INTEGRATED PROASENSE PLATFORM AND ITS ASSESSMENT V2

Deliverable nº: D6.3

EC-GA Number:	612329
Project full title:	The Proactive Sensing Enterprise

		Work Package:	**WP6**
		Type of document:	**Deliverable**
		Date:	**05/10s/2016**
Grant Agreement No 612329			
Partners:	SINTEF, FZI, ICCS, JSI, UNINOVA, NISSA		
Responsible:	**Aleksandar Stojadinovic**		
Title:	**D6.3. Integrated ProaSense platform and its assessment v2**	Version: 1	**Page: 0 / 22**

Deliverable D6.3 Integrated ProaSense platform and its assessment v2

DUE DELIVERY DATE: M35

ACTUAL DELIVERY DATE: OCTOBER 2016 (M35)

Document History

Vers.	Issue Date	Content and changes	Author
0.1	21.09.2016	TOC	Aleksandar Stojadinovic Dominik Reimer
0.2	27.09.2016	Integration/deployment content	Aleksandar Stojadinovic
0.3	30.09.2016	UI content	Dominik Reimer
0.4	05.10.2016	Integrated version	Aleksandar Stojadinovic
1.0	13.10.2016	Reviewed integrated version	Aleksandar Stojadinovic

Document Authors

Partners	Contributors
NISSA	Aleksandar Stojadinovic
FZI	Dominik Reimer

Dissemination level: Public

Document Approvers

Partners	Approvers
UNINOVA	Ana Rita Campos
SINTEF	Brian Elvesæter

Executive Summary

This deliverable elaborates on the conclusions drawn from the first integration and deployment cycle in three aspects:

- The integration approach itself
- Packaging and distribution
- User interface

All of them are assessed separately, mentioning successful solutions we used, but also underlining points where the system proved suboptimal. Most of them come from the packaging and distribution segment.

After that, we describe the analysis we took on how to fix the issues for the second release. The analysis includes a presentation of the state of the art technologies, like containerization, and the reasoning behind the decision to use them. Finally, we describe the implementation in ProaSense.

TABLE OF CONTENTS

Table of Contents

Acronyms

Acronym	Explanation
CBM	Condition Based Maintenance
CBM	Condition Based Maintenance
CEP	Complex Event Processing
DevOps	Development & Operations
DSS	Decision Support System
DTO	Data Transfer Object
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTP	Hypertext Transfer Protocol
MDP	Markov Decision Process
OODA	Observe, Orient, Decide and Act

1. Introduction

1.1 RELATIONS TO OTHER PROASENSE DELIVERABLES AND TASKS

This deliverable is directly related to *Deliverable 6.2 Integrated ProaSense platform and its assessment v1*. That deliverable explains the integration plan and procedure undertaken in the first iteration of the project. Deliverable 6.3 explains the conclusions we made after integrating the first version of the platform, what the challenges and pitfalls were, and explains what has been done to rectify the issues. In addition, the UI developed for the first integrated version was described in a separate document, *Deliverable 6.2.1 ProaSense UI platform v1.* The second iteration of the UI is a part of this deliverable. Deployment shall be elaborated in a separate document, *Deliverable D7.3*, which explains the procedure undertaken at the use-case partners' premises. That deliverable contains the user manuals, while in the previous iteration they were bundled with the predecessor of this deliverable, *Deliverable 6.2.*

1.2 DELIVERABLE STRUCTURE

This deliverable is divided in three main parts: conclusions from the first iteration related to integration and user interface, activities to rectify issues observed in the first iteration and, finally, a description of the installation process and manuals.

In chapter 2, we look back at the integration process in the first version and lessons learned from it. This chapter has three distinct parts, first one related to the integration solution itself, second related to the packaging and distribution of ProaSense to the use case partners and the third one, which describes the conclusions about the UI.

Chapter 3 states the actions done in the second iteration to fix issues explained in chapter 2. It explains possible solutions we researched, which we chose and why. We explain Docker and Docker Compose as chosen solutions, their application in ProaSense and also the UI revamp.

Chapter 4 describes the planned installation progress and references the installation manuals.

2. Conclusions from first iteration

2.1 INTEGRATION SOLUTION

The first iteration of the integrated platform is described in *D6.2, section 3 – Integration Details.* In practice the solutions chosen there showed satisfying stability, were not too complicated to maintain (although ProaSense has an inherently complex architecture) and acceptable performance. With those conclusions, we did not have too much work during the second iteration on changing the architecture, message structure or redefining the integration plan.

2.2 PACKAGING AND DISTRIBUTION

Previous packaging and distribution solutions were good enough to serve the purpose, but we had too optimistic expectations in the quality that those solutions provided.

First of all, the components were packaged in separate ways (Docker images, zipped binaries, installers) and installed using different tools, like using bash scripts, Ansible automation scripts, Docker-compose and so on.

Heterogeneous packaging of separate components introduced unnecessary "friction" while installing. That means each of the installation methods needs some requirements that should be supplied for it to work. Docker-compose needs network ports open, Ansible needs some in-place configuration for it to run properly, other solutions need firewall or proxy reconfiguration, and so on. Overly complex installation, although feasible, leaves a sense of an unfinished product and is a general UX issue.

Secondly, transparent component connections, which should be configured at installation potentially, can lead to misconfigurations, which in return waste time and lead to debugging session.

Taking all of the above into account, the development team sought out for a way to rectify the situation.

2.3 UI

In the first version of the integrated system, the ProaSense user interface consisted of a web page containing links to other ProaSense components that also provide a user interface. For the second iteration, this user interface was improved in terms of usability in order to better reflect the purpose of each component related to the user's current role. In addition, a central authentication system was added that, on the one hand, allows users to log in to the ProaSense system from a single system and, in addition, allows other ProaSense components to receive user details from the authentication system.

3. Activities in second iteration

3.1 MITIGATING RISKS FROM FIRST ITERATION

Since we already used a variety of packaging and deployment techniques throughout the work packages, it was most sensible to first look among them, which performed best. All are a part of present day software development practices (often dubbed DevOps[1]), which research confirmed.

We had three main approaches for executing the installation: **Shell script**[2] for components in work package 2, **Docker**[3] for components in work package 4 and **Ansible**[4] for work package 6.

- **Shell** scripts are a simple way of writing utility programs and tools available on Linux, Mac OS, and in recent time, partially, no Windows. They are designed to be run with a command-line interpreter. Shell scripts are typically written as a series of simple commands mixed in with control-flow structures that are generally not available in pure command-line interfaces. They are massively used for administration and other uses in almost any part of computing. However, we have to put them in perspective of the problem we are trying to solve:
    - Advantages:
        - Omnipresent, they can be found in any operative system (even new releases of Windows)
        - Simple to read and to write
        - Nicely documented
    - Disadvantages:
        - Complex operations become lengthy
        - If not being careful, program complexity can increase and become hard to manage
        - Parametrization and modularization can look “messy”.
        - Hard to debug
        - They only solve the automatization part of the installation, not the software packaging part and it’s complicated to effectively control multiple machines (a cluster) through one script.

[1] https://newrelic.com/devops/what-is-devops
[2] http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
[3] https://www.docker.com/
[4] https://www.ansible.com/

- **Ansible** is a software for automatization, managing and configuring multi-node setups. It executes commands via SSH on remote computers, and with a powerful command descriptions (specified in YAML) and node inventory system it is capable of setting up and maintaining a complete cluster. It is a part of a new generation of DevOps tools like Puppet[5] or Chef[6], but is commercially free to use. It should be noted that in some scenarios Ansible is used not as a substitute for other technologies as shell or Docker, but as a complement, to take over any possible step that the administrator must do by hand. However, we will not use this approach since we would rather trade a few manual steps for easier overall system understanding.
    - Advantages:
        - Expressive and easy to read language
        - Easy to split instructions from parameters from inventory (server list)
        - Easy to test
        - No need to install dependencies on cluster member
    - Disadvantages:
        - Needs to be installed on the "managing" computer, the one that is managing the cluster, which may conflict with firewalls
        - Still does not provide a way to package or distribute software, "only" to execute distributed commands on machines
        - Not so well known among administrators (although the whole field is relatively new)

- **Docker** is a technology for packaging Linux application within *containers*. In a simplified way of speaking, containers hold the software and its dependencies in a single package, which is easily portable and runnable on separate machines or clusters. Docker uses some advanced virtualization technology, and the containers run in lightweight virtual machines. Each container running on one machine is isolated from any other container, and the applications on the host OS. Details will be explained in the next chapter, however main advantages will be listed here

[5] https://puppet.com/
[6] https://www.chef.io/chef/

- Advantages:
    - Provides a way of packaging software (Docker builds)
    - Provides a way of orchestrating containers (Docker-compose)
    - Containers can be run on single machines or in a cluster (multiple cluster implementations exist: Swarm, Google Kubernetes, Mesos…) and containers run with cloud providers
    - System could be installed by running one script only with simple inputs from the user
- Disadvantages:
    - All hosts need to have Docker installed
    - Docker is still a technology in development, some updates can cause backward incompatible changes so it is important to care about the Docker version the image is built and the target one.
    - There is a lot of things involved behind the scenes so inexperienced system administrators can be overwhelmed
    - Building images might be slow (which is an issue in development, not production).

Considering everything, we chose to package all of the components into Docker, or colloquially, "dockerize" them. The details of Docker are explained in the next chapter.

3.2 DOCKER & DOCKER COMPOSE

As we previously roughly described, Docker is technology built on Linux operating system level virtualization. Operating system level virtualization presents is a way of "sharing" a Kernel between multiple isolated user spaces. Docker uses these features to create packages of applications, settings and dependencies. Those packages are called *images*. Instantiated images (images put into work) are called *containers*. In other terminology, containers to images are what objects are to classes in object oriented programming.

Containers do not contain an operative system, but they use the underlying host kernel functions. What they do include is a "base image", created from a Linux distribution that includes additional binaries, and file system settings which differ from distribution to distribution. Usually Docker images are created with very lightweight base images. This is shown in the right side of Figure 1.

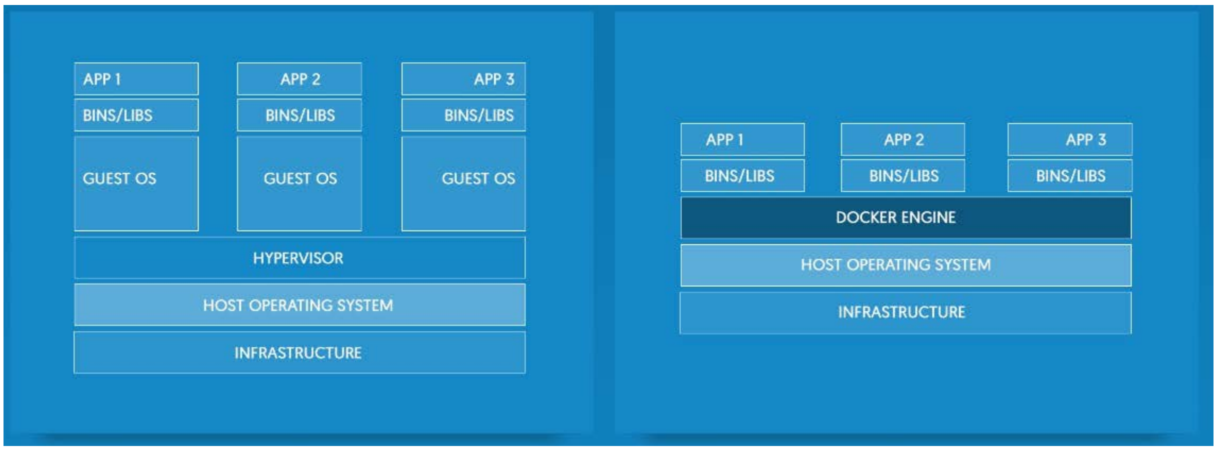


Figure 1 - Docker/VM comparison (source: docker.com)

On the figure, the *bin/libs* on the right side actually represent the base image. The left represents a classical virtual machine organization, like the one we used in Proasense earlier. A host operating system with a hypervisor runs other virtual machines with a complete OS, and then atop of that, we can see binaries, libraries and the application.

It is important to say that the base images can be extended and repackaged to create new base images. For example, Alpine Linux[7] is a simple, Linux distribution which has only basic features over the kernel. It is used for creating the Alpine[8] base image. The image can be used for creating a new image by installing Java. That way we got alpine-java[9], which is a base image created for running Java applications atop of them.

New images are created using *Dockerfiles.* A Dockerfile is a textual file with instructions on how to build the image. It starts with a declaration of the base image used followed with commands to copy files to the image from the machine on which the image is created, extracting them, which commands to run after a container is created from the image, which ports should be opened and so on. There are numerous commands, and they are available in the Docker documentation[10].

```
# A basic apache server. To use either add or bind mount content under /var/www
FROM ubuntu:12.04

MAINTAINER Kimbro Staken version: 0.1

RUN apt-get update && apt-get install -y apache2 && apt-get clean && rm -rf /var/lib/apt/lists/*

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
```

[7] https://alpinelinux.org/
[8] https://hub.docker.com/_/alpine/
[9] https://hub.docker.com/r/anapsix/alpine-java/
[10] https://docs.docker.com/engine/reference/builder/

```
ENV APACHE_LOG_DIR /var/log/apache2

EXPOSE 80

CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

Figure 2 - Dockerfile example

The Dockerfile in Figure 2 creates an image with from a Ubuntu 12.04 base image. It also declares:

- A maintainer with the MAINTEINER keyword
- What to run during image build with the RUN keyword (installs apache2 in this example)
- Setting environment variables with ENV keyword
- The container, after being run, should open port 80 for the application it contains (in this case apache2)
- At the end, with the CMD keyword we define which application should run along with the container. It is important for this application to run in the foreground. When it shut downs, the container exits as well, and the application exit status is reported to the Docker daemon.

Often enough multiple containers need to be orchestrated for one application to run. That is a modern tendency with microservices and event driven architectures. That is why Docker-compose exists. It is important to say that Docker-compose is not the only way to orchestrate containers. There are other orchestration solutions like Kubernetes or Mesos. Docker-compose is the official Docker orchestration tool. Docker-compose runs on one machine, with a Docker Swarm cluster (the official Docker clustering tool) or Rancher Cattle cluster. Kubernetes, for example, does not have anything in common with Docker-compose. It is an “opinionated” system that means it reimplements the Docker orchestration process from the ground up.

A Docker Compose file is a YAML file defining services, networks and volumes. The default path for a Compose file is ./docker-compose.yml. An example is shown in Figure 3.

```
es:
  image: elasticsearch
  container_name: es
web:
  image: salex89/sf-foodtrucks
  command: python app.py
  ports:
    - "5000"
  links:
      - es
```

Figure 3 - Docker Compose example

This is a simple Docker Compose file which orchestrates two images. The first one is with elasticsearch and the other one is with a simple Python app. The image names are, if not configured otherwise, searched on Docker Hub (https://hub.docker.com/). When the author of the image builds it, they can upload it to the main Docker image repository for others to use. Referencing it from a Docker Compose file means it should be pulled from the repostitory. In the case of the *web* image, we can see that port 5000, exposed by the container by default, will just be exposed as port 5000 on the host machine. In other cases we would maybe forward the container's port 80 to port 4347 on the host machine (to avoid port clashing) and so on.

The links portion is especially interesting. Using it, we can link two containers, like connecting them through a network. In the background Docker puts an entry into the *HOSTS* file of the linked container, which points to the target container, and that hostname can be referenced from applications in that container.

Among other important settings, through the Docker-compose file we can set environment variables to be injected into the containers. Those environment variables can be used for runtime parameterization.

It is not necessary to use the Docker Hub to host Docker images. Self-hosted, 3rd party implementations exist which can be secured and used for private purposes only. Before building a Dockerfile or Docker-compose script the user can specify another repository to be used, with credentials if needed.

3.3 DOCKER IN PROASENSE

The overall usage of Docker in ProaSense is straightforward. We decided to use Docker in ProaSense to package all of the components. Partners are instructed to package their components into Docker images, in the way they find it suitable. The Docker images should be pushed to a private Docker repository maintained by FZI (https://laus.fzi.de:8201/).

Partners are expected to provide a Docker-compose file per component, which transcribes a Docker-compose file per partner, and parametrize the connections to other components through the environment variables.

The files can be used separately, to install the components independently of ProaSense. It is also possible to "merge" the Docker-compose files into one, and to be able to install the complete ProaSense platform in one run. This can be done in place, when the installation is planed.

One part that is slightly more complicated to dockerize is the Storm cluster. That is why a separate Docker-compose file is provided to deploy a Storm production-ready multi-node cluster ("docker-compose-production.yml"). With that deployment scheme it is necessary to manually deploy the topology to the cluster.

3.4 UI

The overall objective of the ProaSense user interface is to provide a common entry point to the whole ProaSense system. As ProaSense consists of various modules containing multiple user interfaces targeted at different roles, the choice of an appropriate technology suitable for the provision of a unified UI is a challenging task. Therefore, an evaluation of possible technologies needed to be conducted in order to fulfil the requirements of all involved parties. In general, two different opportunities exist to develop an integrated user interface. The first option is to develop a single user interface that implements the full feature set provided by all components. In this scenario, required interfaces are provided by the backend implementations of all ProaSense components. However, the disadvantage of such a solution is that it requires an agreement on a single technology used for the frontend implementation. As several ProaSense components are built to provide an extensive user interface (e.g., the pipeline editor of the WP3 component and the KPI modeller of the WP5 component), this solution would require different components to be based on the same technology. Therefore, we decided to build a lightweight integrated user interface that, on the one hand, provides a single point of entry to the ProaSense system, but on the other hand does not make any assumptions on the individual frontend technology used by a component.

This solution also better supports individual exploitation of single ProaSense components, for instance, standalone version of these components can be provided to users that do only require a subset of the full ProaSense platform. Therefore, instead of a monolithic architecture our approach allows the integration of technically independent ProaSense components under one roof. As already mentioned, each component can be either provided as a set of Docker images provided as a Docker-compose file, or as a whole system consisting of all components and the integrated user interface in a single cocker-compose file.

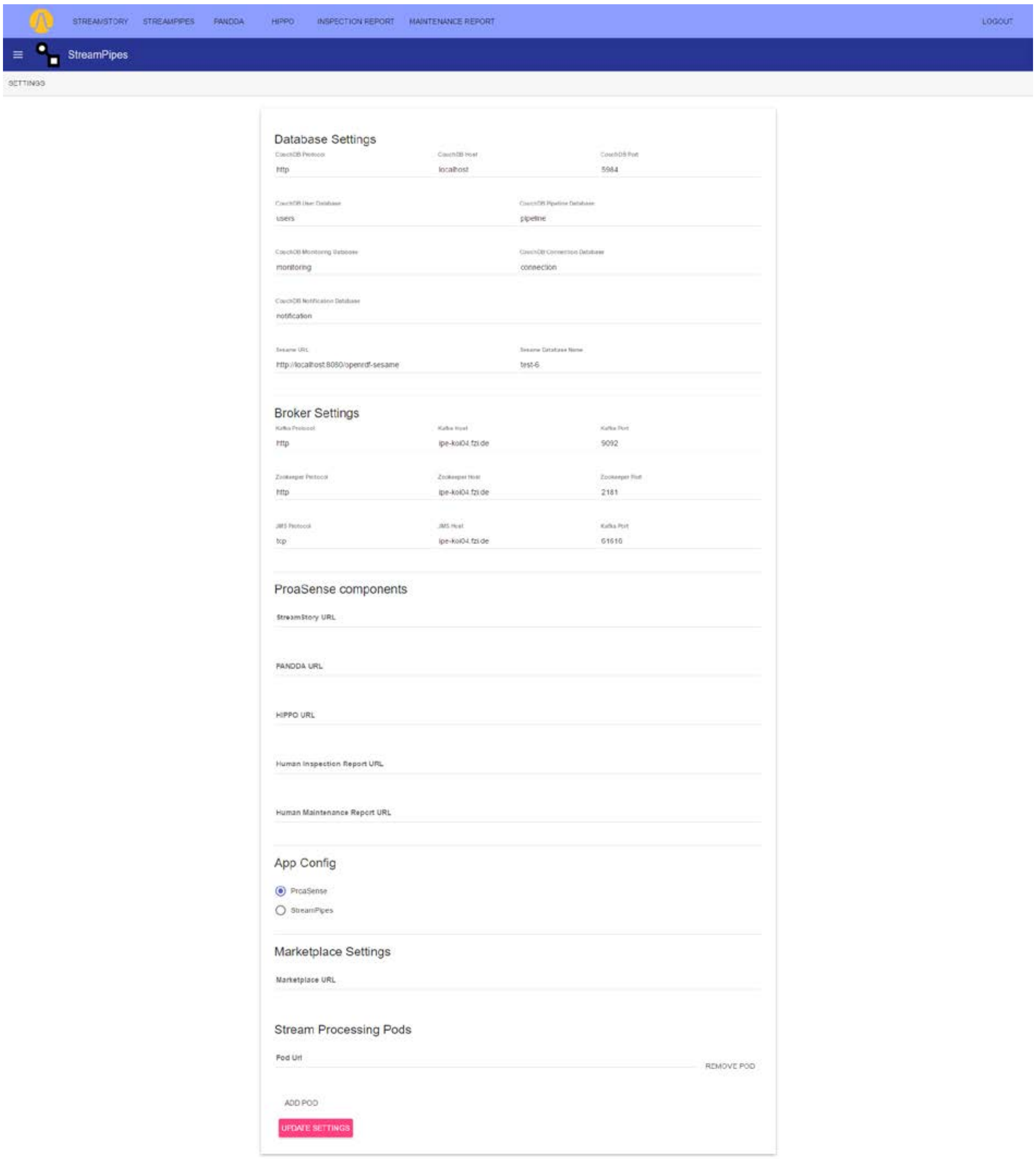
The integrated ProaSense user interface is based on a component written in AngularJS and can be configured in order to integrate the various ProaSense components depending on the actual installation. In the following sections, we describe the configuration, navigation and authentication mechanisms provided by the integrated user interface.

3.4.1 CONFIGURATION

The configuration dialog allows users to setup and configure the system. It is provided automatically at the first time the ProaSense system is initialized and can be accessed via the “Settings” dialog within the left main navigation bar. Figure 4 shows the main settings dialog. For each ProaSense component that has a user interface, a link targeting at the main entry point of the component's UI can be provided. This includes the following components:

- StreamStory: Allows users to analyse event streams and to configure models that are used to create predictions.
- StreamPipes: Allows users to define stream processing pipelines in an ad-hoc manner.
- PANDDA: Allows users to access the ProaSense descision dashboard.
- Business Improvement Analyzer: Allows to model and analyse key performance indicators.
- Inspection report: A simple user interface (HTML form) to submit an inspection report to the ProaSense system.
- Maintenance report: A simple user interface (HTML form) to submit a maintenance report to the ProaSense system.

Furthermore, additional settings can be provided such as connection settings of the Kafka broker used for communication between individual ProaSense components. These settings can be retrieved by all involved ProaSense components using a specified REST interface. If the configuration dialog is called for the first time, an initial admin user has to be provided additionally.



Database Settings

CouchDB Protocol	CouchDB Host	CouchDB Port
http	localhost	5984

CouchDB User Database	CouchDB Pipeline Database
users	pipeline

CouchDB Monitoring Database	CouchDB Connection Database
monitoring	connection

CouchDB Notification Database
notification

Sesame URL	Sesame Database Name
http://localhost:8080/openrdf-sesame	test-6

Broker Settings

Kafka Protocol	Kafka Host	Kafka Port
http	ipe-koi04.fzi.de	9092

Zookeeper Protocol	Zookeeper Host	Zookeeper Port
http	ipe-koi04.fzi.de	2181

JMS Protocol	JMS Host	Kafka Port
tcp	ipe-koi04.fzi.de	61616

ProaSense components

StreamStory URL

PANDDA URL

HIPPO URL

Human Inspection Report URL

Human Maintenance Report URL

App Config

(●) ProaSense
( ) StreamPipes

Marketplace Settings

Marketplace URL

Stream Processing Pods

Pod Url

Figure 4: Setup Dialog

The REST API of the configuration interface returns a JSON document that includes currently configured settings. In order to access the configuration, a component needs to be registered with the central authentication system (further explained in section 3.4.2). Table 1 shows an example message returned by the configuration endpoint.

Table 1: Configuration REST API

```
       "couchDbProtocol": "http",
       "couchDbHost": "localhost",
       "couchDbPort": 5984,
       "sesameUrl": "http://localhost:8080/openrdf-sesame",
       "sesameDbName": "rdfstore",
       "hippoUrl": "",
       "panddaUrl": "",
       "streamStoryUrl": "",
       "humanInspectionReportUrl": "",
       "humanMaintenanceReportUrl": "",
       "kafkaProtocol": "http",
       "kafkaPort": 9092,
       "kafkaHost": "ipe-koi04.fzi.de",
       "jmsProtocol": "tcp",
       "jmsPort": 61616,
       "jmsHost": "ipe-koi04.fzi.de",
       "zookeeperProtocol": "http",
       "zookeeperPort": 2181,
       "zookeeperHost": "url",
       "couchDbUserDbName": "users",
       "couchDbPipelineDbName": "pipeline",
       "couchDbConnectionDbName": "connection",
       "couchDbMonitoringDbName": "monitoring",
       "couchDbNotificationDbName": "notification",
       "appConfig": "ProaSense",
       "marketplaceUrl": "",
       "podUrls": [
         ""
       ]
     }
```

3.4.2 AUTHENTICATION

In order to allow users to log in to the ProaSense system, a central authentication interface is required. This authentication system should be able to support a centralized login mechanism by integrating distributed, standalone components. The integrated UI provides a login page as illustrated in figure 5. In addition, the system can be configured so that users are able to register themselves, which is especially important for demo purposes. In this settings, users can also select a role they would like to evaluate. If the system is used in a more production-oriented environment, the self-registration feature can be turned off, so that only users that have an administrator role are able to add new users to the system.

The architecture of the authentication system is shown in Figure 6. In order to use the centralized authentication server, the individual components needs to embed an iframe into their own user interface. The URL of the *iframe* links to a RESTful interface of the authentication server. If the user is already authenticated, a cookie is transmitted to this server. Afterwards, this cookie is verified. If the cookie is considered valid, another REST endpoint of the requesting ProaSense component is called. The authentication server submits a token to this component. This token can be used in order to get user credentials as illustrated in Table 2. This message contains information on the currently logged in user along with further information such as the email address. In order to support multiple views of the ProaSense system for multiple user roles, a set of roles assigned to the user and permissions are submitted within the authentication message.

In cases where the user is not authenticated successfully, the login screen is shown to the user.

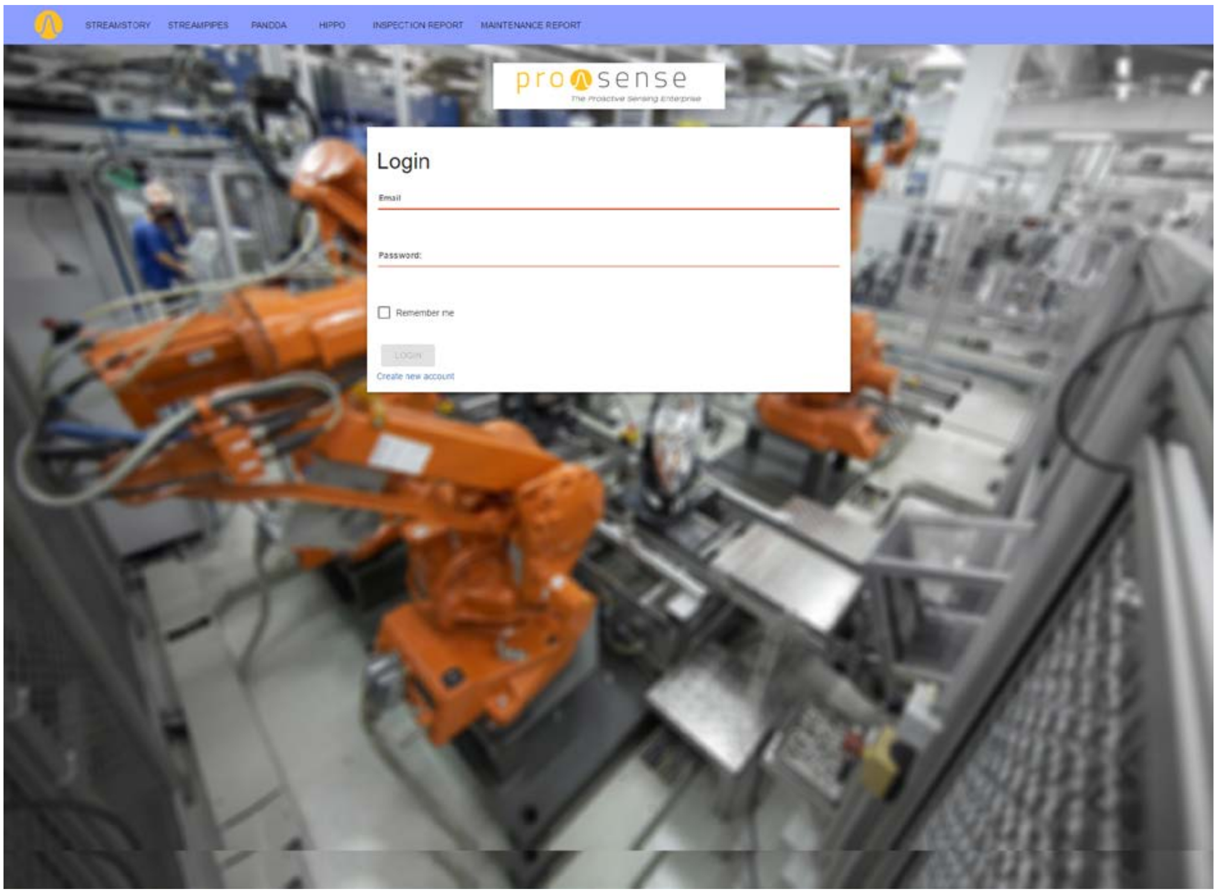


Figure 5: Login Screen

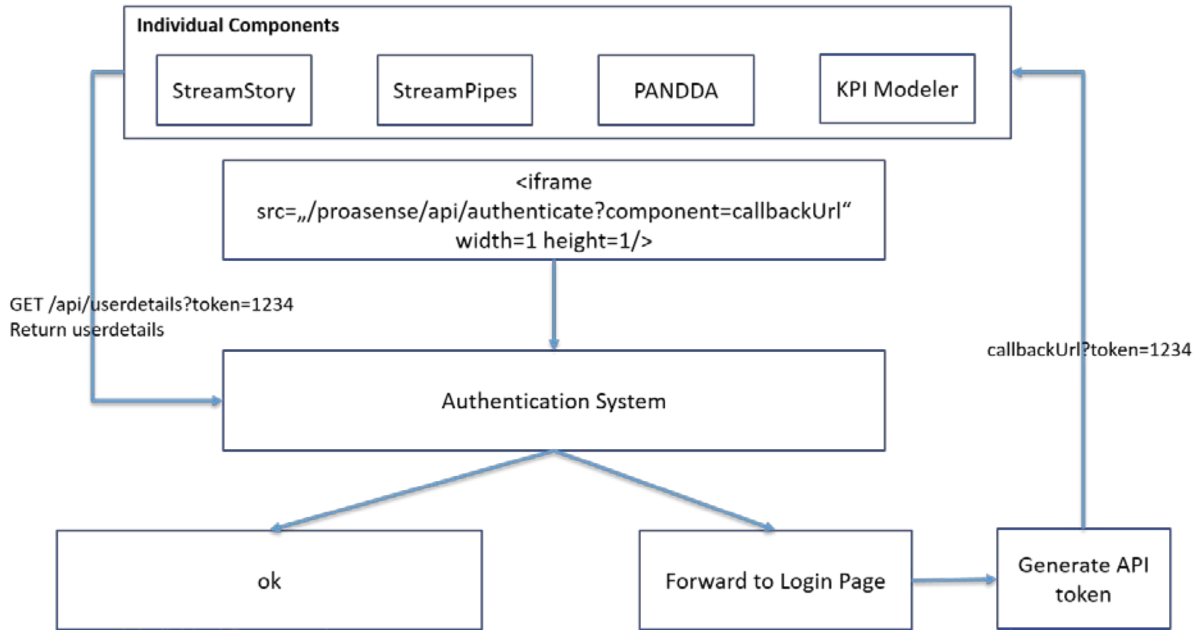


Figure 6: Architecture of the Authentication System

Table 2: Authentication message

`{"info":{"authc":{"principal":{"email":"user@proasense.eu","apiKey":""},"credentials":{"name":"user@proasense.eu"}},"authz":{"roles":["USER_DEMO","SYSTEM_ADMINISTRATOR"],"permissions":[]}},"success":true}`

4. Installation and manuals

4.1 INSTALLATION PROCESS

The installation plan is very similar to the one from the first installation, only executed much quicker due to the simplified packaging.

The reference platform installation will be executed on the Nissatech servers. Also the MHWirth evaluation is slightly complicated to execute on MHWirth's servers since the lack of a reliable data sources and influences out of the project scope. However, MHWirth staff will execute the ProaSense installation on Nissatech servers and evaluate ProaSense on them.

At Hella it is possible to install ProaSense and test it in practice, and the evaluation for Hella will use that instance.

At Nissatech each partner has its own machine for each work package. Therefore, at Nissatech we will try to install each component separately, and distribute on multiple servers, while at Hella we will execute the installation with one composite Docker-compose file, and the scaling will be handled with the underlying infrastructure.

4.2 INSTALLATION MANUALS

Installation manuals are generally prone to changes after the finalization of this deliverable. For this reason, they are hosted on the ProaSense website, on the URL: http://www.proasense.eu/manuals/ .

5. Conclusion

As we described, most of the work for WP6 in the second integrated version was related to the packaging and UI.

During development we immediately noticed a big improvement in the proposed way of standard packaging. Migrating, restarting, modifying and redeploying parts of ProaSense was massively simpler and we had far less installation and configuration bugs. The use case partners also expressed great relief when they were presented with the simplified installation process.

On the other hand, the centralized authentication made the system introduced a higher level of security, which is always necessary in industrial systems. It also made the system to feel as a homogenous, compact product.