Co-funded by the Horizon 2020
Framework Programme of the European Union

Deliverable 5.4
First Version of TeNDER Platform

Work Package 5: Services Integration and Technical Validation

affecTive basEd iNtegrateD carE for betteR Quality of Life: TeNDER Project

Grant Agreement ID: 875325

Start date: 1 November 2019

End date: 31 October 2022

Funded under programme(s): H2020-SC1-DTH-2018-2020/H2020-SC1-DTH-2019

Topic: SC1-DTH-11-2019 Large Scale pilots of personalised & outcome based integrated care

Funding Scheme: IA - Innovation action

Disclaimer

The TeNDER consortium consists of the following Partners.

Table 1 - Consortium Partners List

No	Name	Short name	Country
1	UNIVERSIDAD POLITECNICA DE MADRID	UPM	Spain
2	MAGGIOLI SPA	MAG	Italy
3	DATAWIZARD SRL	DW	Italy
4	UBIWHERE LDA	UBI	Portugal
5	ELGOLINE DOO	ELGO	Slovenia
6	ETHNIKO KENTRO EREVNAS KAI TECHNOLOGIKIS ANAPTYXIS	CERTH	Greece
7	VRIJE UNIVERSITEIT BRUSSEL	VUB	Belgium
8	FEDERATION EUROPEENNE DES HOPITAUX ET DES SOINS DE SANTE	HOPE	Belgium
9	SERVICIO MADRILENO DE SALUD	SERMAS	Spain
10	SCHON KLINIK BAD AIBLING SE & CO KG	SKBA	Germany
11	UNIVERSITA DEGLI STUDI DI ROMA TOR VERGATA	UNITOV	Italy
12	SLOVENSKO ZDRUZENJE ZA POMOC PRI DEMENCI - SPOMINCICA ALZHEIMER SLOVENIJA	SPO	Slovenia
13	ASOCIACION PARKINSON MADRID	APM	Spain

Document Information

Project short name and Grant Agreement ID	TeNDER (875325)
Work package	WP5
Deliverable number	D5.4
Deliverable title	First version of TeNDER platform
Responsible beneficiary	MAG
Involved beneficiaries	UPM, DW, UBI, ELG, CERTH
Type[1]	DEM
Dissemination level[2]	PU
Contractual date of delivery	31 August 2021
Last update	31 August 2021

[1] **R:** Document, report; **DEM:** Demonstrator, pilot, prototype; **DEC:** Websites, patent fillings, videos, etc.; **OTHER**; ETHICS: Ethics requirement; ORDP: Open Research Data Pilot.

[2] **PU:** Public; **CO:** Confidential, only for members of the consortium (including the Commission Services).

Document History

Version	Date	Status	Authors, Reviewers	Description
v 0.1	16/06/2021	Draft	Panos Karkazis (MAG)	Project deliverable template.
v 0.2	09/07/2021	Draft	Thanasis Makropoulos, Dimitris Papadopoulos (CERTH)	Information on 5.1 and 5.1.1
V 0.3	13/07/2021	Draft	Luís Santos (UBI)	Part of section 5.2.2 and 5.2.3
V0.4	20/07/2021	Draft	Gustavo Hernández (UPM)	General review and contributions to 5.1.2
V0.5	22/07/2021	Draft	Luís Santos (UBI)	Sections 5.2.5 and 5.2.8 without tests
V0.6	29/07/2021	Draft	Luís Santos (UBI)	Adding tests in Section 5.2.5
V0.7	1/08/2021	Draft	Panos Karkazis (MAG)	Add content on monitoring section
V0.8	4/08/2021	Draft	Luís Santos (UBI)	Adding tests in Section 5.2.3
V0.9	5/08/2021	Draft	Luís Santos (UBI)	Adding tests in Section 5.2.2
V0.10	6/08/2021	Draft	Luís Santos (UBI)	HL7 API Description
V0.11	16/08/2021	Draft	Panos Karkazis (MAG)	General review contributions
V0.12	25/08/2021	Draft	Paride Criscio (DW)	Contribution section 5.3
V0.13	26/08/2021	Draft	Panos Karkazis (MAG)	Review, Editing
V0.14	30/08/2021	Draft	Thanasis Makropoulos, Dimitris Papadopoulos (CERTH)	Review, Editing
V1.0	31/08/2021	Final	Gustavo Hernández	Final peer Review

Acronyms and Abbreviations

Acronym/Abbreviation	Description
APIs	Application Programming Interfaces
ATDD	Acceptance Dest Driven Development
CD	Continuous Delivery
CI	Continuous Integration
DMS	Data Management System
EHR	Electronic Health Record
FHIR	Fast Healthcare Interoperability Resources
GUI	Graphical User Interface
HAPI	HL7 Application Programming Interface
HL7	Health Level 7
HLS	High-level Services
HTTP	Hypertext Transfer Protocol
LLS	Low Level Subsystem
NBI	North Bound Interface
QoS	Quality of Service
REST	Representational State Transfer
SBI	South Bound Interface
TeNDER	affecTive basEd iNtegrateD carE for betteR Quality of Life
VM	Virtual Machine

Contents

List of Figure

List of Tables

Executive Summary

This deliverable describes the first version of the TeNDER platform as an integrated open ecosystem based on the requirements defined in WP1 and WP2. It also discusses the tools that are used to support Continuous Integration (CI)/Continuous Delivery (CD) and testing in the context of the Tasks 5.4 and 5.5. Moreover, a brief description for each component of TeNDER platform is provided and a detailed presentation of the selected tools and the development, production, and monitoring environments that are used is given.

These tools support the platform development and guarantee the allocation of the appropriate resources for the deployment and execution of the services. Furthermore, this document provides the definition of several types of manual and automated testing procedures on component level as well as integration and qualification tests of the system platform. The first results from integration and validation tests prove that the platform is functional and meets the requirements for the first wave of piloting. As the development of the platform is in progress, the final version of the TeNDER platform and the future updates regarding the testing and validation procedures will be presented in the D5.5, which is the last deliverable of the WP5.

1 INTRODUCTION

This document presents the selected set of tools for supporting software development, technical validation and deployment of the TeNDER platform, based on CI/CD approach and a comprehensive resource allocation monitoring system.

The adoption of the CI/CD approach for the TeNDER software development enhances the co-design process and minimizes the elapsed time between the definition of the software requirements and their integration to the next software release, the so called "cycle time" [1]. In particular, the CI enables developers to regularly merge their code changes into a central repository and trigger automated procedures for building and testing their components in order to address bugs quickly and improve software quality. On the other hand, CD is the next step of CI, that enables the delivery of the component for system and integration testing and then for the release in production. This does not mean that every change is delivered automatically in production, but that there is a testing mechanism that can ensure that every change is applicable at any time [2]. According to the best practices for CI [3], the following tools should be part of a state-of-the-art CI framework:

- a tool for source version control
- a tool for automatic building, dependency checking and automatic testing
- a tool to keep tracking of the issues in order to fix them immediately
- a tool for automatic deployment, providing the capability to deploy on demand any version of software to any environment

In this context, pipelines have been defined for each one of the components which describe a typical workflow with the steps that source code goes through to make its way to production, and involves code building, testing, and deploying to any environment. All the code is kept on a binary repository that manages the version control and provides all the latest versions of the components for deployment on the staging environment, execution of the integration and quality acceptance tests before being deployed on the production environment. This phase is part of the CD and is done by specific scripts for packaging, deploying and changing configuration often called as configuration management tools. The most appropriate tools for the CD are (a) repository for binary distribution; (b) tools for deployment and test applications in any environment and (c) report mechanism for providing feedback to the developers and testers.

Furthermore, the knowledge of the utilization status of the available computational resources in every environment is crucial to guarantee that any service of the platform has the appropriate resources to function properly. In addition, the resource allocation per application is also interesting information for the detection of potential bugs like memory leaks etc. Therefore, the design and deployment of a comprehensive monitoring solution as an additional tool is mandatory for the validation of the platform and the assurance of the provided Quality of Service (QoS).

The document is structured as follows:

Section 1 is the introduction of the deliverable discussing the contribution and scope of the document.

Section 2 presents the CI/CD tools which are used for the development of the TeNDER platform as well as the different deployment environments.

Section 3 presents the testing procedures that have been implemented until now, focusing on the first version of system integration and qualification tests.

Section 4 describes the tools and the architecture of the monitoring system that has been deployed for the monitoring of resource allocation.

Section 5 provides a brief technical description of the components of the TeNDER platform focusing on the integration of each one of them with the CI/CD mechanism for automated build, test, and deployment.

Section 6 presents the central documentation server which provides documentation for all TeNDER RESTful APIs.

Section 7 concludes the document.

2 Hosting Infrastructure Components

This section presents the hosting infrastructure and the tools used for the development, testing, and deployment of the TeNDER services. TeNDER uses DevOps[3] and CI/CD approaches that enable the exploitation of the platform as an integrated ecosystem based on the requirements set by the WP6 regarding the three phases of piloting. Building on top of these technologies TeNDER enables the management and control of the DevOps cycle for the continuous deployment and integration of added value services and their components.
However, in order to support the above-mentioned activities, a consistent infrastructure was created which supports virtualisation of HW resources, i.e. processing power, memory, storage and network resources. For this reason, the approach used is the introduction of incremental steps towards integration, validation and testing of TeNDER components exploiting three environments (aka infrastructure versions). These infrastructures are: (a) Development infrastructure; (b) Stage infrastructure and (c) Production infrastructure. This section discusses mostly the deployment of these infrastructures as well as the description of the tools that are used in the context of TeNDER in MAG premises.

2.1 CI/CD workflow

The core component of the development process TeNDER is a private instance of GitLab [4] which provides a collaborative environment for software development, version control and CI/CD management. The private instance of the GitLab is hosted on a dedicated Virtual Machine (VM) in MAG cloud infrastructure in which every developer has two factor authorized access (Figure 1) and she/he can create her/his own repository and add other users as members. GitLab also provides a complete CI/CD framework which uses pipelines.

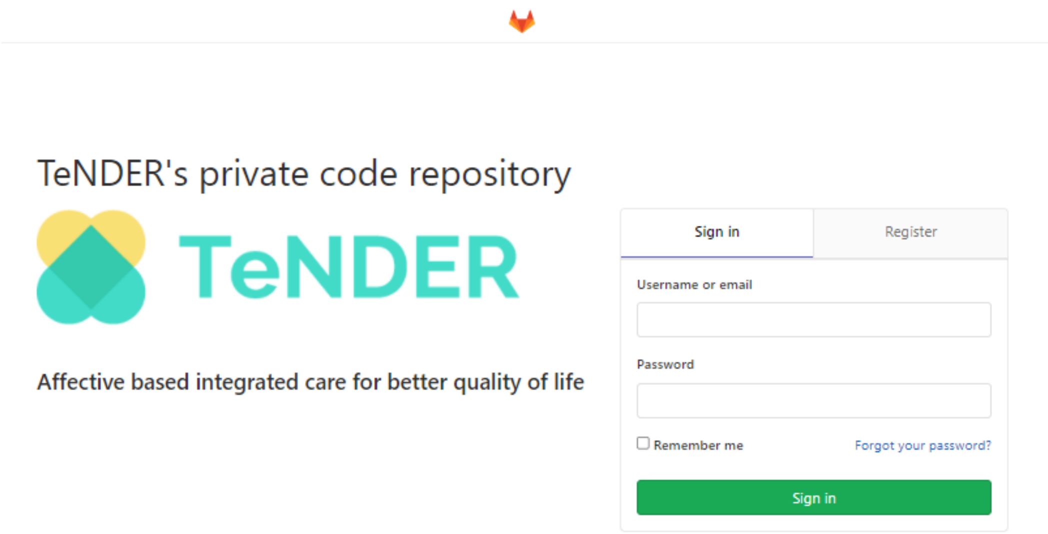

Figure 1 TeNDER's GitLab login

[3] Gitlab DevOps approach. Available at: [DevOps | GitLab]

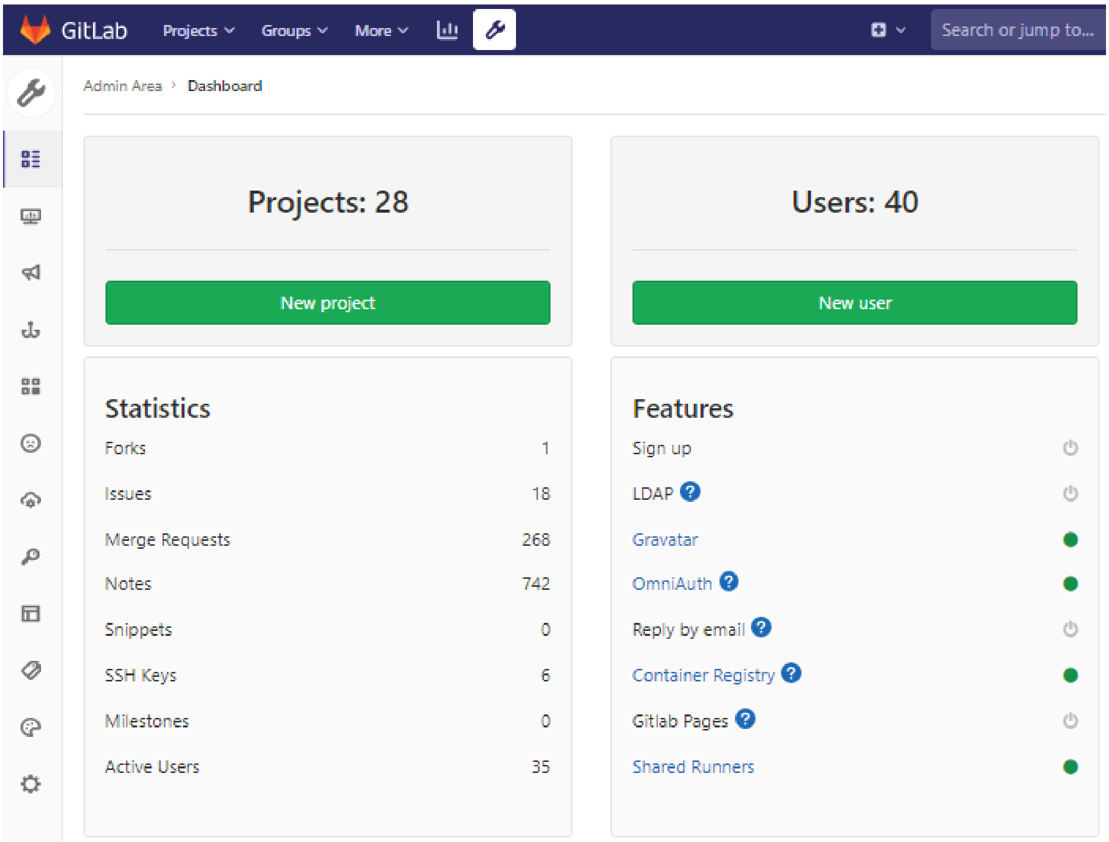

Figure 2 TeNDER GitLab users - projects

At this point, there are 28 active projects and 40 developers who created code for the TeNDER platform (Figure 2). A generic view of the CI/CD workflow is shown Figure 3, in which we can see how the GitLab tool in integrated with the stage and production environments based on CI/CD pipelines.

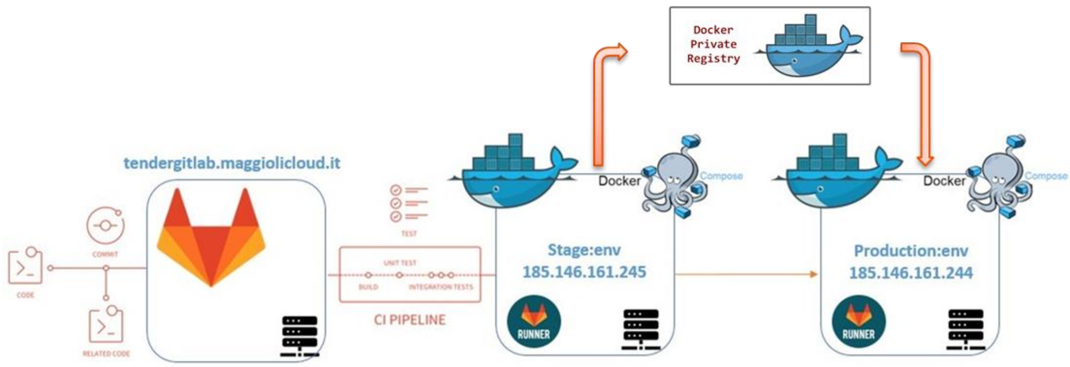

Figure 3 CI/CD workflow

2.1.1 CI/CD Pipelines

Every time a developer pushes small code chunks to a project hosted in a Git repository, she/he triggers a pipeline (Figure 4) of scripts to build, test, and validate the code changes before merging them into the main branch. Then, the CI/CD framework deploys the new version of the component to stage environment.

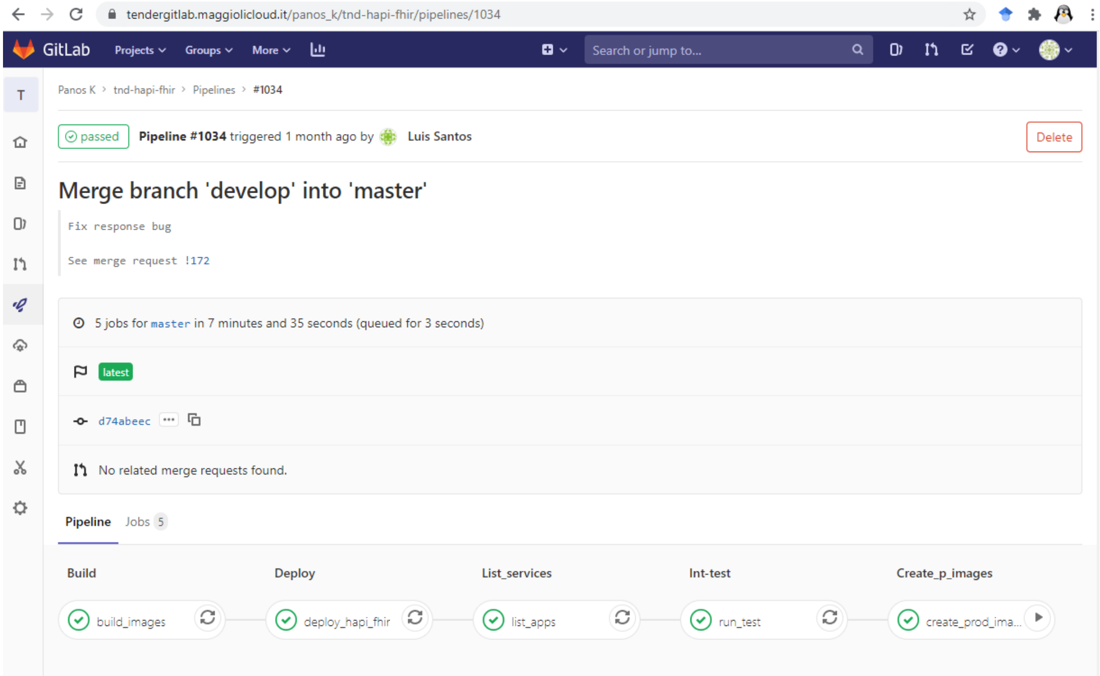

Figure 4 CI/CD pipeline

Each pipeline consists of a set of jobs which can also be triggered manually through the web interface by pressing the appropriate button. When the pipeline is triggered, the job is assigned to the runner process which links to the specific repository and executes it in environment of our choice. The output of the job is displayed in real time to the GitLab web interface. This set of steps can be shifted or skipped depending on developer's requirements. Next, we describe in detail the different steps of a typical the CI/CD pipeline.

2.1.1.1 Container build

TeNDER uses Docker containers as host for its components, so the first step is to build the images for all components that are developed in the repository and push them to the private docker registry of TeNDER. The building of a Docker container can be scripted as follows:

```
build_images:
  only:
    refs:
      - master
  stage: build
  script:
    - docker build -t hapi-fhir-server -f dockerfile/stage/Dockerfile .
    - docker tag hapi-fhir-server tender-registry:5000/hapi-fhir-server:sta
    - docker push tender-registry:5000/hapi-fhir-server:sta
  tags:
    - stage
```

Where:

- **`docker build`**: The instruction to build the container.
- **`-f Dockerfile`**: The location of the Dockerfile.

- **-t tender-registry:5000/<container_image_name>:** The name of the container image. The first part is the internal docker registry, and the second part is the image name.

2.1.1.2 Unit Tests

During the unit test stage developers can perform software testing of an individual unit or component. This kind of tests isolates a section of code and verify its correctness. The use of containers has a significant advantage in designing and executing unit tests. The developer is not required to create mock-ups of each component as it depends on his implementation. For example, in case of databases, sometimes it is time expensive to build a mock-up. With Docker, it is quick and straightforward to just start a docker container with the database and connect the under-test container to it. Once a container passes the unit tests, the test database can be easily removed.

2.1.1.3 Service deployment

During the deployment stage the containers of each service are deployed on the stage environment. The containers can be started separately, or the developers can use technologies like docker-compose, docker swarm etc.

```
deploy_hapi_fhir:
  only:
    refs:
      - master
  stage: deploy
  script:
    - docker-compose -f docker-compose-stage.yml up -d hapi-fhir-server
  tags:
    - stage
```

Where:

- **Docker-compose up**: The instruction to start the docker-compose.
- **-f docker-compose-stage.yml**: The location of the docker compose file.

2.1.1.4 Integration Tests

After the successful deployment of the service in stage environment the developers can trigger other CI/CD pipelines using the provided API from the GitLab. In TeNDER we use this functionality to execute end-to-end integration tests after the deployment of each service in the stage env. More details regarding the integration tests are available in section 3.2.

```
run_test:
    stage: int-test
    script:
        - apk update
        - apk add curl
        - curl -s -X POST
          -F token=0f2c5b4019231cd48f49fe229746f2 \
          -F ref=master \
          -F "variables[TEST_SCRIPT]=int-test-hfir.sh" \
         https://tendergitlab.maggiolicloud.it/api/v4/projects/32/trigger/pipeli
         ne
    tags:
     - stage
```

Where:

- **`curl POST <test_uri>`**: Trigger integration test through API
- **`-F token=<token>`**: Authorization token
- **-F ref=<branch>:** Repository branch
- **-F "variables[<variable_name>]=<value>":** Set variable value

2.1.1.5 Create production images

The images that are used in the production env are created from the last job of the pipeline of each repository. This job is triggered manually from the developer each time she/he decides to promote the current version of his service from the stage to the production environment. During this stage, the latest version of the stage images are tagged with the appropriate version number and they are pushed to private docker registry.

```
create_prod_images:
  only:
    refs:
      - master
  stage: create_p_images
  script:
    - docker rmi tender-registry:5000/hapi-fhir-server:prod || true
    - docker build -t tender-registry:5000/hapi-fhir-server:prod -f
dockerfile/prod/Dockerfile .
    - docker push tender-registry:5000/hapi-fhir-server:prod
  when: manual
  tags:
    - stage
```

The history of the executed jobs as well the outputs logs are kept by the CI/CD tool and are available to the developer at any time (Figure 5).

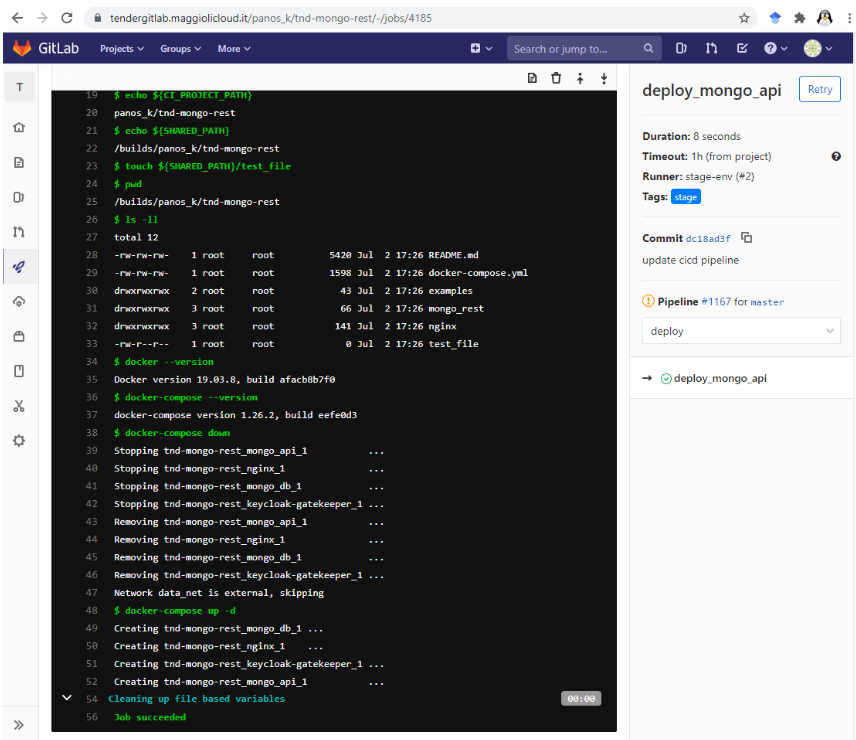

Figure 5 Job execution output

In case an error occurs during the execution of one of the jobs, the entire pipeline fails and a notification email is sent to the involved users (developers/maintainers etc) who triggered the pipeline either by pushing new code or manually through web interface (Figure 6).

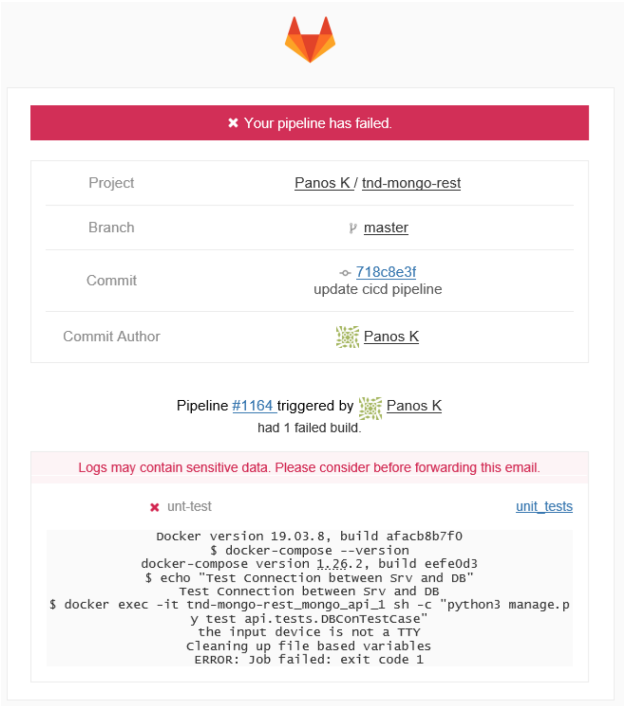

Figure 6 Error notification mail

Following these methodologies, developers are able to catch bugs and errors early in the development cycle and ensure that all the code deployed to production complies with the established code standards.

Another component of the CI/CD framework is private Docker registry (Figure 7) which is used for the storage of different versions of the TeNDER components for both stage and production environments. It is worth to mention that in TeNDER we also created a public repository (Figure 8) for container images hosted in Docker Hub for the services running on the Low Level Sub System (LLS).

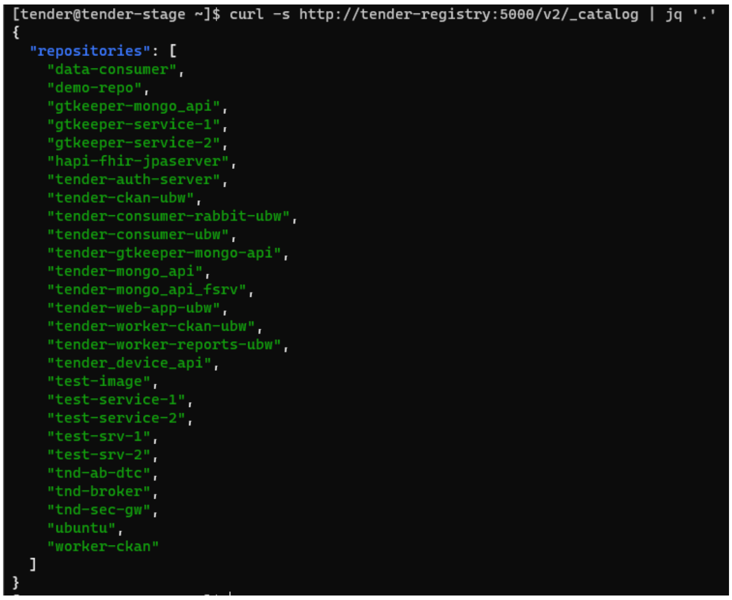

Figure 7 TeNDER's private docker instance

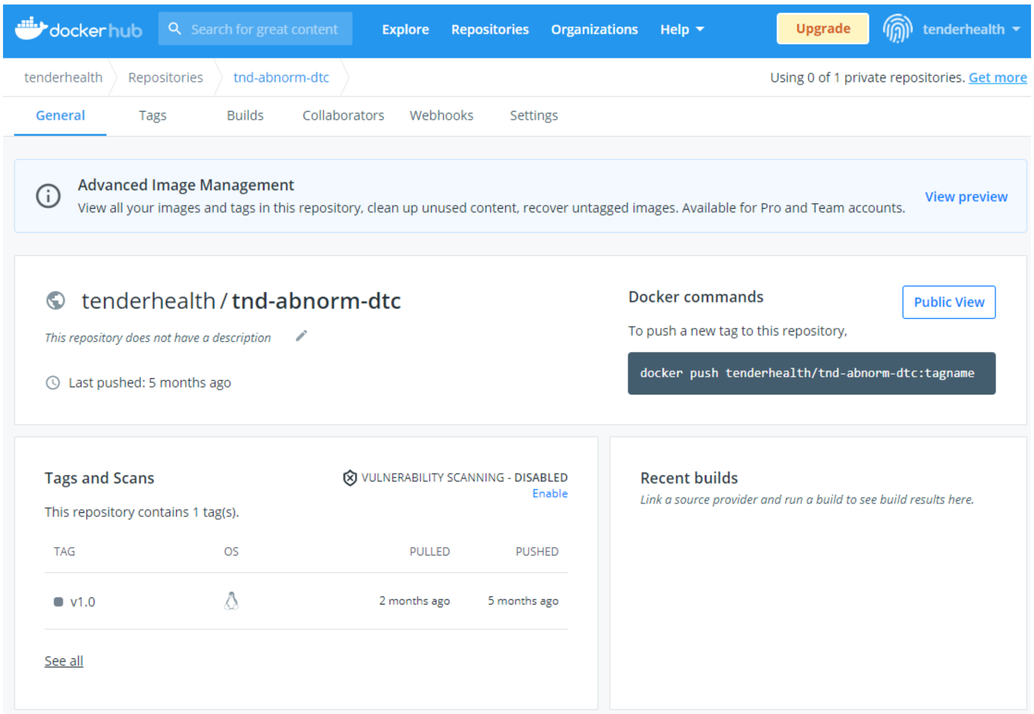

Figure 8 Docker Hub TeNDER repository

2.2 Development Environment

The Development environment is a workspace for developers to test anything they want without worrying about affecting any other users or developers working on a live deployment. In most cases, a development environment is set up on a local server or on the machine that developer uses, so the source code is ready to be executed and modified if needed. So, in TeNDER, developers use the tools and technologies of their choice (i.e. programming language, frameworks, IDE etc) and build the appropriated docker containers. Next, the service is executed in their local development environment.

2.3 Stage Environment

The Stage environment is hosted on the MAG cloud infrastructure, and it consist of VMs in which all the High-level Services (HLS) of TeNDER are deployed and tested based on the CI/CD framework provided from the GitLab repository. Based on specific CI/CD pipelines each component can be built, deployed and tested in the stage env using the Gitlab runners that are installed in the stage environment server. This approach provides to the developers an area where the entire TeNDER platform is deployed, in which any new version of a component can be deployed, tested, and validated before the deployment in the production. The stage environment is scalable, and it can be enhanced with more resources by adding new servers as the TeNDER platform grows. Table 2 shows the predefined flavor of the servers that are host the stage environment and Figure 9 presents the list of the running services in stage server.

Table 2 Stage server flavor

Server flavor	
vCPUs	4
RAM memory	8GB
IP address	185.146.161.245
Storage	120GB
#NICs	2
OS	Centos 7
Software	Gitlab runner Docker/Docker Compose

```
[tender@tender-stage ~]$ docker ps --format "{{.Names}}"
upm-fitbit-server_backend_1
upm-fitbit-server_mongo_1
worker_reports
worker_ckan
tnd-mongo-rest_mongo_api_1
tnd-mongo-rest_keycloak-gatekeeper_1
tnd-mongo-rest_nginx_1
tnd-mongo-rest_mongo_db_1
web_app
ckan
tender-consumer-ubw
netdata
tnd-dt-consumer
tnd-broker
hapi-fhir-jpaserver
auth-server_keycloak_1
auth-server_keycloak_db_1
auth-server_traefik_1
tnd-doc_swagger-ui_1
cool_bhaskara
cadvisor
upbeat_heisenberg
datapusher
hapi-fhir-postgres
db
redis
solr
```

Figure 9 TeNDER services running on stage env

2.4 Production Environment

The production environment is also hosted on the MAG cloud infrastructure, and it consists of virtual machines in which the stable version of the TeNDER platform is deployed and offered to the end users. The deployment is based on a CI/CD pipeline especially created of this process and the platform can be deployed either all at once or we can manually deploy each component separately. In any case, the update process can be done without affecting the user's data because all databases are mounted on virtual volumes which are not affected from the redeployment of the components. The CI/CD pipeline is kept in a separate repository in TeNDER's private Gitlab called tnd-production [5]. Following the same design approach of the stage, the production environment, this is also scalable and it can easily be scaled up by adding new servers to provide the appropriate resources to ensure the QoS level to the users. Table 3 shows the predefined flavor of the servers that host the production environment and Figure 10 presents the list of the running services in stage server.

Table 3 Production server flavor

Server flavor	
vCPUs	6
RAM memory	12GB
IP address	185.146.161.244
Storage	160GB
#NICs	2
OS	Centos 7
Software	Gitlab runner Docker/Docker Compose

```
[tender@tender-prod ~]$ docker ps --format "{{.Names}}"
web_app
mongo-rest_mongo_api_1
mongo-rest_nginx_1
mongo-rest_keycloak-gatekeeper_1
tender-consumer-ubw
netdata
device-api_backend_1
device-api_mongo_1
tnd-dt-consumer
tnd-broker
hapi-fhir-jpaserver
auth-server_keycloak_1
auth-server_traefik_1
hapi-fhir-postgres
cadvisor
auth-server_keycloak_db_1
ckan
mongo-rest_mongo_db_1
db
solr
redis
datapusher
```

Figure 10 TeNDER services running on production env

3 TESTING AND VALIDATION

This section discusses the tools that were evaluated to support the CI/CD workflow and to help the developers not only to monitor their deployments in stage and production environments but also to evaluate the functionality and the performance of the platform. These tools, include available open-source solutions and frameworks used for automation, validation and testing. During the first year of the project, we introduced a set of new tools that are mentioned in the following section. Most of the tools require expertise, and a learning curve to be digested and adopted by the team members. Nevertheless, as the development on new components and services is an ongoing procedure, we focused on practicing and evaluating the most known tools in order to be able to include them in the testing procedure as the TeNDER platforms expands.

3.1 Testing tools

As part of the followed methodology, several open-source tools and frameworks were considered. This section presents a brief overview of the frameworks and tools considered and the ones finally used.
The considered frameworks were:

- **Watir** [6] stands for "Web Application Testing in Ruby" and it is an open source Ruby library for automating tests. Watir interacts with a browser the same way people do clicking links, filling out forms and validating text.
- **Robot** [7] is a generic test automation framework for acceptance testing and Acceptance Dest-Driven Development (ATDD). It has easy-to-use tabular test data syntax and it utilizes the keyword-driven testing approach. Its testing capabilities can be extended by test libraries implemented either with Python or Java, and users can create new higher-level keywords from existing ones using the same syntax that is used for creating test cases.
- **pyTest** [8] is a python-based test framework for testing applications and python libraries. It is used from command line and requires tests to be formatted in a specific way so the framework can identify and execute them.
- **Shell** – UNIX [9] shell scripting may be used to create testing scripts that use the available Application Programming Interfaces (APIs) to make integration and validation tests.
- **Jmeter** [10] is a 100% pure Java and has an Ubuntu installer in order to be used by command line to perform the tests or via GUI. It may be used to test performance both on static and dynamic resources. It can be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyse overall performance under deferent load types.
- **Apache HTTP server benchmarking tool** [11] is a load testing and benchmarking tool for Hypertext Transfer Protocol (HTTP) server. It can be run from command line and it is very simple to use. A quick load testing output can be obtained in just one minute. As it does not need too much familiarity with load and performance testing concepts, it is suitable for beginners and intermediate users. To use this tool, no complex setup is required. Moreover, it can either be installed automatically with Apache web server, or it can be installed separately as Apache utility

The above list contains a small part of the available solutions for code test and automation. Furthermore, many programming languages and frameworks have developed their own testing libraries which in many cases are very flexible and easy to use. So, in TeNDER we let free the developers to decide which tool they want to use based on the technology that they choose and the testing requirements of their implementations.

3.2 Integration tests

The integration phase is composed by a set of tests with the main goal of testing the interaction between the different components of the TeNDER paltform. For this purpose, a series of tests was created starting with the deployment of all the containers in the stage server. At the end of the deployment phase the CI/CD pipeline triggers the integration tests that are hosted in different repository [12] through a specific API call to the TeNDER GitLab.
Each integration test is implemented as a bash script and is executed by the appropriate CI/CD pipeline. Currently the integration tests perform end-to-end testing between the following components:

- Message broker (RabbitMQ)
- Message consumers
- Remote Document Database
- Authorization and Authentication server
- TeNDER EHR (HAPI FHIR server)

There are three ways of executing the integration tests:

- Triggered by schedule: Every night at 3 - 4 am all tests are executed periodically (Figure 11).

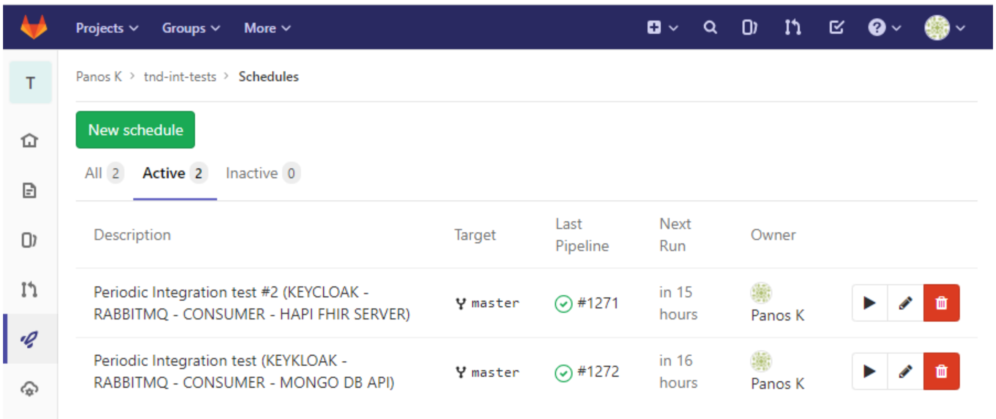

Figure 11 Periodic integration tests

- Triggered by other Pipeline: There is the possibility to trigger a pipeline from another one. So, after the build, deploy, test jobs of the deployment pipeline we trigger the specific integration test. (See subsection 2.1.1.4)

- Triggered manually: An integration test can be triggered manually from the web interface of the TeNDER GitLab.

At this point the end-to-end testing involves the all the active components of the TeNDER platform. However, as the development of new components and their integration with the TeNDER platform proceeds, these tests are going to enhanced with new ones in the future.
Currently, one of the typical integration tests which is executed each (sequentially) or after each day code push consiting of the following phases (Figure 12):

- **Phase 1:** Token Request from authorization and authentication server.

- **Phase 2:** Registration of new Users and Devices. At this point, we create new Users (ex Doctor and Patient) and we create new devices (ex sleep-tracker, smart-band, kinect etc) correlated with the specific patient for test purposes.
- **Phase 3:** Publish simulated data to the message broker (RabbitMQ).
- **Phase 4:** Retrieve data from Remote Document Database using the appropriated HTTP API.
- **Phase 5:** Retrieve data from TeNDER EHR (HAPI FHIR) using the appropriated HTTP API.
- **Phase 6:** Clean the environment.

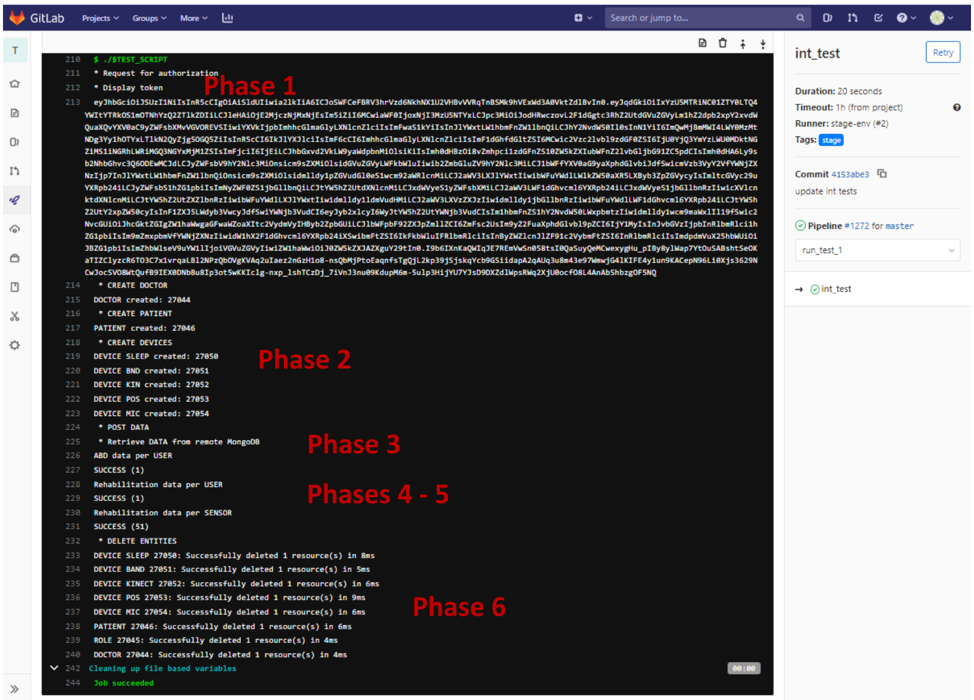

Figure 12 Integration test execution

3.3 Qualification test

The qualification phase aims to evaluate the performance of the platform. During this testing phase, we perform tests related to the functionalities, performance, security, and conformance with the requirements. Following the same approach with the integration tests, a specific repository was created to facilitate the qualification tests based on the CI/CD tools. Temporally, the stage environment is used, but as the development process of the platform proceeds the creation of a new environment dedicated to the qualification testing will be considered. This is necessary because many tests execute stress actions to measure the performance of the platform and identify potential "breakpoints" of the services etc. This kind of actions cannot be performed either in production or in the stage environments. At this point, we have developed qualification tests for the South Bound Interface (SBI) of the TeNDER platform consisting of an AMQP/SSL interface for collecting data from pilot site implementation. The qualification test uses the perf-test library provided from the RabbitMQ [13] which performs throughput tests on specific queues. This tool is capable to create a number of publishers/consumers, but in this case, we use only publishers as we want to measure also the performance of the TeNDER platform on consuming data. Note that this tool can achieve high rates for publishing (up to 80 to 90K messages per second and connection). Next, we present results from three different test scenarios.

Scenario 1

This test publishes 1000 messages in total with concurrency level of 100 messages. As we see from the output of the test we sent all the messages in 19 secs with average publishing rate of 99msg/sec. However, the TeNDER platform required ~20 secs to consume and process the incoming messages (Figure 13).

Command:

```
./stress-test-nbi.sh -q "sum-rehab" -r 100 -c 1000 -u 127.0.0.1 -m "re-hub.json"
```

Output:

```
queue: sum-rehab
rate: 100
tolal number of messages: 1000
uri: 127.0.0.1
msg file: re-hub.json
id: test-143715-071, starting producer #0
id: test-143715-071, starting producer #0, channel #0
id: test-143715-071, time: 1.005s, sent: 91 msg/s
id: test-143715-071, time: 2.005s, sent: 100 msg/s
id: test-143715-071, time: 3.005s, sent: 100 msg/s
id: test-143715-071, time: 4.006s, sent: 99 msg/s
id: test-143715-071, time: 5.016s, sent: 100 msg/s
id: test-143715-071, time: 6.025s, sent: 100 msg/s
id: test-143715-071, time: 7.025s, sent: 100 msg/s
id: test-143715-071, time: 8.025s, sent: 100 msg/s
id: test-143715-071, time: 9.025s, sent: 100 msg/s
id: test-143715-071, time: 10.026s, sent: 99 msg/s
id: test-143715-071, sending rate avg: 99 msg/s
id: test-143715-071, receiving rate avg: 0 msg/s
Messages publishing took 19 secs
```

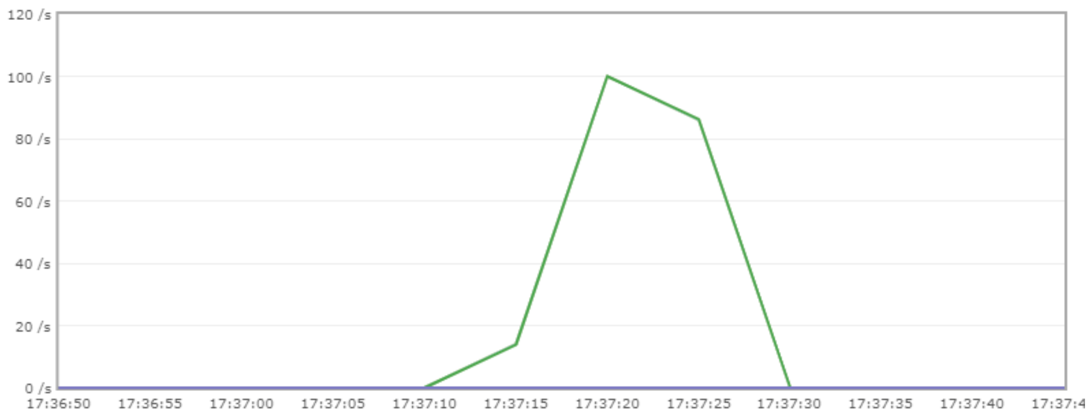

Figure 13 TeNDER platform consuming rate (Scenario 1)

Scenario 2

In the second scenario, 10000 messages in total are published with concurrency level of 1000 messages. As we see from the output of the test the test tool managed to send all the messages in 17 secs with average publishing rate of 989msg/sec. However, in this case the

TeNDER platform required ~40 secs to consume and process the incoming messages (Figure 14).

Command:

```
./stress-test-nbi.sh -q "sum-rehab" -r 1000 -c 10000 -u 127.0.0.1 -m "re-
hub.json"
```

Output:

```
queue: sum-rehab
rate: 1000
tolal number of messages: 10000
uri: 127.0.0.1
msg file: re-hub.json
id: test-145004-901, starting producer #0
id: test-145004-901, starting producer #0, channel #0
id: test-145004-901, time: 1.000s, sent: 901 msg/s
id: test-145004-901, time: 2.000s, sent: 1000 msg/s
id: test-145004-901, time: 3.001s, sent: 999 msg/s
id: test-145004-901, time: 4.001s, sent: 1001 msg/s
id: test-145004-901, time: 5.001s, sent: 1000 msg/s
id: test-145004-901, time: 6.001s, sent: 1000 msg/s
id: test-145004-901, time: 7.001s, sent: 1000 msg/s
id: test-145004-901, time: 8.001s, sent: 1000 msg/s
id: test-145004-901, time: 9.001s, sent: 1000 msg/s
id: test-145004-901, time: 10.001s, sent: 1000 msg/s
id: test-145004-901, sending rate avg: 989 msg/s
id: test-145004-901, receiving rate avg: 0 msg/s
Messages publishing took 17 secs
```

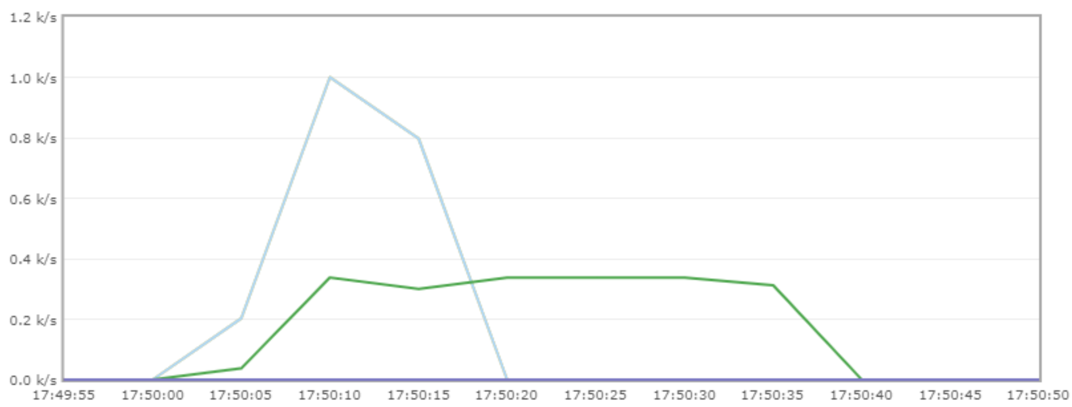

Figure 14 TeNDER platform consuming rate (Scenario 2)

Scenario 3

In the third scenario, we stressed further the platform increasing the concurrency level to 3000 messages. In this case the test tool managed to send all the messages in 10 secs with average publishing rate of 2901 msg/sec. However, the TeNDER platform required ~40 secs to consume and process the incoming messages (Figure 15).

Command:

```
./stress-test-nbi.sh -q "sum-rehab" -r 3000 -c 10000 -u 127.0.0.1 -m "re-
hub.json"
```

Output:

```
queue: sum-rehab
rate: 3000
tolal number of messages: 10000
uri: 127.0.0.1
msg file: re-hub.json
id: test-150258-827, starting producer #0
id: test-150258-827, starting producer #0, channel #0
id: test-150258-827, time: 1.000s, sent: 2695 msg/s
id: test-150258-827, time: 2.001s, sent: 2997 msg/s
id: test-150258-827, time: 3.001s, sent: 3003 msg/s
id: test-150258-827, sending rate avg: 2901 msg/s
id: test-150258-827, receiving rate avg: 0 msg/s
Messages publishing took 10 secs
```

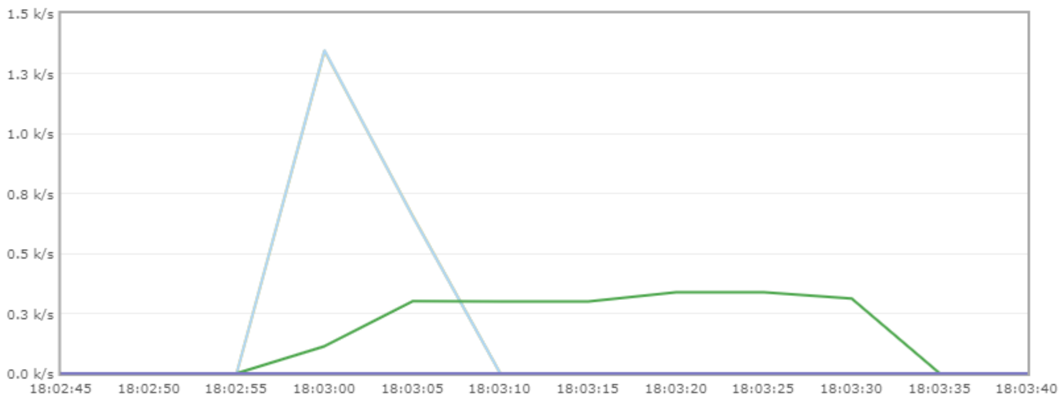

Figure 15 TeNDER platform consuming rate (Scenario 3)

The results from the qualification test provided some very useful conclusions regarding the performance of the first version of the TeNDER platform. First, the adoption of a publish/subscribe message broker as SBI of the TeNDER platform provided high incoming throughput more than 3000 msg/sec which overcomes the current requirements from the first wave of pilots in TeNDER. However, the platform consumes and processes the incoming messages in average 340 mgs/sec (green line of Figure 15). This performance is acceptable for the first wave of pilots but it could be improved in the future versions.

More qualification tests regarding the North Bound Interface (NBI) are under development and they will be presented in the D5.5.

4 MONITORING RESOURCES

TeNDER platform implements an open, service-oriented architecture which aims to cover all the operational aspects from actual realization, test, trials and support the pilots in the WP6. To achieve this goal, it is necessary to provide the appropriate tools in order to guarantee (a) the integration of the services developed in WP3 and WP4 and (b) the appropriate resources allocation for service deployment in the deferent environment (i.e. stage, production etc). Therefore, in TeNDER we designed and deployed a state-of-the-art monitoring and analysis framework based on open-source tools for collecting performance metrics from every deployment site. This monitoring system is installed in a separate VM (Table 4) running on MAG's cloud infrastructure and collects data from the HLS services and the LLS services running on pilot sites. Additionally, to guarantee the resource allocation TeNDER's monitoring system collects information related to the available resources of the servers in stage and production environments.
Under this perspective, it is of paramount importance to collect monitoring data from as many possible sources. In the implemented system, there are four different types of sources for collecting data:

1) Containers (i.e. services running as docker containers)
2) VMs (i.e. service running on VMs or VMs hosting stage/production environments)
3) Physical servers (i.e. physical machines hosting TeNDER services)
4) Network traffic (i.e. network traffic on physical and virtual level)

Apart from the collection and the process of monitoring data related to the performance of the TeNDER's services and infrastructure, the monitoring framework will accommodate specific alerting rules for real-time notification events. In this respect, the monitoring framework will offer the capability to developers to define service-specific metrics and rules, whose violation will generate alerts.

Table 4 Monitoring server flavor

Server flavor	
vCPUs	4
RAM memory	8GB
IP address	185.146.161.250
Storage	120GB
#NICs	2
OS	Centos 7
Software	Docker/Docker Compose

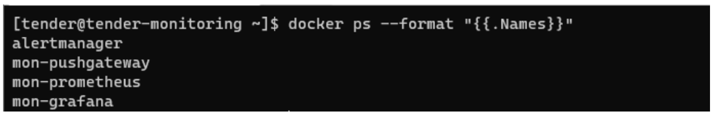

Figure 16 Monitoring tools

Monitoring system architecture

TeNDER's monitoring solution complies with the scalability requirement of the services-oriented architecture of the TeNDER platform because the selected tools are Cloud Native (CN) implementations, and the proposed design can easily integrate new types of monitoring targets without the need for difficult configurations or down-time. So, in case that we need to scale up the production environment by adding a new server, the only necessary action is to update the configuration file of the Prometheus monitoring server[4]. Moreover, for large scale deployments Prometheus Monitoring servers supports a distributed (cascaded) architecture. The local Prometheus servers collect and store metric data from the services deployed in the LLS/HLS, while only the alerts are sent to the federated Prometheus server for further processing and forwarding to the appropriate users. Another scalability requirement concerns the large flow of data from the monitoring agents to the monitoring server and its respective database that might affect the service performance in extreme cases. To overcome these potential problems the monitoring system (a) is configured to store monitoring data of a specific period and (b) in cases of large deployment is able adopt the cascade architecture mentioned above. At the current development status of the TeNDER platform the monitoring system can be accommodated a by a single server deployment. The detail architecture is shown in the Figure 17.

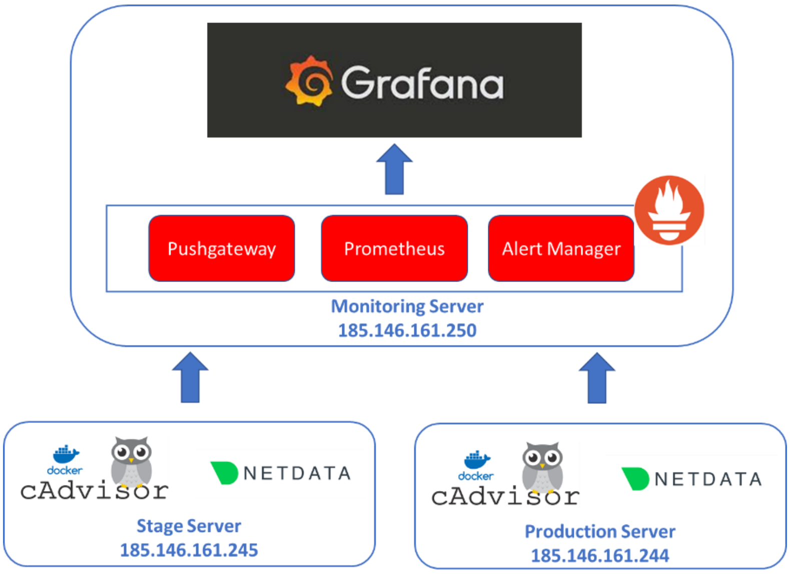

Figure 17 Monitoring framework architecture

The architecture of the monitoring system consists of the following components:

Monitoring tools:

- Prometheus server [14] stands as the central point of event monitoring, storage and alerting. All performance metrics are collected, using a HTTP pull model, and stored in a timeseries database. Some of the key features that make this server suitable for the proposed architecture are: (a) use of a flexible query language (PromQL), which makes easier the interconnection with external systems (b) existence of many opensource implementations (exporters) for exposing monitoring metrics from various applications, to create new ones (c) autonomy as there is no reliance on complex distributed storage mechanisms and (d) new monitoring targets can be

[4] Prometheus Server Reference: Available at https://prometheus.io/docs/introduction/overview/

easily added via reconfiguration or by using the file-based service discovery mechanisms.

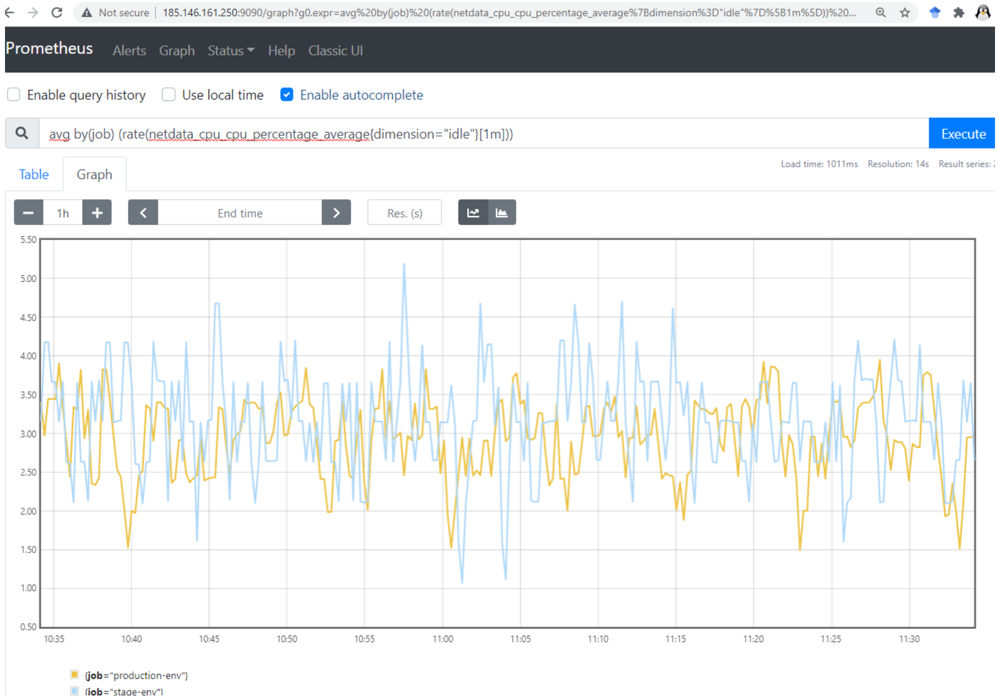

Figure 18 Prometheus chart

- The Prometheus Pushgateway [15] allows batch jobs, running on LLS in pilot sites, to expose their metrics to Prometheus. Since this kind of jobs may not exist long enough to be scraped, they can instead push their metrics to a Pushgateway. The Pushgateway then exposes these metrics to Prometheus server (Figure 19).

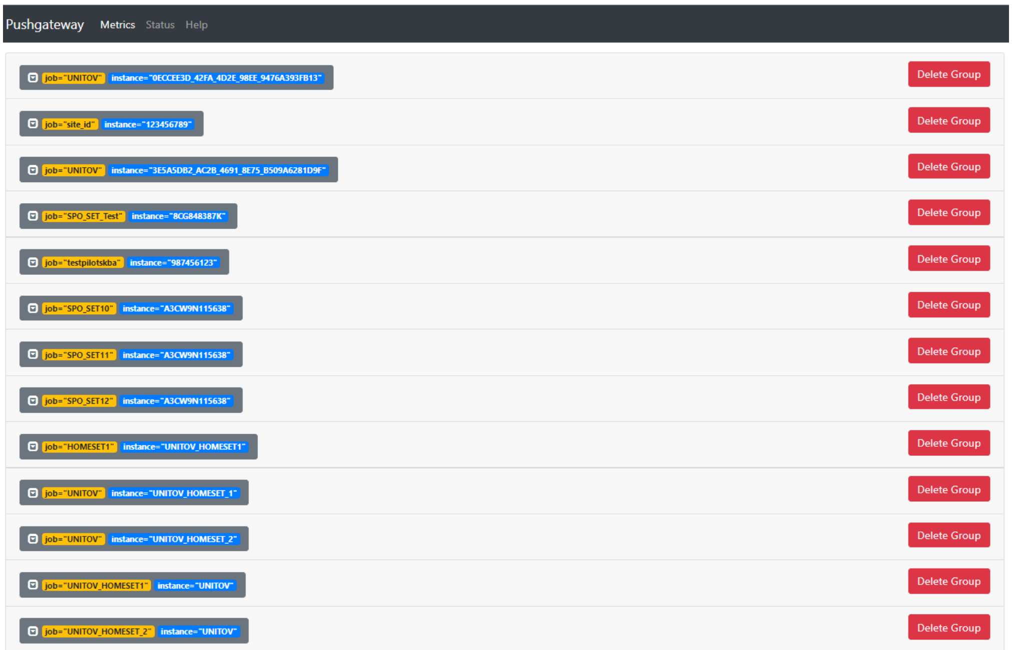

Figure 19 Performance metrics from LLS

- Alertmanager [16] handles alerts sent by client applications such as Prometheus server. It takes care of deduplicating, grouping, and routing them to the correct receiver integrations such as email, PagerDuty, or OpsGenie. It also takes care of silencing and inhibition of alerts.

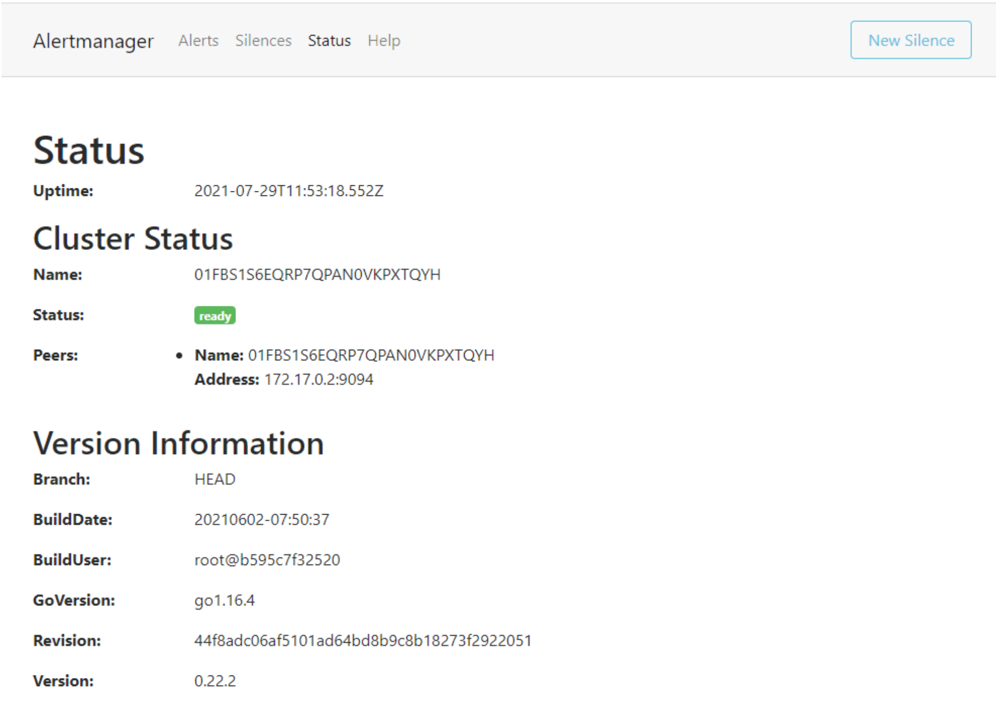

Figure 20 Prometheus Alertmanager

- Grafana [17] is an open-source solution for running data analytics, pulling up metrics that make sense of the massive amount of data and it provides interactive visualization web dashboards (Figure 21).

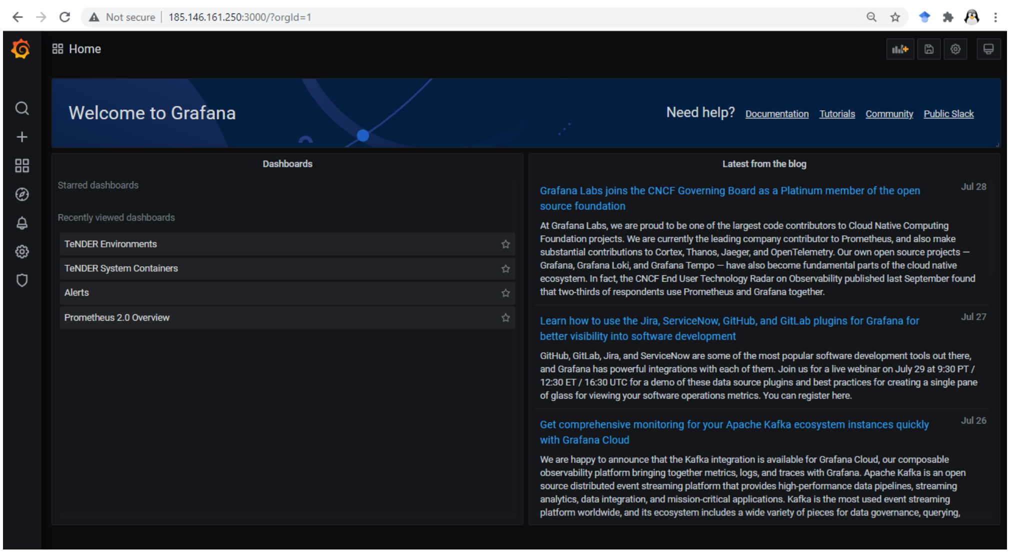

Figure 21 TeNDER dashboards on Grafana

Monitoring Agents:

- Netdata.io [18] is a powerful real-time monitoring agent which collects thousands of metrics from systems, hardware, virtual machines, and applications with zero configuration. It runs permanently on the physical/virtual servers, containers, cloud deployments, and edge/IoT devices, and is perfectly safe to install on your systems mid-incident without any preparation (Figure 22).

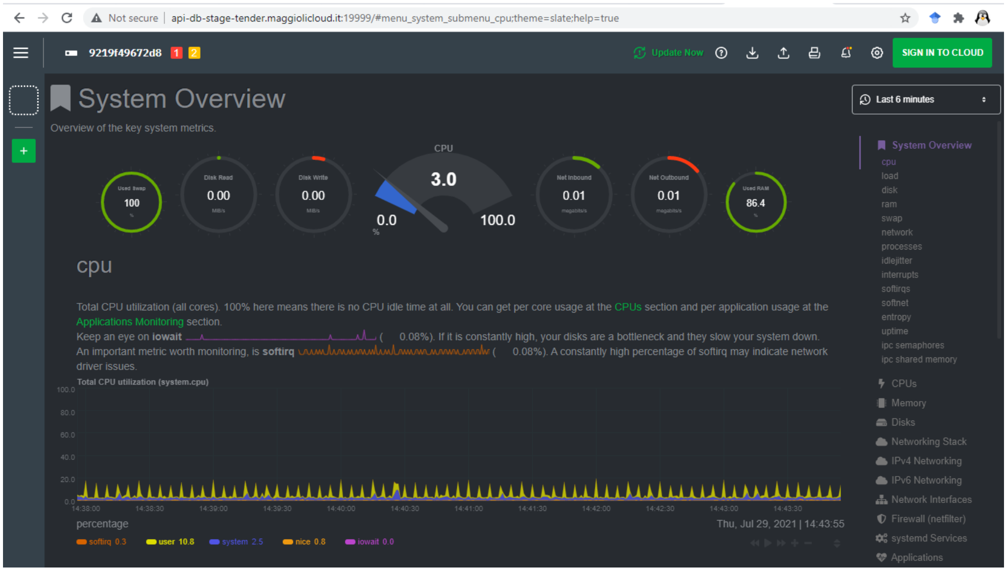

Figure 22 Netdata web GUI

- cAdvisor [19] provides metrics of the resource usage and performance characteristics of the running containers. It is a running daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics (Figure 23).

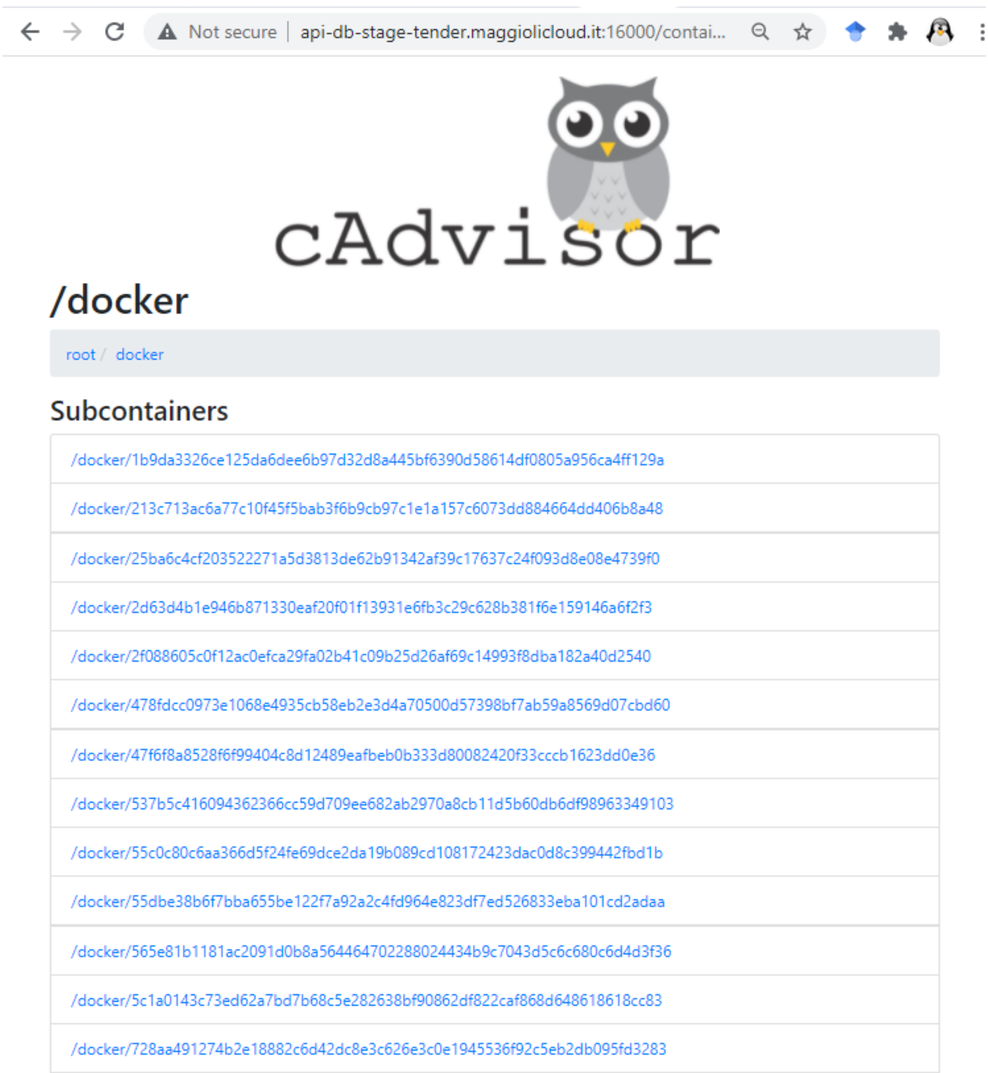

Figure 23 List of running containers in cAdvisor GUI

4.1 Stage and Production environments

The monitoring information from the stage and the production environments is collected by the Netdata monitoring agent, then the Prometheus server scrapes the arriving messages periodically and stores the information collected to its local time-series database. Next, a Grafana dashboard has been configured which uses Prometheus server as data source and visualizes the data via interactive charts. In this way the administrator of the infrastructure can select the environment of his choice and see in one dashboard all the critical performance metrics and the current resource utilization (Figure 24 and Figure 25).

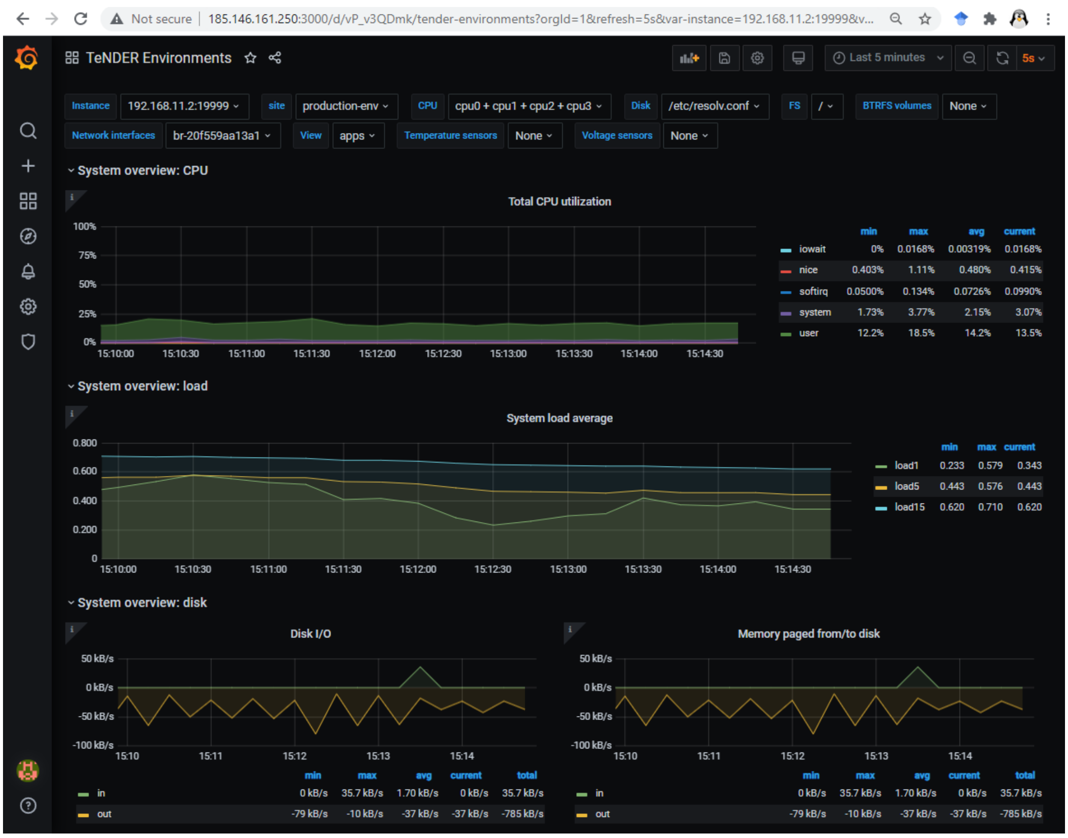

Figure 24 Recourse allocation in the production environment

Figure 25 Recourse allocation in the stage environment

4.2 TeNDER HLS services

In TeNDER platform all the services of the HLS are hosted on docker containers. Therefore, it is crucial not only to monitor the resource allocation on stage or production level but also to investigate how the available resources are distributed to each one of the running containers. For this reason, we adopted the cAdvisor monitoring agent which provides detailed information about the status of the running containers. Following the same approach, Prometheus pulls periodically data from cAdvisor agents running on stage and production environments, and stores them in its local timeseries database. Next, a specific interactive dashboard has been developed to visualize the resource utilization per container (Figure 26).

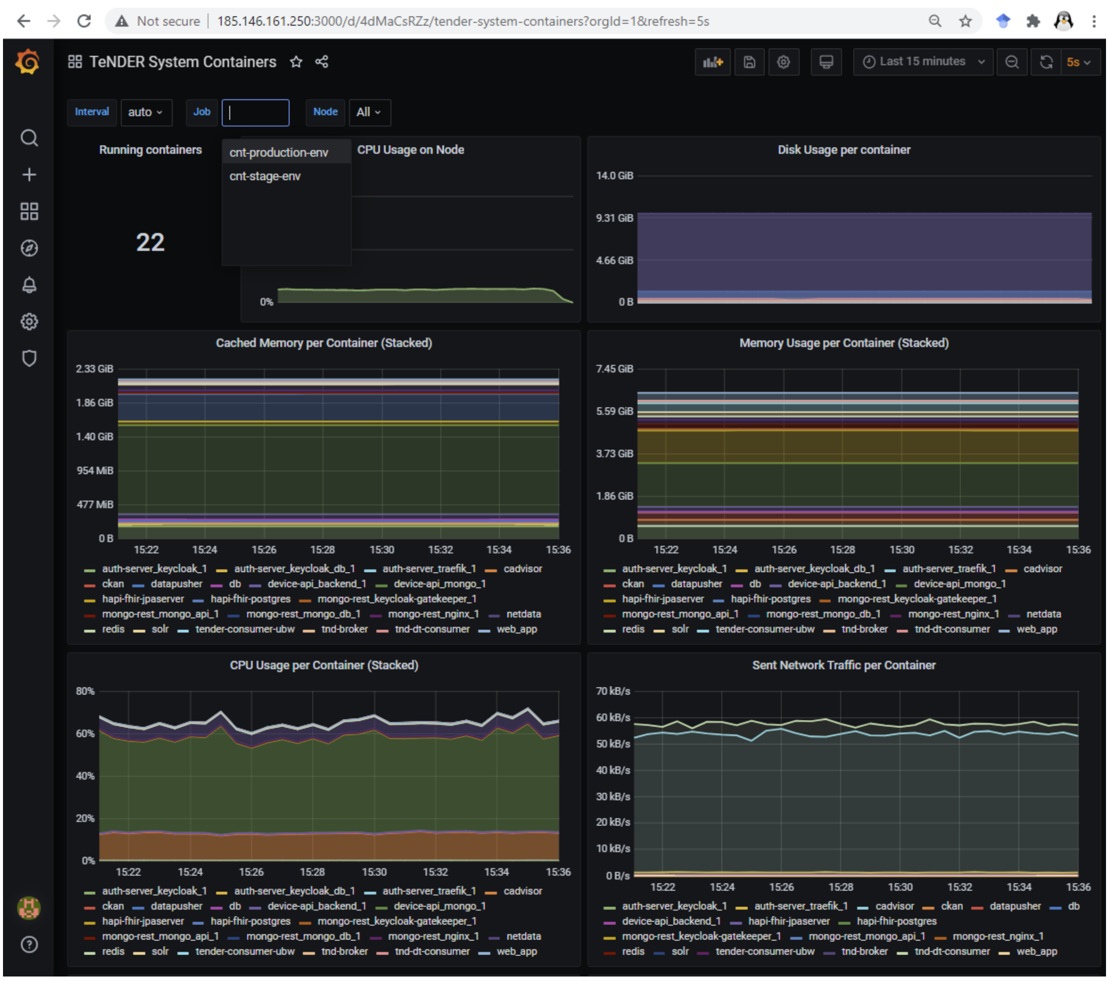

Figure 26 Recourse allocation per running container

4.3 TeNDER LLS services

TeNDER LLS is mainly composed of several components for gathering and processing information from heterogeneous sensing devices (i.e. Depth Sensors, wearables, sleep trackers, position trackers etc.). These components implement several functionalities including data storage, processing, synchronization, anonymization as well as event detection and activity recognition. Next, the events are sent to HLS to support useful functionalities for the TeNDER stakeholders. The first version of the TeNDER platform consists of the following components:

- HeTRA Server

- HeTRA Client
- Abnormal detection Module
- Local document database (MongoDB)

The LLS components are installed in physical machines on each pilot site (i.e. homes, rehabilitation rooms, hospitals, etc.). In some cases, the components are executed for a specific period under the supervision of a health professional (ex. during a set of exercises in a rehabilitation room), but in some other cases the LLS run in unattended mode as daemon services. For example, in the home installations, a hard requirement is to monitor the performance status of the LLS services in an automated manner and generate the appropriate notification in case of service or network failure. For this reason, during the LLS installation process, a health-check bash script is installed which runs periodically and checks the operational status of the LLS components and reports the status to the Prometheus Pushgateway server (Figure 27). The report data do not contain any personal information (i.e. user name, IP address etc) but only the installation id and the current status of each component.

Command:

```
Powershell.exe
-executionpolicy bypass
-File c:\tender\tnd-install\pushmetric.ps1
-Job %siteID% -Instance %copmID%
-Metric %METRIC%
-Value 0
```

Report Script (pushmetric.ps1):

```
param(
     [parameter(Mandatory=$false)]
     [string]$Job,
     [string]$Instance,
     [string]$Metric,
     [string]$Value
)
$mt = "$Metric $Value`n "Invoke-WebRequest
-Uri http://185.146.161.250:32657/metrics/job/$Job/instance/$Instance
-UseBasicParsing
-Method POST -Body $mt > $null
```

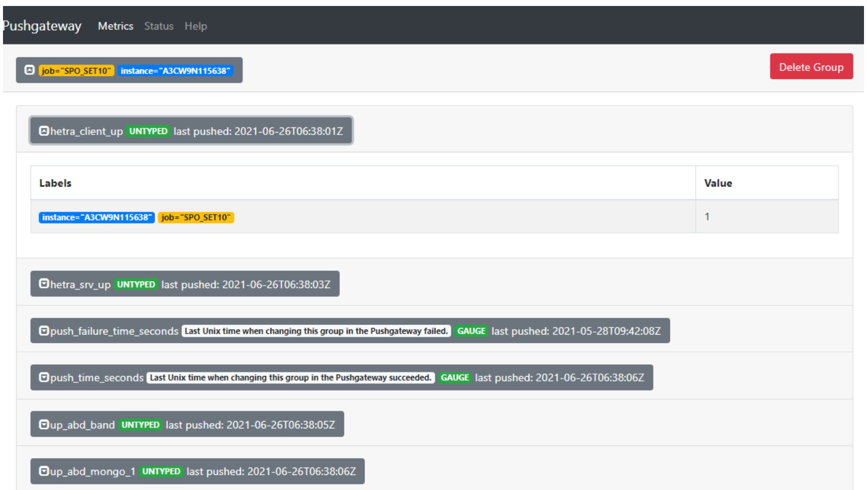

Figure 27 Metrics from SPOMINCICA installation site

4.4 Alerting

One of the most useful features of the TeNDER monitoring system is the alerting mechanism, which offers near real-time notifications to the service developers and the administrators of the infrastructure. Alerting in the Prometheus context is separated into two parts. The first one has to do with the definition of the alerting rules in the Prometheus server (Figure 28), and the second one is the actual management of the alert events which it takes place in the Alertmanager. The Alertmanager receives the alert events from one or many Prometheus servers and then performs management actions including silencing, inhibition, aggregation and sending out notifications via methods such as email, on-call notification systems, and chat platforms as shown in figures 28-35.

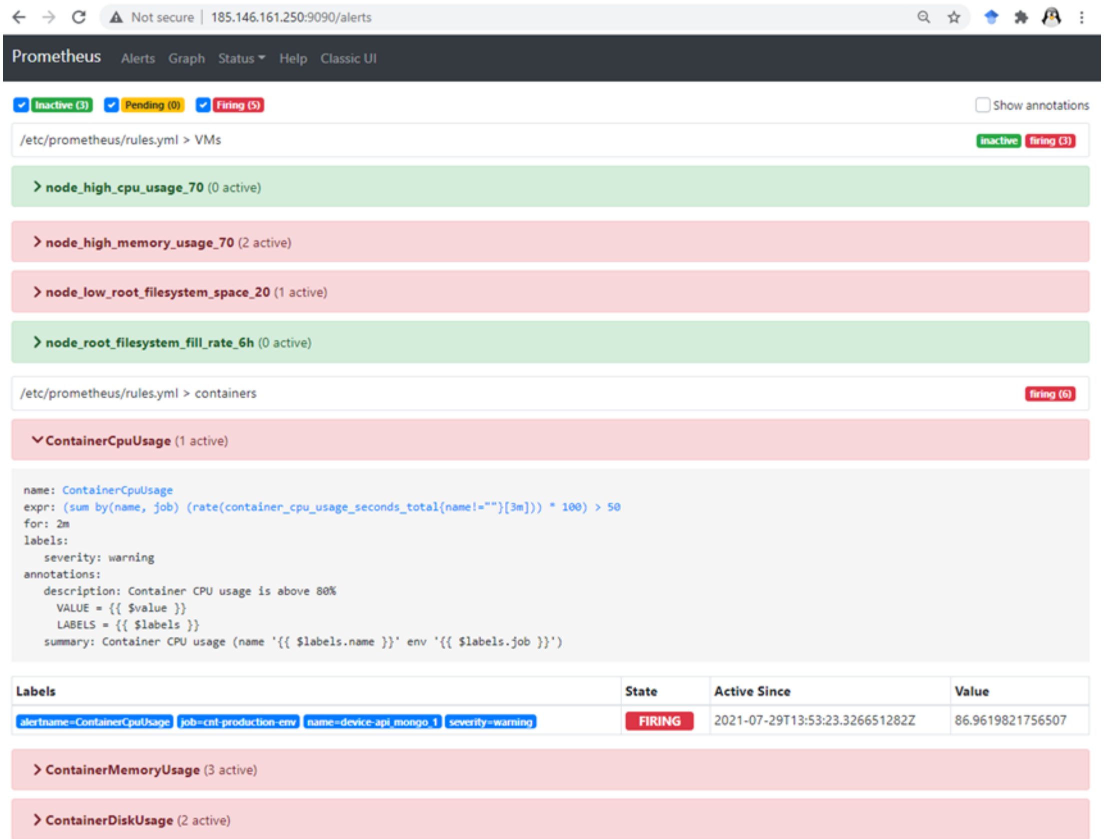

Figure 28 Rules status in Prometheus server

The main steps to setting up alerting and notifications are:

- Setup and configure the Alertmanager
- Configure Prometheus to talk to the Alertmanager
- Create alerting rules in Prometheus

Currently, the alerting rules have been organized in two groups, one related to the status of the servers and another on the running containers.

Some of the already applied rules are the following:

Servers:

1. **CPU utilazation**
   Description: Server CPU utilazation over 70% for more than 1 minute.

```
alert:  node_high_cpu_usage_70
expr:  avg by(job) (rate(netdata_cpu_cpu_percentage_average{dimension="idle"}[1m])) > 70
for: 1m
annotations:
  description: {{ $labels.job }} on '{{ $labels.job }}' CPU usage is at {{ humanize $value
  }}%.
  summary: CPU alert for container node '{{ $labels.job }}'
```

Figure 29 Server CPU utilization rule

2. **Memory utilazation**
   Description: Server memory utilazation over 70% for more than 1 minute.

```
alert:  node_high_memory_usage_70
expr:  100 / sum by(job) (netdata_system_ram_MiB_average) * sum by(job)
(netdata_system_ram_MiB_average{dimension=~"free|cached"}) < 30
for: 1m
annotations:
  description: {{ $labels.job }} memory usage is {{ humanize $value}}%.
  summary: Memory alert for container node '{{ $labels.job }}'
```

Figure 30 Server memory utilization rule

3. **Storage utilazation**
   Description: Server storage utilazation over 80% .

```
alert:  node_low_root_filesystem_space_20
expr:  100 / sum by(job) (netdata_disk_space_GiB_average{family="/"}) * sum by(job)
(netdata_disk_space_GiB_average{dimension=~"avail|cached",family="/"}) < 20
for: 1m
annotations:
  description: {{ $labels.job }} root filesystem space is {{ humanize $value}}%.
  summary: Root filesystem alert for container node '{{ $labels.job }}'
```

Figure 31 Server storage utilization 80%

Description: Filesystem is predicted to run out of space within the next 6 hours at current write rate.

```
alert:  node_root_filesystem_fill_rate_6h
expr:  predict_linear(netdata_disk_space_GiB_average{dimension=~"avail|cached",family="/"}
[1h], 6 * 3600) < 0
for: 1h
labels:
  severity: critical
annotations:
  description: Container node {{ $labels.job }} root filesystem is going to fill up in 6h.
  summary: Disk fill alert for Swarm node '{{ $labels.job }}'
```

Figure 32 Filesystem run-out prediction rule

Containers:

1. **CPU utilazation**
   Description: Container CPU usage over 50% for more than 2 minutes.

```
alert:   ContainerCpuUsage
expr:   (sum by(name, job) (rate(container_cpu_usage_seconds_total{name!=""}[3m])) * 100)
> 50
for: 2m
labels:
  severity: warning
annotations:
  description: Container CPU usage is above 80%
    VALUE = {{ $value }}
    LABELS = {{ $labels }}
  summary: Container CPU usage (name '{{ $labels.name }}' env '{{ $labels.job }}')
```

Figure 33 Container CPU usage rule

2. **Memory utilazation**
   Description: Container memory usage over 1.5GB for more than 2 minutes.

```
alert:   ContainerMemoryUsage
expr:   sum by(name, job) (container_memory_rss{name!=""}) > 1.5e+09
for: 2m
labels:
  severity: warning
annotations:
  description: Container Memory usage is above 1.5GB
    VALUE = {{ $value }}
    LABELS = {{ $labels }}
  summary: Container Memory usage (name '{{ $labels.name }}' env '{{ $labels.job }}')
```

Figure 34 Container memory usage rule

3. **Storage utilazation**
   Description: Container disk usage over 2GB for more than 2 minutes.

```
alert:   ContainerDiskUsage
expr:   container_fs_usage_bytes{name!=""} > 2e+09
for: 2m
labels:
  severity: warning
annotations:
  description: Container Disk usage is above 2GB
    VALUE = {{ $value }}
    LABELS = {{ $labels }}
  summary: Container Disk usage (name '{{ $labels.name }}' env '{{ $labels.job }}')
```

Figure 35 Container disk usage rule

5 TeNDER PLATFORM

5.1 Low Level Subsystem

The low-level subsystem (LLS) is mainly composed of several sensing modules that gather information from the patients. Several subsystems are interoperating at diverse locations (in each of the TeNDER countries) to collect information from several devices (including depth sensors, wearables, sleep trackers among others). These modules are divided into several categories that safely store the information, as described in Figure 36, where the data are processed, synchronized, and a set of separated solutions to transform the data collected into the useful functionalities for the TeNDER stakeholders. This module, formally known as Activity Recognition, is responsible of orchestrating the events detector (i.e. fall down, festination, etc.).

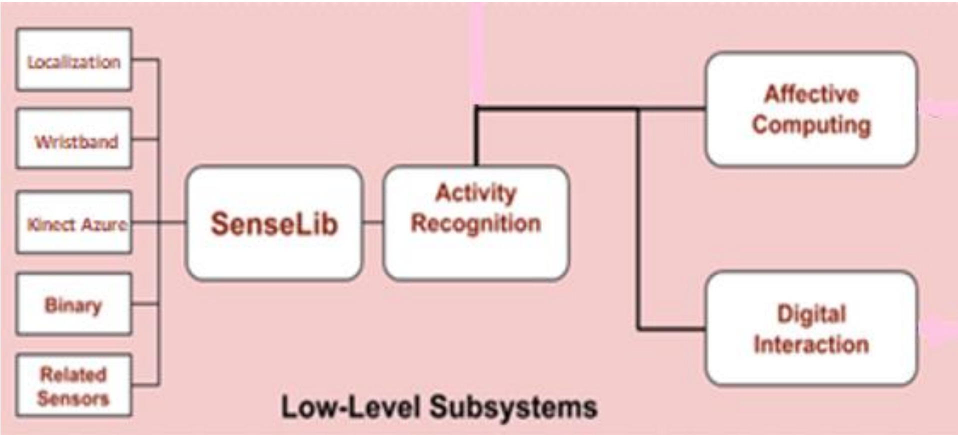

Figure 36 TeNDER Low Level Subsystem

HeTra subsystem is the core subsystem of the LLS. It enables tracking patient and offers to the high-level subsystem's modules the functionality to track specific health characteristics, from direct health situation information to periodical test results and feedback from professionals. Moreover, this subsystem gives the opportunity to the users to choose which health characteristics to track and, also, provides an efficient feedback mechanism that, along with user activity recognition and, through multimodal fusion, allows for the extraction of valuable conclusions regarding the patient's health status.

HeTra is responsible for the data acquisition from the sensors as well as HeTra delivers the acquired data to the Abnormal Behaviour Detector (ABD) subsystem that is part of TeNDER LLS and to the Multimodal Fusion (MMF) subsystem which is part of the HLS.

HeTra does not only deliver the raw data as acquired from the sensors but it also provides techniques in order to extract features that will be useful for subsequent analysis. This analysis is performed in SENSELib. This library includes sensor data acquisition tools as well as specific algorithms for data processing (tracking, skeleton smoothing, dimensionality reduction etc.).

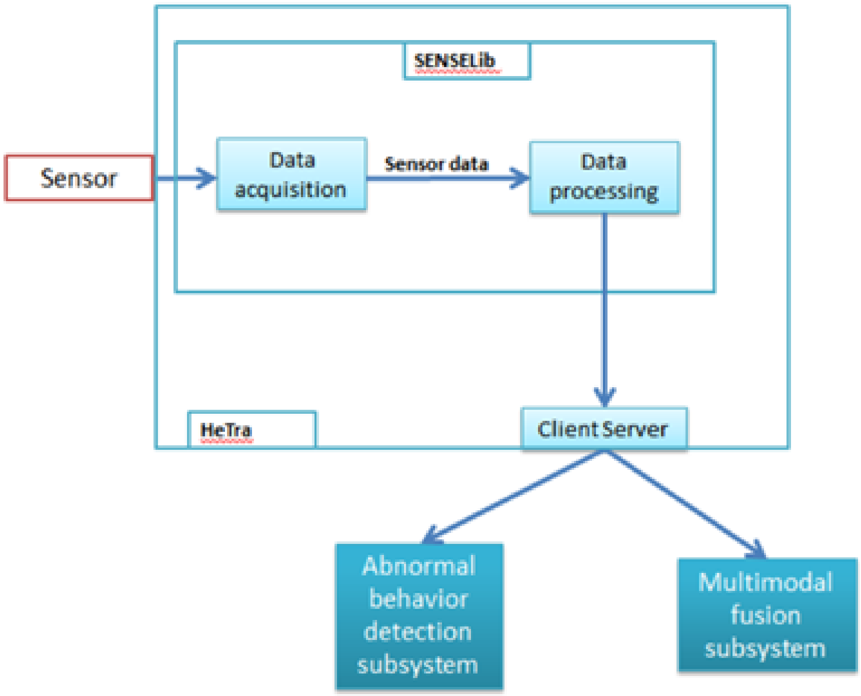

Figure 37 SenseLib schematic description

A client of HeTra runs on the other subsystems of TeNDER (ABD and MMF subsystems) through which the communication with HeTra will take place.

SENSELib is a part of the TeNDER's open API system and is used to develop HeTra subsystem. This library provides mainly two types of functionalities, i.e., acquisition and processing (Figure 37) based on the following modules:

- Multi-Sensorial Capturing module
- Digital Interaction Module
- Abnormal Behaviour Detection module
- Affective Computing module
- Localization tracking module
- Kinect Azure tracking module

5.1.1 HeTRA server and client (CERTH)

Description

From a front-end perspective, the HeTra tool is comprised of two separate applications the HeTra client and the HeTra server, each of them having its own GUI. Using these GUIs, the user may check the connectivity with the sensors, select the type of data to be acquired (e.g. in the case of the Kinect v02 and the Azure Kinect sensor RGB, Depth and IR frames could be captured). In addition, HeTra enables the acquired raw data from the sensors to be stored locally in a Mongo database instance. For example, by clicking on the button "**Connect and**

check Devices" HeTra server looks for a response from the selected devices. Then, once response is taken, the user can click on the "**Begin Acquisition**" button to start acquiring data from the sensors. Additionally, the user may click on the "**Save to DB**" button to save sensors' raw data to the MongoDB. Furthermore, in the cases in which data acquisition is deployed via secure API calls (Localization, Sleep Sensor, Wristband sensors), using the HeTra Server GUI, the user may fill in the specific URLs which contain the sensor IDs from which he/she needs to acquire data from.

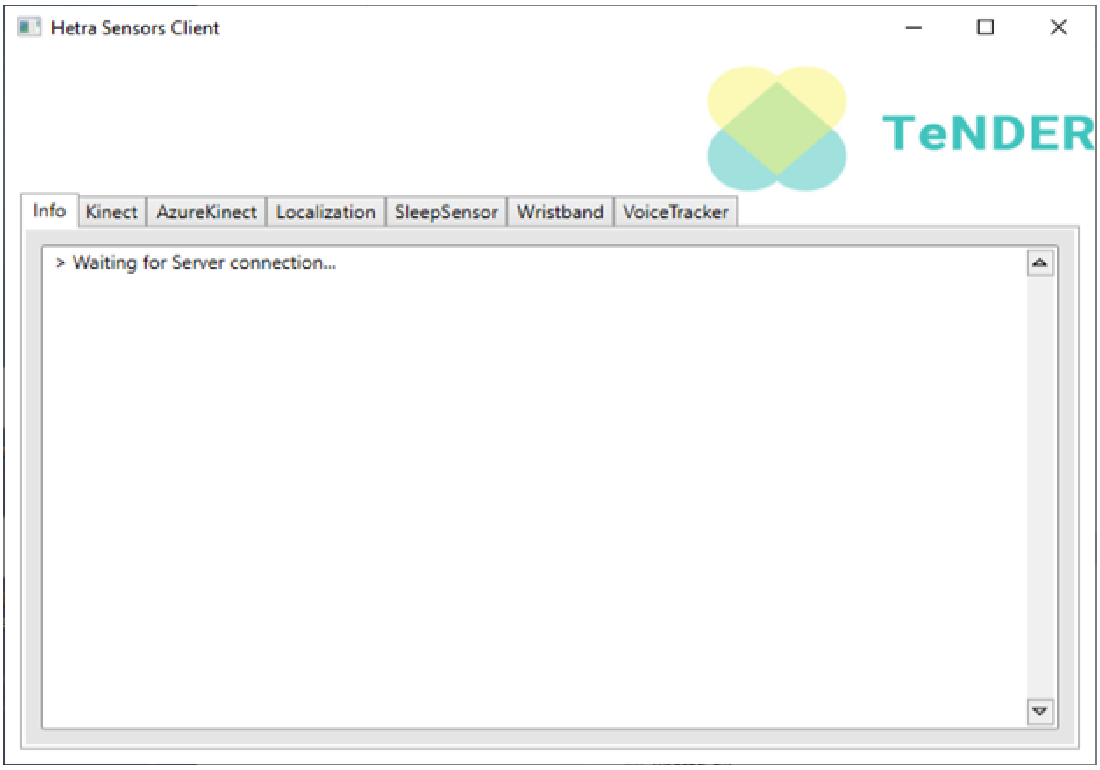

Figure 38 HeTra Client GUI.

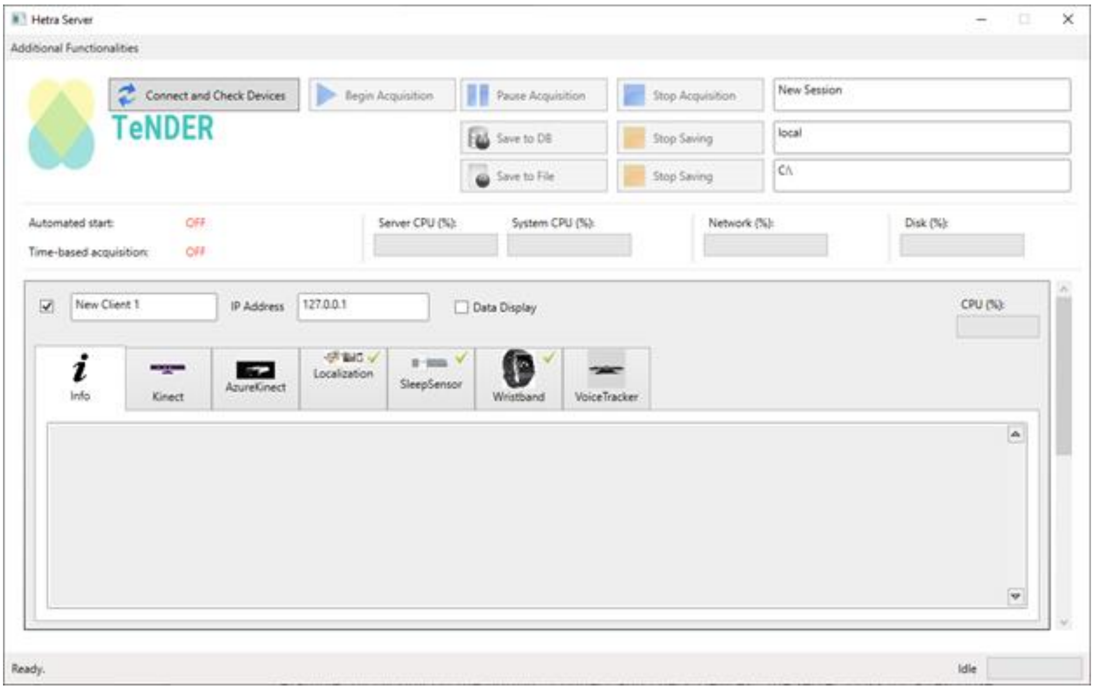

Figure 39 HeTra Server GUI.

From a back-end perspective, the HeTra tool provides the ability to collect data from cameras (Kinectv02, Azure Kinect), collect raw data from sensors (Localization, Sleep Sensor, Wristband) via secure API calls and additionally gather data from microphones (Voice Tracker). All these collections can be orchestrated and synchronized through HeTra and may be further exploited from the other modules of the TeNDER ecosystem maintaining the privacy of the users.

Software Dependencies

- Windows 10 (64 bit)
  Pro, Enterprise, Education (Build 17134 or higher), Home (version 1903 or higher)
- Python 3.7.3
- Kinect Runtime 2.0
- PyAudio-0.2.11

Build – Deployment

```
msbuild HeTraClient.sln
msbuild HeTraServer.sln
```

Tests

Table 5 Senselib test

Test name	Senselib
Test Purpose	Check the methods responsible for acquiring data from different sensors as well as methods for processing the acquired data.
Pre-test conditions	Run the test in the solution in Visual Studio 2019

Test Tool	VSTest.Console.exe (Visual Studio 2019)
Test description	Check Senselib project.
Test Verdict	The library of Senselib is functional

Command:

```
vstest.console.exe Test_Sencelib.dll
```

Table 6 Client test

Test name	**HeTraClient**
Test Purpose	Check the visualization and communication between of the sensors data with the main server.
Pre-test conditions	Run the test in the solution in Visual Studio 2019
Test Tool	VSTest.Console.exe (Visual Studio 2019)
Test description	Check HeTra_Client project.
Test Verdict	HetraClient.exe is functional

Command:

```
vstest.console.exe Test_Client.dll
```

Table 7 Server test

Test name	**HeTra Server**
Test Purpose	Check the collection of the data and the performing tasks of consultation to other instances/apps to ingest the data into the TeNDER local storage.
Pre-test conditions	Run the test in the solution in Visual Studio 2019
Test Tool	VSTest.Console.exe (Visual Studio 2019)
Test description	Check HeTra_Server project.
Test Verdict	HetraServer.exe is functional

Command:

```
vstest.console.exe Test_Server.dll
```

We can conduct all the tests from the Visual Studio IDE during the development process. In the following figures, the execution and the test results are depicted.

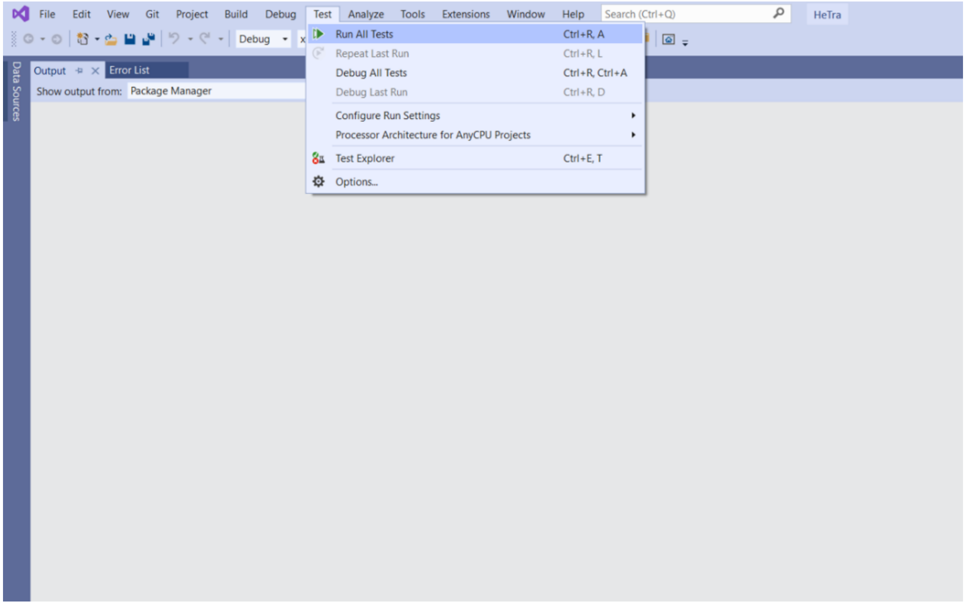

Figure 40 Unit Tests execution

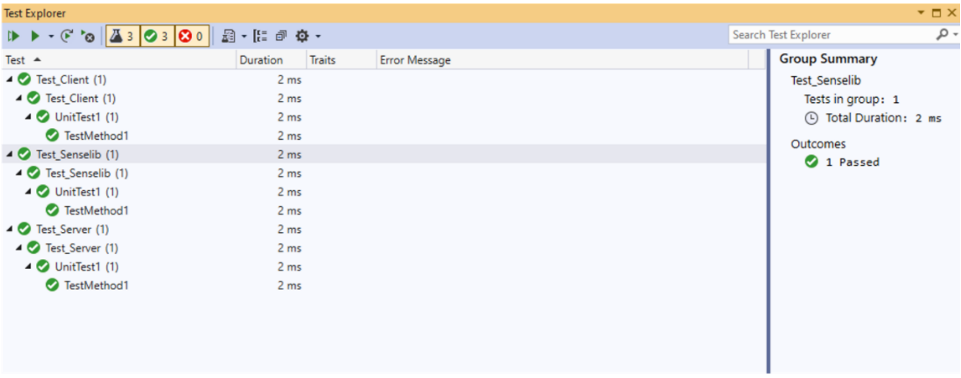

Figure 41 Unit Test execution results

5.1.2 Abnormal detection Module (UPM)

Description

This module comprises all the functionalities of interest for the patients, family, caregivers and health professionals for the care delivery. The module is mainly composed of two types of functionalities: The “real time events” and the “non-real time events”. The former group contains those events that require immediate attention including high Heart Rate, the fall down, patient leaving the house among other events. The latter group contains the non-real time events that will be reported in periodic messages.

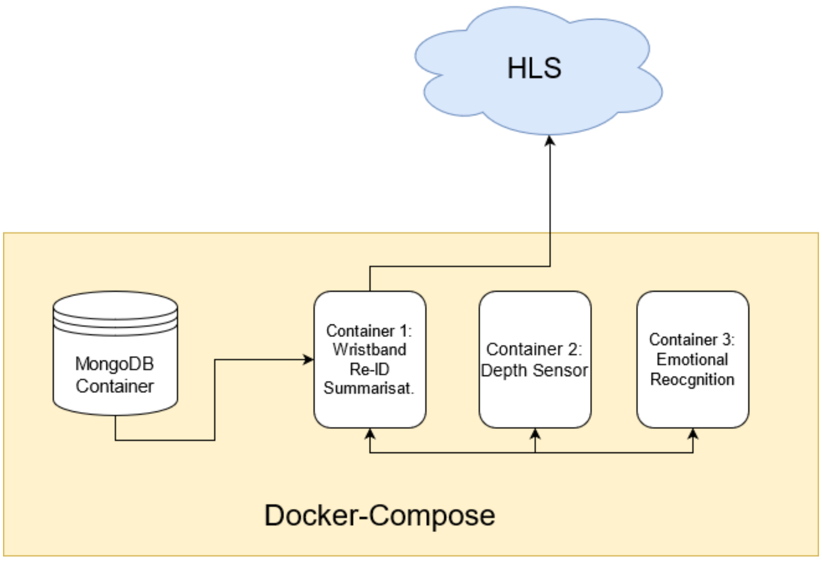

Figure 42 Abnormal detection module

In order to boost the modularity of the system, the CI/CD approach was adopted in TeNDER. It implies the separation into containers. These containers will oversee the implementation of functionalities associated to a particular sensor as described in Figure 42. Therefore, there is a main "orchestrator container" which extracts the information from the local mongo database. This library enables the access to the data. A set of functionalities containerised including:

- **The depth sensor container**. Implemented in Python 2.7, using Keras framework and the Microsoft CNTK Deep Learning library. This container is connected to the mongo via pymongo library and the central node.
- **The abd_band container.** Implemented in Python 3.6, includes the functionalities for the Fitbit-band, the microphone, the sleep sensor among others. It relies on Tensorflow 2.0.

Software Dependences

- Docker (ver. 19.03.8)
- docker-compose (ver. 1.26.2)

Build – Deployment

Build the container images:

```
docker build -f tnd-ab-dtc -t  tenderdev/abd_band:latest .
docker build -f tnd-ab-dtc2 -t  tenderdev/abd_kinect:latest .
```

Instantiate the service:

```
docker-compose up -d
```

Tests

The components and the provided endpoints are tested after the deployment phase.

Table 8 Mongo connectivity test.

Test name	ABD test
Test Purpose	Check ABD accesses to database and RabittMQ queues
Pre-test conditions	Docker installed
Test Tool	pyTest library
Test description	1. Receive a 200 code from database query 2. Post successfully test message into RabittMQ
Test Verdict	ABD is correctly connected and running

Command:

```
docker exec -t tnd-mongo-rest_mongo_api_1 sh -c "pytest tests”
```

Output:

```
======================= test session starts ====================================
platform linux -- Python 3.7.3, pytest-5.4.3, py-1.9.0, pluggy-0.13.1
cachedir: tests
rootdir: tests
collected 2 items

test_mongo.py                                                            [100%]
test_broker.py                                                           [100%]

======================= 2 passed in 0.2s =====================================
```

5.2 High Level Subsystem

5.2.1 Proxy and Authorization server (MAG)

Description

TeNDER secured gateway provides secure access and SSO service to all users and systems to TeNDER ecosystem. Currently, the implementation of the TeNDER gateway consists of a reverse proxy (Traefik [20]) and an authentication/authorization server (Keycloack [21]). The proxy server implements load balancing and handles the HTTPs certificates and the authorization server guarantees that each user will have access only in the service and data that are related to his account and role. Furthermore, this approach can be easily integrated with several infrastructure components i.e. Docker, Swarm Kubernetes, etc.

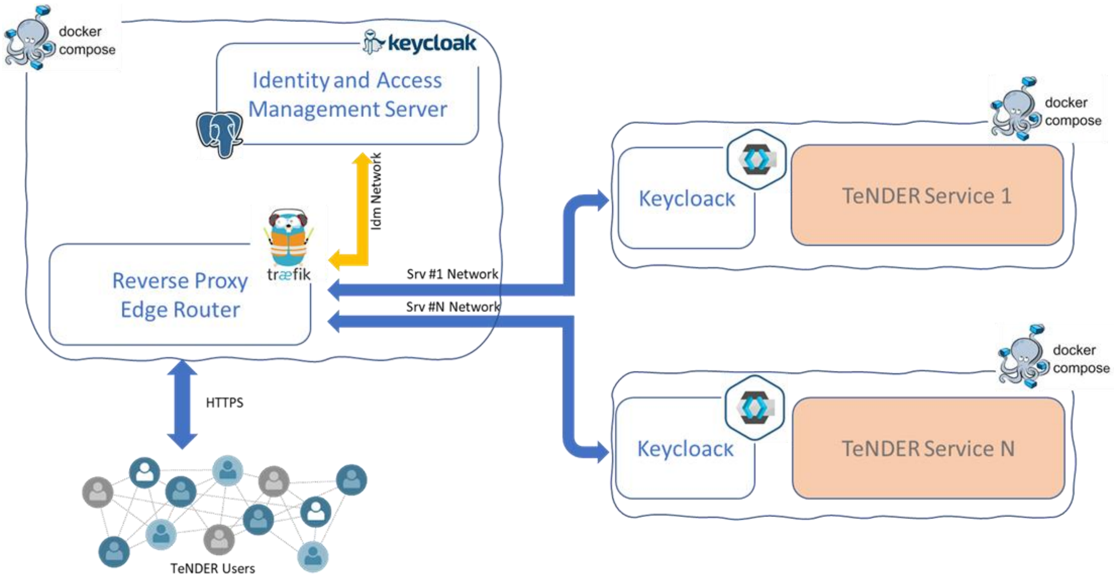

Figure 43 TeNDER secure proxy server

Software Dependences

- Docker (ver. 19.03.8)
- docker-compose (ver. 1.26.2)
- Postgres DB (ver. 11.2)
- Existence of two docker networks (idm_net, data_net)

Build – Deployment

Build the container images:

```
docker build -f auth-server/Dockerfile -t tender-auth-server .
```

Instantiate the service:

```
apk add --no-cache --upgrade bash
./create_networks.sh
docker-compose -f auth-server/docker-compose.yaml up -d
```

Access endpoints:

The administrator of the platform can use the GUI (Figure 44) interface of Keycloak server in order change the current configuration either by modifying the pre-loaded realms or by creating new ones. The GUI is available on the following endpoint:

```
https://auth-stage-tender.maggiolicloud.it/auth/
```

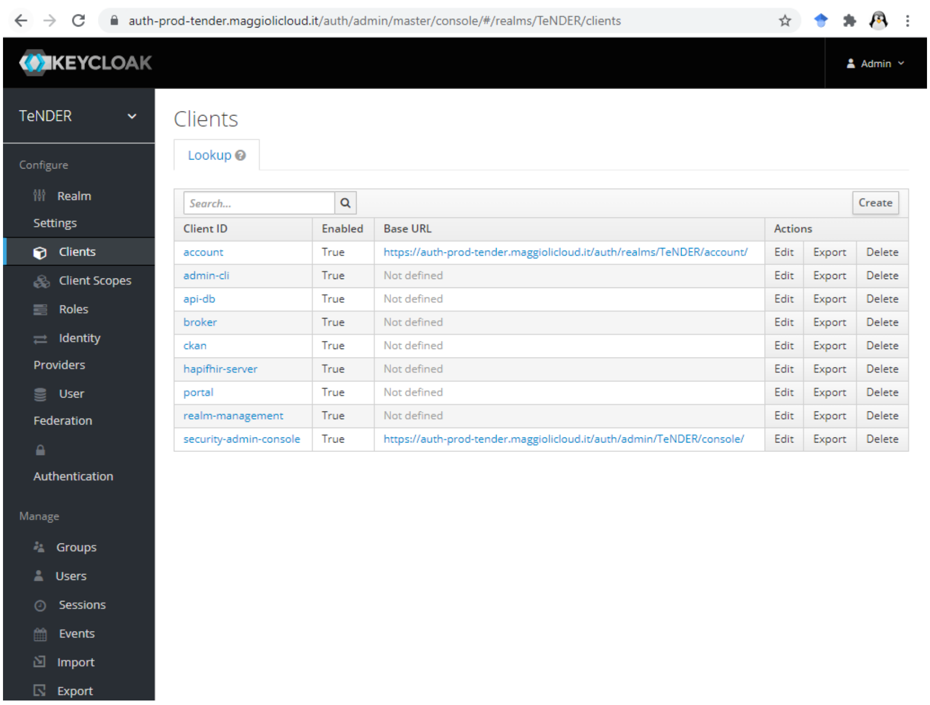

Figure 44 Clients on TeNDER realm.

Also, the proxy server offers web GUI (Figure 45) which provides the operational status of the server regarding the active backends/frontend endpoints and the health status of the microservices that are exposed to the public network. These dashboards are available on the following endpoints:

```
http://stage-tender.maggiolicloud.it:8081/dashboard/
```

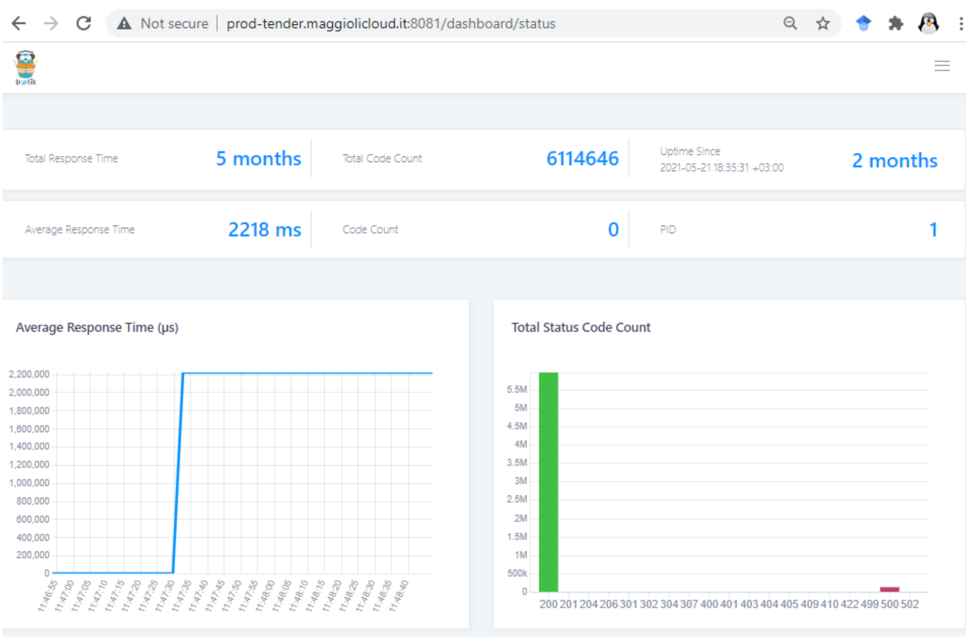

Figure 45 Traefik health dashboard

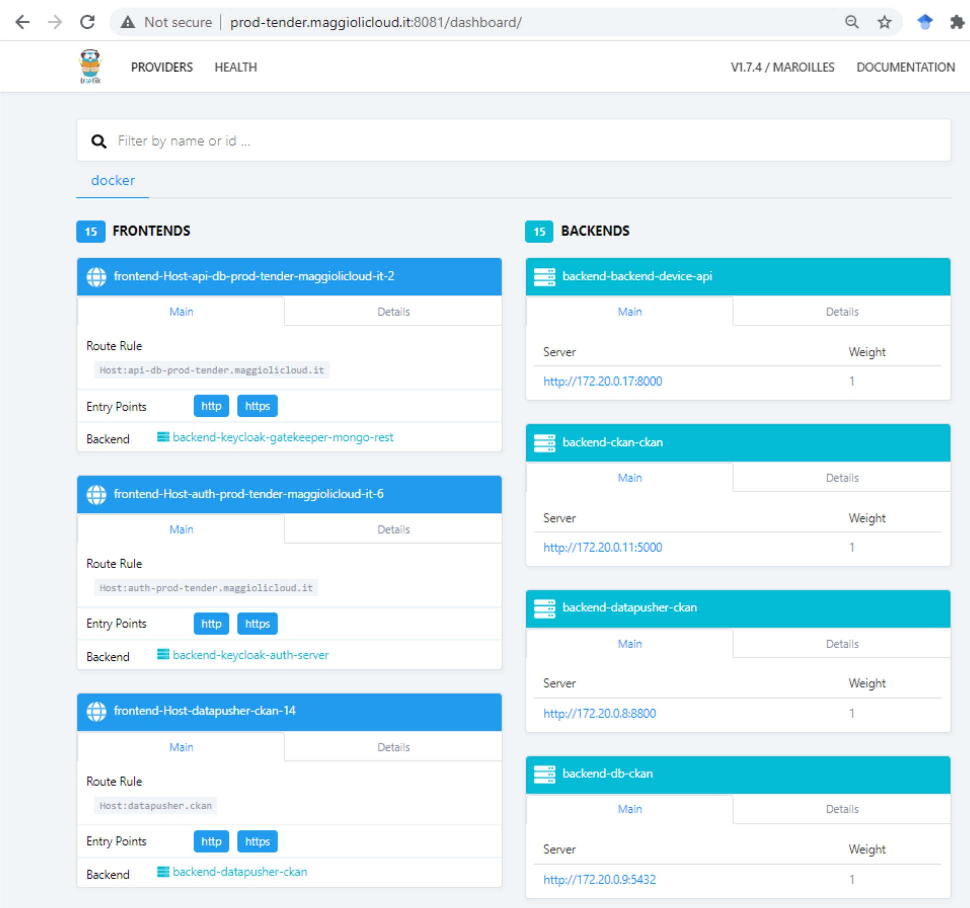

Figure 46 Traefik backend services

CI/CD Pipeline

```
image:
  name: docker/compose:latest
  entrypoint: ["/bin/sh", "-c"]
variables:
  GIT_STRATEGY: clone
  WORK_DIR: ${CI_PROJECT_NAME}
  BRANCH: ${CI_COMMIT_REF_NAME}
  REGISTRY: tender-registry:5000
  SHARED_PATH: /builds/$CI_PROJECT_PATH
stages:
  - build
  - deploy
  - tests
  - create_p_images
  - list_services
build_images:
    stage: build
    script:
        - docker build -f auth-server/Dockerfile -t tender-auth-server .
        - docker tag tender-auth-server $REGISTRY/tender-auth-server:sta
        - docker push $REGISTRY/tender-auth-server:sta
    tags:
     - stage
```

```
deploy_sec_gw:
    stage: deploy
    script:
        - apk add --no-cache --upgrade bash
        - ./create_networks.sh
        - docker-compose -f auth-server/docker-compose.yaml down
        - docker-compose -f auth-server/docker-compose.yaml up -d
    tags:
     - stage
tests:
    stage: unit_tests
    script:
        - tests/test-containers-status.sh
        - tests/test-realm.sh
        - tests/test-endpoints.sh
     tags:
     - stage
create_prod_images:
    stage: create_p_images
    script:
      - docker tag $REGISTRY/tender-auth-server:sta $REGISTRY/tender-auth-
server:prod
      - docker push $REGISTRY/tender-auth-server:prod
    when: manual
    tags:
     - stage
list_apps:
    stage: list_services
    script:
        - docker network ls
        - docker-compose -f auth-server/docker-compose.yaml ps
   tags:
     - stage
```

Tests

The provided APIs by the servers and their configuration are tested after the deployment phase.

Table 9 Test containers status

Test name	Application server connectivity
Test Purpose	Check the operational status of the containers consisting of the proxy and authorization service.
Pre-test conditions	The proxy and authorization service running on stage environment
Test Tool	Bash Automated Testing System (BATS)
Test description	1. Retrieve operational status of keyckoak, keycloak db and traefik containers. 2. Check if all containers are in running state
Test Verdict	Service has been deployed successfully

Command:

```
cd ./tests
./test-containers-status.sh
```

Output:

```
✓ check keycloak container
✓ check keycloak database container
✓ check traefik container

3 tests, 0 failures
```

Table 10 Test Keycloak configuration

Test name	**Application server connectivity**
Test Purpose	Check in the TeNDER realm has been loaded in keycloak server.
Pre-test conditions	The proxy and authorization service running on stage environment
Test Tool	Bash Automated Testing System (BATS)
Test description	1. Retrieve operational relams from keyckoak api 2. Check if TeNDER realm is configured
Test Verdict	Keycloak has ben configured correctly

Command:

```
cd ./tests
./ test-realm.sh
```

Output:

```
✓ check TeNDER Realm

1 test, 0 failures
```

Table 11 Test Proxy and Authorization servers endpoints

Test name	**Application server connectivity**
Test Purpose	Check the endpoints of keycloak and traefik servers
Pre-test conditions	The proxy and authorization service running on stage environment
Test Tool	Bash Automated Testing System (BATS)
Test description	1. Perform http GET request to the `https://auth-stage-tender.maggiolicloud.it/auth/` 2. Check HTTP response code (200) 3. Perform http GET request to the `http://stage-tender.maggiolicloud.it:8081/dashboard/` 4. Check HTTP response code (200)
Test Verdict	HTTP endpoints are available

Command:

```
cd ./tests
./test-endpoints.sh
```

Output:

```
✓ check keycloak endpoint
✓ check traefik endpoint

2 tests, 0 failures
```

5.2.2 Message broker and Consumer

Description

The collected data from the sensors are processed and then sent to the HAPI FHIR server, where a specific data structure and fields are needed to correctly store the information on it. To communicate with the HAPI FHIR server, there is an internal API which provides the necessary endpoints to efficiently read and write data. In order to optimize the workflow from the sensor's data collection to the HAPI FHIR server, a message broker was added and configured through the usage of rabbitMQ. By having a message broker, all the collected data, are handled, and published to specific topics where, depending on the rules implemented, hold them in queues that are consumed by authorized receivers (credentials are needed to have permission to get the messages).

As a receiver, there is a consumer, developed in Java, which is responsible for receiving the messages, serializing them into FHIR HL7 patterns and storing them in the server. Specific queues were created to help identify different messages which need specific serialization.

Important to mention that this implementation is fully integrated with the central Authorization and Authentication server of the platform.

Software Dependences

Message broker:

- RabbitMQ 3.7.4

Consumer:

- Maven 3.6.3
- Java JDK 11

Build – Deployment

Message broker:
Build the container images:

```
docker build --no-cache=true -t tnd-broker .
```

Instantiate the service:

```
docker run -d -p 8585:15671 -p 41757:5672 -p 51757:5671 -p 9419:9419 --name tnd-
broker -h tender-rmq $REGISTRY/tnd-broker:sta
```

Management access endpoint (Accessible from internal network only):

```
http://stage-tender.maggiolicloud.it:8585/
```

Consumer:

Build the container images:

```
docker build -t $REGISTRY/tender-consumer-ubw:prod -f dockerfile/prod/Dockerfile
. --build-arg TZ=UTC
```

Instantiate the service:

```
docker-compose -f workers/docker-compose.yml up -d
```

CI/CD Pipeline

Message broker:

```
image: docker
variables:
  WORK_DIR: ${CI_PROJECT_NAME}
  BRANCH: ${CI_COMMIT_REF_NAME}
  REGISTRY: tender-registry:5000
before_script:
  - docker info
stages:
  - build
  - test
  - deploy
  - list_services
  - create_p_images
build_project:
    stage: build
    script:
      - docker build --no-cache=true -t tnd-broker .
      - docker tag tnd-broker $REGISTRY/tnd-broker:sta
      - docker push $REGISTRY/tnd-broker:sta
    tags:
     - stage
deploy_project:
    stage: deploy
    script:
      - docker ps -a -q --filter "name=tnd-broker" | grep -q . && docker stop
tnd-broker && docker rm -fv tnd-broker
      - docker run -d -p 8585:15671 -p 41757:5672 -p 51757:5671 -p 9419:9419 --
name tnd-broker -h tender-rmq $REGISTRY/tnd-broker:sta
      - sleep 10
    tags:
     - stage
list_apps:
    stage: list_services
    script:
      - docker logs tnd-broker
      - docker logs tnd-dt-consumer
    tags:
     - stage
tests:
    stage: test
    script:
        - tests/message_execution_status.sh
        - tests/int-test-fir.sh
    tags:
    - stage
create_prod_images:
```

```
    stage: create_p_images
    script:
      - docker tag $REGISTRY/tnd-broker:sta $REGISTRY/tnd-broker:prod
      - docker push $REGISTRY/tnd-broker:prod
    when: manual
    tags:
     - stage
```

Consumer:

```
image:
  name: docker/compose:1.21.2
  entrypoint: ["/bin/sh", "-c"]
variables:
  GIT_STRATEGY: clone
  REGISTRY: tender-registry:5000
stages:
  - build
  - deploy
  - test
  - list_services
  - create_p_images
build_images:
  only:
    refs:
      - master
    variables:
      - $TENDER
  stage: build
  script:
    - echo "Build consumer image"
    - docker build --no-cache -t tender-consumer-ubw -f
dockerfile/stage/Dockerfile . --build-arg TZ=UTC
    - docker tag tender-consumer-ubw $REGISTRY/tender-consumer-ubw:sta
    - docker push $REGISTRY/tender-consumer-ubw:sta
  tags:
    - stage
deploy_workers:
  only:
    refs:
      - master
    variables:
      - $TENDER
  stage: deploy
  script:
    - echo "Remove consumer container"
    - docker stop consumer-rabbit || true
    - docker rm consumer-rabbit || true
    - echo "Recreate container"
    - docker-compose -f docker-compose-stage.yml up -d --force-recreate
  tags:
    - stage
list_apps:
  only:
    refs:
      - master
    variables:
      - $TENDER
  stage: list_services
  script:
    - docker logs tender-consumer-ubw
    - docker ps
  tags:
    - stage
```

```
tests:
    stage: test
    script:
        - tests/consumer_execution_status.sh
        - tests/int-test-fir.sh
    tags:
    - stage
create_prod_images:
  only:
    refs:
      - master
    variables:
      - $TENDER
  stage: create_p_images
  script:
    - echo "Remove old consumer prod image"
    - docker rmi $REGISTRY/tender-consumer-ubw:prod || true
    - echo "Remove old scheduler prod image MISSING"
    - echo "Build and push new consumer prod image"
    - docker build -t $REGISTRY/tender-consumer-ubw:prod -f
dockerfile/prod/Dockerfile . --build-arg TZ=UTC
    - docker push $REGISTRY/tender-consumer-ubw:prod
  when: manual
  tags:
    - stage
```

Tests

Both components are tested together to test the workflow from the publish to the receive and data handling.

Table 13 - Test Message Broker and Consumer's Execution and Workflow

Test name	Application server connectivity
Test Purpose	Check the execution of the rabbitMQ instance and if its topics and queues are correctly created. If so, verify if it's ready to store all the data published and, when a request to receive the data is made, the information is pop to the consumer that will handle the information and store it in the HAPI FHIR server.
Pre-test conditions	The rabbitMQ running in a local or cloud environment. Deploy consumer in a local or cloud environment. Have HAPI FHIR server working in a local or cloud environment.
Test Tool	Shell Script (sh)
Test description	1. In case of TeNDER staging environment, perform a curl command to the IP and Port where the message broker is instantiated; 2. Publish data to the message broker; 3. Check if information was stored in HAPI FHIR server.
Test Verdict	Message broker and consumer are running and working properly

Command:

```
bash tests/consumer_execution_status.sh
```

Output:

```
HTTP/1.1 200 OK
content-length: 1391
content-type: text/html
date: Thu, 05 Aug 2021 18:59:30 GMT
etag: "804493663"
last-modified: Wed, 02 Jun 2021 10:21:22 GMT
server: Cowboy
```

Command:

```
bash int-test-fir.sh
```

Output:

```
* Request for authorization

* Display token
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJoSWFCeFBRV3hrVzd6NkhNX1U2VHBvVVRqTnBSMk9hVExWd3A0VktZdl
CJpc3MiOiJodHRwczovL2F1dGgtc3RhZ2UtdGVuZGVyLm1hZ2dpb2xpY2xvdWQuaXQvYXV0aC9yZWFsbXMvVGVOREVSIiwiYXVkIjpbImh
g5OGQ5ZiIsInR5cCI6IkJlYXJlciIsImF6cCI6ImhhcGlmaGlyLXNlcnZlciIsImF1dGhfdGltZSI6MCwic2Vzc2lvbl9zdGF0ZSI6IjZl
kZXIubWFnZ2lvbGljbG91ZC5pdCIsImh0dHA6Ly9sb2NhbGhvc3Q6ODEwMCJdLCJyZWFsbV9hY2Nlc3MiOnsicm9sZXMiOlsidGVuZGVyL
Olsidmlldy1pZGVudGl0eS1wcm92aWRlcnMiLCJ2aWV3LXJlYWxtIiwibWFuYWdlLWlkZW50aXR5LXByb3ZpZGVycyIsImltcGVyc29uYX
S1jbGllbnRzIiwicXVlcnktdXNlcnMiLCJtYW5hZ2UtZXZlbnRzIiwibWFuYWdlLXJlYWxtIiwidmlldy1ldmVudHMiLCJ2aWV3LXVzZXJ
I6WyJtYW5hZ2UtYWNjb3VudCIsIm1hbmFnZS1hY2NvdW50LWxpbmtzIiwidmlldy1wcm9maWxlIl19fSwic2NvcGUiOiJhcGktZGIgZW1h
hZG1pbiIsIm9mZmxpbmVfYWNjZXNzIiwidW1hX2F1dGhvcml6YXRpb24iXSwibmFtZSI6IkFkbWluIFRlbmRlciIsInByZWZlcnJlZF91c
VQoAJBqQWqeZFUYXMvUFp-f K WvQgANHxHezrp12cIgQYr eyG1ZNUCN0nUc3GYJBK8MZLDuGnB9hNZ mD1fq2QNpkdaspLbnXRGHAU14
BdqEkRzpJbjpGPn4BqQhnaxDtx0HKcw0FxwV4zsLwGv-2bPl2IKTqBHgAd7RxlUeLvI0AzIOPAT0NHuYjRw1AI2woqXoPQWWgQclh2JdA

 * CREATE DOCTOR
DOCTOR created: 27958

 * CREATE PATIENT
PATIENT created: 27960

 * CREATE DEVICES
DEVICE SLEEP created: 27964
DEVICE BND created: 27965
DEVICE KIN created: 27966
DEVICE POS created: 27967
DEVICE MIC created: 27968

 * POST DATA
ABD data published with success
Band data published with success
Rehab data published with success

 * Waiting for the consumer to handle the data...

 * Data Handled. Getting the data from HAPI FHIR

GET Sleep tracker data
SUCCESS (2)
GET Heart bit and Steps data
SUCCESS (1)
GET position tracker data
SUCCESS (1)
GET Fall detection data
SUCCESS (3)
GET Rehub data
SUCCESS (1)
```

Figure 42 - Publish/Receive and HAPI FHIR's data storage check.

5.2.3 Electronic Health Record server

Description

For the electronic health record, an instance of the HAPI FHIR server was integrated. It provides a full implementation of the HL7 FHIR standard for healthcare interoperability, designed to facilitate the flexible integration of FHIR resources in applications/systems, allowing different clients to connect.

Regarding server interaction, the FHIR standard implementation provides an HTTP API to enable CRUD operations (create, delete, read and update) on the database, supporting different deployment schemes and relational databases.

For the TeNDER project, the server was deployed with the standard tools, having a PostgreSQL instance integrated as an open-source object-relational database system and an API interaction using the structure already defined for each resource [22].

Concerning authentication, a new layer was implemented in the server to verify the request's authenticity. Since the service that manages the authentication is Keycloak, which is used on every TeNDER component that needs authorization management, a token must be used to validate them. The token is generated in the login phase through Keycloak's API and returned to the final user who will use it in every request made to the HAPI FHIR as an Authorization token.

Software Dependencies

- Maven 3.6.3
- Java JDK 11
- Tomcat 9
- Java JRE 11

Build – Deployment

Build the container images

```
docker build -t $REGISTRY/hapi-fhir-jpaserver:prod -f dockerfile/prod/Dockerfile
```

Instantiate the service

```
docker-compose -f hapi-fhir/docker-compose.yml up -d
```

Access endpoint

```
https://hapi-prod-tender.maggiolicloud.it/hapi-fhir-jpaserver/fhir/<resource>
```

CI/CD Pipeline

```
image:
  name: docker/compose:1.21.2
  entrypoint: ["/bin/sh", "-c"]
variables:
  GIT_STRATEGY: clone
  WORK_DIR: ${CI_PROJECT_NAME}
  BRANCH: ${CI_COMMIT_REF_NAME}
  REGISTRY: tender-registry:5000
  SHARED_PATH: /builds/$CI_PROJECT_PATH
stages:
  - build
  - deploy
  - list_services
  - int-test
  - create_p_images
build_images:
  only:
    refs:
      - master
```

```
  stage: build
  script:
    - docker build -t hapi-fhir-jpaserver -f dockerfile/stage/Dockerfile .
    - docker tag hapi-fhir-jpaserver $REGISTRY/hapi-fhir-jpaserver:sta
    - docker push $REGISTRY/hapi-fhir-jpaserver:sta
  tags:
    - stage
deploy_hapi_fhir:
  only:
    refs:
      - master
  stage: deploy
  script:
    - docker-compose -f docker-compose-stage.yml up -d --build hapi-fhir-
jpaserver-start
  tags:
    - stage
list_apps:
  only:
    refs:
      - master
  stage: list_services
  script:
    - docker-compose logs
    - docker-compose ps
  tags:
    - stage
run_test:
    stage: int-test
    script:
        - apk update
        - apk add curl
        - curl -s -X POST -F token=0f2c5b4019231cd48f49fe229746f2 -F ref=master -
F "variables[TEST_SCRIPT]=int-test-hfir.sh"
https://tendergitlab.maggiolicloud.it/api/v4/projects/32/trigger/pipeline
    tags:
     - stage
create_prod_images:
  only:
    refs:
      - master
  stage: create_p_images
  script:
    - docker rmi $REGISTRY/hapi-fhir-jpaserver:prod || true
    - docker build -t $REGISTRY/hapi-fhir-jpaserver:prod -f
dockerfile/prod/Dockerfile .
    - docker push $REGISTRY/hapi-fhir-jpaserver:prod
  when: manual
  tags:
    - stage
```

Tests

The HAPI FHIR already provides internal tests in every package used every time the image is built. It assures the correct integration of the packages with valid versions and allows the inclusion of custom tests.

For the TeNDER case, were added interceptors and filters to increase the necessary logic into specific requests before their processing and guarantee the correct workflow.

Table 13 - HAPI FHIR HTTP API connectivity test

Test name	**Test snapshots versions and usability**
Test Purpose	Each HAPI FHIR's package should be tested to always have the correct versions and to avoid package bugs in deployment phase
Pre-test conditions	A local or cloud environment with mvn (Maven) installed
Test Tool	Maven
Test description	1. Go to each HAPI FHIR's package 2. Run internal tests 3. Wait until all tests passed
Test Verdict	All external packages are updated and working properly

Command:

```
mvn -P ALLMODULES,NOPARALLEL clean install
```

Output:

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary for HAPI-FHIR 5.4.0-PRE1-SNAPSHOT:
[INFO]
[INFO] HAPI-FHIR .......................................... SUCCESS [  0.819 s]
[INFO] HAPI FHIR - Deployable Artifact Parent POM ......... SUCCESS [  1.382 s]
[INFO] HAPI FHIR BOM ...................................... SUCCESS [  0.735 s]
[INFO] HAPI FHIR - Core Library ........................... SUCCESS [ 18.740 s]
[INFO] HAPI FHIR - Server Framework ....................... SUCCESS [ 15.426 s]
[INFO] HAPI FHIR - Validation Resources (DSTU3) ........... SUCCESS [  2.632 s]
[INFO] HAPI FHIR - Client Framework ....................... SUCCESS [  9.304 s]
[INFO] HAPI FHIR Test Utilities ........................... SUCCESS [  5.619 s]
[INFO] HAPI FHIR Structures - DSTU3 ....................... SUCCESS [ 52.907 s]
[INFO] HAPI FHIR - Validation Resources (FHIR R4) ......... SUCCESS [  2.600 s]
[INFO] HAPI FHIR Structures - FHIR R4 ..................... SUCCESS [07:53 min]
[INFO] HAPI FHIR - Validation Resources (DSTU2) ........... SUCCESS [  1.155 s]
[INFO] HAPI FHIR Structures - HL7.org DSTU2 ............... SUCCESS [ 22.734 s]
[INFO] HAPI FHIR - Validation Resources (FHIR R5) ......... SUCCESS [  1.731 s]
[INFO] HAPI FHIR Structures - FHIR R5 ..................... SUCCESS [ 17.570 s]
[INFO] HAPI Tinder Plugin ................................. SUCCESS [ 16.256 s]
[INFO] HAPI FHIR Structures - DSTU2 (FHIR v1.0.0) ......... SUCCESS [ 28.203 s]
[INFO] HAPI FHIR - Validation Resources DSTU2.1 (2016May) . SUCCESS [  1.609 s]
[INFO] HAPI FHIR Structures - DSTU2.1 (2016May) ........... SUCCESS [ 21.497 s]
[INFO] HAPI FHIR - Converter .............................. SUCCESS [ 12.352 s]
[INFO] HAPI FHIR - Validation ............................. SUCCESS [02:33 min]
[INFO] HAPI FHIR OkHttp Client ............................ SUCCESS [ 10.817 s]
[INFO] HAPI FHIR JAX-RS Server ............................ SUCCESS [ 27.975 s]
[INFO] HAPI FHIR - Master Data Management ................. SUCCESS [ 13.576 s]
[INFO] HAPI FHIR JPA Model ................................ SUCCESS [ 10.029 s]
[INFO] HAPI FHIR Search Parameters ........................ SUCCESS [ 36.261 s]
[INFO] HAPI FHIR JPA API .................................. SUCCESS [  6.263 s]
[INFO] HAPI FHIR Subscription Server ...................... SUCCESS [ 33.781 s]
[INFO] HAPI FHIR JPA Server - Batch Task Processor ........ SUCCESS [  5.336 s]
[INFO] HAPI FHIR JPA Server ............................... SUCCESS [24:27 min]
[INFO] HAPI FHIR TestPage Overlay ......................... SUCCESS [  6.210 s]
[INFO] HAPI FHIR - Docs ................................... SUCCESS [  6.763 s]
[INFO] HAPI FHIR JPA Server Test Utilities ................ SUCCESS [  7.436 s]
[INFO] HAPI Tinder Plugin - Test Project .................. SUCCESS [ 20.751 s]
[INFO] HAPI FHIR JPA Server - Clinical Quality Language ... SUCCESS [02:49 min]
[INFO] HAPI FHIR JAX-RS Server - Example .................. SUCCESS [ 19.754 s]
[INFO] HAPI FHIR JPA Server - Migration ................... SUCCESS [ 51.044 s]
[INFO] HAPI FHIR JPA Server - Master Data Management ...... SUCCESS [01:13 min]
[INFO] HAPI FHIR Sample RESTful Server .................... SUCCESS [  2.108 s]
[INFO] HAPI FHIR - fhirtest.uhn.ca Deployable WAR ......... SUCCESS [  8.945 s]
[INFO] HAPI FHIR - Android ................................ SUCCESS [  8.934 s]
[INFO] HAPI FHIR - Command Line Client - Server WAR ....... SUCCESS [  9.632 s]
[INFO] HAPI FHIR - Command Line Client - API .............. SUCCESS [ 10.983 s]
[INFO] HAPI FHIR - Command Line Client - Base Project ..... SUCCESS [  0.007 s]
[INFO] HAPI FHIR - Command Line Client - Application ...... SUCCESS [ 36.433 s]
[INFO] HAPI FHIR - Distribution Archive ................... SUCCESS [  0.019 s]
[INFO] HAPI FHIR JAX-RS Server Kotlin test ................ SUCCESS [ 12.265 s]
[INFO] HAPI FHIR - Minimal Dependency Test - Client ....... SUCCESS [  4.175 s]
[INFO] HAPI FHIR - Minimal Dependency Test - Server ....... SUCCESS [  7.487 s]
[INFO] hapi-fhir-spring-boot-autoconfigure ................ SUCCESS [ 12.366 s]
[INFO] hapi-fhir-spring-boot-starter ...................... SUCCESS [  1.088 s]
[INFO] hapi-fhir-spring-boot .............................. SUCCESS [  0.010 s]
[INFO] hapi-fhir-spring-boot-samples ...................... SUCCESS [  0.011 s]
[INFO] hapi-fhir-spring-boot-sample-client-apache ......... SUCCESS [  1.371 s]
[INFO] hapi-fhir-spring-boot-sample-client-okhttp ......... SUCCESS [  1.445 s]
[INFO] hapi-fhir-spring-boot-sample-server-jersey ......... SUCCESS [ 12.505 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  49:17 min
[INFO] Finished at: 2021-08-04T19:22:41+01:00
[INFO] ------------------------------------------------------------------------
```

Figure 40 TeNDER's EHR unit tests

Table 14 HAPI FHIR Interceptors Tests

Test name	Test snapshots versions and usability
Test Purpose	Some requests made to HAPI FHIR are intercepted to add more logic and combine information with other microservices
Pre-test conditions	A local or cloud environment with HAPI FHIR running
Test Tool	Shell Script (sh)

Test description	1. Simulate calls with interceptors 2. Assert positive responses 3. Wait until all tests passed
Test Verdict	All interceptors are working properly

Command:

```
bash int-test.sh
```

Output:

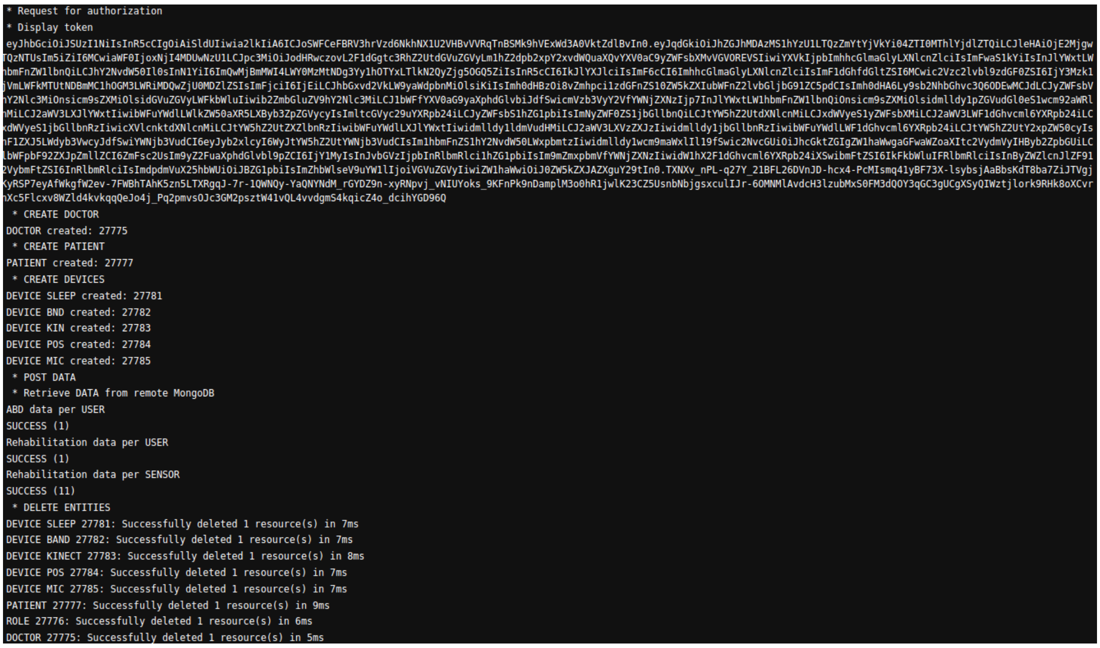

Figure 41 HAPI FHIR Interceptors Tests.

5.2.4 Remote Document DB

Description

One of the databases which consists of the TeNDER platform is a document-based mongo DB. In this DB anonymized data coming from the LLS, through the message broker, are stored to be further analysed by TeNDER services. To enhance the secure interconnection between the DB and the rest of the services regardless the programming language and the technology which are used from the rest services an HTTP REST API has been developed. This API can be accessed directly from the internal services of the platform via an internal private network. In case which TeNDER platform is deployed in different servers the same API is provided over HTTPS and it is fully integrated with central Authorization and Authentication server of the platform.

Software Dependences

- django 3.0.4
- mongoengine 0.19.1
- django-rest-framework-mongoengine 3.4.1
- pymongo 3.10.1

Build – Deployment

Build the container images:

```
docker build -f mongo_rest/Dockerfile -t tender-mongo_api .
docker build -f nginx/Dockerfile -t tender-mongo_api_fsrv .
```

Instantiate the service:

```
docker-compose up -d
```

Access endpoint:

```
https://api-db-stage-tender.maggiolicloud.it/api/v1/docs/
```

CI/CD Pipeline

```
image:
  name: docker/compose:latest
  entrypoint: ["/bin/sh", "-c"]
variables:
  GIT_STRATEGY: clone
  WORK_DIR: ${CI_PROJECT_NAME}
  BRANCH: ${CI_COMMIT_REF_NAME}
  REGISTRY: tender-registry:5000
  SHARED_PATH: /builds/$CI_PROJECT_PATH
before_script:
  - docker --version
  - docker-compose --version
stages:
  - build
  - deploy
  - unt-test
  - int-test
  - create_p_images
build_images:
    stage: build
    script:
        - docker build -f mongo_rest/Dockerfile -t tender-mongo_api .
        - docker build -f nginx/Dockerfile -t tender-mongo_api_fsrv .
        - docker build -f nginx/Dockerfile-gk -t tender-gtkeeper-mongo-api .
        - docker tag tender-mongo_api $REGISTRY/tender-mongo_api:sta
        - docker tag tender-mongo_api_fsrv $REGISTRY/tender-mongo_api_fsrv:sta
        - docker tag tender-gtkeeper-mongo-api $REGISTRY/tender-gtkeeper-mongo-
api:sta
        - docker push $REGISTRY/tender-mongo_api:sta
        - docker push $REGISTRY/tender-mongo_api_fsrv:sta
        - docker push $REGISTRY/tender-gtkeeper-mongo-api:sta
    tags:
     - stage
deploy_mongo_api:
    stage: deploy
    script:
        - docker-compose down
        - docker-compose up -d
    tags:
     - stage
unit_tests:
    stage: unt-test
    script:
        - docker exec -t tnd-mongo-rest_mongo_api_1 sh -c "python3 manage.py test
api.tests.DBConTestCase"
        - docker exec -t tnd-mongo-rest_mongo_api_1 sh -c "python3 manage.py test
api.tests.ApisTestCase"
```

```
    tags:
     - stage
run_test:
    stage: int-test
    script:
        - apk add curl
        - curl -s -X POST -F token=0f2c5b4019231cd48f49fe229746f2 -F ref=master -
F "variables[TEST_SCRIPT]=int-test.sh"
https://tendergitlab.maggiolicloud.it/api/v4/projects/32/trigger/pipeline
    tags:
     - stage
create_prod_images:
    stage: create_p_images
    script:
      - docker tag $REGISTRY/tender-mongo_api:sta $REGISTRY/tender-mongo_api:prod
      - docker tag $REGISTRY/tender-mongo_api_fsrv:sta $REGISTRY/tender-
mongo_api_fsrv:prod
      - docker tag $REGISTRY/tender-gtkeeper-mongo-api:sta $REGISTRY/tender-
gtkeeper-mongo-api:prod
      - docker push $REGISTRY/tender-mongo_api:prod
      - docker push $REGISTRY/tender-mongo_api_fsrv:prod
      - docker push $REGISTRY/tender-gtkeeper-mongo-api:prod
    when: manual
    tags:
     - stage
```

Tests

All the components and their endpoints are tested during the deployment phase in the stage environment.

Table 12 Mongo HTTP API connectivity test.

Test name	Application server connectivity
Test Purpose	Check connectivity between application server and MongoDB
Pre-test conditions	The MongoDB REST service running on stage environment
Test Tool	`django.test`
Test description	3. Create a new record via HTTP POST 4. Retrieve the data based on patient ID 5. Check if the posted and retrieved data are equal
Test Verdict	Application server has access to Mongo

Command:

```
docker exec -t tnd-mongo-rest_mongo_api_1 sh -c "python3 manage.py test
api.tests.DBConTestCase”
```

Output:

```
System check identified no issues (0 silenced)...........
-------------------------------------------------------------------
Ran 1 test in 0.012s
OK
```

Table 13 Mongo HTTP API Rehabilitation test.

Test name	Rehabilitation service API
Test Purpose	Check the API for Rehabilitation data

Pre-test conditions	The MongoDB REST service running on stage environment
Test Tool	`django.test`
Test description	1. Retrieve data from `/api/v1/summarization/rehabilitation/` 2. Check HTTP status code (200 OK)
Test Verdict	API is functional

Table 14 Mongo HTTP API smart band test.

Test name	**Smart-band service API**
Test Purpose	Check the API for smart-band data
Pre-test conditions	The MongoDB REST service running on stage environment
Test Tool	`django.test`
Test description	1. Retrieve data from `/api/v1/summarization/ band/` 2. Check HTTP status code (200 OK)
Test Verdict	API is functional

Table 15 Mongo HTTP API ABD test.

Test name	**Abnormal detection service (ABD)**
Test Purpose	Check the API for ADB data
Pre-test conditions	The MongoDB REST service running on stage environment
Test Tool	`django.test`
Test description	1. Retrieve data from `/api/v1/summarization/adb/` 2. Check HTTP status code (200 OK)
Test Verdict	API is functional

Command:

```
docker exec -t tnd-mongo-rest_mongo_api_1 sh -c "python3 manage.py test
api.tests.ApisTestCase"
```

Output:

```
System check identified no issues (0 silenced)..........
-----------------------------------------------------------------
Ran 10 tests in 0.036s
OK
```

5.2.5 Web GUI (UBI)

Description

The TeNDER project, besides the mobile application developed for patients and caregivers, had to provide an efficient and interactive way for administrators and health professionals to interact with the system. Since these users will mostly manage resources, the solution was the development of a web application. Each user has his private area and access only to the information that his role allows.

Until now, there are two developed interfaces: the administrator and health professional interface. The administrator interface is structured to provide the needed functionalities for the correct management of its organization, users, and devices. Each administrator is related to an organization, and all the users and devices created are managed by him. Besides creating users and devices, the administrator can: edit, delete, and change their status (active or deactivate); create relations between users; filter and visualize quantitative information; check the devices usage timespan and other helpful measures.

Regarding the health professional interface, its focus is on the user patients. Each health professional has his patients and can be accessed individually in the web application. The details page of each patient provides an organized custom dashboard, where the user can monitor and follow the patient's collected information from its general information to sleep tracker, localization tracker, and many others.

Software Dependences

- Docker
- NodeJS
- Browser with Javascript support
- Npm or Yarn

Build – Deployment

Build the container images:

```
docker build --no-cache -t $REGISTRY/tender-web-app-ubw:prod -f
deployment/dockerfile/production/Dockerfile . --build-arg TZ=UTC
```

Instantiate the service:

```
docker-compose -f webapp/docker-compose.yml up -d
```

Access endpoint:

```
https://prod-tender.maggiolicloud.it/
```

CI/CD Pipeline

```
image:
  name: docker/compose:1.21.2
  entrypoint: ["/bin/sh", "-c"]

variables:
  GIT_STRATEGY: clone
  WORK_DIR: ${CI_PROJECT_NAME}
  BRANCH: ${CI_COMMIT_REF_NAME}
  REGISTRY: tender-registry:5000
  SHARED_PATH: /builds/$CI_PROJECT_PATH

before_script:
  - echo ${CI_PROJECT_PATH}
  - echo ${SHARED_PATH}
```

```
  - touch ${SHARED_PATH}/test_file
  - pwd
  - ls -ll
  - docker --version
  - docker-compose --version

stages:
  - build
  - deploy
  - list_services
  - create_p_images
  - promote

build_image:
  only:
    refs:
      - staging
  stage: build
  script:
    - docker --version
    - docker info
    - docker rmi web-app || true
    - docker rmi $REGISTRY/tender-web-app-ubw:sta || true
    - echo "Build web app image"
    - docker build --no-cache -t web-app -f
deployment/dockerfile/staging/Dockerfile . --build-arg TZ=UTC
    - docker tag web-app $REGISTRY/tender-web-app-ubw:sta
    - docker push $REGISTRY/tender-web-app-ubw:sta
  tags:
    - stage

deploy_web:
  only:
      refs:
        - staging
  stage: deploy
  script:
    - echo "Remove web app container"
    - docker stop web_app || true
    - docker rm web_app || true
    - echo "Recreate container"
    - docker-compose -f deployment/docker-compose-stage.yml up -d --force-
recreate
  tags:
    - stage

list_apps:
  only:
      refs:
        - staging
  stage: list_services
  script:
    - docker logs web_app
    - docker ps
  tags:
    - stage

create_prod_images:
  only:
    refs:
      - staging
  stage: create_p_images
  script:
    - docker rmi $REGISTRY/tender-web-app-ubw:prod || true
```

```
    - docker build --no-cache -t $REGISTRY/tender-web-app-ubw:prod -f
deployment/dockerfile/production/Dockerfile . --build-arg TZ=UTC
    - docker push $REGISTRY/tender-web-app-ubw:prod
  when: manual
  tags:
    - stage

promote to staging:
  stage: promote
  when: on_success
  only:
    - master
  before_script:
    - apk --no-cache add git
    - export GIT_AUTHOR_NAME="Ubiwhere Release Tools"
    - export GIT_AUTHOR_EMAIL="ci@ubiwhere.com"
    - mkdir /root/.ssh/ && echo "${SSH_PRIVATE_KEY}" > /root/.ssh/id_rsa
    - git config --global http.sslverify "false"
    - git remote add maggioli "Error! Hyperlink reference not valid.}"
  script:
    - git config user.name "Ubiwhere Release Tools"
    - git config user.email "ci@ubiwhere.com"
    - git pull maggioli staging
    - git push maggioli HEAD:staging
```

Tests

At this stage, the tests implemented are only for auxiliary / utility functions. The main reasons are:

- A lot of these functions depend on frameworks or data that can change between browsers and locales, and since the Jest [23] testing environment is locale- and browser-agnostic, it can pick up on issues that can often go undetected;
- In future stages, a lot of the frontend structure might change radically, so it was decided to not implement unit tests for screens and components.

To run the tests, a Powershell or Git Bash terminal with Node.js and npm installed is needed, then from project's folder execute the following:

Command:

```
npm run test
```

Output:

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	55.33	36.55	37.62	57.69	
src	25	2.08	13.33	23.85	
config.js	100	50	100	100	12
globalStyles.js	83.33	100	50	100	
utils.js	100	100	100	100	
validations.js	11.7	0	0	11.7	... 69,171,172,174
src/assets/icons/Moods/Happy	50	0	0	50	
index.js	50	0	0	50	5
src/assets/icons/Moods/Sad	50	0	0	50	
index.js	50	0	0	50	5
src/components/CustomCalendarDay	100	100	100	100	
utils.js	100	100	100	100	
src/containers/Admin/CreateCaregiver	100	62.5	100	100	
utils.js	100	62.5	100	100	68,69,70
src/containers/Admin/CreateDevice	85.71	60	66.67	85.71	
utils.js	85.71	60	66.67	85.71	46
src/containers/LanguageSelector	100	100	100	100	
utils.js	100	100	100	100	
src/containers/Localization	61.29	50	36.84	64.29	
utils.js	61.29	50	36.84	64.29	... 23,126,127,131
src/containers/PatientInfo	100	100	100	100	
utils.js	100	100	100	100	
src/containers/SafetyAndWellbeing	94.44	92.59	85.71	100	
utils.js	94.44	92.59	85.71	100	65
src/containers/SleepDiary	50.91	3.51	13.79	62.79	
utils.js	50.91	3.51	13.79	62.79	... 74,185,206,219
src/mocks	100	100	100	100	
mocks.js	100	100	100	100	
src/scenes/Admin/ListDevices	100	80	100	100	
utils.js	100	80	100	100	14,17
src/scenes/Doctor/PatientsList	91.67	80	100	90.91	
utils.js	91.67	80	100	90.91	36
src/scenes/Home	52.17	29.58	35.71	57.14	
utils.js	52.17	29.58	35.71	57.14	... 68,73,80,94,98

```
Test Suites: 13 passed, 13 total
Tests:       25 passed, 25 total
Snapshots:   0 total
Time:        17.518s
Ran all test suites.
```

Figure 42 – Web UI tests

For better tests comprehension, the following tables will give an overview and explanation of each test.

Table 16 Utils Component for Custom Calendar Day

Target	getDotColor() function
Purpose	Checks if correct colors are returned for the Reminder calendar dots that appear on the days.

Table 17 Utils Component for Custom List Item

Target	parseName() function
Purpose	Checks if the function accurately converts a name in string format to JSON object with “family” and “given” fields

Table 18 Utils Component for Custom Calendar Day

Target	getPatientName() function
Purpose	Checks if the function accurately fetches a patient's name given a reference string and patient list, and returns a string of the full name of the patient concatenated with their username

Table 19 Utils for Admin's Container

Target	checkLocationValidation() function
Purpose	Checks if the function accurately returns the correct values when supplied with a Kinect device form and a patient list that may or may not be empty

Table 20 Utils for Language Selector's Container

Target	getFlagEmoji() function
Purpose	Checks if the function accurately returns the correct country emoji for each of the available language locales, or no emoji if no available or valid locale is provided.

Table 21 Utils for Localization's Container

Target	getPatientLocations() function
Purpose	Checks if the function accurately returns an array of strings of each of the Environment present in a given user's data
Target	formatHourLabel() function
Purpose	Checks if the function returns a human-readable hour string ("HH:MM") when provided with a calculated hour from localization data (ex: 1.30 -> 01:30)
Target	secondsToHours() function
Purpose	Checks if the function correctly converts a number of seconds into a number of hours (ex: 3600 -> 1.00)

Target	secondsToMinutes() function
Purpose	Checks if the function correctly converts a number of seconds into a number of minutes (ex: 300 -> 5.00)
Target	secondsToHoursMinutes() function
Purpose	Checks if the function correctly converts a number of seconds into a formatted hour / minute string (ex: 3600 -> “1h00m”)
Target	getBackgroundColor() function
Purpose	Checks if the function accurately returns the intended color for each of the locations bars to be displayed on the graph

Table 22 Utils for Patient Info’s Container

Target	convertRoleToTranslation() function
Purpose	Checks if the function accurately converts a provided string to a Snake Case equivalent used for i18n translation keys (ex: “Formal Caregiver” -> “formal_caregiver”

Table 23 Utils for Safety and Wellbeing’s Container

Target	getEmotionIcon() function
Purpose	Checks if the function accurately returns the Happy icon component or Sad icon component (used in the Emotional State information component) depending on the provided emotion string
Target	getEmotionHighlightColor() function
Purpose	Checks if the function accurately returns the correct color depending on the provided emotion string (used for color-coding Emotional State info)
Target	calculateEmotionalState() function
Purpose	Checks if the function accurately converts a list of Emotional State values into a JSON object containing the number of “happy”

	instances, "sad" instances, and the total number of values

Table 24 Utils for Sleep Diary's Container

Target	formatHourLabel() function
Purpose	Checks if the function returns a human-readable hour string when provided with a calculated hour from localization data (see Containers/Localization/Utils)
Target	secondsToHours() function
Purpose	Checks if the function correctly converts a number of seconds into a number of hours (see Containers/Localization/Utils)
Target	secondsToMinutes() function
Purpose	Checks if the function correctly converts a number of seconds into a number of minutes (see Containers/Localization/Utils)
Target	secondsToHoursMinutes() function
Purpose	Checks if the function correctly converts a number of seconds into a formatted hour / minute string (see Containers/Localization/Utils)

Table 25 Utils for List Devices Admin's Scenes

Target	generateInterval() function
Purpose	Checks if the function correctly converts two ISO format dates into a JSON object detailing the interval between them in days, minutes or hours

Table 26 Utils for Patient List Doctor's Scenes

Target	sortByName() function
Purpose	Checks if the function correctly returns 1 or -1 depending on alphabetical sorting of 2 provided patient names (this function is

	used as a comparator for an Array.sort() call)
Target	filterByName() function
Purpose	Checks if the function correctly returns true or false depending on the correspondence between a provided search string and a provided patient (this function is used as a comparator for an Array.filter() call)

Table 27 Utils for Home's Scenes

Target	sortByName() function
Purpose	Checks if the function correctly returns 1 or -1 depending on alphabetical sorting of 2 provided user names (see Scenes/Doctor/PatientsList/Utils)
Target	filterByName() function
Purpose	Checks if the function correctly returns true or false depending on the correspondence between a provided search string and a provided user (see Scenes/Doctor/PatientsList/Utils)

Table 28 Utils for Root

Target	parseName() function
Purpose	Checks if the function accurately converts a name in string format to JSON object with "family" and "given" fields (see Components/CustomListItem/Utils)

5.2.6 Smart Band Server (UPM)

Description

For the gathering and processing of accelerometer and heartrate encrypted raw data it was needed a common server where all the wristbands of the project send this type of data to be filtered and retrieved from other modules to be used for other purposes.

From this server there are available calls to collect decrypted individual packages (last or current) or whole day data.

In order to save the data, it is used a MongoDB and the API calls are supported from a Flask module under Python 3.6

Software Dependences

- Docker
- Docker-compose
- MongoDB

Build – Deployment

Build and push the container images:

```
docker build -t tender_device_api .
docker tag tender_device_api $REGISTRY/tender_device_api:sta
docker push $REGISTRY/tender_device_api:sta
```

Deploy the service:

```
docker-compose down
docker-compose up -d
```

Access endpoints

Stage env:

```
https://fitbit-stage-tender.maggiolicloud.it/
```

Production env:

```
https://fitbit-prod-tender.maggiolicloud.it/
```

CI/CD Pipeline

```
image:
  name: docker/compose:latest
  entrypoint: ["/bin/sh", "-c"]

variables:
  WORK_DIR: ${CI_PROJECT_NAME}
  BRANCH: ${CI_COMMIT_REF_NAME}
  REGISTRY: tender-registry:5000
  SHARED_PATH: /builds/$CI_PROJECT_PATH

before_script:
  - docker --version
  - docker-compose  --version

stages:
  - build
  - test
  - deploy
  - list_services
  - create_p_images
```

```
build_project:
  stage: build
  script:
    - docker build -t tender_device_api .
    - docker tag tender_device_api $REGISTRY/tender_device_api:sta
    - docker push $REGISTRY/tender_device_api:sta
  tags:
    - stage

deploy_project:
  stage: deploy
  script:
   - docker-compose down
   - docker-compose up -d
  tags:
   - stage

list_apps:
  stage: list_services
  script:
    - docker-compose logs
    - docker-compose ps
  tags:
    - stage

tests:
    stage: test
    script:
        - tests/test-endpoint.sh
        - tests/test-mongo-cnt-status.sh
     tags:
     - stage

create_prod_images:
    stage: create_p_images
    script:
      - docker tag $REGISTRY/tender_device_api:sta
$REGISTRY/tender_device_api:prod
      - docker push $REGISTRY/tender_device_api:prod
    when: manual
    tags:
     - stage
```

Tests

All the components and the provided endpoints are tested during the deployment phase.

Test backend API:

Table 29 Smart Band backend test

Test name	Smart Band Backend
Test Purpose	Check the API for smart-band data
Pre-test conditions	Backend API running
Test Tool	Bash Automated Testing System (BATS)
Test description	1. Retrieve data from https://fitbit-stage-tender.maggiolicloud.it/status 2. Check HTTP status code (200 OK)
Test Verdict	API is functional

Command:

```
cd ./tests
./test-endpoint.sh
```

Output:

```
✓ check fitbit-server endpoint

1 tests, 0 failures
```

Test MongoDB:

Table 30 Database test

Test name	**Smart Band Backend**
Test Purpose	Check MongoDB availability
Pre-test conditions	MongoDB running
Test Tool	Bash Automated Testing System (BATS)
Test description	1. Retrieve operational status of mongoDB container. 2. Check if the container is in running state.
Test Verdict	API is functional

Command:

```
cd ./tests
./test-mongo-cnt-status.sh
```

Output:

```
✓ check MongoDB container

1 tests, 0 failures
```

5.2.7 Recommender System

Description

Recommender System module supports the patients profile generation and throw recommendations based in profile and sensors events.

It consists in a flask module that makes queries to EHR to retrieve useful information related with possible recommendations and after a processing and clustering process, results are also posted into EHR to be consumed by mobile and web user interfaces.

Into this service also are included a pseudo-anonymization process to, through a web interface, transform any word of any language of the included in the project into an anonymized code that can be safely stored into the database without compromise personal information. In the same way, this tool also provides the meaning of a given code previously anonymized.

In addition, the tokens of Fitbit users are stored in this service into a database in order to use these tokens to call Fitbit API to get wristbands information.

All these functionalities are performed using a flask module and a PostgreSQL database under Python 3.6

Software Dependences

- Docker
- Docker-compose
- PostgreSQL

Build – Deployment

Build and push docker container images

```
docker build -t tender_device_api .
docker tag tender_device_api $REGISTRY/tender_device_api:sta
docker push $REGISTRY/tender_device_api:sta
```

Instantiate service

```
docker-compose down
docker-compose up -d
```

Access endpoints

Stage env:

```
https://recommender-stage-tender.maggiolicloud.it/
```

Production env:

```
https://recommender-prod-tender.maggiolicloud.it/
```

CI/CD Pipeline

```
image:
  name: docker/compose:latest
  entrypoint: ["/bin/sh","-c"]

variables:
  WORK_DIR: ${CI_PROJECT_NAME}
  BRANCH: ${CI_COMMIT_REF_NAME}
  REGISTRY: tender-registry:5000
  SHARED_PATH: /builds/$CI_PROJECT_PATH

before_script:
  - docker --version
  - docker-compose --version

stages:
  - build
  - test
  - deploy
  - list_services
  - create_p_images

build_project:
  stage: build
```

```
  script:
    #- docker rmi tender_recommender_api
    #- docker rmi --force $REGISTRY/tender_recommender_api:sta
    - docker build -t tender_recommender_api .
    - docker tag tender_recommender_api $REGISTRY/tender_recommender_api:sta
    - docker push $REGISTRY/tender_recommender_api:sta
  tags:
    - stage

deploy_project:
  stage: deploy
  script:
    - docker-compose down
    - docker-compose up -d
  tags:
    - stage

list_apps:
  stage: list_services
  script:
    - docker-compose logs
    - docker-compose ps
  tags:
    - stage

tests:
    stage: test
    script:
        - tests/test-recom-endpoint.sh
        - tests/test-postgres.sh
    tags:
    - stage

create_prod_images:
  stage: create_p_images
  script:
    - docker tag $REGISTRY/tender_recommender_api:sta
$REGISTRY/tender_recommender_api:prod
    - docker push $REGISTRY/tender_recommender_api:prod
  when: manual
  tags:
    - stage
```

Tests

All the components and the provided endpoints are tested during the deployment phase in the stage environment.

Table 31 Recommender HTTP API test

Test name	Recommender Backend
Test Purpose	Check the API for smart-band data
Pre-test conditions	Backend API running
Test Tool	Bash Automated Testing System (BATS)
Test description	1. Retrieve data from https://recommender-stage-tender.maggiolicloud.it/status 2. Check HTTP status code (200 OK)
Test Verdict	API is functional

Command:

```
cd ./tests
./test-recom-endpoint.sh
```

Output:

```
✓ check Recommender endpoint

1 tests, 0 failures
```

Table 32 PostgreSQL DB test

Test name	**Recommended DB**
Test Purpose	Check MongoDB availability
Pre-test conditions	MongoDB running
Test Tool	Bash Automated Testing System (BATS)
Test description	1. Retrieve operational status of PostgreSQL containers. 2. Check if the container is in running state
Test Verdict	API is functional

Command:

```
cd ./tests
./ test-postgres.sh
```

Output:

```
✓ check postgress container

1 tests, 0 failures
```

5.2.8 Questionary Server

Description

The questionnaire server is a platform that needs to support the creation and filling of questionnaires and, at the same time, to have a structure capable of storing data publicly and privately. Since TeNDER will provide its services to several organizations, this server has to provide a solution where users can be associated with an organization and access management through roles and privacy measures.

To fill these requirements, it instantiated an open-source Data Management System (DMS) named CKAN [24].

CKAN is a powerful data management system that makes data accessible by providing tools to streamline publishing, sharing, finding, and using data.

By making open data websites, CKAN is capable of providing pretty good management and publishing collections of information. It's used by national and local governments, research institutions, and other organizations that collect a lot of diverse data, which reinforces its efficiency and usability.

It is open-source software with a good number of active contributors, which gives greater security in terms of support and constant improvement of the platform. Additionally, CKAN can be changed and extended with the inclusion of one or more CKAN extensions.

Since CKAN does not provide the creation and filling of questionnaires, the best approach was the creation of an extension to fulfil this purpose. It's an objective, efficient, and usable CKAN extension where users can create and fill questionnaires and manage the gathered data. Joining the CKAN's necessary tools and functionalities for the easy and correct management of data (open or not) with this extension, the platform will manage a new way of gathering information. Additionally, the questionnaires' responses will be stored in specific datasets to posteriorly be sent to the HAPI FHIR server to centralize all data and provide it to the web and mobile applications.

Software Dependences

- CKAN 2.8 Docker Image (okfn/docker-ckan)
- Docker
- Python 2.7
- Browser with Javascript support

Build – Deployment

Build the container images

```
docker build --no-cache -t tender-ckan-ubw . --build-arg TZ=UTC
```

Instantiate the service

```
docker-compose up -d
```

Access endpoint

```
https://qst-prod-tender.maggiolicloud.it/
```

CI/CD Pipeline

```
image:
  name: docker/compose:1.21.2
  entrypoint: ["/bin/sh", "-c"]
variables:
  GIT_STRATEGY: clone
  WORK_DIR: ${CI_PROJECT_NAME}
  BRANCH: ${CI_COMMIT_REF_NAME}
  REGISTRY: tender-registry:5000
  SHARED_PATH: /builds/$CI_PROJECT_PATH
stages:
  - build
  - tests
  - deploy
  - list_services
  - create_p_images
build_images:
    stage: build
    script:
        - docker build --no-cache -t tender-ckan-ubw . --build-arg TZ=UTC
        - docker tag tender-ckan-ubw $REGISTRY/tender-ckan-ubw:sta
```

```
        - docker push $REGISTRY/tender-ckan-ubw:sta
    tags:
     - stage
deploy_ckan:
    stage: deploy
    script:
        - docker-compose down
        - docker-compose up -d
    tags:
     - stage
list_apps:
    stage: list_services
    script:
        - docker-compose logs
        - docker-compose ps
    tags:
     - stage
tests:
    stage: tests
    script:
        - tests/ckan_execution_status.sh
        - tests/ckan-test.sh
    tags:
    - stage
create_prod_images:
    stage: create_p_images
    script:
      - docker tag $REGISTRY/tender-ckan-ubw:sta $REGISTRY/tender-ckan-ubw:prod
      - docker push $REGISTRY/tender-ckan-ubw:prod
    when: manual
    tags:
     - stage
```

Tests

Table 33 - Test CKAN Execution.

Test name	**Application server connectivity**
Test Purpose	Check the execution of the CKAN instance and if the new extension were correctly installed.
Pre-test conditions	The CKAN running in a local or cloud environment with all the extensions installed.
Test Tool	Shell Script (sh)
Test description	1. In case of CKAN staging environment, perform a curl command to the IP where the CKAN is instantiated (`curl -I https://qst-stage-tender.maggiolicloud.it`) 2. Perform request to CKAN to know which extensions are installed and running.
Test Verdict	CKAN is running and working properly

Command:

```
bash ckan_execution_status.sh
```

Output:

```
HTTP/2 200
```

```
cache-control: private
content-type: text/html; charset=utf-8
set-cookie:
ckan=b89d69c9399f9bca321208b495d4463e8c02dd2fa827fe076c564e7e923e76c60b11400c;
Path=/
content-length: 12958
date: Thu, 12 Aug 2021 13:36:02 GMT
```

Table 34 - Test CKAN Questionnaires Extension

Test name	**Application server connectivity**
Test Purpose	Check if the questionnaires extension is working properly
Pre-test conditions	The CKAN running in a local or cloud environment with all the extensions installed.
Test Tool	Shell Script (sh)
Test description	1. Perform request to create a questionnaire; 2. Perform request to fill a questionnaire; 3. Perform request to get the questionnaire.
Test Verdict	CKAN questionnaire extension is working properly

Command:

```
bash ckan-test.sh
```

Output:

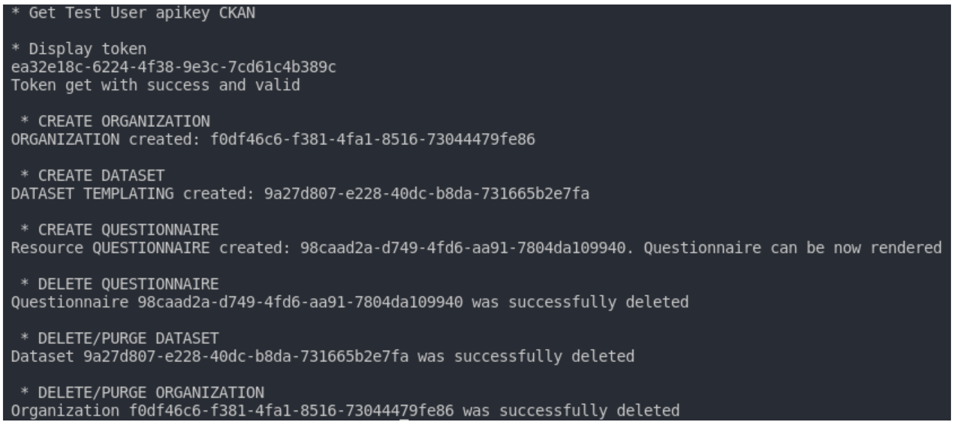

Figure 47 CKAN test results

5.3 Hybrid Mobile application

Description:

The TeNDER Hybrid Mobile application was designed to help people affected by specific health and mental diseases. In fact, the UI is simple and intuitive to facilitate the users' navigation and is compliant with specific accessibility standards. Currently, the application is organised on four different sections: Services, Home, Messages and Suggestions (Figure 48) and supports three services:

- "Health", collects statistics about patients' health conditions, for instance heart rate or blood pressure.
- "Reminders" and provides a calendar where the users can manage their activities, events and appointments.
- "Sleep diary", which shows statistics and monthly or weekly reports about the user's sleep quality.

Moreover, the application interfaces were designed to meet the criteria for three end-user groups:

- Patients;
- Caregivers;
- Professionals

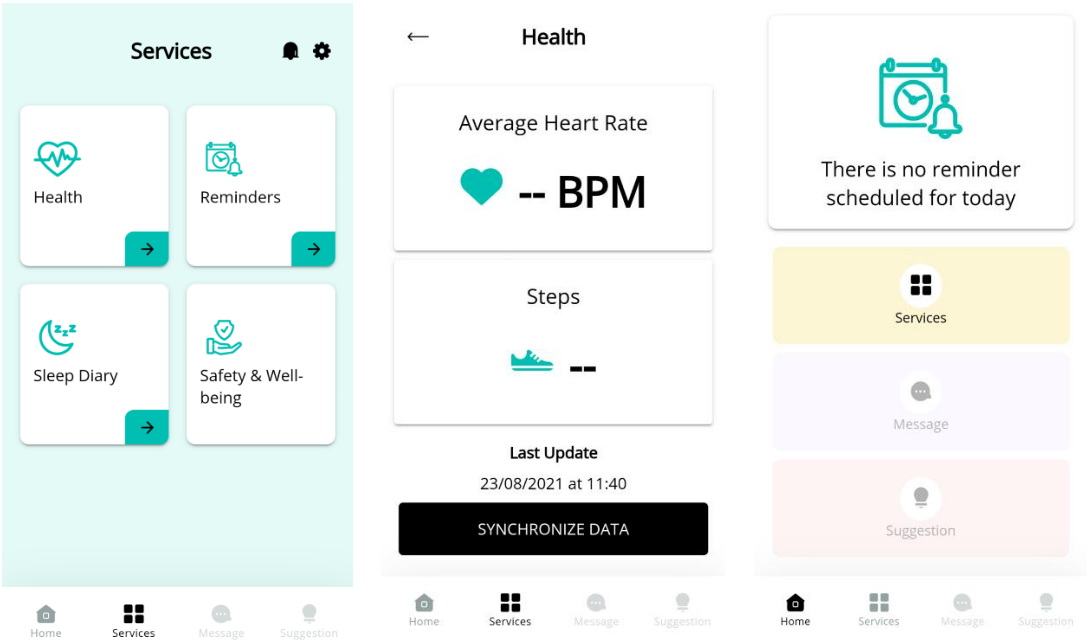

Figure 48 TeNDER's App Sections

Each user can access a profile area where they can modify their information and, in the case of patients, they can specify their doctor or caregiver, who can monitor the patient using the same app but a different type of user.

The TeNDER application follows an iterative process to be implemented from the beginning to the end (more details in D5.1). At a very first stage, the tech team proposed a set of services and wireframes to explain to the end users' the main ideas to put into practice and show the potential of the Tender App. Later on, the collection of user requirements was split into two phases: Pre-pilot requirements gathering and post-pilot requirements gathering: the first aimed to obtain the first impressions and ideas of the end-users without having a prototype. The second is focused on the analysis of feedback provided by end-users after the first pilot execution. Those phases are covered and well explained in paragraph 2 of the Deliverable D5.1 Report on TeNDER interfaces.

Software Dependences

Technologies involved in the Tender App development are:

- Google's Firebase
- Ionic 6 (Typescript and HTML/CSS UI side)
- Android OS 8+
- iOS (in progress)

Installation

The TeNDER mobile application currently supports Android OS 8+ (later on for iOS), and is available in following link:

https://drive.google.com/drive/folders/12UbYnozsMDChx3AvkeN8QD3GJEUToFp5

Tests

Given the ever-increasing complexity of the TeNDER app, manual and automated tests were designed to manage quality control, tests were split into manual and automatic tests.

The manual testing by technical involves verification on features like resolution of the display (the quality density or color brightness of the display components), space disposition and frame/bottoms adaptability and layout structure in different devices. Moreover, the functionalities and UI features were tested by end users, before and during the pilot execution.

Automated testing focuses on verifying the correct application functionality, because every time a piece of source code is modified, the overall application needs to be tested again. With manual testing, it is not feasible to test the application in a holistic way. The first step to start automating the process of testing was to describe what the system does: From the identification of functions that software is expected to perform, a creation of input data and output based on the specification, the actual test case execution and comparison of actual and expected outputs.

TeNDER application is the result of a cooperative work where many functionalities are provided mostly via API, each partner is responsible for making sure their modules or services are functioning correctly (correct output, reasonable response times, etc). Thus, developers involved in the implementation of the UI perform a simple verification during the back-end and front-end integration.

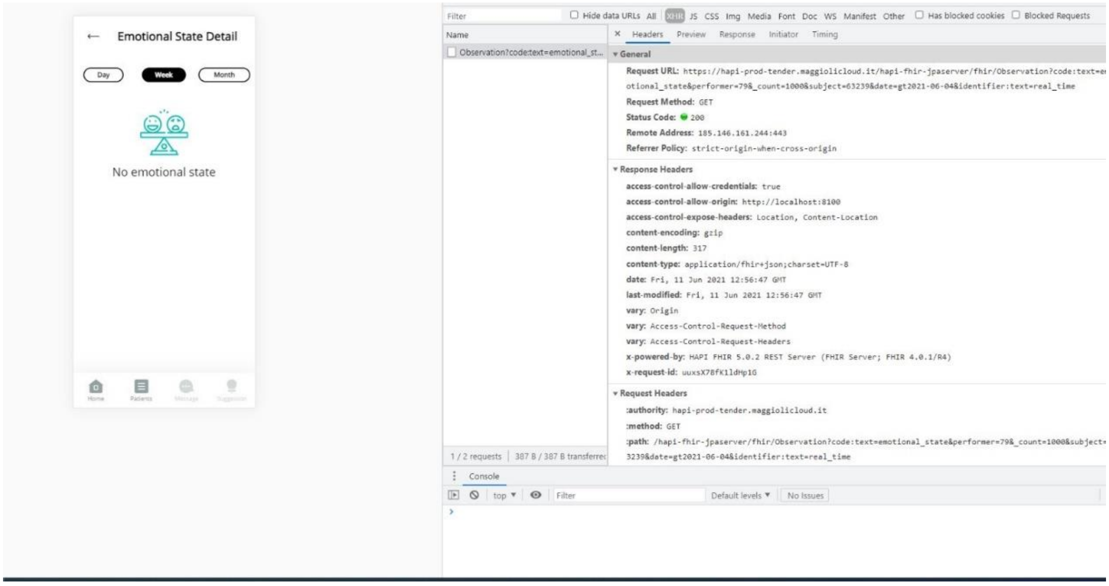

Figure 49 An example of API response form console

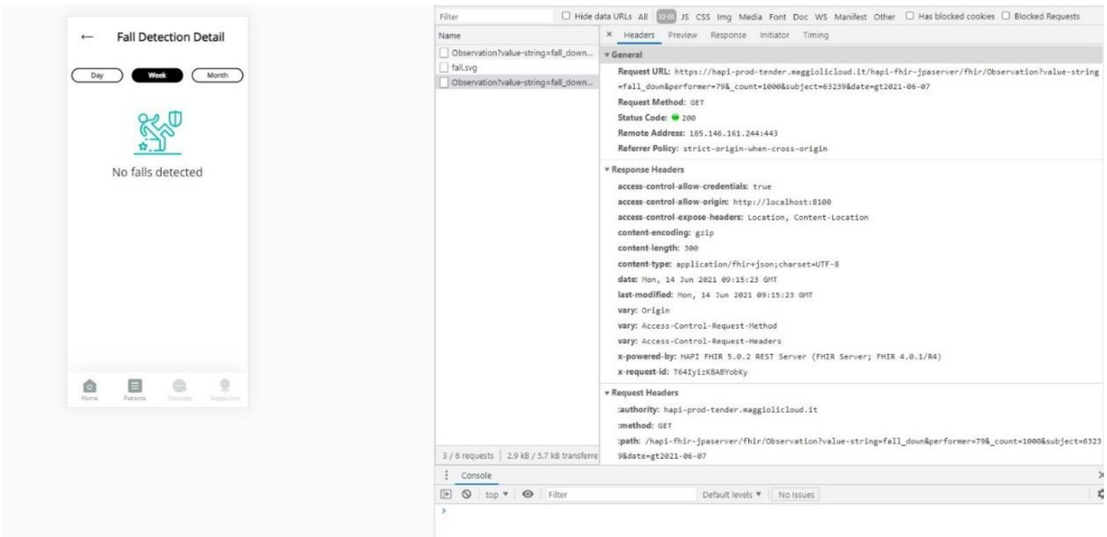

Figure 50 An example of API response form Console

The “Reminder” functionality does not refer to an external module or API so for that implementation unit test has been performed. The following images show their results.

Test Case ID	BU_001	**Test Case Description**	Test the trigger of new reminders		
Created By	Marco Di Gioia	**Reviewed by**	Marco Di Gioia	**Version**	1

QA Tester's Log Check if a new reminder has been created

Tester's Name	Giuseppe Di Puglia	**Date Tested**	June 13, 2021	**Test Case**	Pass

S #	Prerequisites:
1	
2	
3	
4	

S #	Test Data
1	Reminder Module: PatientID = 26303
2	App: username = caregiver1907@test.com
3	App: password = test1234
4	

Test Scei Verify on task

Step #	Step Details	Expected Results	Actual Results	Pass / Fail / Not executed / Suspended
1	Create a new reminder for Patient 26303	See a new reminder on mobile app	As Expected	Pass
2	Query get on SQLite	Get 0 row	As Expected	Pass
3	Run unittest for task	No error	As Expected	Pass
4	Query get on SQLite	Get 1 row	As Expected	Pass

Figure 51 Unit test to check a new reminder creation

Test Case ID	BU_002	**Test Case Description**	Test the trigger of changed reminder		
Created By	Marco Di Gioia	**Reviewed by**	Marco Di Gioia	**Version**	1

QA Tester's Log Check if a reminder has been changed

Tester's Name	Giuseppe Di Puglia	**Date Tested**	June 13, 2021	**Test Case**	Pass

S #	Prerequisites:
1	BU_001

S #	Test Data
1	Reminder Module: PatientID = 26303

Test Scenario Verify on task

Step #	Step Details	Expected Results	Actual Results	Pass / Fail / Not executed / Suspended
1	Query get on SQLite	There is at least 1 row	As Expected	Pass
2	Run unittest for task	No error	As Expected	Pass
3	Query get on SQLite	last_modify_at field is	As Expected	Pass

Figure 52 Unit test to check an updated of an event in the Reminder section

Test Case ID	BU_003	**Test Case Description**	Test the assistant vocal reminder feature		
Created By	Marco Di Gioia	**Reviewed by**	Marco Di Gioia	**Version**	1

QA Tester's Log Speak Phrase Notice

Tester's Name	Giuseppe Di Puglia	**Date Tested**	June 13, 2021	**Test Case**	Pass

S #	Prerequisites:
1	BU_001

S #	Test Data
1	Reminder Module: PatientID = 26303

Test Scenario Verify on task

Step #	Step Details	Expected Results	Actual Results	Pass / Fail / Not executed / Suspended
1	Run unittest for task	No error	As Expected	Pass
2	Listen Google TTS voice	A spoken phrase can be heard	As Expected	Pass

Figure 53 Unit test to check vocal reminder

est Case ID	BU_004	Test Case Description	Test the assistant vocal reminder feature		
reated By	Marco Di Gioia	Reviewed by	Marco Di Gioia	Version	1

A Tester's Log	Speak Phrase Event

ester's Name	Giuseppe Di Puglia	Date Tested	June 13, 2021	Test Case	Pass

S #	Prerequisites:
1	BU_001
2	
3	
4	

S #	Test Data
1	Reminder Module: PatientID = 26303
2	
3	
4	

est Scenario	Verify on task

Step #	Step Details	Expected Results	Actual Results	Pass / Fail / Not executed /
1	Run unittest for task speak_phrase	No error	As Expected	Pass
2	Listen Google TTS voice	A spoken phrase can be	As Expected	Pass
3	Query get on SQLite	First row has state=3	As Expected	Pass

Figure 54 Unit test to check speak phrase event

Test Case ID	BU_002	Test Case Description	Test the deletion of a reminder		
Created By	Marco Di Gioia	Reviewed by	Marco Di Gioia	Version	1

QA Tester's Log	Check if a reminder has been deleted

Tester's Name	Giuseppe Di Puglia	Date Tested	June 13, 2021	Test Case	Pass

S #	Prerequisites:
1	BU_001

S #	Test Data
1	Reminder Module: PatientID = 26303

Test Scenario	Verify on task

Step #	Step Details	Expected Results	Actual Results	Pass / Fail / Not executed /
1	Query get on SQLite	There is at least 1 row	As Expected	Pass
2	Run unittest for task	No error	As Expected	Pass
3	Query get on SQLite	state field is -1	As Expected	Pass

Figure 55 Unit test to check reminder deletion

6 APPLICATION PROGRAMABLE INTERFACES

TeNDER platform provides APIs for interconnection with external EHR systems and the communication between the internal services of the ecosystem. The APIs that are accessible via the public network are integrated with the authorization and authentication server and support secure connection over HTTPS protocol. On the other hand, the internal APIs are accessible only via the private internal network in TeNDER cloud infrastructure, but they can be offered to outside services if needed in the future, following the same approach. At this point of the development, the APIs that are public are the ones from the HAPI-FHIR server and the internal ones are (a) the Remote DB, (b) the Smart Band and c.

To enhance the interconnection with the external EHR systems and the internal services the consortium decided to create a special documentation server that provides the appropriate documentation for all the available RESTful APIs of the platform. The TenDER's documentation is available on the URL:

```
https://docs-stage-tender.maggiolicloud.it/
```

and it supports the following specifications:

- **OpenAPI** [25] is an API description format for REST APIs. An OpenAPI file describes the entire API, including: (a) the available endpoints (/users) and operations on each endpoint (GET /users, POST /users); (b) operation parameters Input and output for

each operation; (c) authentication methods; (d) contact information; (e) license, terms of use etc.
The API specifications can be written in YAML or JSON. The format is easy to learn and readable to both humans and machines.

- **Swagger** is a set of open-source tools built around the OpenAPI Specification and it can be used for design, build, document, and consume REST APIs. The major Swagger tools include: (a) swagger Editor [26], which is a browser-based editor where you can write OpenAPI specs; (b) swagger UI [27], which renders OpenAPI specs as interactive API documentation and (c) swagger Codegen [28], which generates server stubs and client libraries from an OpenAPI spec.

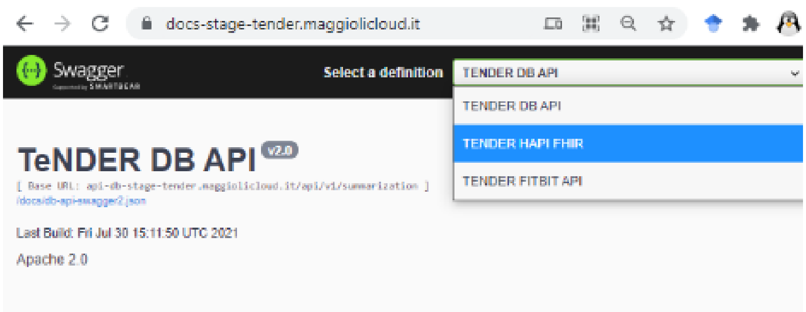

Figure 56 TeNDER's documentation server

6.1 EHR API (HL7) (UBI)

The HAPI FHIR instance provides a Java API for HL7 FHIR Clients and Servers [29]. This means that the API follows the structure and rules of HL7 resources, which garantees the correct requests and workflow between end-users and the platform.

In the following image (Figure 57) the EHR API's Swagger can be partially visualize, which is already deployed in

https://docs-stage-tender.maggiolicloud.it/?urls.primaryName=TENDER%20HAPI%20FHIR. All available requests are listed and ready to be tested, having examples to facilitate the developer understading.

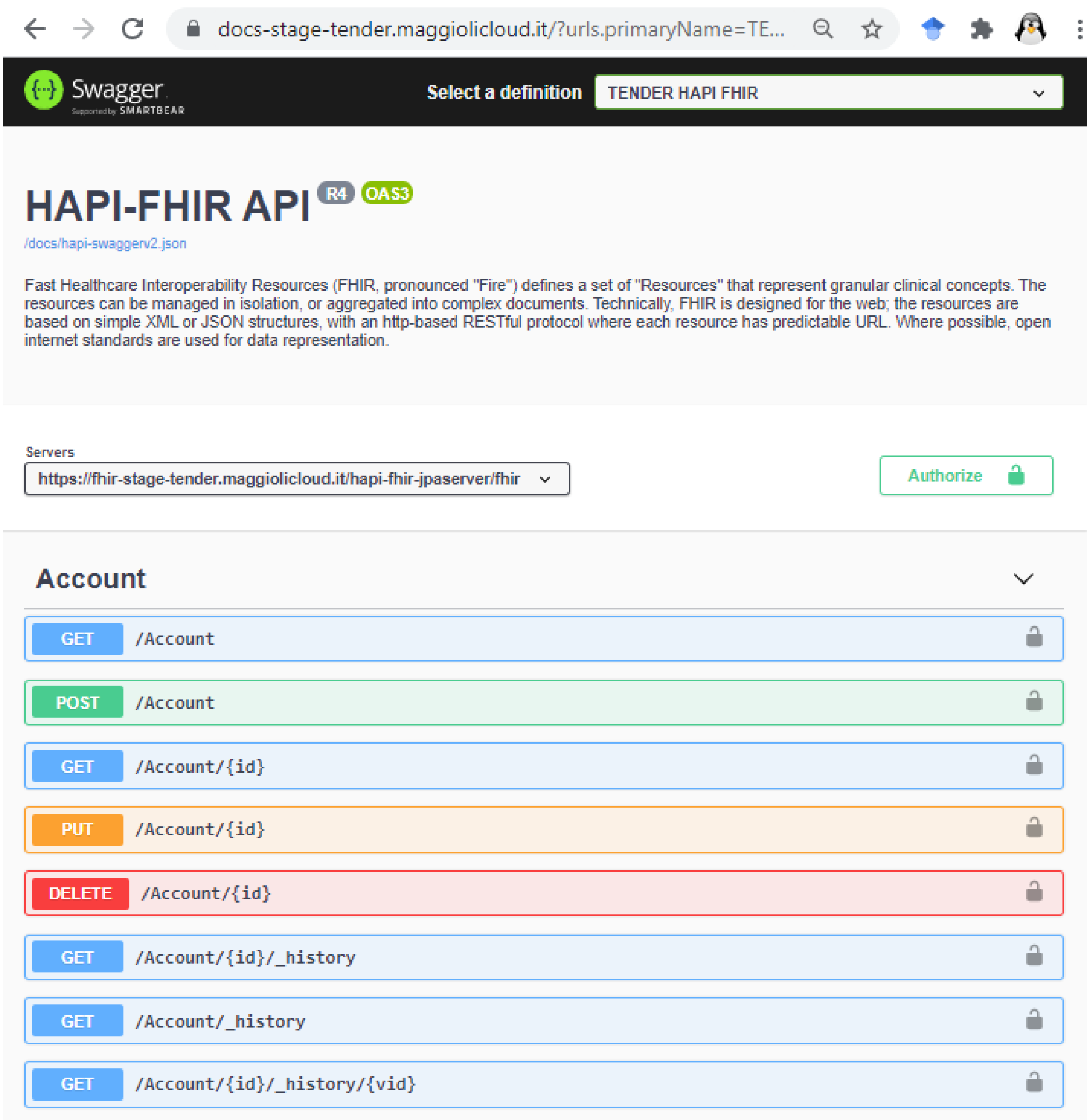

Figure 57 API documentation of the HAPI FHIR server

At the bottom of the page, the schema of each available resource is exposed. Through Figure 58, is possible to verify the 'Account' resource, providing: all its attributes; if they are mandatory; its type, and overall structure. Once again, this information is crucial for the efficient and fast development of components that will communicate with the server.

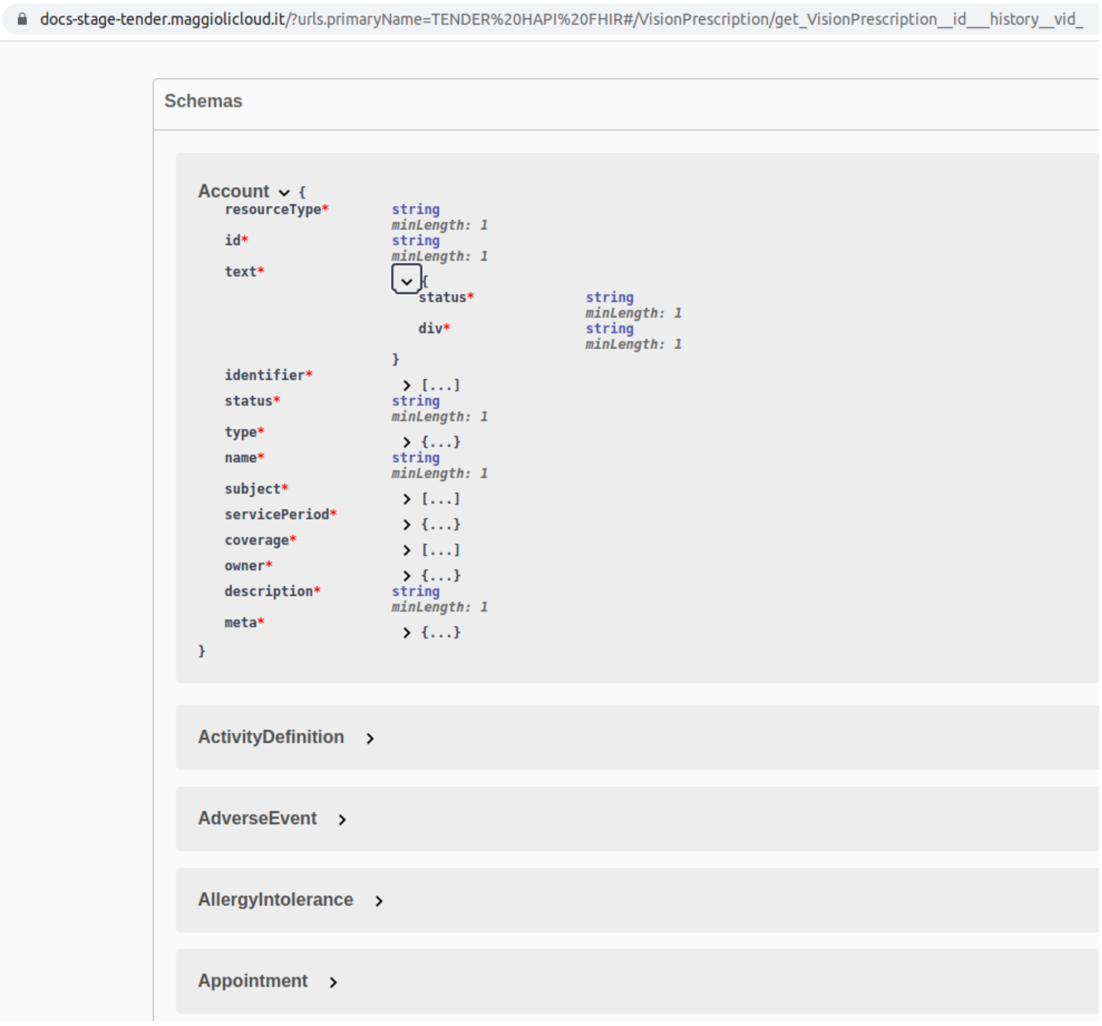

Figure 58 HAPI FHIR server API schemas

For a better view and understanding of the schemas and their attributes, the following link (https://www.hl7.org/fhir/resourcelist.html) lists all the resources where each resource can be accessed, providing a better explanation. The following image (Figure 59) provides an example of the 'Patient' resource's structure in JSON format.

```
{
  "resourceType" : "Patient",
  // from Resource: id, meta, implicitRules, and language
  // from DomainResource: text, contained, extension, and modifierExtension
  "identifier" : [{ Identifier }], // An identifier for this patient
  "active" : <boolean>, // Whether this patient's record is in active use
  "name" : [{ HumanName }], // A name associated with the patient
  "telecom" : [{ ContactPoint }], // A contact detail for the individual
  "gender" : "<code>", // male | female | other | unknown
  "birthDate" : "<date>", // The date of birth for the individual
  // deceased[x]: Indicates if the individual is deceased or not. One of these 2:
  "deceasedBoolean" : <boolean>,
  "deceasedDateTime" : "<dateTime>",
  "address" : [{ Address }], // An address for the individual
  "maritalStatus" : { CodeableConcept }, // Marital (civil) status of a patient
  // multipleBirth[x]: Whether patient is part of a multiple birth. One of these 2:
  "multipleBirthBoolean" : <boolean>,
  "multipleBirthInteger" : <integer>,
  "photo" : [{ Attachment }], // Image of the patient
  "contact" : [{ // A contact party (e.g. guardian, partner, friend) for the patient
    "relationship" : [{ CodeableConcept }], // The kind of relationship
    "name" : { HumanName }, // A name associated with the contact person
    "telecom" : [{ ContactPoint }], // A contact detail for the person
    "address" : { Address }, // Address for the contact person
    "gender" : "<code>", // male | female | other | unknown
    "organization" : { Reference(Organization) }, // C? Organization that is associated with the
contact
    "period" : { Period } // The period during which this contact person or organization is vali
d to be contacted relating to this patient
  }],
  "communication" : [{ // A language which may be used to communicate with the patient about his
or her health
    "language" : { CodeableConcept }, // R!  The language which can be used to communicate with
the patient about his or her health
    "preferred" : <boolean> // Language preference indicator
  }],
  "generalPractitioner" : [{ Reference(Organization|Practitioner|
   PractitionerRole) }], // Patient's nominated primary care provider
  "managingOrganization" : { Reference(Organization) }, // Organization that is the custodian of
the patient record
  "link" : [{ // Link to another patient resource that concerns the same actual person
    "other" : { Reference(Patient|RelatedPerson) }, // R!  The other patient or related person r
esource that the link refers to
    "type" : "<code>" // R!  replaced-by | replaces | refer | seealso
  }]
}
```

Figure 59 Patient resource structure
(from: https://www.hl7.org/fhir/patient.html)

Still in this page, the filters, relationships and attribute's type explanation can be found or easily redirected to the correct page.

For the TeNDER case, it was used a Postman collection [30] to reinforce and improve the documentation and facilitate the API testing. The collection contains the main requests with several examples, grouped by resource. The main purpose to use it was the tool to define different environments, which helps the developers to switch between environments in seconds. Important to mention that the collection is in continuous improvement and the new version are always provided to avoid deprecated requests or examples.

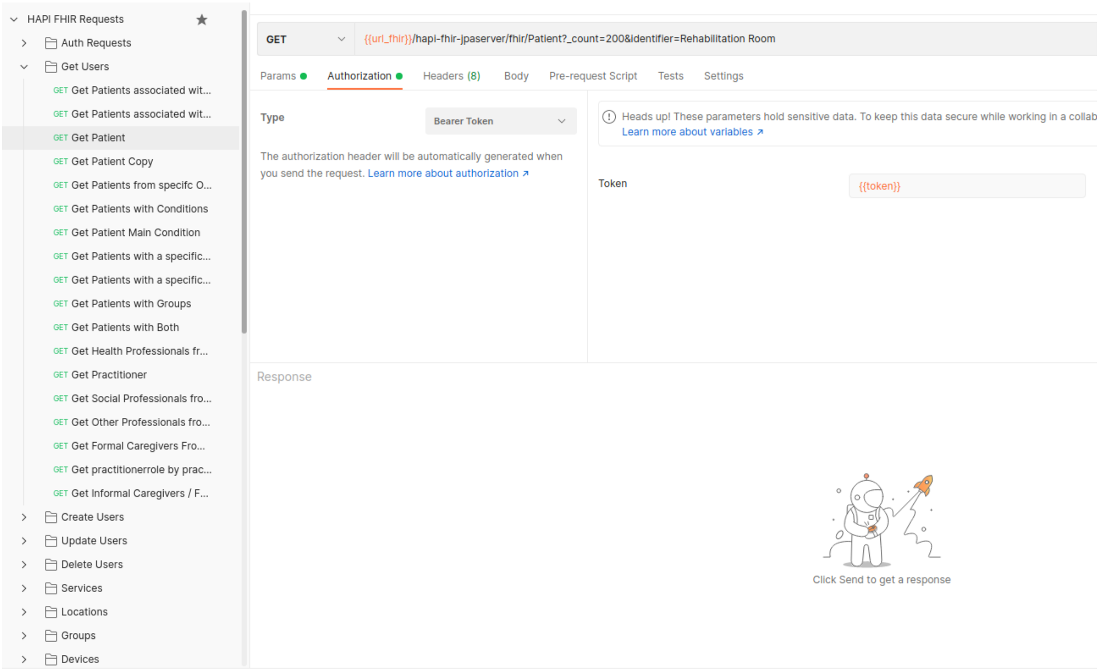

Figure 60 Part of Postman collection

6.2 Remote DB Rest APIs

TeNDER Mongo DB API provides the entry point for data coming from devices and sensors to the TeNDER high-level subsystem. The main purpose of this implementation is to provide a unified access to remote MongoDB of the TeNDER using HTTP RESTful API. This approach is very useful because every service can access the database without any specific dependencies (i.e. software libraries/plugins etc). The API is provided on two endpoints, over authenticated https for external access and over http for access by TeNDER services from the internal network.

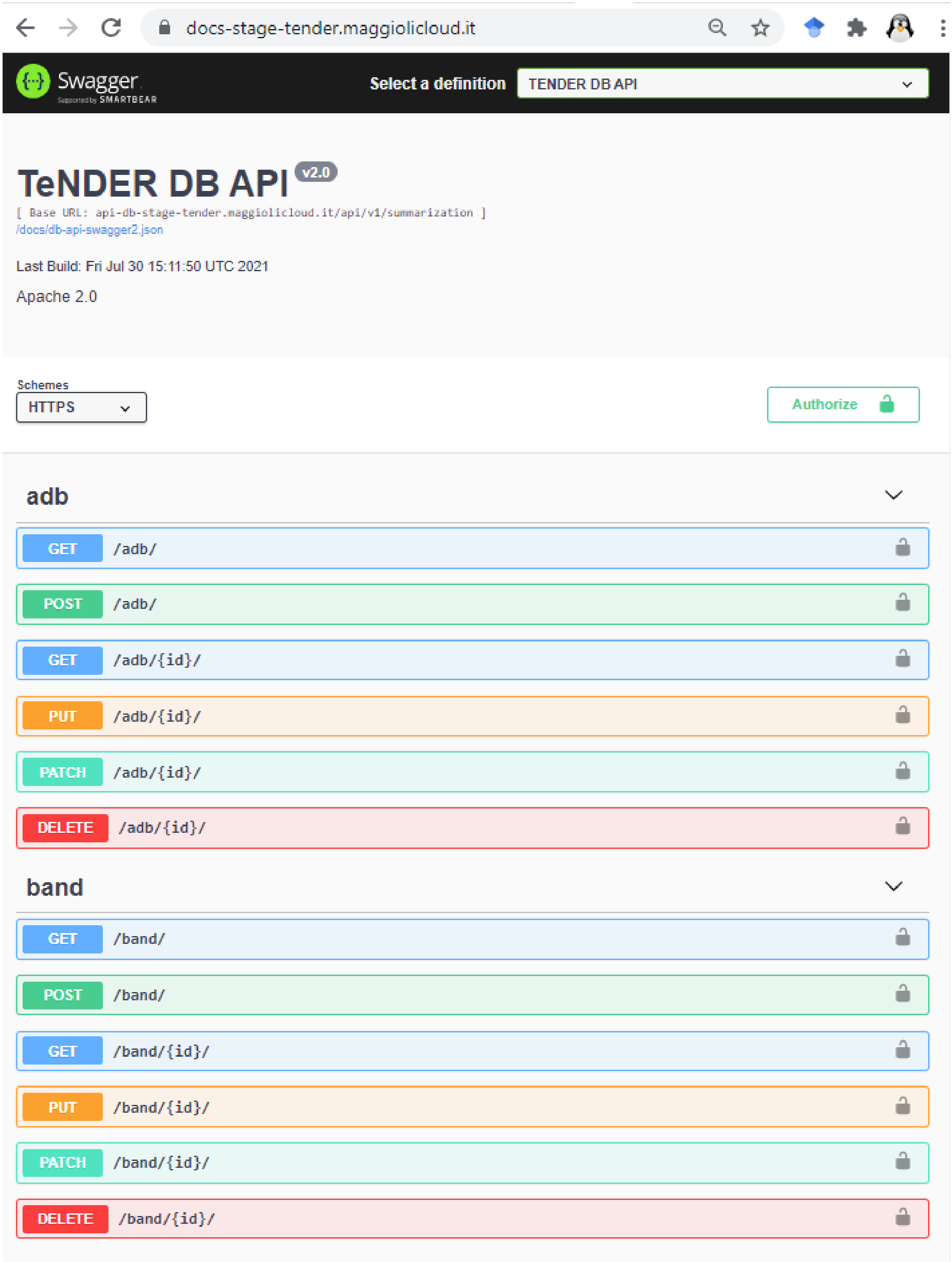

Figure 61 API documentation of the Remote Mongo DB

6.3 Smart Band APIs

Health Wearable REST API manages all the access of data from bands into the project database with the corresponding needed pre-process, as the encrypting. The main purpose of this API is to offer a good interface between bands and database and between data and other modules of the project which need consume this data as well. This API is structured with one POST endpoint to save data and other four GET endpoints to receive this data considering different filtering approaches.

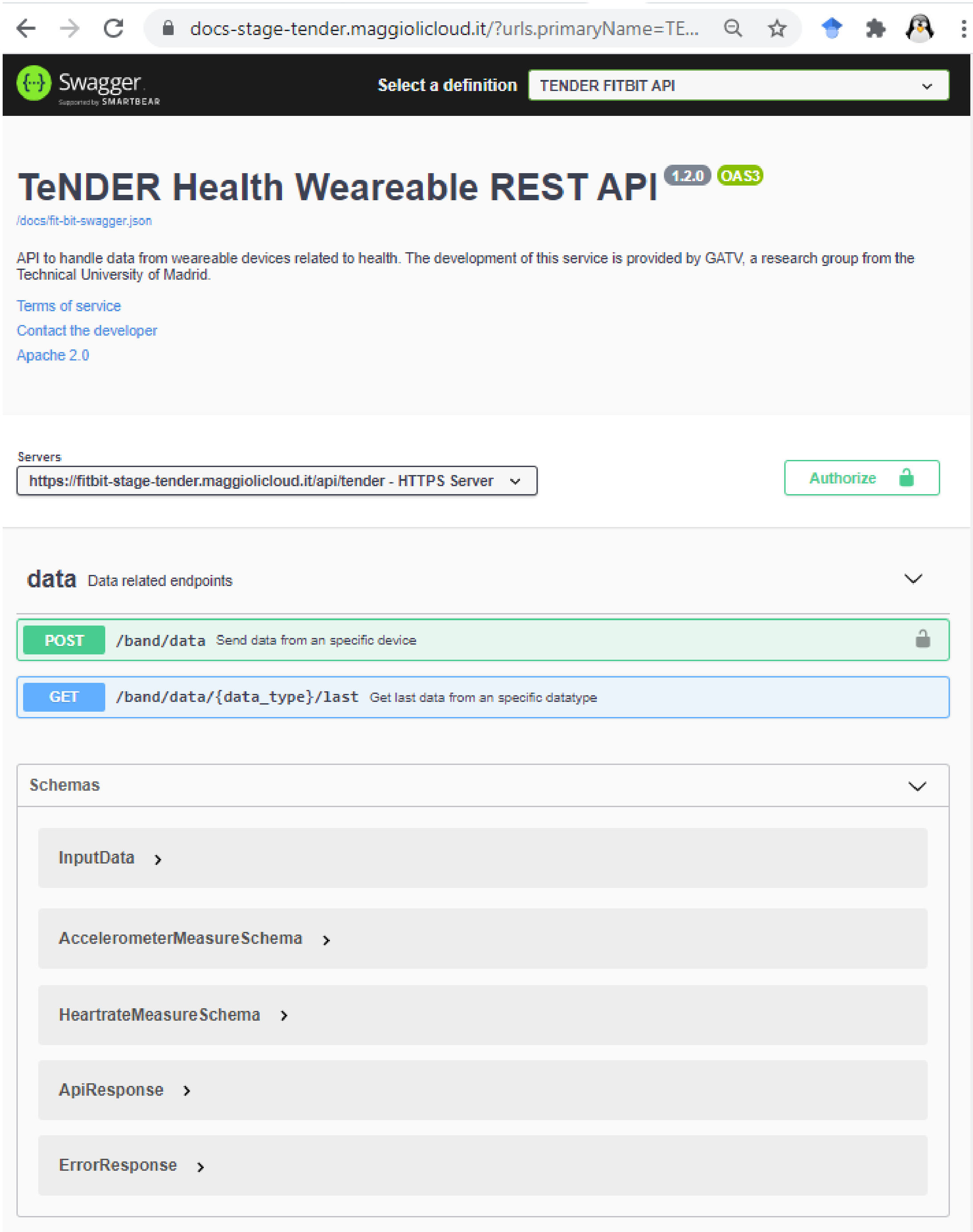

Figure 62 API documentation of the Smart Band API

7 CONCLUSIONS

In this deliverable, we have described the first version of the TeNDER platform as it is used in the first wave of pilots. The document presents the tools and methodologies used to drive the software development in a CI/CD approach, in which the TeNDER development cycle is based. This has been used to facilitate the development efforts in WP3 and WP4, providing the tools and methodologies to embrace this development philosophy. Moreover, the use of software management and automated continuous integration tools (i.e. GitLab, pipelines etc) allowed the developers to integrate the outcome of the work in an agile way, continuously pushing improvements and integrating them progressively. This approach has allowed us to avoid the likely risk of needing a complex and long phase of integration at the end of the development process, too late to ensure the suitable level of software quality. Furthermore, we designed and deployed an open-source monitoring system to collect information regarding the resource allocation from all the deployment environments and for all services. Finally, we listed and detailed the first version of the integration and qualification tests of the platform, which have been designed and developed to ensure the functionalities expected for the first version. As the development process continues, new services and applications will be added, so during the remaining time of the project we will focus on the improvement of the current integration and qualification testing procedures and on the design of new ones. The last deliverable of the WP5 (D5.5) will present the final version of the TeNDER platform with all its components and the tests.

REFERENCES

[1] "Cycle Time " en.wiktionary.org. https://en.wiktionary.org/wiki/cycle_time (accessed Aug. 23, 2021).

[2] P. Webteam, "Continuous Delivery Vs. Continuous Deployment" puppet.com. https://tinyurl.com/zsoenks (accessed Aug. 23, 2021).

[3] "Continuous Integration," *martinfowler.com*. https://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration (accessed Aug. 23, 2021).

[4] "*TeNDER GitLab Repository*" *TeNDER GitLab*. *http://tendergitlab.maggiolicloud.it/*. (accessed Aug. 23, 2021).

[5] "*TeNDER production deployment*" *TeNDER GitLab*. *https://tendergitlab.maggiolicloud.it/panos_k/tnd-production* (accessed Aug. 23, 2021).

[6] W. Project, "Watir Project," *watir.com*. http://watir.com/. https://robotframework.org (accessed Aug. 23, 2021).

[7] "Robot Framework," *robotframework.org*. https://robotframework.org (accessed Aug. 23, 2021).

[8] "pytest: helps you write better programs — pytest documentation," *pytest.org*. https://pytest.org/en/latest/ (accessed Aug. 23, 2021).

[9] "Welcome to bats-core's documentation" *bats-core.readthedocs.io*. https://bats-core.readthedocs.io/en/stable/index.html (accessed Aug. 23, 2021).

[10] "Apache JMeter" *Apache.org*, 2019. https://jmeter.apache.org/. (accessed Aug. 23, 2021).

[11] "ab - Apache HTTP server benchmarking tool," *Apache.org*, 2019. https://httpd.apache.org/docs/2.4/programs/ab.html. (accessed Aug. 23, 2021).

[12] "*TeNDER integration tests*" *TeNDER GitLab*. *http://tendergitlab.maggiolicloud.it/panos_k/tnd-int-tests*. (accessed Aug. 23, 2021).

[13] "RabbitMQ PerfTest," *rabbitmq.github.io*. https://rabbitmq.github.io/rabbitmq-perf-test/stable/htmlsingle/ (accessed Aug. 23, 2021).

[14] "GitHub - prometheus/prometheus: The Prometheus monitoring system and time series database.," *GitHub*. https://github.com/prometheus/prometheus (accessed Aug. 23, 2021).

[15]"GitHub - prometheus/pushgateway: Push acceptor for ephemeral and batch jobs.," *GitHub*. https://github.com/prometheus/pushgateway (accessed Aug. 23, 2021).

[16]"GitHub - prometheus/alertmanager: Prometheus Alertmanager," *GitHub*. https://github.com/prometheus/alertmanager (accessed Aug. 23, 2021).

[17]"GitHub - grafana/grafana," *GitHub*". https://github.com/grafana/grafana. (accessed Aug. 23, 2021).

[18]"GitHub - netdata/netdata: Real-time performance monitoring" *GitHub*. https://github.com/netdata/netdata (accessed Aug. 23, 2021).

[19]"GitHub - google/cadvisor," *GitHub*. https://github.com/google/cadvisor (accessed Aug. 23, 2021).

[20]"Traefik Labs: Makes Networking Boring," *Traefik Labs: Makes Networking Boring*. https://traefik.io/ (accessed Aug. 23, 2021).

[21]"Keycloak," *www.keycloak.org*. https://www.keycloak.org/ (accessed Aug. 23, 2021).

[22]"Http - FHIR v4.0.1," *www.hl7.org*. https://www.hl7.org/fhir/http.html (accessed Aug. 23, 2021).

[23]"Jest - Delightful JavaScript Testing," *Jestjs.io*, 2017. https://jestjs.io/ (accessed Aug. 23, 2021).

[24]"CKAN - The open source data management system," *ckan.org*. https://ckan.org/ (accessed Aug. 23, 2021).

[25]"*GitHub* - OpenAPI-Specification/3.0.2.md at main · OAI/OpenAPI-Specification," *GitHub*. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md (accessed Aug. 23, 2021).

[26]"Swagger Editor," *editor.swagger.io*. http://editor.swagger.io/?_ga=2.93506717.1281844120.1627895131-200039773.1627656792 (accessed Aug. 23, 2021).

[27]"API Code & Client Generator | Swagger Codegen," *Swagger.io*, 2021. https://swagger.io/swagger-codegen/ (accessed Aug. 23, 2021).

[28]"REST API Documentation Tool | Swagger UI," *Swagger.io*, 2021. https://swagger.io/swagger-ui/ (accessed Aug. 23, 2021).

[29]"GitHub - hapifhir/hapi-fhir: HAPI FHIR - Java API for HL7 FHIR Clients and Servers," *GitHub*. https://github.com/hapifhir/hapi-fhir (accessed Aug. 23, 2021).

[30]Postman, “Postman | The Collaboration Platform for API Development,” *Postman*, 2021. https://www.postman.com/.