Digital Twin as a Service (DTaaS)

DTaaS Development Team

Table of contents

1. What is the DTaaS Platform?

The Digital Twin as a Service (DTaaS) software platform is designed to **Build, Use and Share** digital twins (DTs)[^1].

🦾 **Build**: DTs are constructed on DTaaS using reusable DT assets available on the platform.

🧑‍🏫 🧑‍🔧 **Use**: DTs can be executed on the DTaaS platform.

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

Here is an overview of the DTaaS platform 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.

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

1.2 References

[1]: Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." Simulation 101.3 (2025): 287-311.

2. User

2.1 DTaaS for Users

2.1.1 User Guide

This guide is intended for users of the DTaaS platform. Access to a live installation of the DTaaS platform is required. The simplest option is the localhost installation scenario.

The following user-specific Slides and Video provide the conceptual framework behind composable digital twins in the DTaaS platform.

2.1.2 Motivation

A central question in Digital Twin (DT) software platforms is how to enable collaborative activities:

- Building digital twins (DTs)
- Utilizing DTs independently
- Sharing DTs with other users
- Providing existing DTs as a service to other users

Additionally, DT software platforms must address:

- Support for DT lifecycle management
- Scalability through flexible convention over configuration

2.1.3 Existing Approaches

Several solutions have been proposed in recent literature to address these challenges. Notable approaches include:

- Focus on data from Physical Twins (PTs) for analysis, diagnosis, and planning
- Sharing DT assets across upstream and downstream stakeholders
- Evaluating different models of PT
- DevOps methodologies for Cyber Physical Systems (CPS)
- Scaling DT execution and ensemble management of related DTs
- Support for PT product lifecycle

2.1.4 Our Approach

The DTaaS platform adopts the following principles[1]:

- Support for transition from existing workflows to DT frameworks
- Creation of DTs from reusable assets
- Enabling users to share DT assets
- Offering DTs as a Service
- Integration of DTs with external software systems
- Separation of configurations for independent DT components

2.1.5 References

[1]: Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." Simulation 101.3 (2025): 287-311.

2.2 Overview

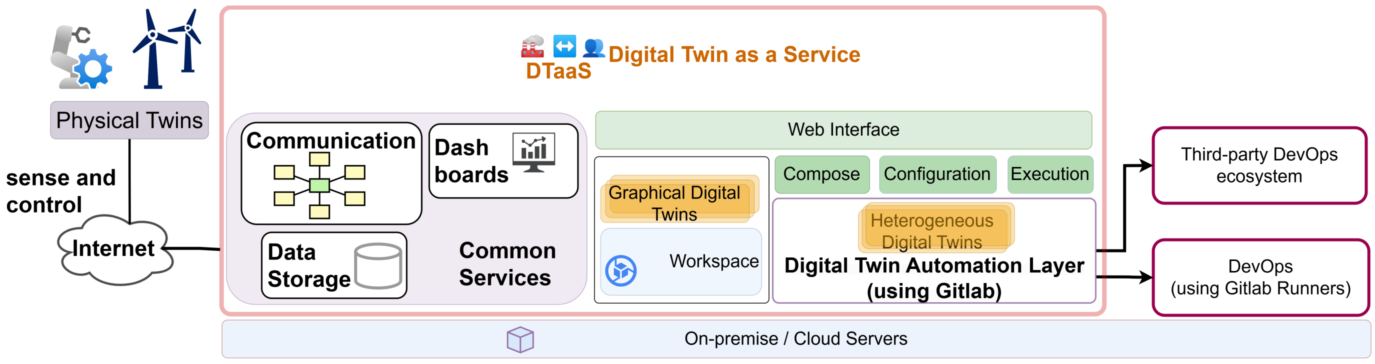
2.2.1 Advantages

The DTaaS software platform provides the following advantages:

- Support for heterogeneous Digital Twin implementations
- CFD, Simulink, co-simulation, FEM, ROM, ML, and other paradigms
- Integration with existing Digital Twin frameworks
- Provision of Digital Twin as a Service capabilities[^1]
- Facilitation of collaboration and asset reuse
- Private workspaces for verification of reusable assets and trial executions of DTs
- Cost effectiveness through shared infrastructure

2.2.2 Software Features

Each installation of the DTaaS platform includes the features illustrated in the following diagram.


Physical Twins
sense and control
Internet
Digital Twin as a Service
DTaaS
Communication
Dash boards
Data Storage
Common Services
Web Interface
Graphical Digital Twins
Workspace
Compose
Configuration
Execution
Heterogeneous Digital Twins
Digital Twin Automation Layer (using Gitlab)
Third-party DevOps ecosystem
DevOps (using Gitlab Runners)
On-premise / Cloud Servers

All users are provided with dedicated workspaces. These workspaces are containerized implementations of Linux Desktops. The user desktops are isolated, ensuring that installations and customizations performed in one workspace do not affect other user workspaces. Graphical digital twins can be executed within these private workspaces.

Each user workspace is provisioned with pre-installed development tools. These tools are accessible directly through a web browser. The following tools are currently available:

Tool	Advantage
Jupyter Lab	Enables flexible creation and use of digital twins and their components through a web browser. All native JupyterLab use cases are supported.
Jupyter Notebook	Facilitates web-based management of files and library assets.
VS Code in the browser	A widely-adopted IDE for software development. Digital twin-related assets can be developed within this environment.
ungit	An interactive git client enabling repository management through a web browser.

In addition, an xfce-based remote desktop is accessible via a VNC client. The VNC client is available directly in the web browser. Desktop software supported by xfce can also be executed within the workspace.

The workspaces maintain Internet connectivity, enabling Digital Twins running in the workspace to interact with both internal and external services.

A DT automation layer is provided for managing DT automation tasks. This layer facilitates creation, modification, and execution of DTs on both on-premise infrastructure and commercial cloud (DevOps) service providers.

The DTaaS software platform includes several pre-installed services. The currently available services are:

Service	Advantage
InfluxDB	Internet of Things (IoT) device management and data visualization platform. This service stores data for digital twins and provides alerting capabilities.
RabbitMQ	Communication broker facilitating message exchange between physical and digital twins.
Grafana	Visualization dashboard service for digital twin data presentation.
MQTT	Lightweight data transfer broker for IoT devices and physical twins providing data to digital twins.
MongoDB	NoSQL document database for storing metadata from physical twins.
PostgreSQL	SQL database server for storing historical and time-series data.
ThingsBoard	an Internet of Things (IoT) device management and data visualization platform

Users can publish and reuse digital twin assets available on the platform. Additionally, digital twins can be executed and made available as services to external clients[1]. These clients need not be registered users of the DTaaS installation.

2.2.3 References

[1]: Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." Simulation 101.3 (2025): 287-311.

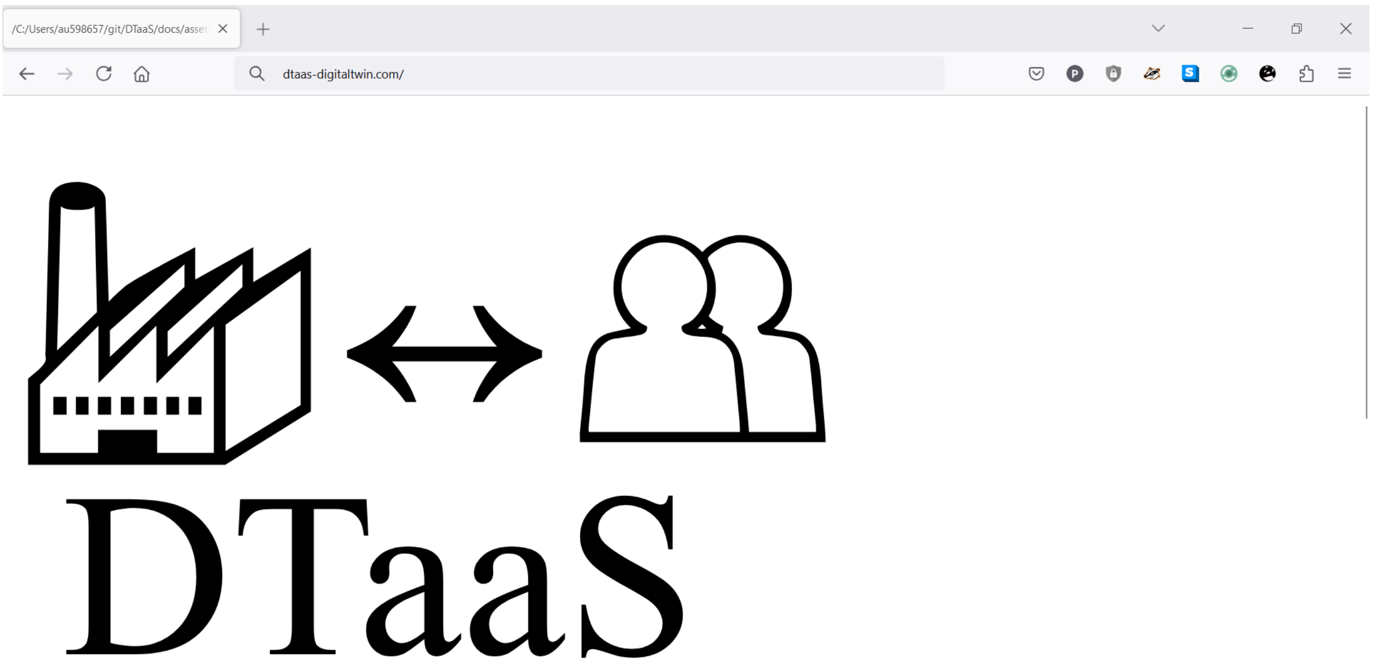
2.3 Website

2.3.1 DTaaS Website Screenshots

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

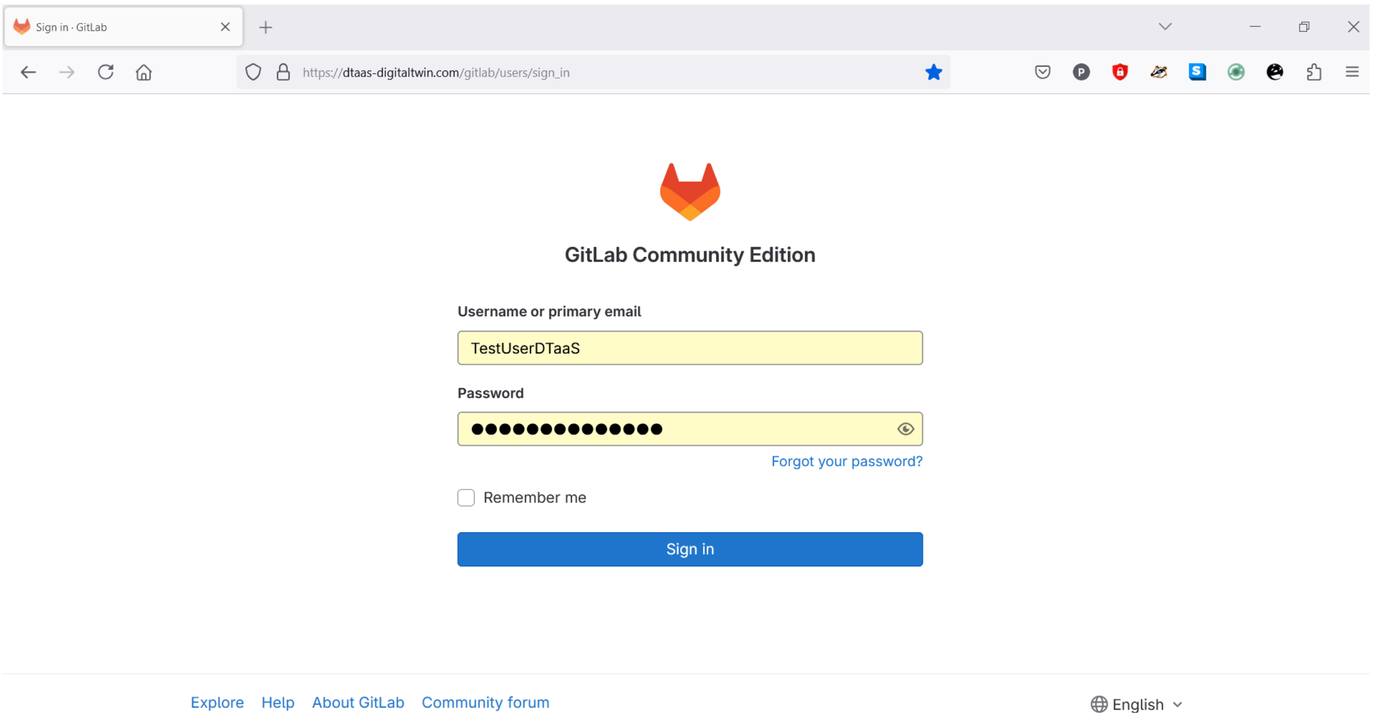
Visit the DTaaS Installation

Navigation begins by visiting the website of the DTaaS instance for which the user is registered.



Redirected to Authorization Provider

The browser redirects to the GitLab Authorization page for the DTaaS.



Username or primary email

TestUserDTaaS

Password

●●●●●●●●●●●●●●

Forgot your password?

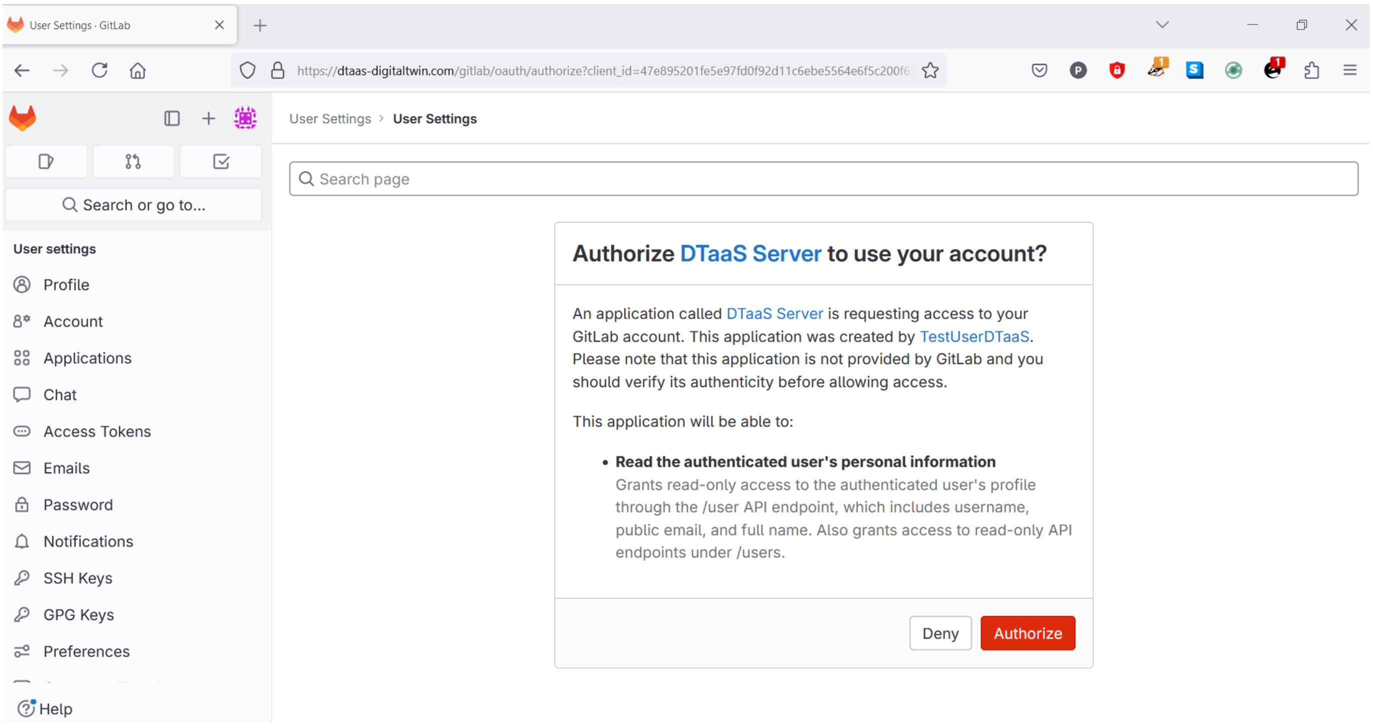
Remember me

Sign in

The email/username and password should be entered. If the email ID registered with the DTaaS matches a GitLab Login email ID.

The browser redirects to the OAuth 2.0 Application page.

Permit DTaaS Server to Use GitLab



Authorize DTaaS Server to use your account?

An application called DTaaS Server is requesting access to your GitLab account. This application was created by TestUserDTaaS. Please note that this application is not provided by GitLab and you should verify its authenticity before allowing access.

This application will be able to:

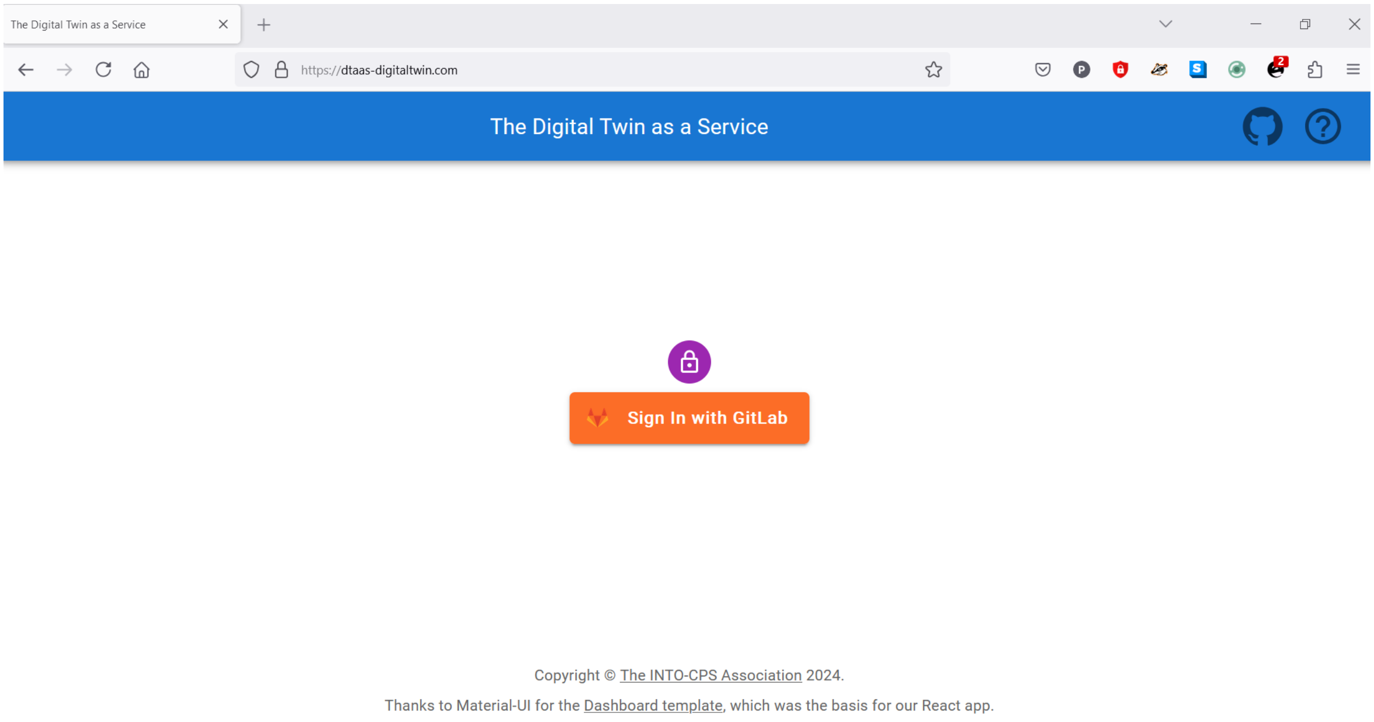
- **Read the authenticated user's personal information**
  Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.

Deny | Authorize

Clicking on Authorize permits the OAuth 2.0 application to access the information associated with the GitLab account. This is a required step.

After successful authentication, redirection to the login page of the DTaaS website occurs.

The DTaaS website employs an additional layer of security - the third-party authorization protocol known as OAuth 2.0. This protocol provides secure access to a DTaaS installation for users with active accounts at the selected OAuth 2.0 service provider. This implementation also uses GitLab as the OAuth 2.0 provider.


The Digital Twin as a Service
https://dtaas-digitaltwin.com
The Digital Twin as a Service
Sign In with GitLab

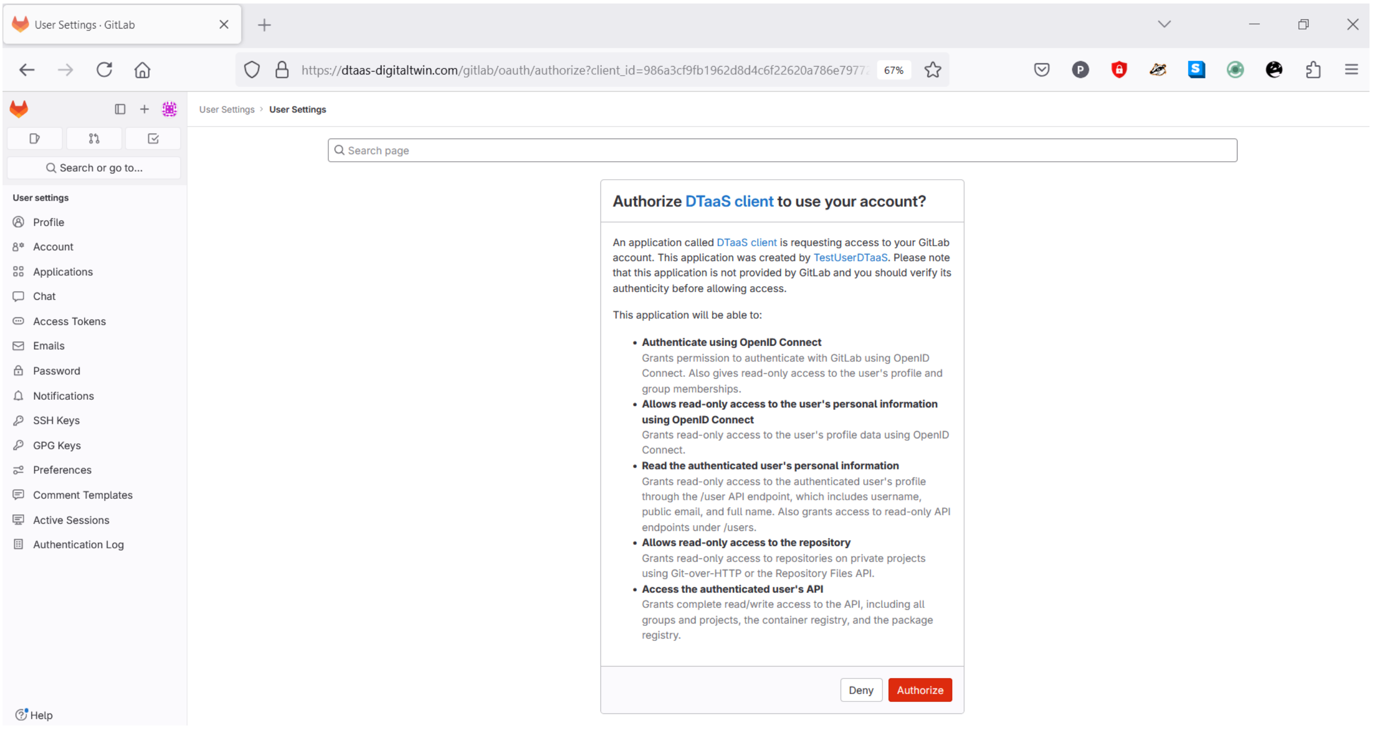
Copyright © The INTO-CPS Association 2024.

Thanks to Material-UI for the Dashboard template, which was the basis for our React app.

The GitLab signin button is displayed. Clicking this button redirects to the GitLab instance providing authorization for DTaaS. Re-authentication to GitLab is not required, unless explicit logout from the GitLab account has occurred.

Permit DTaaS Website to Use GitLab

The DTaaS website requires permission to use the GitLab account for authorization. The **Authorize** button must be clicked.



Authorize DTaaS client to use your account?

An application called DTaaS client is requesting access to your GitLab account. This application was created by TestUserDTaaS. Please note that this application is not provided by GitLab and you should verify its authenticity before allowing access.

This application will be able to:

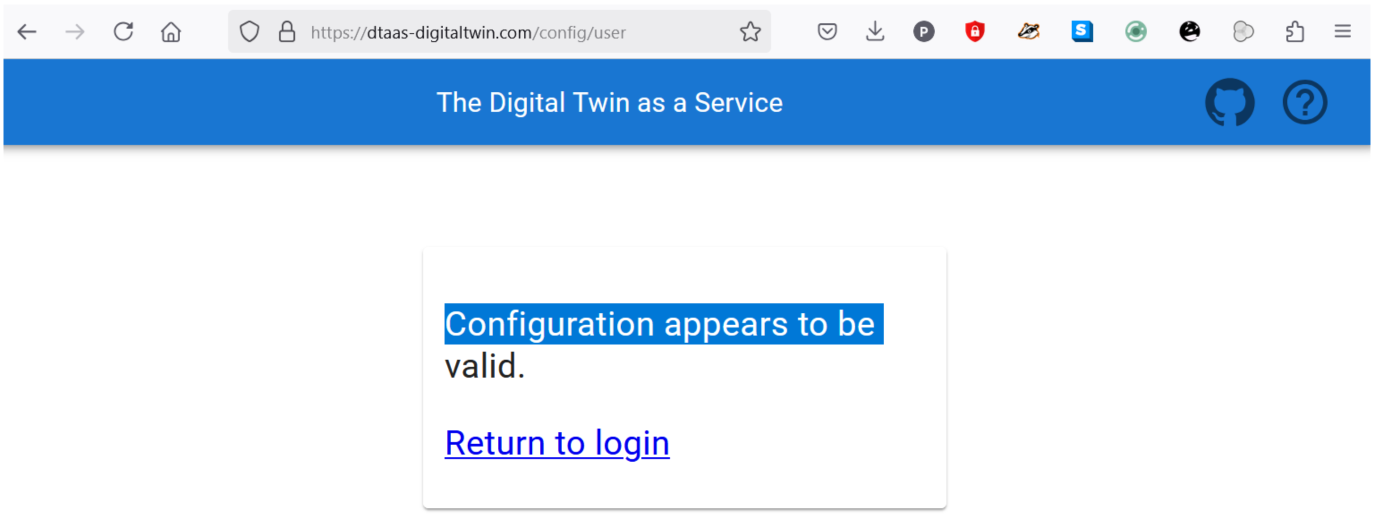
- **Authenticate using OpenID Connect**
  Grants permission to authenticate with GitLab using OpenID Connect. Also gives read-only access to the user's profile and group memberships.
- **Allows read-only access to the user's personal information using OpenID Connect**
  Grants read-only access to the user's profile data using OpenID Connect.
- **Read the authenticated user's personal information**
  Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.
- **Allows read-only access to the repository**
  Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.
- **Access the authenticated user's API**
  Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.

After successful authorization, redirection to the **Library** page of the DTaaS website occurs.

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

Check Website Access

For troubleshooting login issues, the website configuration can be verified by navigating to https://foo.com/config/user. The following display indicates a correctly configured application.


https://dtaas-digitaltwin.com/config/user

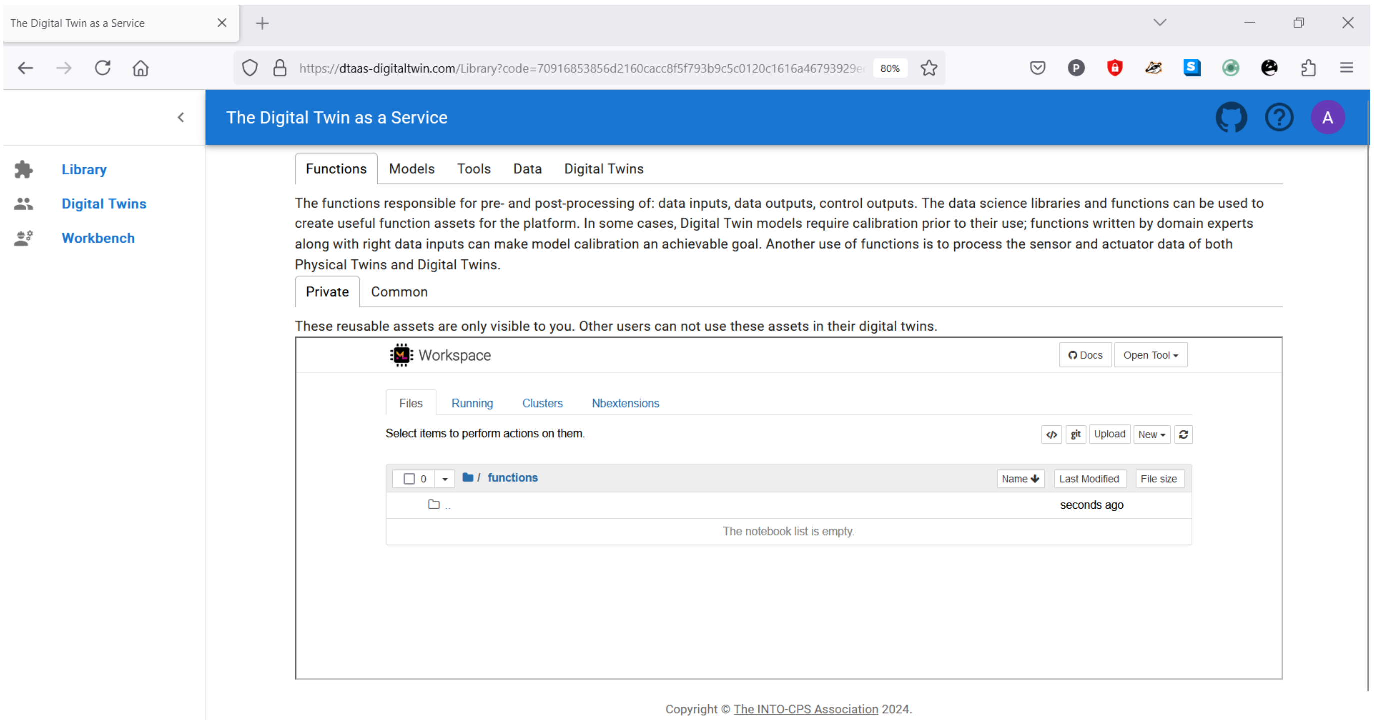
The Digital Twin as a Service

Configuration appears to be valid.

Return to login

Menu Items

The menu is hidden by default. Only the icons of menu items are visible. Clicking on the ☰ icon in the top-left corner of the page reveals the menu.


The Digital Twin as a Service
https://dtaas-digitaltwin.com/Library?code=70916853856d2160cacc8f5f793b9c5c0120c1616a46793929e
80%
The Digital Twin as a Service
A
Library
Digital Twins
Workbench
Functions
Models
Tools
Data
Digital Twins

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.

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

Select items to perform actions on them.

The notebook list is empty.

Copyright © The INTO-CPS Association 2024.

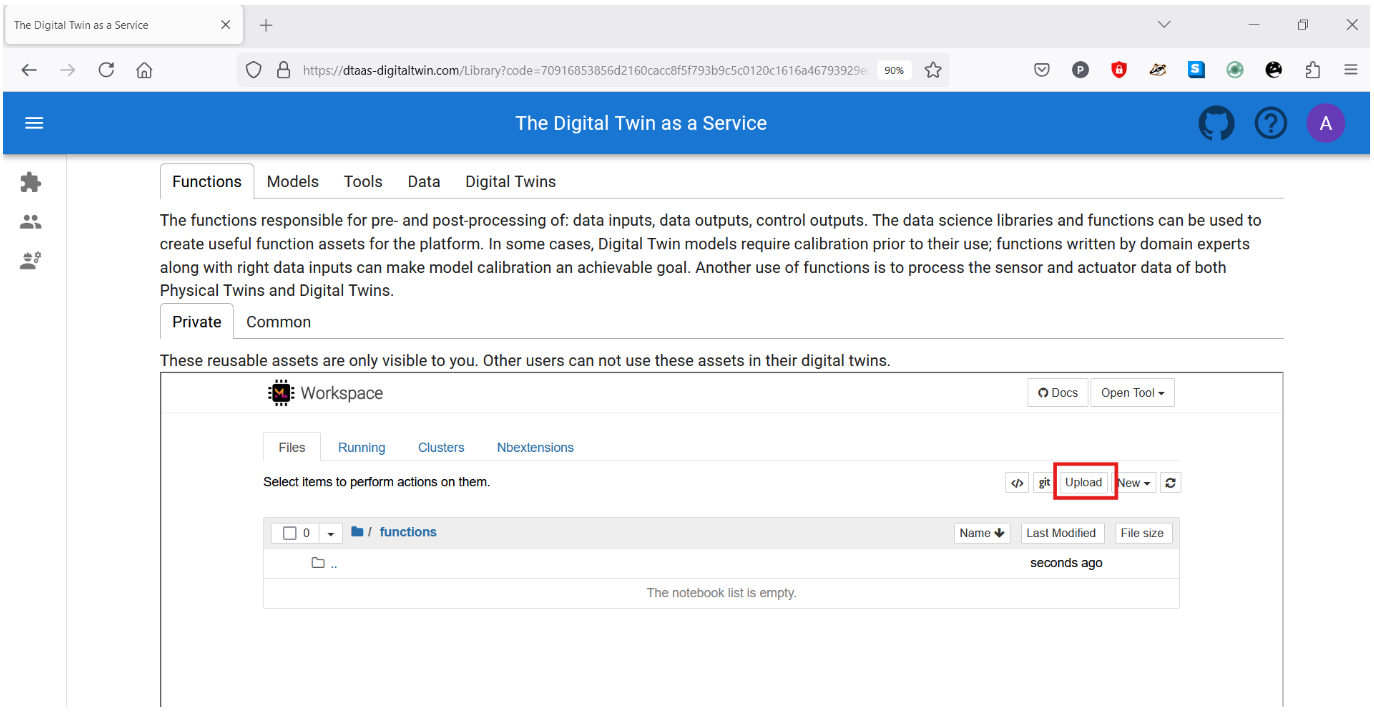
Three menu items are available:

Library: For management of reusable library assets. Files can be uploaded, downloaded, created, and modified on this page.

Digital Twins: For management of digital twins. A Jupyter Lab page is presented from which digital twins can be executed.

Workbench: Not all digital twins can be managed within Jupyter Lab. Additional tools are available on this page.

Library Page



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.

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

Select items to perform actions on them.

The notebook list is empty.

Five tabs are displayed, each corresponding to one type of digital twin asset. Each tab provides help text to guide users on the asset type.

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.

Two sub-tabs exist: **private** and **common**. Library assets in the private category are visible only to the logged-in user, while library assets in the common category are available to all users.

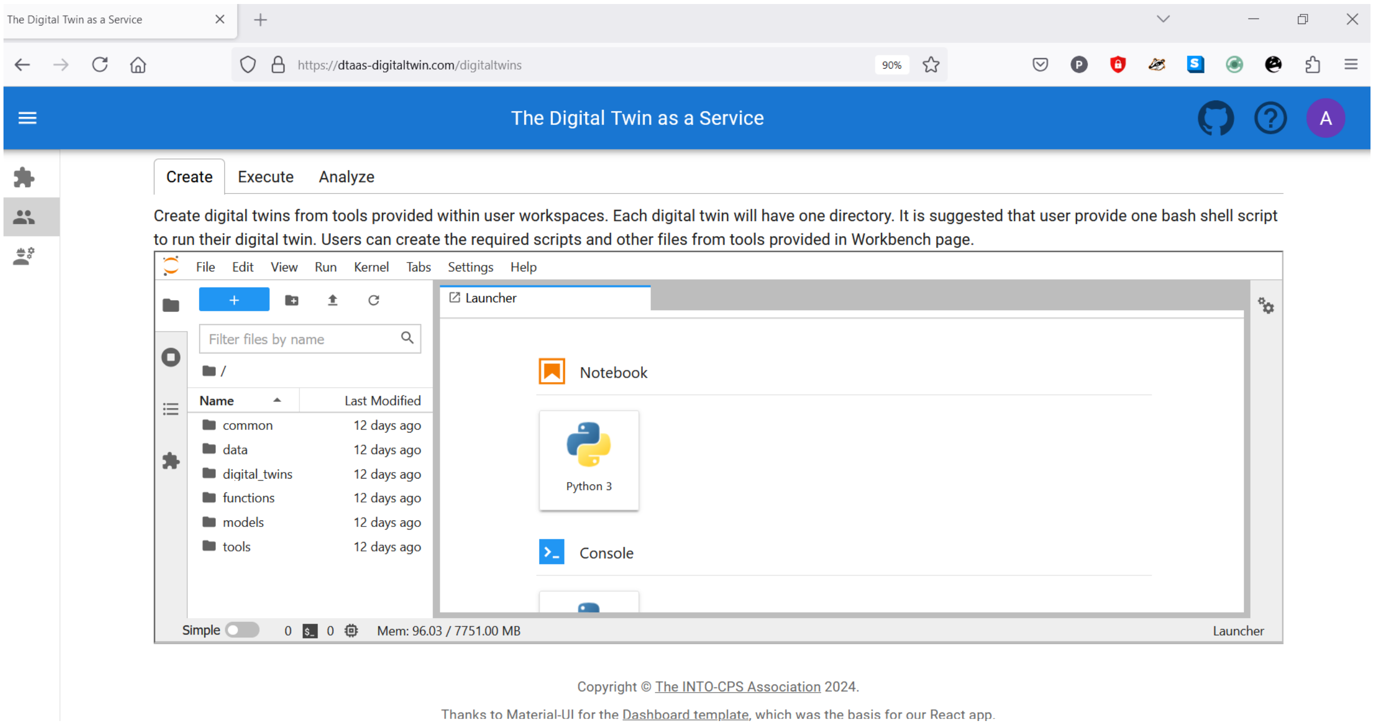
Further explanation on the placement of reusable assets within each type and the underlying directory structure on the server is available on the assets page.

Note

Assets (files) can be uploaded using the **upload** button.

The file manager is based on Jupyter Notebook, and all tasks available in Jupyter Notebook can be performed here.

Digital Twins Page


The Digital Twin as a Service
https://dtaas-digitaltwin.com/digitaltwins
The Digital Twin as a Service
Create
Execute
Analyze
File Edit View Run Kernel Tabs Settings Help
Filter files by name
Launcher
Notebook
Python 3
Console
Simple
Mem: 96.03 / 7751.00 MB
Launcher

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.

Name	Last Modified
common	12 days ago
data	12 days ago
digital_twins	12 days ago
functions	12 days ago
models	12 days ago
tools	12 days ago

Copyright © The INTO-CPS Association 2024.

Thanks to Material-UI for the Dashboard template, which was the basis for our React app.

The digital twins page contains three tabs, and the central pane opens Jupyter Lab. The three tabs provide helpful instructions on suggested tasks for the **Create - Execute - Analyze** lifecycle phases of a digital twin. More explanation is available on the lifecycle 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

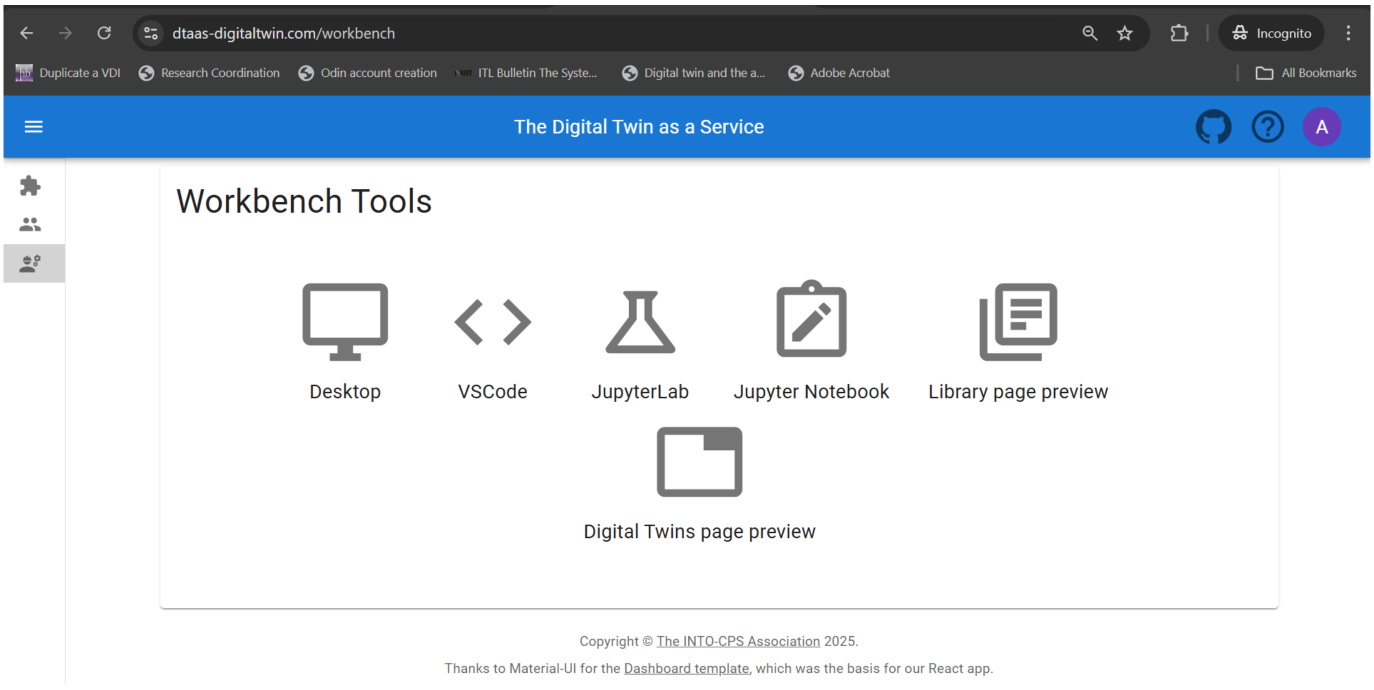
The analysis of digital twins requires running of digital twin script from user workspace. The execution results placed within data directory are processed by analysis scripts and results are placed back in the data directory. These scripts can either be executed from VSCode and graphical results or 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) displayed in the file manager are also available in Jupyter Lab. Additionally, a git plugin is installed in Jupyter Lab that enables linking files with external git repositories.

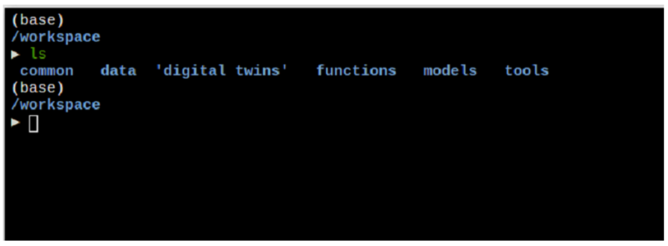
Workbench

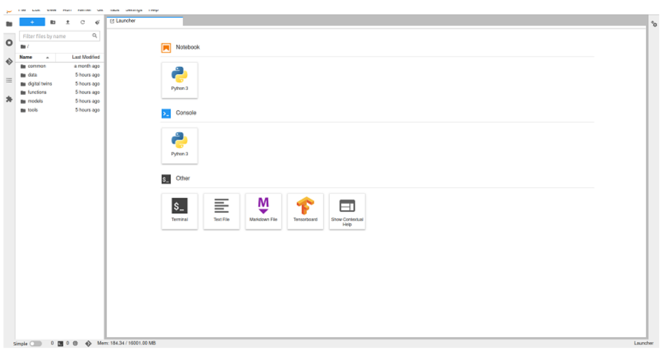
The **workbench** page provides links to four integrated tools:

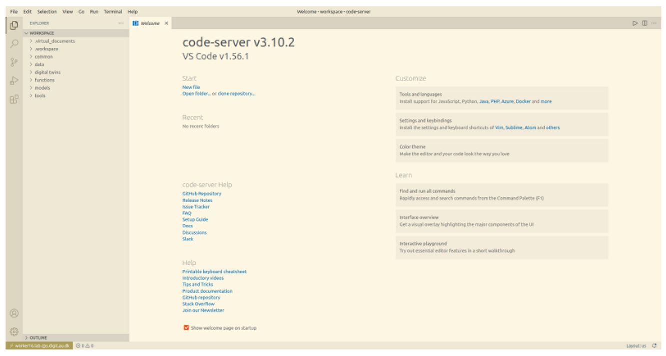
- Desktop
- VS Code
- Jupyter Lab
- Jupyter Notebook

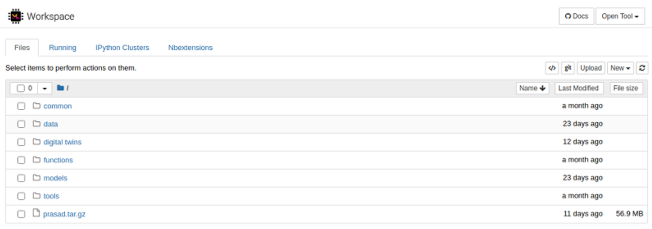

dtaas-digitaltwin.com/workbench
Incognito
Duplicate a VDI
Research Coordination
Odin account creation
ITL Bulletin The Syste...
Digital twin and the a...
Adobe Acrobat
All Bookmarks
The Digital Twin as a Service
A
Workbench Tools
Desktop
VSCode
JupyterLab
Jupyter Notebook
Library page preview
Digital Twins page preview
Copyright © The INTO-CPS Association 2025.
Thanks to Material-UI for the Dashboard template, which was the basis for our React app.

Screenshots of the pages opened in new browsers are shown:


```
(base)
/workspace
▶ ls
 common   data  'digital twins'   functions   models   tools
(base)
/workspace
▶ 
```



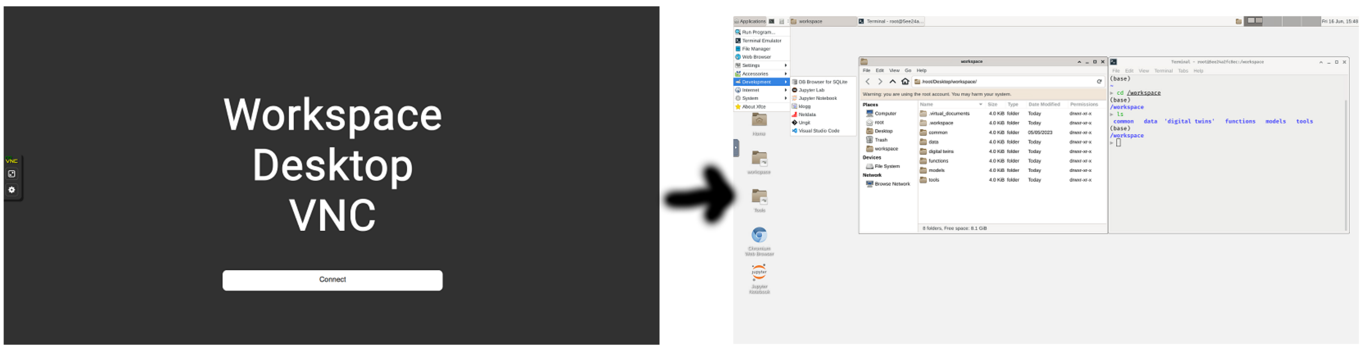



Workspace

Files | Running | IPython Clusters | Nbextensions

Select items to perform actions on them.

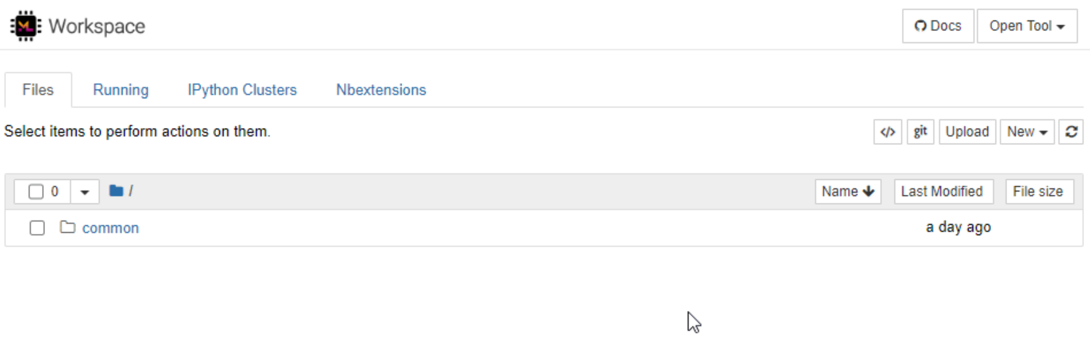
Name	Last Modified	File size
common	a month ago	
data	23 days ago	
digital twins	12 days ago	
functions	a month ago	
models	23 days ago	
tools	a month ago	
prasad.tar.gz	11 days ago	56.9 MB



The hyperlinks open in new browser tabs.

Terminal

The Terminal hyperlink does not exist on the workbench page. For terminal access, the tools dropdown in Jupyter Notebook should be used.


Workspace

Docs | Open Tool

Files | Running | IPython Clusters | Nbextensions

Select items to perform actions on them.

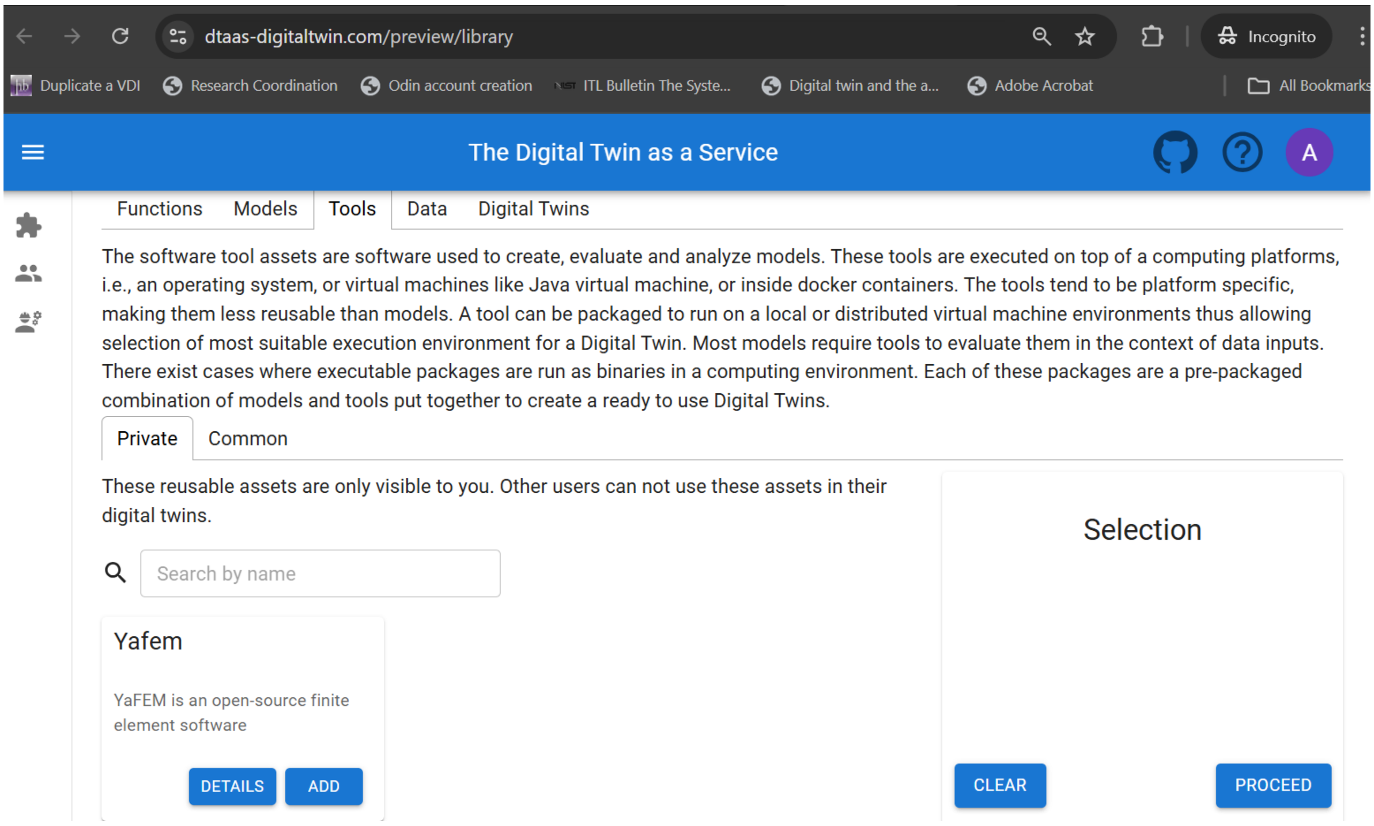
Name	Last Modified	File size
common	a day ago	

The **workbench** also has two links to DevOps-based implementation of composable digital twins.

- Library Page Preview
- Digital Twins Page Preview

LIBRARY PREVIEW PAGE

This page has the same philosophy of Library page and provides similar user interface.



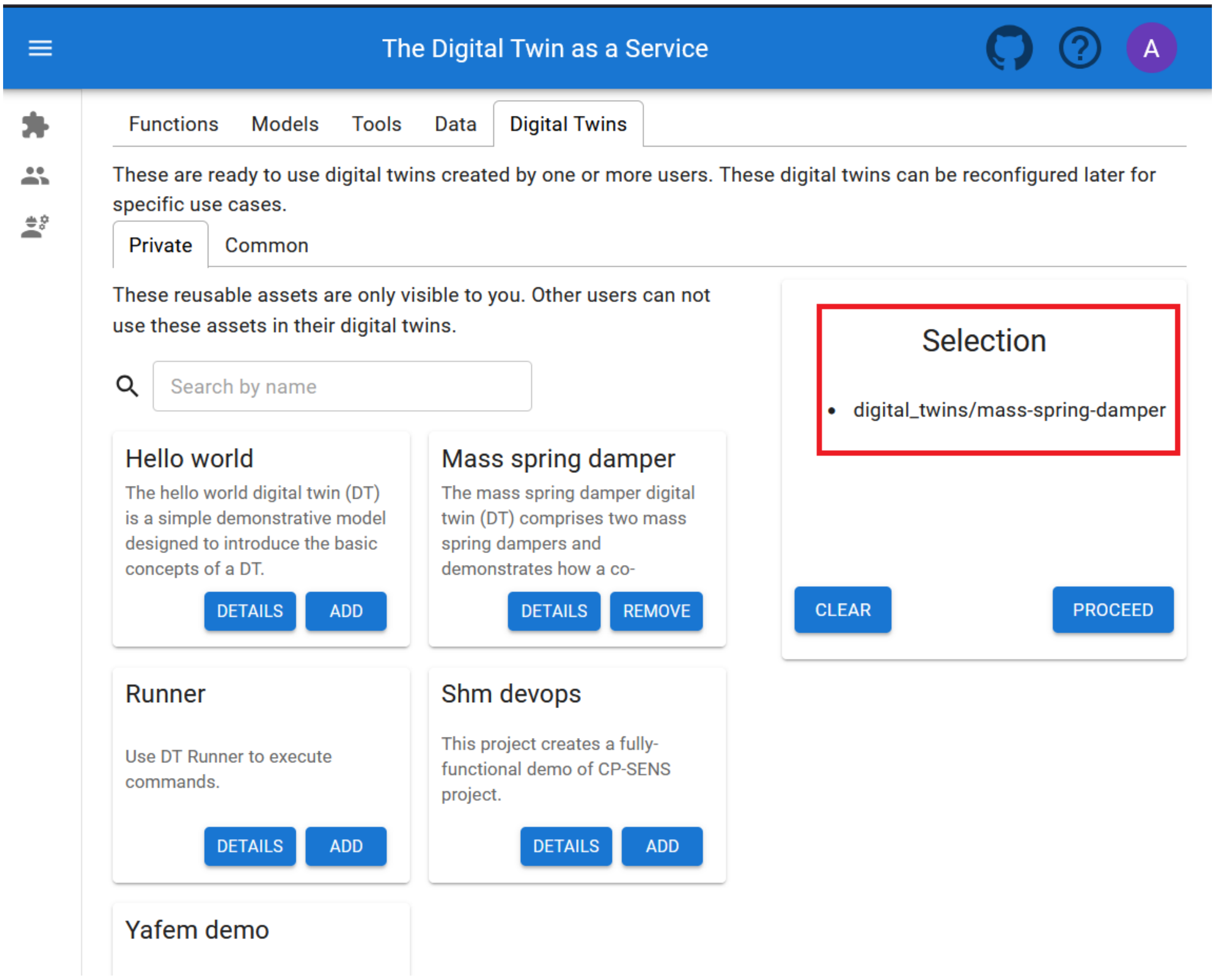
The software tool assets are software used to create, evaluate and analyze models. These tools are executed on top of a computing platforms, i.e., an operating system, or virtual machines like Java virtual machine, or inside docker containers. The tools tend to be platform specific, making them less 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.

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

Yafem

YaFEM is an open-source finite element software

Unlike the Library page, this preview page uses digital twin assets stored in a GitLab repository. New digital twins can be composed by selecting the required library assets.



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

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

Hello world

The hello world digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

Mass spring damper

The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-

Runner

Use DT Runner to execute commands.

Shm devops

This project creates a fully-functional demo of CP-SENS project.

Yafem demo

Selection

- digital_twins/mass-spring-damper

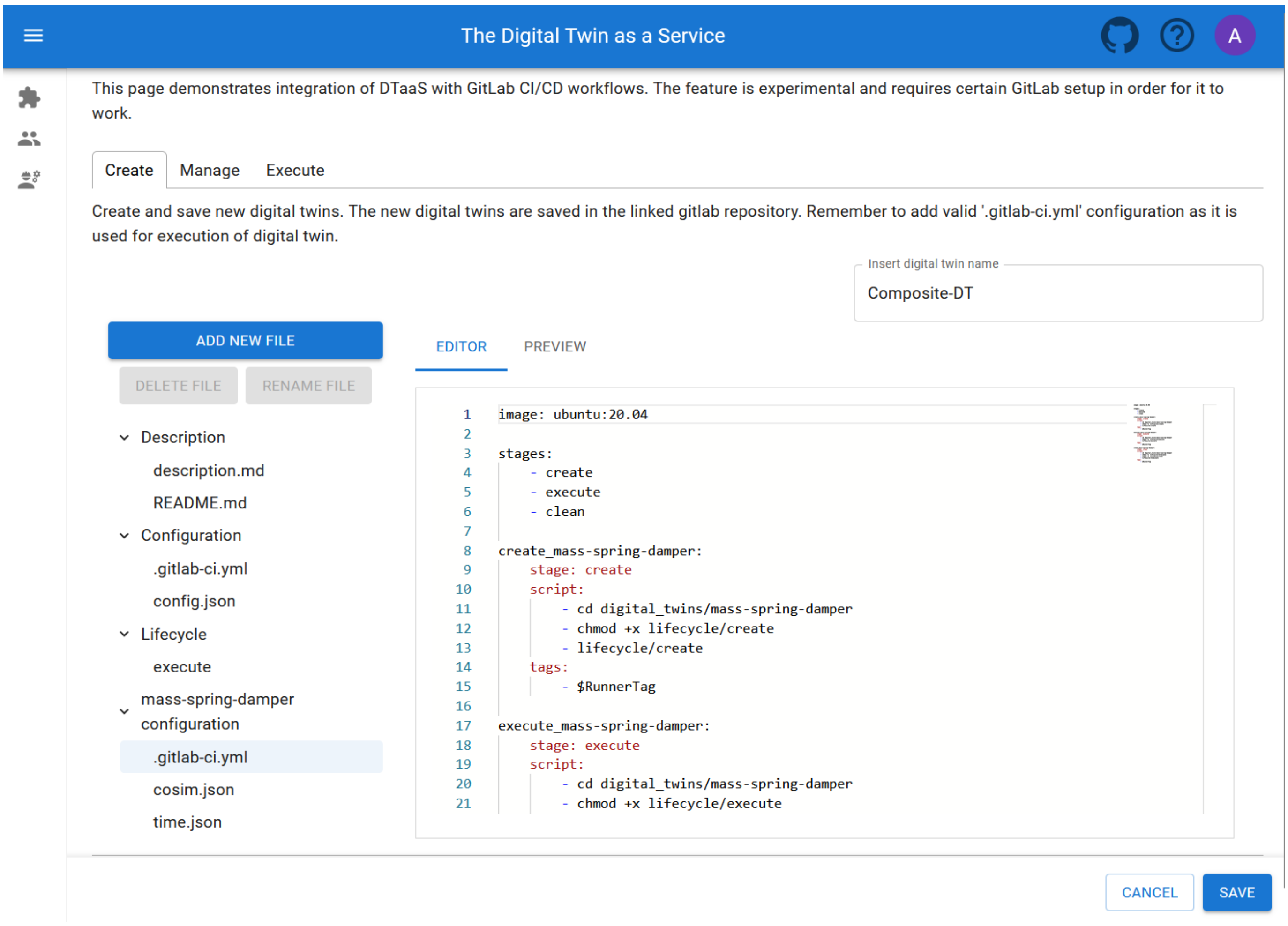
Upon clicking **Proceed** button, the digital twins create tab is opened.

Digital Twins Preview Page

The **Digital Twins Preview Page** provides means of managing digital twins using the DevOps methodology. This page has three tabs, namely **Create**, **Manage** and **Execute**.

CREATE TAB

The library assets selected will be used on the **Create Tab** for creating new digital twins. The new digital twins are saved in the linked GitLab repository. Remember to add valid `.gitlab-ci.yml` configuration as it is used for execution of digital twin.



This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.

Create and save new digital twins. The new digital twins are saved in the linked gitlab repository. Remember to add valid '.gitlab-ci.yml' configuration as it is used for execution of digital twin.

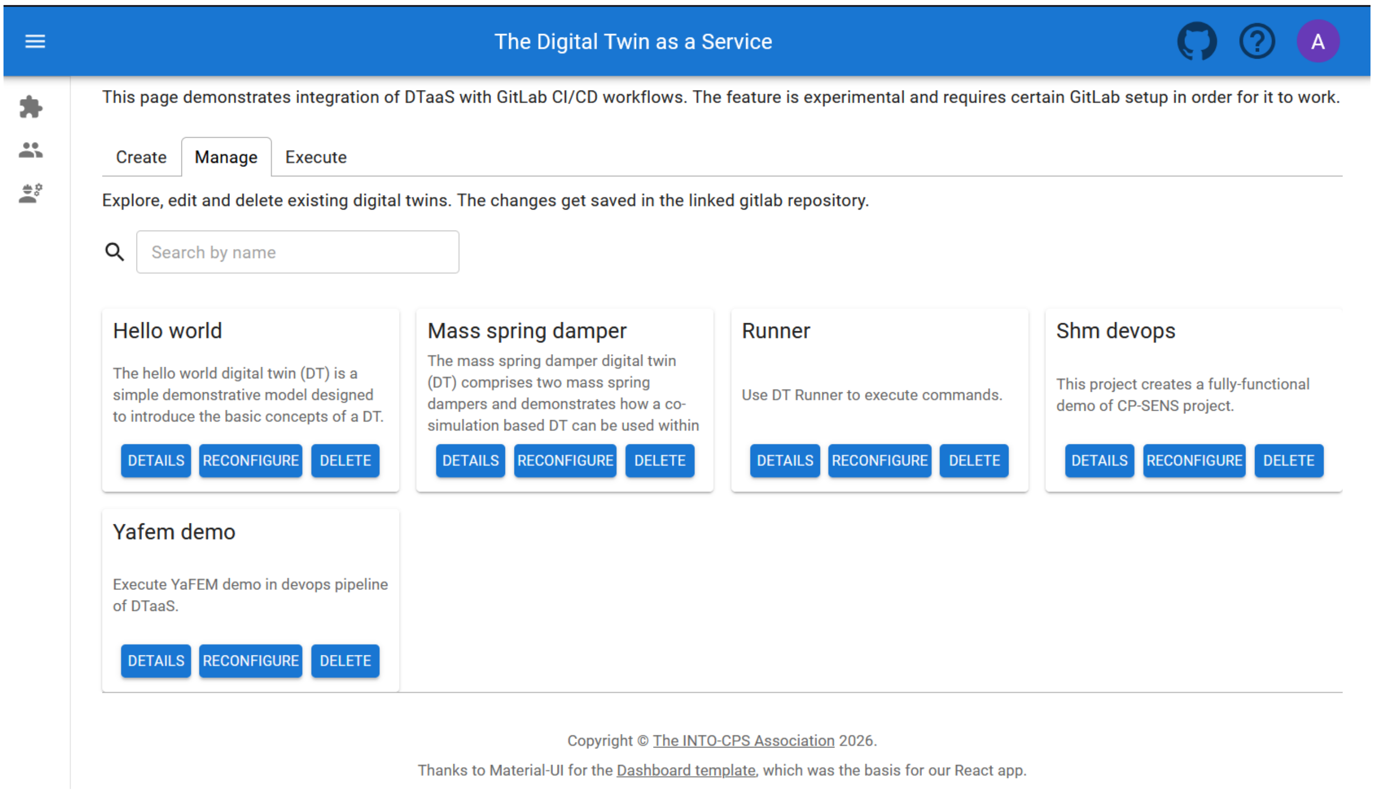
Insert digital twin name: Composite-DT

- Description
  - description.md
  - README.md
- Configuration
  - .gitlab-ci.yml
  - config.json
- Lifecycle
  - execute
- mass-spring-damper configuration
  - .gitlab-ci.yml
  - cosim.json
  - time.json

```
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
```

MANAGE TAB

Complete descriptions of digital twins can be read.



This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.

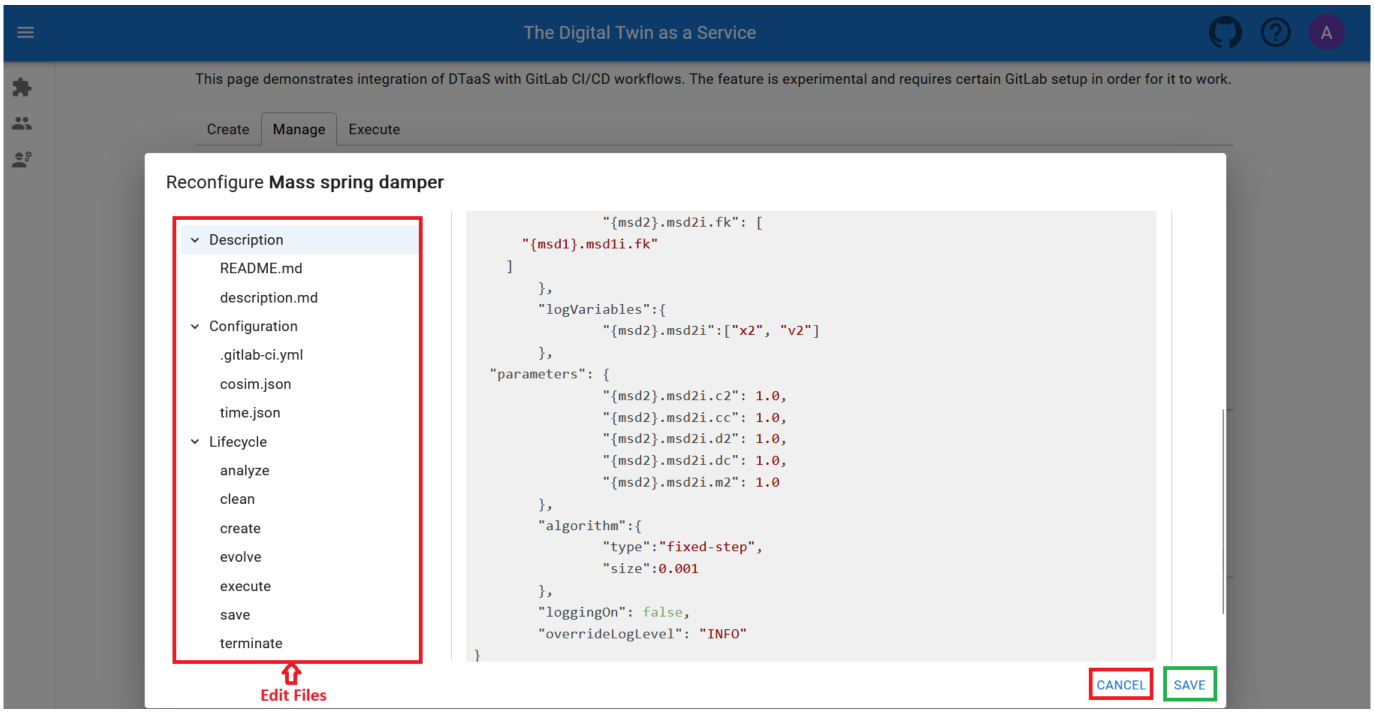
Explore, edit and delete existing digital twins. The changes get saved in the linked gitlab repository.

Hello world	The hello world digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.
Mass spring damper	The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within
Runner	Use DT Runner to execute commands.
Shm devops	This project creates a fully-functional demo of CP-SENS project.
Yafem demo	Execute YaFEM demo in devops pipeline of DTaaS.

Copyright © The INTO-CPS Association 2026.

Thanks to Material-UI for the Dashboard template, which was the basis for our React app.

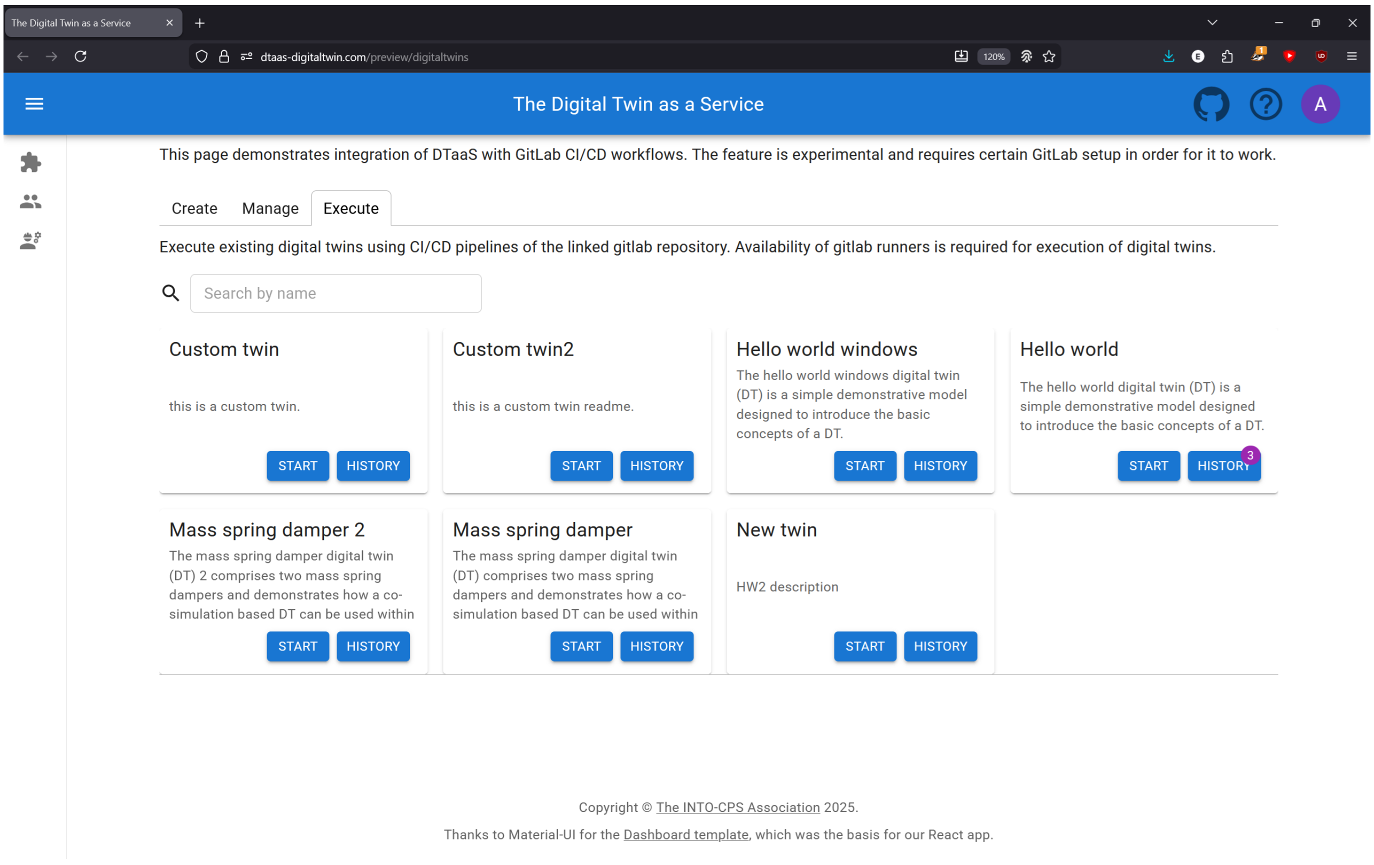
If necessary, a digital twin can be deleted, removing it from the workspace along with all associated data. Digital twins can also be reconfigured.


The Digital Twin as a Service
This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.
Create
Manage
Execute
Reconfigure Mass spring damper
Description
README.md
description.md
Configuration
.gitlab-ci.yml
cosim.json
time.json
Lifecycle
analyze
clean
create
evolve
execute
save
terminate
Edit Files
CANCEL
SAVE

```
                "{msd2}.msd2i.fk": [
       "{msd1}.msd1i.fk"
     ]
        },
        "logVariables":{
                "{msd2}.msd2i":["x2", "v2"]
        },
  "parameters": {
                "{msd2}.msd2i.c2": 1.0,
                "{msd2}.msd2i.cc": 1.0,
                "{msd2}.msd2i.d2": 1.0,
                "{msd2}.msd2i.dc": 1.0,
                "{msd2}.msd2i.m2": 1.0
        },
        "algorithm":{
                "type":"fixed-step",
                "size":0.001
        },
        "loggingOn": false,
        "overrideLogLevel": "INFO"
}
```

EXECUTE TAB

Digital Twins can be executed using GitLab CI/CD workflows. Multiple digital twins can be executed simultaneously.


The Digital Twin as a Service
dtaas-digitaltwin.com/preview/digitaltwins
The Digital Twin as a Service
Create
Manage
Execute
Search by name
START
HISTORY
3

This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.

Execute existing digital twins using CI/CD pipelines of the linked gitlab repository. Availability of gitlab runners is required for execution of digital twins.

Custom twin

this is a custom twin.

Custom twin2

this is a custom twin readme.

Hello world windows

The hello world windows digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

Hello world

The hello world digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

Mass spring damper 2

The mass spring damper digital twin (DT) 2 comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within

Mass spring damper

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

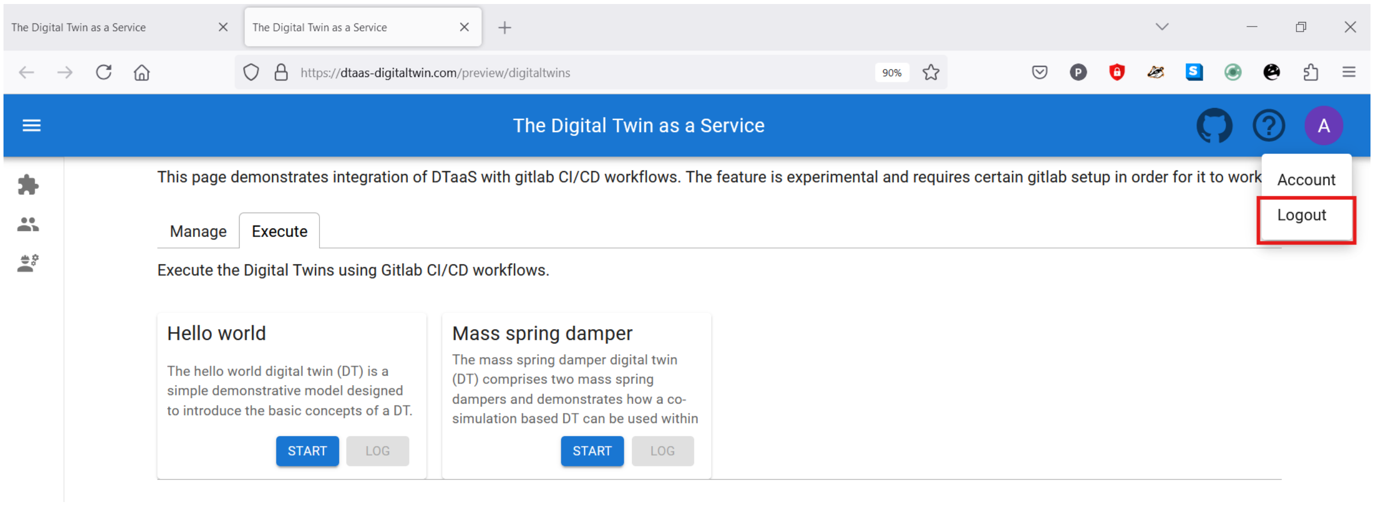
New twin

HW2 description

Copyright © The INTO-CPS Association 2025.

Thanks to Material-UI for the Dashboard template, which was the basis for our React app.

Finally logout


The Digital Twin as a Service
The Digital Twin as a Service
https://dtaas-digitaltwin.com/preview/digitaltwins
90%
The Digital Twin as a Service
A
Account
Logout
Manage
Execute
START
LOG

This page demonstrates integration of DTaaS with gitlab CI/CD workflows. The feature is experimental and requires certain gitlab setup in order for it to work

Execute the Digital Twins using Gitlab CI/CD workflows.

Hello world

The hello world digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

Mass spring damper

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

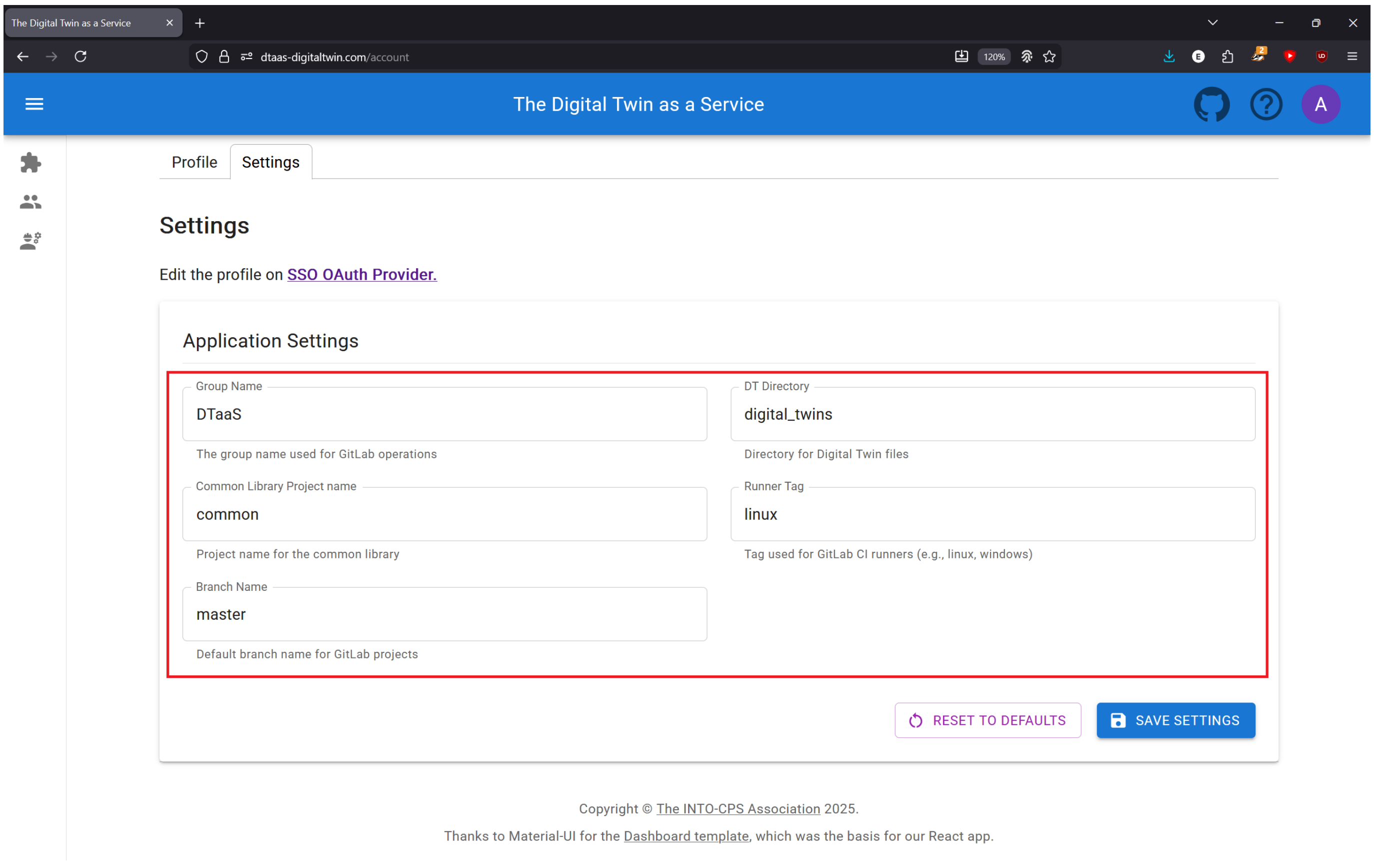
The browser must be closed to completely exit the DTaaS software platform.

2.3.2 Settings

Settings are important both during initial DTaaS setup and during use when working with different configurations. All the most important settings have been consolidated on one page for both of these scenarios.

⚙️ Changing Settings

The parameters used for sending and receiving information to the storage and execution services (e.g., GitLab) may need adjustment to match the infrastructure. Navigation to **Account** using the top-right purple 🅰️ icon followed by selection of the **Settings** tab displays the following adjustable parameters:


The Digital Twin as a Service
dtaas-digitaltwin.com/account
The Digital Twin as a Service
Profile
Settings
RESET TO DEFAULTS
SAVE SETTINGS

Settings

Edit the profile on SSO OAuth Provider.

Application Settings

Group Name: DTaaS
The group name used for GitLab operations

DT Directory: digital_twins
Directory for Digital Twin files

Common Library Project name: common
Project name for the common library

Runner Tag: linux
Tag used for GitLab CI runners (e.g., linux, windows)

Branch Name: master
Default branch name for GitLab projects

Copyright © The INTO-CPS Association 2025.
Thanks to Material-UI for the Dashboard template, which was the basis for our React app.

1. Group Name
2. DT Directory
3. Common Library Project name
4. Runner Tag
5. Branch Name

Following is a description of what each parameter does.

GROUP NAME

The Group Name denotes the highest level of organizational abstraction concerned with on the storage service, namely Groups. A GitLab group is required to use the DTaaS. Within the group, projects reside, which must match the usernames of system users. More information about the file organization is available. This parameter must be set to the case-insensitive name of the group.

Default: DTaaS

COMMON LIBRARY PROJECT NAME

One project within the group serves as the Digital Twins *Library*. Through the DTaaS, the files inside the Library are accessible to all users and can be copied to individual user projects as needed. This parameter specifies the project name of the Library, and must match that name.

Default: common

DT DIRECTORY

Within the common library and user projects, files related to Digital Twins are stored within a designated folder. This is the name chosen for that folder.

Default: Digital_Twins

BRANCH NAME

This parameter determines which branch to search for data (Twins, Functions, etc.) within user and library projects. This parameter also determines which branch's Digital Twins are executed.

Default: master

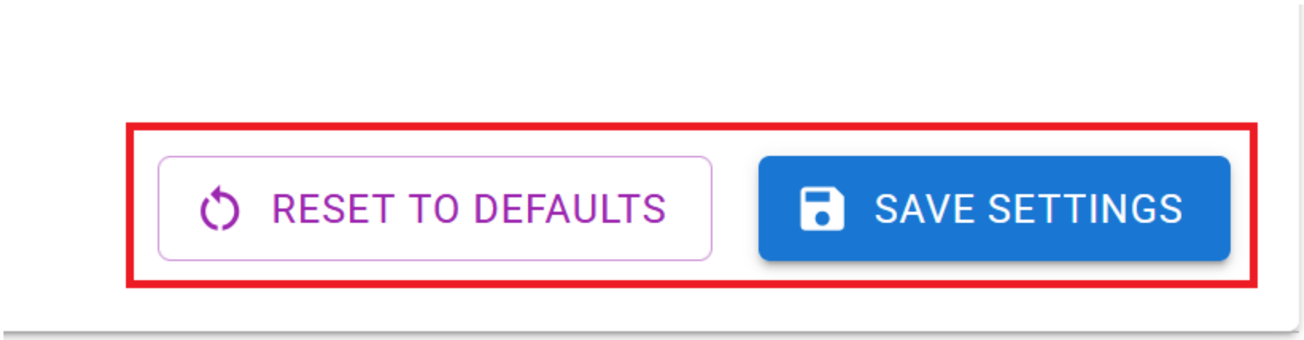
RUNNER TAG

The (GitLab) runners responsible for executing Digital Twin code must be associated with a tag. Only one tag can be specified, and it **cannot** be left blank, or the job of running the twin will not be processed. This is a limitation of the DTaaS.

Default: linux

💾 Saving and Resetting your changes

When satisfied with the changes, **SAVE SETTINGS** must be pressed for them to persist after leaving the Settings page. If a mistake was made, the settings can be reset to their default values by pressing the **RESET TO DEFAULTS** button. The reset values are saved automatically, so additional saving is not required. The Saving and Resetting buttons on the page:



The default values can be found and modified in the code if needed.

Return to the DevOps pages (i.e., Digital Twin Preview) is now possible. **Refreshing ensures fresh data from the remote repository is fetched**, and the Digital Twins should be visible, ready to be executed, edited, and shared.

💭 Summary

This document has described how to edit the settings for initializing the DTaaS to a project and for continuous use (i.e., modifying Runner Tag and Branch). The need to save changes and how to return to default values if a mistake is made have been discussed.

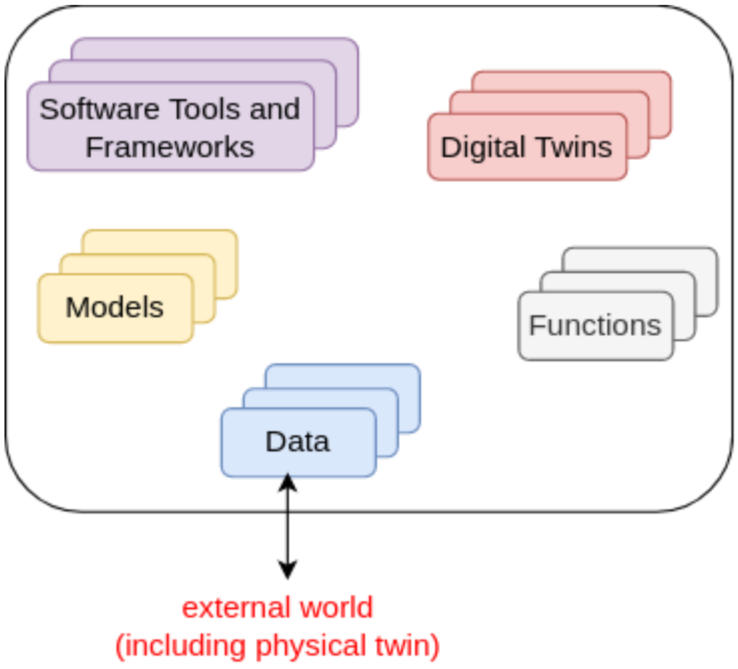
2.4 Reusable Assets

2.4.1 Reusable Assets

The reusability of digital twin assets facilitates efficient work with digital twins. Reusability of assets is a fundamental feature of the platform[1].

Kinds of Reusable Assets

The DTaaS platform categorizes all 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/
```

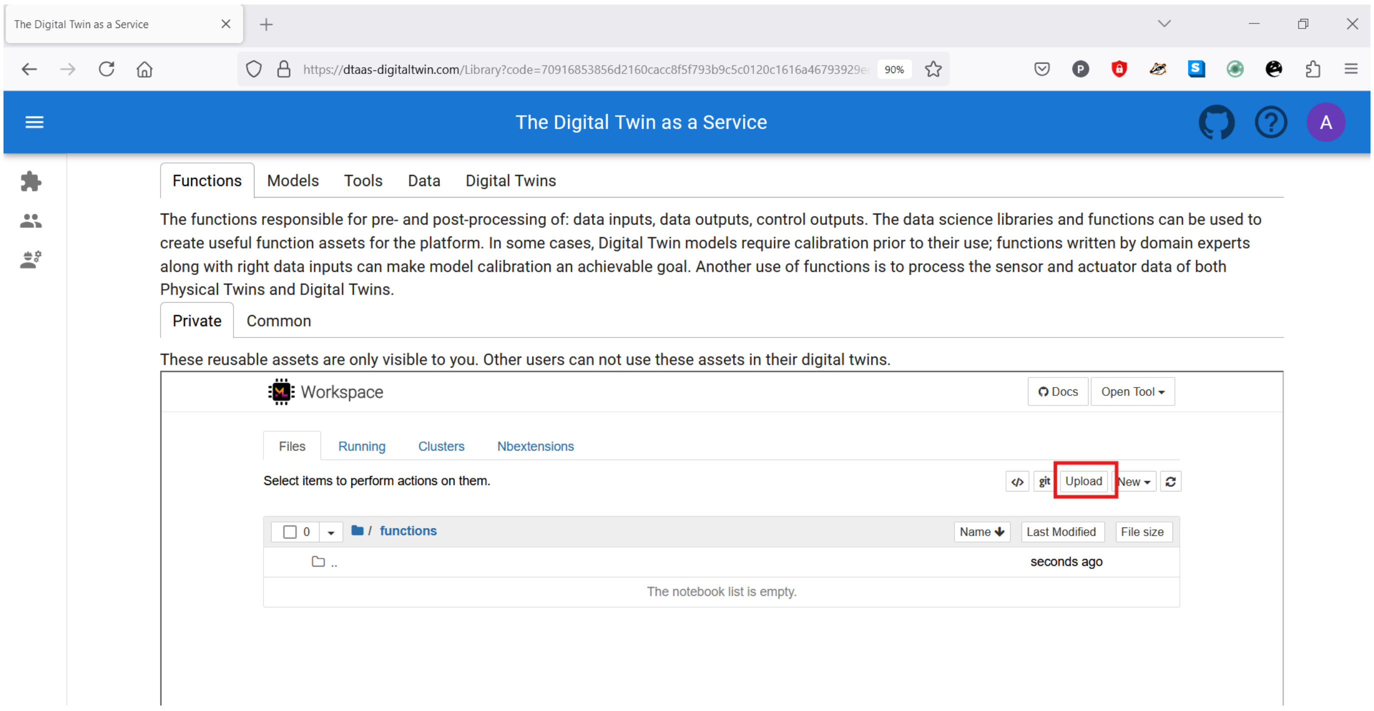
Tip

The DTaaS is agnostic to the format of assets. The only requirement is that they are files which can be uploaded on the Library page. Directories can be compressed as single files and uploaded. The files can be decompressed into directories 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 the Library page of the website.



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.

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

Select items to perform actions on them.

The notebook list is empty.

Navigation into a directory followed by clicking on the **upload** button allows uploading files or directories into the workspace. These assets then become available in all workbench tools. New assets can also be created on the page by clicking on the **new** dropdown menu. This simple web interface allows creation of text-based files. Other files must be uploaded using the **upload** button.

The user workbench provides the following services:

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

Users can also bring DT assets into user workspaces from external sources using any of the above-mentioned services. 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.

References

[1]: Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." Simulation 101.3 (2025): 287-311.

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.

The assets page describes suggested storage conventions for library assets.

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

Application Programming Interface (API)

The lib microservice application provides services at two end points:

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

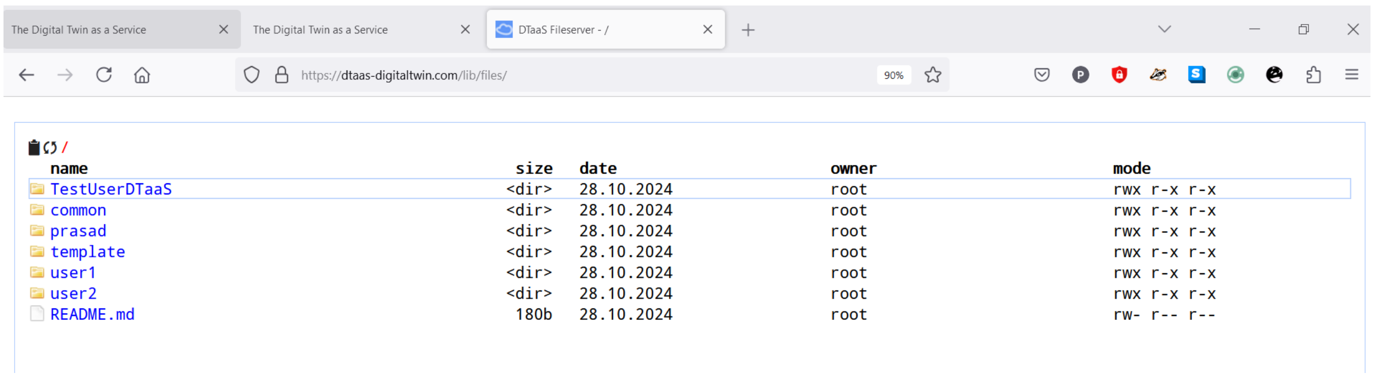
HTTP Endpoint: `http://foo.com/lib/files`

HTTP PROTOCOL

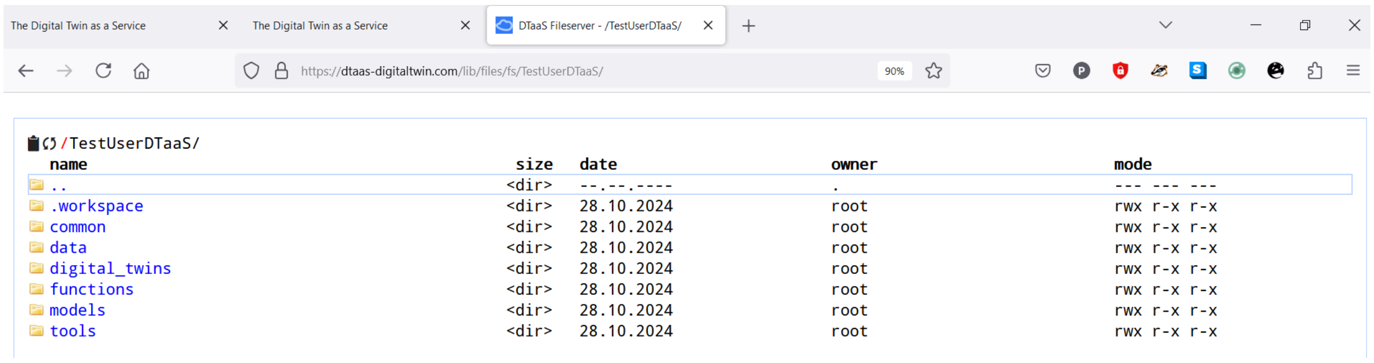
Endpoint: `localhost:PORT/lib/files`

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

Here are sample screenshots.


The Digital Twin as a Service
The Digital Twin as a Service
DTaaS Fileserver - /
https://dtaas-digitaltwin.com/lib/files/
90%

```
/
name              size   date         owner   mode
TestUserDTaaS     <dir>  28.10.2024   root    rwx r-x r-x
common            <dir>  28.10.2024   root    rwx r-x r-x
prasad            <dir>  28.10.2024   root    rwx r-x r-x
template          <dir>  28.10.2024   root    rwx r-x r-x
user1             <dir>  28.10.2024   root    rwx r-x r-x
user2             <dir>  28.10.2024   root    rwx r-x r-x
README.md         180b   28.10.2024   root    rw- r-- r--
```


The Digital Twin as a Service
The Digital Twin as a Service
DTaaS Fileserver - /TestUserDTaaS/
https://dtaas-digitaltwin.com/lib/files/fs/TestUserDTaaS/
90%

```
/TestUserDTaaS/
name              size   date         owner   mode
..                <dir>  --.--.----   .       --- --- ---
.workspace        <dir>  28.10.2024   root    rwx r-x r-x
common            <dir>  28.10.2024   root    rwx r-x r-x
data              <dir>  28.10.2024   root    rwx r-x r-x
digital_twins     <dir>  28.10.2024   root    rwx r-x r-x
functions         <dir>  28.10.2024   root    rwx r-x r-x
models            <dir>  28.10.2024   root    rwx r-x r-x
tools             <dir>  28.10.2024   root    rwx r-x r-x
```

GRAPHQL PROTOCOL

Endpoint: `localhost:PORT/lib`

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

The query schema can be examined and sample queries can be tested there. GraphQL queries must be sent as HTTP POST requests to receive responses.

The library microservice provides 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 endpoints. The format of accepted queries is:

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

```json
{
  "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

```http
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
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.

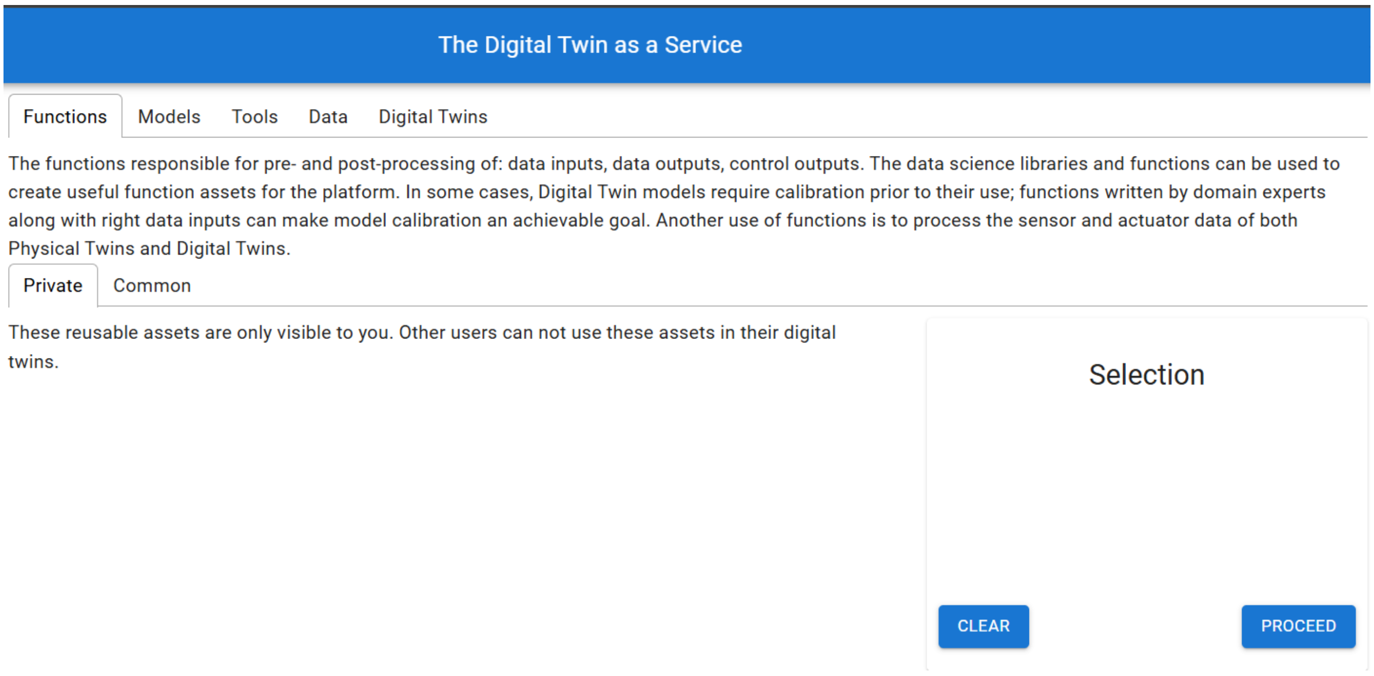
2.4.3 Reusable Assets and DevOps

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

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.

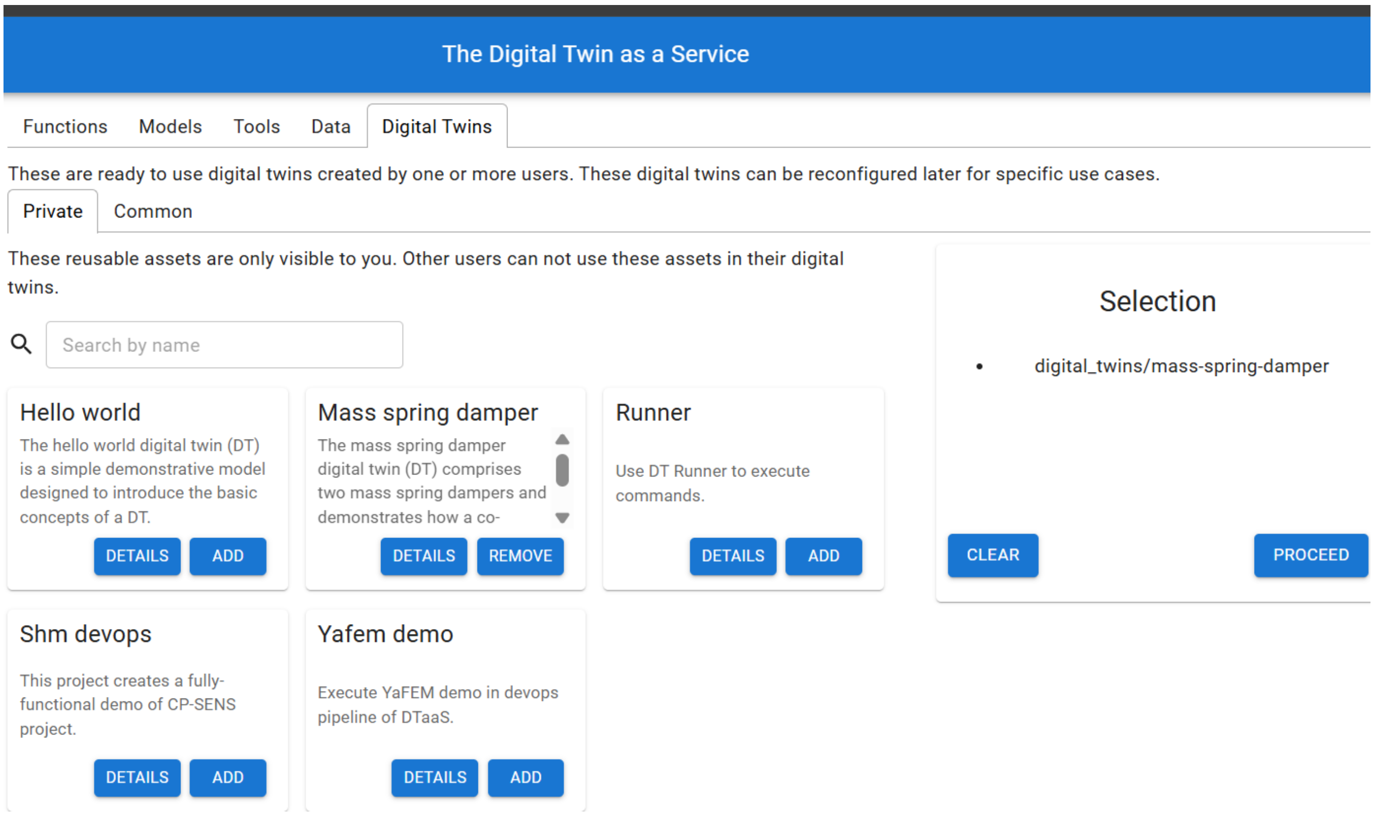
Once these requirements are satisfied, the **Library** page shows all the reusable assets available stored in the linked GitLab repository. An empty list is shown if there are no assets of a specific category.



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.

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

The page gets populated with any existing assets. All available DTs are shown in the following figure.



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

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

Hello world
The hello world digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

Mass spring damper
The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-

Runner
Use DT Runner to execute commands.

Shm devops
This project creates a fully-functional demo of CP-SENS project.

Yafem demo
Execute YaFEM demo in devops pipeline of DTaaS.

Selection
- digital_twins/mass-spring-damper

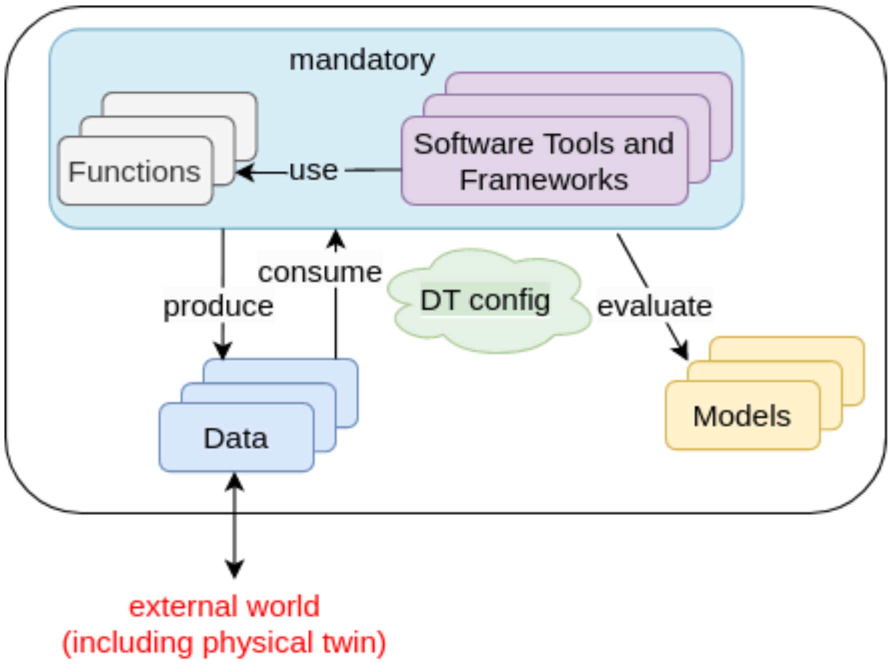
Any existing DT asset can be selected and it gets added to the **Selection** pane on the right. After selecting all the required assets, you can click on *Proceed* button to transition to DT create stage

2.5 Digital Twins

2.5.1 Create a Digital Twin

The first step in digital twin creation involves utilizing the available assets within the workspace. For assets or files residing on a local computer that need to be accessible in the DTaaS workspace, the instructions provided in library assets should be followed.

Dependencies exist among the library assets. These dependencies are illustrated below.


mandatory
Functions
use
Software Tools and Frameworks
produce
consume
DT config
evaluate
Data
Models
external world (including physical twin)

A digital twin can only be created by linking assets in a meaningful way. This relationship can be expressed using the following 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 a DT from D, M, T, and F assets. The indicates zero or more instances of an asset, and indicates one or more instances of an asset.

The DT configuration specifies the relevant assets to use and the potential parameters to be set for these assets. When a DT requires RabbitMQ, InfluxDB, or similar services supported by the platform, the DT configuration must include access credentials for these services.

This generic DT definition is based on DT examples observed in practice. Deviation from this definition is permissible. The only requirement is the ability to execute the DT from either the command line or a graphical desktop environment.

Tip

For users new to Digital Twins who may not have distinct digital twin assets but rather a single directory containing all components, it is recommended to upload this monolithic digital twin into the **digital_twin/your_digital_twin_name** directory.

Example

The Examples repository contains a co-simulation setup for a mass-spring-damper system. This example demonstrates the application of co-simulation techniques 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/` directory contains `input` and `output` data for the mass-spring-damper digital twin.

The two FMU models required for this digital twin are located in the `models/` directory.

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

The digital twin configuration is specified in the `digital twins/mass-spring-damper` directory. The co-simulation configuration is defined in two JSON files: `cosim.json` and `time.json`. Documentation for the digital twin can be placed in `digital twins/mass-spring-damper/document.md`.

The launch program for this digital twin is located in `digital twins/mass-spring-damper/lifecycle/execute`. This launch program executes the co-simulation digital twin, which runs until completion and then terminates. The programs in `digital twins/mass-spring-damper/lifecycle` are responsible for lifecycle management of this digital twin. The lifecycle page provides further explanation of 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.

Execution of this digital twin involves the following steps:

1. Navigate to the Workbench tools page of the DTaaS website and open VNC Desktop. This opens a new tab in the browser.
2. A page with VNC Desktop and a connect button is displayed. Click on Connect to establish a connection to the Linux Desktop of the workspace.
3. Open a Terminal (the black rectangular icon in the top left region of the tab) and enter the following commands.
4. Download the example files by following the instructions provided in the examples overview.
5. Navigate to the digital twin directory and execute:

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

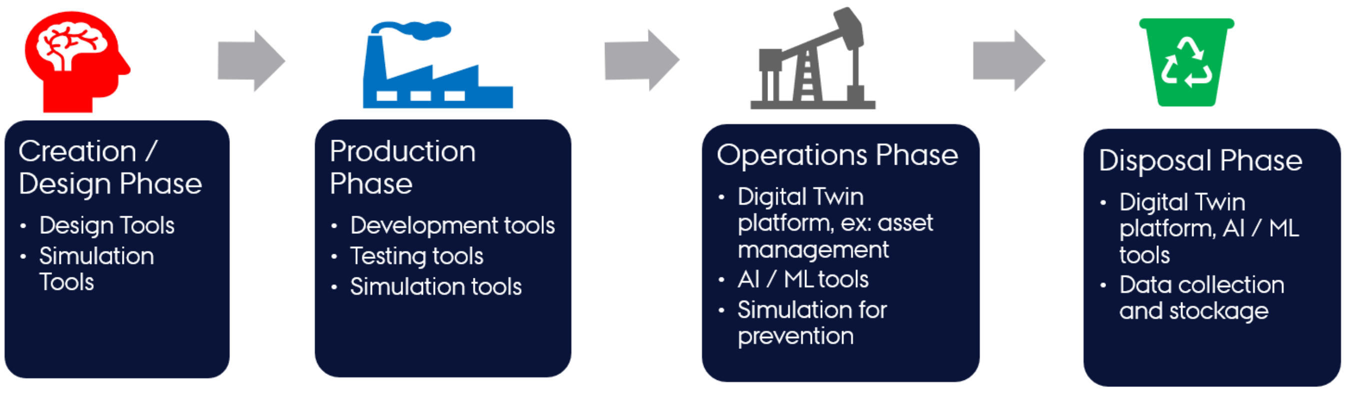
The final command executes the mass-spring-damper digital twin and stores the co-simulation output in `data/mass-spring-damper/output`.

2.5.2 ♻️ Digital Twin Lifecycle

Physical products in the real world undergo a product lifecycle. A simplified four-stage product lifecycle is illustrated here.

A digital twin tracking physical products (twins) must evolve in conjunction with the corresponding physical twin.

The possible activities undertaken in each lifecycle phase 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

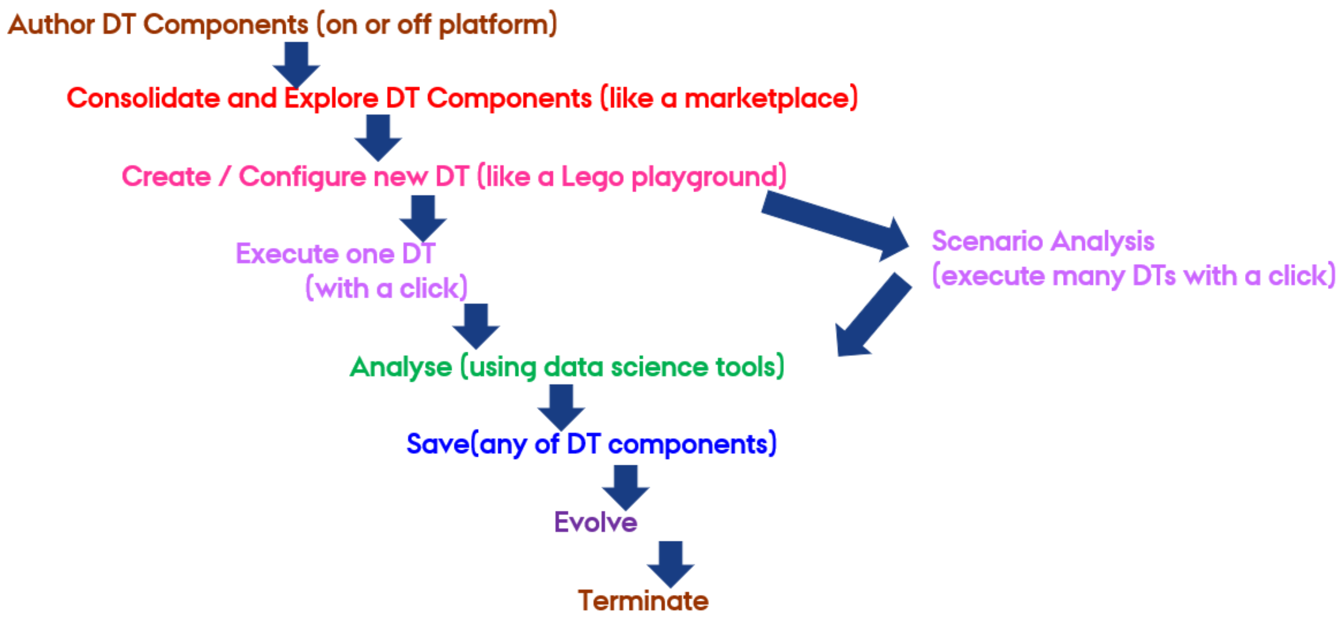
The four-phase lifecycle has been extended to a lifecycle with eight phases. The new phase names and the typical activities undertaken in each phase are outlined in this section[1].

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

Phase	Main Activities
explore	Selection of suitable assets based on user requirements and verification of their compatibility for DT creation.
create	Specification of DT configuration. For existing DTs, no creation phase is required at the time of reuse.
execute	Automated or manual execution of a DT based on its configuration. The DT configuration must be verified before starting execution.
analyse	Examination of DT outputs and decision-making. Outputs may include text files or visual dashboards.
evolve	Reconfiguration of DT primarily based on analysis results.
save	Preservation of DT state to enable future recovery.
terminate	Cessation of DT execution.

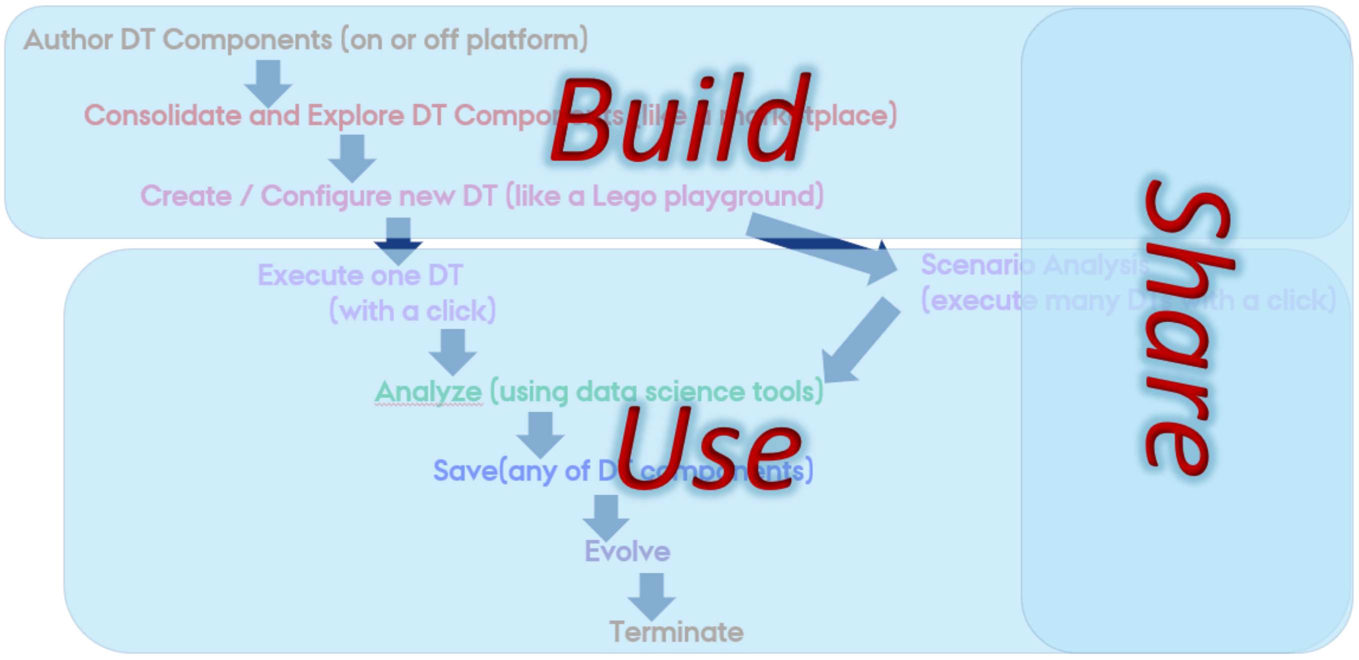
A digital twin faithfully tracking the physical twin lifecycle must support all the phases. Digital twin engineers may also add additional phases to their implementations. Consequently, the DTaaS platform is designed to accommodate the needs of diverse DTs.

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


Author DT Components (on or off platform)
Consolidate and Explore DT Components (like a marketplace)
Create / Configure new DT (like a Lego playground)
Execute one DT (with a click)
Scenario Analysis (execute many DTs with a click)
Analyse (using data science tools)
Save(any of DT components)
Evolve
Terminate

This representation shows only one possible pathway. The sequence of steps may be altered as needed.

It is possible to map the lifecycle phases to the **Build-Use-Share** approach of the DTaaS platform.


Author DT Components (on or off platform)
Consolidate and Explore DT Components (like a marketplace)
Build
Create / Configure new DT (like a Lego playground)
Execute one DT (with a click)
Scenario Analysis (execute many DTs with a click)
Share
Analyze (using data science tools)
Use
Save(any of DT components)
Evolve
Terminate

Although not mandatory, maintaining a matching code structure facilitates DT creation and management within the DTaaS platform. The following structure is recommended:

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

A dedicated program exists for each phase of the 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 approach for implementing lifecycle phases within DTaaS is to create scripts. These scripts can be implemented as shell scripts.

Example Lifecycle Scripts

The following example programs/scripts demonstrate management of three phases in the lifecycle of the **mass-spring-damper DT**.

lifecycle/execute

```bash
#!/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 phase utilizes the DT configuration, FMU models, and Maestro tool to execute the digital twin. The script also stores the output of co-simulation in `/workspace/data/mass-spring-damper/output`.

A DT may not support a specific lifecycle phase. This intention can be expressed with an empty script and a helpful message if deemed necessary.

lifecycle/analyze

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

The lifecycle programs can invoke other programs in the codebase. In the case of the `lifecycle/terminate` program, it calls another script to perform the necessary operations.

lifecycle/terminate

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

References

[1]: Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." Simulation 101.3 (2025): 287-311.

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
```

Warning

The assets being displayed on the Library preview page come from the `master` branch of the backing GitLab project. Please create a branch named `master` and make it the default branch. This must be done for all the user repositories including the common repository.

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

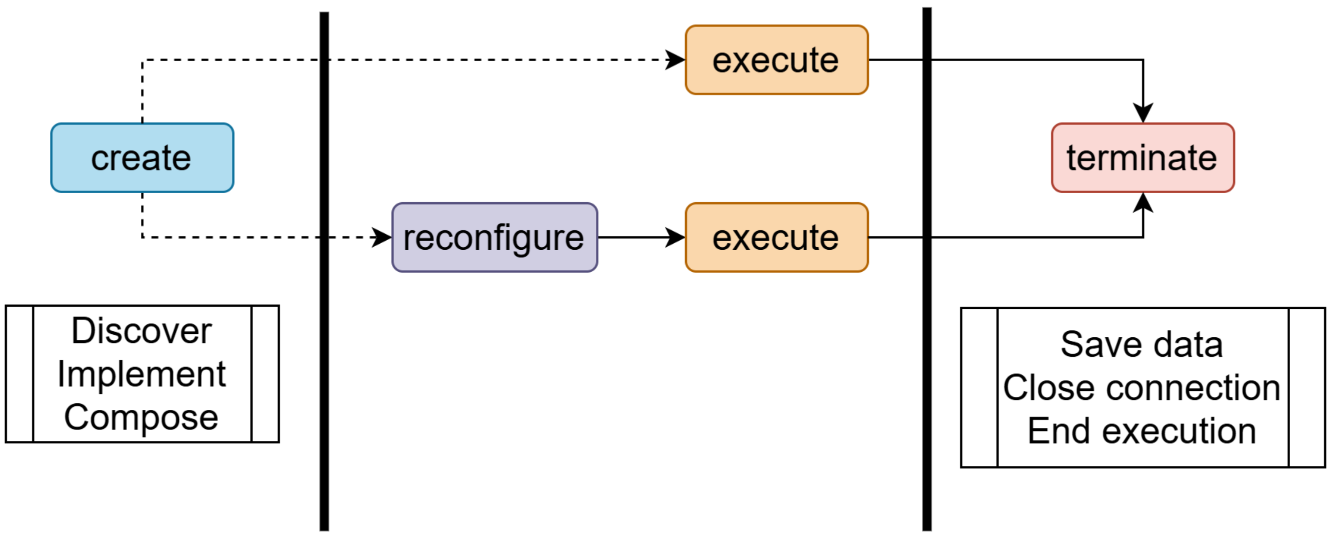
DevOps has been a well established software development practice. We are bringing out an experimental feature of integration DevOps in the DTaaS.

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.

DT LIFECYCLE

The DT preview implements the **Create**, **Manage** and **Execute** stages of a DT lifecycle. The suggested sequence of use for different lifecycle stages are:

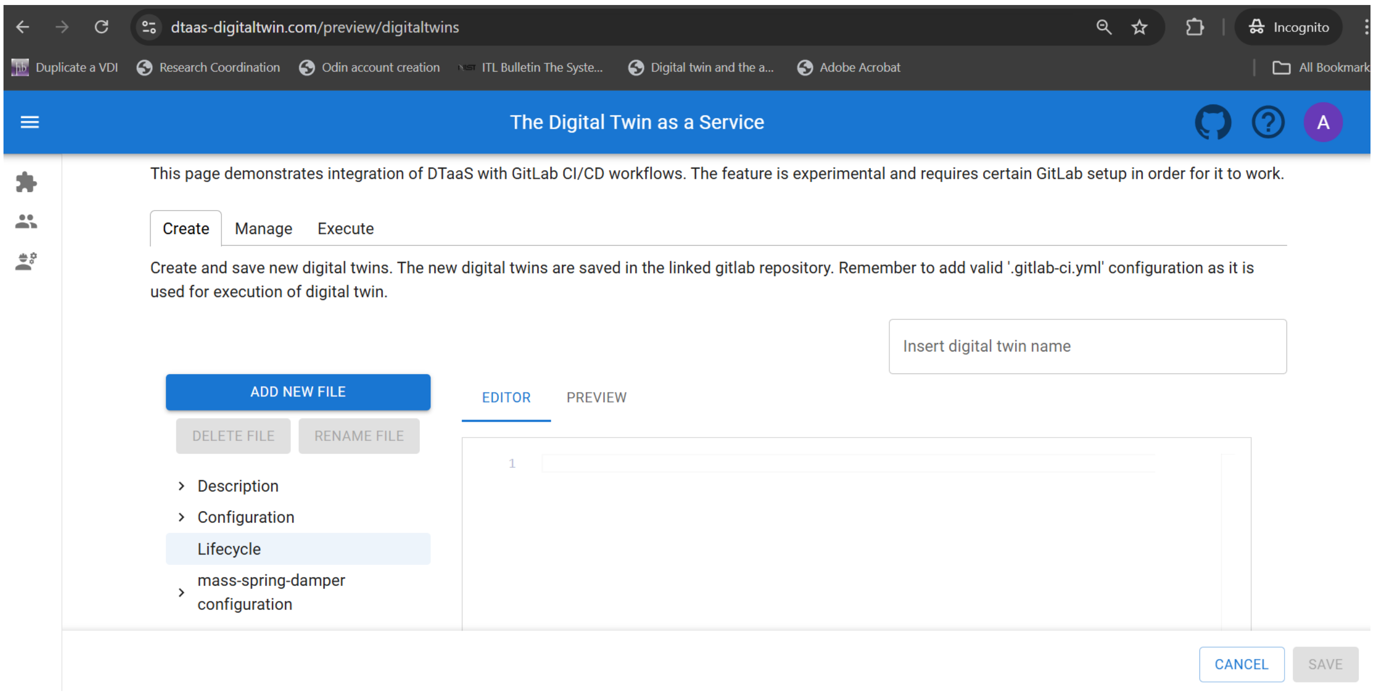


.

There are dedicated tabs for **Create**, **Manage** and **Execute** stages. The selection of DT assets for Create stage happens via Library preview page. The Manage tab fulfills reconfigure feature. The Execute and Terminate are managed on the Execute tab.

CREATE TAB

The users select reusable DT assets and arrive on the Create tab. The following figure shows DT creation page after selecting *mass-spring-damper* as a reusable asset for the new DT.


This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.

Create and save new digital twins. The new digital twins are saved in the linked gitlab repository. Remember to add valid '.gitlab-ci.yml' configuration as it is used for execution of digital twin.

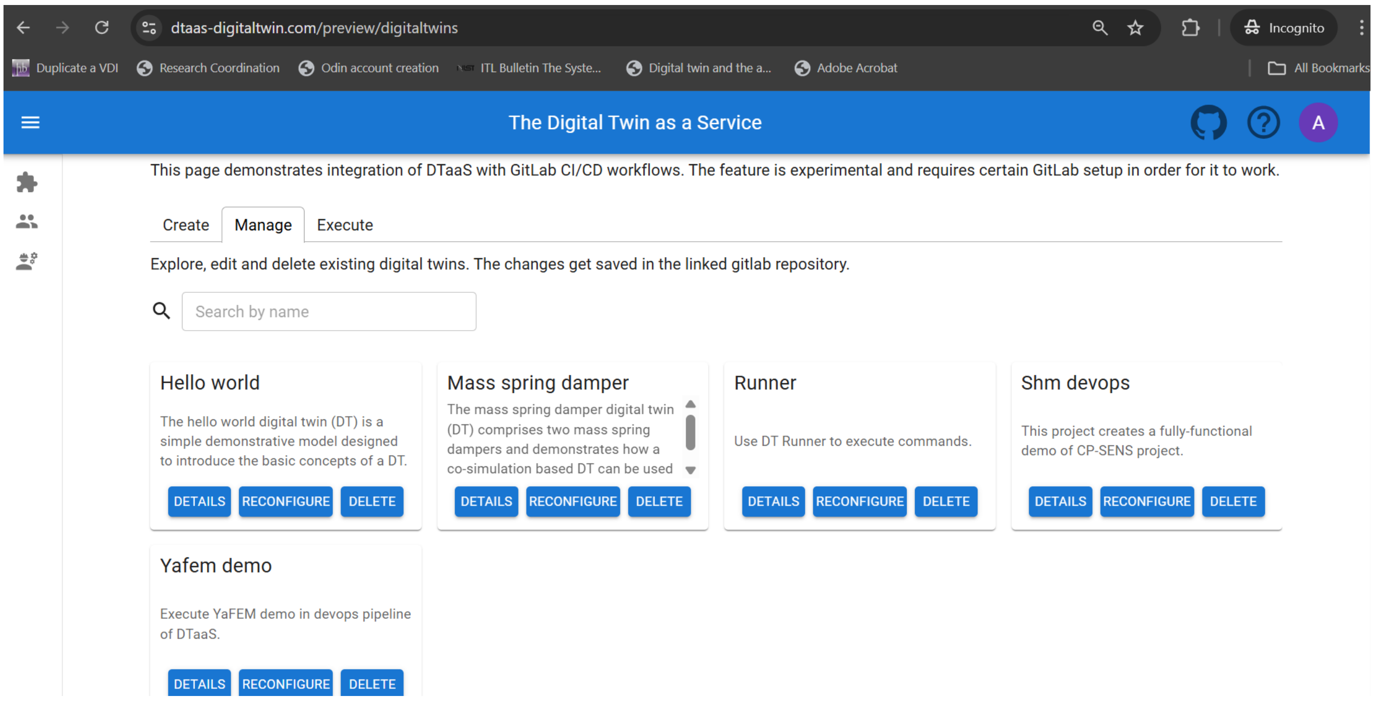
The left-side menu shows the possibility of creating necessary structure and elements for a new DT. Each reusable asset selected for this new DT appears on the left menu. Its configuration can be updated as well.

These files on the left menu correspond to three categories.

- **Description**: contains *README.md* providing comprehensive description of DT and *description.md* providing a brief description. The brief description is shown in the DT tabs and clicking on the *Details* button shows the complete README.md The *Details* button is available only on the Manage page.
- **Configuration**: Contains a *.gitlab-ci.yaml* for running the required lifecycle scripts and operations of the DT. Additional json and yaml files can be added to create configuration for a new DT.
- **Lifecycle**: These are the DT lifecycle scripts.

The *Add New File* button can be used to add new files in all the three categories. Finally, click on *SAVE* button to save the new DT. Newly created DTs become immediately available on the **Manage** and **Execute** tabs.

MANAGE TAB


dtaas-digitaltwin.com/preview/digitaltwins

The Digital Twin as a Service

This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.

Create | Manage | Execute

Explore, edit and delete existing digital twins. The changes get saved in the linked gitlab repository.

Search by name

Hello world
The hello world digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.
DETAILS RECONFIGURE DELETE

Mass spring damper
The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used
DETAILS RECONFIGURE DELETE

Runner
Use DT Runner to execute commands.
DETAILS RECONFIGURE DELETE

Shm devops
This project creates a fully-functional demo of CP-SENS project.
DETAILS RECONFIGURE DELETE

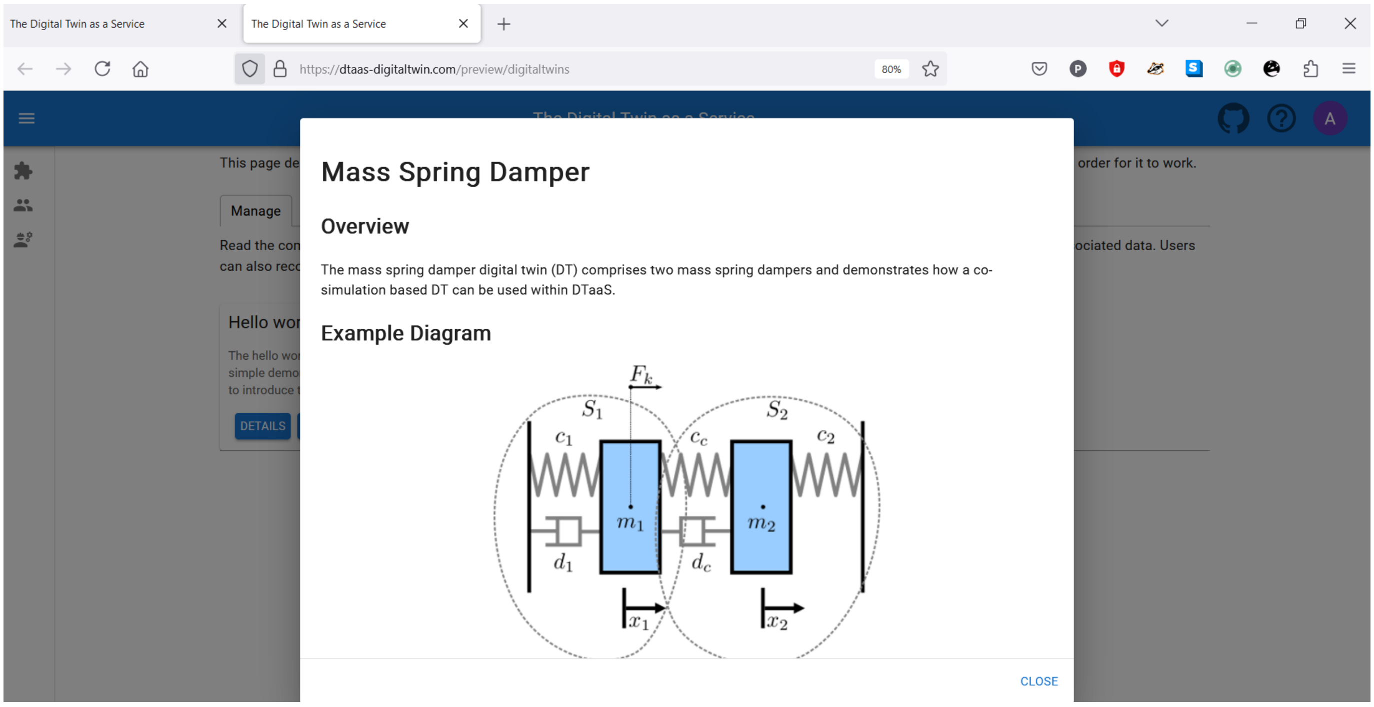
Yafem demo
Execute YaFEM demo in devops pipeline of DTaaS.
DETAILS RECONFIGURE DELETE

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.


https://dtaas-digitaltwin.com/preview/digitaltwins

Mass Spring Damper

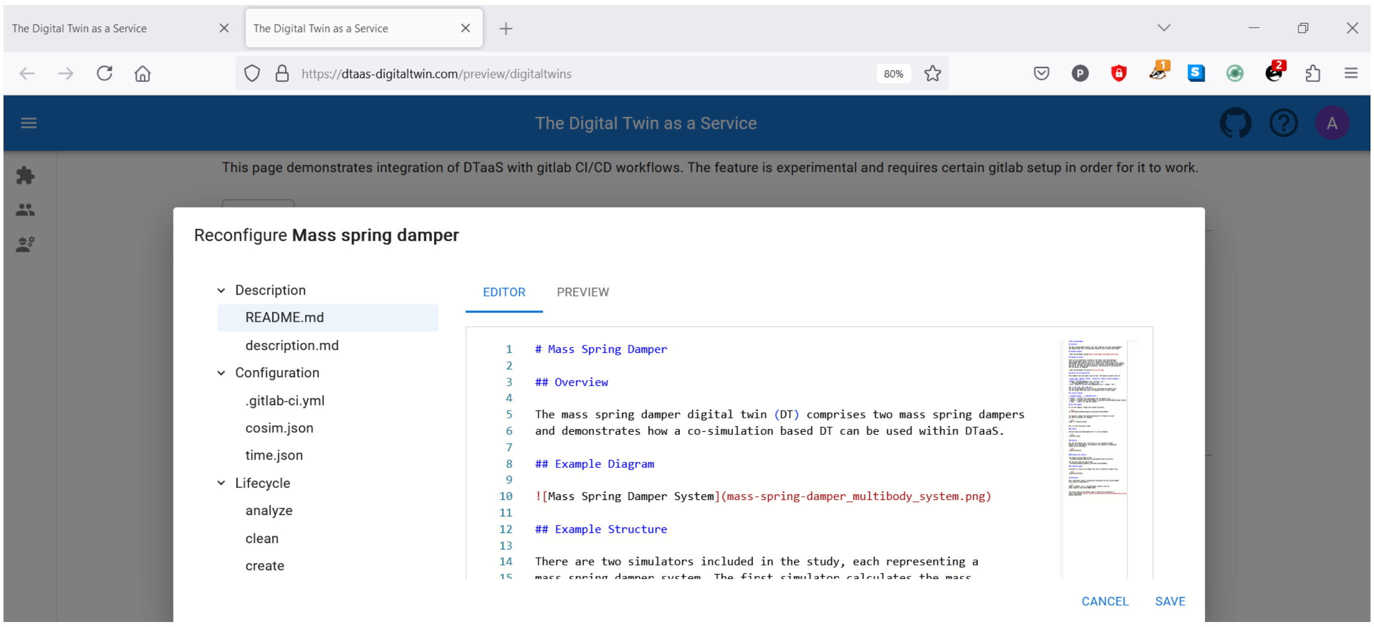
Overview

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

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**

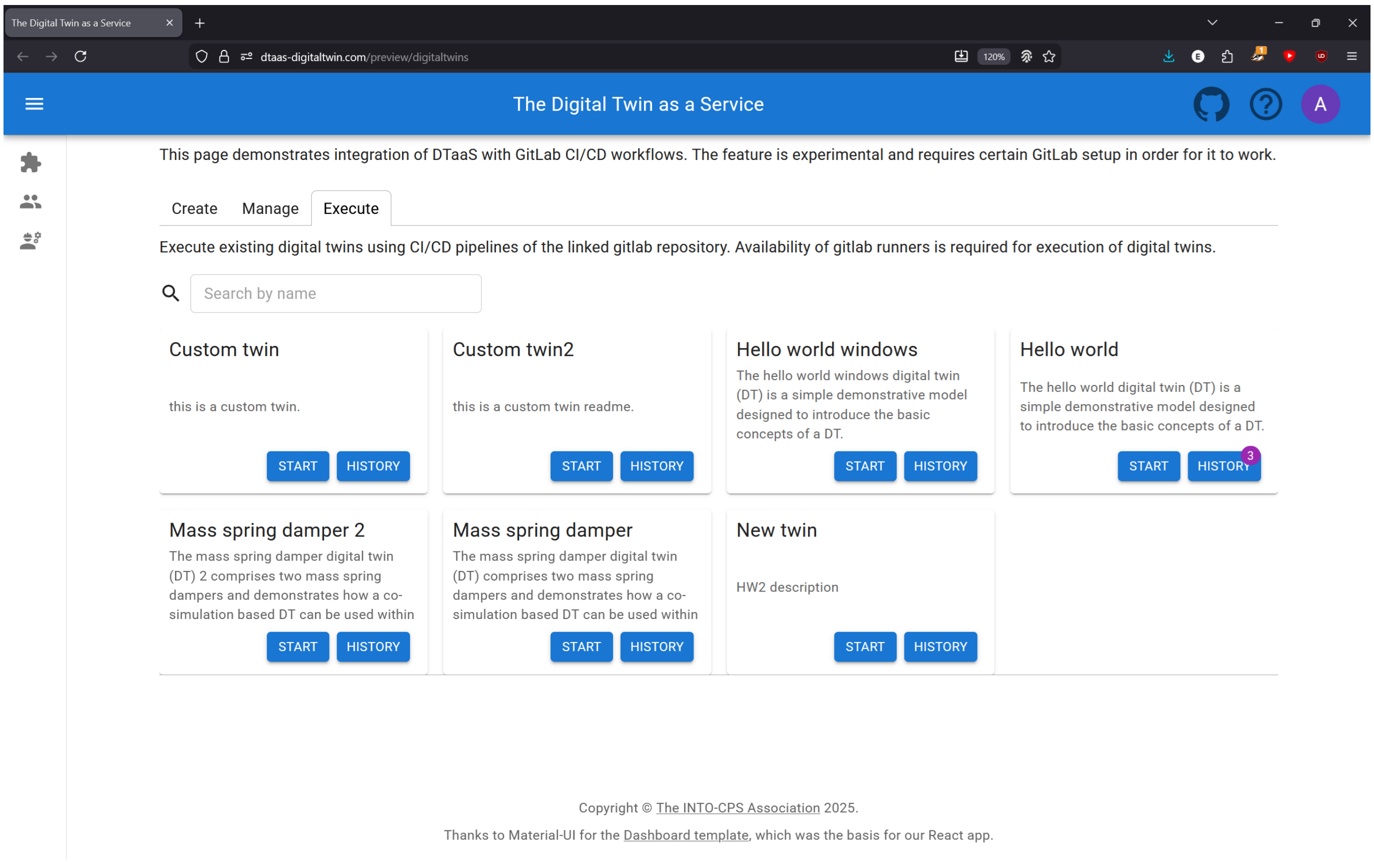


This page demonstrates integration of DTaaS with gitlab CI/CD workflows. The feature is experimental and requires certain gitlab setup in order for it to work.

Reconfigure **Mass spring damper**

```
# 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

![Mass Spring Damper System](mass-spring-damper_multibody_system.png)

## Example Structure

There are two simulators included in the study, each representing a
```

EXECUTE TAB


The Digital Twin as a Service
dtaas-digitaltwin.com/preview/digitaltwins
The Digital Twin as a Service
Create
Manage
Execute
Search by name
START
HISTORY

This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.

Execute existing digital twins using CI/CD pipelines of the linked gitlab repository. Availability of gitlab runners is required for execution of digital twins.

Custom twin

this is a custom twin.

Custom twin2

this is a custom twin readme.

Hello world windows

The hello world windows digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

Hello world

The hello world digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

Mass spring damper 2

The mass spring damper digital twin (DT) 2 comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within

Mass spring damper

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

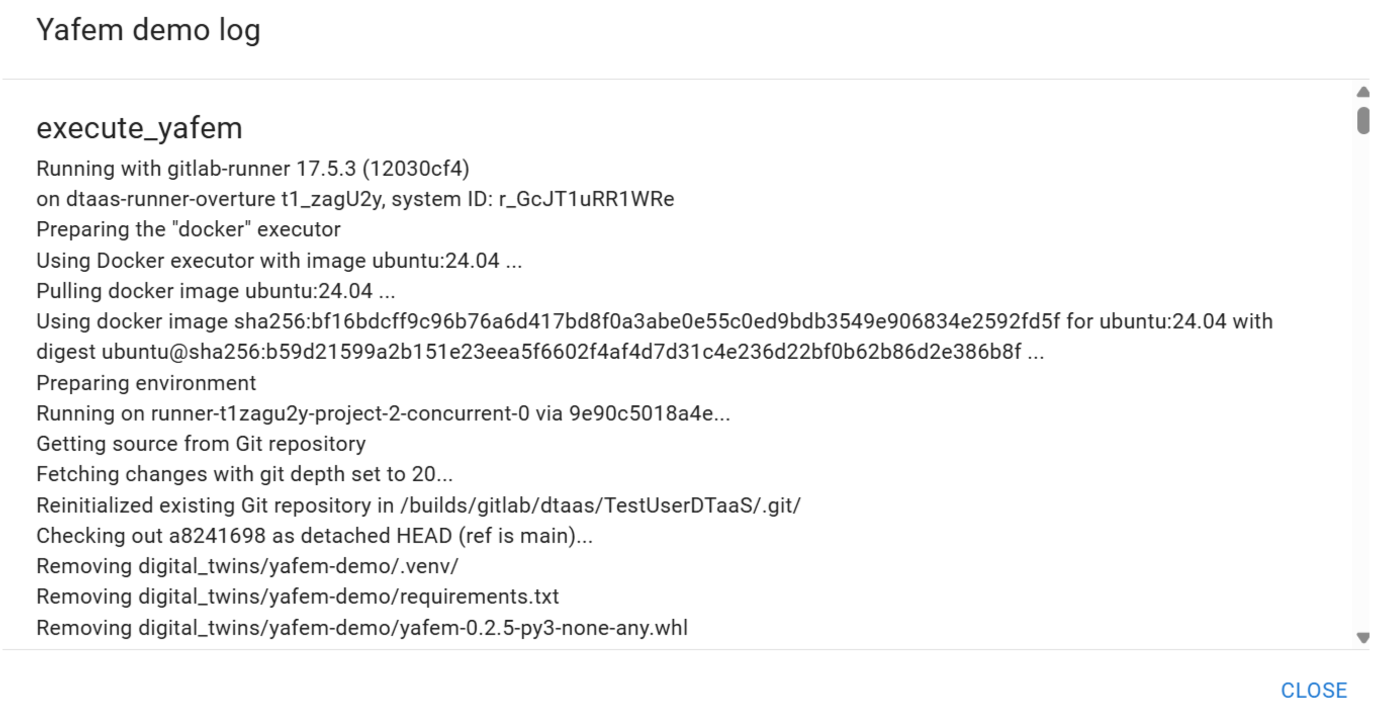
New twin

HW2 description

Copyright © The INTO-CPS Association 2025.

Thanks to Material-UI for the Dashboard template, which was the basis for our React app.

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.



```
Running with gitlab-runner 17.5.3 (12030cf4)
on dtaas-runner-overture t1_zagU2y, system ID: r_GcJT1uRR1WRe
Preparing the "docker" executor
Using Docker executor with image ubuntu:24.04 ...
Pulling docker image ubuntu:24.04 ...
Using docker image sha256:bf16bdcff9c96b76a6d417bd8f0a3abe0e55c0ed9bdb3549e906834e2592fd5f for ubuntu:24.04 with
digest ubuntu@sha256:b59d21599a2b151e23eea5f6602f4af4d7d31c4e236d22bf0b62b86d2e386b8f ...
Preparing environment
Running on runner-t1zagu2y-project-2-concurrent-0 via 9e90c5018a4e...
Getting source from Git repository
Fetching changes with git depth set to 20...
Reinitialized existing Git repository in /builds/gitlab/dtaas/TestUserDTaaS/.git/
Checking out a8241698 as detached HEAD (ref is main)...
Removing digital_twins/yafem-demo/.venv/
Removing digital_twins/yafem-demo/requirements.txt
Removing digital_twins/yafem-demo/yafem-0.2.5-py3-none-any.whl
```

Setting Allowed Values

Some Setting values will cause problems if used. In this document we will have an in-depth look at the values that are allowed and values that are not. We will also see some of the expected errors and finally go over some troubleshooting steps for this page.

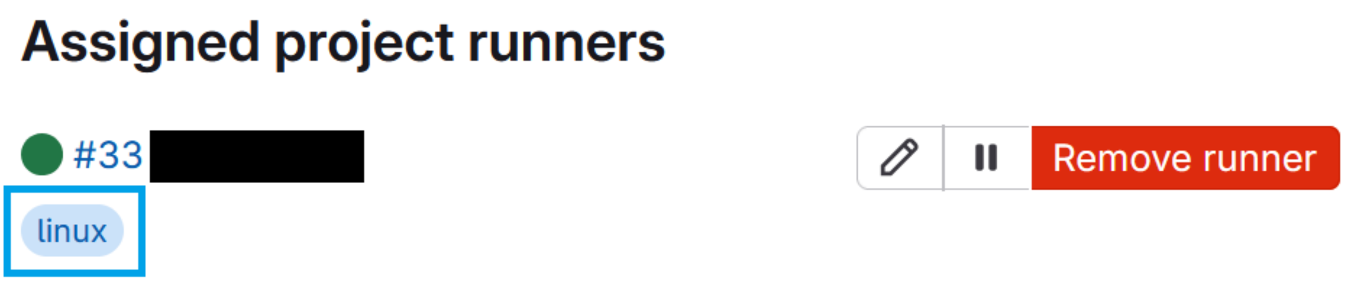
If you want to understand what each parameter does, please read the settings document.

Note: You must have the appropriate access rights to all the resources that you connect to the application.

RUNNER TAG

- Allowed values

This parameter has to be a text string like "linux", "ubuntu", "2" and match some Runner's tags on the execution service:



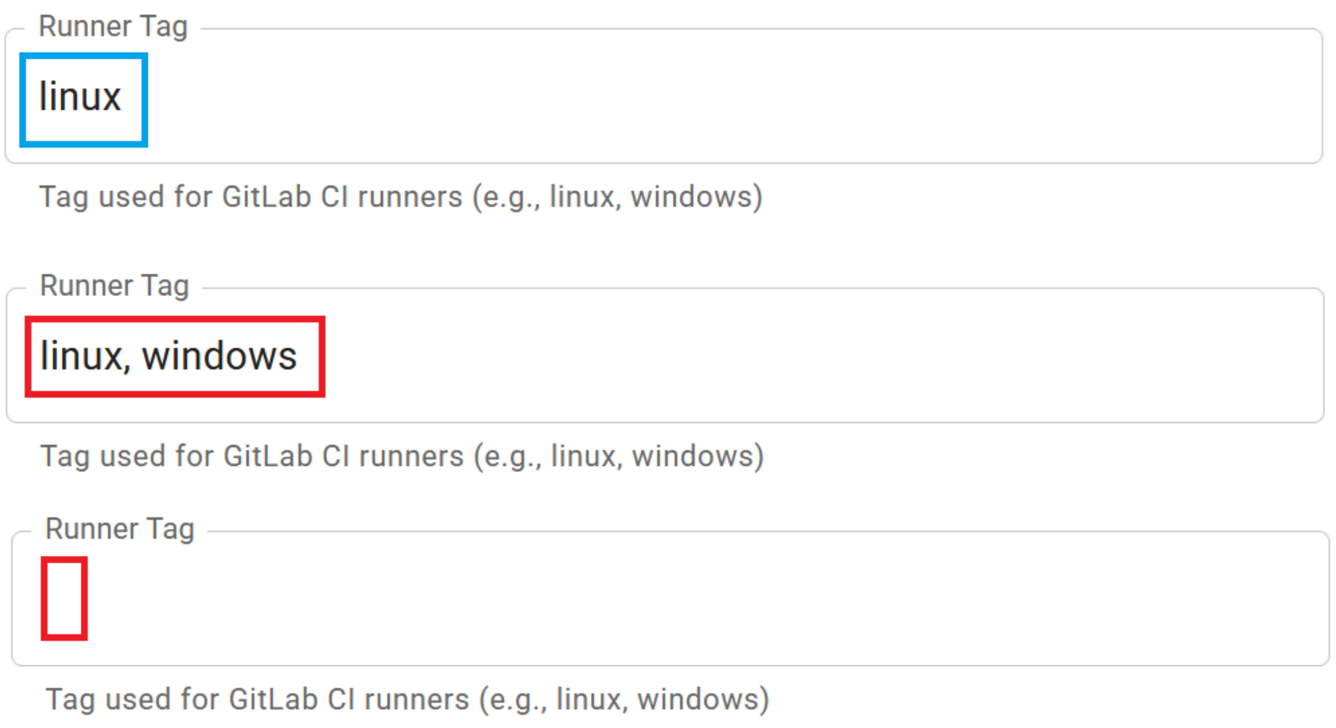
Multiple values, like "linux, windows" is not supported and will be treated as one tag.

- Not allowed values

If the **Runner Tag** field *doesn't* match a Runner, no jobs will be picked up and the job will hence timeout. If your twins unexpectedly timeout, check that you have spelled the tag correctly.

No tag (i.e. a blank field) is permitted by some services like GitLab, meaning that *any* Runner that is configured to pick up tag-less jobs can pick these jobs up. A limitation of DTaaS is that we require *some* tag. No tags, like other unallowed tags will be ignored.

- Visual examples



Runner Tag: linux

Tag used for GitLab CI runners (e.g., linux, windows)

Runner Tag: linux, windows

Tag used for GitLab CI runners (e.g., linux, windows)

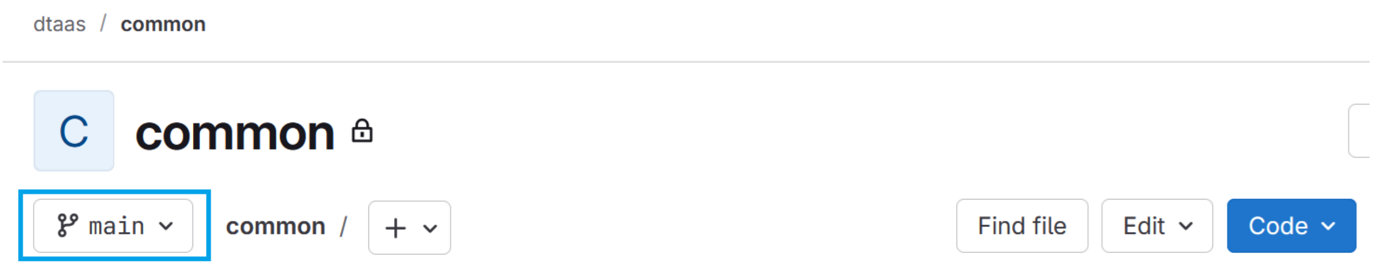
Runner Tag:

Tag used for GitLab CI runners (e.g., linux, windows)

BRANCH

- Allowed values

The **Branch** field must be a text string matching a branch in the user *and* common library repositories. For example: "main" or "master":



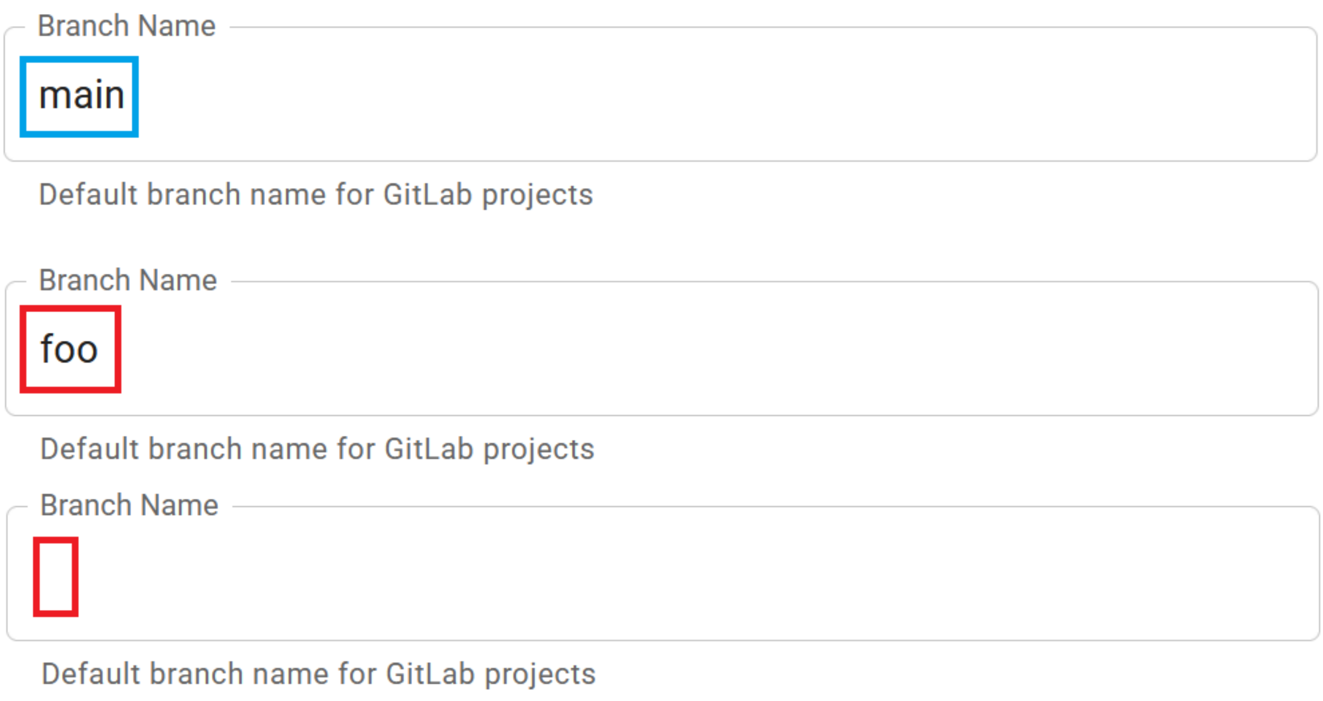
- Not allowed values

You should not leave this field empty or not matching some branch in both repositories. While technically allowed, this may cause errors or unexpected behaviours.

Expected error:

An error occurred while fetching assets: GitbeakerRequestError: 404 Tree Not Found

- Visual examples


Branch Name
main
Default branch name for GitLab projects
Branch Name
foo
Default branch name for GitLab projects
Branch Name
Default branch name for GitLab projects

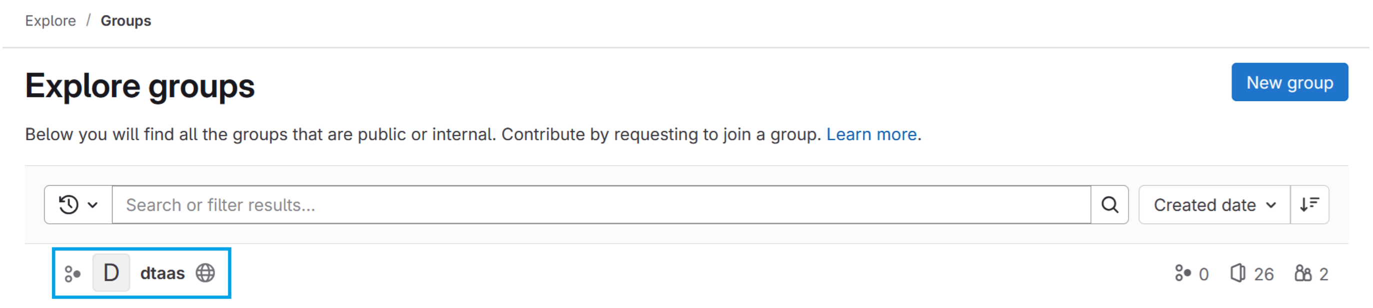
GROUP NAME

- Allowed values

The **Group Name** field again requires a text string, and it must further match some group in the storage service (e.g. GitLab).

Example:

DTaaS


Explore / Groups

Explore groups

Below you will find all the groups that are public or internal. Contribute by requesting to join a group. Learn more.

Group names are case insensitive on GitLab, but we recommend matching the case exactly for consistency and to remove it as a source of error.

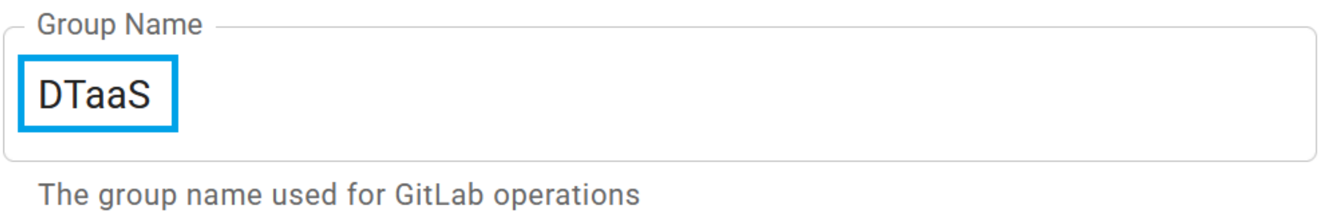
- Not allowed values

If the group doesn't exist or the field is left blank, you will experience errors upon accessing Digital Twins, etc. Make sure that you write something here.

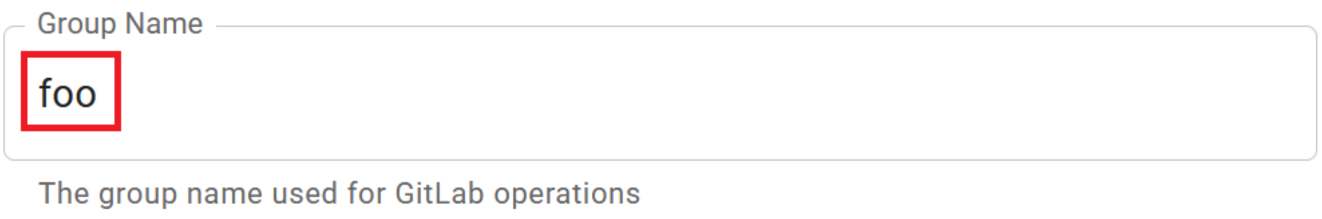
Expected error:

An error occurred while fetching assets: GitbeakerRequestError: 404 Group Not Found

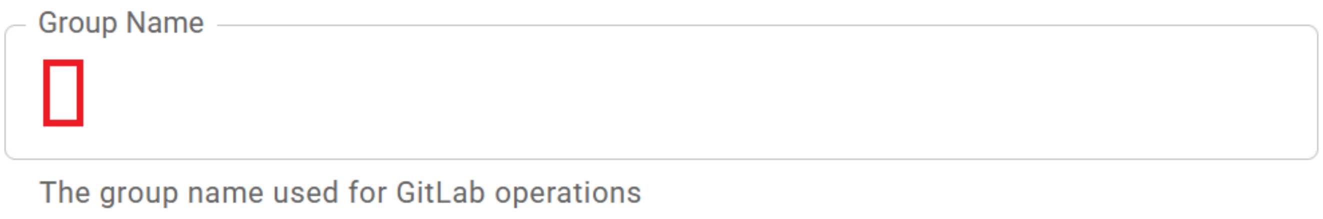
- Visual examples


Group Name: DTaaS

The group name used for GitLab operations


Group Name: foo

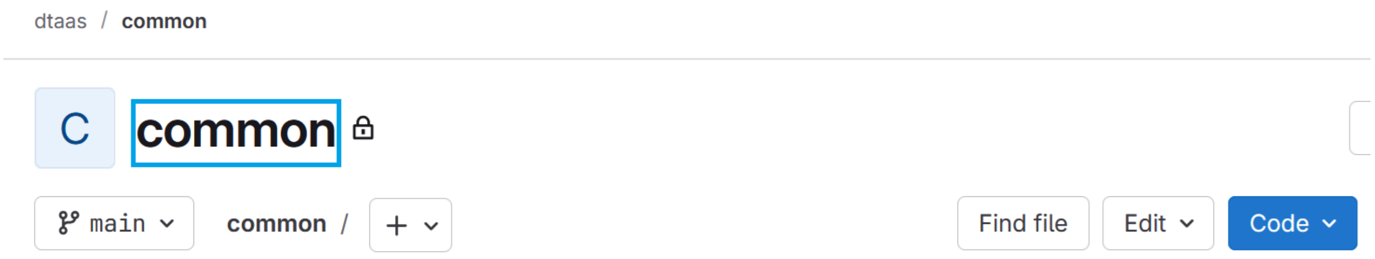
The group name used for GitLab operations


Group Name:

The group name used for GitLab operations

COMMON LIBRARY PROJECT NAME

- Allowed values The **Common Library Project name** parameter is a text string. It must correlate to the repository responsible for keeping shared resources. Examples: "common" and "library":



Failure to match a repository will break the DevOps page.

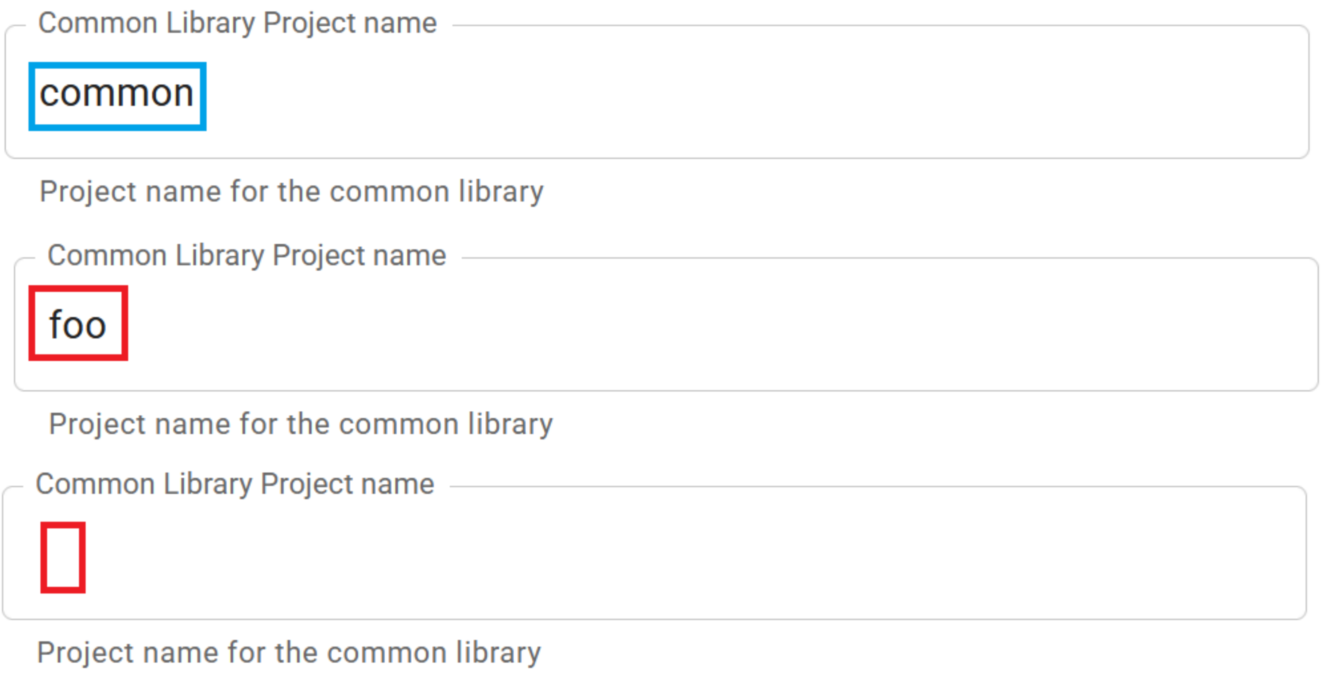
Expected error:

An error occurred while fetching assets: Error: Common project not found

- Not allowed values

You should not leave this field blank or have it not match some repository. It is inadvisable to have it match a user repository.

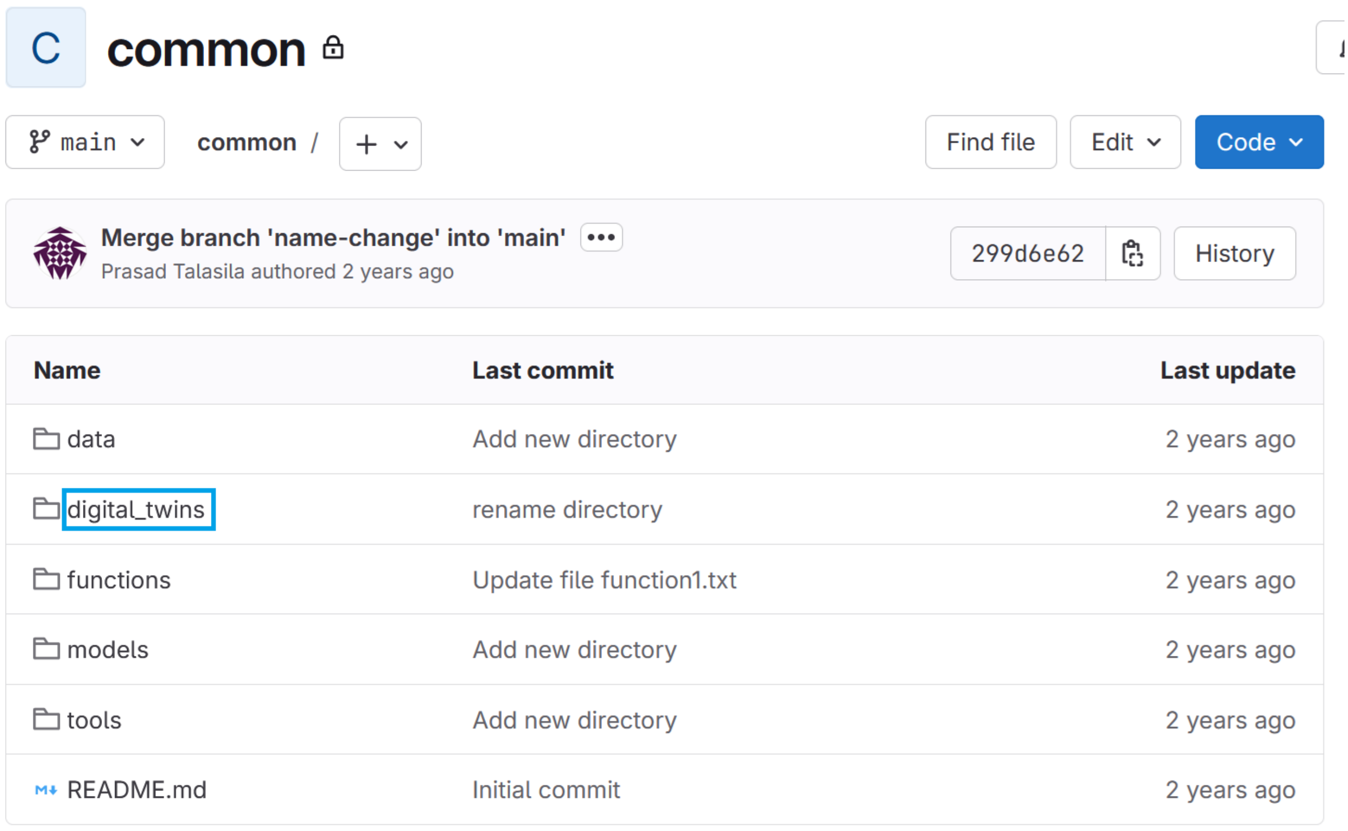
- Visual examples


Common Library Project name
common
Project name for the common library
Common Library Project name
foo
Project name for the common library
Common Library Project name
Project name for the common library

DT DIRECTORY

- Allowed values

Like the other fields, this must be a text string. It should match the name of the folder within the common library and most importantly the user repository where the Digital Twins are stored.

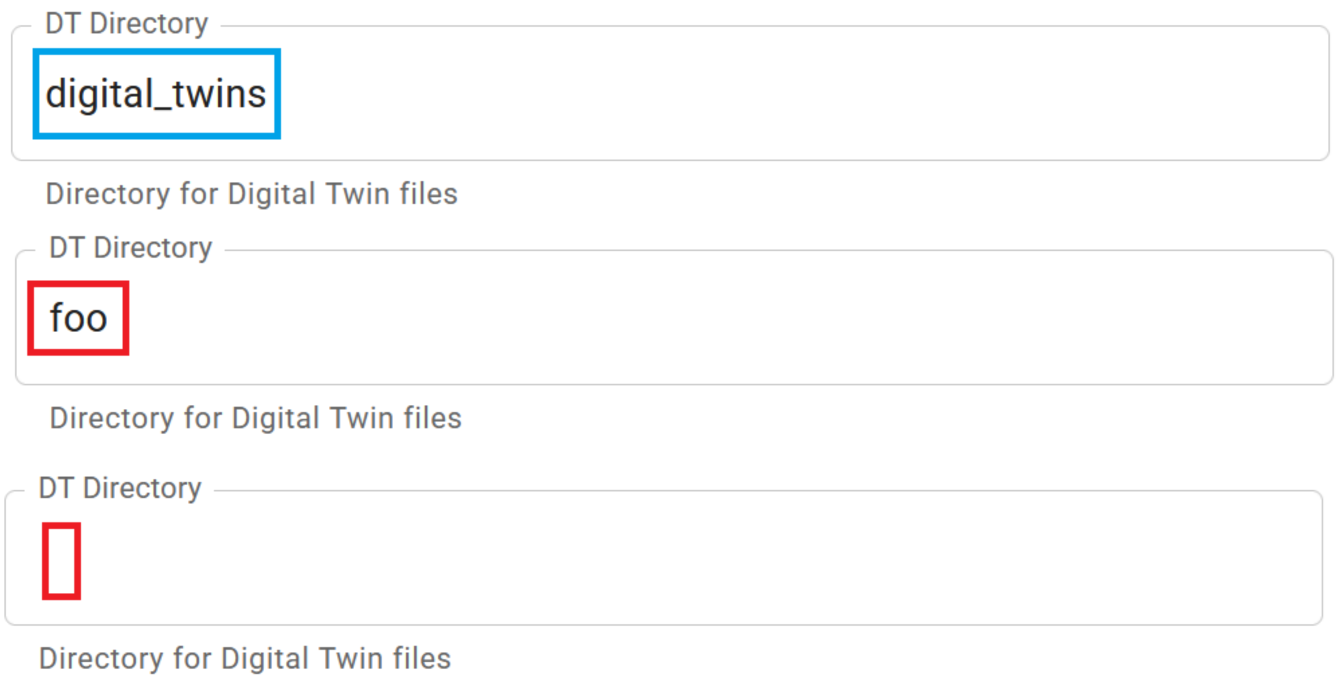


Merge branch 'name-change' into 'main'
Prasad Talasila authored 2 years ago
299d6e62

Name	Last commit	Last update
data	Add new directory	2 years ago
digital_twins	rename directory	2 years ago
functions	Update file function1.txt	2 years ago
models	Add new directory	2 years ago
tools	Add new directory	2 years ago
README.md	Initial commit	2 years ago

• Not allowed values

Do not leave this blank or erroneously mapped. This will lead to silent failure with the Digital Twins not loading or Digital Twins from a previous set up to show up.


DT Directory
digital_twins
Directory for Digital Twin files
DT Directory
foo
Directory for Digital Twin files
DT Directory
Directory for Digital Twin files

TROUBLESHOOTING

The Digital Twins are not showing up:

Verify that Group, Branch, common and DT directory all are configured exactly. like it is on the backend.

The Digital Twins are timing out:

Verify that the Runner Tag parameter is not a list of tags, that the runner is active and matches its tag.

SUMMARY

We have specified the possible values to correctly connect DTaaS with your storage and service instances and pitfalls. The key insight is not to leave any of the fields blank, having them be of the text string format and properly corresponding to your services and runners. Some troubleshooting steps have further been provided to overcome common hurdles.

Capabilities

Other than creating and editing Digital Twins, you are naturally also able to run them and adjust their execution parameters. The DTaaS allows you to run multiple Digital Twins at the same time and even change settings while they are running, without needing to worry about manually having to fetch your data: They will load when you return to whichever branch and group you ran them on. The settings include both repository and execution options, while all twins can easily be run and checked from the Execution tab under the Preview page.

Read more about Settings, Setting Values and Concurrent Execution on their respective pages.

Now we will go through the specific capabilities of each of these features.

⚙️ SETTINGS

The table below describes the different test setups (valid, invalid, etc.), the resultant behavior and what we deemed to be expected behavior across the features. Following every table, there is a summary and list of potential problems.

| Setting | Expected behaviour | Observed behaviour | Test method | | Runner tag **valid** | Job is picked up by runner with relevant tag and completes without error. | ✅ Same as expected. | Goto Preview page. Go to Account. Change Runner Tag. Go back one page. Execute Hello world twin. Verify runner name based on tag (repeat with 2nd runner) | | Runner tag **invalid** | Job is never picked up. Runner times out after 10 minutes. | ✅ Same as expected. | Change Runner Tag to "foo" (inexistent). Run Hello World twin. | | Runner Tag **no value** | Configuration saves. Running a twin succeeds if appropriate runner exists. | ❌ Not picked up, times out. | Set `run_untagged = false` in local gitlab runner config. Mark "Run untagged jobs" as true in gitlab instance (https://dtaas-digitaltwin.com/gitlab). Change Runner Tag to the empty string on application. Run Hello world twin. | | Branch **valid** | Job runs with the correct branch and completes without error. | ✅ Same as expected. | Make new branch in GitLab instance project. Change Branch name to "master-2". Execute Hello world twin. Verify ref matches branch in execution log. | | Branch **invalid** | Execution tab gracefully displays no twins as branch doesn't exist. | ❌ IF twins are not cached: Throws an error displayed to user: An error occurred while fetching assets: GitbeakerRequestError. IF twins are cached: Job fails. Snackbar says "Execution error for [twin name]". No log available in execution history. | Change Branch name to "master-1" (inexistent). IF twins are cached: Execute Hello world twin. | | Group name **valid** | The twin is runnable. | ✅ Same as expected. | Click "reset to defaults". Run twin. | | Group name **invalid** | Execution tab gracefully displays no twins as group doesn't exist. | ❌ Same as Branch invalid case. | Change group name to "Foo" (inexistent). IF twins are cached: Execute Hello world twin. | | Common name **valid** | Library twins are visible. | ✅ Same as expected. *Private twins also show up in Common twins. | Click "reset to defaults". Goto library. Goto common twins. Inspect twin visibility. | | Common name invalid | Library twins gracefully not visible. | ❌ Displays error: An error occurred while fetching assets: Error: Common project foo not found | Change Common Library Project name to "Foo" (inexistent). Goto common twins. Inspect visibility. | | DT directory valid | Twins are visible. | ✅ Same as expected. | Click "reset to defaults". Goto library. Goto Execution tab. Inspect twin visibility. | | DT directory name invalid | Twins gracefully not visible. | ❌ Depends on the cache. Displays error if there is none: An error occurred while fetching assets: GitbeakerRequestError | Change Common DT directory name to "Foo" (inexistent). Goto Execution tab. Inspect twin visibility. | | Group Name, DT Directory, Common Library Project Name, Branch name no value | Invalid value caught early. Can't save, appropriate feedback of invalid form fill out. | ❌ Same as Branch invalid case. | Change Group Name, DT Directory, Common Library Project Name, Branch name to the empty string. | |*

-- | -- |

--
|

--

| | Runner tag valid | Job is picked up by runner with relevant tag and completes without error. | ✅ Same as expected. | Goto Preview page. Go to Account. Change Runner Tag. Go back one page. Execute Hello world twin. Verify runner name based on tag (repeat with 2nd runner) | | Runner tag invalid | Job is never picked up. Runner times out after 10 minutes. | ✅ Same as expected. | Change Runner Tag to "foo" (inexistent). Run Hello World twin. | | Runner Tag no value | Configuration saves. Running a twin succeeds if appropriate runner exists. | ❌ Not picked up, times out. | Set `run_untagged = false` in local gitlab runner config. Mark "Run untagged jobs" as true in gitlab instance (https://dtaas-digitaltwin.com/gitlab). Change Runner Tag to the empty string on application. Run Hello world twin. | | Branch valid | Job runs with the correct branch and completes without error. | ✅ Same as expected. | Make new branch in GitLab instance project. Change Branch name to "master-2". Execute Hello world twin. Verify ref matches branch in execution log. | | Branch invalid | Execution tab gracefully displays no twins as branch doesn't exist. | ❌ IF twins are not cached: Throws an error displayed to user: An error occurred while fetching assets: GitbeakerRequestError. IF twins are cached: Job fails. Snackbar says "Execution error for [twin name]". No log available in execution history. | Change Branch name to "master-1" (inexistent). IF twins are cached: Execute Hello world twin. | | Group name valid | The twin is runnable. | ✅ Same as expected. | Click "reset to defaults". Run twin. | | Group name invalid | Execution tab gracefully displays no twins as group doesn't exist. | ❌ Same as Branch invalid case. | Change group name to "Foo" (inexistent). IF twins are cached: Execute Hello world twin. | | Common name valid | Library twins are visible. | ✅ Same as expected. Private twins also show up in Common twins. |

Click "reset to defaults". Goto library. Goto common twins. Inspect twin visibility. | | Common name **invalid** | Library twins gracefully not visible. | ❌ Displays error: An error occurred while fetching assets: Error: Common project foo not found | Change Common Library Project name to "Foo" (inexistent). Goto common twins. Inspect visibility. | | DT directory **valid** | Twins are visible. | ✅ Same as expected. | Click "reset to defaults". Goto library. Goto Execution tab. Inspect twin visibility. | | DT directory name **invalid** | Twins gracefully not visible. | ❌ Depends on the cache. Displays error if there is none: An error occurred while fetching assets: GitbeakerRequestError | Change Common DT directory name to "Foo" (inexistent). Goto Execution tab. Inspect twin visibility. | | Group Name, DT Directory, Common Library Project Name, Branch name **no value** | Invalid value caught early. Can't save, appropriate feedback of invalid form fill out. | ❌ Same as Branch invalid case. | Change Group Name, DT Directory, Common Library Project Name, Branch name to the empty string. |

Summary: Changing to valid settings works. Invalid settings in the best case display an error in the HTML (no cache) and otherwise shows stale twins.

Problems:

- Stale state maintained after settings are updated unless page is refreshed.
- Displaying technical errors to the user when the settings are invalid.
- Permitting invalid values such as empty strings when they are required
- No tags runners may not work, but possibly local problem.
- Private twins show up as common twins

⏳ CONCURRENT EXECUTION

Expected behaviour	Observed behaviour	Test method
Both twins run successfully simultaneously.	✅ Same as expected.	Run the same twin twice at the same time.
All twins run successfully simultaneously.	✅ Same as expected.	Run different twins at the same time.

Concurrent Execution Across Different Runners

Expected behaviour	Observed behaviour	Test method
Both twins run successfully simultaneously. They report the correct runner name in the Execution logs.	✅ Same as expected.	Set up 2 ubuntu GitLab runners with different tags. Change Runner Tag to first runner's tag. Execute twin. Change Runner Tag to second runner tag. Execute another twin.
History stays after editing, it is still executing.	✅ Same as expected.	Execute → Edit settings → Execute. Check – is history still there? Does it look correct?

Summary:

All functionality works according to the tests. The logs are still there after changing settings and running any combination of twins and runners work (Only Ubuntu runners tested).

🧩 IMPLEMENTING BACKENDS

The DTaaS is by default set up to work with GitLab as execution and storage backend, but other combinations may fit your needs better. As such, the code base is designed with this flexibility in mind, so you don't have to reimplement everything every time you need a new backend. In future versions of the DTaaS, more backends may be available out of the box, like GitHub and Azure.

💭 SUMMARY

Digital Twins can be queued as needed and the live settings you make during these will not interfere with data retention, while empowering you to test multiple setups at once in one place. For greater flexibility, those with the technical know-how can frictionlessly expand upon the available backends.

Running Multiple Digital Twins at the Same Time

The DTaaS platform allows for executing multiple Digital Twins in tandem with one another. This can save hours of time when dealing with intensive Digital Twins or when you want to deploy them across different resources.

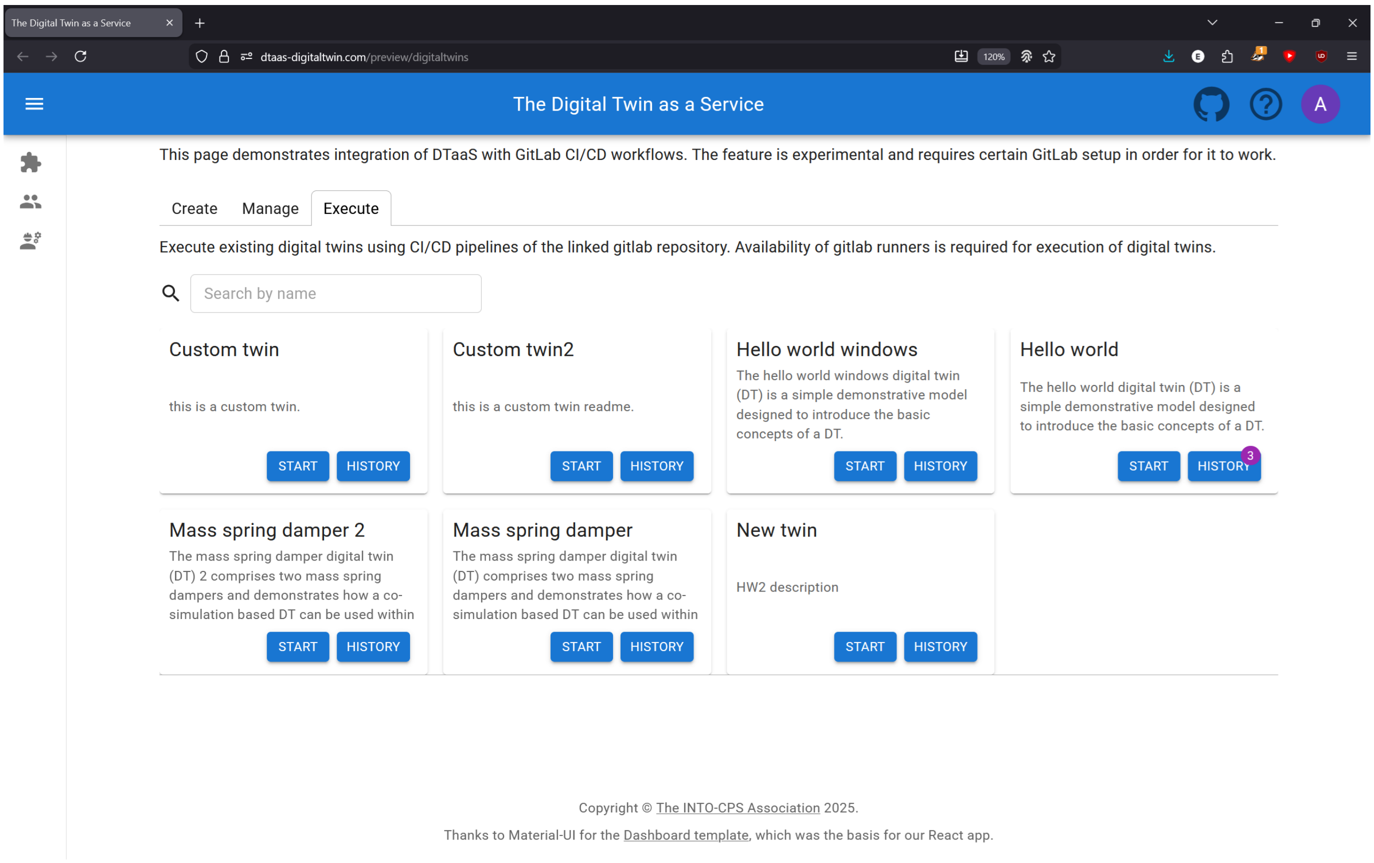
⏯️ RUNNING DIGITAL TWINS

When you want to deploy a Digital Twin you can do this from the *Execution tab* on the *Digital Twin Preview page*. You can find this by clicking on the Workbench icon in the sidebar on the left hand side and locating *Digital Twins Preview* among the other links. Click to open the *Digital Twin Preview page* in a new tab.

In the newly opened tab you will see a series of cards with a name, short description of the Digital Twins and **START** and **HISTORY** buttons. The **START** button sends a *job* to the backend, telling it to run the simulation associated with that Digital Twin, which is defined by its Life Cycle files. You can inspect and change these in the *Edit* tab.

Running Multiple Twins

Pressing the **START** button multiple times queues multiple simulations. You can also queue different kinds of Digital Twins.


The Digital Twin as a Service
dtaas-digitaltwin.com/preview/digitaltwins
The Digital Twin as a Service
Create
Manage
Execute
Search by name
START
HISTORY

This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.

Execute existing digital twins using CI/CD pipelines of the linked gitlab repository. Availability of gitlab runners is required for execution of digital twins.

Custom twin
this is a custom twin.

Custom twin2
this is a custom twin readme.

Hello world windows
The hello world windows digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

Hello world
The hello world digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

Mass spring damper 2
The mass spring damper digital twin (DT) 2 comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within

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

New twin
HW2 description

Copyright © The INTO-CPS Association 2025.
Thanks to Material-UI for the Dashboard template, which was the basis for our React app.

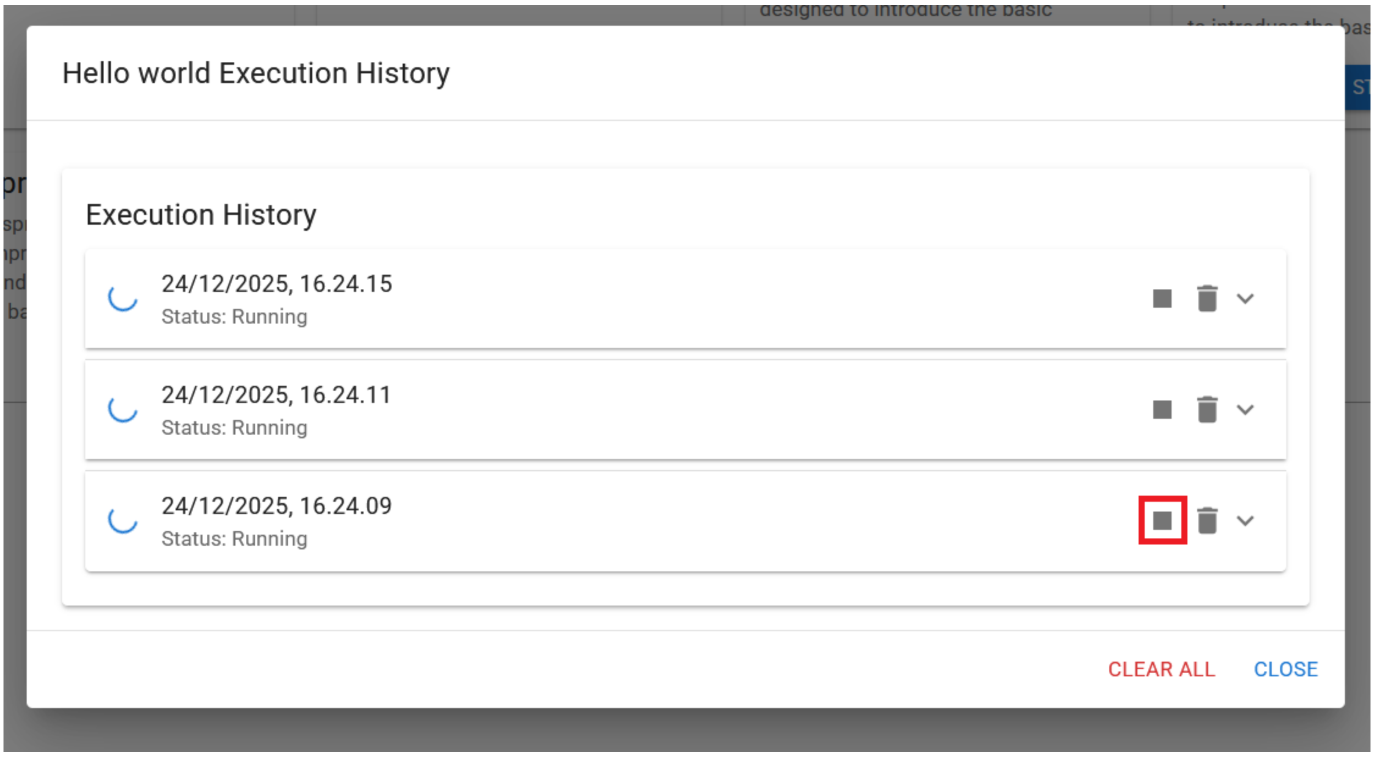
The deployment service (e.g. GitLab) automatically load balances across available runners, so you can set it and forget it.

🗃️ THE EXECUTION LOG

Pushing the **HISTORY** button provides an overview of all current and past simulation trials, distinguished by time of execution and status (Running, Failed, Succeeded or Timed Out). To check how a simulation went, you can click on an entry (or the arrow on the right of the entry) to expand it. Each step in the Life Cycle will then be presented.

If you want to delete an entry, click the trash icon. A verification box will reassure you that you are deleting the right one. You can also delete all entries of a Digital Twin with the **CLEAR ALL** button.

Click the stop symbol in the log to stop an execution:


Hello world Execution History

Execution History

24/12/2025, 16.24.15
Status: Running

24/12/2025, 16.24.11
Status: Running

24/12/2025, 16.24.09
Status: Running

🔧 CHANGING RUNNERS

You can change which runners pick up the jobs in the settings. Read more about this and other settings that are available.

💭 SUMMARY

You have now learned how to navigate the DevOps Execution page: execute multiple Digital Twins at the same time and inspect and delete their logs.

2.6 Working with GitLab

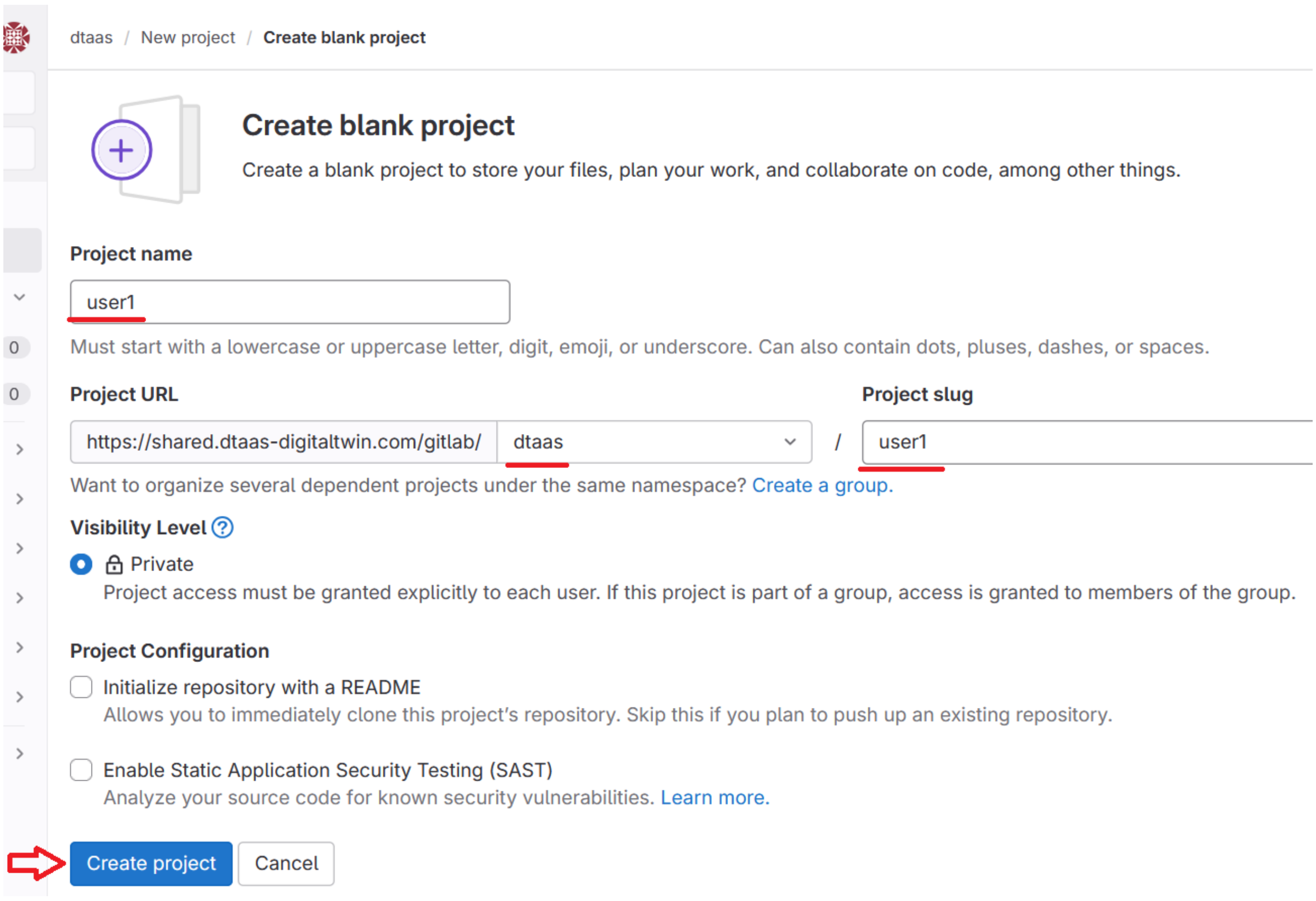
The DTaaS platform relies on GitLab for two purposes:

1. OAuth 2.0 authorization service
2. DevOps service

The admin documentation covers the OAuth 2.0 authorization configuration. This guide addresses the use of git commands and project structure for the GitLab DevOps service within the DTaaS.

2.6.1 Preparation

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


dtaas / New project / Create blank project

Create blank project

Create a blank project to store your files, plan your work, and collaborate on code, among other things.

Project name: user1

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL: https://shared.dtaas-digitaltwin.com/gitlab/ dtaas / Project slug: user1

Want to organize several dependent projects under the same namespace? Create a group.

Visibility Level: ● Private

Project access must be granted explicitly to each user. If this project is part of a group, access is granted to members of the group.

Project Configuration

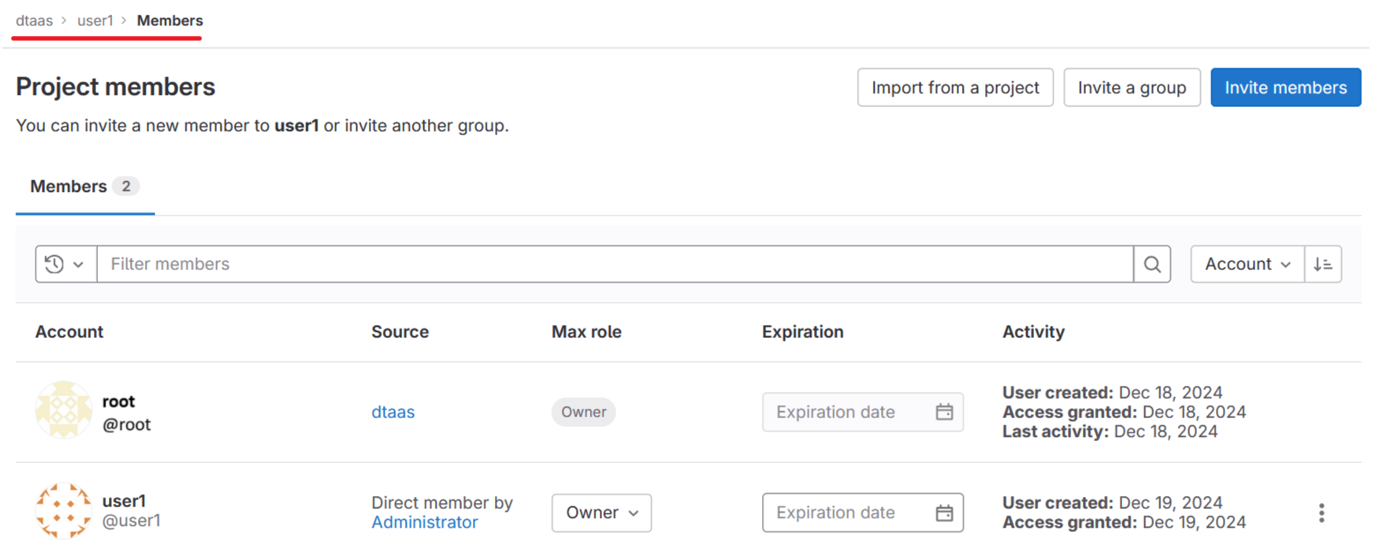
☐ Initialize repository with a README

Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

☐ Enable Static Application Security Testing (SAST)

Analyze your source code for known security vulnerabilities. Learn more.

The user must have ownership permissions over the project.


dtaas > user1 > Members

Project members

You can invite a new member to **user1** or invite another group.

Account	Source	Max role	Expiration	Activity
root @root	dtaas	Owner	Expiration date	**User created:** Dec 18, 2024 **Access granted:** Dec 18, 2024 **Last activity:** Dec 18, 2024
user1 @user1	Direct member by Administrator	Owner	Expiration date	**User created:** Dec 19, 2024 **Access granted:** Dec 19, 2024

Warning

The DTaaS website expects a default branch named `main` to exist. The website client performs all git operations on this branch. The preferred default branch name can be changed in the settings page.

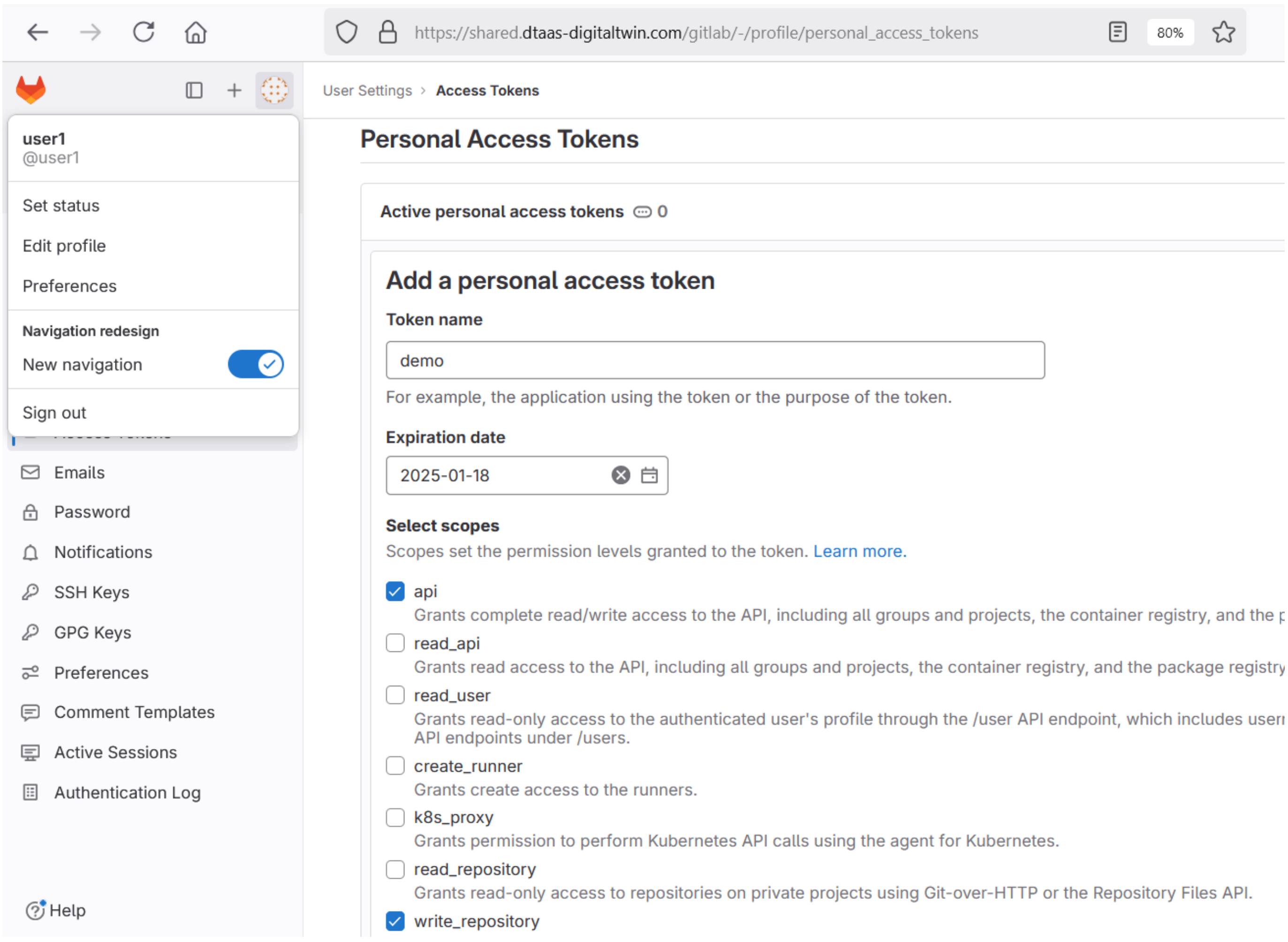
2.6.2 Git commands

Standard git commands and workflows should be utilized. There are two methods for using a GitLab project as a remote git server:

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

This tutorial demonstrates the use of PAT for working with the GitLab server.

The first step is to create a PAT.


https://shared.dtaas-digitaltwin.com/gitlab/-/profile/personal_access_tokens

User Settings › Access Tokens

Personal Access Tokens

Active personal access tokens 0

Add a personal access token

Token name: demo

For example, the application using the token or the purpose of the token.

Expiration date: 2025-01-18

Select scopes

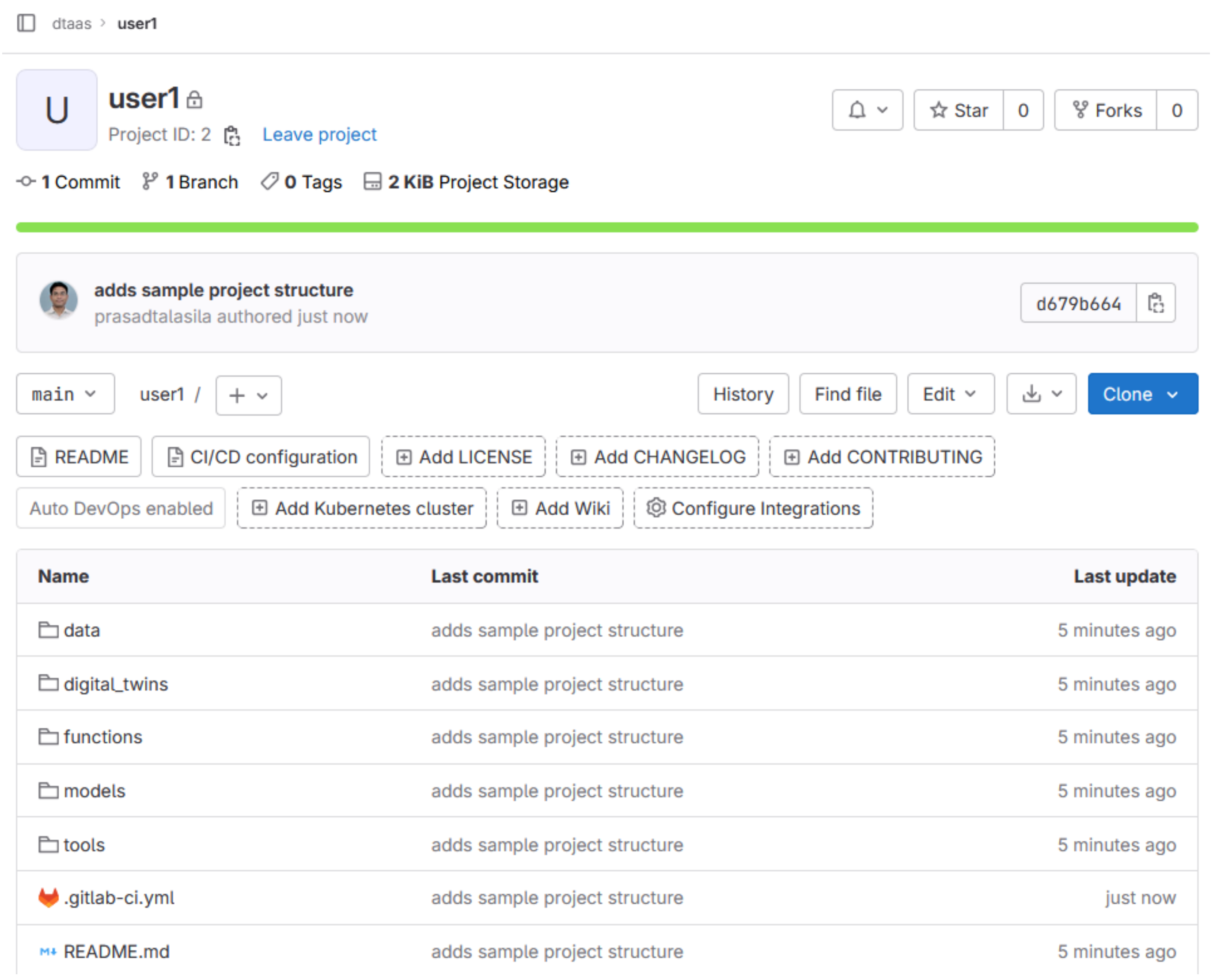
Scopes set the permission levels granted to the token. Learn more.

- [✓] api — Grants complete read/write access to the API, including all groups and projects, the container registry, and the p
- [ ] read_api — Grants read access to the API, including all groups and projects, the container registry, and the package registry
- [ ] read_user — Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes useri API endpoints under /users.
- [ ] create_runner — Grants create access to the runners.
- [ ] k8s_proxy — Grants permission to perform Kubernetes API calls using the agent for Kubernetes.
- [ ] read_repository — Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.
- [✓] write_repository

This token should be copied and used to clone the git repository.

2.6.3 Library Assets

GitLab is used to store the reusable **Library** assets of all users. A mandatory structure exists for storing and using Library assets including digital twins. A properly initialized GitLab project should have the following structure.


dtaas > user1

user1

Project ID: 2 Leave project

Star 0 Forks 0

1 Commit 1 Branch 0 Tags 2 KiB Project Storage

adds sample project structure
prasadtalasila authored just now

d679b664

main user1 / History Find file Edit Clone

README CI/CD configuration Add LICENSE Add CHANGELOG Add CONTRIBUTING
Auto DevOps enabled Add Kubernetes cluster Add Wiki Configure Integrations

Name	Last commit	Last update
data	adds sample project structure	5 minutes ago
digital_twins	adds sample project structure	5 minutes ago
functions	adds sample project structure	5 minutes ago
models	adds sample project structure	5 minutes ago
tools	adds sample project structure	5 minutes ago
.gitlab-ci.yml	adds sample project structure	just now
README.md	adds sample project structure	5 minutes ago

Please pay special attention to `.gitlab-ci.yml`. It must be a valid GitLab DevOps configuration. The example repo provides a sample structure.

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

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

After adding the required Library assets:

```
$git push origin
```

2.6.4 Next Steps

A GitLab runner should be integrated with the project repository. Runners may already be installed with the DTaaS platform. These can be verified on the runners page. Additionally, custom runners can be installed and integrated with the repository.

The Digital Twins Preview can then 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.

Warning

This npm package works only on Linux platforms

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

Several example digital twins have been created for the DTaaS platform. These examples can be explored, and the steps provided in this **Examples** section can be followed to experience features of the DTaaS platform and understand best practices for managing digital twins within the platform. The following slides and video provide an overview of these examples.

Please see the following demos illustrating the use the DTaaS in two projects:

Project	Slides and Videos
CP-SENS project	Project Introduction: slides and video
	Wind turbine testing with the demo inside user workspace
	Python package and demo video
	Videos demonstrating digital twin for structural health monitoring applications: Data Acquisition System: Part-1, Data Acquisition System: Part-2, Operational Model Analysis, and Model Updating
Incubator	video

Copy Examples

The first step is to copy all example code into the user workspace within a user workspace. The provided shell script copies all examples into the `/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 complexity. It is recommended 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

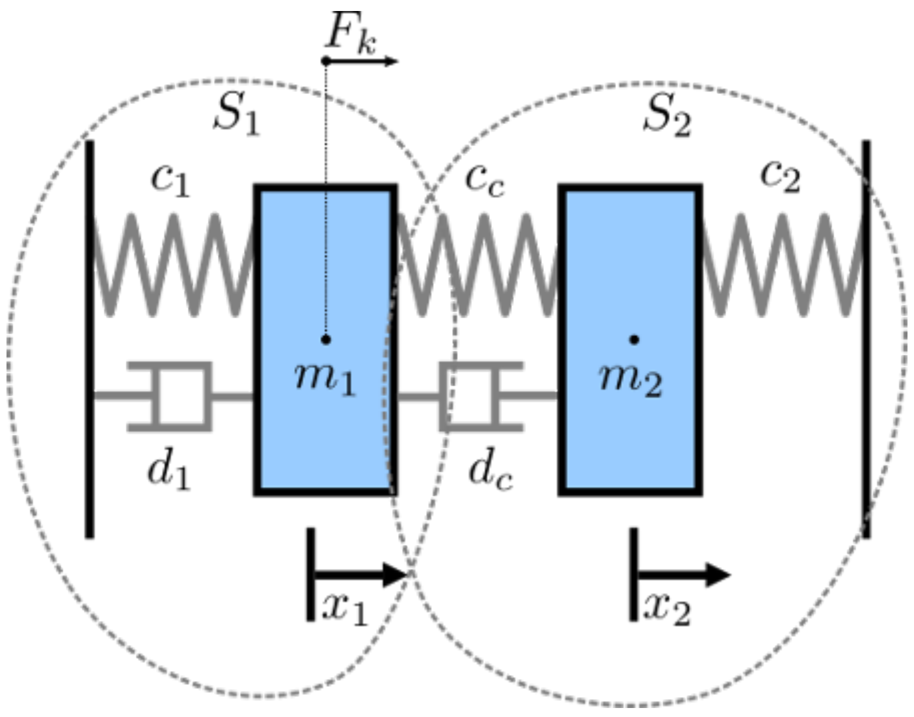
⬇ DTaaS examples

2.8.2 Mass Spring Damper

Overview

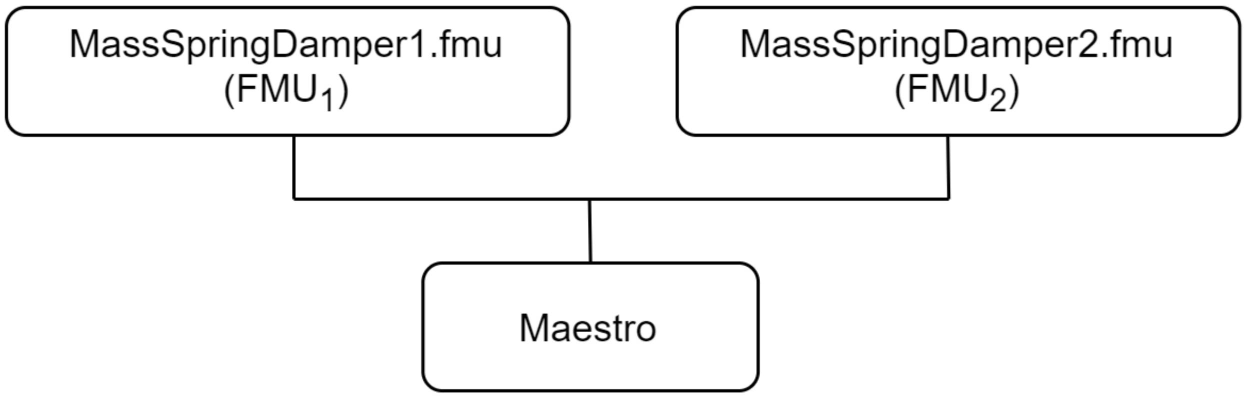
The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within the DTaaS.

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.


MassSpringDamper1.fmu (FMU_1)
MassSpringDamper2.fmu (FMU_2)
Maestro

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. These two files can be modified to customize the DT for specific requirements.

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*.

Run logs can also be viewed 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.

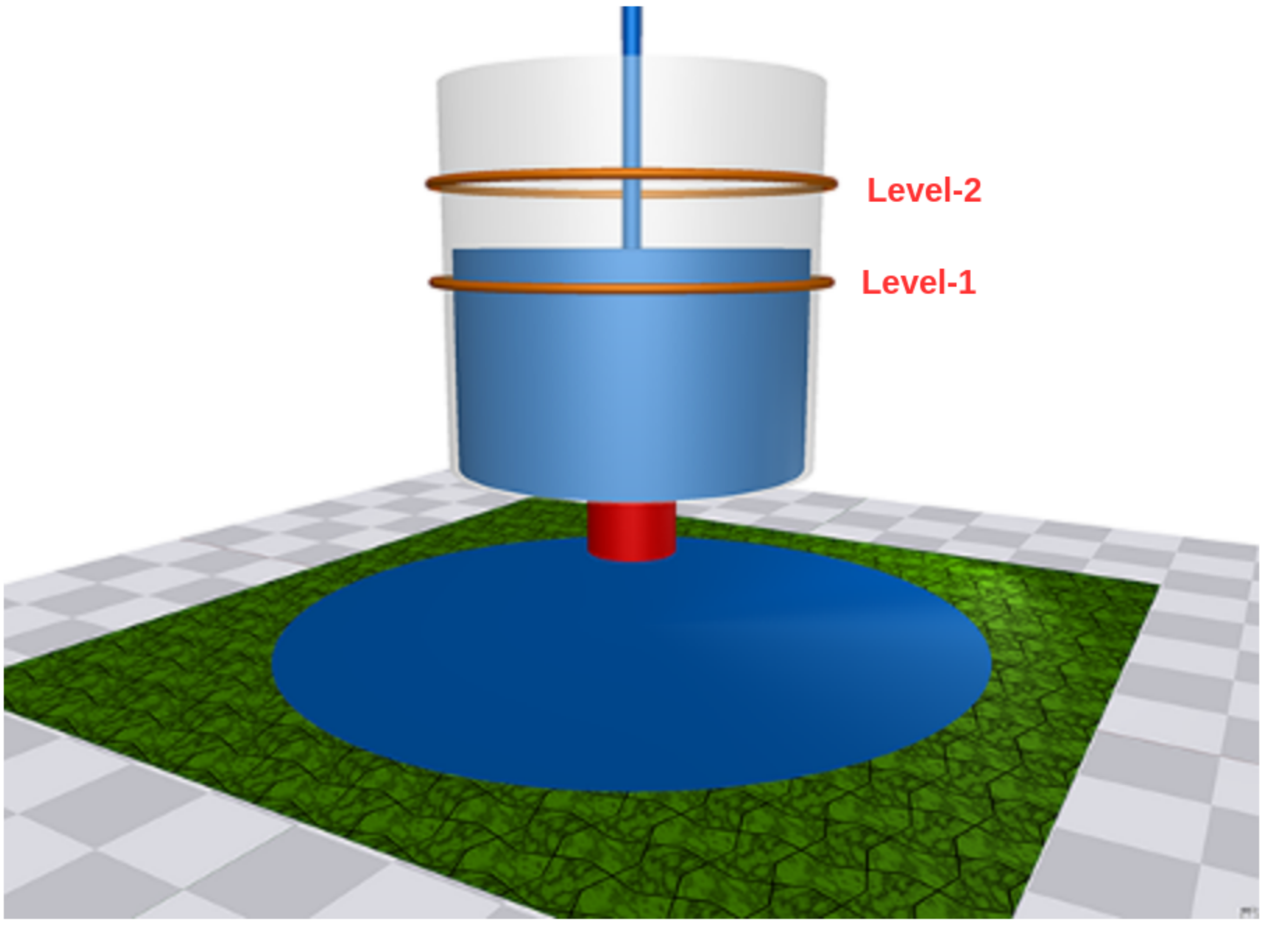
2.8.3 Water Tank Fault Injection

Overview

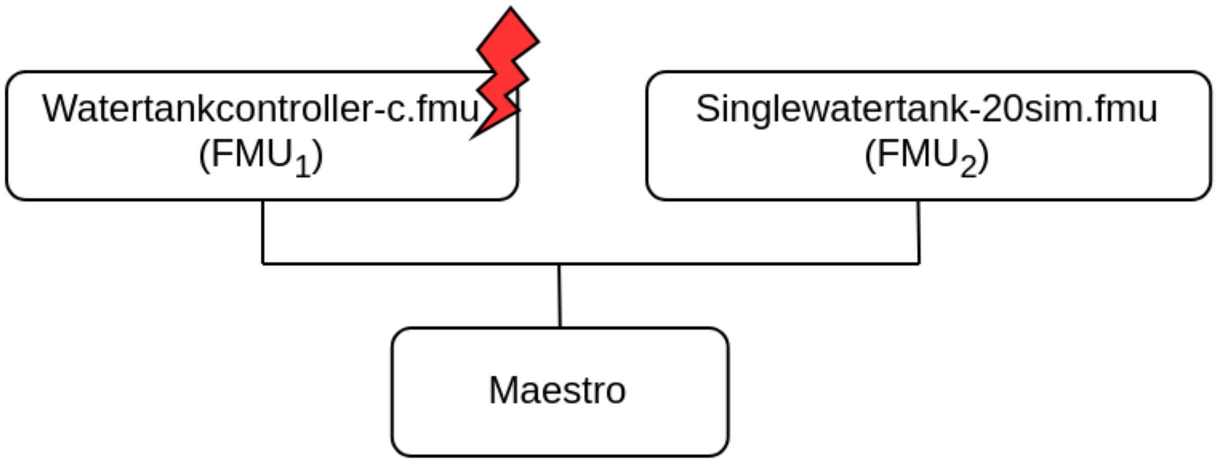
This example demonstrates a fault injection (FI) enabled digital twin (DT). A live DT is subjected to simulated faults received from the environment. The simulated faults are specified as part of DT configuration and can be changed for new instances of DTs.

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. These two files can be modified to customize the DT for specific requirements.

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

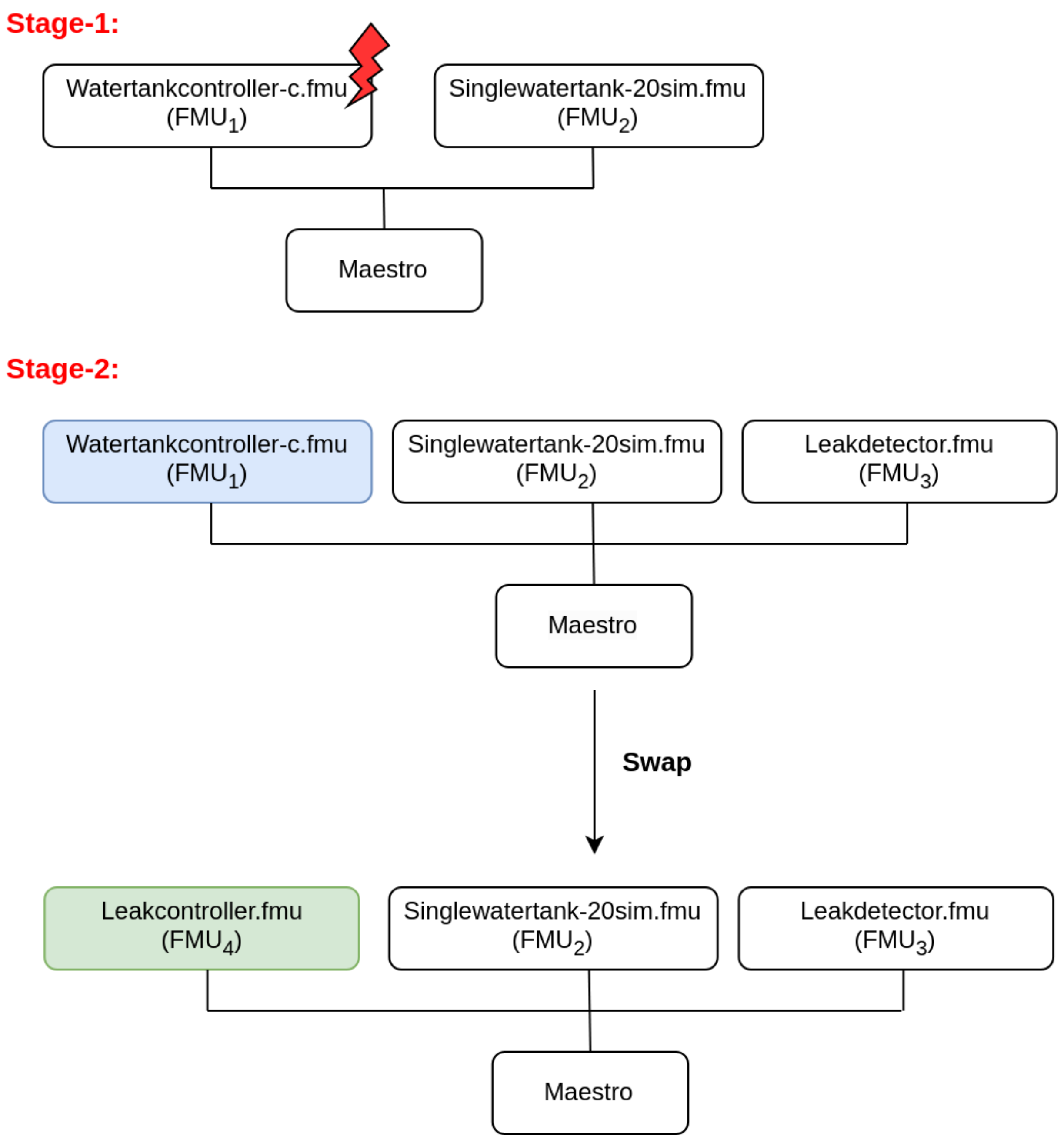
Overview

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

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

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

Example Structure


Stage-1:
Watertankcontroller-c.fmu (FMU1)
Singlewatertank-20sim.fmu (FMU2)
Maestro
Stage-2:
Watertankcontroller-c.fmu (FMU1)
Singlewatertank-20sim.fmu (FMU2)
Leakdetector.fmu (FMU3)
Maestro
Swap
Leakcontroller.fmu (FMU4)
Singlewatertank-20sim.fmu (FMU2)
Leakdetector.fmu (FMU3)
Maestro

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).
```

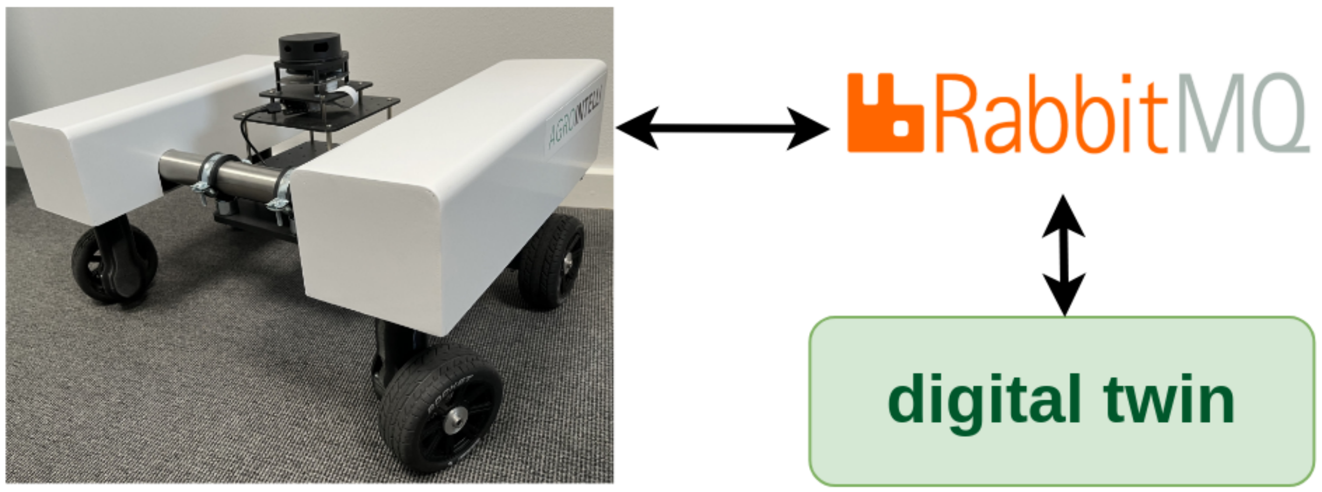
The runtime reconfiguration of co-simulation by modifying the Functional Mockup Units (FMUs) used is further detailed in the paper:

```
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.

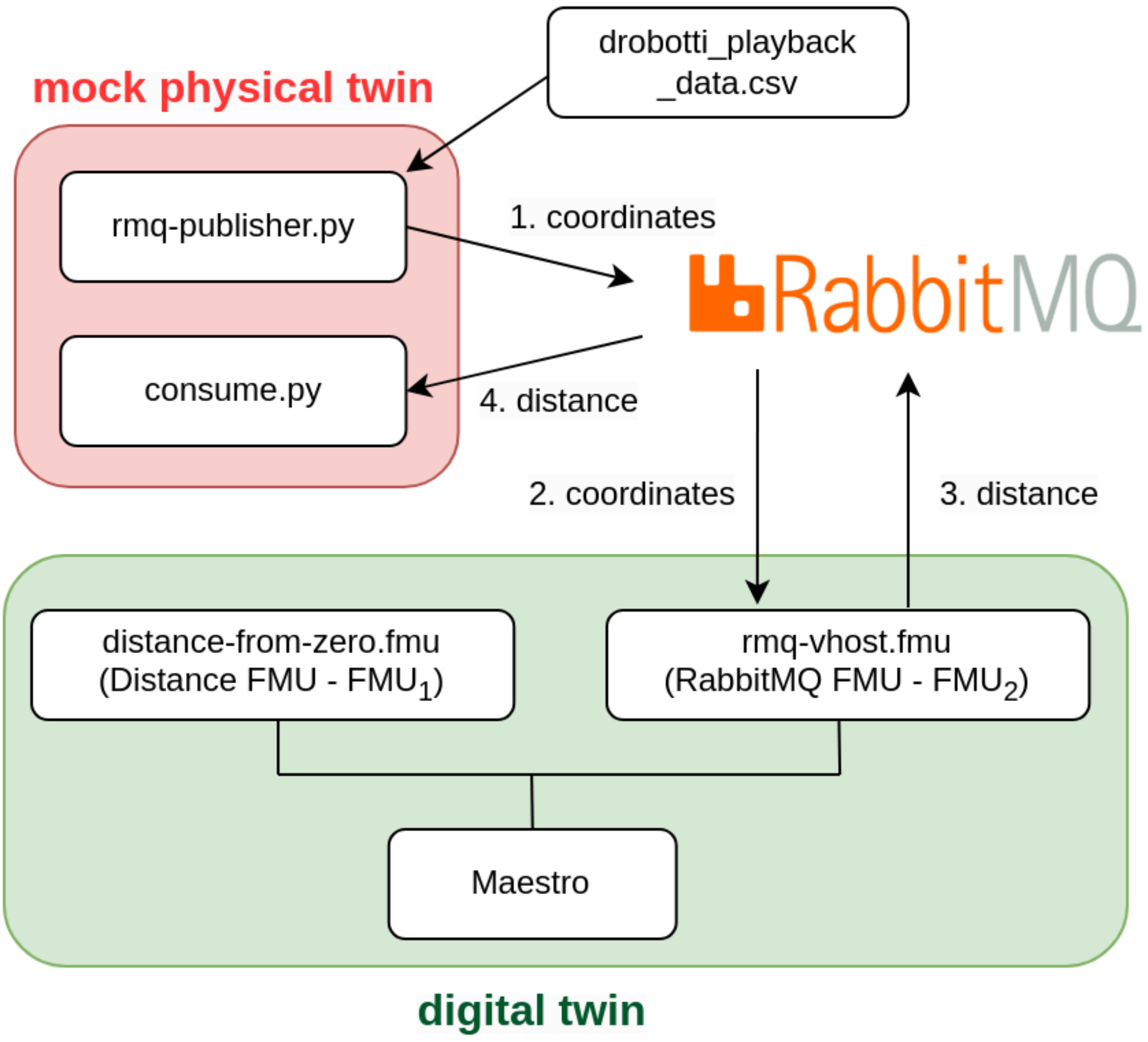


Example Structure

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. These two files can be modified to customize the DT for specific requirements.

The RabbitMQ access credentials need to be provided in `multimodel.json`. The `rabbitMQ-credentials.json` provides RabbitMQ access credentials for mock PT python scripts. The appropriate credentials should be added 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.
```

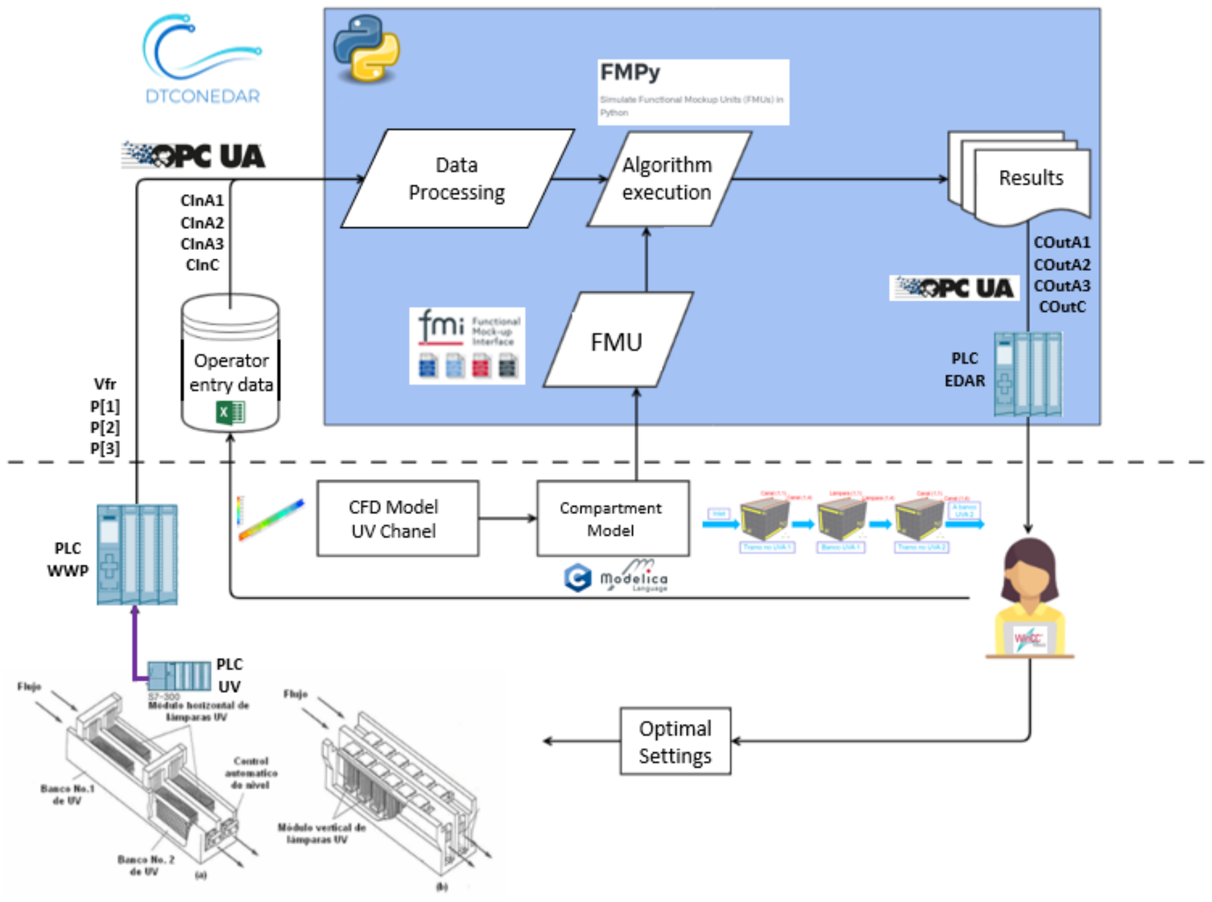
2.8.6 Waste Water Plant with OPC-UA

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.

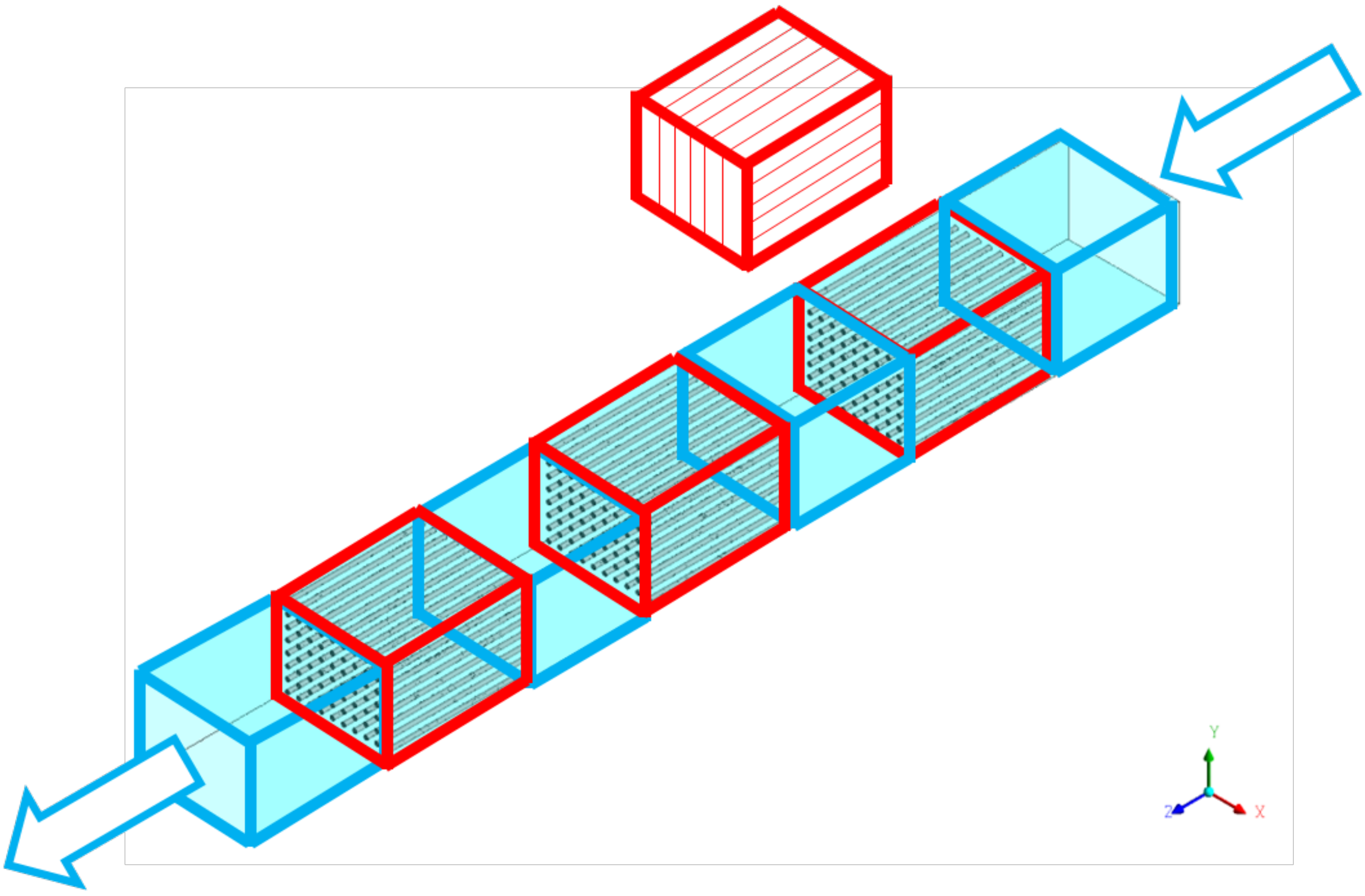


The physical twin of the waste water plant is composed of an ultraviolet channel controlled by a PLC that controls the power of the UV lamps needed to kill all the pathogens of the flow. The channel has 3 groups of UV lamps, therefore the real channel (and is mathematical model) is subdivided into 7 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.

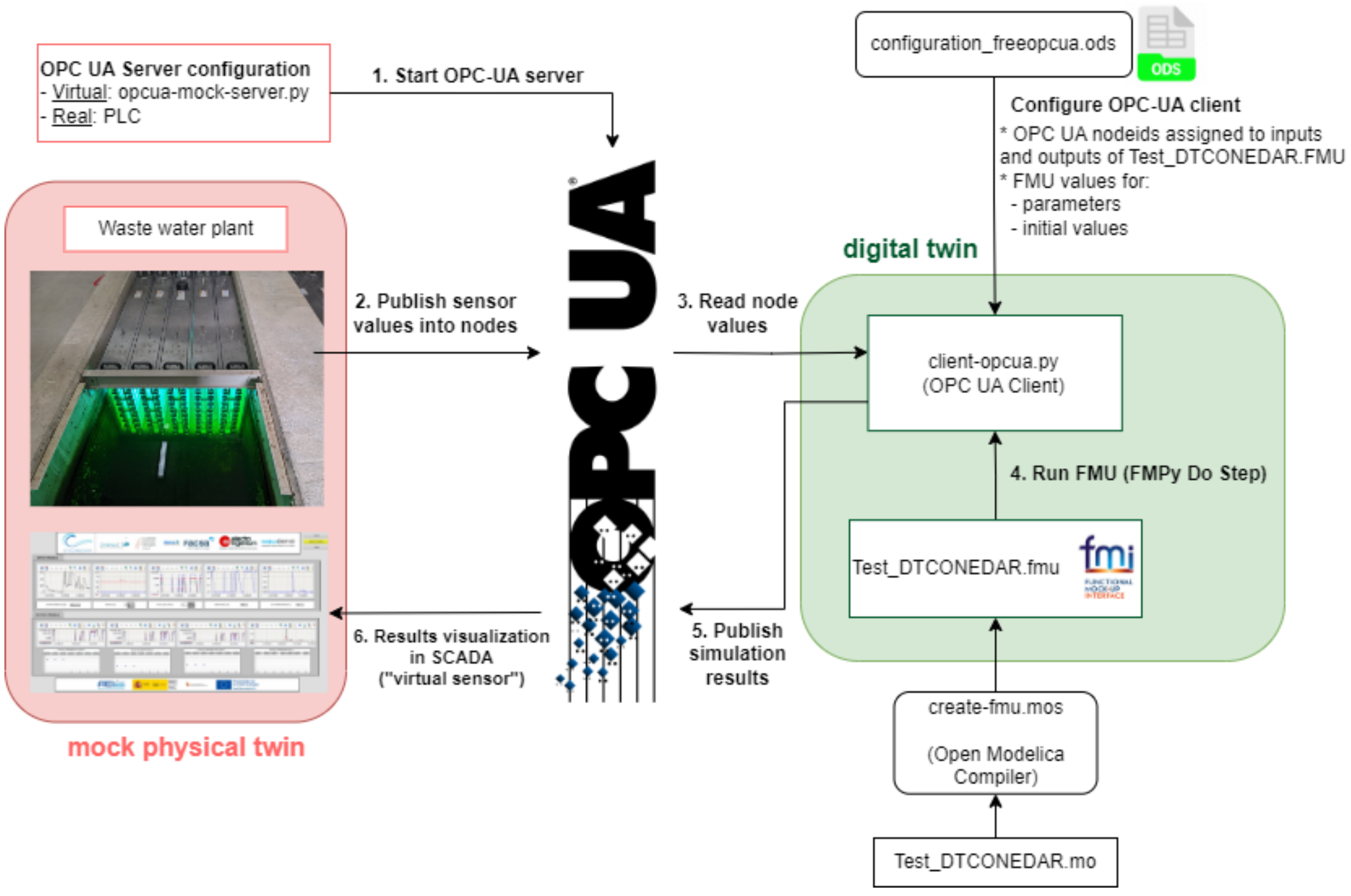
The dose to be applied (related with the power) changes according to the residence time (computed from the measure of the volume flow) and the UV intensity (measured by the intensity sensor).

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*).


OPC UA Server configuration
- Virtual: opcua-mock-server.py
- Real: PLC
1. Start OPC-UA server
configuration_freeopcua.ods
ODS
Configure OPC-UA client
* OPC UA nodeids assigned to inputs and outputs of Test_DTCONEDAR.FMU
* FMU values for:
- parameters
- initial values
digital twin
Waste water plant
2. Publish sensor values into nodes
OPC UA
3. Read node values
client-opcua.py (OPC UA Client)
4. Run FMU (FMPy Do Step)
Test_DTCONEDAR.fmu
fmi
6. Results visualization in SCADA ("virtual sensor")
5. Publish simulation results
create-fmu.mos (Open Modelica Compiler)
Test_DTCONEDAR.mo
mock physical twin

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`.

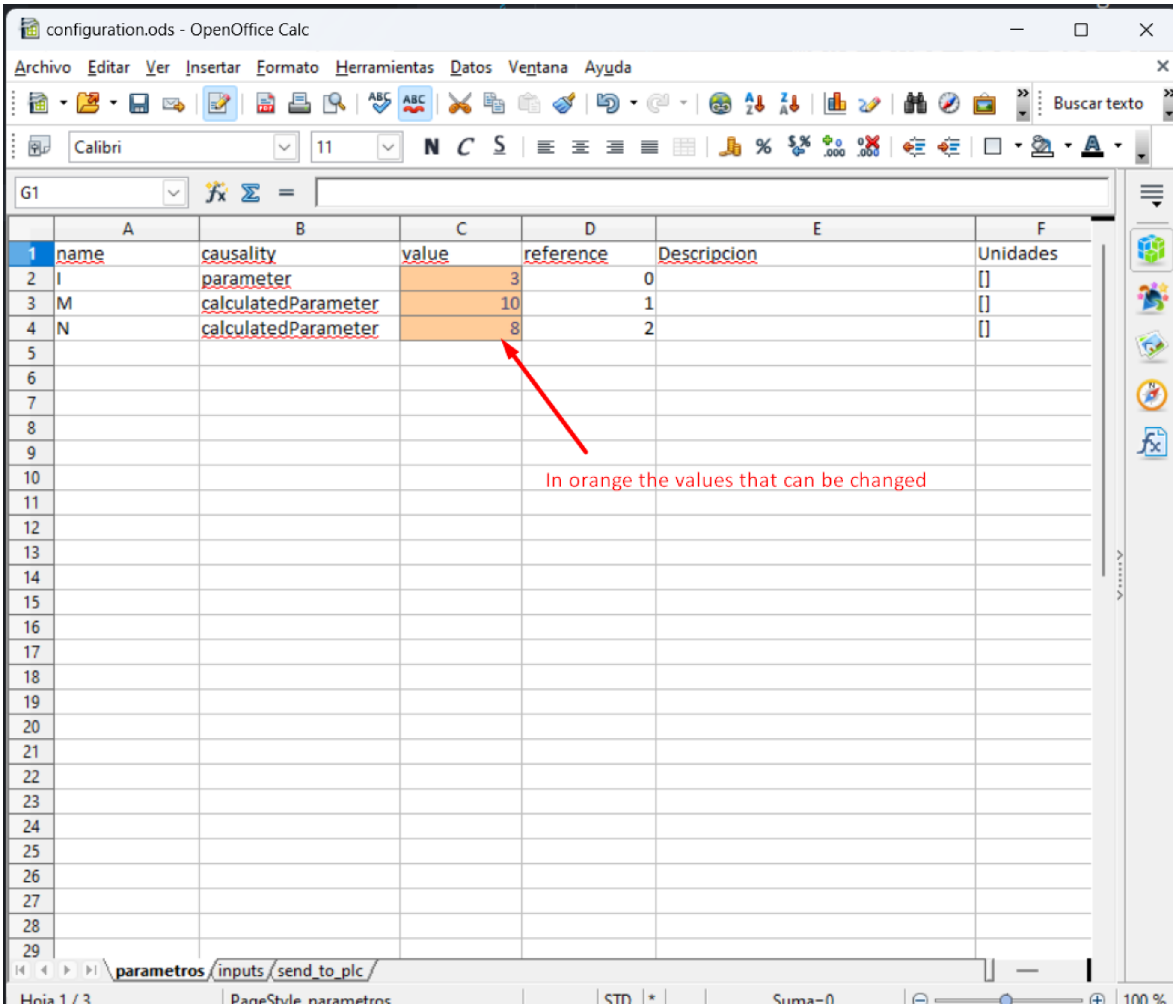
The FMU model description file (`modelDescription.xml` file inside `Test_DTCONEDAR.fmu`) has the information of the value references configuration_freeopcua.ods has the information of the OPC-UA node IDs And both have in common the variable name

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

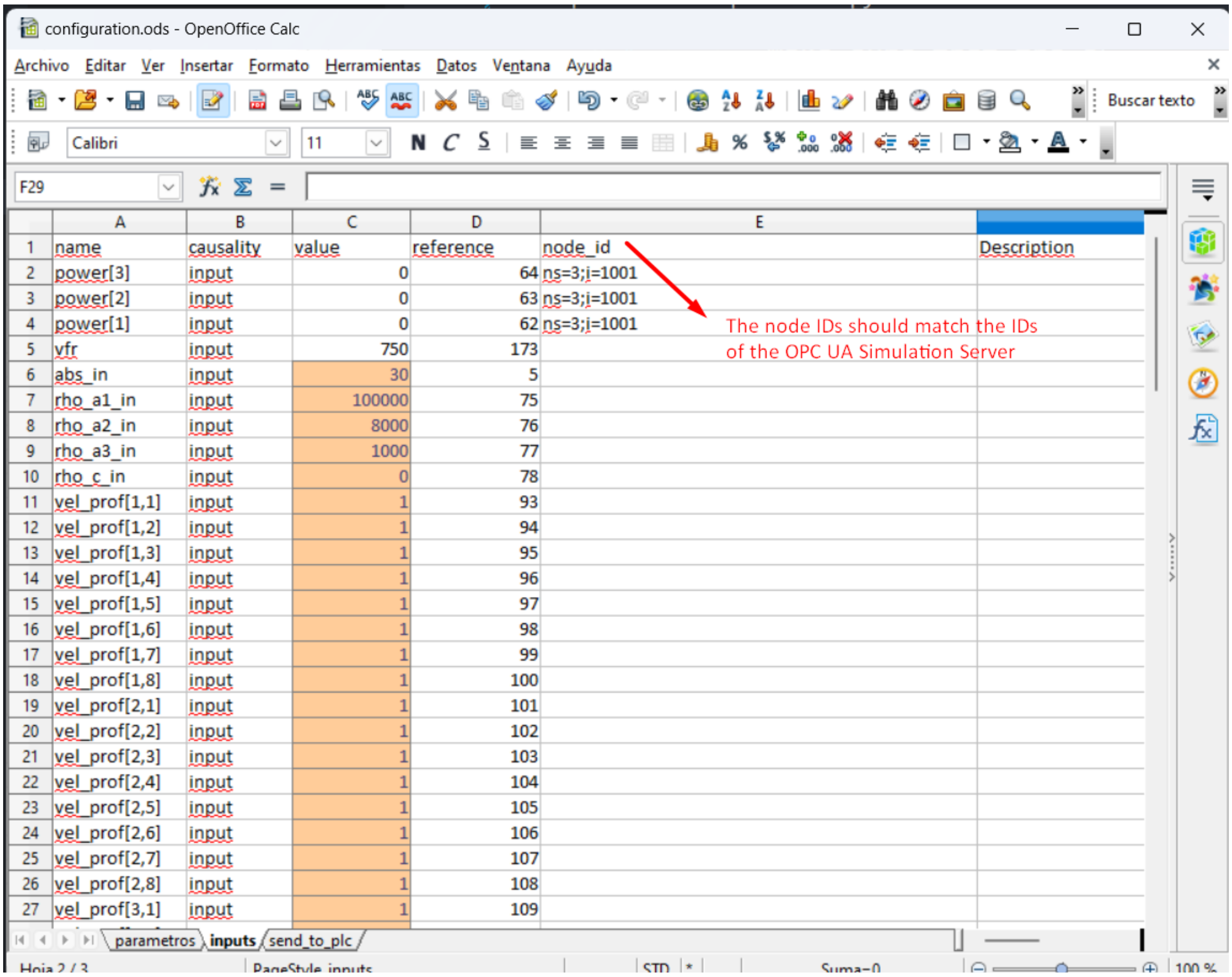
The *configuration_freeopcua.ods* date file is used for customizing the initial input data values used by the server.



	A	B	C	D	E	F
1	name	causality	value	reference	Descripcion	Unidades
2	I	parameter	3	0		[]
3	M	calculatedParameter	10	1		[]
4	N	calculatedParameter	8	2		[]

In orange the values that can be changed

configuration.ods - OpenOffice Calc
Archivo Editar Ver Insertar Formato Herramientas Datos Ventana Ayuda
Buscar texto
Calibri
11
G1
parametros
inputs
send_to_plc
Hoja 1 / 3
PageStyle_parametros
STD
Suma=0
100 %

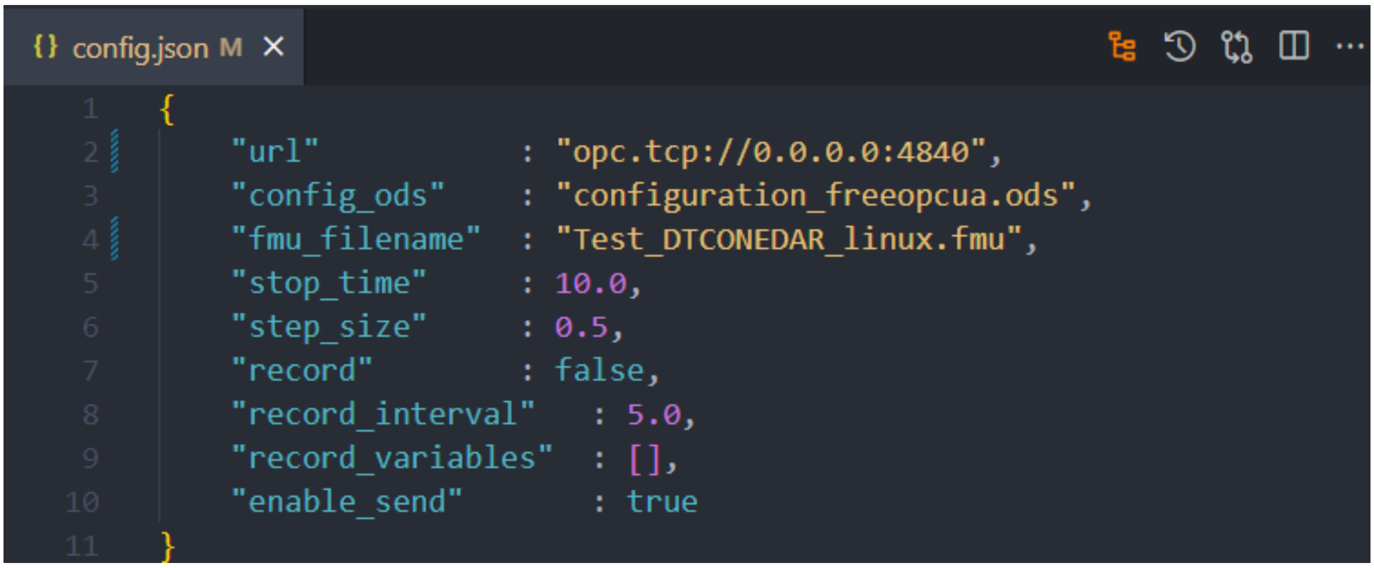

configuration.ods - OpenOffice Calc
Archivo Editar Ver Insertar Formato Herramientas Datos Ventana Ayuda
Buscar texto
Calibri
11
F29
parametros
inputs
send_to_plc
Hoja 2 / 3
PageStyle_inputs
STD
Suma=0
100 %

	A	B	C	D	E	Description
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		

The node IDs should match the IDs of the OPC UA Simulation Server

DT CONFIG

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



```json
{
    "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)
```

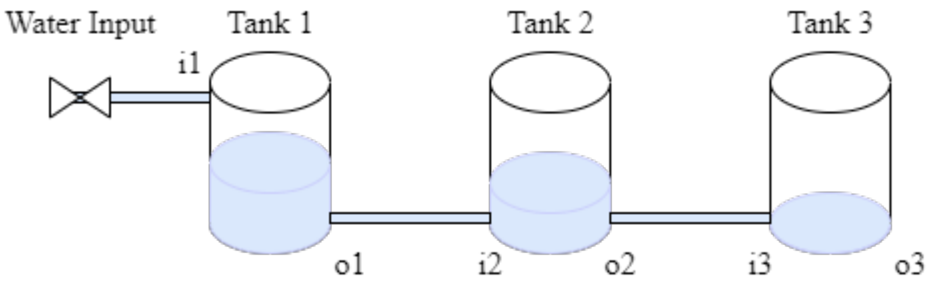
Acknowledgements

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 that represents a system composed of three individual components 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.



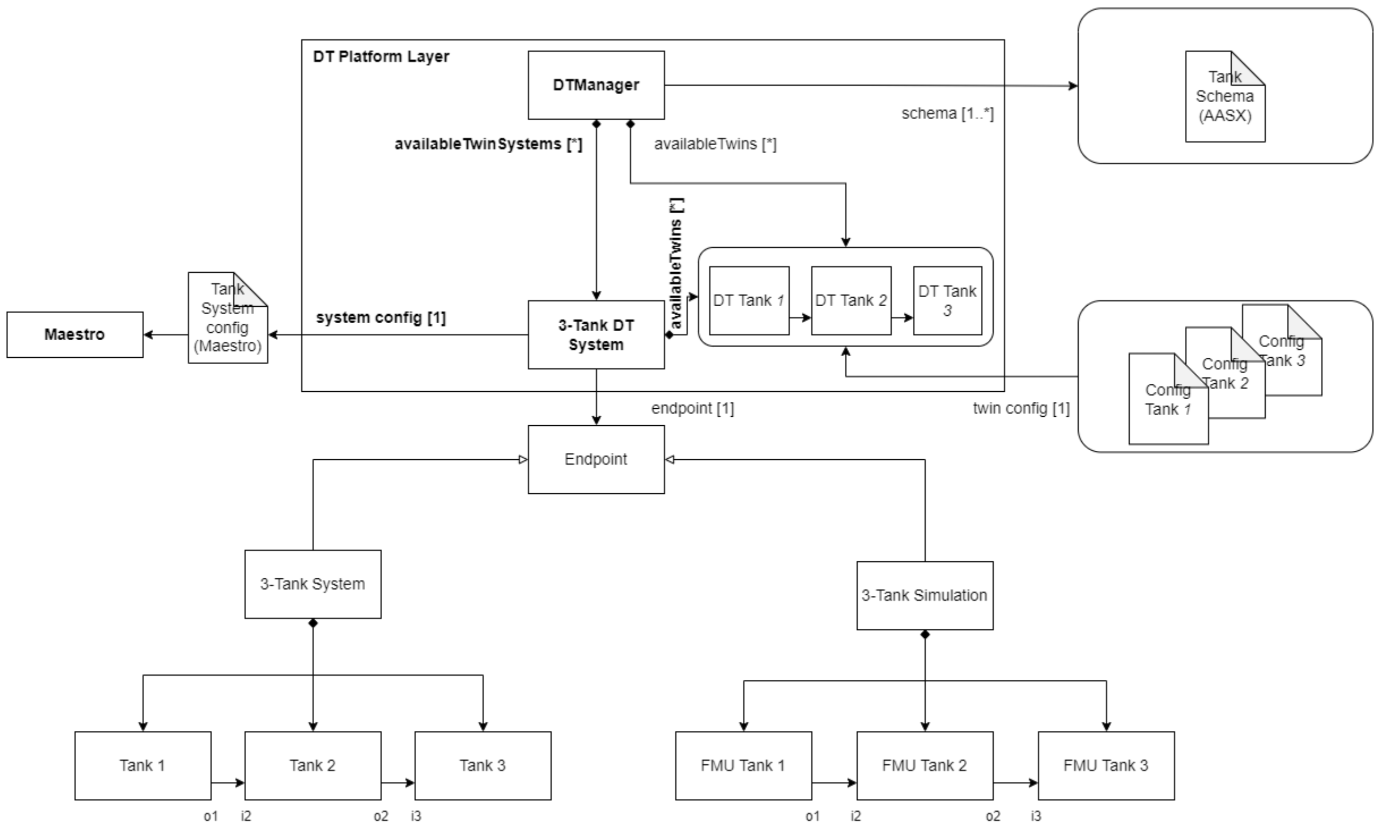
This example contains only the simulated components for demonstration purposes; therefore, there is no configuration for the connection with the physical system.

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 to which they should move.

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.

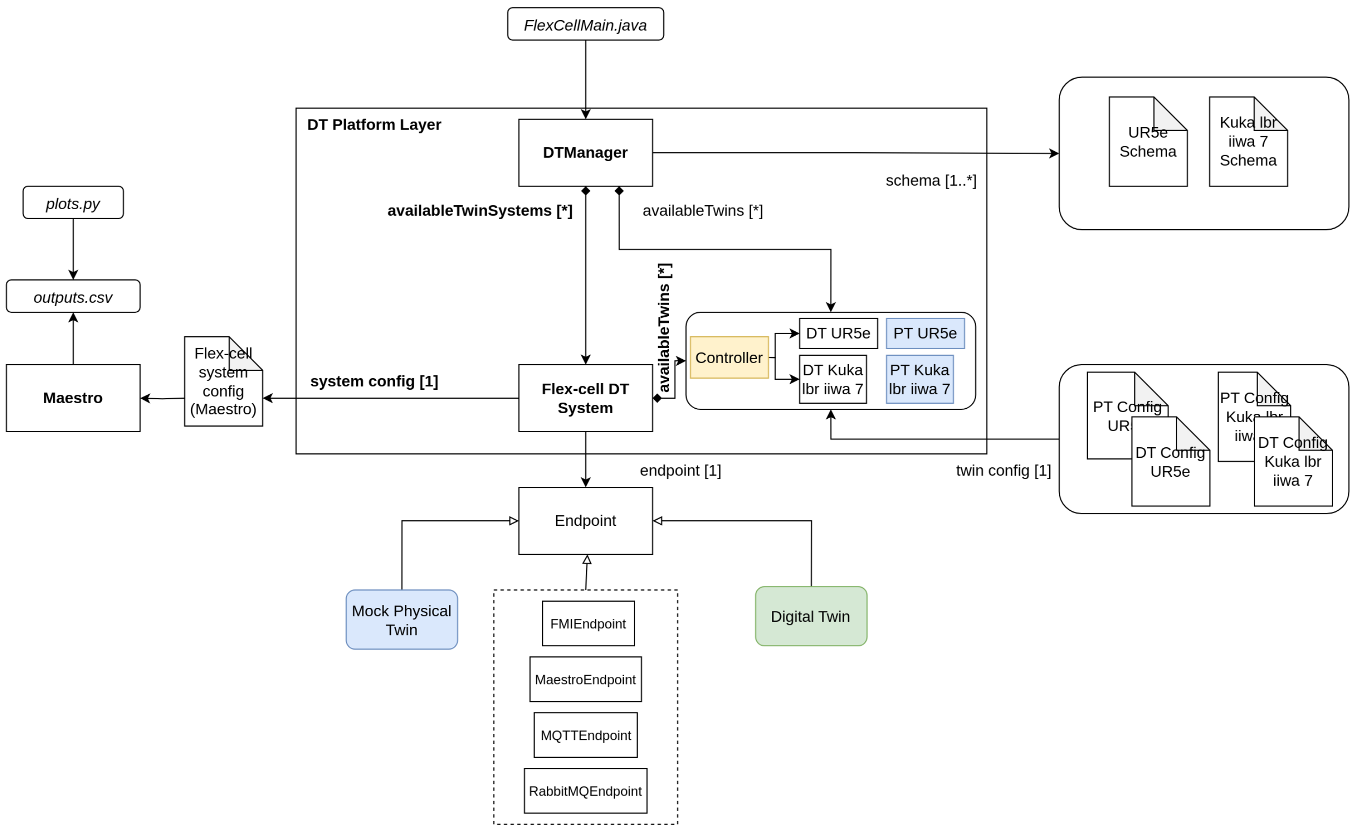
The TwinManager orchestrates the flex-cell DT via the *Flex-cell DT System*, which is composed of 2 smaller Digital Twins, namely, the *DT UR5e* and the *DT Kuka lbr iiwa 7*. The TwinManager also provides the interface for the Physical Twins, namely, *PT UR5e* and *PT Kuka lbr iiwa 7*. Each Physical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

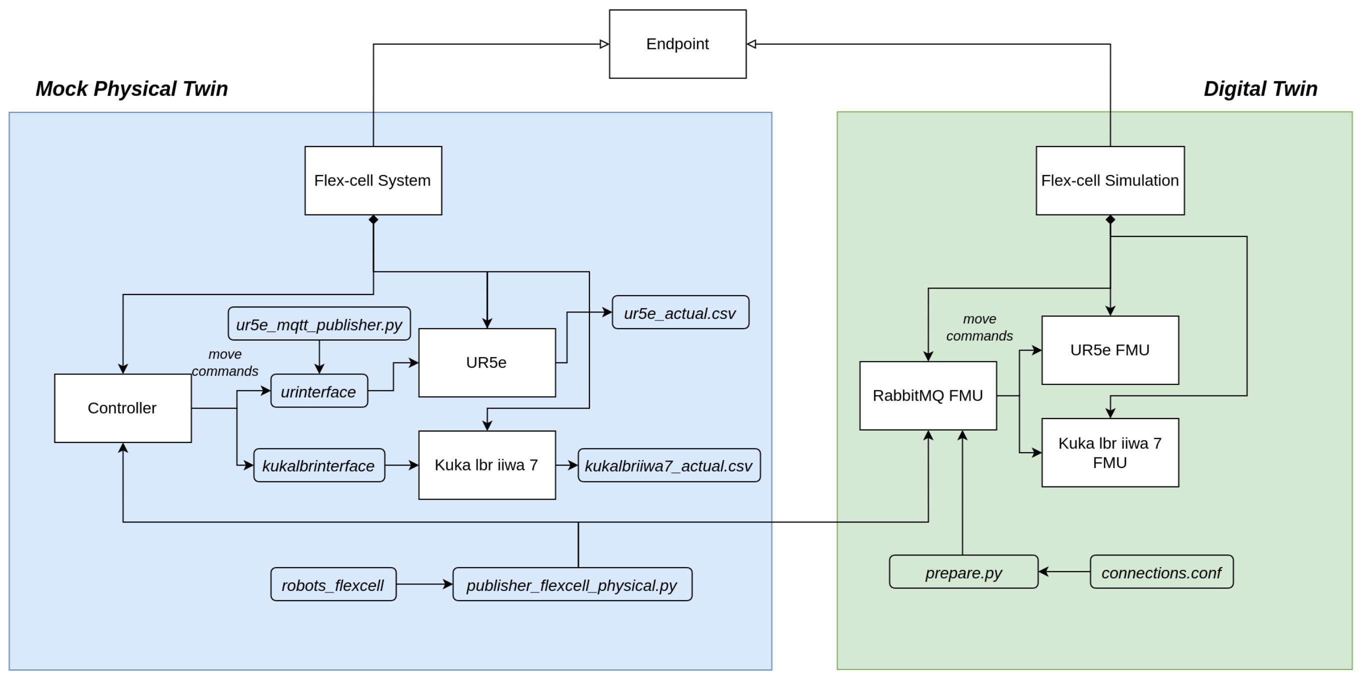
The current endpoints used in this implementation are:

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.




Endpoint
Mock Physical Twin
Digital Twin
Flex-cell System
Flex-cell Simulation
ur5e_mqtt_publisher.py
UR5e
ur5e_actual.csv
move commands
Controller
urinterface
kukalbrinterface
Kuka lbr iiwa 7
kukalbriiwa7_actual.csv
robots_flexcell
publisher_flexcell_physical.py
move commands
UR5e FMU
RabbitMQ FMU
Kuka lbr iiwa 7 FMU
prepare.py
connections.conf

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

This step configures different assets of the DT with connection credentials. The `functions/flex-cell/prepare.py` script is used for this purpose. The only step 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 generates a co-simulation output. The results can also be monitored by updating the `/workspace/examples/tools/flex-cell/FlexCellDTaaS.java` with a specific set of `getAttributeValue` commands, as shown in the code. That main file enables the online execution and comparison of 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**.

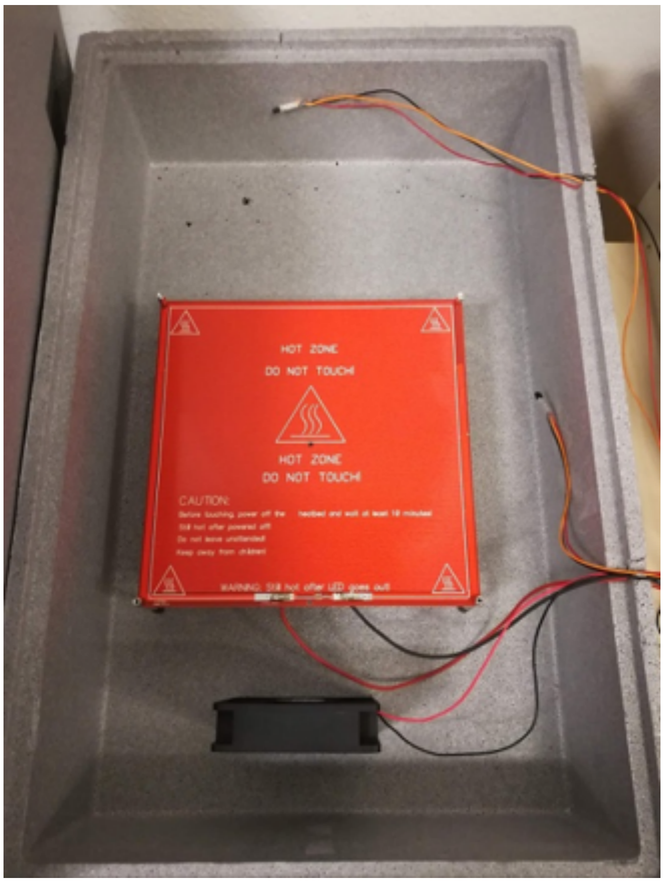
More information about the TwinManager (formerly DT Manager) and the case study is available in:

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.

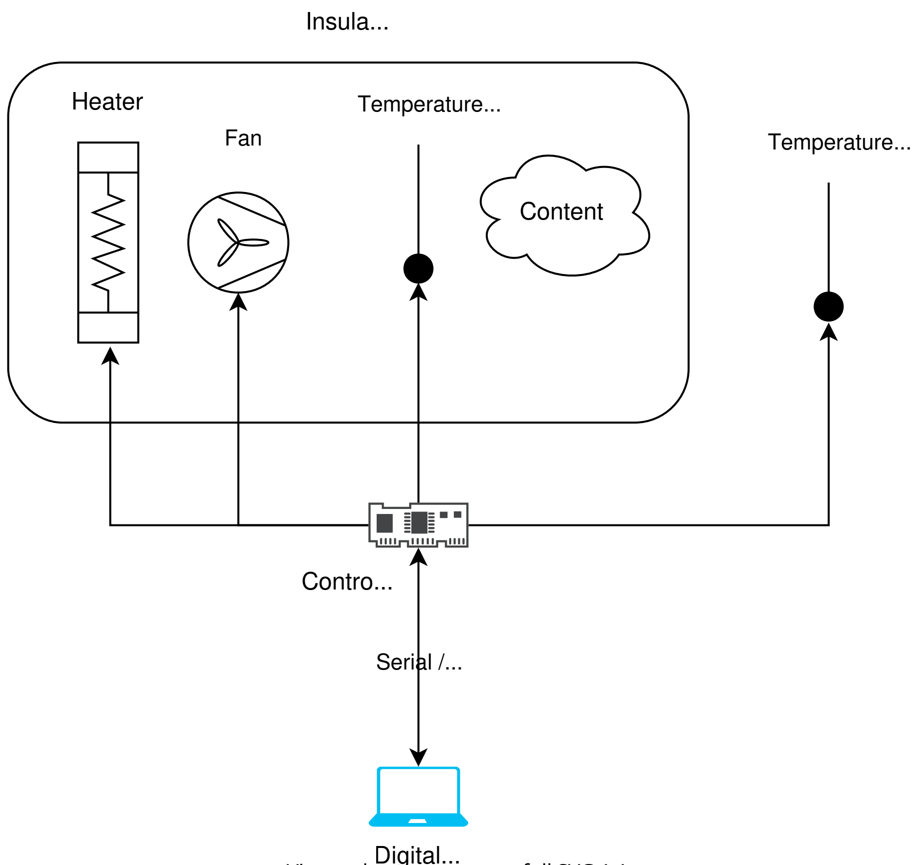
2.8.9 Incubator Digital Twin

Overview

This is a case study of an Incubator intended to demonstrate the steps and processes involved in developing a digital twin system. This incubator is an insulated container with the ability to maintain a temperature and provide heating, but not cooling. A picture of the incubator is provided 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:



Viewer does not support full SVG 1.1

The system consists of:

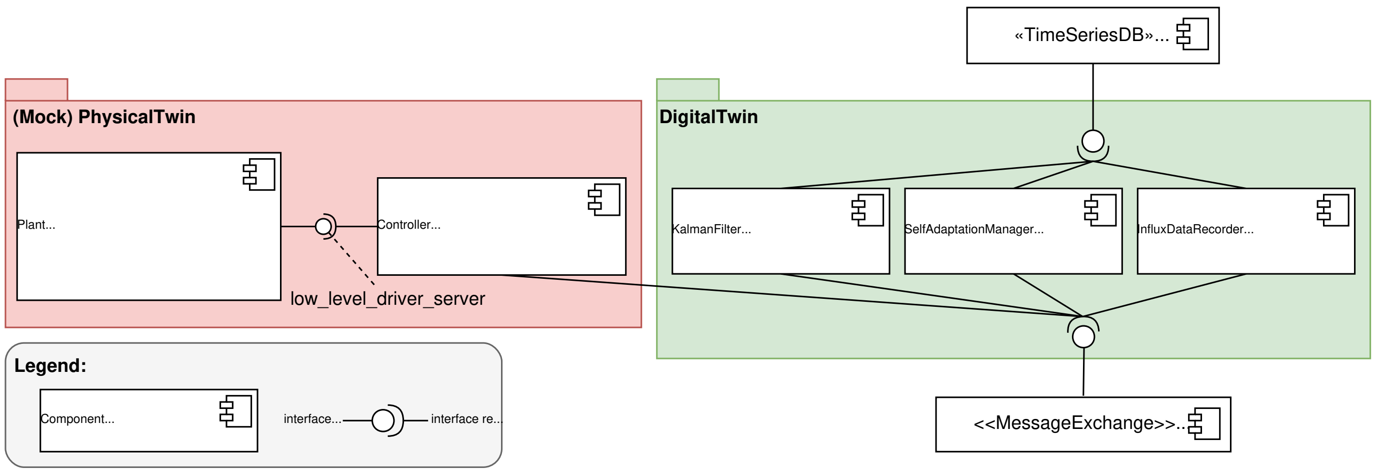
- 1x styrofoam box in order to have an insulated container
- 1x heat source to heat up the content within the Styrofoam box.
- 1x fan to distribute the heating within the box
- 2x temperature sensor to monitor the temperature within the box
- 1x temperature Sensor to monitor the temperature outside the 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 RabbitMQ and InfluxDB instances.

There is one configuration file: `simulation.conf`. The RabbitMQ and InfluxDB configuration parameters must 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.

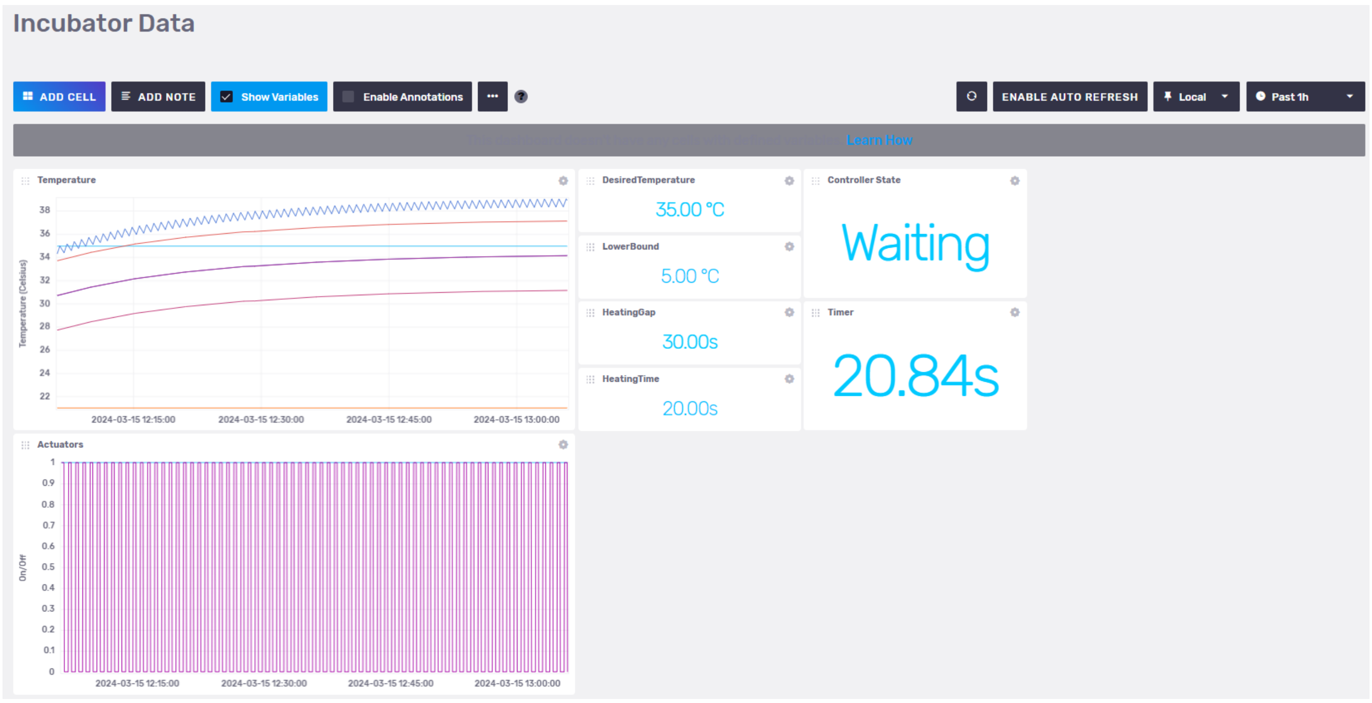
```
lifecycle/clean
```

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 a firefighter mission, it is important to monitor the oxygen levels of each firefighter's Self Contained Breathing Apparatus (SCBA) in the 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

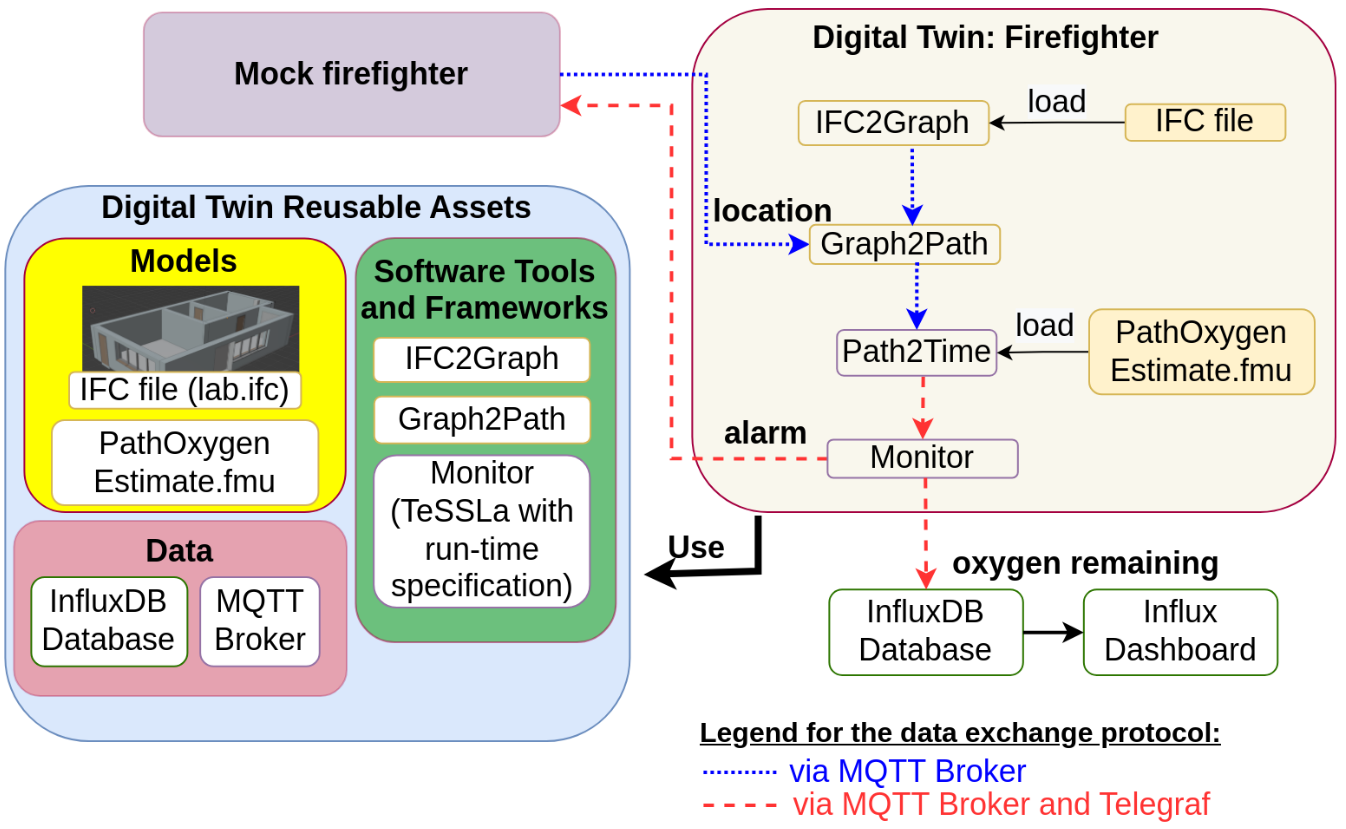
In this example a monitor is implemented, that calculates how much time the firefighers have left, until they need to leave the building. To that end, the inputs used are:

- 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 running this example, the following files must be verified to be 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

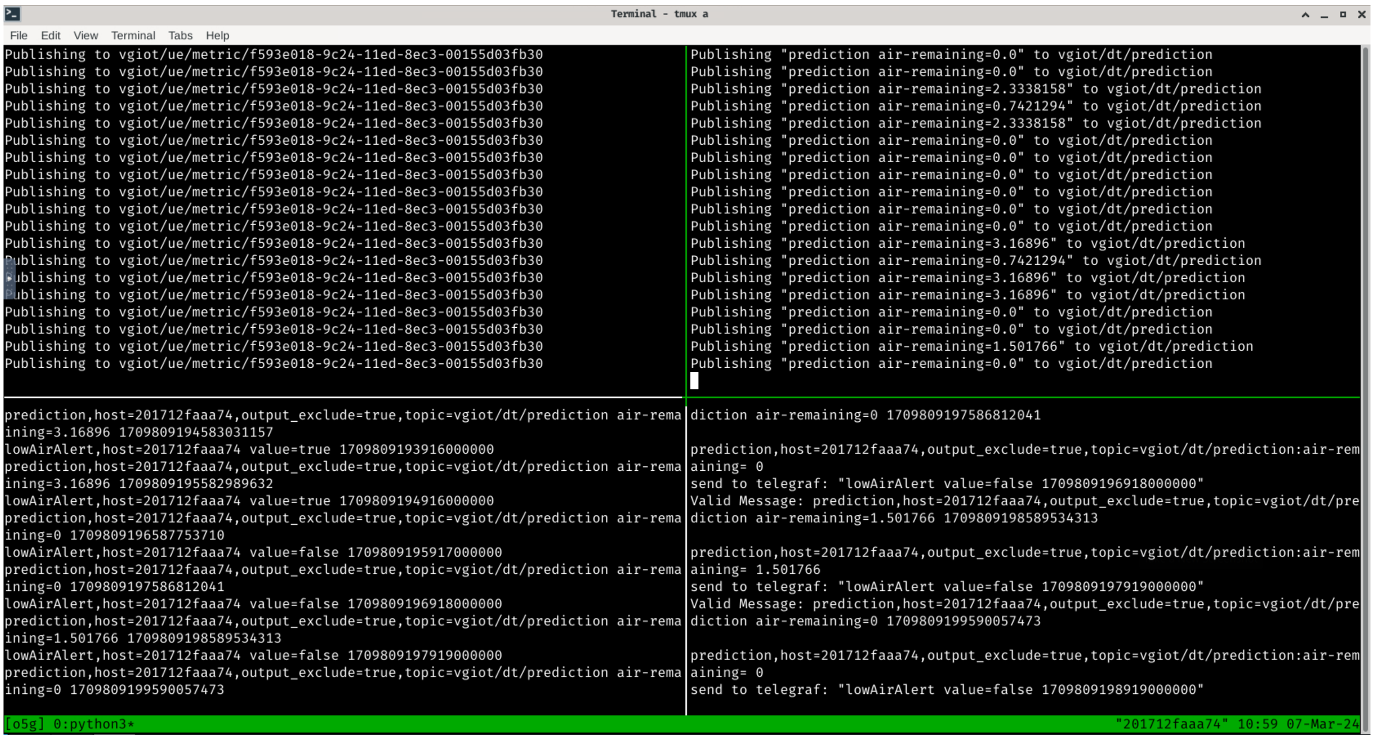
To run the Digital Twin execute

```
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`.



```
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
Publishing to vgiot/ue/metric/f593e018-9c24-11ed-8ec3-00155d03fb30
```

```
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
Publishing "prediction air-remaining=2.3338158" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.7421294" to vgiot/dt/prediction
Publishing "prediction air-remaining=2.3338158" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
Publishing "prediction air-remaining=3.16896" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.7421294" to vgiot/dt/prediction
Publishing "prediction air-remaining=3.16896" to vgiot/dt/prediction
Publishing "prediction air-remaining=3.16896" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
Publishing "prediction air-remaining=1.501766" to vgiot/dt/prediction
Publishing "prediction air-remaining=0.0" to vgiot/dt/prediction
```

```
prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/prediction air-rema
ining=3.16896 1709809194583031157
lowAirAlert,host=201712faaa74 value=true 1709809193916000000
prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/prediction air-rema
ining=3.16896 1709809195582989632
lowAirAlert,host=201712faaa74 value=true 1709809194916000000
prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/prediction air-rema
ining=0 1709809196587753710
lowAirAlert,host=201712faaa74 value=false 1709809195917000000
prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/prediction air-rema
ining=0 1709809197586812041
lowAirAlert,host=201712faaa74 value=false 1709809196918000000
prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/prediction air-rema
ining=1.501766 1709809198589534313
lowAirAlert,host=201712faaa74 value=false 1709809197919000000
prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/prediction air-rema
ining=0 1709809199590057473
```

```
diction air-remaining=0 1709809197586812041

prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/prediction:air-rem
aining= 0
send to telegraf: "lowAirAlert value=false 1709809196918000000"
Valid Message: prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/pre
diction air-remaining=1.501766 1709809198589534313

prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/prediction:air-rem
aining= 1.501766
send to telegraf: "lowAirAlert value=false 1709809197919000000"
Valid Message: prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/pre
diction air-remaining=0 1709809199590057473

prediction,host=201712faaa74,output_exclude=true,topic=vgiot/dt/prediction:air-rem
aining= 0
send to telegraf: "lowAirAlert value=false 1709809198919000000"
```

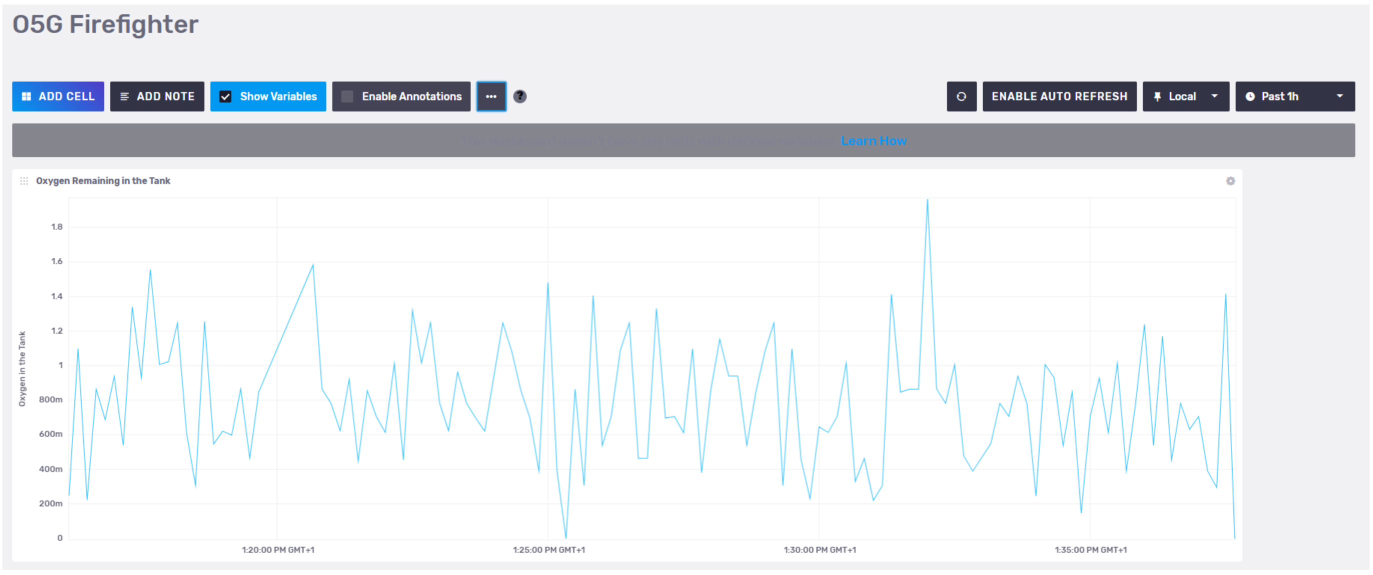
```
[o5g] 0:python3*                                              "201712faaa74" 10:59 07-Mar-24
```

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
```

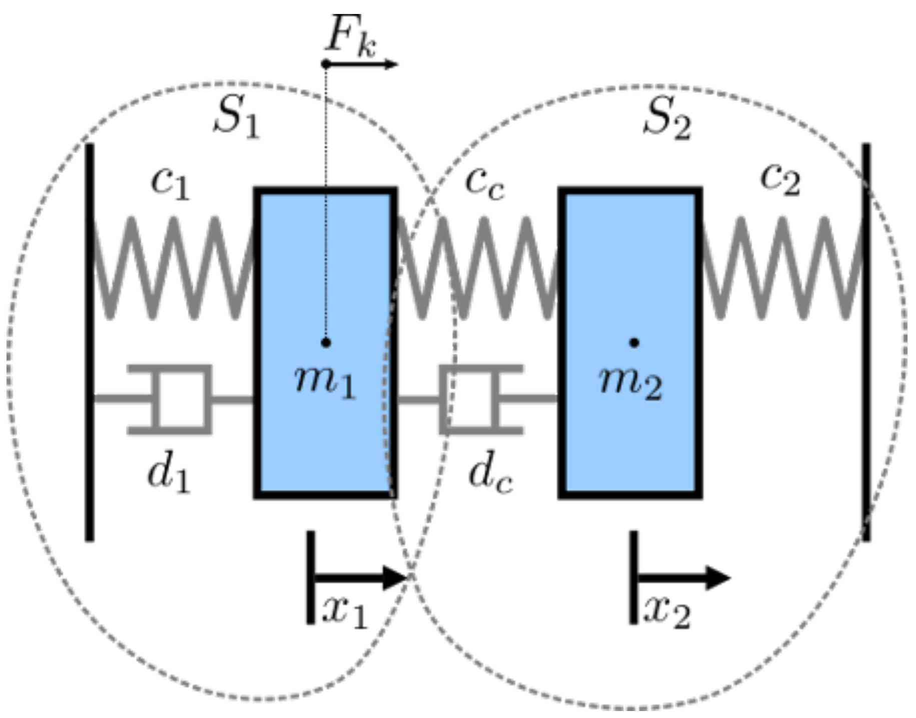
2.8.11 Mass Spring Damper with NuRV Runtime Monitor

Overview

This digital twin is derived from the Mass Spring 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 the DTaaS. This version of the example is expanded with a monitor generated by NuRV. More information about NuRV is available here.

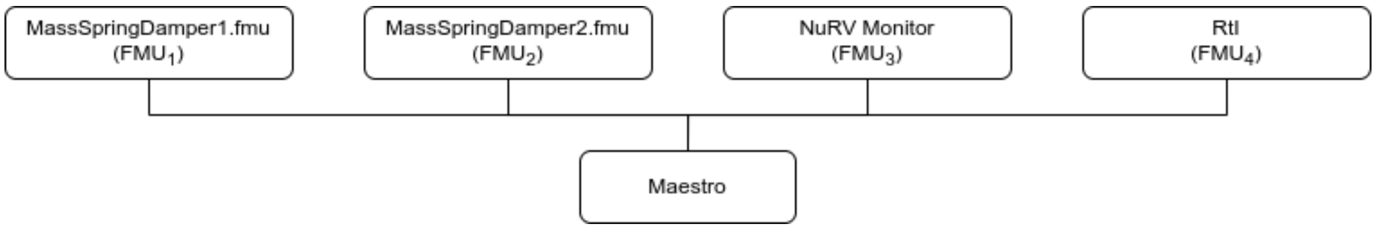
Example Diagram



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.


MassSpringDamper1.fmu (FMU_1)
MassSpringDamper2.fmu (FMU_2)
NuRV Monitor (FMU_3)
Rtl (FMU_4)
Maestro

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. These two files can be modified to customize the DT for specific requirements.

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.

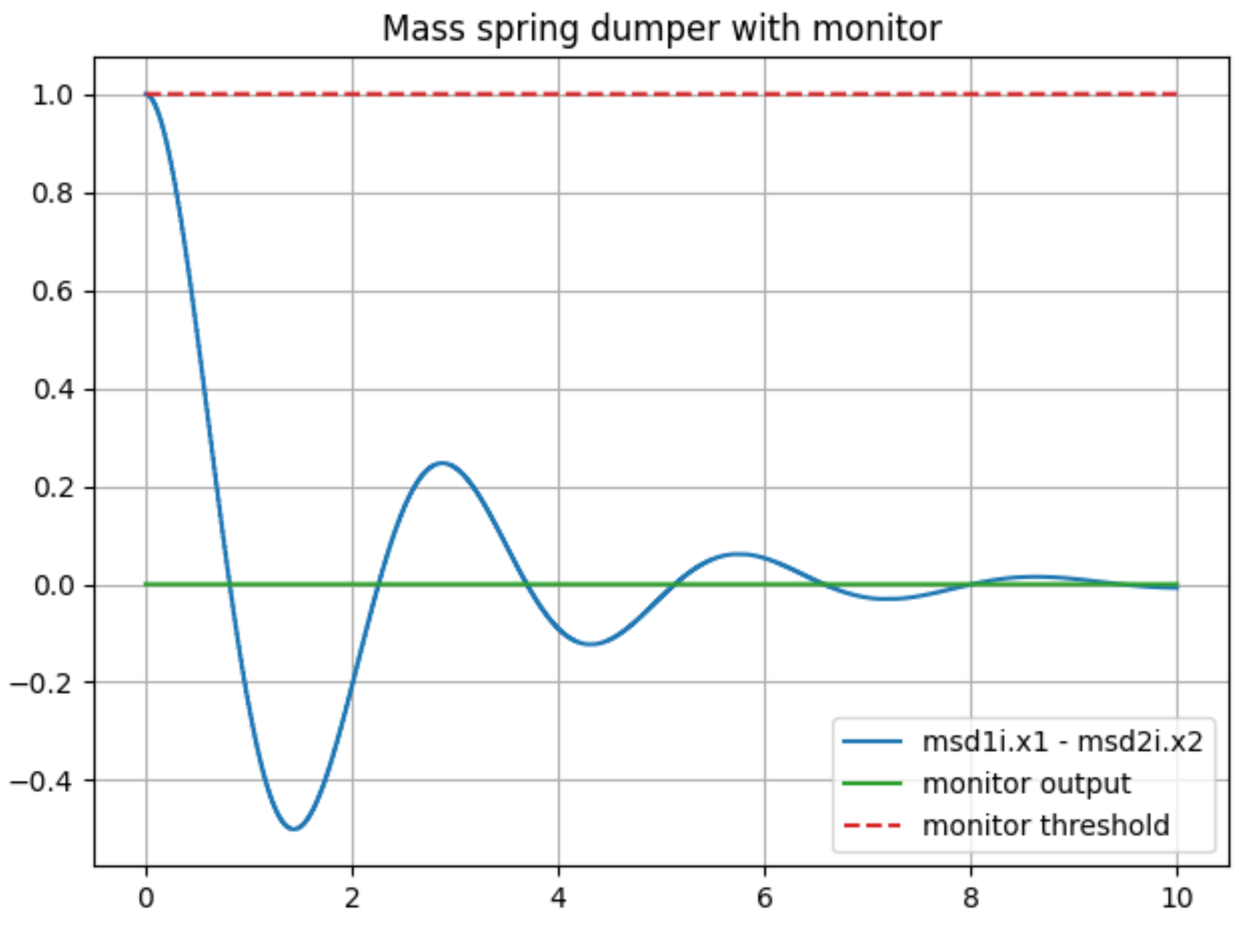
```
lifecycle/execute
```

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.

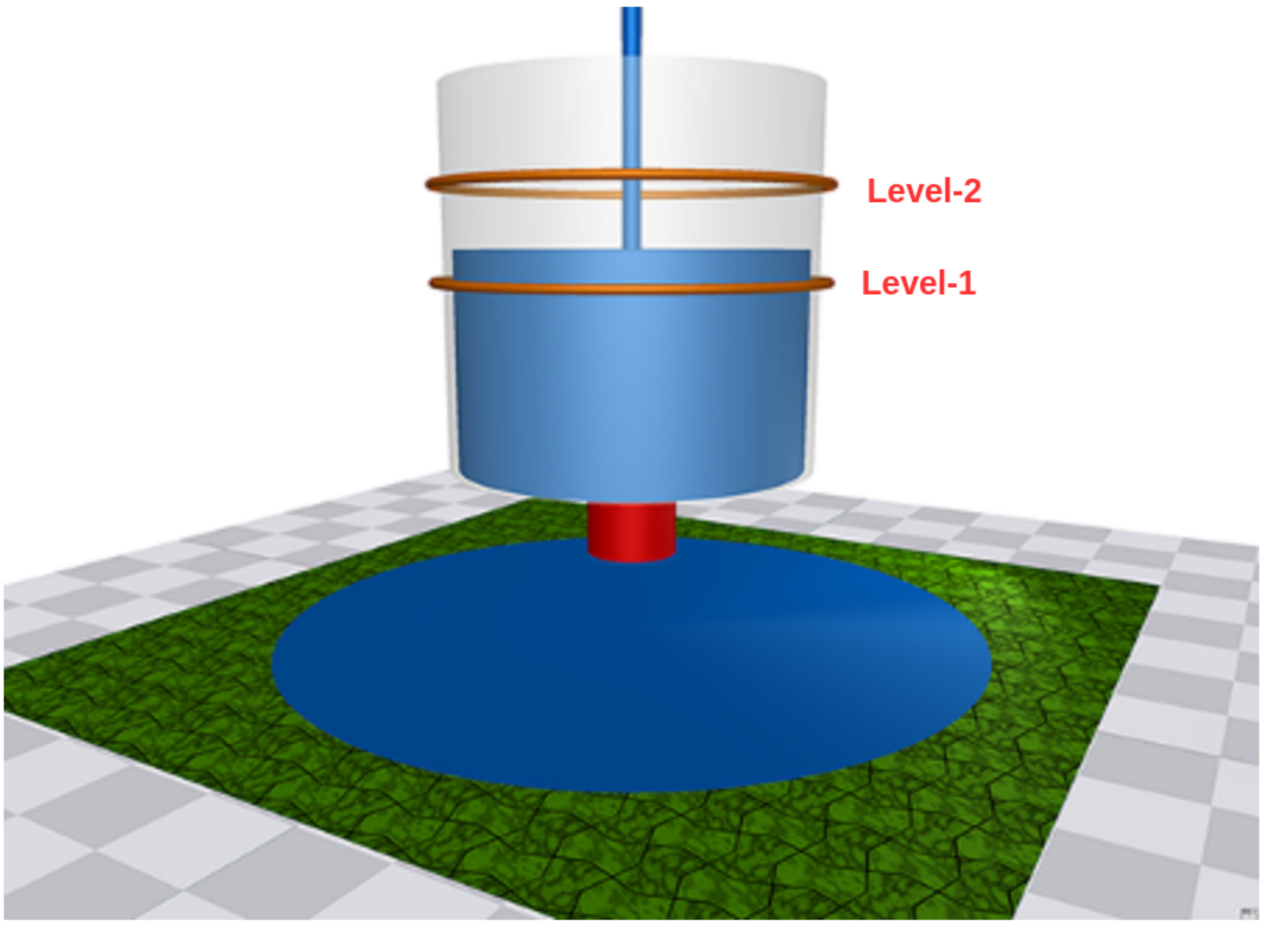
2.8.12 Water Tank Fault Injection with NuRV Runtime Monitor

Overview

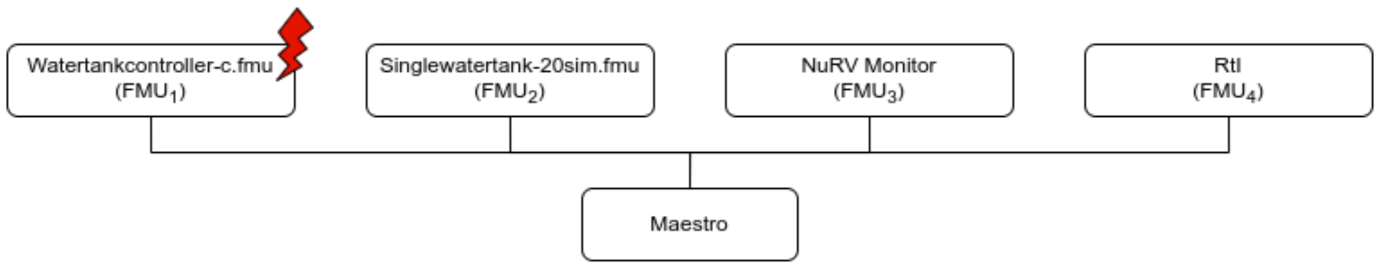
This example demonstrates a fault injection (FI) enabled digital twin. A live DT is subjected to simulated faults received from the environment. The simulated faults are specified as part of DT configuration and can be changed for new instances of DTs. This version of the example is expanded with a monitor 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. These two files can be modified to customize the DT for specific requirements.

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 installed. 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`.

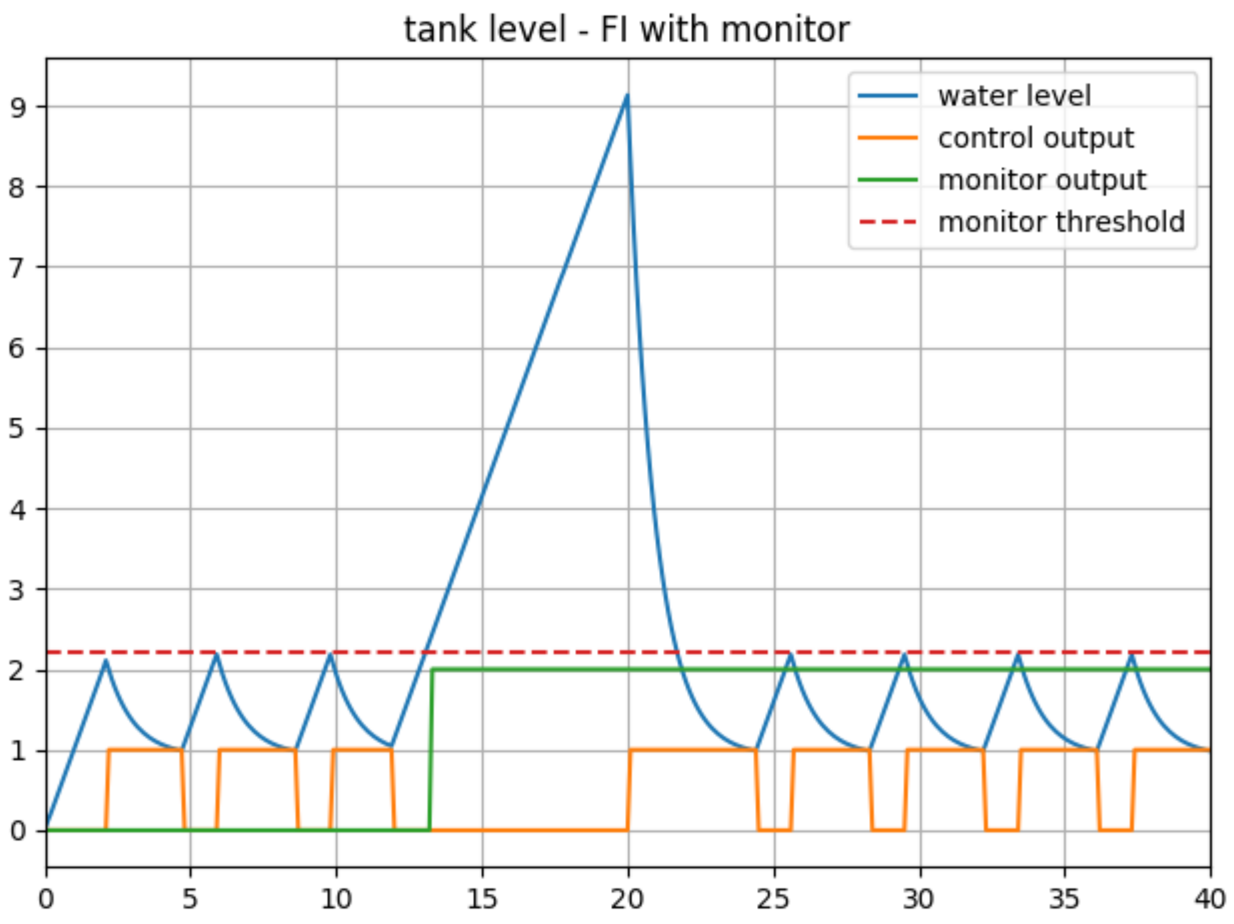
```
lifecycle/execute
```

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

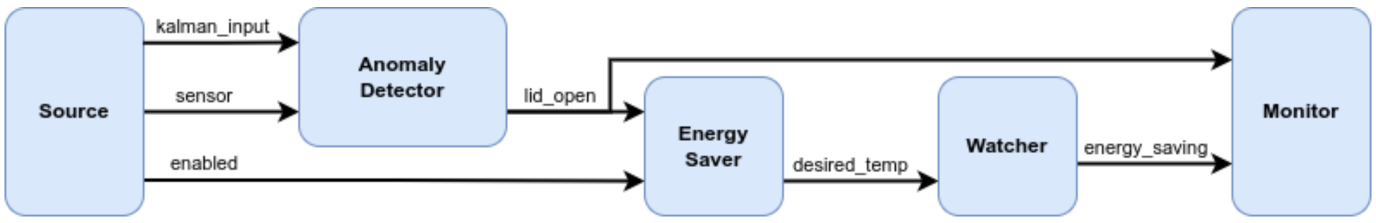
This example validates some components of the Incubator digital twin, by performing a simulation in which the commponents are wrapped inside FMUs, and are then inspected at runtime by some a FMU monitor generated by NuRV. Please note that the link to Incubator digital twin is only provided to know the 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.

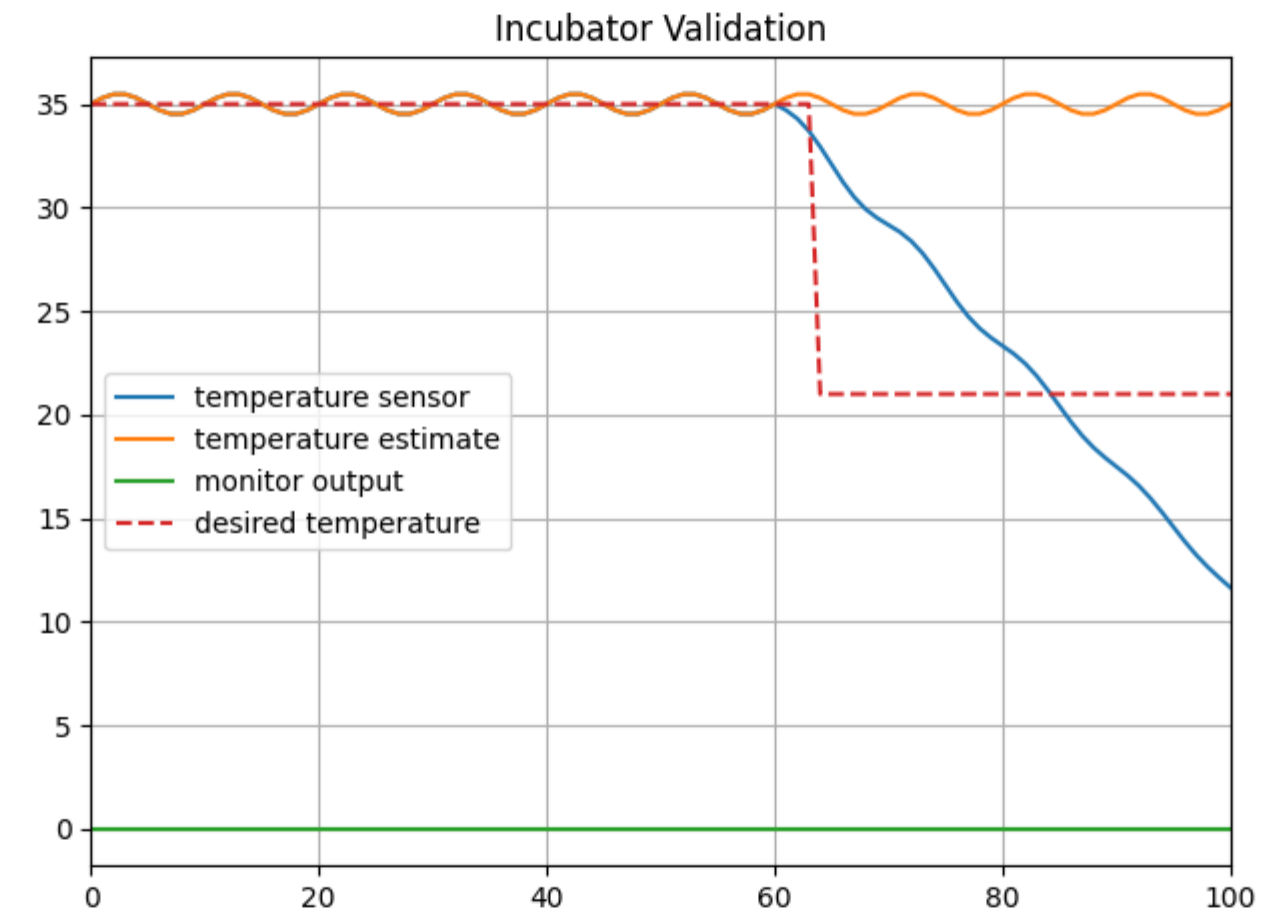
```
lifecycle/execute
```

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.

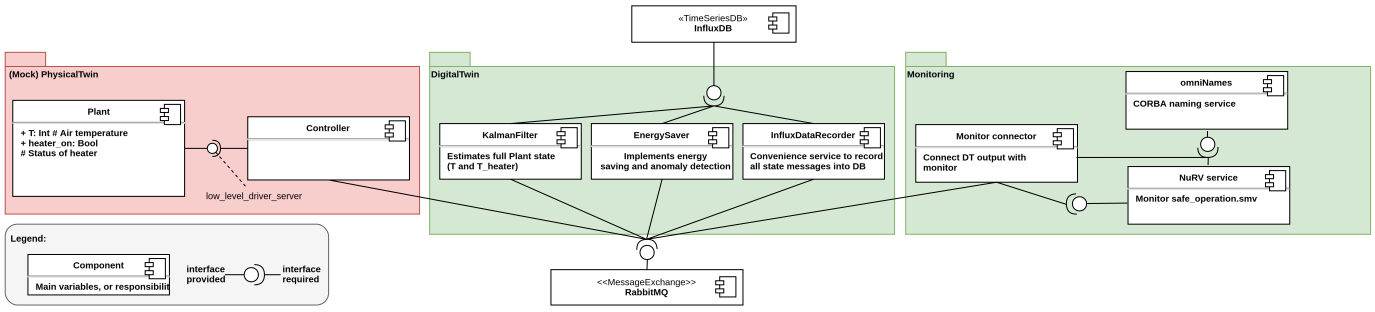
From a monitoring perspective, we wish to verify that within 3 simulation steps of an anomaly detection, the energy saving mode is turned on. To verify this behavior, we construct the property: . Whenever a True or False verdict is produced by the monitor, it is reset, allowing for the detection of repeated 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, the *simulation.conf* file should be configured with the appropriate 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:

.

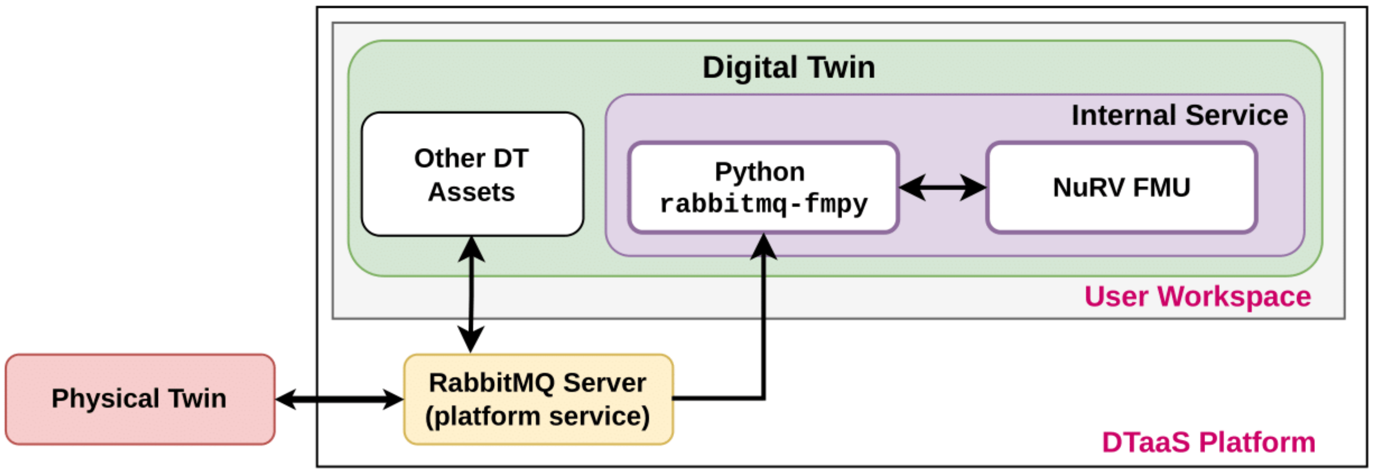
The monitor will output the *unknown* state as long as the property is satisfied and will transition to the *false* state once a violation is detected.

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, the *simulation.conf* file should be configured with the appropriate 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 objective is to install and administer the DTaaS platform for users.

Warning

The DTaaS platform has been developed and tested on docker CE v28. The software does not work on docker CE v29 yet.

The DTaaS platform 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, the installation setup that fits specific needs should be selected.

Installation Setup	Purpose
localhost	Installation of the DTaaS on a local computer for a single user; does not require a web server. *This setup does not require a domain name.*
secure localhost	Installation of the DTaaS on a local computer for a single user over HTTPS with integrated GitLab installation; does not require a web server. *This setup does not require a domain name.*
Server	Installation of the DTaaS on a server for multiple users. The requirements should be reviewed. Hosting over HTTPS with integrated GitLab installation is also available.
One vagrant machine	Installation of the DTaaS on a virtual machine; can be used for single or multiple users.
Two vagrant machines	Installation of the DTaaS on two virtual machines; can be used for single or multiple users.
	The core DTaaS platform is installed on the first virtual machine, and all services (RabbitMQ, MQTT, InfluxDB, Grafana and MongoDB) are installed on the second virtual machine.
Independent Packages	Can be used independently; does not require full installation of the DTaaS.

The installation steps is a recommended starting point for the installation process.

Administer

A CLI is available for adding and deleting users of a running application.

3.1.2 Installation Steps

Complete the DTaaS Platform

The DTaaS platform is available in two flavors. One is **localhost**, which is suitable for single-user, local usage. The other 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 2.0 authorization for both the react client frontend and backend services.

A default frontend authorization application is configured for all localhost installations, and backend authorization is not required for localhost installation.

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

CONFIGURE COMPONENTS

DTaaS is available as a docker compose application. Four docker compose files are provided:

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

INSTALL

The installation instructions on either the localhost or production server pages should be followed.

Independent Packages

Each release of the DTaaS also includes four 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.

Two optional requirements exist for installing the DTaaS.

OAuth 2.0 Provider

The DTaaS platform uses OAuth 2.0 for user authorization. It is possible to use either *gitlab.com* or a custom OAuth 2.0 service provider.

Domain name

The DTaaS platform is a web application and is preferably hosted on a server with a domain name such as *foo.com*. However, installation on a local computer with access at *localhost* is also supported.

3.1.4 Authorization

OAuth 2.0 for React Client

To enable user authorization on the DTaaS React client website, the OAuth 2.0 authorization protocol is used, specifically the PKCE authorization flow. The following steps describe the setup process:

1. Choose a GitLab Server:

- OAuth 2.0 authorization must be set up on a GitLab server. An on-premise GitLab installation is preferrable to commercial https://gitlab.com.
- The GitLab Omnibus Docker can be used for this purpose.
- The OAuth 2.0 application should be configured as an instance-wide authorization type.

2. Determine the Website's Hostname:

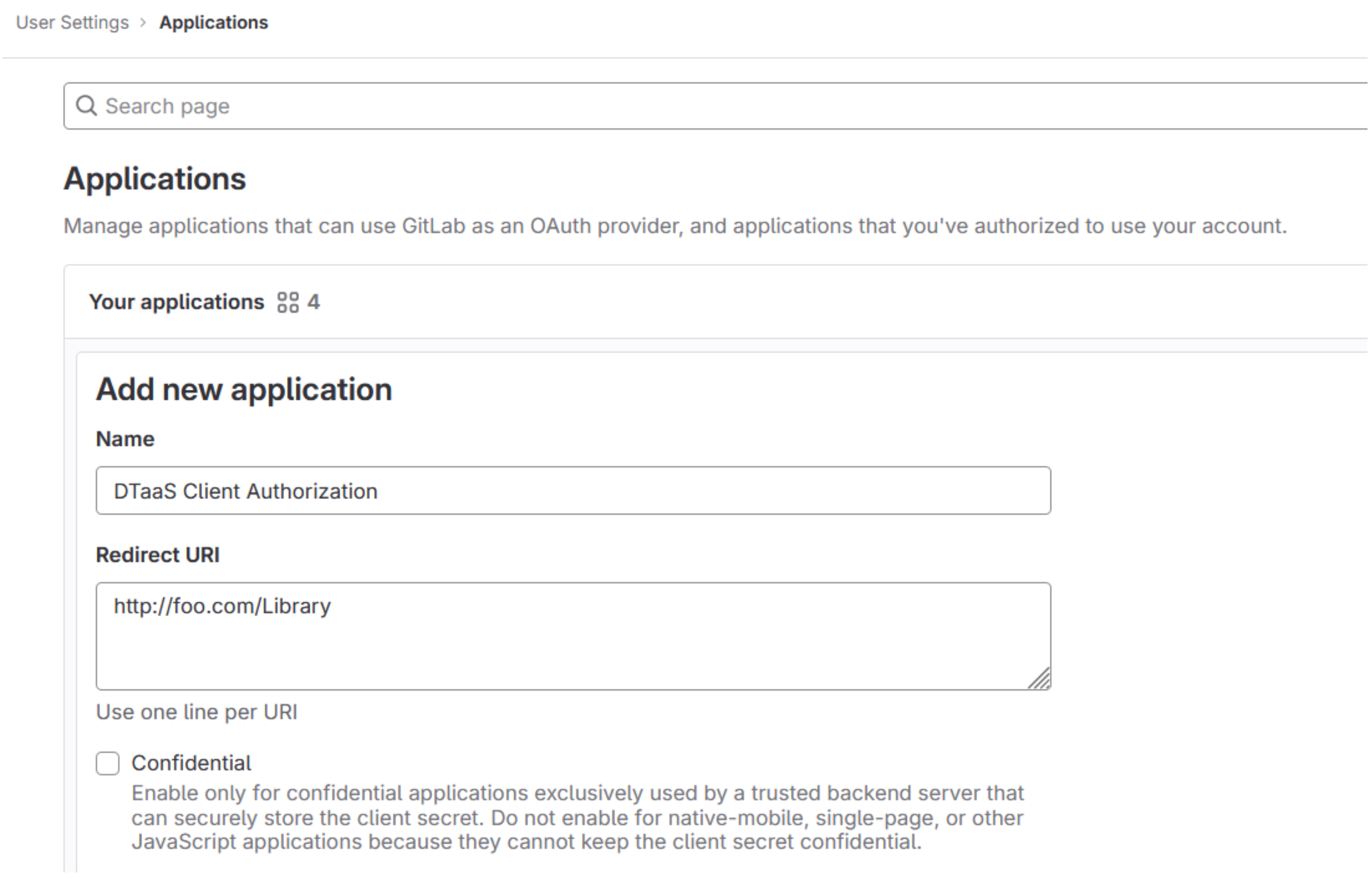
- Before setting up OAuth on GitLab, the hostname for the website must be determined. Using a self-hosted GitLab instance is recommended, which will be used in other parts of the DTaaS platform.

3. Define Callback and Logout URLs:

- For the PKCE authorization flow to function correctly, two URLs are required: a callback URL and a logout URL.
- The callback URL informs the OAuth 2.0 provider of the page where signed-in users should be redirected. It differs from the landing homepage of the DTaaS platform.
- The logout URL specifies where users will be directed after logging out.

4. OAuth 2.0 Application Creation:

- During the creation of the OAuth 2.0 application on GitLab, the scope must be specified. The openid, profile, read_user, read_repository, and api scopes should be selected.


User Settings > Applications

Search page

Applications

Manage applications that can use GitLab as an OAuth provider, and applications that you've authorized to use your account.

Your applications 4

Add new application

Name

DTaaS Client Authorization

Redirect URI

http://foo.com/Library

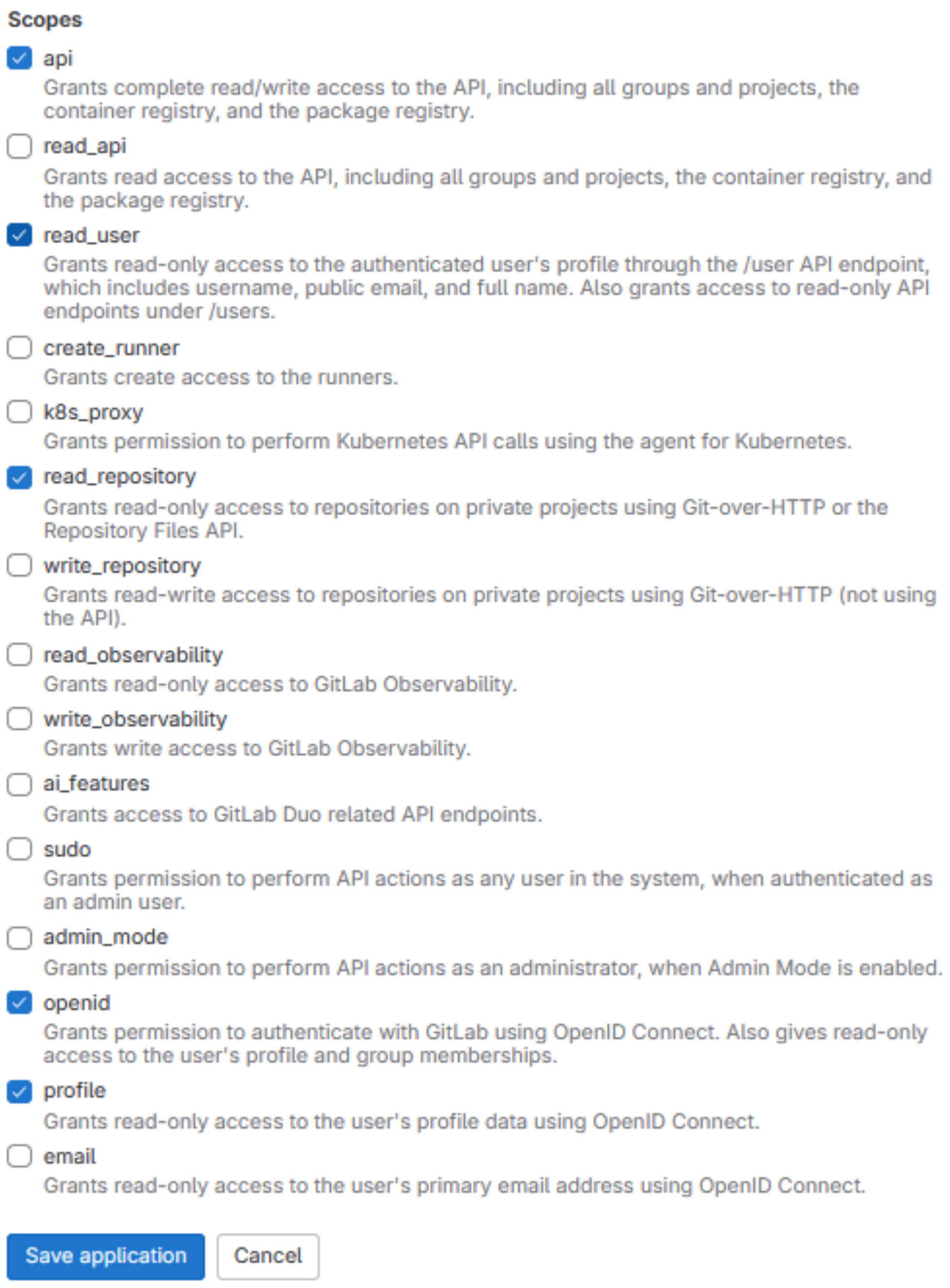
Use one line per URI

☐ Confidential

Enable only for confidential applications exclusively used by a trusted backend server that can securely store the client secret. Do not enable for native-mobile, single-page, or other JavaScript applications because they cannot keep the client secret confidential.

5. Application ID:

- After successfully creating the OAuth 2.0 application, GitLab generates an application ID. This is a long string of HEX values required for the configuration files.



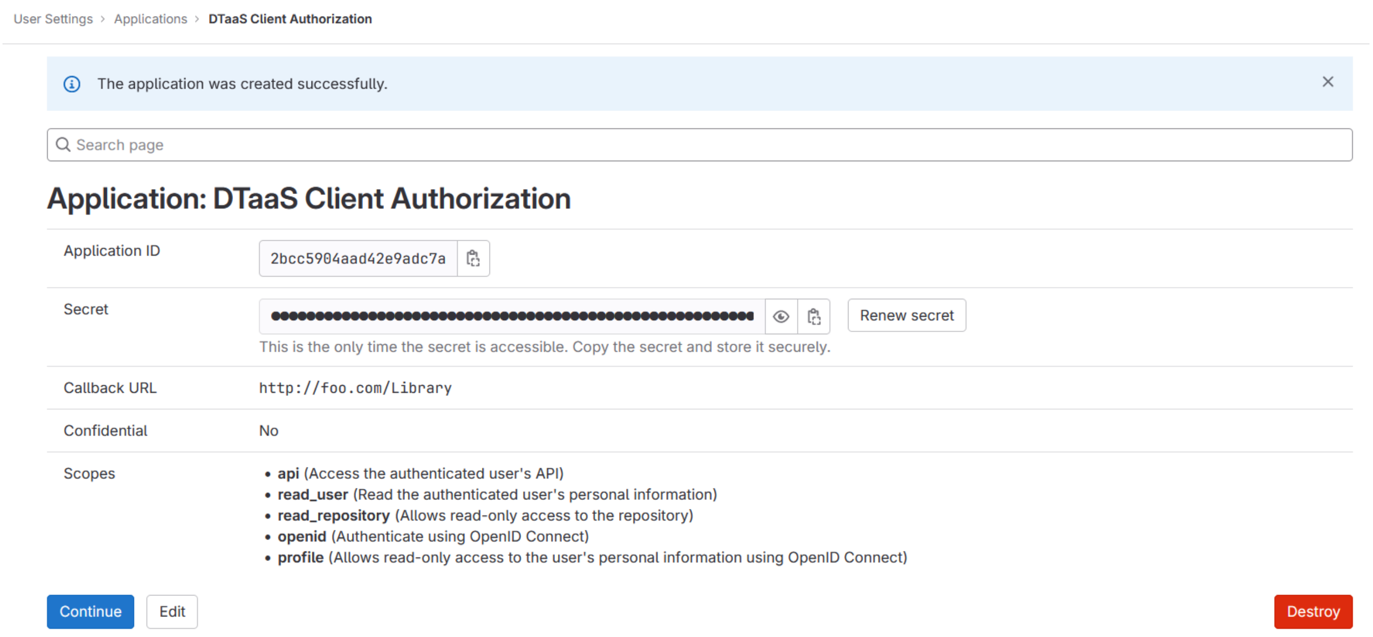
Scopes

- [✓] api
  Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.
- [ ] read_api
  Grants read access to the API, including all groups and projects, the container registry, and the package registry.
- [✓] read_user
  Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.
- [ ] create_runner
  Grants create access to the runners.
- [ ] k8s_proxy
  Grants permission to perform Kubernetes API calls using the agent for Kubernetes.
- [✓] read_repository
  Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.
- [ ] write_repository
  Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).
- [ ] read_observability
  Grants read-only access to GitLab Observability.
- [ ] write_observability
  Grants write access to GitLab Observability.
- [ ] ai_features
  Grants access to GitLab Duo related API endpoints.
- [ ] sudo
  Grants permission to perform API actions as any user in the system, when authenticated as an admin user.
- [ ] admin_mode
  Grants permission to perform API actions as an administrator, when Admin Mode is enabled.
- [✓] openid
  Grants permission to authenticate with GitLab using OpenID Connect. Also gives read-only access to the user's profile and group memberships.
- [✓] profile
  Grants read-only access to the user's profile data using OpenID Connect.
- [ ] email
  Grants read-only access to the user's primary email address using OpenID Connect.

6. Required Information from OAuth 2.0 Application:

- The following information from the OAuth 2.0 application registered on GitLab is required:

GitLab Variable Name	Variable Name in Client env.js	Default Value
OAuth 2.0 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



The application was created successfully.

Application: DTaaS Client Authorization

Application ID	2bcc5904aad42e9adc7a
Secret	●● This is the only time the secret is accessible. Copy the secret and store it securely.
Callback URL	http://foo.com/Library
Confidential	No
Scopes	• **api** (Access the authenticated user's API) • **read_user** (Read the authenticated user's personal information) • **read_repository** (Allows read-only access to the repository) • **openid** (Authenticate using OpenID Connect) • **profile** (Allows read-only access to the user's personal information using OpenID Connect)

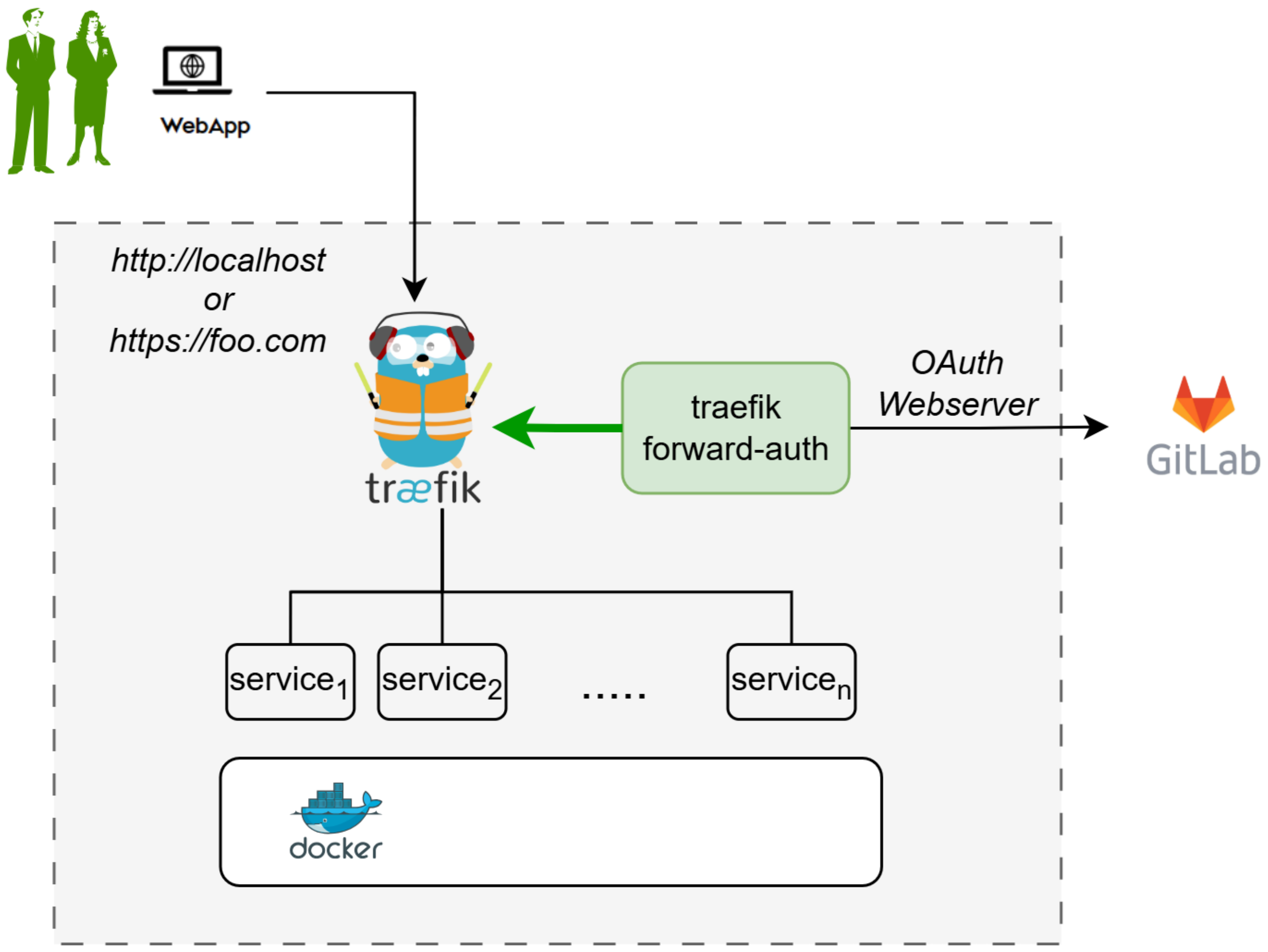
7. Create User Accounts:

User accounts must be created in GitLab for all usernames chosen during installation. The *trial* installation script includes two default usernames - *user1* and *user2*. For all other installation scenarios, accounts with specific usernames must be created on GitLab.

OAuth 2.0 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 OAuth 2.0 provider, but within the DTaaS GitLab is used as authorization provider. The OAuth 2.0 web / server application authorization flow is utilized.

The following steps outline the configuration process:

1. Choose GitLab Server:

- OAuth 2.0 authorization must be set up on a GitLab server. An on-premise GitLab installation is preferrable to commercial https://gitlab.com.
- The GitLab Omnibus Docker can be used for this purpose.
- The OAuth 2.0 application should be configured as an instance-wide authorization type. The options to generate client secret and trusted application should be selected.

2. Determine Website Hostname:

Before setting up OAuth 2.0 on GitLab, the hostname for the website should be determined. A self-hosted GitLab instance is recommended, which can be used in other parts of the DTaaS platform.

3. Determine Callback and Logout URLs:

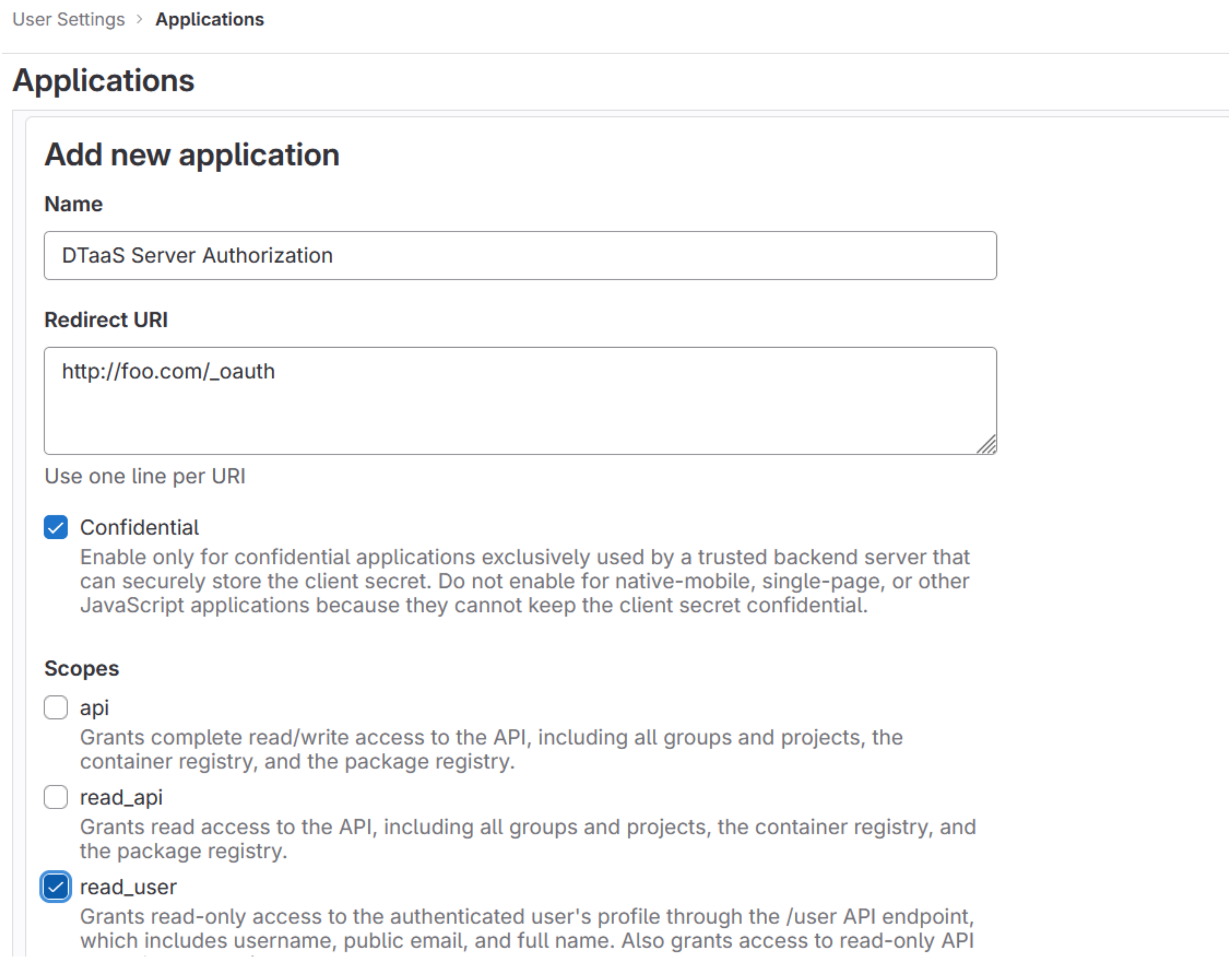
For the web / server authorization flow to function correctly, two URLs are required: a *callback URL* and a *logout URL*.

- The callback URL informs the OAuth 2.0 provider of the page where signed-in users should be redirected. It represents the landing homepage of the DTaaS platform. (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 2.0 application setup.

4. Create OAuth 2.0 Application:

OAuth 2.0 application setup on GitLab can be located at Edit **Profile** -> **Application** https://gitlab.foo.com/-/profile/applications.

During the creation of the OAuth 2.0 application on GitLab, the scope must be specified. The ***read_user*** scope should be selected.


User Settings > Applications

Applications

Add new application

Name

DTaaS Server Authorization

Redirect URI

http://foo.com/_oauth

Use one line per URI

[✓] Confidential
Enable only for confidential applications exclusively used by a trusted backend server that can securely store the client secret. Do not enable for native-mobile, single-page, or other JavaScript applications because they cannot keep the client secret confidential.

Scopes

[ ] api
Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.

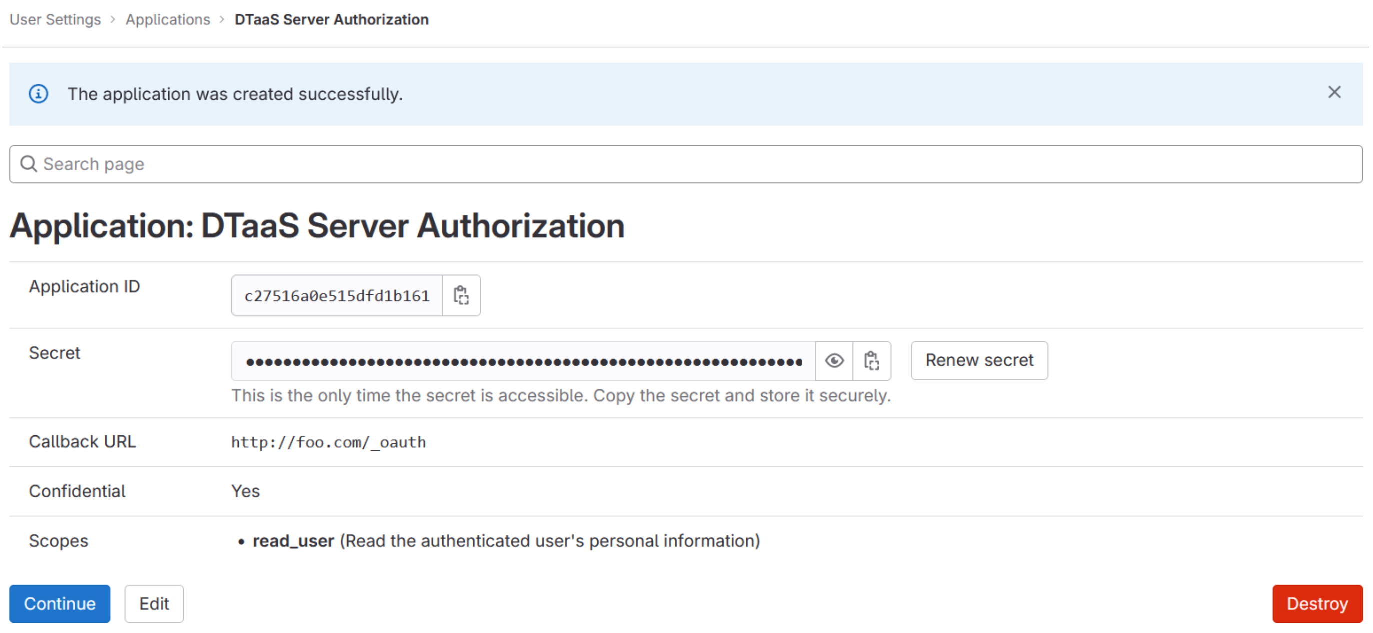
[ ] read_api
Grants read access to the API, including all groups and projects, the container registry, and the package registry.

[✓] read_user
Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API

5. Copy Application Credentials:

After successfully creating the OAuth 2.0 application, GitLab generates an *application ID* and *client secret*.

Both these values are long strings of HEX values that are required for the configuration files.


User Settings > Applications > DTaaS Server Authorization

The application was created successfully.

Search page

Application: DTaaS Server Authorization

Application ID	c27516a0e515dfd1b161
Secret	•• Renew secret This is the only time the secret is accessible. Copy the secret and store it securely.
Callback URL	http://foo.com/_oauth
Confidential	Yes
Scopes	• **read_user** (Read the authenticated user's personal information)

6. Checklist: Required Information from OAuth 2.0 Application:

The following information is required from the OAuth 2.0 application registered on GitLab:

GitLab Variable Name	Variable Name in .env of docker compose file	Default Value
OAuth 2.0 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 2.0 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 2.0, 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 2.0 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.local.yml --env-file .env up \
  -d --force-recreate traefik-forward-auth
```

- For a server instance running in HTTP mode:

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

- For a server instance running in HTTPS mode:

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

3.1.5 Configuration

Configure Client Website

This page describes the various configuration options for the 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_LIBRARY_PREVIEW: 'Endpoint for the Library page preview',
    REACT_APP_WORKBENCHLINK_DT_PREVIEW: "Endpoint for the Digital Twins page preview",
    REACT_APP_CLIENT_ID: 'AppID genereated by the gitlab OAuth 2.0 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 2.0 scopes. These should match with the scopes set in gitlab OAuth 2.0 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_LIBRARY_PREVIEW: '/preview/library',
    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_LIBRARY_PREVIEW: '/preview/library',
    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',
  };
};
```

Configure Library Microservice

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

```yaml
port: '4001'
mode: 'local' or 'git'
local-path: '/home/Desktop/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 relative filepath to the location of the local directory which will be served to users by the Library microservice.

The default values should be replaced with appropriate values for the deployment. This configuration should be saved as a YAML file, for example as `libms.yaml`.

OPERATION MODES

The mode indicates the backend storage for the files. There are two possible modes - `local` and `git`. The files available in the `local-path` are served to users in `local` mode. In the `git` mode, the remote git repos are cloned and they are served to users as local files.

git mode

A fragment of the config for `git` mode is:

```yaml
...
git-repos:
  - user1:
      repo-url: 'https://gitlab.com/dtaas/user1.git'
  - user2:
      repo-url: 'https://gitlab.com/dtaas/user2.git'
  - common:
      repo-url: 'https://gitlab.com/dtaas/common.git'
```

Here, `user1`, `user2` and `common` are the local directories into which the remote git repositories get cloned. The name of the repository need not match with the local directory name. For example, the above configuration enables library microservice to clone `https://gitlab.com/dtaas/user1.git` repository into `user1` directory. Any git server accessible over HTTP(S) protocol is supported. The `.git` suffix is optional.

The default values should be replaced with appropriate values for the deployment.

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>`.

Further documentation on the use of library microservice is available on this page.

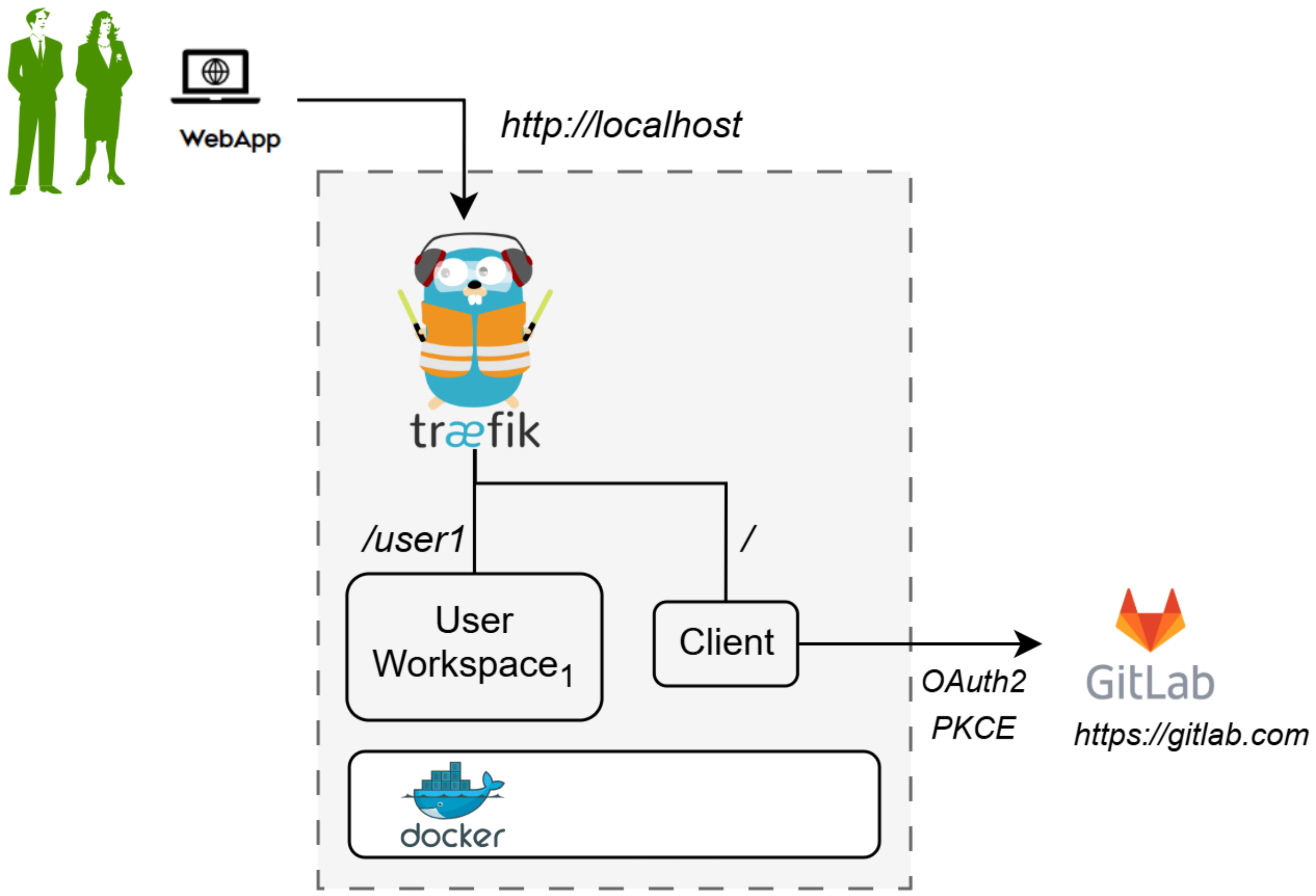
3.1.6 Docker

Install DTaaS on localhost

The installation instructions provided in this document are suitable for running DTaaS on localhost. This installation is intended for single users running DTaaS on their own computers.

DESIGN

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



🗒️ The text starting with `/` at the beginning indicates the URL route at which a certain service is available. For example, the user workspace is available at http://localhost/user1.

REQUIREMENTS

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

- docker desktop / docker CE v28.
- User account on GitLab

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

DOWNLOAD PACKAGE

The software is available as a zip package. The package should be downloaded and unzipped. A new **DTaaS-v0.8.0** folder is created. The remaining installation instructions assume the use of a Windows/Linux/MacOS terminal in the **DTaaS-v0.8.0** folder.

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 file paths mentioned below are relative 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 the local installation scenario. It contains the following environment variables.

All fields should be edited according to the 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.
username1	'user1'	The GitLab username

📋 Important points to note:

1. The path examples given here are for Linux OS. These paths can also be Windows OS compatible paths.
2. The client configuration file is located at `deploy/config/client/env.local.js`. Beyond this, modification of this file is not necessary.

Create User Workspace

The existing filesystem for installation is configured for `user1`. A new filesystem directory must be created for the selected user.

The following commands should be executed from the top-level directory of the DTaaS project.

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

where *username* is the selected username registered on GitLab.

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 a web browser. Sign in using a GitLab account.

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

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

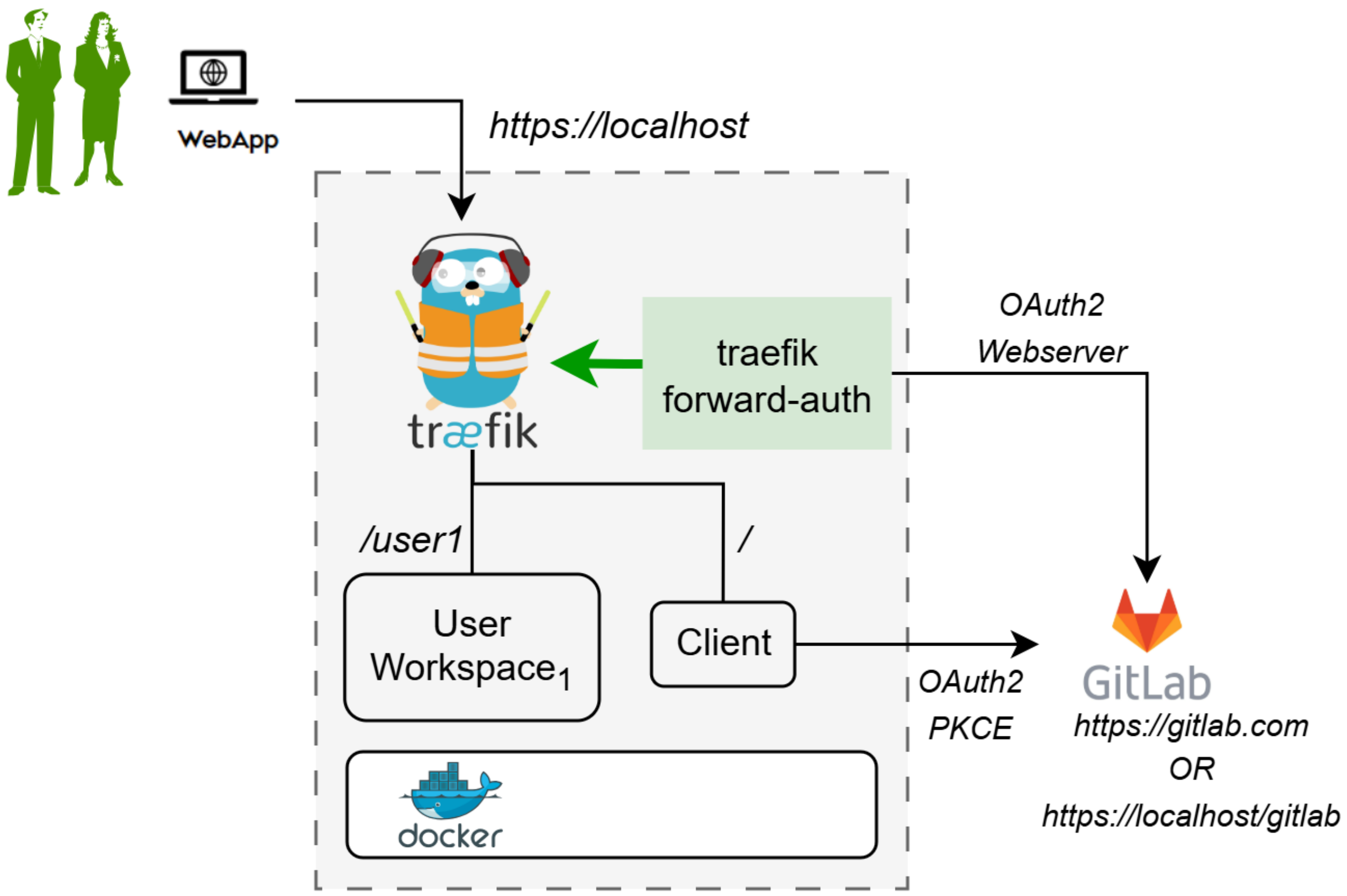
This installation is suitable for single users intending to use DTaaS on their own computers.

The installation instructions provided in this document are appropriate 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 GitLab running on localhost is not required, the simpler localhost setup should be used.

DESIGN

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



🗒️ The text starting with `/` at the beginning indicates the URL route at which a certain service is available. For example, user workspace is available at https://localhost/user1.

REQUIREMENTS

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

- docker desktop / docker CLI with compose plugin
- mkcert

DOWNLOAD PACKAGE

The software is available as a zip package. The package should be downloaded and unzipped. A new **DTaaS-v0.8.0** folder is created. The remaining installation instructions assume the use of a Windows/Linux/MacOS terminal in the **DTaaS-v0.8.0** folder.

📋 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

Create User Workspace

The existing filesystem for installation is configured for `user1`. A new filesystem directory must be created for the selected user.

The following commands should be executed 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

mkcert can be used to generate TLS certificates following this guide. The certificates should 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.

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.

All fields should be edited according to the specific deployment 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.
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.

RUN

Start DTaaS to Integrate GitLab

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
```

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 OAuth 2.0 Application

The frontend website requires OAuth 2.0 application registration on the integrated GitLab. The details of OAuth 2.0 application for the frontend website are available in client docs.

The default OAuth 2.0 client application provided in `env.local.js` functions correctly. However, when running an integrated GitLab instance, this application needs to be created on GitLab running at https://localhost/gitlab.

https://localhost/Library should be used as the Callback URL (`REACT_APP_REDIRECT_URI`).

The GitLab OAuth 2.0 provider documentation provides further guidance on creating this OAuth 2.0 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_WORKBENCHLINK_LIBRARY_PREVIEW: '/preview/library',
    REACT_APP_WORKBENCHLINK_DT_PREVIEW: '/preview/digitaltwins',

    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 OAuth 2.0 client application ID (`REACT_APP_CLIENT_ID`) with that of the newly registered OAuth 2.0 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 a web browser. Users can sign in using their https://localhost/gitlab account.

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

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

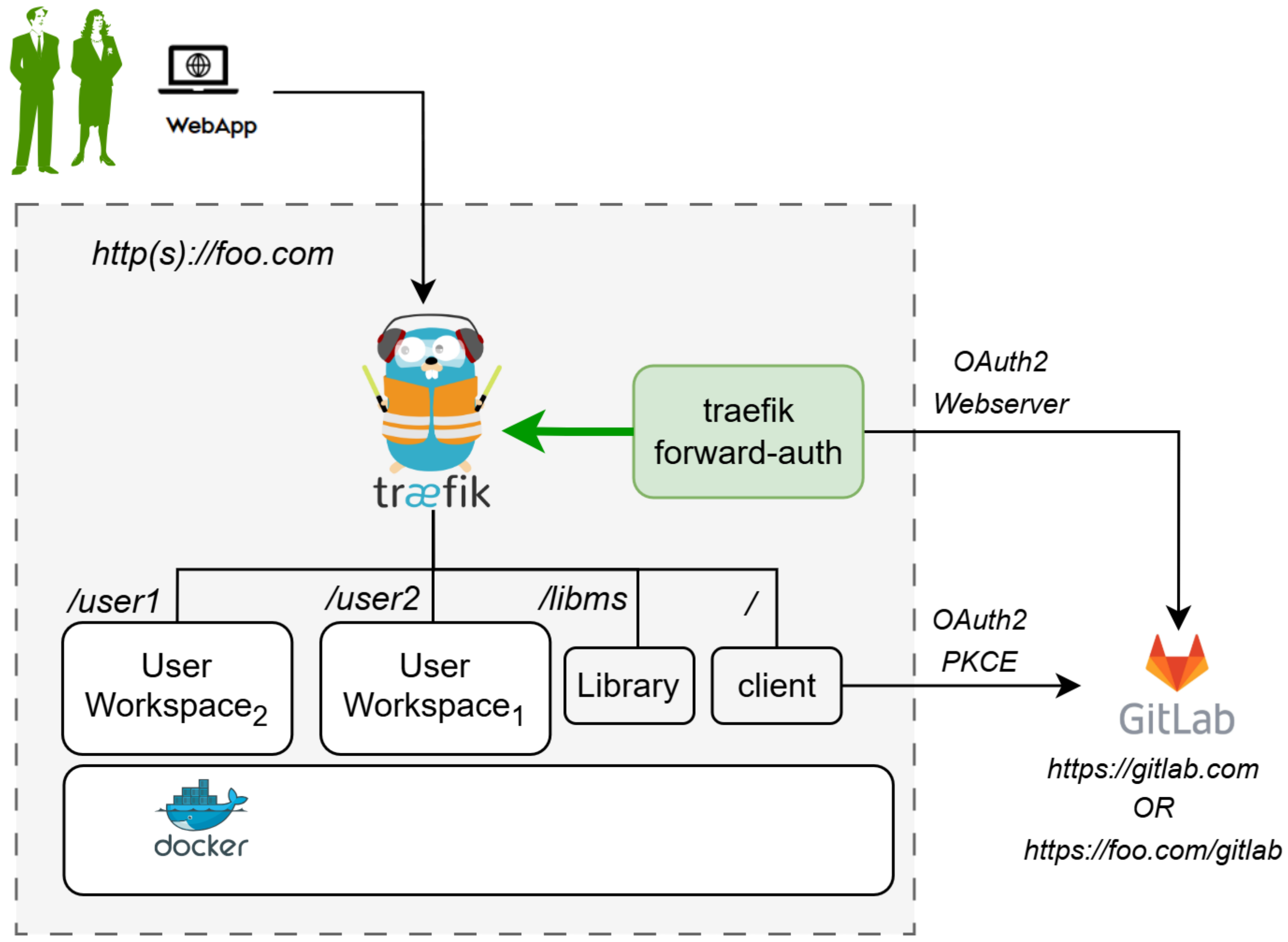
Image sources: Traefik logo, ml-workspace, reactjs, GitLab

Install DTaaS on a Production Server

The installation instructions provided in this document 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 presented here.



🗒️ The text starting with `/` at the beginning indicates the URL route at which a certain service is available. For example, user workspace is available at https://localhost/user1.

In the new application configuration, there are two OAuth 2.0 applications.

REQUIREMENTS

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

Docker with Compose Plugin

Docker installation is mandatory. Docker must be installed on the host computer.

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)

HTTPS functionality can be added to the DTaaS software installation. The required TLS certificates can be created through certbot.

OAuth 2.0 Provider

GitLab Instance - The DTaaS uses GitLab OAuth 2.0 authorization for user authorization. Either an on-premise instance of GitLab can be used, or 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.

OAuth 2.0 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 OAuth 2.0 protocol.

- The frontend website is a React single page application (SPA).
- The details of OAuth 2.0 application for the frontend website are in client docs.
- The OAuth 2.0 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 OAuth 2.0 applications and link them to the intended DTaaS installation.

The GitLab OAuth 2.0 provider documentation provides further guidance on creating these two OAuth 2.0 applications.

DOWNLOAD PACKAGE

The software is available as a zip package. The package should be downloaded and unzipped. A new **DTaaS-v0.8.0** folder is created. The remaining installation instructions assume the use of a Windows/Linux/MacOS terminal in the **DTaaS-v0.8.0** folder.

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.

All fields should be edited according to the specific deployment 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 2.0 application
OAUTH_CLIENT_SECRET	'xx'	The Secret of your server OAuth 2.0 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 2.0 application registered on https://gitlab.com for client. The corresponding OAuth 2.0 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 2.0 application for secure production deployments.

Create User Workspace

The existing filesystem for installation is configured for `files/user1`. A new filesystem directory must be created for the selected user.

The following commands should be executed 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
```

The usernames and email addresses should be changed to match the user accounts on the OAuth 2.0 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/libms.yaml:/dtaas/libms/libms.yaml
      - ${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 a web browser. Users can sign in using accounts linked to either *gitlab.com* or the local GitLab instance.

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

Users may need to click Sign in to GitLab on the Client page and authorize access to the displayed 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 document provides instructions for creating a custom Operating System virtual disk for running the DTaaS platform. The virtual disk is managed by **vagrant**. The purpose is two-fold:

- Provide cross-platform installation of the DTaaS platform. Any operating system supporting the use of vagrant software utility can support installation of the DTaaS platform.
- 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 additional software installation is desired for developers, the `Vagrantfile` needs to be modified. The existing `Vagrantfile` has two lines:

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

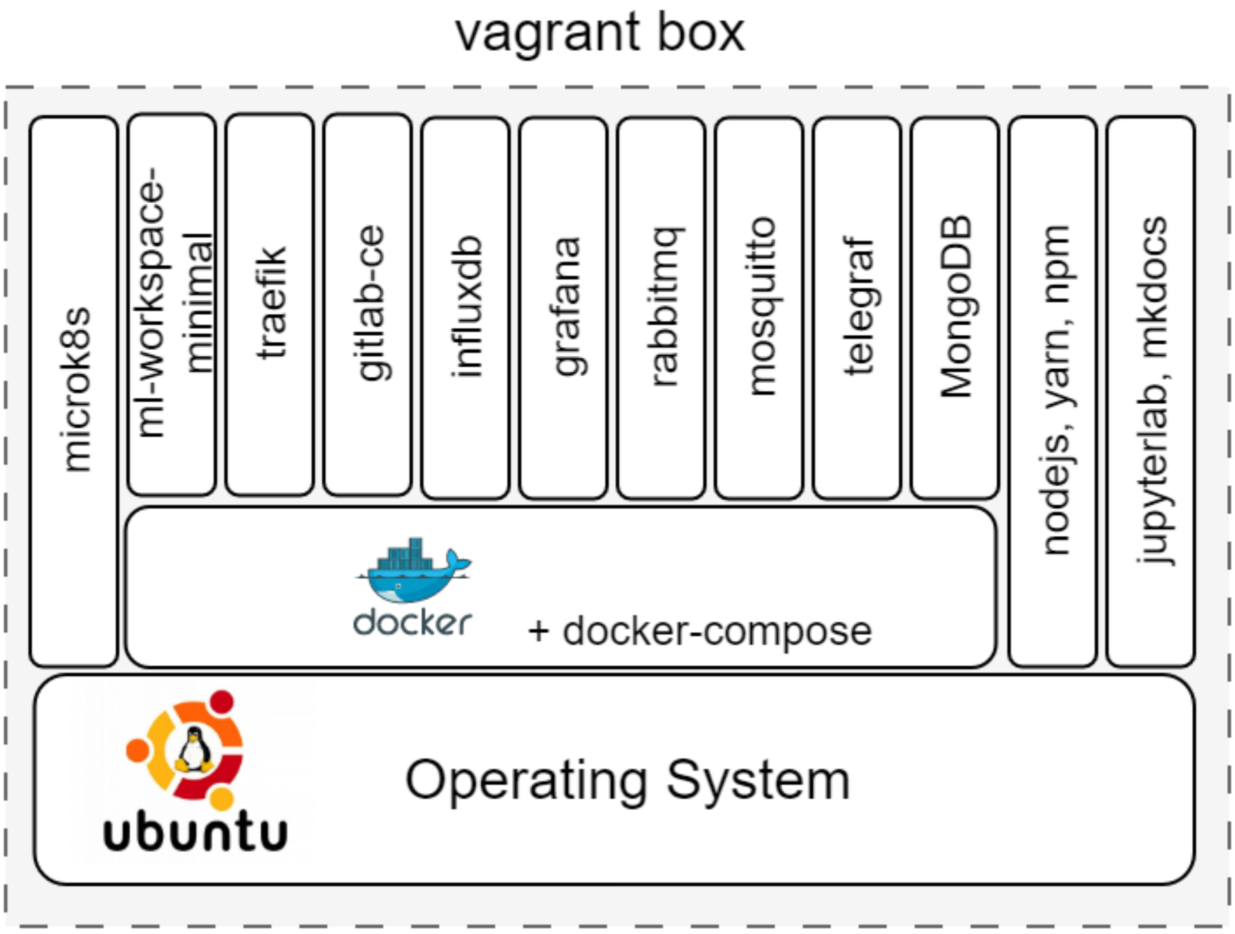
This vagrant box installed for users will have the following items:

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.


vagrant box
microk8s
ml-workspace-minimal
traefik
gitlab-ce
influxdb
grafana
rabbitmq
mosquitto
telegraf
MongoDB
nodejs, yarn, npm
jupyterlab, mkdocs
docker
+ docker-compose
ubuntu
Operating System

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

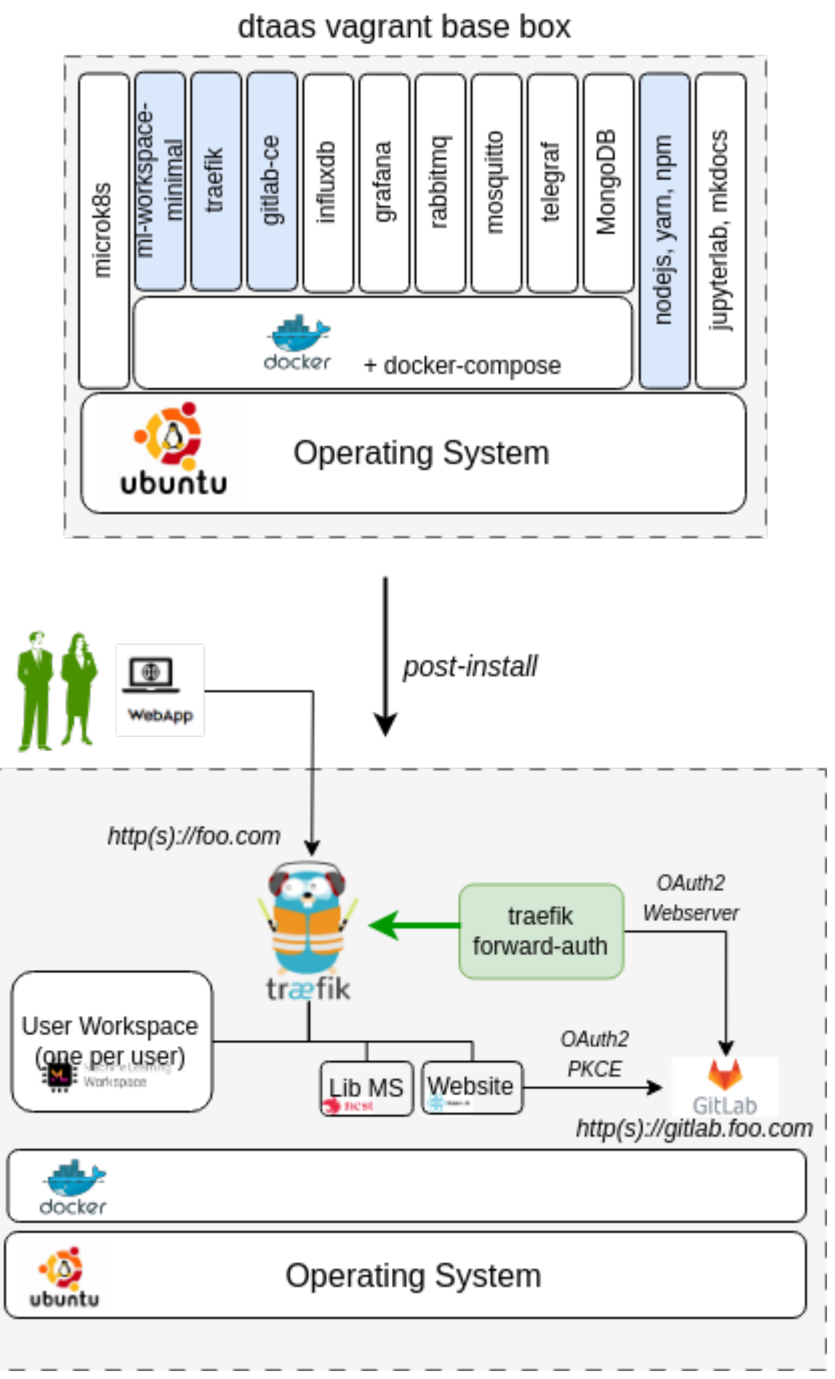
These are installation instructions for running DTaaS platform inside one Vagrant Virtual Machine. The setup requires a machine that can allocate 16GB RAM, 8 vCPUs and 50GB Hard Disk space to the vagrant box.

CREATE BASE VAGRANT BOX

Create the **dtaas** Vagrant box. An SSH key pair - *vagrant* and *vagrant.pub* - will have been created. The *vagrant* file is the private SSH key and is needed for the next steps. The *vagrant* SSH private key should be copied into the current directory (`deploy/vagrant/single-machine`). This key is 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 platform on one single vagrant machine. A graphical illustration of a successful installation is presented 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 integrated GitLab setup is documented on this page.

CONFIGURE SERVER SETTINGS

A dummy `foo.com` URL has been used for illustration. This should be changed to the actual unique website URL.

The following steps should be performed to make this installation work in the local environment.

Update the **Vagrantfile**. The 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 the DTaaS platform in two Vagrant virtual machines (VMs). In this setup, all user workspaces are run on server1 while all platform services are 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

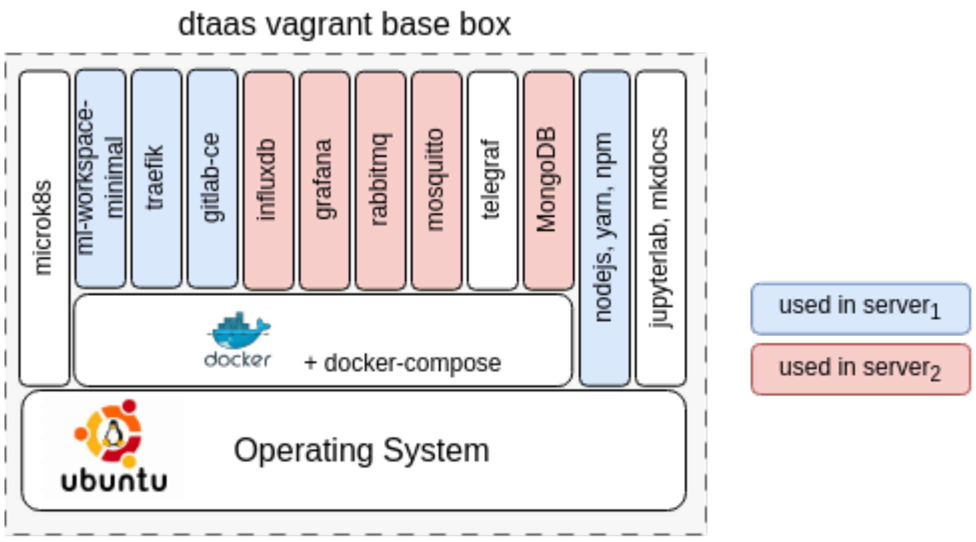
Under the default configuration, two user workspaces are provisioned on **server1**. The default installation setup also installs InfluxDB, Grafana, RabbitMQ and MQTT services on **server2**. If you would like to install more services, you can create shell scripts to install the same on **server2**.

CREATE BASE VAGRANT BOX

Create the **dtaas** Vagrant box. An SSH key pair - *vagrant* and *vagrant.pub* - will have been created. The *vagrant* file is the private SSH key and is needed for the next steps. The *vagrant* SSH private key should be copied into the current directory (`deploy/vagrant/two-machine`). This key is useful for logging into the vagrant machines created for two-machine deployment.

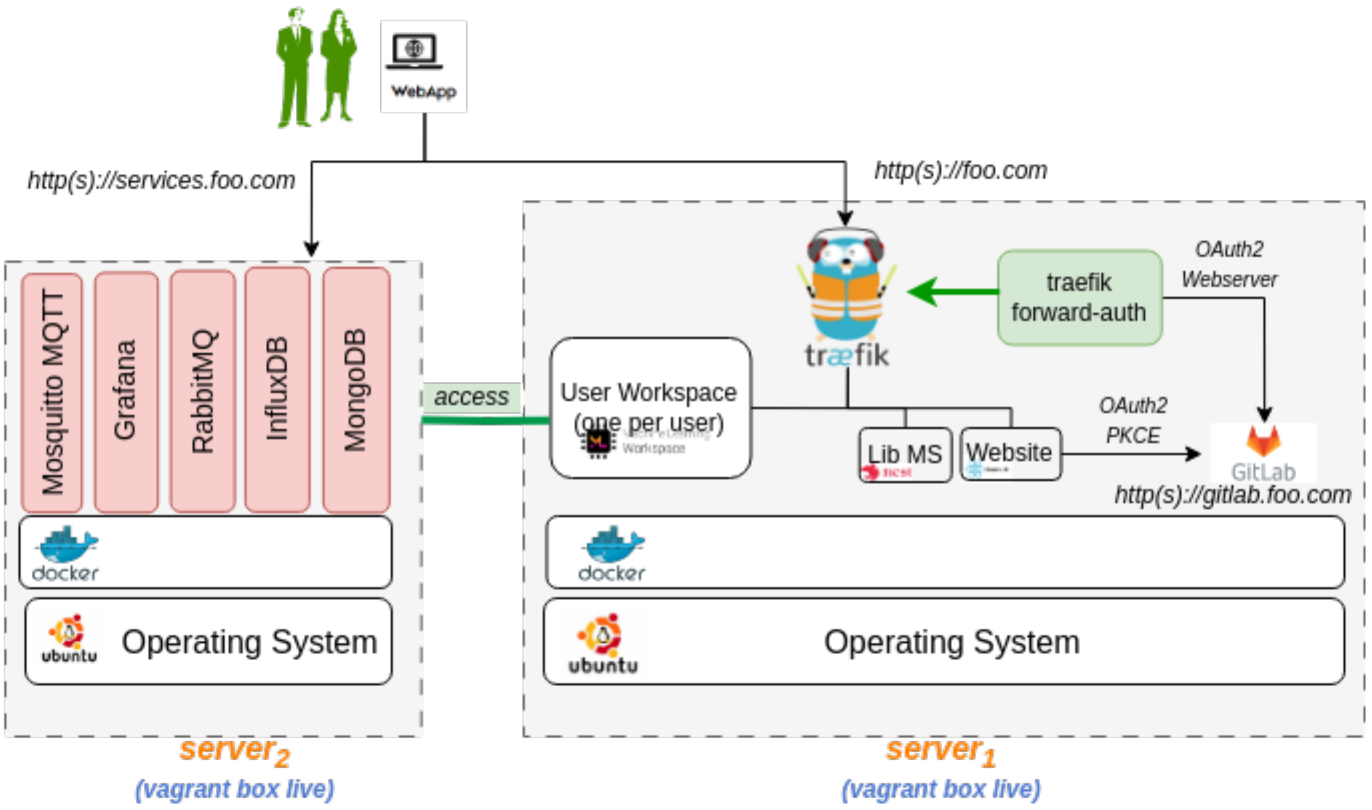
TARGET INSTALLATION SETUP

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



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

A graphical illustration of a successful installation is presented 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 have been used for illustration. These should be changed to the actual unique website URLs.

The first step is to define the network identity of the two VMs. For this, the *server name*, *hostname* and *MAC address* are required. The hostname is the network URL at which the server can be accessed on the web. The following steps should be performed to make this work in the 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 Platform

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 Platform Services

DTaaS Services CLI

A command-line tool for managing DTaaS platform services including MongoDB, InfluxDB, RabbitMQ, and Grafana.

🗒️ The CLI does not install ThingsBoard and PostgreSQL services. See the commands in manual install page for installing these two services.

FEATURES

- **Project Initialization:** Generate project structure with config and data directories
- **Automated Setup:** One command setup of TLS certificates and permissions
- **Service Management:** Start, stop, and check status of all services
- **User Management:** Easy creation of user accounts in InfluxDB and RabbitMQ
- **Cross platform:** Works on Linux, macOS, and Windows
- **Configuration-driven:** Reads settings from `config/services.env`

INSTALLATION

Prerequisites

- Python 3.10 or higher
- Docker
- TLS certificates

Install from Wheel Package

Run the following commands from a virtual environment. Install the standalone wheel package using pip:

```
cd deploy/services/cli
pip install dtaas_services-0.1.0-py3-none-any.whl
```

This installs the `dtaas-services` command.

To verify the installation:

```
dtaas-services --help
```

QUICK START

1. Navigate to where you want to set up the services and generate the project structure:

```
dtaas-services generate-project
```

This creates: * `config/` directory with configuration templates * `data/` directory for service data * `compose.services.secure.yml` for Docker Compose

1. Update `config/services.env` with your environment values:
2. `SERVICES_UID` - User ID for service file ownership
3. `SERVICES_GID` - Group ID for service file ownership
4. `SERVER_DNS` - Your server hostname
5. Port numbers for each service
6. Update `config/credentials.csv` with user accounts (format: `username,password`)

USAGE

Service Setup

After generating the project and configuring your settings:

```
dtaas-services setup
```

This command will:

- Copy TLS certificates to the correct locations
- Set up MongoDB certificates and permissions
- Set up InfluxDB certificates and permissions
- Set up RabbitMQ certificates and permissions

Permission Requirements:

This command requires access to the Docker daemon. You have two options:

1. **Recommended:** Add your user to the docker group (run once):

```
sudo usermod -aG docker $USER
newgrp docker
```

Then run the command without sudo:

```
dtaas-services setup
```

1. **Alternative:** Run with sudo:

```
sudo dtaas-services setup
```

Service Management

Start all services:

```
dtaas-services start
```

Stop all services:

```
dtaas-services stop
```

Restart services:

```
dtaas-services restart
```

Check service status:

```
dtaas-services status
```

Remove services (with confirmation prompt):

```
dtaas-services remove
```

Remove services and their volumes:

```
dtaas-services remove --volumes
```

User Account Management

1. Edit `config/credentials.csv` with user accounts (format: `username,password`)
2. Add users to InfluxDB and RabbitMQ:

```
dtaas-services user add
```

This will create user accounts with appropriate permissions in both services.

COMMANDS REFERENCE

`dtaas-services generate-project`

Generates the project structure with config, data directories, and compose file.

Options:

- `--path` - Directory to generate project structure (default: current directory)

Example:

```
dtaas-services generate-project --path /path/to/project
```

`dtaas-services setup`

Performs complete service setup including certificates and permissions.

Example:

```
dtaas-services setup
```

`dtaas-services start`

Starts all platform services using Docker Compose.

Options:

- `-s, --services` - Comma-separated list of specific services to start

Examples:

```
# Start all services
dtaas-services start

# Start specific services
dtaas-services start --services influxdb,rabbitmq
```

`dtaas-services stop`

Stops all running platform services.

Options:

- `-s, --services` - Comma-separated list of specific services to stop

Examples:

```
# Stop all services
dtaas-services stop

# Stop specific services
dtaas-services stop -s mongodb,grafana
```

`dtaas-services restart`

Restarts platform services.

Options:

- `-s, --services` - Comma-separated list of specific services to restart

Examples:

```
# Restart all services
dtaas-services restart

# Restart specific services
dtaas-services restart --services influxdb
```

`dtaas-services remove`

Removes platform services and optionally their volumes. Prompts for confirmation before removal.

Note: When volumes are removed with `--volumes`, the data directories are automatically recreated empty to ensure successful reinstallation of services.

Options:

- `-s, --services` - Comma-separated list of specific services to remove
- `-v, --volumes` - Remove volumes as well (data will be deleted but directories preserved)

Examples:

```
# Remove all services (with confirmation)
dtaas-services remove

# Remove specific services
dtaas-services remove --services influxdb,rabbitmq

# Remove all services and their volumes
dtaas-services remove --volumes

# Remove specific services with volumes
dtaas-services remove -s mongodb -v
```

`dtaas-services status`

Shows the current status of all services.

Options:

- `-s, --services` - Comma-separated list of specific services to check

Examples:

```
# Show status of all services
dtaas-services status

# Show status of specific services
dtaas-services status --services influxdb
```

`dtaas-services user add`

Adds user accounts to InfluxDB and RabbitMQ from `config/credentials.csv`.

Example:

```
dtaas-services user add
```

TROUBLESHOOTING

Permission Issues (Linux/macOS)

If you encounter permission errors when setting up services, ensure you run the setup command with appropriate privileges:

```
sudo -E env PATH="$PATH" dtaas-services setup
```

Docker Connection Issues

Ensure Docker daemon is running:

```
docker ps
```

Platform Services

It is recommended to install certain third-party software for use by digital twins running inside the DTaaS platform. *These services can only be installed in secure (TLS) mode.*

The following services can be installed:

- **PostgresSQL**: SQL database server
- **ThingsBoard**: is an Internet of Things (IoT) device management and data visualization platform
- **Influx** time-series database and dashboard service
- **Grafana** visualization and dashboard service
- **RabbitMQ** AMQP broker and its' management interface The **MQTT plugin** of this broker has been enabled. So, it can also be used as **MQTT** broker.
- **MongoDB** NoSQL database server

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 platform and the PT located outside the platform.

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.

DIRECTORY AND FILE STRUCTURE

- **config** is used for storing the service configuration
- **data** is used by the services for storing data
- **certs** is used for storing the TLS certificates needed by the services.
- **script** contains scripts for installation of services and creation of user accounts
- **log** contains service logs for ThingsBoard service
- *compose.services.secure.yml* helps with installation of RabbitMQ, MongoDB, Grafana and InfluxDB services.
- *compose.thingsboard.secure.yml* helps with installation of PostgreSQL, and ThingsBoard services.

There are two additional directories, namely **GitLab** and **runner**. These directories are related to installation of integrated GitLab and its runner. The instructions in this page are not related to **GitLab** and **runner** installation.

DOWNLOAD PACKAGE

The software is available as a zip package. The package should be downloaded and unzipped. A new **DTaaS-v0.8.0** folder is created. The remaining installation instructions assume the use of a Windows/Linux/MacOS terminal in the **DTaaS-v0.8.0** folder.

The steps outlined here should be followed for installation. The `services.foo.com` website hostname is used for illustration. This should be replaced with the appropriate server hostname. These steps assume that the DTaaS repository has been downloaded and navigation to the `deploy/services` directory has been completed.

CREATE COMMON CONFIG

1. Copy `config/services.env.template` into `config/services.env`.
2. Update `config/services.env` with suitable values for your environment.

   Take special care in setting strong passwords.

INSTALL POSTGRESQL AND THINGSBOARD

Configure

- Obtain the TLS certificates from LetsEncrypt and copy them.

```
cp -R /etc/letsencrypt/archive/services.foo.com certs/.
mv certs/services.foo.com/privkey1.pem certs/services.foo.com/privkey.pem
mv certs/services.foo.com/fullchain1.pem certs/services.foo.com/fullchain.pem
```

- Adjust permissions of certificates for PostgreSQL user in docker container.

```
cp certs/services.foo.com/privkey.pem \
  certs/services.foo.com/postgres.key
cp certs/services.foo.com/fullchain.pem \
  certs/services.foo.com/postgres.crt
chown 999:999 certs/services.foo.com/postgres.key \
  certs/services.foo.com/postgres.crt
chmod 600 certs/services.foo.com/postgres.key
chmod 644 certs/services.foo.com/postgres.crt
```

- Adjust permissions of certificates for ThingsBoard user in docker container.

```
cp certs/services.foo.com/privkey.pem \
  certs/services.foo.com/thingsboard-privkey.pem
cp certs/services.foo.com/fullchain.pem \
  certs/services.foo.com/thingsboard-fullchain.pem
chown 799:799 certs/services.foo.com/thingsboard-*.pem
chmod 600 certs/services.foo.com/thingsboard-privkey.pem
chmod 644 certs/services.foo.com/thingsboard-fullchain.pem
```

- Set required permissions for ThingsBoard data and log directories.

```
chown -R 799:799 data/thingsboard
chown -R 799:799 log/thingsboard
```

Install

- Start PostgreSQL and run ThingsBoard install.

```
docker compose -f compose.thingsboard.secure.yml \
  --env-file config/services.env \
  up -d postgres
docker compose -f compose.thingsboard.secure.yml \
  --env-file config/services.env \
  run --rm -e INSTALL_TB=true -e LOAD_DEMO=false thingsboard-ce
```

Once ThingsBoard is installed, the service can be started.

- Start or stop services.

```
docker compose -f compose.thingsboard.secure.yml \
  --env-file config/services.env up -d thingsboard-ce
docker compose -f compose.thingsboard.secure.yml \
  --env-file config/services.env down thingsboard-ce
```

Add New User Accounts for ThingsBoard

The password for the default ThingsBoard system admin should be changed as soon as possible. The following commands can be used to change the password and add a new tenant to the **ThingsBoard** service.

```
chmod +x script/thingsboard.py
python3 -m venv .venv
source .venv/bin/activate
pip install requests
python3 script/thingsboard.py
```

Troubleshooting

If the PostgreSQL logs show errors like:

- `ERROR: relation "ts_kv" does not exist`
- `ERROR: relation "ts_kv_latest" does not exist`

this usually means that the ThingsBoard service started before the database schema was fully created.

To fix this:

1. Stop all services:

```
docker compose -f compose.thingsboard.secure.yml \
  --env-file config/services.env down
```

1. Delete the data folders:

```
rm -rf data/thingsboard/*
rm -rf data/postgres/*
rm -rf log/thingsboard/*
```

1. Start PostgreSQL and run ThingsBoard install again:

```
docker compose -f compose.thingsboard.secure.yml \
  --env-file config/services.env up -d postgres
docker compose -f compose.thingsboard.secure.yml \
  --env-file config/services.env \
  run --rm -e INSTALL_TB=true -e LOAD_DEMO=false thingsboard-ce
```

INSTALLATION STEPS FOR OTHER SERVICES

Please follow the steps outlined here for installation. `script/service_setup.py`, is provided to streamline the setup of TLS certificates and permissions for MongoDB, InfluxDB, and RabbitMQ services.

The script has the following features:

- **Automation:** Automates all manual certificate and permission steps for MongoDB, InfluxDB, and RabbitMQ as described above.
- **Cross-platform:** Works on Linux, macOS, and Windows.
- **Configuration-driven:** Reads all required user IDs, group IDs, and hostnames from `config/services.env`.

Run Install Script

Install Python dependencies before running the script:

```
pip install -r script/requirements.txt
```

Run the installation script

```
cd deploy/services
sudo python3 script/service_setup.py
```

The script will:

- Combine and set permissions for MongoDB certificates.
- Copy and set permissions for InfluxDB and RabbitMQ certificates.
- Use the correct UID/GID values from `config/services.env`.
- Start the Docker Compose services automatically after setup.

If any required variable is missing, the script will exit with an error message.

This automation reduces manual errors and ensures your service containers have the correct certificate files and permissions for secure operation.

USE

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

service	external url
RabbitMQ Broker	services.foo.com:8083
RabbitMQ Broker Management Website	services.foo.com:8084
MQTT Broker	services.foo.com:8085
Influx	services.foo.com:8086
PostgreSQL	services.foo.com:5432
MongoDB database	services.foo.com:8087
Grafana	services.foo.com:8088
ThingsBoard	services.foo.com:8089

Please note that the TCP ports used by the services can be changed by updating the `config/service.env` file and rerunning the docker commands.

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 the DTaaS platform will not be able to utilize these services from their user workspaces.

NEW USER ACCOUNTS

There are ready to use scripts for adding accounts in **InfluxDB** and **RabbitMQ** services.

Copy the user accounts template and add user account credentials.

```
cp config/credentials.csv.template config/credentials.csv
# edit credentials.csv file
```

Use the following commands to add new users to **InfluxDB** service.

```
# on host machine
docker cp script/influxdb.py influxdb:/influxdb.py
docker cp config/credentials.csv influxdb:/credentials.csv
docker exec -it influxdb bash
# inside docker container
python3 influxdb.py
```

Use the following commands to add new users to **RabbitMQ** service.

```
# on host machine
docker cp script/rabbitmq.py rabbitmq:/rabbitmq.py
docker cp config/credentials.csv rabbitmq:/credentials.csv
docker exec -it rabbitmq bash
# inside docker container
python3 rabbitmq.py
```

3.2 Integrated Gitlab

3.2.1 Local GitLab Instance

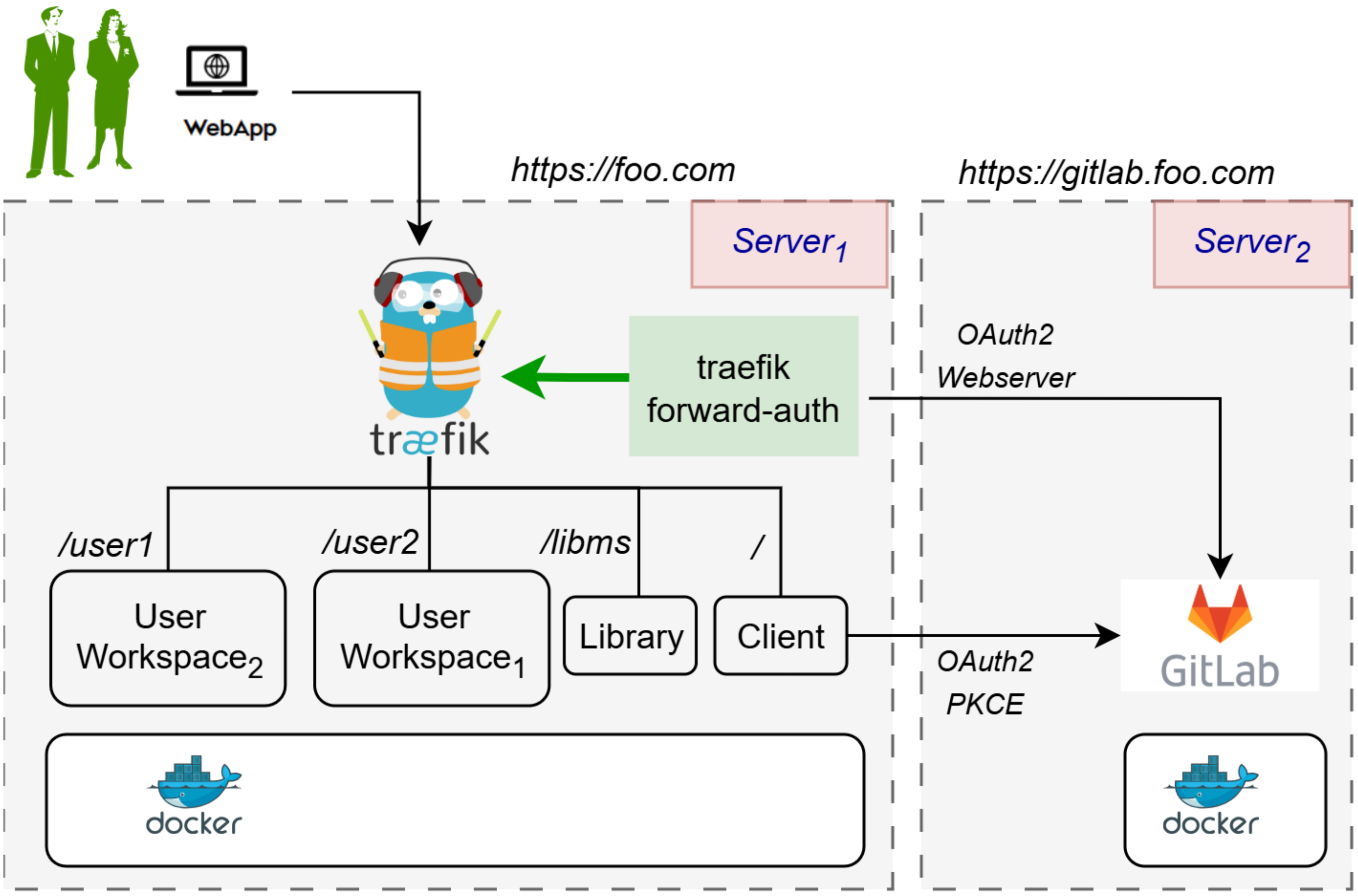
This guide provides instructions for installing a dedicated local GitLab instance. This GitLab installation can be used as an OAuth 2.0 authorization provider and DevOps backend for the DTaaS platform.

Design

Two possible methods exist for installing GitLab alongside the DTaaS:

- At a dedicated domain name (e.g., *gitlab.foo.com*)
- At a URL path on an existing WWW server (e.g., foo.com/gitlab)

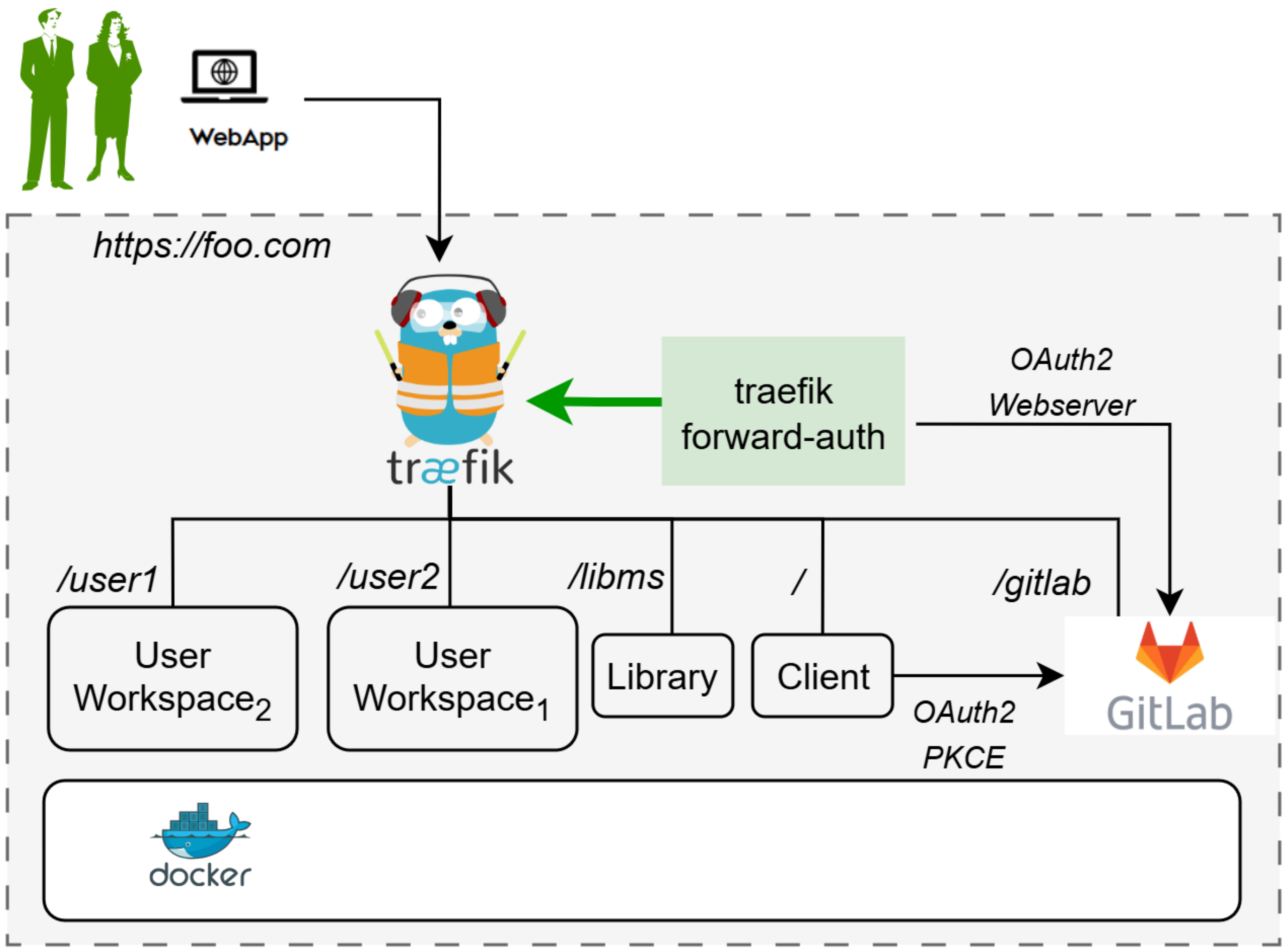
The first is a two-server installation setup where GitLab and DTaaS are installed on separate servers. An illustration of this setup is shown below.



🗒️ The text starting with `/` at the beginning indicates the URL route at which a certain service is available. For example, user workspace is available at https://localhost/user1.

The above figure shows integration of the DTaaS with a GitLab instance hosted at separate hostname, for example at https://gitlab.foo.com.

The second installation setup involves installation of both the GitLab and the DTaaS on the same server. An illustration of the integrated single-server installation setup is shown below.



This figure shows integration of GitLab instance hosted along side the DTaaS. The integrated GitLab is hosted behind the Traefik proxy.

This guide illustrates the installation of GitLab at: foo.com/gitlab. However, the instructions and `compose.gitlab.yml` can be adapted to install GitLab at a dedicated domain name.

Download Package

The software is available as a zip package. The package should be downloaded and unzipped. A new **DTaaS-v0.8.0** folder is created. The remaining installation instructions assume the use of a Windows/Linux/MacOS terminal in the **DTaaS-v0.8.0** folder.

Configure and Install

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 platform 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 platform 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
DTAAS_DIR	'/Users/username/DTaaS'	Full path to the DTaaS 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

The administrator username for GitLab is: `root`. The password for this user account will be available in: `config/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.

Use

After running the container, your local GitLab instance will be available at 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 document helps with installation of GitLab along side DTaaS application. But the OAuth 2.0 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 provides instructions for integrating a local GitLab instance with a DTaaS server installation and integrating the OAuth 2.0 Authorization feature with the DTaaS installation. The installation of Gitlab should be completed before attempting the integration steps described here.

After following this guide, the GitLab instance will be integrated as an OAuth 2.0 provider for both the 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

The existing guides should be followed to set up the DTaaS web application over HTTPS connection on either localhost (https://localhost) or a custom domain (https://foo.com).

Note

Steps related to configuring OAuth 2.0 application tokens at https://gitlab.com may be ignored. The initial installation will host the local GitLab instance, on which the OAuth 2.0 application tokens will later be created.

2. SET UP THE GITLAB INSTANCE

The guide should be followed to set up a GitLab instance.

After this step, a functioning GitLab instance (at either https://localhost/gitlab or https://foo.com/gitlab) will be available, along with login credentials for the root user.

3. CREATE USERS

The newly installed GitLab only contains a `root` user. The users specified in installation configuration files (`.env.local` / `.env.server`) must be created in this integrated GitLab server.

4. CREATE OAUTH 2.0 TOKENS IN GITLAB

Log in as a non-root user and follow these guides to create OAuth 2.0 Application Tokens for the backend and client. Note that the backend is not required for https://localhost installation.

After this step, credentials for the application tokens titled "DTaaS Server Authorization" and "DTaaS Client Authorization" will be available for use in the next step.

5. USE VALID OAUTH 2.0 APPLICATION TOKENS

The OAuth 2.0 tokens generated on the GitLab instance can now be used to enable authorization.

If the DTaaS platform is hosted at https://localhost, 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 platform is hosted at https://foo.com, 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 2.0 client ID and client secret from the **DTaaS Server Authorization** token.

Restart Services

LOCALHOST INSTALLATION

The updated OAuth 2.0 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 2.0 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 the setup has been completed correctly:

1. A functioning path-prefixed GitLab instance will be available at `https://foo.com/gitlab` that can be used 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 `localhost`).

Federation of DTaaS Installations

It is possible to use a single GitLab to serve multiple instances of the DTaaS installations. Please see DTaaS and DevOps video for an overview of

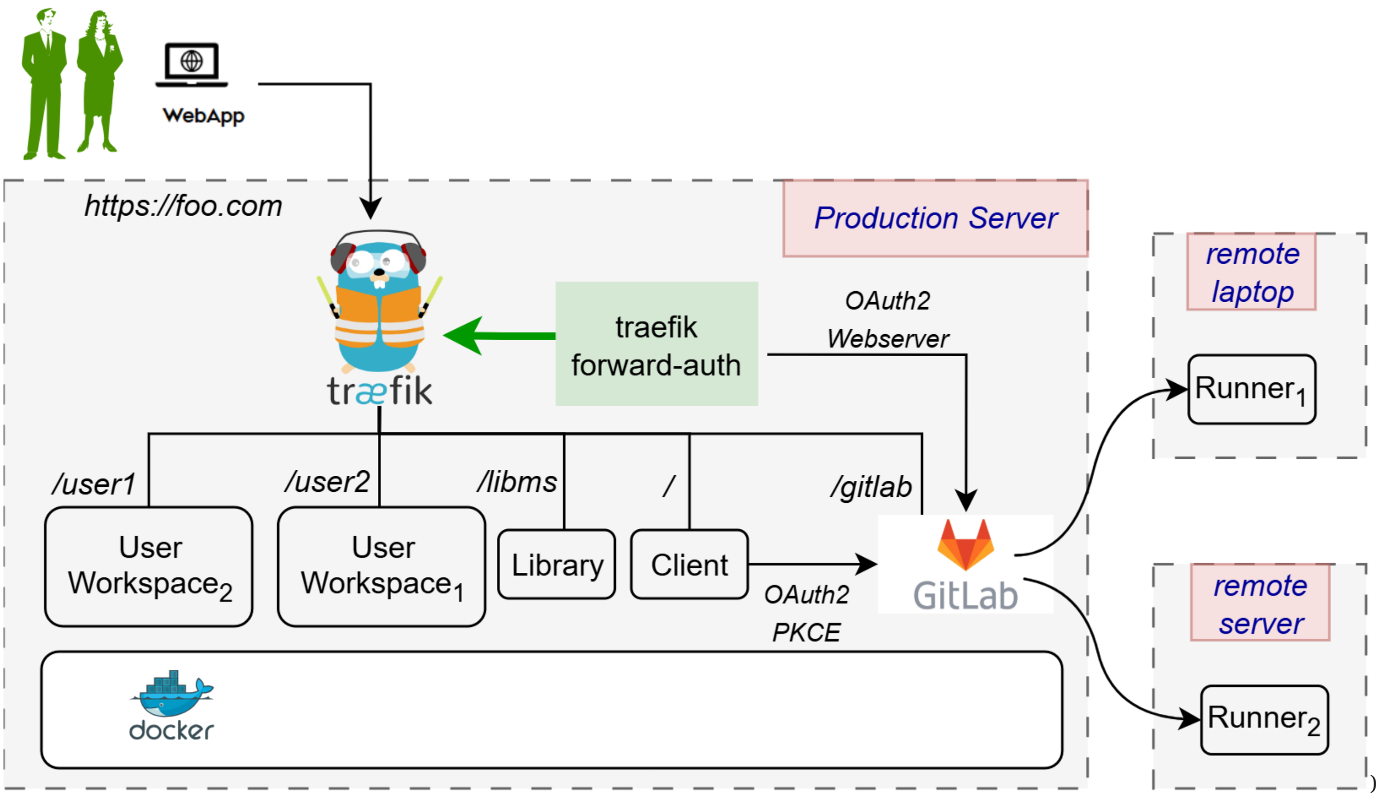
- Features in DTaaS v0.7 (Timestamps: 00:00 to 10:24)
- DTaaS and DevOps (Timestamps: 10:25 to 16:04)
- Federation of DTaaS (Timestamps: 16:05 till the end)

3.2.3 Gitlab Runner

GitLab Runner Integration

This document outlines the steps needed to create a `gitlab-runner` that will be responsible for the execution of Digital Twins. Many such runners can be installaed and linked with the integrated GitLab.

An illustration of the intended installation setup is shown below.



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 platform, refer to our GitLab instance document and our GitLab integration guide.

The rest of this document assumes you have a running DTaaS platform 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

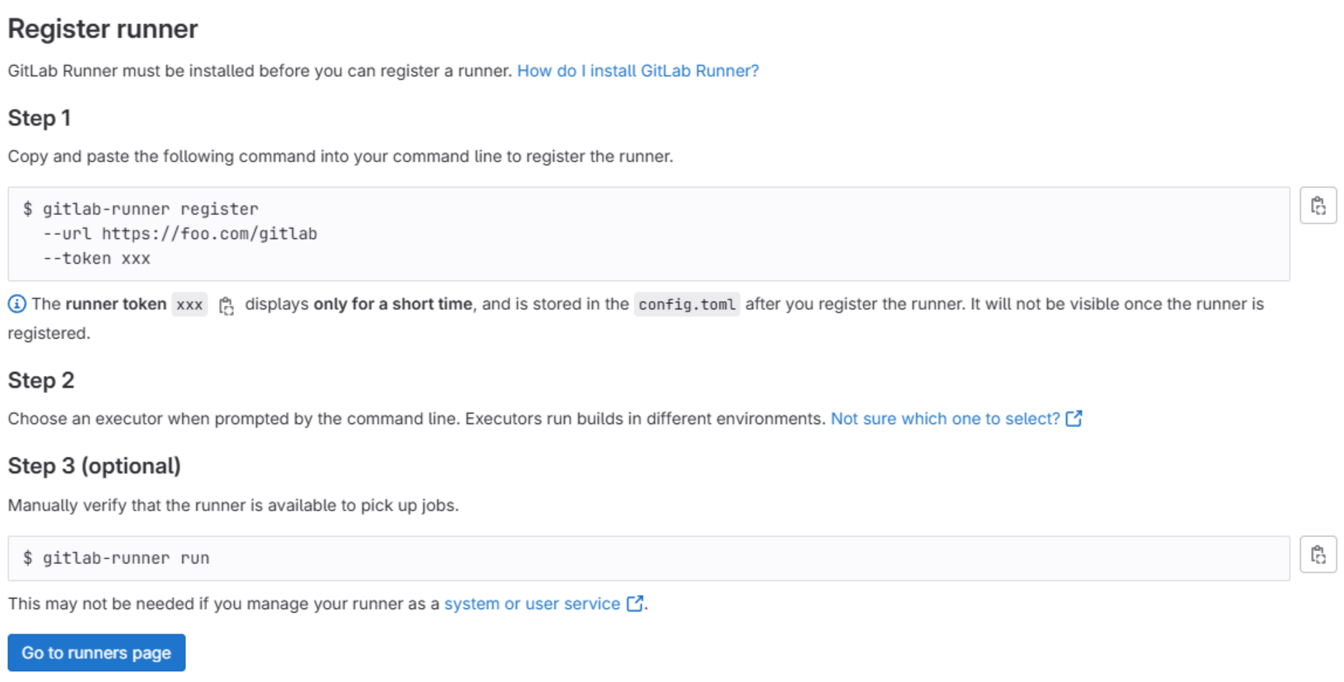
First, we will obtain the token necessary to register the runner for the GitLab instance. Open your GitLab instance (remote or local) and depending on your choice of runner scope, follow the steps given below:

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:



Register runner

GitLab Runner must be installed before you can register a runner. How do I install GitLab Runner?

Step 1

Copy and paste the following command into your command line to register the runner.

```
$ gitlab-runner register
  --url https://foo.com/gitlab
  --token xxx
```

The **runner token** `xxx` displays **only for a short time**, and is stored in the `config.toml` after you register the runner. It will not be visible once the runner is registered.

Step 2

Choose an executor when prompted by the command line. Executors run builds in different environments. Not sure which one to select?

Step 3 (optional)

Manually verify that the runner is available to pick up jobs.

```
$ gitlab-runner run
```

This may not be needed if you manage your runner as a system or user service.

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 platform, 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:

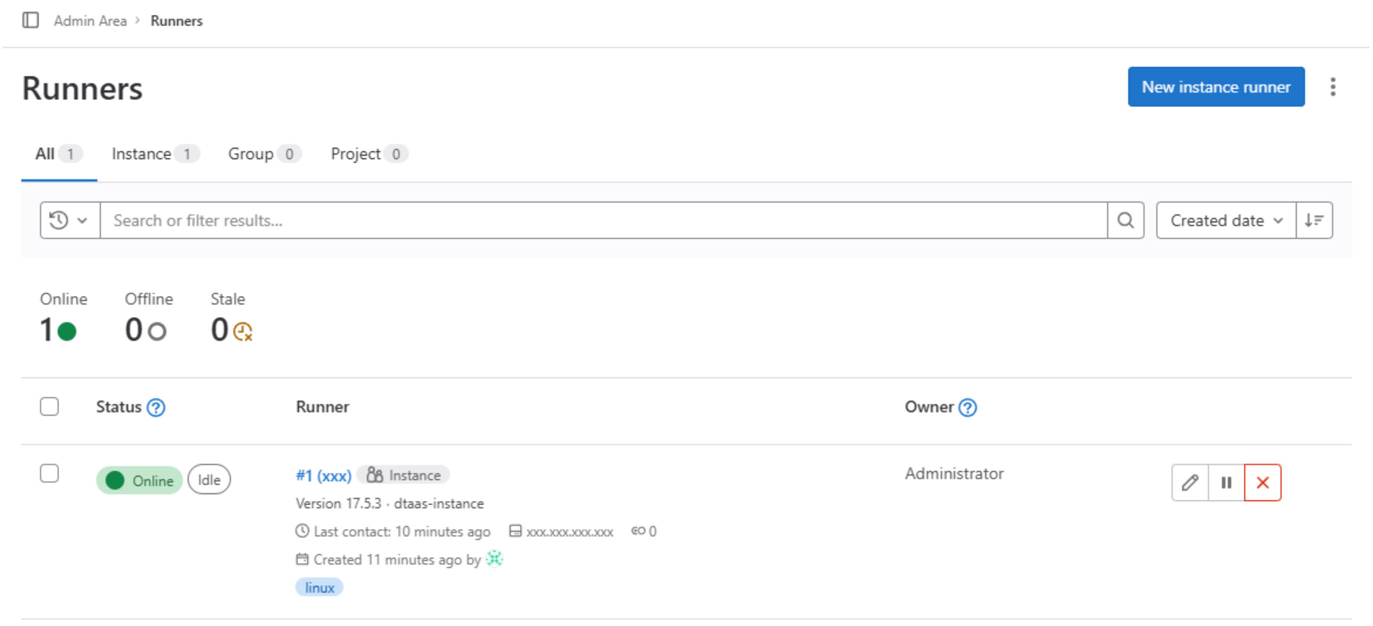
1. Go to the DTaaS home directory (`DTaaS_DIR`) and execute one of the following commands.
2. Localhost Installation

```
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:


Admin Area › Runners

Runners

New instance runner

All 1 Instance 1 Group 0 Project 0

Search or filter results...

Created date

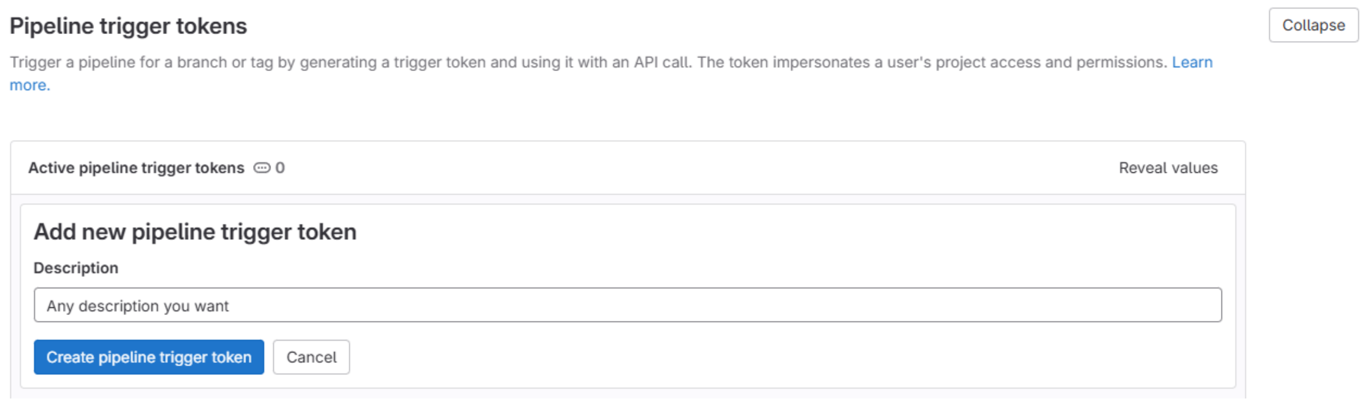
Online	Offline	Stale
1 ●	0 ○	0

Status	Runner	Owner
Online Idle	#1 (xxx) Instance Version 17.5.3 · dtaas-instance Last contact: 10 minutes ago xxx.xxx.xxx.xxx 0 Created 11 minutes ago by linux	Administrator

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.


Pipeline trigger tokens

Collapse

Trigger a pipeline for a branch or tag by generating a trigger token and using it with an API call. The token impersonates a user's project access and permissions. Learn more.

Active pipeline trigger tokens 0

Reveal values

Add new pipeline trigger token

Description

Any description you want

Create pipeline trigger token

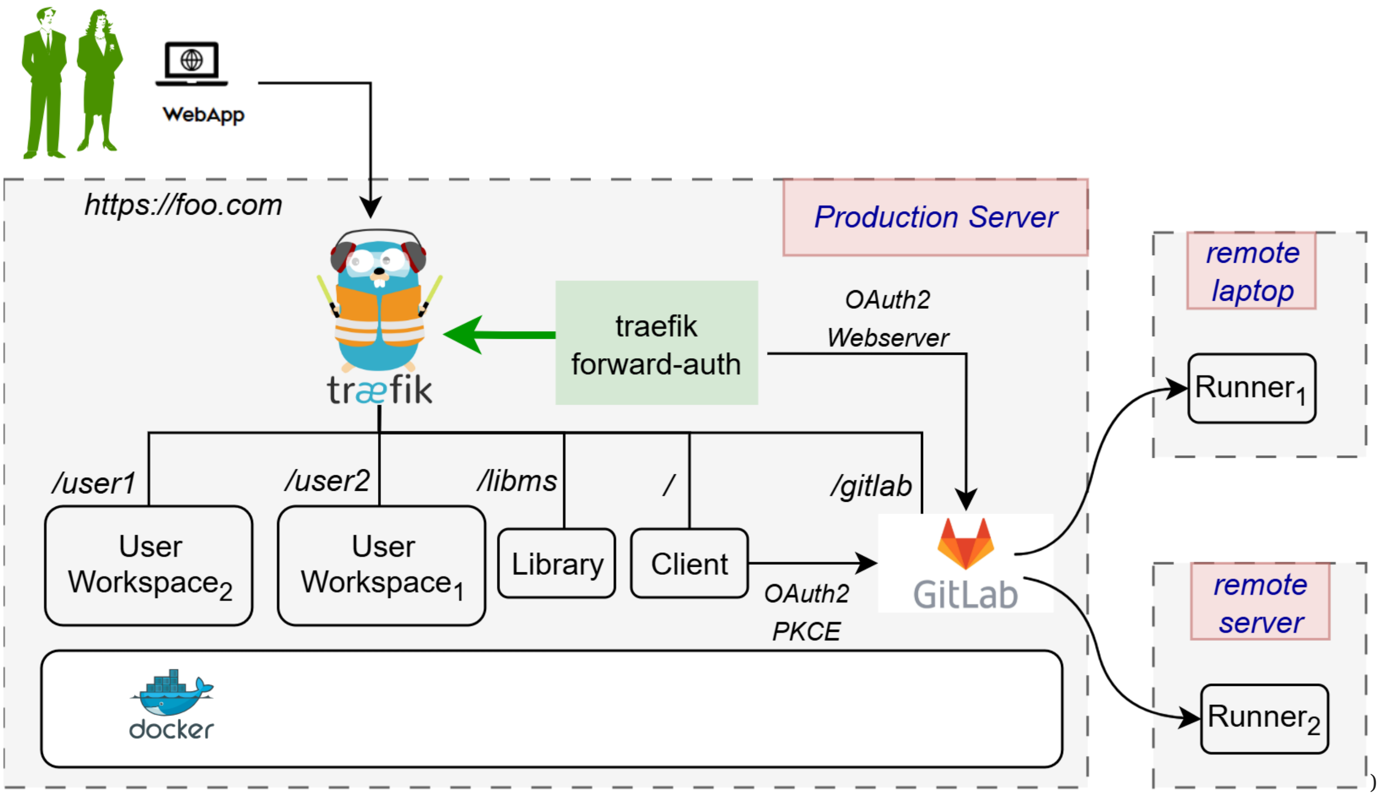
Cancel

You can now use the Digital Twins Preview Page to manage and execute your digital twins.

Setting up GitLab Runners with Docker on Windows for DTaaS

This guide documents how to properly set up and configure GitLab runners with Docker on Windows for the DTaaS platform.

This document outlines the steps needed to properly set up and configure GitLab runners on Windows. An illustration of the intended installation setup is shown below.



There are two installation scenarios:

STEP-BY-STEP SETUP PROCESS

1. Install GitLab Runner

Download and install GitLab Runner for Windows from GitLab's official download page.

```
# Navigate to your download directory
cd C:\path\to\download\folder

# Run the GitLab Runner
.\gitlab-runner.exe install

# Start the service
.\gitlab-runner.exe start
```

2. Getting a token

To get your GitLab token first head to your page should look something like this https://dtaas-digitaltwin.com/gitlab/dtaas/USERNAME, then do the following:

1. settings -> CI/CD -> Runners
2. Now press **New Project Runner**
3. Add tag `linux`
4. Leave the rest as is, and press **Create Runner**
5. Now you get your token! SAVE IT!

You now have your runner token.

3. Register Your Runner for DTaaS

For the DTaaS project, you need to register your runner using the specific GitLab instance URL and token:

```
# Register the runner for DTaaS
.\gitlab-runner.exe register --url "https://foo.com/gitlab" --token ""

# When prompted, enter:
# - Name: [Your machine name or any preferred name]
# - Executor: docker
# - Default Docker image: ruby:2.7
# - Tags: linux
```

This configuration is designed for the DTaaS digital twins which require a Linux environment to run properly. The pipelines use shell scripts with commands like `chmod +x`, which need a Linux-compatible environment.

4. Configure Your config.toml

The most important part is properly configuring your `config.toml` file, which is typically located at `C:\Users\YourUsername\.gitlab-runner\config.toml` or in the directory where you downloaded and ran the gitlab-runner executable.

DTaaS Configuration for Windows Hosts

Here's the recommended configuration for running DTaaS Digital Twins on Windows:

```
concurrent = 1
check_interval = 0
connection_max_age = "15m0s"
shutdown_timeout = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "Some name"
  url = "https://dtaas-digitaltwin.com/gitlab"
  id = 3
  token = "YOUR_RUNNER_TOKEN"
  token_obtained_at = 2025-03-17T19:50:25Z
  token_expires_at = 0001-01-01T00:00:00Z
  executor = "docker"
  [runners.custom_build_dir]
    enabled = false
  [runners.cache]
    MaxUploadedArchiveSize = 0
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.feature_flags]
    FF_NETWORK_PER_BUILD = false
  [runners.docker]
    tls_verify = false
    image = "ruby:2.7"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache"]
    shm_size = 0
    network_mtu = 0
  tags = ["linux"]
```

6. Restart the Runner

After making these configuration changes, restart your runner:

```
.\gitlab-runner.exe stop
.\gitlab-runner.exe start
```

VERIFYING YOUR SETUP

Run a verification check to ensure your runner is properly configured:

```
.\gitlab-runner.exe verify
```

A successful verification will show something like:

```
Verifying runner... is valid        runner=YourRunnerToken
```

UNDERSTANDING THE DTAAS PROJECT SETUP

The DTaaS project uses a specific structure for running digital twins:

1. Digital twins are contained in the `digital_twins` directory
2. Each digital twin has lifecycle scripts (create, execute, terminate, clean)
3. The GitLab CI/CD pipeline triggers these scripts based on user actions
4. The scripts need a Linux environment to execute properly

COMMON ERRORS

When attempting to run GitLab CI/CD pipelines on Windows with Docker, you might encounter these errors:

```
ERROR: Failed to remove network for build
ERROR: Job failed: invalid volume specification: "c:\\cache"
```

These errors are typically caused by:

1. Incorrect Docker executor configuration
2. Windows-style path specification not being compatible with Docker
3. Mismatch between the runner's executor type and the pipeline requirements

CONCLUSION

By following this guide, you should be able to properly set up GitLab runners with Docker on Windows for the DTaaS project and avoid the common configuration errors. The most crucial aspects are using the Docker (Linux) executor and properly formatting the volume paths.

Remember that for the DTaaS digital twins, using the standard Docker executor is required, even when running on a Windows host, since the scripts are designed to run in a Linux environment.

3.3 DTaaS Command Line Interface

The DTaaS Command Line Interface (CLI) is a command line tool for managing a DTaaS installation.

3.3.1 Prerequisite

The DTaaS platform with base users and essential containers must be operational before the CLI can be utilized.

3.3.2 Installation

The CLI is available as a Python package that can be installed via pip.

It is recommended to install the CLI in a virtual environment.

The installation steps are as follows:

- Change the working folder:

```
cd <DTaaS-directory>/cli
```

- It is recommended to use a virtual environment. A virtual environment should be created and activated.
- To install the CLI:

```
pip install dtaas
```

3.3.3 Usage

Note

The base DTaaS platform should be up and running before adding/deleting users with the CLI.

Configure

The CLI uses *dtaas.toml* as configuration file. A sample configuration file is given here.

```toml
# This is the config for DTaaS CLI

name = "Digital Twin as a Service (DTaaS)"
version = "0.2.1"
owner = "The INTO-CPS-Association"
git-repo = "https://github.com/into-cps-association/DTaaS.git"

[common]
# Server hostname either localhost or a valid hostname, ex: foo.com
server-dns = "localhost"
# absolute path to the DTaaS application directory
# Specify the directory of DTaaS installation
# Linux example
path = "/Users/username/DTaaS"
# Windows example
#path = "C:\\Users\\XXX\\DTaaS"
# Note: You have to either use / or \\ when specifying path, else you would get
# "Error while getting toml file: dtaas.toml, Invalid unicode value"

[common.resources]
# Default resource limits applied when creating user workspace containers.
# Keys:
# - cpus: integer count of virtual CPUs to allocate to the container
# - mem_limit: memory limit string accepted by Docker (e.g. "4G", "512M")
# - pids_limit: maximum number of processes the container may create
# - shm_size: size for /dev/shm (shared memory), e.g. "512m"
#
# Adjust these values to match your host capacity and tenancy policy.
cpus = 4
mem_limit = "4G"
pids_limit = 4960
shm_size = "512m"

# Example: Increase memory and lower CPU for heavier-memory workloads
# cpus = 2
# mem_limit = "8G"


[users]
# matching user info must present in this config file
add = ["username1","username2", "username3"]
delete = ["username2", "username3"]
...
```

NOTES

- Edits to `dtaas.toml` affect new user containers created after the change.
- To apply updated limits to existing containers, recreate or restart the user container(s) (for example by removing and re-adding the user workspace via the CLI or by restarting the container in Docker Compose).
- Use units (`M` , `G`) for memory and shared memory values.

Select Template

The *cli* uses YAML templates provided in this directory to create new user workspaces. The available templates are:

1. *user.local.yml*: localhost installation
2. *User.server.yml*: multi-user web application over HTTP
3. *user.server.secure.yml*: multi-user web application over HTTPS

It should be noted that the *cli* is not capable of detecting the difference between HTTP and HTTPS modes of the web application. When serving the web application over HTTPS, an additional step is required.

```
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, the *users.add* list in *dtaas.toml* should be populated 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"]
```

The working directory must be the *cli* directory.

Then execute:

```
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 process 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 complete. 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 accomplished as follows:
- Navigate 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, the *users.delete* list in *dtaas.toml* should be populated 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"]
```

- The working directory must be the *cli* directory.

Then execute:

```
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	Useful only for DevOps features. The workspace features will not be available in 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 that serves 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 the npm package of the library microservice and running the same as a standalone service.

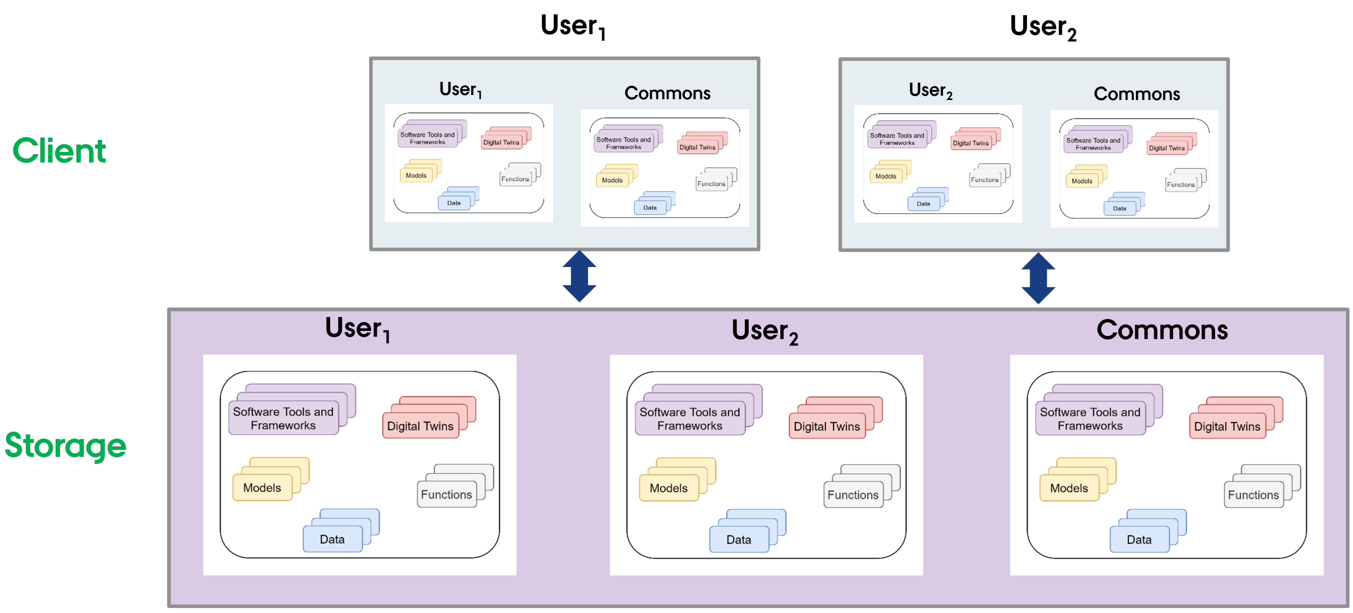
SETUP THE FILE SYSTEM

Outside the DTaaS Platform

The package can be used independently of the DTaaS. In this use case, no specific file structure is required. Any valid file directory is sufficient.

Inside the DTaaS Platform

The users of the DTaaS expect the following file system structure for their reusable assets.



A skeleton file structure is available in the DTaaS codebase. This can be copied to create a file system for 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
Usage: libms [options]

The lib microservice is a file server. It supports file transfer
over GraphQL and HTTP protocols.

Options:
  -c, --config <file>  provide the config file (default libms.yaml)
  -H, --http <file>    enable the HTTP server with the specified config
  -h, --help           display help for libms
```

Both the options are not mandatory.

Please see configuration for explanation of configuration conventions. The config is saved `libms.yaml` file by convention. If `-c` is not specified 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

```
$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 `libms.yaml`, for example as `libms-config.yaml`, you can run

```
$libms -c "config/libms-config.yaml"
```

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.

Protocol Support

The **libms** supports GraphQL protocol by default. 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 that serves files over GraphQL and HTTP API.

It has two features:

- Provide a listing of directory contents.
- Transfer a file to the user.

This document provides instructions for running a docker container to provide a standalone library microservice.

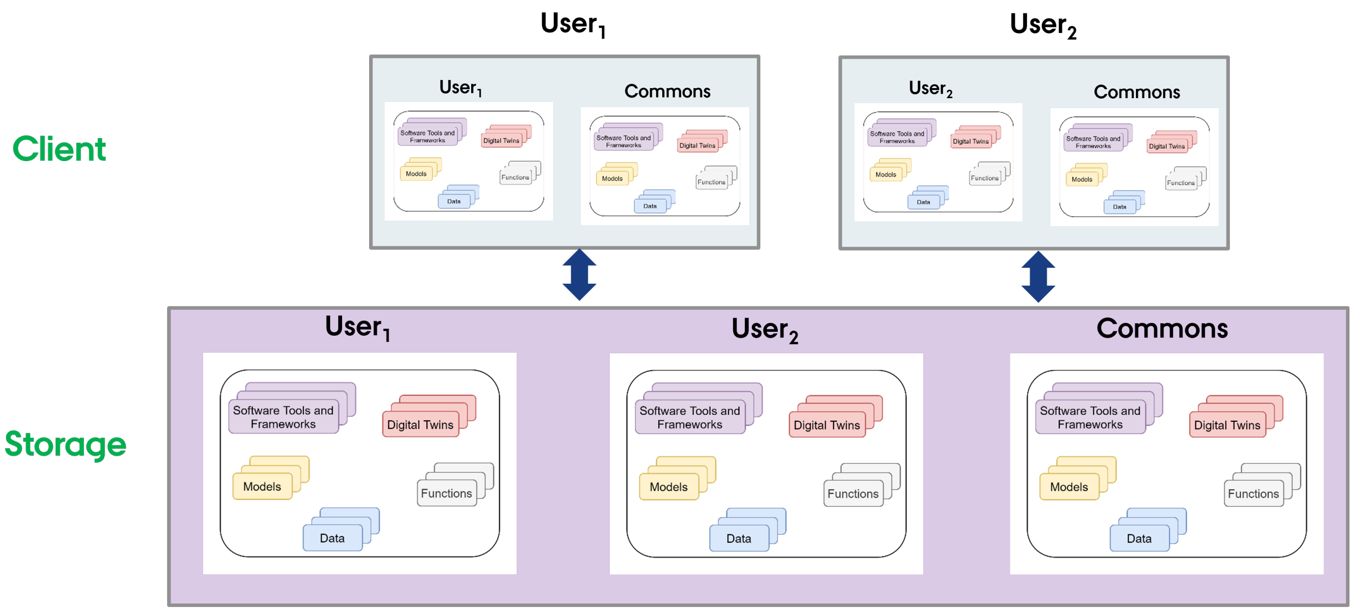
SETUP THE FILE SYSTEM

Outside the DTaaS Platform

The package can be used independently of the DTaaS. In this use case, no specific file structure is required. A valid file directory named `files` is sufficient and should be placed in the directory from which `compose.lib.yml` will be run.

Inside the DTaaS Platform

The users of DTaaS expect the following file system structure for their reusable assets.



A skeleton file structure is available in the DTaaS codebase. This can be copied to create a file system for users. The directory containing the file structure should be named `files` and placed in the directory from which `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.

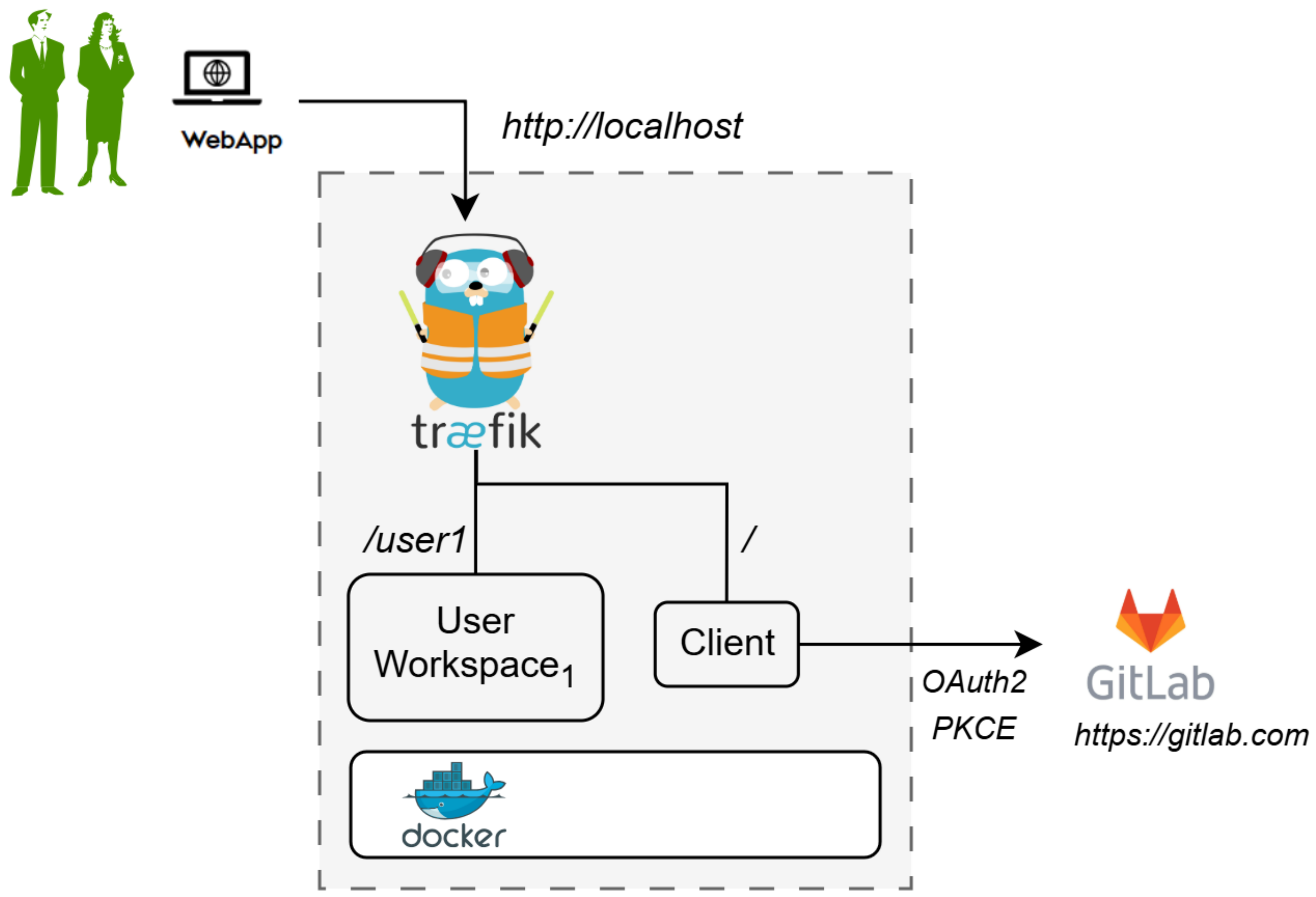
3.5 Guides

3.5.1 Install DTaaS on localhost (GUI)

The installation instructions provided in this document 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.

Download Package

The software is available as a zip package. The package should be downloaded and unzipped. A new **DTaaS-v0.8.0** folder is created. The remaining installation instructions assume the use of a Windows/Linux/MacOS terminal in the **DTaaS-v0.8.0** folder.

In this guide we will assume the contents of the zip file have been extracted to the directory: `/Users/username/DTaaS`.

> **Tip**
>
> 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 `/Users/username/DTaaS` in this guide.

Starting Portainer

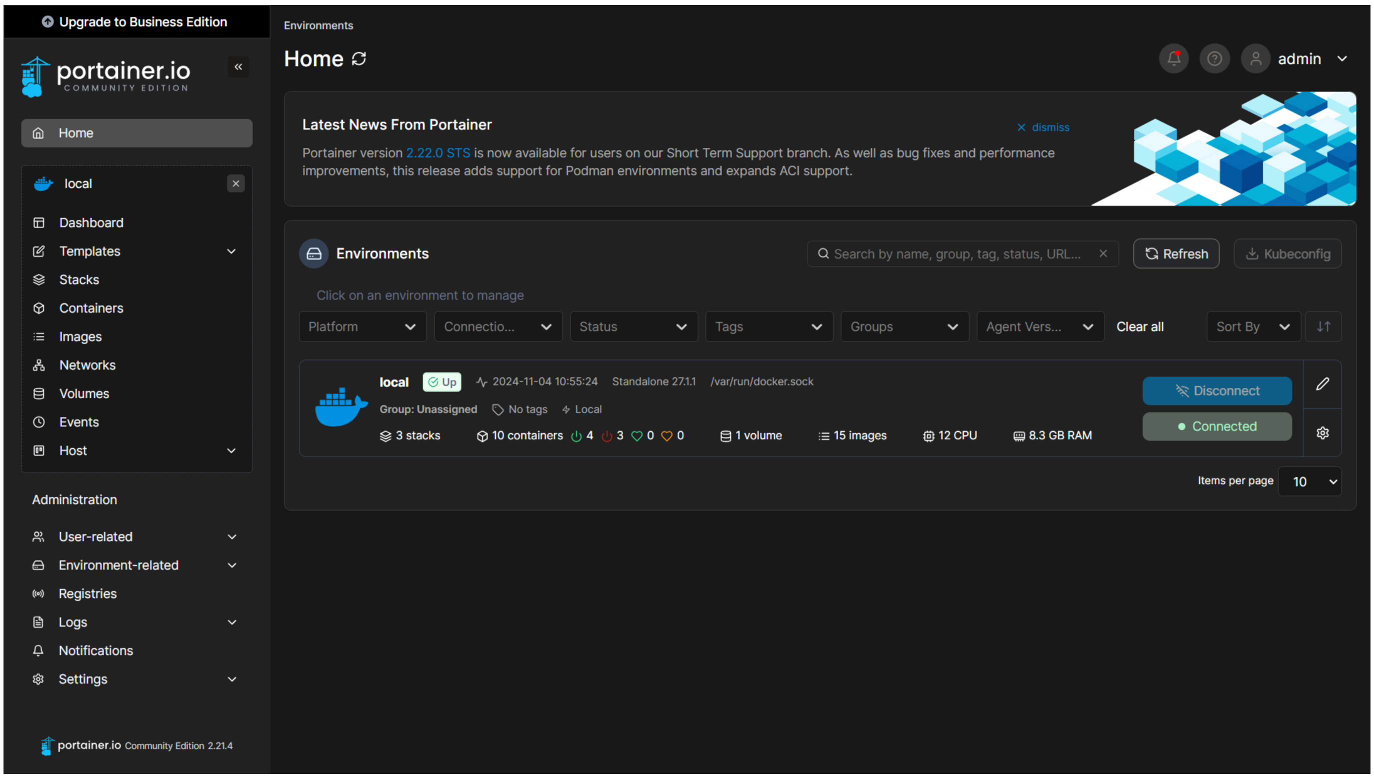
The GUI used to run the application and docker containers will be provided by Portainer Community Edition. It is itself a Docker container that will create a website at `https://localhost:9443`, which will present a graphical interface for starting and stopping the application.

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:



Latest News From Portainer
dismiss
Portainer version 2.22.0 STS is now available for users on our Short Term Support branch. As well as bug fixes and performance improvements, this release adds support for Podman environments and expands ACI support.

local Up 2024-11-04 10:55:24 Standalone 27.1.1 /var/run/docker.sock
Group: Unassigned No tags Local
3 stacks 10 containers 4 3 0 0 1 volume 15 images 12 CPU 8.3 GB RAM

> **Tip**
>
> The next time you wish to start the Portainer server, run `docker start portainer`.

Configuration

CREATE USER WORKSPACE

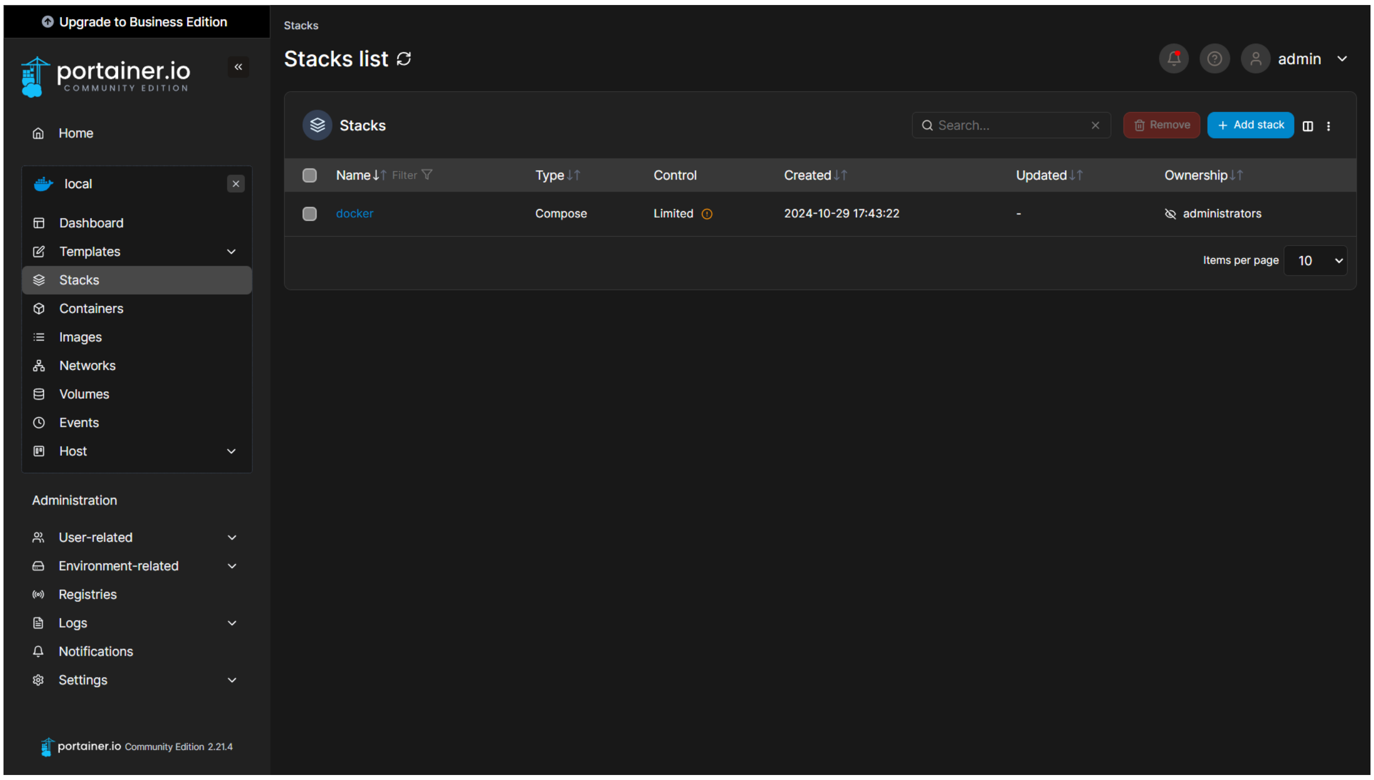
The existing filesystem for installation is setup for `user1`. A new filesystem directory needs to be created for the selected user.

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.

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

CREATING THE PORTAINER STACK



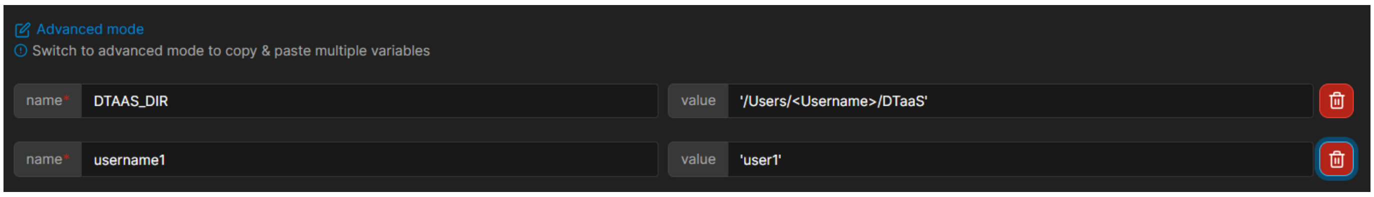
Name	Type	Control	Created	Updated	Ownership
docker	Compose	Limited	2024-10-29 17:43:22	-	administrators

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`.

Tip

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*.



name: DTAAS_DIR value: '/Users/<Username>/DTaaS'

name: username1 value: 'user1'

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	'/Users/username/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

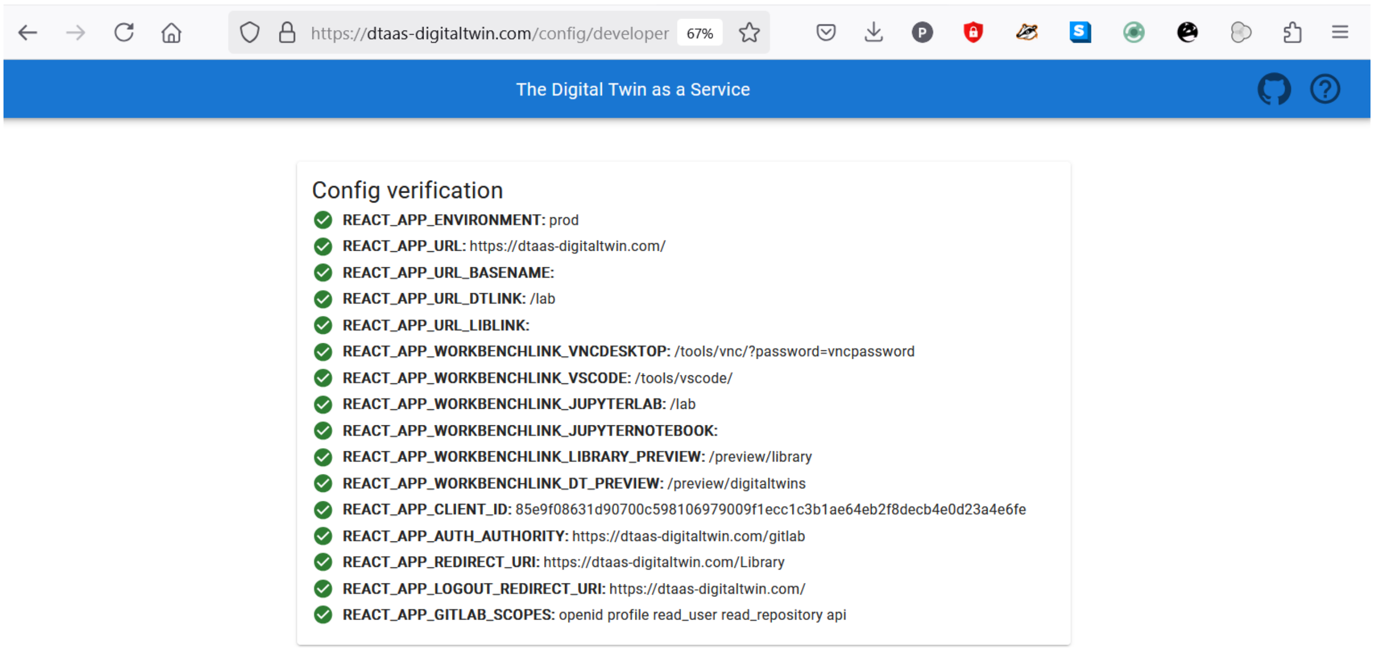
The library microservice is not included in the localhost installation scenario.

References

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

3.5.2 Check Client Configuration

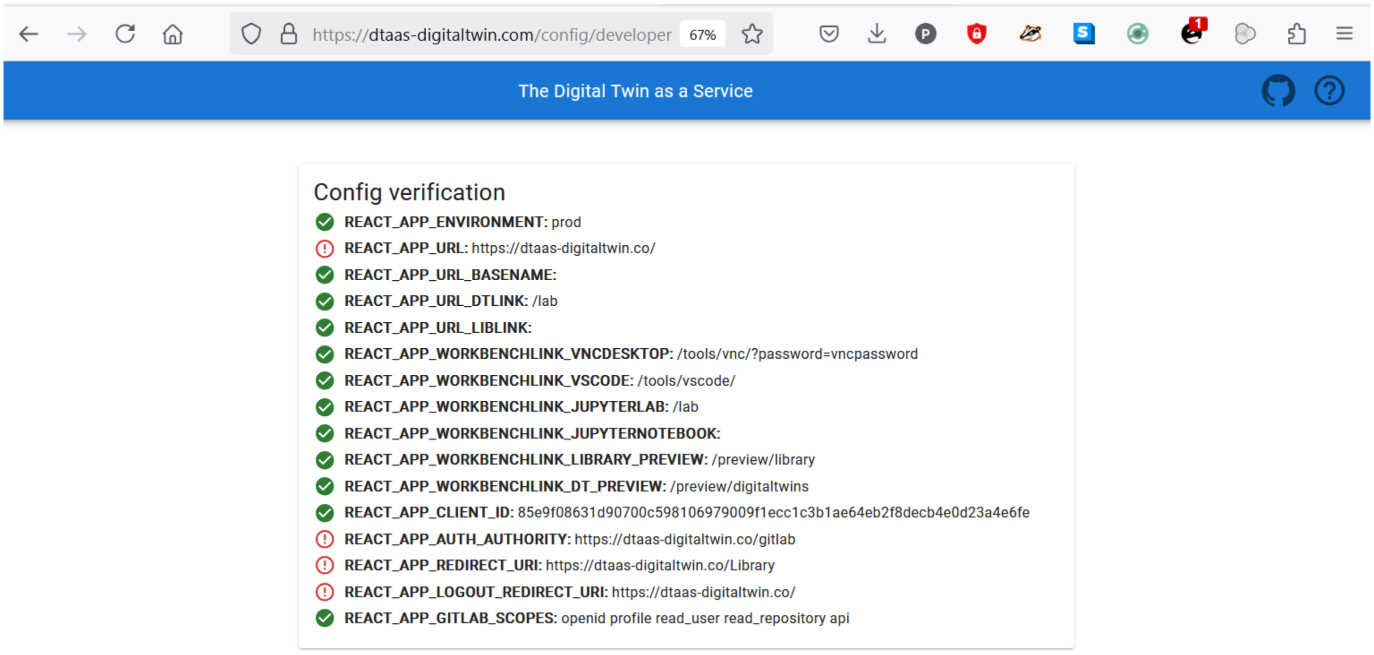
One of the common errors is the incorrect configuration of the react website. There is now a helpful checklist to verify the client configuration. A correct configuration shows the following result.


https://dtaas-digitaltwin.com/config/developer
67%
The Digital Twin as a Service

Config verification

- ✓ **REACT_APP_ENVIRONMENT:** prod
- ✓ **REACT_APP_URL:** https://dtaas-digitaltwin.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_LIBRARY_PREVIEW:** /preview/library
- ✓ **REACT_APP_WORKBENCHLINK_DT_PREVIEW:** /preview/digitaltwins
- ✓ **REACT_APP_CLIENT_ID:** 85e9f08631d90700c598106979009f1ecc1c3b1ae64eb2f8decb4e0d23a4e6fe
- ✓ **REACT_APP_AUTH_AUTHORITY:** https://dtaas-digitaltwin.com/gitlab
- ✓ **REACT_APP_REDIRECT_URI:** https://dtaas-digitaltwin.com/Library
- ✓ **REACT_APP_LOGOUT_REDIRECT_URI:** https://dtaas-digitaltwin.com/
- ✓ **REACT_APP_GITLAB_SCOPES:** openid profile read_user read_repository api

In case of incorrect client website URL, the following result will be shown.


https://dtaas-digitaltwin.com/config/developer
67%
The Digital Twin as a Service

Config verification

- ✓ **REACT_APP_ENVIRONMENT:** prod
- ❗ **REACT_APP_URL:** https://dtaas-digitaltwin.co/
- ✓ **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_LIBRARY_PREVIEW:** /preview/library
- ✓ **REACT_APP_WORKBENCHLINK_DT_PREVIEW:** /preview/digitaltwins
- ✓ **REACT_APP_CLIENT_ID:** 85e9f08631d90700c598106979009f1ecc1c3b1ae64eb2f8decb4e0d23a4e6fe
- ❗ **REACT_APP_AUTH_AUTHORITY:** https://dtaas-digitaltwin.co/gitlab
- ❗ **REACT_APP_REDIRECT_URI:** https://dtaas-digitaltwin.co/Library
- ❗ **REACT_APP_LOGOUT_REDIRECT_URI:** https://dtaas-digitaltwin.co/
- ✓ **REACT_APP_GITLAB_SCOPES:** openid profile read_user read_repository api

3.5.3 Add User

This page provides steps for adding a user to a DTaaS installation. The username **alice** is used here to illustrate the steps involved in adding a user account.

The following steps should be performed:

1. Add user to GitLab instance:

A new account for the new user should be added on the GitLab instance. The username and email of the new account should be noted.

2. Create User Workspace:

The DTaaS CLI should be used to bring up the workspaces for new users. This brings up the containers without backend authorization.

3. Add backend authorization for the user:

- Navigate 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.

3.5.4 Remove User

This page provides steps for removing a user from a DTaaS installation. The username **alice** is used here to illustrate the steps involved in removing a user account.

The following steps should be performed:

1. Remove an existing user with the DTaaS CLI.

2. Remove backend authorization for the user:

- Navigate 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 from GitLab instance (optional):

The GitLab docs provide additional guidance.

4. The user account is now deleted.

Caveat

The two base users that the DTaaS platform was installed with cannot be deleted. Only the extra users that have been added to the software can be deleted.

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.

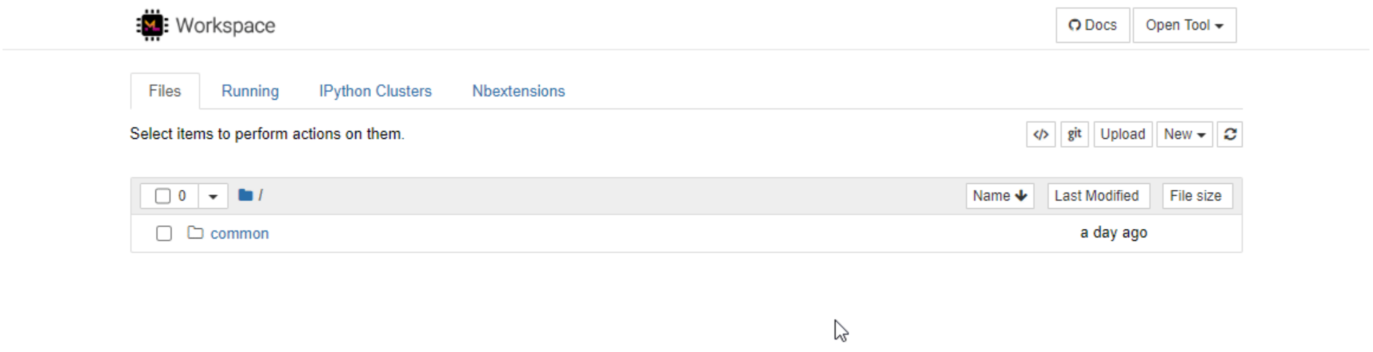
1. Step:

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`.

3.5.7 Renewing LetsEncrypt Certificates

LetsEncrypt certificates expire every three months and must be renewed to prevent certificate validation errors in client web browsers. This guide documents the certificate renewal process for DTaaS platform installations using LetsEncrypt certificates.

Overview

The certificate renewal process involves three main phases:

1. **Certificate Generation**: Renewing certificates using LetsEncrypt certbot
2. **Certificate Deployment**: Copying new certificates to appropriate directories
3. **Service Restart**: Restarting affected services to load new certificates

Prerequisites

- Administrative access to the DTaaS server
- LetsEncrypt certbot installed on the system
- Valid domain name configured for certificate generation
- Access to Docker commands

Certificate Renewal Process

STEP 1: GENERATE NEW CERTIFICATES

Use LetsEncrypt certbot to renew existing certificates:

```
# Test renewal process without actually renewing
sudo certbot renew --dry-run

# Renew all certificates
sudo certbot renew

# Renew specific certificate for domain
sudo certbot renew --cert-name example.com
```

STEP 2: LOCATE CERTIFICATE FILES

After successful renewal, locate the new certificate files:

```
# Standard LetsEncrypt certificate location
ls -la /etc/letsencrypt/live/your-domain.com/

# Certificate files:
# - fullchain.pem: Full certificate chain
# - privkey.pem: Private key
# - cert.pem: Certificate only
# - chain.pem: Certificate chain only
```

STEP 3: DEPLOY CERTIFICATES TO DTAAS

Copy the renewed certificates to the appropriate DTaaS directories:

```
# Copy certificates to DTaaS docker deployment
sudo cp /etc/letsencrypt/live/your-domain.com/fullchain.pem \
  /path/to/DTaaS/deploy/docker/certs/your-domain.com/
sudo cp /etc/letsencrypt/live/your-domain.com/privkey.pem \
  /path/to/DTaaS/deploy/docker/certs/your-domain.com/

# Copy certificates to DTaaS services deployment
sudo cp /etc/letsencrypt/live/your-domain.com/fullchain.pem \
  /path/to/DTaaS/deploy/services/certs/your-domain.com/
sudo cp /etc/letsencrypt/live/your-domain.com/privkey.pem \
  /path/to/DTaaS/deploy/services/certs/your-domain.com/
```

STEP 4: VERIFY CONTAINER VOLUME MAPPINGS

Before restarting services, verify the volume mappings to ensure certificates are mounted correctly:

```
# Inspect container volume mappings
docker inspect <container-name>

# Look for volume mounts in the output
# Example: "/host/path/certs:/container/path/certs:ro"
```

STEP 5: RESTART DTAAS SERVICE GATEWAY

Navigate to the DTaaS docker deployment directory and restart Traefik:

```
cd /path/to/DTaaS/deploy/docker
docker compose -f compose.server.secure.yml --env-file .env.server up \
  -d --force-recreate traefik
```

STEP 6: RESTART PLATFORM SERVICES

Navigate to the services directory and restart individual services:

```
cd /path/to/DTaaS/deploy/services

# Restart Grafana
docker stop grafana-server
docker start grafana-server

# Restart InfluxDB
docker stop influxdb
docker start influxdb
```

STEP 7: CONFIGURE RABBITMQ CERTIFICATES

RabbitMQ requires a specific certificate format. Create a copy of the private key and restart the service:

```
# Create RabbitMQ-specific private key
cp /path/to/DTaaS/deploy/services/certs/your-domain.com/privkey.pem \
   /path/to/DTaaS/deploy/services/certs/your-domain.com/privkey-rabbitmq.pem

# Restart RabbitMQ
docker restart rabbitmq
```

STEP 8: CONFIGURE MONGODB CERTIFICATES

MongoDB requires a combined certificate file containing both the certificate chain and private key:

```
# Create combined certificate file
cat /path/to/DTaaS/deploy/services/certs/your-domain.com/fullchain.pem \
    /path/to/DTaaS/deploy/services/certs/your-domain.com/privkey.pem > \
    /path/to/DTaaS/deploy/services/certs/your-domain.com/combined.pem

# Restart MongoDB with new certificates
cd /path/to/DTaaS/deploy/services
docker compose -f compose.services.secure.yml \
  --env-file config/services.env up -d --force-recreate mongodb
```

Verification

After completing the renewal process, verify that certificates are properly installed:

```
# Test HTTPS connectivity
curl -I https://your-domain.com

# Check Docker service logs for certificate errors
docker logs traefik
docker logs grafana-server
docker logs mongodb
```

Troubleshooting

COMMON ISSUES

Certificate Not Found Error : Verify that certificate files exist in the specified directories and have correct permissions (typically 644 for certificates, 600 for private keys).

Service Restart Failures : Check Docker service logs for specific error messages. Ensure that certificate paths in Docker Compose files match the actual file locations.

Browser Certificate Warnings : Clear browser cache and verify that the certificate chain is complete. Check that intermediate certificates are included in the fullchain.pem file.

LOG ANALYSIS

Monitor service logs for certificate-related errors:

```
# Monitor Traefik logs
docker logs traefik

# Monitor service logs
docker logs grafana-server
docker logs influxdb
docker logs rabbitmq
docker logs mongodb
```

Security Considerations

- Store private keys with restrictive permissions (600 or 640)
- Regularly monitor certificate expiration dates
- Implement automated monitoring to alert before certificate expiration
- Maintain backups of certificate files
- Use strong file system permissions on certificate directories

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 the DTaaS platform?

The DTaaS platform is a software platform on which digital twins can be created and executed. The features page provides an overview of the capabilities available in DTaaS.

What is the scope and current capabilities of the DTaaS platform?

1. DTaaS is a web-based interface that allows invocation of various tools related to work to be performed with one or more DTs.
2. DTaaS permits users to run DTs in their private workspaces. These user workspaces are based on the Ubuntu 20.04 Operating system.
3. DTaaS can help create reusable DT assets only if DT asset authoring tools can operate in the Ubuntu 20.04 xfce desktop environment.
4. DTs are executables from the DTaaS platform perspective. Users are not constrained to work with DTs in a specific manner. DTaaS suggests creation of DTs from reusable assets and provides a suggestive structure for DTs. The examples provide more insight into the DTaaS workflow. However, this suggested workflow is not mandatory.
5. DTs can be run as services with REST API from within user workspaces, which can facilitate service-level DT composition.

What can not be done inside the DTaaS platform?

1. DTaaS as such does not help install DTs obtained from external sources.
2. The current user interface of the 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.

Is there any fundamental difference between commercial solutions like Ansys Twin Builder and the DTaaS platform?

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. However, users have private workspaces based on a Linux xfce Desktop environment. Users can install proprietary and commercially licensed software in their workspaces. A screencast demonstrates using Matlab Simulink within the DTaaS platform. Licensed software installed by one user is not available to other users.

4.3 Digital Twin Assets

Can the DTaaS platform 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 a workspace are only available to that user.

4.4 Digital Twin Models

Can the DTaaS platform create new DT models?

DTaaS is not a model creation tool. Model creation tools can be placed inside DTaaS to create new models. The DTaaS platform itself does not create digital twin models but can help users create digital twin models. Linux desktop/terminal tools can be run inside DTaaS. Thus, models can be created inside DTaaS and executed using tools that run on Linux. Windows-only tools cannot run in DTaaS.

How can the DTaaS platform 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, the DTaaS platform 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 the DTaaS platform support only the information models (or behavioral models) or some other kind of models?

The DTaaS platform 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 platform 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 the DTaaS platform 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?

The DTaaS platform 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 Digital Twin DevOps Automation

Can a DT execute forever?

The web UI imposes a 10-minute timeout. The users can manually terminate an ongoing execution. The best choice would be for a DT to execute terminate script which consequently concludes the execution and returns the logs.

Is the DT execution really scalable?

This capacity of DT execution infrastructure is dependent on GitLab and the available compute power available to runners of GitLab. GitLab itself does not impose limits on the maximum number of runners.

I have many GitLab runners attached with my GitLab repository? Which one is used?

This is indeterminate. You can't rely on the order and location of execution for a DT.

4.7 Data Management

Can the DTaaS platform 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 platform provids InfluxDB, PostgreSQL, RabbitMQ, MQTT, MongoDB and ThingsBoard 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 the DTaaS.

Is the DTaaS platform able to transmit data to cloud in real time?

Yes via platform services.

Which transmission protocol does the DTaaS platform allow?

InfluxDB, RabbitMQ, MQTT and anything else that can be used from Cloud service providers.

Does the DTaaS platform 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 the DTaaS platform 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.

4.8 Platform Native Services on the DTaaS Platform

Is the DTaaS platform 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. It is possible to perform anomaly detection using the platform services such as Grafana, ThingsBoard and InfluxDB.

4.9 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 S
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
The DTaaS Platform	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.

4.10 GDPR Concerns

Does your platform adhere to GDPR compliance standards? If so, how?

The DTaaS 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 platform installation. The administrators of DTaaS platform 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 the DTaaS platform can be installed inside corporate server hosted behind network firewalls 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 the DTaaS. The first-level is HTTP basic authorization over secure HTTPS connection. The second-level is the OAuth 2.0 PKCE authorization flow for each user. The OAuth 2.0 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

This guide provides an overview of the contribution workflow for the Digital Twin as a Service (DTaaS) project. Contributors are encouraged to review the Code of Conduct to ensure a respectful and collaborative community environment.

The following sections outline the contribution process, from opening an issue to creating a pull request (PR), conducting reviews, and merging the PR.

5.1.1 Project Goals

The DTaaS platform aims to facilitate the creation, management, and execution of Digital Twins (DTs) through a service-oriented architecture that promotes reusability of DT assets [1]. This documentation assists development team members in understanding the DTaaS project software design and development processes.

Additional resources include:

- Developer Slides
- Video Presentation
- Research Paper

5.1.2 💻 Development Environment

A devcontainer configuration is provided in `.devcontainer/devcontainer.json` for the project. This configuration offers a dockerized development environment and represents the recommended approach for establishing a consistent development setup. DevContainer provides the most straightforward method for initializing the development environment.

5.1.3 🏗️ Development Workflow

A structured development workflow is employed to manage collaboration among multiple developers. Each contributor should adhere to the following procedure:

1. Create a fork of the main repository into a personal GitHub account.
2. Configure Qlty, Codecov, and SonarQube for the forked repository. Note that Codecov does not require a secret token for public repositories.
3. Utilize Node.js 24 and Python 3.12 development environments.
4. Follow the Fork, Branch, PR workflow methodology.
5. Develop within the fork and create a PR from the working branch to the `feature/distributed-demo` branch. The PR triggers all GitHub Actions, Qlty, SonarQube, and Codecov checks.
6. Address all issues identified during the automated checks.
7. Upon successful verification, submit a PR to the `feature/distributed-demo` branch of the upstream DTaaS repository.
8. The PR undergoes review and is merged by either project administrators or maintainers.

Each PR should represent a meaningful contribution that satisfies either a well-defined user story or improves code quality.

5.1.4 ✨ Coding Agents and Editors

The project makes extensive use of coding agents. The primary usage scenarios include:

👍 Co-development assistance

👍 Code review support

👍 Draft pull requests for prototyping ideas

This project is structured as a monorepo and includes copilot instructions that inform GitHub Copilot about the project structure and software development conventions.

Modern Code IDEs such as VS Code and Cursor provide native integration with coding agents, requiring no additional configuration for LLM-driven development workflows. Contributors should disclose code generated by coding agents, particularly when such code embeds significant programming logic.

The following practices are strictly prohibited:

👎 Contributing unknown or unreviewed code

👎 Failing to understand the implications of generated code

Summary: Contributors must fully understand their contributions and should not delegate critical thinking to coding agents.

5.1.5 👁 Code Quality

Project code quality is assessed through the following mechanisms:

- **Static Analysis**: Linting issues are identified by Qlty. Installation of Qlty CLI is recommended. Execute the command `qlty check --no-fail --sample 5 --no-formatters` to identify code quality issues. Note that Qlty only analyzes files that differ from the default branch (`feature/distributed-demo`).
- **Security Scans**: Code quality and security issues are identified by SonarQube.
- **Test Coverage**: Coverage reports are collected by Codecov.
- **Continuous Integration**: All GitHub Actions must complete successfully.

Qlty

Qlty performs static analysis, linting, and style checks to ensure optimal code quality for project contributions.

While newly introduced issues appear directly on the PR page, specific issues can be addressed by visiting the issues or code section of the Qlty dashboard.

Code contributions should not introduce new quality issues. If such issues arise, they should be resolved immediately using the appropriate suggestions from Qlty. In exceptional cases, an ignore flag may be added, though this approach should be employed sparingly.

Codecov

Codecov maintains test coverage metrics for the entire project. For detailed information about testing practices and workflows, refer to the testing documentation.

GitHub Actions

The project defines multiple GitHub Actions. All pull requests and direct commits must achieve successful status on all GitHub Actions.

5.1.6 References

[1] Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." Simulation 101.3 (2025): 287-311.

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
`PACKAGE_NAME`	The name with which libms npm package must be published

Similarly, Github actions require the following secrets to be obtained from npmjs:

Secret Name	Explanation
`NPM_TOKEN`	Token to publish npm packages to the default npm registry.
`PACKAGE_NAME`	The name with which libms npm package must be published
`NPM_RUNNER_PACKAGE_NAME`	The name with which runner npm package must be published

Remember to add these secrets to Github Secrets Setting of your fork.

5.4 System

5.4.1 🏰 System Overview

The Digital Twin as a Service (DTaaS) platform is designed to support the complete digital twin (DT) lifecycle, enabling users to create, configure, execute, and share digital twins through reusable assets[1]. The platform architecture reflects established principles for realising digital twins in practice[2], while also supporting advanced use cases such as runtime verification of autonomous systems[3].

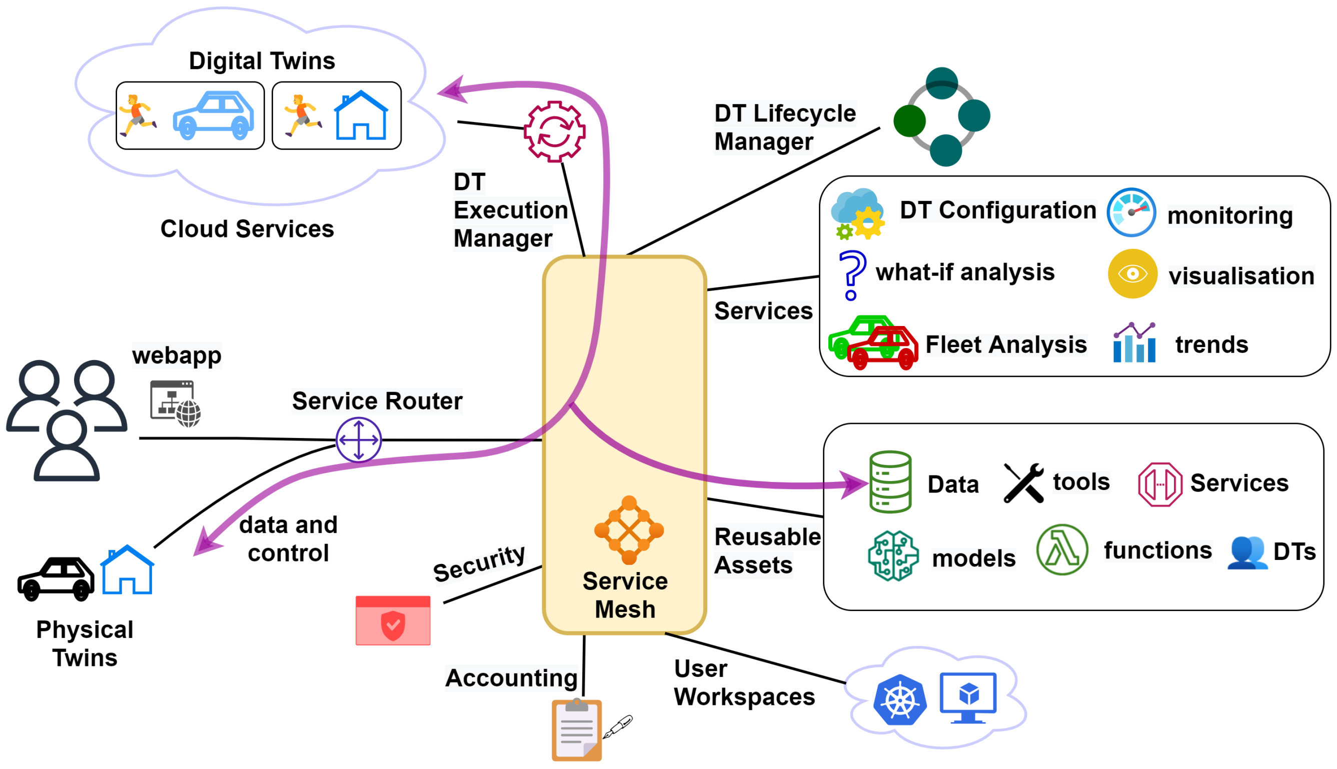
User Requirements

The platform provides the following core capabilities:

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 is 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

Users interact with the software platform through a web application. The service router serves as the single point of entry for direct access to platform services and 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.

For a more detailed view, refer to 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
Platform Services	Third-party Services (MQTT, InfluxDB, RabbitMQ, Grafana, PostgreSQL, and ThingsBoard
DT Lifecycle Manager	Not available yet
Security	GitLab client OAuth 2.0 and server OAuth 2.0
Digital Twins as Services	DT Runner
Accounting	Not available yet
Execution Manager	Not available yet

References

Font sources: fileformat

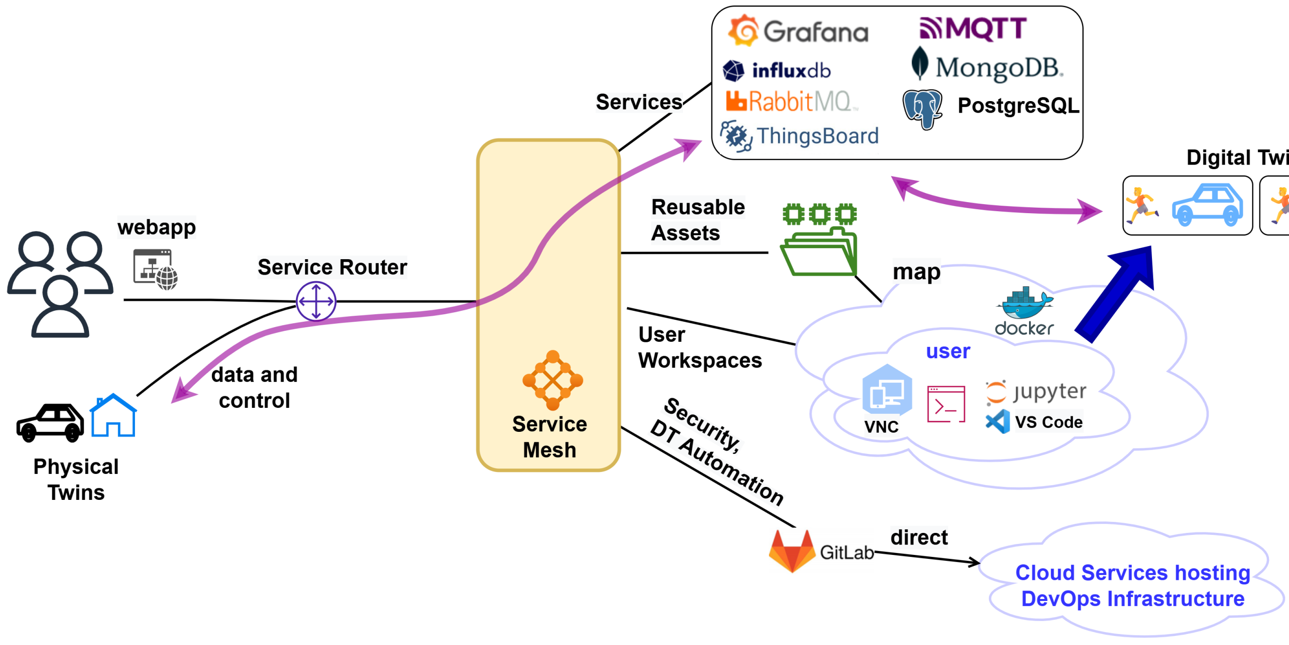
[1]: Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." Simulation 101.3 (2025): 287-311.

[2]: Talasila, Prasad, et al. "Realising digital twins." The engineering of digital twins. Cham: Springer International Publishing, 2024. 225-256.

[3]: Kristensen, Morten Haahr, et al. "Runtime Verification of Autonomous Systems Utilizing Digital Twins as a Service." 2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). IEEE, 2024.

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 2.0 protocol.

GATEWAY AUTHORIZATION

The Traefik gateway has OAuth 2.0 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: ThingsBoard, InfluxDB, Grafana, RabbitMQ, MQTT, MongoDB, and PostgreSQL

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:

- Create npm package for DevOps features of React Client
- Improve python package of platform services and DTaaS CLI
- Upgrade software stack of user workspaces
- Increased automation of installation procedures
- 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 Authentication Microservice (Auth MS) operates according to the OAuth 2.0 RFC specification. This document provides a brief summary of the OAuth 2.0 technology and its implementation within the DTaaS platform.

Entities

OAuth 2.0, 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 OAuth 2.0 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, the Authorization server (which requests user approval) and the Resource (User Identity) provider can be two different servers. However, in the DTaaS implementation, the GitLab instance handles both functions through different API endpoints. The underlying concepts remain the same. Therefore, this discussion focuses on the three main entities: the User, the OAuth 2.0 Client, and the GitLab instance.

THE OAUTH 2.0 CLIENT

Many platforms allow the initialization of an OAuth 2.0 client. For the DTaaS implementation, GitLab is used by creating an "application" within GitLab. However, it is not necessary to initialize a client using the same platform as the identity provider; these are separate concerns. The DTaaS OAuth 2.0 client is initialized by creating and configuring a GitLab instance-wide application. There are two main elements in this configuration:

- **Redirect URI**: This is the URI to which users are redirected after they approve sharing information with the client.
- **Scopes:** These define the types and levels of access that the client can have over the user's profile. For the DTaaS, only the "read user" scope is required, which permits access to the user's profile information for identity verification.

After the GitLab application is successfully created, a Client ID and Client Secret are generated. These credentials can be used in any application, effectively making that application an OAuth 2.0 Client. For this reason, the Client Secret must never be shared. The DTaaS Auth MS uses this Client ID and Client Secret, thereby functioning as an OAuth 2.0 Client application capable of following the OAuth 2.0 workflow to verify user identity.

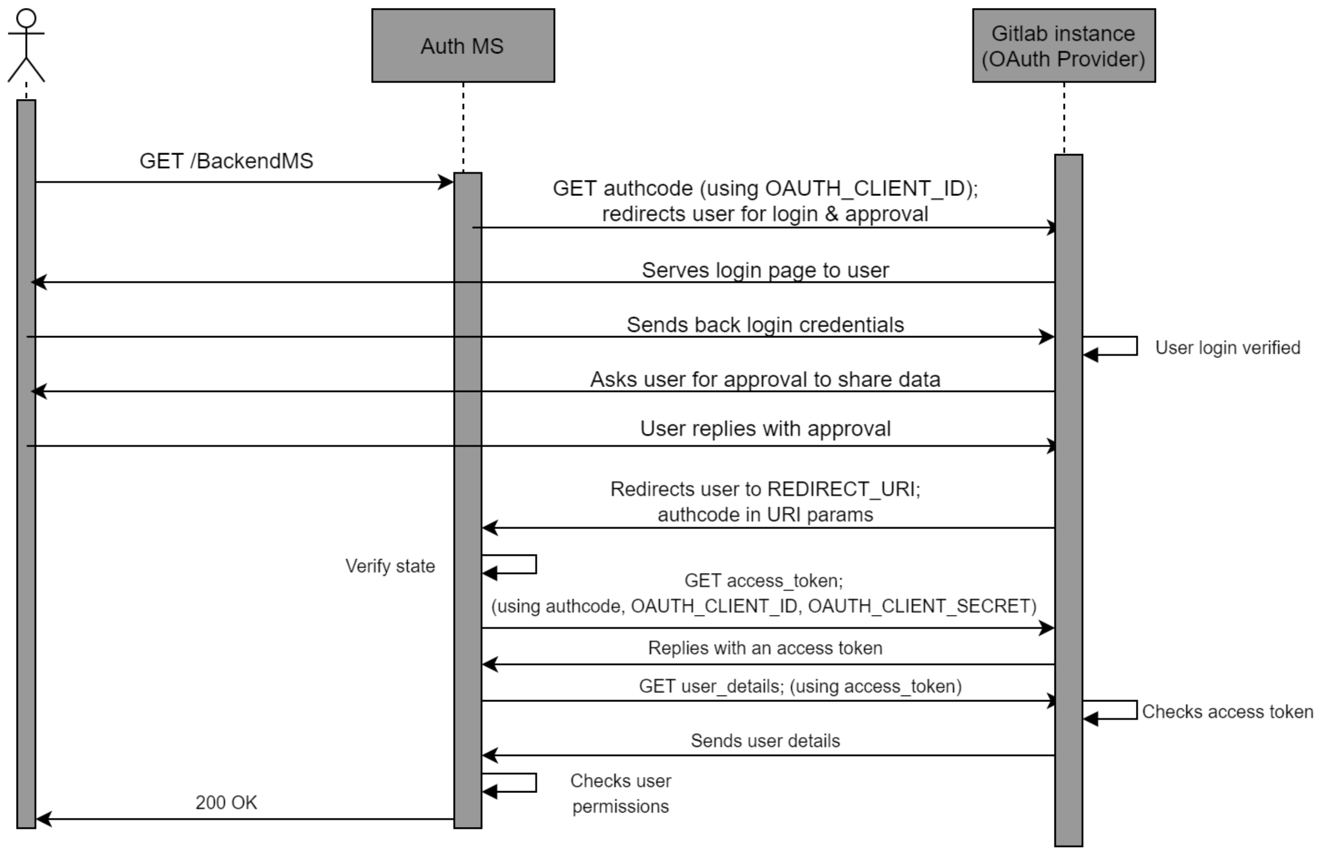
OAuth 2.0 Workflows

Two major OAuth 2.0 flows are employed in the DTaaS platform.

OAUTH 2.0 AUTHORIZATION CODE FLOW

This flow involves several steps and the exchange of an authorization code for access tokens to ensure secure authorization. This flow is used by the DTaaS Auth MS, which is responsible for securing all backend DTaaS services.

The OAuth 2.0 workflow is initiated by the Client (Auth MS) whenever user identity verification is required. The flow begins when the Auth MS sends an authorization request to GitLab. The Auth MS attempts to obtain an access token, which enables it to gather user information. Once user information is retrieved, the Auth MS can verify the user's identity and determine whether the user has permission to access the requested resource.


Auth MS
Gitlab instance (OAuth Provider)
GET /BackendMS
GET authcode (using OAUTH_CLIENT_ID); redirects user for login & approval
Serves login page to user
Sends back login credentials
User login verified
Asks user for approval to share data
User replies with approval
Redirects user to REDIRECT_URI; authcode in URI params
Verify state
GET access_token; (using authcode, OAUTH_CLIENT_ID, OAUTH_CLIENT_SECRET)
Replies with an access token
GET user_details; (using access_token)
Checks access token
Sends user details
Checks user permissions
200 OK

The requests made by the Auth MS to the OAuth 2.0 provider are shown in abbreviated form. A detailed explanation of the workflow specific to the DTaaS can be found in the Auth MS implementation documentation.

OAUTH 2.0 PKCE (PROOF KEY FOR CODE EXCHANGE) FLOW

PKCE is an extension to the OAuth 2.0 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, such as authorization code interception.

The DTaaS client website login is implemented using the PKCE OAuth 2.0 flow. Further details about this flow are available in the Auth0 documentation.

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

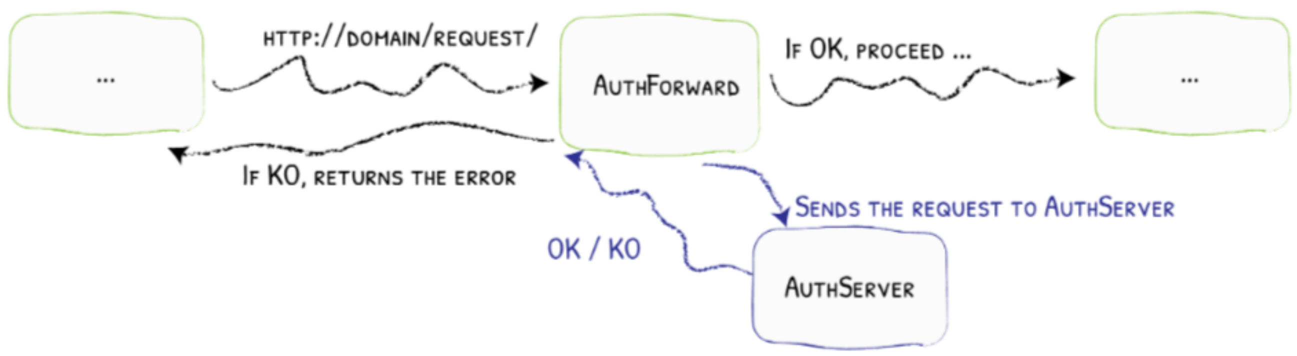
For our purpose, we require the Auth MS to be able to handle only requests of the general form ”Is User X allowed to access /BackendMS/example?”.

If the user’s identity is correctly verified though the GitLab OAuth 2.0 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 OAuth 2.0 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

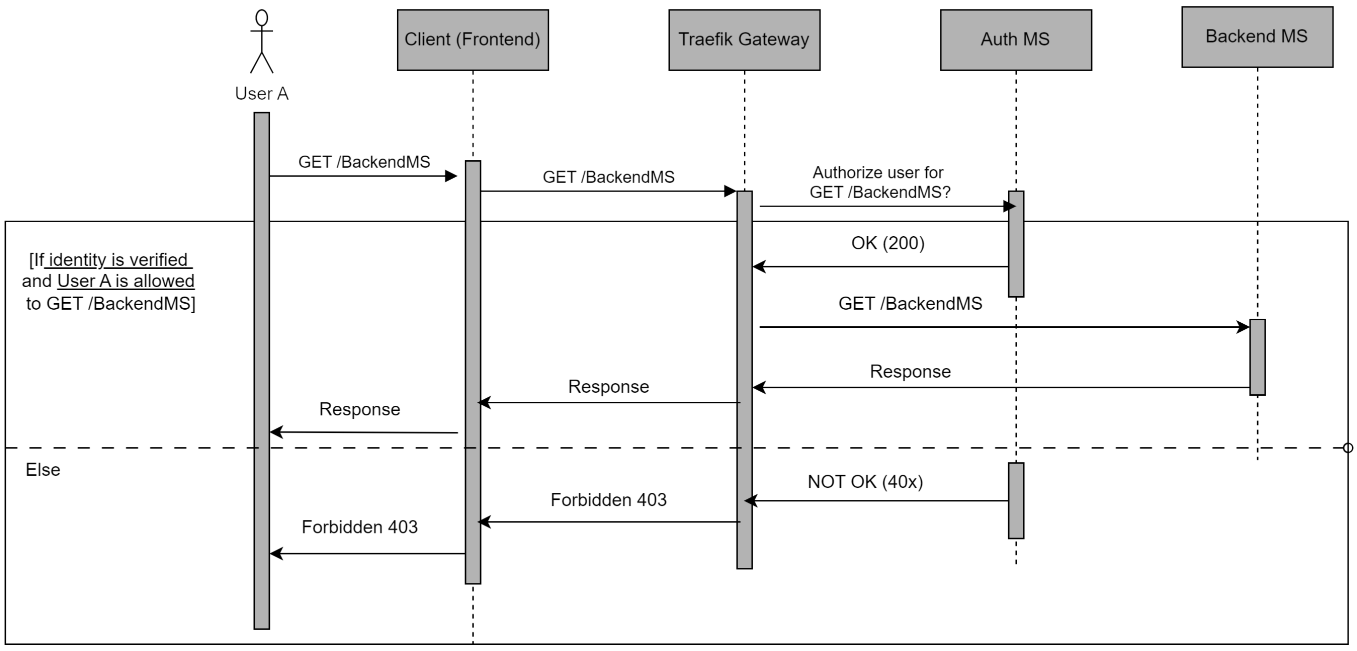


(source: Treafik documentation)

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 platform.
- 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 OAuth 2.0 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 OAuth 2.0 technology to be able to understand the content here better.

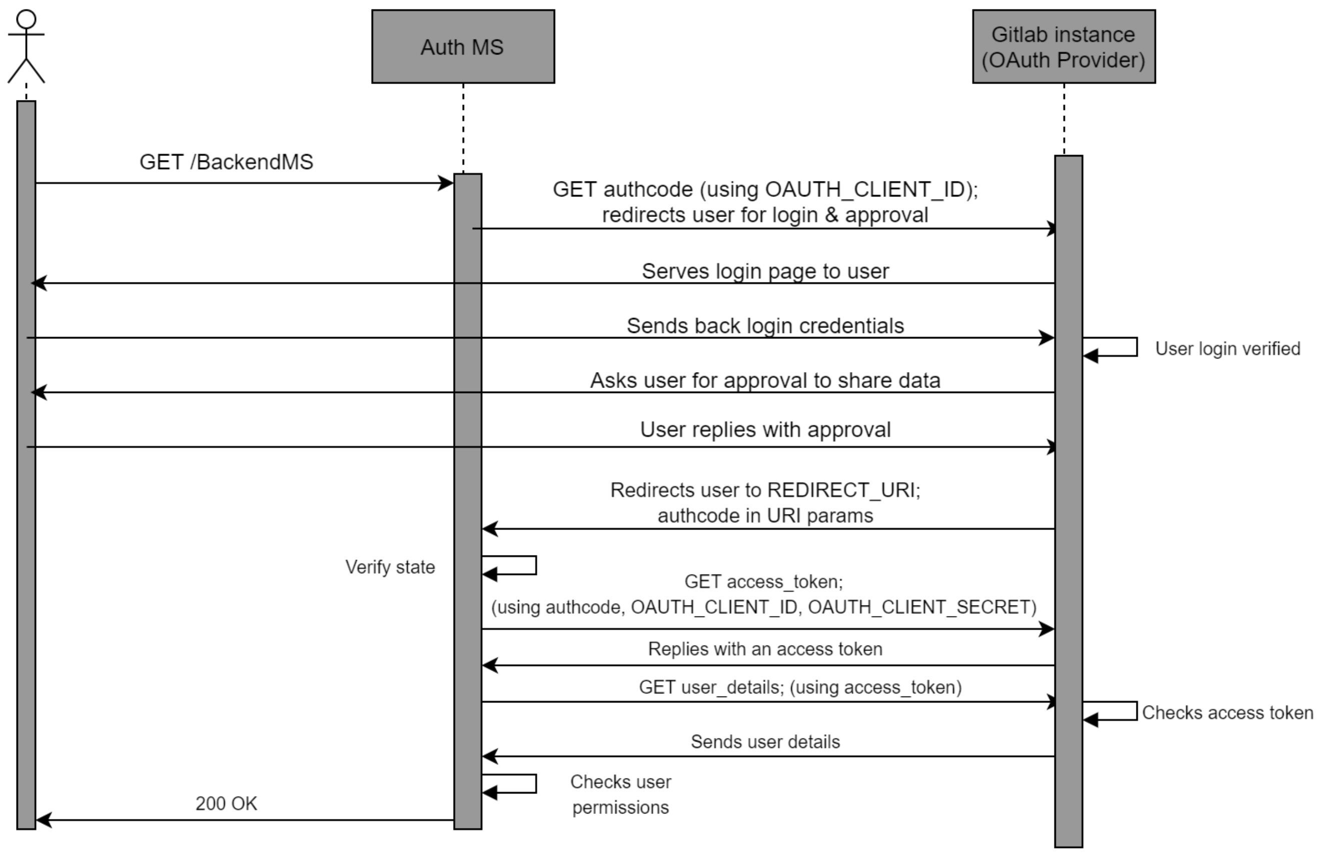
Workflow

USER IDENTITY USING OAUTH 2.0

We define some constants that will help with the following discussion:

- CLIENT ID: The OAuth 2.0 Client ID of the Auth MS
- CLIENT SECRET: The OAuth 2.0 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 OAuth 2.0 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)


Auth MS
Gitlab instance (OAuth Provider)
GET /BackendMS
GET authcode (using OAUTH_CLIENT_ID); redirects user for login & approval
Serves login page to user
Sends back login credentials
User login verified
Asks user for approval to share data
User replies with approval
Redirects user to REDIRECT_URI; authcode in URI params
Verify state
GET access_token; (using authcode, OAUTH_CLIENT_ID, OAUTH_CLIENT_SECRET)
Replies with an access token
GET user_details; (using access_token)
Checks access token
Sends user details
Checks user permissions
200 OK

A successful OAuth 2.0 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 OAuth 2.0 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 OAuth 2.0 provider redirects the user to the login page. Here the user logs into their protected account with their username/email ID and password.
- The OAuth 2.0 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 OAuth 2.0 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 OAuth 2.0 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 OAuth 2.0 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:

```yaml
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 OAuth 2.0 identification. If the action is set to ”auth”, requests on this route will require User identity OAuth 2.0 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 OAuth 2.0 process is carried out and the user identity is retrieved, but all known user identities (i.e. all users that successfully complete the OAuth 2.0) 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 2.0, 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 Fundamental Concepts in Software Testing

Definition of Software Testing

Software testing is a procedure to investigate the quality of a software product across different scenarios. It can also be defined 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 its design and development.

Importance of Software Testing

Software testing is essential for identifying defects and errors introduced during different development phases. Testing also ensures that the product under test works as expected across expected scenarios — a stronger test suite results in greater confidence in the product being built. One important benefit of software testing is that it enables developers to make incremental changes to source code while ensuring that current changes do not break the functionality of previously existing code.

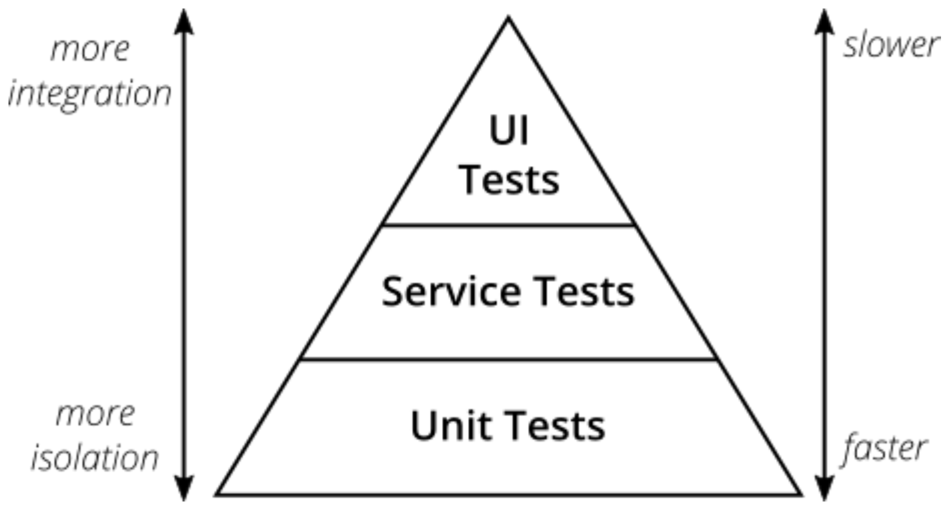
Test Driven Development (TDD)

Test Driven Development (TDD) 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 rather than validation. In other words, TDD provides a methodology for thinking through requirements or design before writing functional code.

Behaviour Driven Development (BDD)

Behaviour Driven Development (BDD) is a software development process that emerged from TDD. It includes the practice of writing tests first, but focuses on tests that describe behavior rather than tests that verify a unit of implementation. This approach provides software development and management teams with shared tools and a shared process for collaborating on software development. BDD is largely facilitated through the use of a simple domain-specific language (DSL) employing natural language constructs (e.g., English-like sentences) that can express behavior and 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)

The DTaaS project follows a testing workflow in accordance with the test pyramid diagram shown 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 and maintaining code. When good unit tests are written and executed every time code is changed, defects introduced due to the change can be promptly identified.
- When code is designed to be less interdependent to facilitate unit testing, the unintended impact of changes to any code is reduced.
- The cost, in terms of time, effort, and money, of fixing a defect detected during unit testing is lower compared to defects detected at higher levels.

UNIT TESTS IN DTAAS

Each component of the DTaaS project uses a unique technology stack; therefore, the packages used for unit tests differ across components. The `test/` directory of each component contains information about the specific unit test packages employed.

Integration Tests

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. The DTaaS project uses an integration server for software development as well as integration testing.

The existing integration tests are performed at the component level. Integration tests between components have not yet been implemented; this task has been deferred to future development.

End-to-End Tests

Testing code changes through the end-user interface of the software is essential to verify that the code produces the desired effect for users. End-to-End tests in DTaaS require a functional setup.

End-to-end testing capabilities in DTaaS are currently limited; further development of this testing layer has been deferred to future work.

Feature Tests

A software feature can be defined as changes made to the system to add new functionality or modify existing functionality. Each feature is characterized by being useful, intuitive, and effective. It is important to test new features upon implementation and to ensure that they do not break the functionality of existing features. Feature tests are therefore essential for maintaining software quality.

The DTaaS project does not currently include feature tests. Cucumber is planned for use in implementing feature tests in future development.

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 document describes the building and use of different Docker files for development and installation of the DTaaS platform.

NOTE: A local Docker CE installation is a prerequisite for using Docker workflows.

5.7.1 Folder Structure

There are four dockerfiles for building the containers:

- **client.dockerfile**: Dockerfile for building the client application container.
- **client.built.dockerfile**: Dockerfile for copying an already built client application into docker image. This dockerfile copies `client/build` directory and serves it from inside the docker 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.

In addition, there are docker compose and configuration files.

- **compose.dev.yml:** Docker Compose configuration for development environment.
- **.env**: environment variables for docker compose file
- **conf.dev** OAuth 2.0 configuration required by the Traefik forward-auth service

5.7.2 Build and Publish Docker Images

The github workflows publish docker images of client website and libms to github and docker hub.

Developer Usage

Docker images are useful for development purposes. Developers are advised to build the required images locally on their computers for use during development. The images can be built using

```
docker compose -f compose.dev.yml build
```

5.7.3 Running Docker Containers

The following steps describe how to use the application with Docker.

The DTaaS platform requires multiple configuration files. The list of configuration files to be modified is provided for each scenario.

Development Environment

This scenario is intended for software developers.

The following configuration files require updating:

1. **docker/.env** : Refer to the Docker installation documentation for guidance on updating this configuration file.
2. **docker/conf.dev** : Refer to the Docker installation documentation for guidance on updating this configuration file.
3. **client/config/local.js** : Refer to the client configuration documentation for guidance on updating this configuration file.
4. **servers/lib/config/libms.dev.yaml** : Refer to the library microservice configuration documentation for guidance on updating this configuration 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
```

Accessing the Application

The application should be accessed through the port mapped to the Traefik container, e.g., `localhost`.

5.8 Publish NPM packages

The DTaaS platform 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 the DTaaS platform.

The publication steps for library microservice and runner are 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 Command Line Interface

The Command Line Interface (CLI) provides administrators with programmatic control over the DTaaS platform. This administrative tool manages provisioning and deprovisioning of user accounts.

5.9.1 Package Structure

The CLI package is organized as follows:

```
cli/
├── src/                          # Application source code
│   ├── __init__.py                # Package initialization
│   ├── cmd.py                     # Click command groups and handlers
│   └── pkg/                       # Package modules
│       ├── config.py              # Configuration management (TOML parsing)
│       ├── constants.py           # Constant definitions
│       ├── users.py               # User lifecycle operations
│       └── utils.py               # File I/O and utilities
├── tests/                        # Test suites
│   ├── test_cli.py                # CLI integration tests
│   ├── test_cmd.py                # Command handler tests
│   ├── test_config.py             # Configuration tests
│   ├── test_users.py              # User operations tests
│   ├── test_utils.py              # Utility function tests
│   └── data/                      # Test fixtures
├── examples/                     # Example configuration files
├── dtaas.toml                    # Default configuration file
├── users.local.yml               # Local deployment template
├── users.server.yml              # Server deployment template
├── users.server.secure.yml       # Secure server template
└── pyproject.toml                # Poetry project configuration
```

5.9.2 Architecture and Design

The CLI is implemented as a Python package using the Click framework for command parsing. The design separates configuration management, user operations, and utility functions into distinct classes and utility functions.

```
classDiagram
    class dtaas {
        +admin() group
    }

    class admin {
        +user() group
    }

    class user {
        +add() command
        +delete() command
    }

    class Config {
        -data: dict
        +get_config() tuple
        +get_from_config(key) tuple
        +get_common() tuple
        +get_users() tuple
        +get_path() tuple
        +get_server_dns() tuple
        +get_add_users_list() tuple
        +get_delete_users_list() tuple
        +get_resource_limits() tuple
    }

    class users {
        +add_users(config_obj) error
        +delete_user(config_obj) error
        +get_compose_config(username, server, path, resources) tuple
        +create_user_files(users, file_path) void
        +start_user_containers(users) error
        +stop_user_containers(users) error
    }

    class utils {
        +import_yaml(filename) tuple
        +export_yaml(data, filename) error
        +import_toml(filename) tuple
        +replace_all(obj, mapping) tuple
        +check_error(err) void
    }
    dtaas --> admin : contains
    admin --> user : contains
    user --> Config : uses
    user --> users : calls
    users --> utils : uses
    Config --> utils : uses
```

Key Modules

Module	Purpose
`cmd.py`	Defines Click command groups and command handlers
`config.py`	Configuration management via TOML file parsing
`users.py`	User lifecycle operations (add/delete)
`utils.py`	File I/O and template substitution utilities

Configuration Module

The `Config` class serves as the central configuration manager, reading from `dtaas.toml`. It provides accessor methods that return tuples of `(value, error)`, following a Go-style error handling pattern that enables explicit error propagation without exceptions.

Key configuration sections include:

- **common**: Server DNS, installation path, and resource limits
- **users**: Lists of users to add or delete

User Operations Module

The user operations module manages the complete lifecycle of user workspaces:

1. **Workspace Creation**: Copies template directory structure for each user
2. **Container Configuration**: Generates Docker Compose service definitions with configurable resource limits (CPU, memory, shared memory, process limits)
3. **Container Orchestration**: Starts and stops user containers via Docker Compose commands

Utility Functions

The utilities module provides:

- **YAML/TOML I/O**: Safe file reading and writing operations
- **Template Substitution**: Recursive placeholder replacement in nested data structures (strings, lists, dictionaries)
- **Error Checking**: Helper function for converting errors to exceptions

5.9.3 Sequence Diagram

The following diagram illustrates the user addition workflow:

```
sequenceDiagram
    actor Admin
    participant CLI as Click CLI
    participant Config as Config Class
    participant Users as users.py
    participant Utils as utils.py
    participant Docker as Docker Compose
    participant FS as File System

    Admin ->> CLI: dtaas admin user add
    activate CLI

    CLI ->> Config: Config()
    activate Config
    Config ->> Utils: import_toml("dtaas.toml")
    Utils -->> Config: config data
    deactivate Config

    CLI ->> Users: add_users(config_obj)
    activate Users

    Users ->> Utils: import_yaml("compose.users.yml")
    Utils -->> Users: compose dict

    Users ->> Config: get_add_users_list()
    Config -->> Users: user list

    Users ->> Config: get_server_dns()
    Config -->> Users: server DNS

    Users ->> Config: get_path()
    Config -->> Users: installation path

    Users ->> Config: get_resource_limits()
    Config -->> Users: resource limits

    loop For each user
        Users ->> FS: Copy template directory
        Users ->> Users: get_compose_config()
        Users ->> Users: Add service to compose
    end

    Users ->> Utils: export_yaml(compose)
    Utils ->> FS: Write compose.users.yml

    Users ->> Docker: docker compose up -d
    Docker -->> Users: Container started

    Users -->> CLI: Success
    deactivate Users

    CLI -->> Admin: "Users added successfully"
    deactivate CLI
```

5.9.4 Error Handling Pattern

The CLI employs a consistent error handling strategy throughout the codebase:

1. **Functions return errors as values**: Most functions return a tuple of `(result, error)` rather than raising exceptions
2. **Explicit error propagation**: Callers check for errors and propagate them up the call stack
3. **Click exceptions at boundaries**: At the CLI entry points, errors are converted to `click.ClickException` for user-friendly output

This pattern provides explicit control flow and facilitates testing by making error paths explicit and testable.

5.9.5 Resource Limits Configuration

The CLI supports configurable resource limits for user containers, enabling administrators to control resource consumption per user and reduce the possibility of a single user making excessive use of limited computing resources. The default resource limits are:

Parameter	Description	Example Value
`shm_size`	Shared memory size	`512m`
`cpus`	CPU core allocation	`2`
`mem_limit`	Memory limit	`2g`
`pids_limit`	Maximum process count	`100`

5.10 React Website

The website serves as the primary interface through which end-users interact with the DTaaS platform. The application is implemented as a React single page web application, providing a modern and responsive user experience for digital twin management operations.

5.10.1 Architecture Overview

The client application follows a layered architecture pattern, separating concerns into distinct modules for routing, state management, UI components, and backend communication. The application employs Redux Toolkit for centralized state management and implements the Strategy pattern for backend abstraction, enabling support for multiple storage and execution backends.

Core Architecture Patterns

The backend communication layer implements interfaces, the builder pattern, and dependency injection to achieve backend-agnostic operations. Each domain object (e.g., `DigitalTwin`, `LibraryAsset`) receives its backend dependency at construction time, allowing different backends to be used interchangeably.

5.10.2 Package Structure

```
client/
├── src/                        # Application source code
│   ├── index.tsx                # Application entry point
│   ├── AppProvider.tsx          # Redux and theme providers
│   ├── routes.tsx               # Route definitions
│   ├── components/              # Reusable UI components
│   │   ├── asset/                # Asset-related components
│   │   ├── execution/            # Execution history components
│   │   ├── logDialog/            # Log display dialogs
│   │   ├── route/                # Route-specific components
│   │   └── tab/                  # Tab navigation components
│   ├── model/                   # Domain models and backend layer
│   │   └── backend/              # Backend abstraction layer
│   │       ├── interfaces/       # Interface definitions
│   │       ├── gitlab/           # GitLab implementation
│   │       ├── state/            # Backend state slices
│   │       └── util/             # Backend utilities
│   ├── page/                    # Page layout components
│   │   ├── Layout.tsx            # Main authenticated layout
│   │   ├── LayoutPublic.tsx      # Public pages layout
│   │   └── Menu.tsx              # Navigation menu
│   ├── route/                   # Feature route modules
│   │   ├── account/              # User account management
│   │   ├── auth/                 # Authentication flow
│   │   ├── config/               # Configuration pages
│   │   ├── digitaltwins/         # Digital twin management
│   │   ├── library/              # Library asset browsing
│   │   └── workbench/            # User workbench interface
│   ├── store/                   # Redux state management
│   │   ├── store.ts              # Store configuration
│   │   ├── auth.slice.ts         # Authentication state
│   │   ├── menu.slice.ts         # Menu navigation state
│   │   ├── settings.slice.ts     # Application settings
│   │   └── snackbar.slice.ts     # Notification state
│   ├── util/                    # Utility functions
│   │   ├── auth/                 # Authentication utilities
│   │   ├── configUtil.ts         # Configuration helpers
│   │   └── envUtil.ts            # Environment utilities
│   ├── preview/                 # DevOps preview features (need refactoring)
│   │   ├── components/           # Preview UI components
│   │   ├── route/                # Preview routes
│   │   ├── store/                # Preview state
│   │   └── util/                 # Preview utilities
│   └── database/                # Static configuration data
├── test/                       # Test suites
│   ├── unit/                    # Unit tests (Jest)
│   ├── integration/             # Integration tests
│   ├── e2e/                     # End-to-end tests (Playwright)
│   └── preview/                 # Preview feature tests
├── public/                     # Static assets
└── config/                     # Build configurations
    ├── dev.js                  # Development config
    ├── prod.js                 # Production config
    └── test.js                 # Test environment config
```

5.10.3 Key Components

Backend Abstraction Layer

The `model/backend/` directory implements a pluggable backend architecture:

Component	Purpose
`Backend`	Interface for server communication
`Instance`	Maintains backend connection state and logs
`DigitalTwin`	Domain model for digital twin operations
`LibraryAsset`	Domain model for library asset management
`FileHandler`	Handles file operations across backends

State Management

Redux slices manage application state:

Slice	Purpose
`auth.slice`	Authentication state and user session
`menu.slice`	Navigation menu visibility and selection
`settings.slice`	Application preferences and configuration
`snackbar.slice`	Toast notifications and alerts
`digitalTwin.slice`	Digital twin execution state

Route Modules

Feature modules organized by domain:

Module	Purpose
`digitaltwins/`	Create, execute, and monitor digital twins
`library/`	Browse and manage reusable assets
`workbench/`	Interactive user workspace interface
`account/`	User profile and settings management

5.11 Library Microservice

The Library Microservice exposes workspace files via two principal mechanisms: (1) a GraphQL API and (2) an HTTP file server accessible from web browsers. This service interfaces with the local file system and GitLab to provide uniform GitLab-compliant API access to files. The microservice serves as a critical component of the DTaaS platform, enabling both human users and digital twins to access reusable assets stored in the platform's library structure.

Warning

This microservice is still under heavy development. It is still not a good replacement for file server we are using now.

5.11.1 Architecture and Design

The microservice is built using NestJS framework with Apollo GraphQL. The architecture employs the Strategy pattern via a factory service, enabling support for multiple file storage backends (local filesystem, GitLab). The GraphQL API provided by the library microservice shall be compliant with the GitLab GraphQL service.

5.11.2 Package Structure

```
servers/lib/
├── src/                          # Application source code
│   ├── main.ts                   # Application entry point
│   ├── bootstrap.ts              # NestJS bootstrap configuration
│   ├── app.module.ts             # Root application module
│   ├── schema.gql                # Auto-generated GraphQL schema
│   ├── types.ts                  # GraphQL type definitions
│   ├── config/                   # Configuration module
│   │   ├── config.module.ts      # Configuration DI module
│   │   ├── config.service.ts     # Configuration service
│   │   ├── config.interface.ts   # Configuration interface
│   │   ├── config.model.ts       # Configuration model
│   │   └── util.ts               # Configuration utilities
│   ├── files/                    # Files module (core functionality)
│   │   ├── files.module.ts       # Files DI module
│   │   ├── files.resolver.ts     # GraphQL resolver
│   │   ├── files-service.factory.ts  # Backend factory
│   │   ├── interfaces/           # Service interfaces
│   │   │   └── files.service.interface.ts
│   │   ├── local/                # Local filesystem backend
│   │   │   ├── local-files.module.ts
│   │   │   └── local-files.service.ts
│   │   └── git/                  # GitLab backend
│   │       ├── git-files.module.ts
│   │       └── git-files.service.ts
│   ├── enums/                    # Enumeration definitions
│   │   └── config-mode.enum.ts   # Backend mode enum
│   └── cloudcmd/                 # CloudCmd file browser integration
├── test/                         # Test suites
└── dist/                         # Compiled output
```

5.11.3 Key Components

GraphQL Resolver

The `FilesResolver` class exposes two GraphQL queries:

Query	Purpose
`listDirectory`	Returns directory contents (files and folders)
`readFile`	Returns file content as raw text

File Service Interface

The `IFilesService` interface defines the contract for all file backends:

```
interface IFilesService {
  listDirectory(path: string): Promise<Project>;
  readFile(path: string): Promise<Project>;
  getMode(): CONFIG_MODE;
  init(): Promise<any>;
}
```

Backend Implementations

Backend	Purpose
`LocalFilesService`	Accesses files from local filesystem
`GitFilesService`	Accesses files from GitLab repository

Factory Pattern

The `FilesServiceFactory` creates the appropriate backend service based on configuration, enabling runtime selection of the storage backend.

5.11.4 UML Diagrams

Class Diagram

```
classDiagram
    class FilesResolver {
    -filesService: IFilesService
    +listDirectory(path: string): Promise<Project>
    +readFile(path: string): Promise<Project>
    }

    class FilesServiceFactory {
    -configService: ConfigService
    -localFilesService: LocalFilesService
    +create(): IFilesService
    }

    class LocalFilesService {
    -configService: ConfigService
    -getFileStats(fullPath: string, file: string): Promise<Project>
    +listDirectory(path: string): Promise<Project>
    +readFile(path: string): Promise<Project>
    }

    class ConfigService {
    +get(propertyPath: string): any
    }

    class IFilesService{
    listDirectory(path: string): Promise<Project>
    readFile(path: string): Promise<Project>
    }

    IFilesService <|-- FilesResolver: uses
    IFilesService <|.. LocalFilesService: implements
    IFilesService <|-- FilesServiceFactory: creates
    ConfigService <|-- FilesServiceFactory: uses
    ConfigService <|-- LocalFilesService: uses
```

Sequence Diagram

```
sequenceDiagram
    actor Client
    actor Traefik

    box LightGreen Library Microservice
    participant FR as FilesResolver
    participant FSF as FilesServiceFactory
    participant CS as ConfigService
    participant IFS as IFilesService
    participant LFS as LocalFilesService
    end

    participant FS as Local File System DB
    Client ->> Traefik : HTTP request
    Traefik ->> FR : GraphQL query
    activate FR

    FR ->> FSF : create()
    activate FSF

    FSF ->> CS : getConfiguration("MODE")
    activate CS

    CS -->> FSF : return configuration value
    deactivate CS

    alt MODE = Local
    FSF ->> FR : return filesService (LFS)
    deactivate FSF

    FR ->> IFS : listDirectory(path) or readFile(path)
    activate IFS

    IFS ->> LFS : listDirectory(path) or readFile(path)
    activate LFS

    LFS ->> CS : getConfiguration("LOCAL_PATH")
    activate CS

    CS -->> LFS : return local path
    deactivate CS

    LFS ->> FS : Access filesystem
    alt Filesystem error
        FS -->> LFS : Filesystem error
        LFS ->> LFS : Throw new InternalServerErrorException
        LFS -->> IFS : Error
    else Successful file operation
        FS -->> LFS : Return filesystem data
        LFS ->> IFS : return Promise<Project>
    end
    deactivate LFS
    activate IFS
    end

    alt Error thrown
    IFS ->> FR : return Error
    deactivate IFS
    FR ->> Traefik : return Error
    Traefik ->> Client : HTTP error response
    else Successful operation
    IFS ->> FR : return Promise<Project>
    deactivate IFS
    FR ->> Traefik : return Promise<Project>
    Traefik ->> Client : HTTP response
    end

    deactivate FR
```

5.12 Runner Microservice

The Runner microservice provides script execution capabilities for digital twins within the DTaaS platform. This service accepts HTTP requests to execute pre-configured commands and returns execution status and logs to the caller.

5.12.1 Package Structure

```
servers/execution/runner/
├── src/                          # Application source code
│   ├── main.ts                   # Application entry point
│   ├── app.module.ts             # Root NestJS module
│   ├── app.controller.ts         # REST API controller
│   ├── execa-manager.service.ts   # Command execution manager
│   ├── execa-runner.ts           # Command runner implementation
│   ├── queue.service.ts          # Command queue service
│   ├── runner-factory.service.ts  # Runner instance factory
│   ├── validation.pipe.ts        # Request validation pipe
│   ├── config/                   # Configuration module
│   │   ├── commander.ts           # CLI argument parsing
│   │   ├── config.interface.ts    # Configuration interface
│   │   ├── configuration.service.ts  # Configuration service
│   │   └── util.ts                # Configuration utilities
│   ├── dto/                      # Data transfer objects
│   │   └── command.dto.ts         # Command request DTO
│   └── interfaces/               # Interface definitions
│       ├── command.interface.ts   # Command types
│       └── runner.interface.ts    # Runner interface
├── api/                          # API documentation
├── test/                         # Test suites
└── dist/                         # Compiled output
```

5.12.2 Architecture and Design

The microservice is built using NestJS framework and employs the `execa` library for command execution. The architecture implements a command queue pattern for managing execution requests and a factory pattern for creating runner instances.

Core Components

```
classDiagram
    class AppController {
        -manager: ExecaManager
        +getHistory(): ExecuteCommandDto[]
        +newCommand(dto): void
        +cmdStatus(): CommandStatus
    }

    class ExecaManager {
        -commandQueue: Queue
        -config: Config
        +newCommand(name): [boolean, Map]
        +checkStatus(): CommandStatus
        +checkHistory(): ExecuteCommandDto[]
    }

    class Queue {
        -queue: Command[]
        +enqueue(command): void
        +checkHistory(): ExecuteCommandDto[]
        +activeCommand(): Command
    }

    class RunnerFactory {
        +create(command): Runner
    }

    class ExecaRunner {
        -command: string
        -stdout: string
        -stderr: string
        +run(): Promise~boolean~
        +checkLogs(): Map
    }

    class Config {
        -configValues: ConfigValues
        +loadConfig(options): void
        +permitCommands(): string[]
        +getPort(): number
        +getLocation(): string
    }

    AppController --> ExecaManager : uses
    ExecaManager --> Queue : uses
    ExecaManager --> Config : uses
    ExecaManager --> RunnerFactory : uses
    RunnerFactory --> ExecaRunner : creates
```

5.12.3 Key Components

REST API Controller

The `AppController` exposes three endpoints:

Endpoint	Method	Purpose
`/`	POST	Submit a new command for execution
`/`	GET	Get status of the current command
`/history`	GET	Retrieve execution history

Execution Manager

The `ExecaManager` service orchestrates command execution:

1. Validates command against permitted commands list
2. Creates runner instance via factory
3. Queues command for execution
4. Returns execution logs and status

Command Queue

The `Queue` service maintains execution history and tracks the currently active command. Commands are stored with their execution status (valid/invalid).

Runner Interface

The `Runner` interface defines the contract for command executors:

```
interface Runner {
  run(): Promise<boolean>;
  checkLogs(): Map<string, string>;
}
```

Configuration

The service reads configuration from a YAML file specifying:

Setting	Purpose
`port`	HTTP server port
`location`	Base directory for executable scripts
`commands`	Whitelist of permitted command names

5.12.4 Security Considerations

The runner implements a whitelist-based security model where only commands explicitly listed in the configuration file may be executed. This prevents arbitrary command execution attacks.

5.13 DevOps Framework

5.13.1 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 2.0 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.

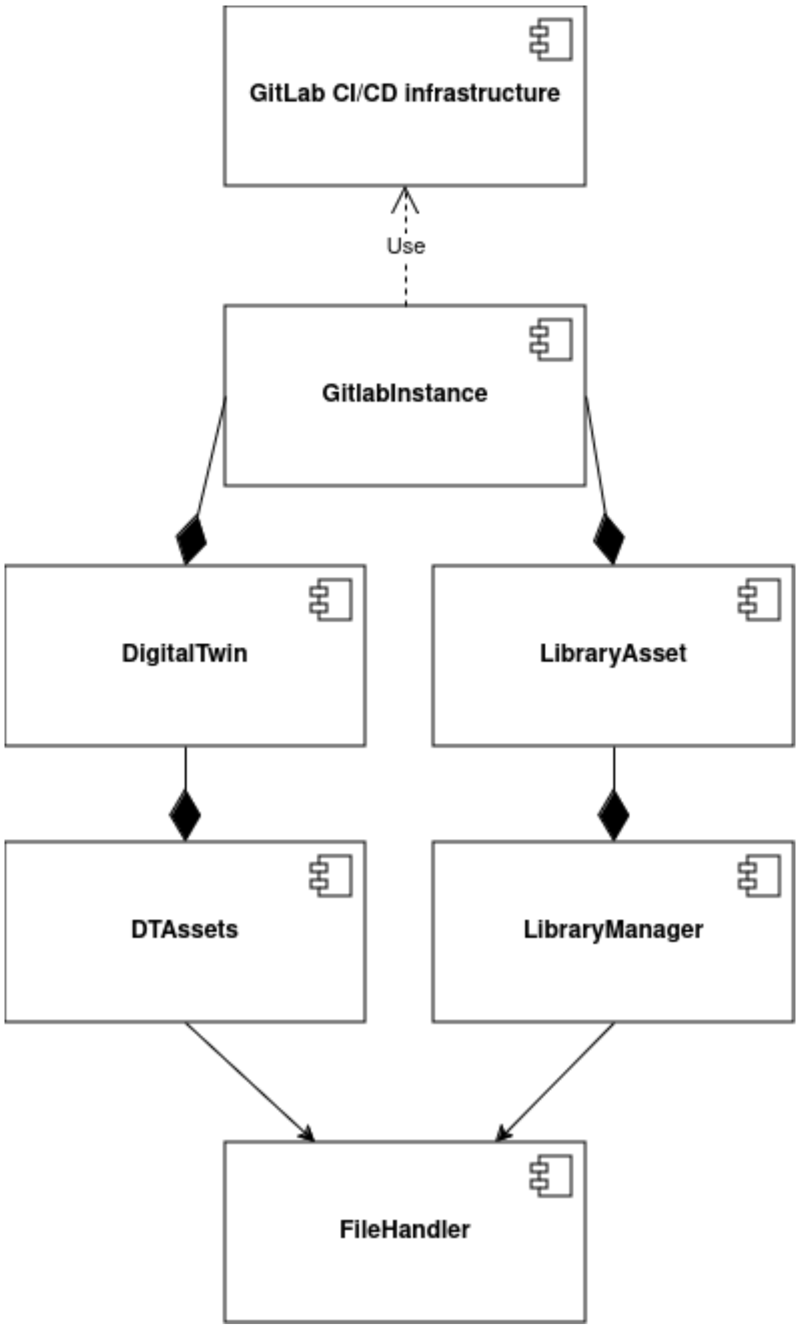
High Level Architecture

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 2.0 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.

5.13.2 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
```

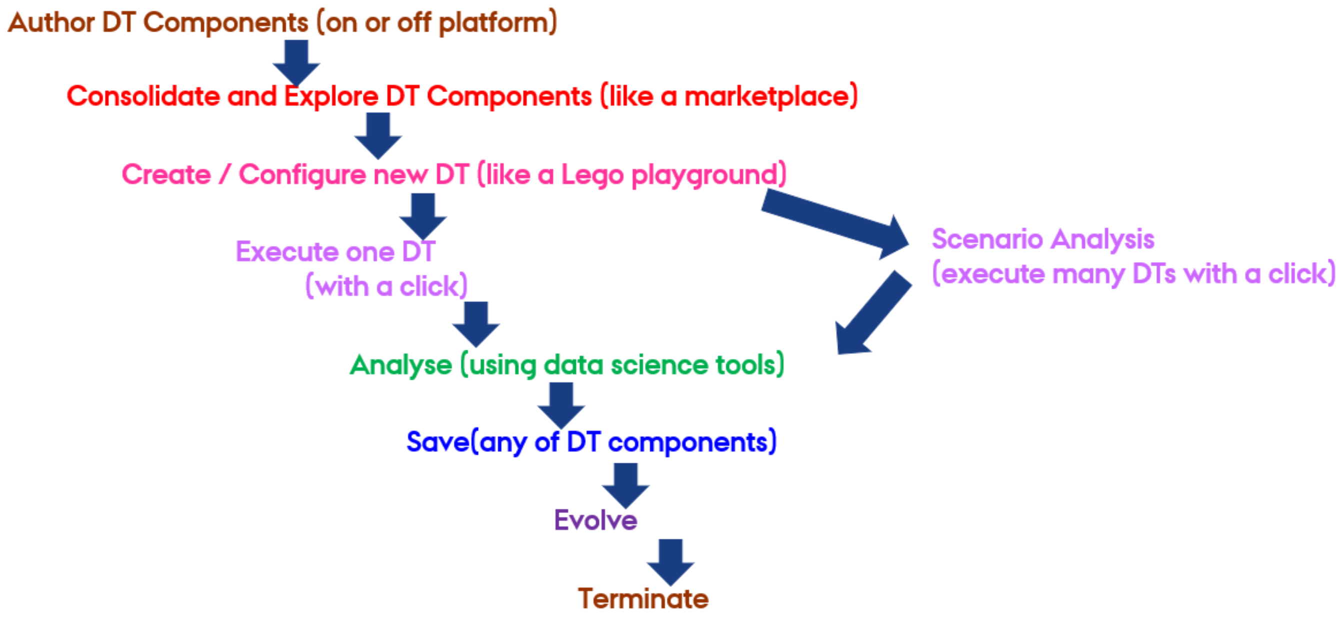
Digital Twin Structure

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.


Author DT Components (on or off platform)
Consolidate and Explore DT Components (like a marketplace)
Create / Configure new DT (like a Lego playground)
Execute one DT (with a click)
Scenario Analysis (execute many DTs with a click)
Analyse (using data science tools)
Save(any of DT components)
Evolve
Terminate

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:

```yaml
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.

5.13.3 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.

5.13.4 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,
  );
  getExecutionLogs();
  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 platform.

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 2.0 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. Known Issues in the Software

If a bug is discovered, an issue should be opened

6.1 Third-Party Software

The explanation given below corresponds to bugs that may be encountered from third party software included in the DTaaS platform. Known issues are listed below.

6.2 GitLab

The GitLab OAuth 2.0 authorization service does not have a way to sign out of a third-party application. Even if a user signs out of the DTaaS platform, GitLab still shows the user as signed in. The next time the sign in button is clicked on the the DTaaS platform page, the login page is not displayed. Instead the user is directly taken to the **Library** page. Therefore, the browser window should be closed after use. Another way to overcome this limitation is to open the GitLab instance (`https://gitlab.foo.com`) and sign out from there. Thus the user needs to sign out of two places, namely the DTaaS platform and GitLab, in order to completely exit the the DTaaS platform.

7. Thanks

7.1 Funding Sources

This project has been funded by the following projects.

- Security for the Digital Twin as a Service Platform - a project sponsored by Thomas B. Thriges Foundation
- CP-SENS (Cyber-Physical Sensing for Machinery and Structures) - a Grand Solution project funded by the Innovation Fund Denmark under the grant agreement 2081-00006B.
- The DIGITbrain (Digital Twins for Manufacturing SMEs) - a European Union's Horizon 2020 research and innovation program under grant agreement No 952071.
- Digital Twins for Cyber-Physical Systems (DiT4CPS) - a project sponsored by the Grundfos Foundation.

7.2 Developers

Please see the list of contributors on code contributors

7.3 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

7.4 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

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-CPS 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-CPS 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 platform utilizes numerous third-party software components. These software components have their own licenses.

8.2.1 User Installations

The software included with the DTaaS installation scripts is listed below:

Software Package	Usage	License
Docker CE	mandatory	Apache 2.0 License
ml-workspace-minimal	mandatory	Apache 2.0 License
NodeJS	mandatory	Custom - Modified MIT
npm	mandatory	Artistic License 2.0
serve	mandatory	MIT
Træfik	mandatory	MIT License
Yarn	mandatory	BSD 2-Clause License
Eclipse Mosquitto	optional	Eclipse Public License-2.0
GitLab CE	optional	MIT License
Grafana	optional	GNU Affero General Public (AGPL) License v3.0
InfluxDB	optional	Apache 2.0 License, MIT License
Mongodb	optional	AGPL License and Server Side Public License (SSPL) v1
RabbitMQ	optional	Mozilla Public License
Telegraf v1.28	optional	MIT License
ThingsBoard	optional	PostgreSQL License

8.2.2 Development Environments

In addition to all 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 License
Jupyter Lab	optional	BSD 3-Clause License
Microk8s v1.27	optional	Apache 2.0 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.