": "active",
  "web_url": "http://gitlab.foo.com/UserX",
  "created_at":"2023-12-03 T10:47:21.970 Z",
  "bio": "",
  "location": "",
  "public_email": null,
  "skype": "",
  "linkedin": "",
  "twitter": "",
  "organization": "",
  "job_title": "",
  "work_information": null,
  "followers": 0,
  "following": 0,
  "is_followed ": false,
  "local_time": null,
  "last_sign_in_at": "2023-12-13 T12:46:21.223 Z",
  "confirmed_at": "2023-12-03 T10:47:21.542 Z ",
  "last_activity_on": "2023-12-13",
  "email": "UserX@localhost",
  "projects_limit": 100000,
}
```

The important fields from this response are the "email", "username" keys. These keys are unique to a user, and thus provide an identity to the user.

- The Auth MS retrieves the values of candidate key fields like "email", "username" from the response. Thus, the Auth MS now knows the identity of the user.

CHECKING USER PERMISSIONS - AUTHORIZATION

An important feature of the Auth MS is to implement access policies for DTaaS resources. We may have requirements that certain resources and/or microservices in DTaaS should only be accessible to certain users. For example, we may want that /BackendMS/user1 should only be accessible to the user who has username user1. Another example may be that we may want /BackendMS/group3 to only be available to users who have an email ID in the domain @gmail.com. The Auth MS should be able to impose these restrictions and make certain services selectively available to certain users. There are two steps to doing this:

- Firstly, the user's identity should be known and trusted. The Auth MS should know the identity of a user and believe that the user is who they claim to be. This has been achieved in the previous section
- Secondly, this identity should be analysed against certain rules or against a database of allowed users, to determine whether this user should be allowed to access the requested resource.

The second step requires, for every service, either a set of rules that define which users should be allowed access to the service, or a database of user identities that are allowed to access the service. This database and/or set of rules should use the user identities, in our case the email ID or username, to decide whether the user should be allowed or not. This means that the rules should be built based on the

kind of username/ email ID the user has, say maybe using some RegEx. In the case of a database, the database should have the user identity as a key. For any service, we can simply look up if the key exists in the database or not and allow/deny the user access based on that.

In the sequence diagram, the Auth MS has a self-request marked as ”Checks user permissions” after receiving the user identity from the Gitlab instance. This is when the Auth MS compares the identity of the user to the rules and/or database it has for the requested service. Based on this, if the given identity has access to the requested resource, the Auth MS responds with a 200 OK. This finally marks a succcessful authentication, and the user can now access the requested resource. Note: Again, the Auth MS and user do not communicate directly. All requests/responses of the Auth MS are with the Traefik gateway, not the User directly. Infact, the Auth MS is the external server used by the ForwardAuth middleware of the specific route, and communicates with this middleware. If the authentication is successful, The gateway forwards the request to the specific resource when the 200 OK is recieved, else it drops the request and returns the error code to the user.

Implementation

TRAEFIK-FORWARD-AUTH

The implementation approach is setting up and configuring the open source thomseddon/traefik-forward-auth for our specific use case. This would work as our Auth microservice.

The traefik-forward-auth software is available as a docker.io image. This works as a docker container. Thus there are no dependency management issues. Additionally, it can be added as a middleware server to traefik routers. Thus, it needs atleast Traefik to work along with it properly. It also needs active services that it will be controlling access to. Traefik, the traefikforward-auth service and any services are thus, treated as a stack of docker containers. The main setup needed for this system is configuring the compose.yml file.

There are three main steps of configuring the Auth MS properly.

- The traefik-forward-auth service needs to be configured carefully. Firstly, we set the environment variables for our specific case. Since, we are using Gitlab, we use the generic-oauth provider configuration. Some important variables that are required are the OAuth2 Client ID, Client Secret, Scope. The API endpoints for getting an AUTHCODE, exchanging the code for an access token and getting user information are also necessary

Additionally, it is necessary to create a router that handles the REDIRECT URI path. This router should have a middleware which is set to traefik-forward-auth itself. This is so that after approval, when the user is taken to REDIRECT URI, this can be handled by the gateway and passed to the Auth service for token exchange. We add the ForwardAuth middleware here, which is a necessary part of our design as discussed before. We also add a load balancer for the service. We also need to add a conf file as a volume, for selective authorization rules (discussed later). This is according to the suggested configuration. Thus, we add the following to our docker services:

```
traefik-forward-auth:
image: thomseddon/traefik-forward-auth:latest
volumes:
  - <filepath>/conf:/conf
environment:
  - DEFAULT_PROVIDER = generic - oauth
  - PROVIDERS_GENERIC_OAUTH_AUTH_URL=https://gitlab.foo.com/oauth/authorize
  - PROVIDERS_GENERIC_OAUTH_TOKEN_URL=https://gitlab.foo.com/oauth/token
  - PROVIDERS_GENERIC_OAUTH_USER_URL=https://gitlab.foo.com/api/v4/user
  - PROVIDERS_GENERIC_OAUTH_CLIENT_ID=OAUTH_CLIENT_ID
  - PROVIDERS_GENERIC_OAUTH_CLIENT_SECRET=OAUTH_CLIENT_SECRET
  - PROVIDERS_GENERIC_OAUTH_SCOPE = read_user
  - SECRET = a - random - string
  # INSECURE_COOKIE is required if
  # not using a https entrypoint
  - INSECURE_COOKIE = true
labels:
  - "traefik.enable=true"
  - "traefik.http.routers.redirect.entryPoints=web"
  - "traefik.http.routers.redirect.rule=PathPrefix(/_oauth)"
  - "traefik.http.routers.redirect.middlewares=traefik-forward-auth"
  - "traefik.http.middlewares.traefik-forward-auth.forwardauth.address=http://traefik-forward-auth:4181"
  - "traefik.http.middlewares.traefik-forward-auth.forwardauth.authResponseHeaders=X-Forwarded-User"
  - "traefik.http.services.traefik-forward-auth.loadbalancer.server.port=4181"
```

- The traefik-forward-auth service should be added to the backend services as a middleware.

To do this, the docker-compose configurations of the services need to be updated by adding the following lines:

```
      - "traefik.http.routers.<service-router>.rule=Path(/<path>)"
      - "traefik.http.routers.<service-router>.middlewares=traefik-forward-auth"
```

This creates a router that maps to the required route, and adds the auth middleware to the required route.

- Finally, we need to set user permissions on user identities by creating rules in the conf file. Each rule has a name (an identifier for the rule), and an associated route for which the rule will be invoked. The rule also has an action property, which can be either ”auth” or ”allow”. If action is set to ”allow”, any requests on this route are allowed to bypass even the OAuth2 identification. If the action is set to ”auth”, requests on this route will require User identity OAuth2 and the system will follow the sequence diagram. For rules with action=”auth”, the user information is retrieved. The identity we use for a user is the user’s email ID. For ”auth” rules, we can configure two types of User restrictions/permissions on this identity:
- Whitelist - This would be a list of user identities (email IDs in our case) that are allowed to access the corresponding route.
- Domain - This would be a domain (example: gmail.com), and only email IDs (user identities) of that domain (example: johndoe@gmail.com) would be allowed access to the corresponding route.

Configuring any of these two properties of an ”auth” rule allows us to selectively permit access to certain users for certain resources. Not configuring any of these properties for an ”auth” rule means that the OAuth2 process is carried out and the user identity is retrieved, but all known user identities (i.e. all users that successfully complete the OAuth) are allowed to access the resource.

DTaaS currently uses only the whitelist type of rules.

These rules can be used in 3 different ways described below. The exact format of lines to be added to the conf file are also shown.

- No Auth - Serves the Path(‘/public‘) route. A rule with action=”allow” should be imposed on this.

```
rule.noauth.action=allow
rule.noauth.rule=Path(`/public`)
```

- User specific: Serves the Path(‘/user1‘) route. A rule that only allows ”user1@localhost” identity should be imposed on this

```
rule.onlyu1.action=auth
rule.onlyu1.rule=Path(`/user1`)
rule.onlyu1.whitelist=user1@localhost
```

- Common Auth - Serves the Path(‘/common‘) route. A rule that requires OAuth, i.e. with action=”allow”, but allows all valid and known user identities should be imposed on this.

```
rule.all.action = auth
rule.all.rule = Path(`/common`)
```

5.6 Testing

5.6.1 ? Common Questions on Testing

What is Software Testing

Software testing is a procedure to investigate the quality of a software product in different scenarios. It can also be stated as the process of verifying and validating that a software program or application works as expected and meets the business and technical requirements that guided design and development.

Why Software Testing

Software testing is required to point out the defects and errors that were made during different development phases. Software testing also ensures that the product under test works as expected in all different cases – stronger the test suite, stronger is our confidence in the product that we have built. One important benefit of software testing is that it facilitates the developers to make incremental changes to source code and make sure that the current changes are not breaking the functionality of the previously existing code.

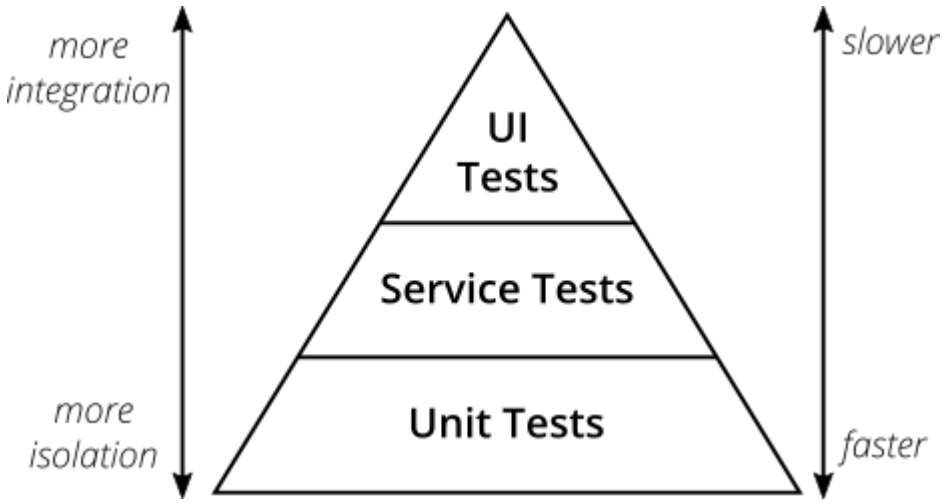
What is TDD

TDD stands for **Test Driven Development**. It is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. The goal of TDD can be viewed as specification and not validation. In other words, it's one way to think through your requirements or design before your write your functional code.

What is BDD

BDD stands for "Behaviour Driven Development". It is a software development process that emerged from TDD. It includes the practice of writing tests first, but focuses on tests which describe behavior, rather than tests which test a unit of implementation. This provides software development and management teams with shared tools and a shared process to collaborate on software development. BDD is largely facilitated through the use of a simple domain-specific language (DSL) using natural language constructs (e.g., English-like sentences) that can express the behavior and the expected outcomes. Mocha and Cucumber testing libraries are built around the concepts of BDD.

5.6.2 Testing workflow



(Ref: Ham Vocke, The Practical Test Pyramid)

We follow a testing workflow in accordance with the test pyramid diagram given above, starting with isolated tests and moving towards complete integration for any new feature changes. The different types of tests (in the order that they should be performed) are explained below:

Unit Tests

Unit testing is a level of software testing where individual units/ components of a software are tested. The objective of Unit Testing is to isolate a section of code and verify its correctness.

Ideally, each test case is independent from the others. Substitutes such as method stubs, mock objects, and spies can be used to assist testing a module in isolation.

BENEFITS OF UNIT TESTING

- Unit testing increases confidence in changing/ maintaining code. If good unit tests are written and if they are run every time any code is changed, we will be able to promptly catch any defects introduced due to the change.
- If codes are already made less interdependent to make unit testing possible, the unintended impact of changes to any code is less.
- The cost, in terms of time, effort and money, of fixing a defect detected during unit testing is lesser in comparison to that of defects detected at higher levels.

UNIT TESTS IN DTAAS

Each component DTaaS project uses unique technology stack. Thus the packages used for unit tests are different. Please check the `test/` directory of a component to figure out the unit test packages used.

Integration tests

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. In DTaaS, we use an integration server for software development as well as such tests.

The existing integration tests are done at the component level. There are no integration tests between the components. This task has been postponed to future.

End-to-End tests

Testing any code changes through the end user interface of your software is essential to verify if your code has the desired effect for the user. End-to-End tests in DTaaS a functional setup.

There are end-to-end tests in the DTaaS. This task has been postponed to future.

Feature Tests

A Software feature can be defined as the changes made in the system to add new functionality or modify the existing functionality. Each feature is said to have a characteristics that is designed to be useful, intuitive and effective. It is important to test a new feature when it has been added. We also need to make sure that it does not break the functionality of already existing features. Hence feature tests prove to be useful.

The DTaaS project does not have any feature tests yet. Cucumber shall be used in future to implement feature tests.

5.6.3 References

1. Arthur Hicken, Shift left approach to software testing
2. Justin Searls and Kevin Buchanan, Contributing Tests wiki.
3. This wiki has good explanation of TDD and test doubles.

5.7 Docker workflow for DTaaS

This readme will explain the building and use of different docker files for use in development and installation of the DTaaS software.

NOTE: A local docker installation with compose plugin is a pre-requisite for using docker workflows.

5.7.1 Folder Structure

There are two dockerfiles for building the containers:

- **client.dockerfile**: Dockerfile for building the client application container.
- **libms.dockerfile**: Dockerfile for building the library microservice container from source code.
- **libms.npm.dockerfile**: Dockerfile for building the library microservice container from published npm package at npmjs.com. This Dockerfile is only used during publishing. It is used neither in the development builds nor in Github actions.

There is a specific compose file for development:

The **compose.dev.yml:** file is the docker compose file for development environment.

5.7.2 Build and Publish Docker Images

The github workflows publish docker images of client website and libms to github and docker hub.

Developers

Use of docker images is handy for developers as well. It is suggested that developers build the required images locally on their computer and use them for development purposes. The images can be built using

```
docker compose -f compose.dev.yml build
```

5.7.3 Running Docker Containers

Follow these steps to use the application with docker.

The DTaaS application requires multiple configuration files. The list of configuration files to be modified are given for each scenario.

Development Environment

This scenario is for software developers.

The configuration files to be updated are:

1. **docker/.env** : please see docker installation docs for help with updating this config file
2. **docker/conf.dev** : please see docker installation docs for help with updating this config file
3. **client/config/local.js** please see client config for help with updating this config file
4. **servers/lib/config/libms.dev.yaml** please see lib config for help with updating this config file

The docker commands need to be executed from this directory (`docker`). The relevant docker commands are:

```
docker compose -f compose.dev.yml up -d #start the application
docker compose -f compose.dev.yml down  #terminate the application
```

Access the Application

You should access the application through the PORT mapped to the Traefik container. e.g. `localhost`

5.8 Publish NPM packages

The DTaaS software is developed as a monorepo with multiple npm packages.

5.8.1 Default npm registry

The default registry for npm packages is npmjs. The freely-accessible public packages are published to the **npmjs** registry. The publication step is manual for the runner.

```
npm login --registry="https://registry.npmjs.org"
cat ~/.npmrc  #shows the auth token for the registry
//registry.npmjs.org/:_authToken=xxxxxxxxxx
yarn publish --registry="https://registry.npmjs.org" \
  --no-git-tag-version --access public
```

At least one version of runner package is published to this registry for each release of DTaaS.

The publication step for library microservice is automated via github actions.

5.8.2 Github npm registry

The Github actions of the project publish packages. The only limitation is that the users need an access token from Github.

5.8.3 Private Registry

Setup private npm registry

Since publishing to npmjs is irrevocable and public, developers are encouraged to setup their own private npm registry for local development. A private npm registry will help with local publish and unpublish steps.

We recommend using verdaccio for this task. The following commands help you create a working private npm registry for development.

```
docker run -d --name verdaccio -p 4873:4873 verdaccio/verdaccio
npm adduser --registry http://localhost:4873 #create a user on the verdaccio registry
npm set registry http://localhost:4873/
yarn config set registry "http://localhost:4873"
yarn login --registry "http://localhost:4873" #login with the credentials for yarn utility
npm login #login with the credentials for npm utility
```

You can open `http://localhost:4873` in your browser, login with the user credentials to see the packages published.

PUBLISH TO PRIVATE NPM REGISTRY

To publish a package to your local registry, do:

```
yarn install
yarn build #the dist/ directory is needed for publishing step
yarn publish --no-git-tag-version #increments version in package.json, publishes to registry
yarn publish #increments version in package.json, publishes to registry and adds a git tag
```

The package version in package.json gets updated as well. You can open `http://localhost:4873` in your browser, login with the user credentials to see the packages published. Please see verdaccio docs for more information.

If there is a need to unpublish a package, ex: `@dtaas/runner@0.0.2` , do:

```
npm unpublish  --registry http://localhost:4873/ @dtaas/runner@0.0.2
```

To install / uninstall this utility for all users, do:

```
sudo npm install  --registry http://localhost:4873 -g @dtaas/runner
sudo npm list -g # should list @dtaas/runner in the packages
sudo npm remove --global @dtaas/runner
```

🚀 Use the packages

The packages available in private npm registry can be used like the regular npm packages installed from npmjs.

For example, to use `@dtaas/runner@0.0.2` package, do:

```
sudo npm install  --registry http://localhost:4873 -g @dtaas/runner
runner # launch the digital twin runner
```

5.9 DevOps Framework

5.9.1 Overview

Expectations From a DevOps Framework

The functional requirements of the system include the automation of pipelines and the management of Digital Twins (DTs) via Application Programming Interface (API). Consequently, the framework was designed to facilitate the comprehensive automation of the DT lifecycle, minimizing the necessity for manual intervention. The system must be capable of managing the dynamic configuration of pipelines, utilizing variables that permit the customization of pipeline behavior according to the data provided by the user, such as the designation of the DT.

Integration with GitLab is another fundamental requirement. The framework must be able to interact with GitLab to execute CI/CD pipelines via API calls using Gitbeaker as a wrapper. Users must authenticate via GitLab's OAuth mechanism, and the system must automatically manage the authentication tokens and trigger tokens needed to start pipelines. Additionally, the system must automatically retrieve key information from the user's GitLab repository, such as the list of available DTs.

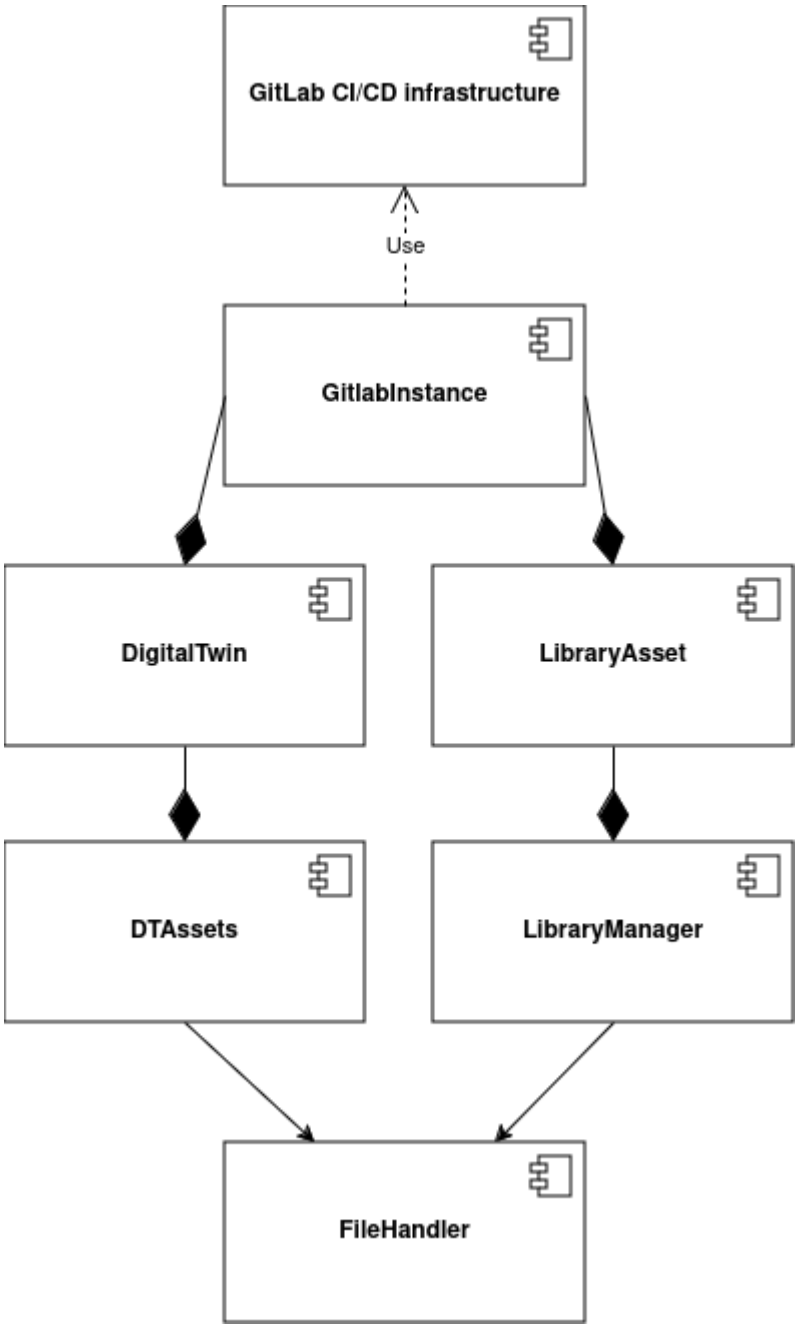
High Level Architecture

We use a DevOps framework to enable interaction with the DTs via APIs calls, so that users can start, monitor and manage their DTs via the web application.

The architectural design of the DevOps framework was intended to facilitate the management of DTs. It is based on two key elements:

- **The GitLab CI/CD infrastructure**, which employs a parent-child pipeline hierarchy. The objective of this infrastructure is to enable the triggering of a pipeline of a specific DT by simply passing the necessary data as parameters, such as the name of the DT and the tag of the runner that will execute the pipeline.
- **Classes implemented in the code**, which utilize Gitbeaker to realize the APIs required for interaction with DTs.

The component diagram below illustrates how the infrastructure consists of three main classes: `DigitalTwin`, `LibraryAsset`, and `GitlabInstance`.



The distinction between the `DigitalTwin` and `LibraryAsset` classes was necessary to separate the full management of a DT from an asset visualized through the library. The `LibraryAsset` class provides a significantly reduced set of functionality compared to the `DigitalTwin`, focusing only on asset visualization.

Intermediate classes were introduced to ensure a clear separation of file management responsibilities: `DTAssets` and `LibraryManager`. These classes implement the necessary logic to mediate between a `DigitalTwin` or `LibraryAsset` and the `FileHandler` class. The `FileHandler` class has a single responsibility: to make API calls to files via GitBeaker. This design allows for the separation of high-level logic from low-level file operations.

The infrastructure requires that the `DigitalTwin` class and the `LibraryAsset` class include an instance of `GitlabInstance`. This composition relationship emphasizes the dependency between these classes, where a `DigitalTwin` or a `LibraryAsset` instance cannot function independently without a `GitlabInstance`. The `GitlabInstance` class provides the essential services required for interacting with GitLab, including API integrations and pipeline management.

The `GitlabInstance` class serves as the interface to the realized CI/CD infrastructure. By utilizing the GitLab class imported from GitBeaker and initialized as its attribute, `GitlabInstance` facilitates the execution of pipelines and other CI/CD-related tasks. This architecture ensures that the infrastructure remains modular and adheres to the principles of single responsibility and clear dependency management.

Gitbeaker

GitBeaker is a client library for Node.js that enables users to interact with the GitLab API. In particular, `gitbeaker/rest` is a specific version of the Gitbeaker package that allows users to submit requests to GitLab's REST API.

One of the most significant features of Gitbeaker is the provision of support for a range of authentication methods, including the use of personal tokens and OAuth keys. Gitbeaker provides a range of predefined methods for requesting data from the various GitLab APIs, eliminating the need for users to manually construct HTTP requests, thus greatly simplifying the integration process with GitLab.

It automatically handles errors in HTTP requests (and provides meaningful error messages that help diagnose and resolve problems) and is fully compatible with all of GitLab's REST APIs.

Ref: Vanessa Scherma, Design and implementation of an integrated DevOps framework for Digital Twins as a Service software platform, Master's Degree Thesis, Politecnico Di Torino, 2024.

5.9.2 GitLab CI/CD Infrastructure

Given that files in the Library are stored in a Git repository, the approach employed was that of GitLab's parent-child pipelines. In this context, a parent pipeline initiates the execution of another pipeline within the same project, the latter of which is known as the child pipeline. More about pipelines can be found in GitLab's documentation on CI/CD Pipelines.

CI/CD Pipelines

Continuous Integration (CI) and Continuous Deployment (CD) represent two key components of the DevOps methodology.

CI involves frequent integration of code changes into a common repository. Each integration triggers automated builds and tests that permit the detection of issues at an early stage. This practice ensures that the changes made to the code are checked fast enough, reducing the possibilities of integration problems and hence ensuring high-quality software.

CD automates the process of release, ensuring that code changes are automatically tested and prepared for deployment. Teams using CD can deploy updates rapidly and reliably, improving the responsiveness and quality of software. Performed together, CI/CD automates the whole delivery pipeline for software, increasing efficiency and reducing errors. They entirely eliminate, or significantly reduce, the manual human input required for a code change to be moved from a commit to a production environment. The entire process of compilation, testing (including unit, integration, and regression testing), deployment, and infrastructure provisioning is included.

CI/CD practices are explained in more detail in this article by GitLab.

A CI/CD pipeline is a series of automated processes that manage CI and CD of software. They are configured to run automatically, with no need for manual intervention once activated.

GitLab is a single application for the entire DevOps lifecycle, which means it performs all of the basics required for CI/CD in one environment. The documentation provided by GitLab was instrumental in enabling a comprehensive understanding of the CI/CD pipelines.

Pipelines are composed of a number of essential components. *Jobs* delineate the specific tasks to be accomplished, while *stages* define the sequence in which jobs are executed. In this way, stages ensure that each step takes place in the right order and make the pipeline more efficient and consistent. In the event that all jobs within a stage are successfully completed, the pipeline will automatically proceed to the subsequent stage. However, if any of the jobs fail, the flow is interrupted without proceeding.

When a pipeline is initiated, the jobs that have been defined within it are then distributed among the available runners.

GitLab runners are agents within the GitLab Runner application that execute the jobs in accordance with their configuration and the available resources. They can be configured to operate on a variety of platforms, including virtual machines, containers, and physical servers. They can also be managed locally or in a cloud environment.

We use this GitLab parent-child pipeline setup to trigger execution of digital twins stored in a user's GitLab repository.

> **Note**
>
> The recommended practice is to modified these pipelines via the Digital Twins Preview Page.

Parent Pipeline

The parent pipeline was configured as a top-level element. There is a single stage called `triggers`, which is responsible for triggering other child pipelines.

In the `.gitlab-ci.yml` file, triggers are managed for DTs inside the user repository. Each trigger is connected with one distinct DT and becomes active when the corresponding value of the `DTName` variable is provided by the API call. The `RunnerTag` variable is used to specify a custom runner tag that will execute each job in the DT's pipeline.

Below is an explanation of the keywords used in the CI/CD pipeline configuration:

- **Image**: Specifies the Docker image, such as `fedora:41`, providing the environment for the pipeline execution.
- **Stages**: Defines phases in the pipeline, such as triggers, organizing tasks sequentially.
- **Trigger**: Initiates another pipeline or job, incorporating configurations from an external file.
- **Include**: Imports configurations from another file for modular pipeline setups.
- **Rules**: Sets conditions for job execution based on variables or states.
- **If**: A condition within rules specifying when a job should run based on the value of a variable.
- **When**: Specifies the timing of job execution, such as `always`.
- **Variables**: Defines dynamic variables, like `RunnerTag`, used in the pipeline.

Here is an example of such a YAML file that registers a trigger for a DT named `mass-spring-damper`:

```yaml
image: fedora:41

stages:
  - triggers

trigger_mass-spring-damper:
  stage: triggers
  trigger:
    include: digital_twins/mass-spring-damper/.gitlab-ci.yml
  rules:
    - if: '$DTName == "mass-spring-damper"'
      when: always
  variables:
    RunnerTag: $RunnerTag
```

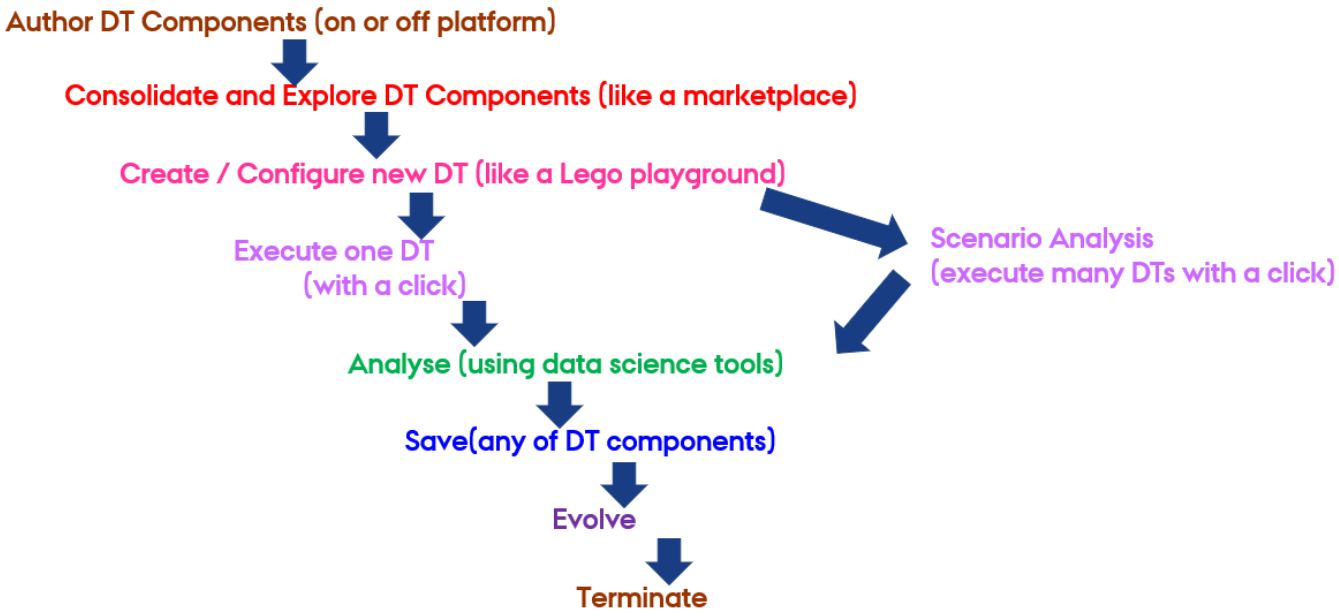
Digital Twin Structure

The `digital_twins` folder contains DTs that have been pre-built by one or more users. The intention is that they should be sufficiently flexible to be reconfigured as required for specific use cases.

Let us look at an example of such a configuration. The dtaas/user1 repository on gitlab.com contains the `digital_twins` directory with a `hello_world` example. Its file structure looks like this:

```
hello_world/
├── lifecycle/
│   ├── clean
│   ├── create
│   ├── execute
│   └── terminate
├── .gitlab-ci.yml
└── description.md
```

The `lifecycle` directory here contains four files - `clean`, `create`, `execute` and `terminate`, which are simple BASH scripts. These correspond to stages in a digital twin's lifecycle.



Child Pipelines

To automate the lifecycle of DT, a child pipeline has been incorporated into its corresponding folder. Regardless of the image provided in the parent pipeline, each child pipeline will use its own specified image specified in its YAML configuration or Ruby's default image.

The following are the explanations of the keywords used within the CI/CD child pipeline based on GitLab's CI/CD YAML syntax reference:

1. `Stage` It defines the steps that happen in a pipeline sequentially, for example, create, execute and clean, to make sure that tasks occur in a specific order.
2. `Script` It lists commands to be run at each step; for example, changing directories, modifying permissions, or running lifecycle scripts.
3. `Tags` It specifies which runner should run the jobs, thereby providing an additional control over where and how the jobs are run.

With the DT `mass-spring-damper` serving as a point of reference, the stages in question are designed to facilitate the creation, execution, and termination of the DT simulation, as well as the cleaning and restoration of the environment to ensure its readiness for future executions. Here is an example of a configuration that defines `create`, `execute` and `clean` as part of the child pipeline:

```
image: ubuntu:20.04

stages:
    - create
    - execute
    - clean

create_mass-spring-damper:
    stage: create
    script:
        - cd digital_twins/mass-spring-damper
        - chmod +x lifecycle/create
        - lifecycle/create
    tags:
        - $RunnerTag

execute_mass-spring-damper:
    stage: execute
    script:
        - cd digital_twins/mass-spring-damper
        - chmod +x lifecycle/execute
        - lifecycle/execute
    tags:
        - $RunnerTag

clean_mass-spring-damper:
    stage: clean
    script:
        - cd digital_twins/mass-spring-damper
        - chmod +x lifecycle/terminate
        - chmod +x lifecycle/clean
        - lifecycle/terminate
    tags:
        - $RunnerTag
```

Ref: Vanessa Scherma, Design and implementation of an integrated DevOps framework for Digital Twins as a Service software platform, Master's Degree Thesis, Politecnico Di Torino, 2024.

5.9.3 API Calls

A GitLab DevOps pipeline can be triggered via an API call using a pipeline trigger token which is created on the GitLab instance, with the following values:

1. `<trigger_token>` : The user GitLab trigger token.
2. `<digital_twin_name>` : The name of the DT (e.g. mass-spring-damper).
3. `<runner_tag>` : The specific tag of the GitLab runner that the user wants to use.
4. `<project_id>` : The ID of the GitLab project, displayed in the project overview page.

The example given below sets the `DTName` variable to the desired DT name, the `RunnerTag` variable to the specified GitLab Runner tag, and ensures the call will be executed in the `main` branch:

```
curl --request POST \
  --form "token=<trigger_token>" \
  --form ref=main \
  --form "variables[DTName]=<digital_twin_name>" \
  --form "variables[RunnerTag]=<runner_tag>" \
  "https://maestro.cps.digit.au.dk/gitlab/api/v4/projects/<project_id>/trigger/pipeline"
```

Ref: Vanessa Scherma, Design and implementation of an integrated DevOps framework for Digital Twins as a Service software platform, Master's Degree Thesis, Politecnico Di Torino, 2024.

5.9.4 Implemented Classes

In order to facilitate the management of the lifecycle of DTs via the web application interfaces, it was necessary to develop specific code within the project client. The code was designed to facilitate efficient API calls through the use of Gitbeaker as a wrapper, as this approach simplifies interactions with GitLab's REST API and reduces the complexity of the project code.

The APIs have been integrated into the front-end by wiring up API endpoints to front-end components, ensuring a seamless data flow. Unit and integration testing was done to ensure the coverage of all functional requirements and solve all problems regarding data consistency, performance, or user experience.

Given below is our implementation of these classes in TypeScript:

```
class GitlabInstance {
  async init();
  async getProjectId();
  async getTriggerToken(projectId: number);
  async getDTSubfolders(projectId: number);
  async getLibrarySubfolders(
    projectId: number,
    type: string,
    isPrivate: boolean,
  );
  executionLogs();
  async getPipelineJobs(
    projectId: number,
    pipelineId: number,
  );
  async getJobTrace(projectId: number, jobId: number);
  async getPipelineStatus(
    projectId: number,
    pipelineId: number,
  );
}

class DigitalTwin {
  async getDescription();
  async getFullDescription();
  private async triggerPipeline();
  async execute();
  async stop(projectId: number, pipeline: string);
  async create(
    files: FileState[],
    cartAssets: LibraryAsset[],
    libraryFiles: LibraryConfigFile[],
  );
  async delete();
  async getDescriptionFiles();
  async getConfigFiles();
  async getLifecycleFiles();
  async prepareAllAssetFiles(
    cartAssets: LibraryAsset[],
    libraryFiles: LibraryConfigFile[],
  );
  async getAssetFiles();
}

class LibraryAsset {
  async getDescription();
  async getFullDescription();
  async getConfigFiles();
}
```

GitlabInstance

The `GitlabInstance` class was created in order to manage the APIs and information related to the GitLab profile, the project, and the user-specific data stored in their account.

The username and the token required to instantiate the Gitbeaker *Gitlab* component, which is required for making the API calls, are retrieved from the session storage, taking the *access_token* of the user already logged into the DTaaS application.

The initialisation of the `GitlabInstance` object is concluded with the execution of the `init()` method, which enables the retrieval and storage of the `projectId` and `triggerToken` attributes. The *projectId* is a unique identifier for projects in GitLab and it is essential for subsequent API calls. For example, it is passed to the method that retrieves a *trigger token*, which is used to trigger CI/CD pipelines in GitLab.

The objective of the `getDTSubfolders` method was to retrieve the names and corresponding descriptions of the DTs of the user, so that these could be shown at the front-end interface. This approach would obviate the user from having to input the name of a DT; hence, saving the

user from possible error and inefficiencies arising from manual input. The user interface makes it easier for the user to deal with DTs by automatizing their selection and manages them more accurately. This implementation also eliminates the necessity for manual input from users for the access token and the username, which are automatically provided via the GitLab OAuth login.

Furthermore, logs maintained in the `GitlabInstance` class improve awareness and transparency over the operations conducted. The final three methods are employed in conjunction to oversee the execution of a DT. In particular, individual logs are saved for each job in the pipeline, and the status of the latter is monitored so that, once the entire pipeline is complete, the results can be displayed in detail within the user interface. In this way, all statuses of each operation are logged for better debugging and performance analysis, including possible errors. Having trace logs exposed to the user means troubleshooting will be more effective and insight into execution and management of DTs will be gained, improving system reliability and user confidence.

DigitalTwin

The DigitalTwin class was created in order to manage the APIs and information related to a specific DT.

The creation of a DigitalTwin object requires a pre-existing GitlabInstance to be associated with the object. It was determined that matching a different GitlabInstance for each DigitalTwin would be the optimal approach to ensure the maintenance of independence between the various DTs. The *api* attribute of GitlabInstance facilitates the execution of Gitbeaker APIs pertinent to the DT.

The class allows a pipeline to be started and stopped, thus giving the user full control of the execution. The execute() method uses the previous methods internally. This approach ensures that there are no errors due to missing design information during the execution of the pipeline. Responsibilities have been divided into smaller methods in order to make the code more modular, facilitating debugging and testing. In both execute() and stop(), the status of operations executed on the DT is monitored, keeping track of them via the logs attribute of GitlabInstance. Errors are identified and tracked, providing a complete view and the ability to monitor performance.

The `descriptionFiles`, `lifecycleFiles` and `configFiles` attributes are used to keep track of the files within the corresponding GitLab folder of the DT, thus enabling the read and modify features.
The create() method enables the creation of a DT and saves all its files in the user's corresponding GitLab folder. Additionally, if the DT is configured as *common*, it is also added to GitLab's shared folder, making it part of the Library and accessible to other users.

Similarly, the delete() method removes a DT from GitLab. If the DT was part of the Library, it is also removed from the shared folder.

A crucial aspect of these two methods is their integration with the DevOps infrastructure. When a DT is created or deleted, the `.gitlab-ci.yml` file of the parent pipeline is updated to add or remove the *trigger_DTName* section associated with the DT. This ensures that a user-created DT can be executed via the web interface without requiring manual updates to pipeline configuration files on GitLab. Instead, these files are automatically updated, providing an effortless user experience and maintaining alignment with the infrastructure.

LibraryAsset

The LibraryAsset class was created in order to manage the APIs and information related to a specific library asset.

It is similar to the DigitalTwin class, but contains only the methods required to display files. This focused design reflects its limited scope and ensures simplicity and clarity for use cases involving the library.

Ref: Vanessa Scherma, Design and implementation of an integrated DevOps framework for Digital Twins as a Service software platform, Master's Degree Thesis, Politecnico Di Torino, 2024.

6. Few issues in the Software

If you find a bug, please open an issue

6.1 Third-Party Software

The explanation given below corresponds to the bugs you may face from third party software included in DTaaS. Known issues are listed below.

6.1.1 ML Workspace

- the docker container might down a bit after two weeks. The only known solution is to restart the docker container. You don't need to restart the complete DTaaS platform, restart of the docker container of ml-workspace is sufficient.
- the terminal tool doesn't seem to have the ability to refresh itself. If there is an issue, the only solution is to close and reopen the terminal from "open tools" drop down of notebook
- terminal app does not show at all after some time: terminal always comes if it is open from drop-down menu of Jupyter Notebook, but not as a direct link.

6.2 Gitlab

- The gilab oauth authorization service does not have a way to sign out of a third-party application. Even if you sign out of DTaaS, the gitlab still shows user as signed in. The next time you click on the sign in button on the DTaaS page, user is not shown the login page. Instead user is directly taken to the **Library** page. So close the brower window after you are done. Another way to overcome this limitation is to open your gitlab instance (`https://gitlab.foo.com`) and signout from there. Thus user needs to sign out of two places, namely DTaaS and gitlab, in order to completely exit the DTaaS application.

7. Contributors

code contributors

7.1 Users

Cláudio Ângelo Gonçalves Gomes, Dmitri Tcherniak, Elif Ecem Bas, Farshid Naseri, Giuseppe Abbiati, Hannes Iven, Hao Feng, Henrik Ejersbo, Lars Vosteen, Lei Zhao, Mirgita Frasheri, Morten Haahr Kristensen, Neena Goveas, Tanusree Roy, Santiago Gil Arboleda, Swarup Mahato, Valdemar Tang

7.2 Example Contributors

Example Name	Contributors
Mass Spring Damper	Prasad Talasila
Water Tank Fault Injection	Henrik Ejersbo and Mirgita Frasheri
Water Tank Model Swap	Henrik Ejersbo and Mirgita Frasheri
Desktop Robotti with RabbitMQ	Mirgita Frasheri
Water Treatment Plant and OPC-UA	Lucia Royo and Alejandro Labarias
Three Water Tanks with DT Manager Framework	Santiago Gil Arboleda
Flex-Cell with Two Industrial Robots	Santiago Gil Arboleda
Incubator	Morten Haahr Kristensen
Firefighters in Emergency Environments	Lars Vosteen and Hannes Iven
Mass Spring Damper with NuRV Runtime Monitor	Alberto Bonizzi
Incubator with NuRV Runtime Monitor	Alberto Bonizzi and Morten Haahr Kristensen
Incubator with NuRV Runtime Monitor Service	Valdemar Tang
Water Tank Fault Injection with NuRV Runtime Monitor	Alberto Bonizzi
Incubator Co-Simulation with NuRV Runtime Monitor FMU	Morten Haahr Kristensen
Incubator with NuRV Runtime Monitor FMU as Service	Valdemar Tang and Morten Haahr Kristensen
Incubator with NuRV Runtime Monitor as Service	Morten Haahr Kristensen and Valdemar Tang

7.3 Documentation

1. Talasila, P., Gomes, C., Mikkelsen, P. H., Arboleda, S. G., Kamburjan, E., & Larsen, P. G. (2023). Digital Twin as a Service (DTaaS): A Platform for Digital Twin Developers and Users arXiv preprint arXiv:2305.07244.
2. Astitva Sehgal for developer and example documentation.
3. Tanusree Roy and Farshid Naseri for asking interesting questions that ended up in FAQs.

8. License

8.1 License

--- Start of Definition of INTO-CPS Association Public License ---

/*

- This file is part of the INTO-CPS Association.
- Copyright (c) 2017-CurrentYear, INTO-CPS Association (ICA),
- c/o Peter Gorm Larsen, Aarhus University, Department of Engineering,
- Finlandsgade 22, 8200 Aarhus N, Denmark.
- All rights reserved.
- THIS PROGRAM IS PROVIDED UNDER THE TERMS OF GPL VERSION 3 LICENSE OR
- THIS INTO-CPS ASSOCIATION PUBLIC LICENSE (ICAPL) VERSION 1.0.
- ANY USE, REPRODUCTION OR DISTRIBUTION OF THIS PROGRAM CONSTITUTES
- RECIPIENT'S ACCEPTANCE OF THE INTO-CPS ASSOCIATION PUBLIC LICENSE OR
- THE GPL VERSION 3, ACCORDING TO RECIPIENTS CHOICE.
- The INTO-CPS tool suite software and the INTO-CPS Association
- Public License (ICAPL) are obtained from the INTO-CPS Association, either
- from the above address, from the URLs: http://www.into-cps.org or
- in the INTO-CPS tool suite distribution.
- GNU version 3 is obtained from:
- http://www.gnu.org/copyleft/gpl.html.
- This program is distributed WITHOUT ANY WARRANTY; without
- even the implied warranty of MERCHANTABILITY or FITNESS
- FOR A PARTICULAR PURPOSE, EXCEPT AS EXPRESSLY SET FORTH
- IN THE BY RECIPIENT SELECTED SUBSIDIARY LICENSE CONDITIONS OF
- THE INTO-CPS ASSOCIATION PUBLIC LICENSE.
- See the full ICAPL conditions for more details.

*/

--- End of INTO-CPS Association Public License Header ---

The ICAPL is a public license for the INTO-CPS tool suite with three modes/alternatives (GPL, ICA-Internal-EPL, ICA-External-EPL) for use and redistribution, in source and/or binary/object-code form:

- GPL. Any party (member or non-member of the INTO-CPS Association) may use and redistribute INTO-CPS tool suite under GPL version 3.
- Silver Level members of the INTO-CPS Association may also use and redistribute the INTO-CPS tool suite under ICA-Internal-EPL conditions.
- Gold Level members of the INTO-CPS Association may also use and redistribute The INTO-CPS tool suite under ICA-Internal-EPL or ICA-External-EPL conditions.

Definitions of the INTO-CPS Association Public license modes:

- GPL = GPL version 3.
- ICA-Internal-EPL = These INTO-CPA Association Public license conditions together with Internally restricted EPL, i.e., EPL version 1.0 with the Additional Condition that use and redistribution by a member of the INTO-CPS Association is only allowed within the INTO-CPS Association member's own organization (i.e., its own legal entity), or for a member of the INTO-CPS Association paying a membership fee corresponding to the size of the organization including all its affiliates, use and redistribution is allowed within/between its affiliates.
- ICA-External-EPL = These INTO-CPA Association Public license conditions together with Externally restricted EPL, i.e., EPL version 1.0 with the Additional Condition that use and redistribution by a member of the INTO-CPS Association, or by a Licensed Third Party Distributor having a redistribution agreement with that member, to parties external to the INTO-CPS Association member’s own organization (i.e., its own legal entity) is only allowed in binary/object-code form, except the case of redistribution to other members the INTO-CPS Association to which source is also allowed to be distributed.

[This has the consequence that an external party who wishes to use the INTO-CPS Association in source form together with its own proprietary software in all cases must be a member of the INTO-CPS Association].

In all cases of usage and redistribution by recipients, the following conditions also apply:

a) Redistributions of source code must retain the above copyright notice, all definitions, and conditions. It is sufficient if the ICAPL Header is present in each source file, if the full ICAPL is available in a prominent and easily located place in the redistribution.

b) Redistributions in binary/object-code form must reproduce the above copyright notice, all definitions, and conditions. It is sufficient if the ICAPL Header and the location in the redistribution of the full ICAPL are present in the documentation and/or other materials provided with the redistribution, if the full ICAPL is available in a prominent and easily located place in the redistribution.

c) A recipient must clearly indicate its chosen usage mode of ICAPL, in accompanying documentation and in a text file ICA-USAGE-MODE.txt, provided with the distribution.

d) Contributor(s) making a Contribution to the INTO-CPS Association thereby also makes a Transfer of Contribution Copyright. In return, upon the effective date of the transfer, ICA grants the Contributor(s) a Contribution License of the Contribution. ICA has the right to accept or refuse Contributions.

Definitions:

"Subsidiary license conditions" means:

The additional license conditions depending on the by the recipient chosen mode of ICAPL, defined by GPL version 3.0 for GPL, and by EPL for ICA-Internal-EPL and ICA-External-EPL.

"ICAPL" means:

INTO-CPS Association Public License version 1.0, i.e., the license defined here (the text between "--- Start of Definition of INTO-CPS Association Public License ---" and "--- End of Definition of INTO-CPS Association Public License ---", or later versions thereof.

"ICAPL Header" means:

INTO-CPS Association Public License Header version 1.2, i.e., the text between "--- Start of Definition of INTO-CPS Association Public License ---" and "--- End of INTO-CPS Association Public License Header ---, or later versions thereof.

"Contribution" means:

a) in the case of the initial Contributor, the initial code and documentation distributed under ICAPL, and

b) in the case of each subsequent Contributor: i) changes to the INTO-CPS tool suite, and ii) additions to the INTO-CPS tool suite;

where such changes and/or additions to the INTO-CPS tool suite originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the INTO-CPS tool suite by such Contributor itself or anyone acting on such Contributor's behalf.

For Contributors licensing the INTO-CPS tool suite under ICA-Internal-EPL or ICA-External-EPL conditions, the following conditions also hold:

Contributions do not include additions to the distributed Program which: (i) are separate modules of software distributed in conjunction with the INTO-CPS tool suite under their own license agreement, (ii) are separate modules which are not derivative works of the INTO-CPS tool suite, and (iii) are separate modules of software distributed in conjunction with the INTO-CPS tool suite under their own license agreement where these separate modules are merged with (weaved together with) modules of The INTO-CPS tool suite to form new modules that are distributed as object code or source code under their own license agreement, as allowed under the Additional Condition of internal distribution according to ICA-Internal-EPL and/or Additional Condition for external distribution according to ICA-External-EPL.

"Transfer of Contribution Copyright" means that the Contributors of a Contribution transfer the ownership and the copyright of the Contribution to the INTO-CPS Association, the INTO-CPS Association Copyright owner, for inclusion in the INTO-CPS tool suite. The transfer takes place upon the effective date when the Contribution is made available on the INTO-CPS Association web site under ICAPL, by such Contributors themselves or anyone acting on such Contributors' behalf. The transfer is free of charge. If the Contributors or the INTO-CPS Association so wish, an optional Copyright transfer agreement can be signed between the INTO-CPS Association and the Contributors.

"Contribution License" means a license from the INTO-CPS Association to the Contributors of the Contribution, effective on the date of the Transfer of Contribution Copyright, where the INTO-CPS Association grants the Contributors a non-exclusive, world-wide, transferable, free of charge, perpetual license, including sublicensing rights, to use, have used, modify, have modified, reproduce and or have reproduced the contributed material, for business and other purposes, including but not limited to evaluation, development, testing, integration and merging with other software and distribution. The warranty and liability disclaimers of ICAPL apply to this license.

"Contributor" means any person or entity that distributes (part of) the INTO-CPS tool chain.

"The Program" means the Contributions distributed in accordance with ICAPL.

"The INTO-CPS tool chain" means the Contributions distributed in accordance with ICAPL.

"Recipient" means anyone who receives the INTO-CPS tool chain under ICAPL, including all Contributors.

"Licensed Third Party Distributor" means a reseller/distributor having signed a redistribution/resale agreement in accordance with ICAPL and the INTO-CPS Association Bylaws, with a Gold Level organizational member which is not an Affiliate of the reseller/distributor, for distributing a product containing part(s) of the INTO-CPS tool suite. The Licensed Third Party Distributor shall only be allowed further redistribution to other resellers if the Gold Level member is granting such a right to it in the redistribution/resale agreement between the Gold Level member and the Licensed Third Party Distributor.

"Affiliate" shall mean any legal entity, directly or indirectly, through one or more intermediaries, controlling or controlled by or under common control with any other legal entity, as the case may be. For purposes of this definition, the term "control" (including the terms "controlling," "controlled by" and "under common control with") means the possession, direct or indirect, of the power to direct or cause the direction of the management and policies of a legal entity, whether through the ownership of voting securities, by contract or otherwise.

NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THE BY RECIPIENT SELECTED SUBSIDIARY LICENSE CONDITIONS OF ICAPL, THE INTO-CPS ASSOCIATION IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the INTO-CPS tool suite and assumes all risks associated with its exercise of rights under ICAPL , including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THE BY RECIPIENT SELECTED SUBSIDIARY LICENSE CONDITIONS OF ICAPL, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE INTO-CPS TOOL SUITE OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

A Contributor licensing the INTO-CPS tool suite under ICA-Internal-EPL or ICA-External-EPL may choose to distribute (parts of) the INTO-CPS tool suite in object code form under its own license agreement, provided that:

a) it complies with the terms and conditions of ICAPL; or for the case of redistribution of the INTO-CPS tool suite together with proprietary code it is a dual license where the INTO-CPS tool suite parts are distributed under ICAPL compatible conditions and the proprietary code is distributed under proprietary license conditions; and

b) its license agreement: i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose; ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits; iii) states that any provisions which differ from ICAPL are offered by that Contributor alone and not by any other party; and iv) states from where the source code for the INTO-CPS tool suite is available, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the INTO-CPS tool suite is made available in source code form:

a) it must be made available under ICAPL; and

b) a copy of ICAPL must be included with each copy of the INTO-CPS tool suite.

c) a copy of the subsidiary license associated with the selected mode of ICAPL must be included with each copy of the INTO-CPS tool suite.

Contributors may not remove or alter any copyright notices contained within The INTO-CPS tool suite.

If there is a conflict between ICAPL and the subsidiary license conditions, ICAPL has priority.

This Agreement is governed by the laws of Denmark. The place of jurisdiction for all disagreements related to this Agreement, is Aarhus, Denmark.

The EPL 1.0 license definition has been obtained from: http://www.eclipse.org/legal/epl-v10.html. It is also reproduced in the INTO-CPS distribution.

The GPL Version 3 license definition has been obtained from http://www.gnu.org/copyleft/gpl.html. It is also reproduced in the INTO-CPS distribution.

--- End of Definition of INTO-CPS Association Public License ---

8.2 Third Party Software

The DTaaS software platform uses many third-party software. These software components have their own licenses.

8.2.1 User Installations

The list of software included with DTaaS installation scripts are:

Software Package	Usage	License
docker v24.0	mandatory	Apache 2.0
ml-workspace-minimal v0.13	mandatory	Apache 2.0
nodejs v20.10	mandatory	Custom - Modified MIT
npm v10.2	mandatory	Artistic License 2.0
serve	mandatory	MIT
Træfik v2.10	mandatory	MIT
yarn v1.22	mandatory	BSD 2-Clause
eclipse-mosquitto v2	optional	Eclipse Public License-2.0
gitlab-ce v16.4	optional	MIT
Grafana v10.1	optional	GNU Affero General Public (AGPL) License v3.0
InfluxDB v2.7	optional	Apache2, MIT
Mongodb v7.0	optional	AGPL License and Server Side Public License (SSPL) v1
Tabbitmq v3-management	optional	Mozilla Public License
Telegraf v1.28	optional	MIT

8.2.2 Development Environments

Inaddition to all the software included in user installations, the DTaaS development environments may use the following additional software packages.

Software Package	Usage	License
Material for mkdocs	mandatory	MIT
Docker-compose v2.20	optional	Apache 2.0
Jupyter Lab	optional	3-Clause BSD
Microk8s v1.27	optional	Apache 2.0
Openssl	optional	Custom License

8.2.3 Package Dependencies

There are specific software packages included in the development of client, library microservice and runner microservice. These packages can be seen in the **package.json** file of the matching directories.

The plugins of *material for mkdocs* might have their own licenses. The list of plugins used are in **requirements.txt** file.