ZDMP: Zero Defects Manufacturing Platform

WP8: Product Quality Assurance

EU ID: D110: Production: Non-Destructive Product Inspection - Vs: 1.0.0A

ZDMP ID: D8.3a

Deliverable Lead and Editor: Mauro Fabrizioli, VSYS

Contributing Partners: IKER, VSYS

Date: 2020-06

Dissemination: Public

Status: EU Approved

Abstract

The deliverables for this task, and all WP5-8 tasks, are software and are of EU type “OTHER”. The software and accompanying material (eg description, instructions) is available on the ZDMP software repository which is updated dynamically. However, for EU formal reporting purposes, this brief cover document provides a formalised pointer to the downloadable software and related content. This deliverable should read in conjunction with the D006-D020 deliverables which document the software process/status for each WP/Task. This deliverable represents the status as at M18 with further living editions at M18 and M48

Grant Agreement: 825631

Document Status

Deliverable Lead	Mauro Fabrizioli, VSYS
Internal Reviewer 1	Ernesto Bedrina, CET
Internal Reviewer 2	Juan José Gabilondo, ETXE
Internal Reviewer 3	Stuart Campbell, ICE
Type	Deliverable
Work Package	WP8: Product Quality Assurance
ID	D110: Production: Non-Destructive Product Inspection
Due Date	2020-06
Delivery Date	2020-06
Status	EU Approved

History

See Annex A.

Status

This deliverable is subject to final acceptance by the European Commission.

Further Information

www.zdmp.eu and mailto:info@zdmp.eu

Disclaimer

Project Partners:

Executive Summary

The main objective of “WP8: Product Quality Assurance” is to ensure the quality of the product along the value chain of the manufacturing process by developing zero defect manufacturing (ZDM) applications based on digital models of manufacturing assets and manufacturing processes. The deliverables of this work package and the WP1 Management work package are divided into software packages and document/reports. In terms of reporting:

- **Process/Status:** Report D110 Production: Non-Destructive Product Inspection of WP8 Product Quality Assurance, as identified in the DOA, focuses on the process/status of the work accomplished in Task T8.3
- **Software:** All WP8 software deliverables of T8.1-T8.4 (type “OTHER”) are available in the ZDMP public repository with access details and install instructions further described in this report which is a ‘current’ extract of the repository

“WP8: Product Quality Assurance” consists of: Modelling, Prediction, Inspection, and Supervision. The tasks of WP8 are the following:

- T8.1 - Characterization and Modelling / Digital Twin
- T8.2 - Pre-Production: Product Quality Prediction / Product Assurance Runtime - Quality Prediction
- T8.3 - Production: Non-Destructive Product Inspection / Non-Destructive Inspection
- T8.4 - Production: Supervision / Product Assurance Runtime – Quality Supervision

This deliverable represents Task T8.3 Production: Non-Destructive Product Inspection which in turn is composed of the following components:

- Non-Destructive Inspection

As reported in the architecture deliverable the purpose of these components is: “To inspect products for defects using non-destructive techniques. This component analyses in-line sensor data such as single values, data stream, and images. This powerful set of functionalities is intended to support zApp designers to build applications for monitoring product quality”.

Each of the components is structured into the following sections:

- General Description
- Architecture Diagram
- Features
- Requirements
- Installation
- How to Use
- Functional Requirements Implementation Status (M18)

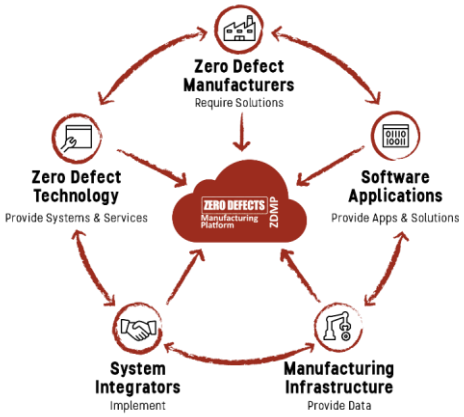
This report covers the period from the project start until M18 with most activity in the M13-M18 period. Further formal deliverables are due M30 and M48 as well as an informal iteration at 24.

Table of Contents

0 Introduction

Due to the cover nature of this deliverable; this introduction is presented in short-form only. For further information please consults D006 - Technical Management: Overview Report.

0.1 ZDMP Project Overview



ZDMP – Zero Defects Manufacturing Platform – is a project funded by the H2020 Framework Programme of the European Commission under Grant Agreement 825631 and conducted from January 2019 until December 2022. It engages 30 partners (Users, Technology Providers, Consultants and Research Institutes) from 11 countries with a total budget of circa 16.2M€. Further information can be found at www.zdmp.eu.

ZDMP aims at providing such an extendable platform for supporting factories with a high interoperability level, to cope with the concept of connected factories to reach the goal of zero-defect production. For this, the platform provides the tools to allow following each step of production, using data acquisition to automatically determine the functioning of each step regarding the quality of the process and product.

0.2 Deliverable Purpose and Scope

The deliverables for this task, and all WP5-8 tasks, are software and are of EU type “OTHER”. The software and accompanying material (eg description, instructions) is available on the ZDMP software repository which is updated dynamically. However, for EU formal reporting purposes, this brief cover document provides a formalised pointer to the downloadable software and related content. This deliverable should read in conjunction with the D006-D020 deliverables which document the software process/status for each WP/Task. This deliverable represents the status as at M18 with further living editions at M18 and M48. Specifically, the DOA states the following regarding this Deliverable:

T8.3	**Production: Non-Destructive Product Inspection**			**VSYS**	**Six Monthly**
D110 D111 D112	Production: Non-Destructive Product Inspection	OTHER (Prototype)	PU	18, (24), 30, 48, Reporting via T1.4.x Series	RDI3-6
This task involves the detection of defects during manufacturing production activity, by the application of effective and efficient non-destructive techniques (NDT) and solutions. These solutions will include the use of advanced sensors, cameras, measuring systems based on Artificial Intelligence, and Artificial Vision. The task will implement detection of defects such as dimensional, machining, welding, injection moulding, painting, and surface finishes whilst taking care to define a common time-base for all sensors. This will preserve the temporal correlation from different data sources and allows a real time analysis (ie Memoria® approach from Video Systems). The NDT inspection techniques will include metrology, radiography, and vibrational analysis thus providing a flexible and configurable machine vision approach. The results of the inspection will feed task T8.4 and will support it regarding decision making. In addition, the results will be used to adjust downstream processing steps (task T7.2) to ensure that the product quality stays in an acceptable range. A first prototype will integrate the different techniques in a general-purpose approach. This prototype will be adapted to the necessities of the different use cases.					

0.3 Target Audience

The primary target audience for this document are the partners and WPs of the project, as well as the EU and reviewers.

0.4 Deliverable Context

The deliverable context is as per Section 0.2:

Primary Preceding documents:

- **D006: Technical Management Overview Report:** Represents the general software status of the project including information on commits and WP5-8 Risks and mitigations
- **D018: Technical Management: WP8 Report:** Represents the process/status and future actions of this work package, including this task. It also includes related KPIs and their status
- **D055: Technical Specification and Update:** Describes the different APIs of the components

0.5 Document Structure

This deliverable is broken down into the following sections:

- **Section 1: Component: Non-Destructive Inspection**

0.6 Document Status

This document is listed in the Description of Action as “public” since it represents the open nature of the project’s software deliverables.

0.7 Document Dependencies

- None

0.8 Glossary and Abbreviations

A definition of common terms related to ZDMP, as well as a list of abbreviations, is available at http://www.zdmp.eu/glossary.

0.9 External Annexes and Supporting Documents

- See the ‘Resources’ grid within the General Description Section of each component

0.10 Reading Notes

- None

0.11 Document Updates

- This is the first version of this document

1 Component: Non-Destructive Inspection

1.1 General Description

This component runs Non-Destructive Inspection algorithms on images using Traditional Machine Vision and Machine Learning techniques. The component embraces a set of computer vision techniques to highlight defects and/or measure product physical quantities, such as dimensional ones, with the goal to allow assessing the quality and the conformity of the inspected product.

Resource	Location
Source Code	Link
Latest Release (v1.0.0)	Download
Builder Open API Spec	Link
Runtime Open API Spec	Link
Video	Coming soon

In addition to this, also a tool to model 3D objects starting from a 3D scanning source has been developed (see additional minor section at the end of each subsections marked as “3DScan” – as per the below). More specifically, it makes the large output files usable for other systems, mainly focusing on anti-collision applications. This is achieved by applying a series of 3D data processing algorithms to simplify and convert to suitable format so that certain criteria (mostly fixed by the anti-collision applications) are fulfilled.

3DScan

Resource	Location
3DScan Source Code	Link
3DScan Latest Release	Not released yet
3DScan Open API Spec	Not defined yet
3DScan Video	Coming soon

The date of generation of these components is: 2020-06-28

1.2 Architecture Diagram

The following diagram shows the position of this component in the ZDMP architecture.

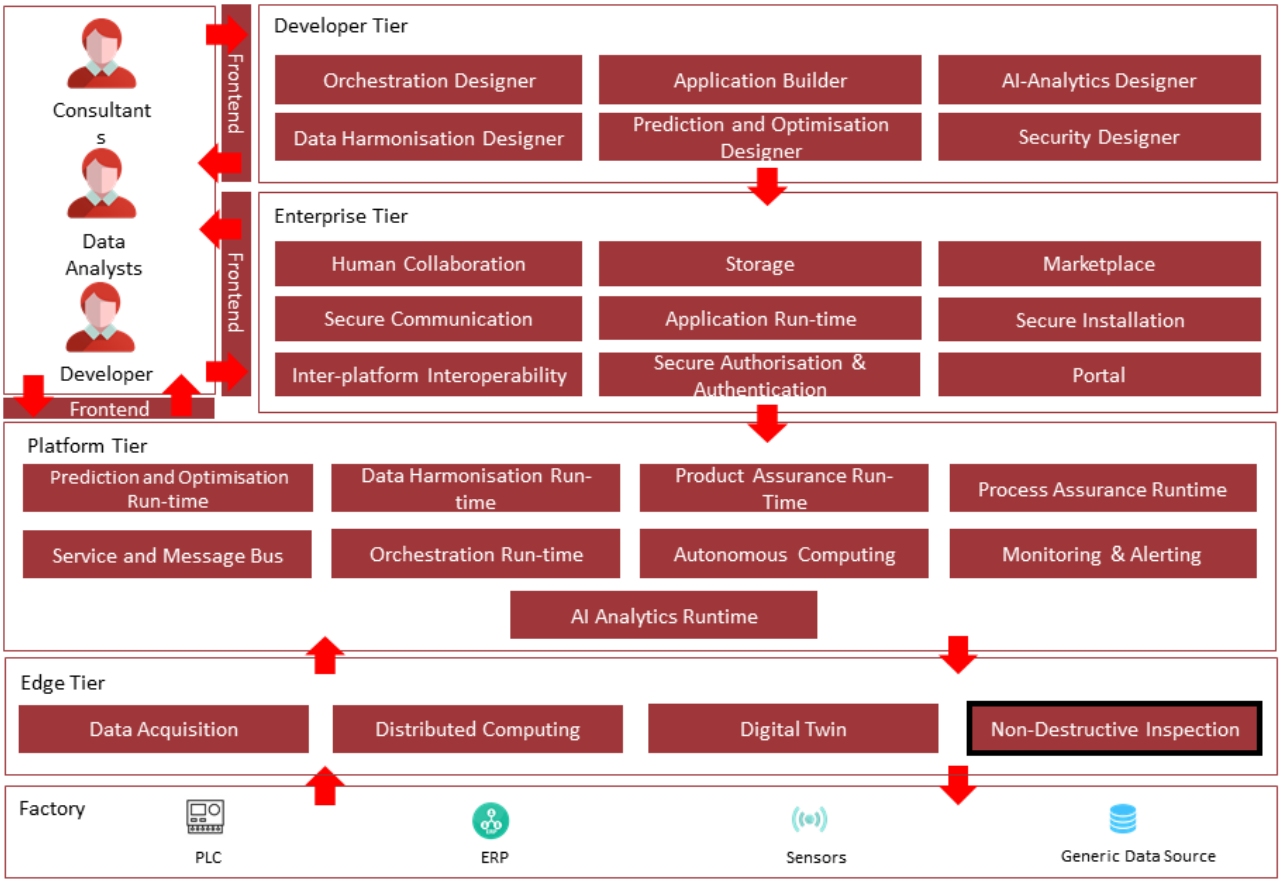


Figure 1: Position of component in ZDMP Architecture

1.3 Features

There are many Computer Vision algorithms that use different techniques, for example traditional Machine Vision, Image Processing, Pattern Recognition and Machine Learning.

This component allows a ZDMP Developer to build complex Computer Vision algorithms, with the help of Graphic Tools for testing and debugging. The algorithm can then be run at runtime, with real-time or historic data from cameras.

In particular, the Designer allows them to visually build an algorithm connecting multiple functions. Functions are atomic algorithms.

Debugging tools, such as the Image Visualizer in the image of Figure 2, help the developer build an algorithm. Seeing the outputs of every function is very useful an instant idea of what it does can be seen. When a developer changes a parameter, they will immediately see the result, without the need of recompiling the component nor a live image source.

The Designer can export the configuration to a file, and the Runtime will use that configuration to process images.

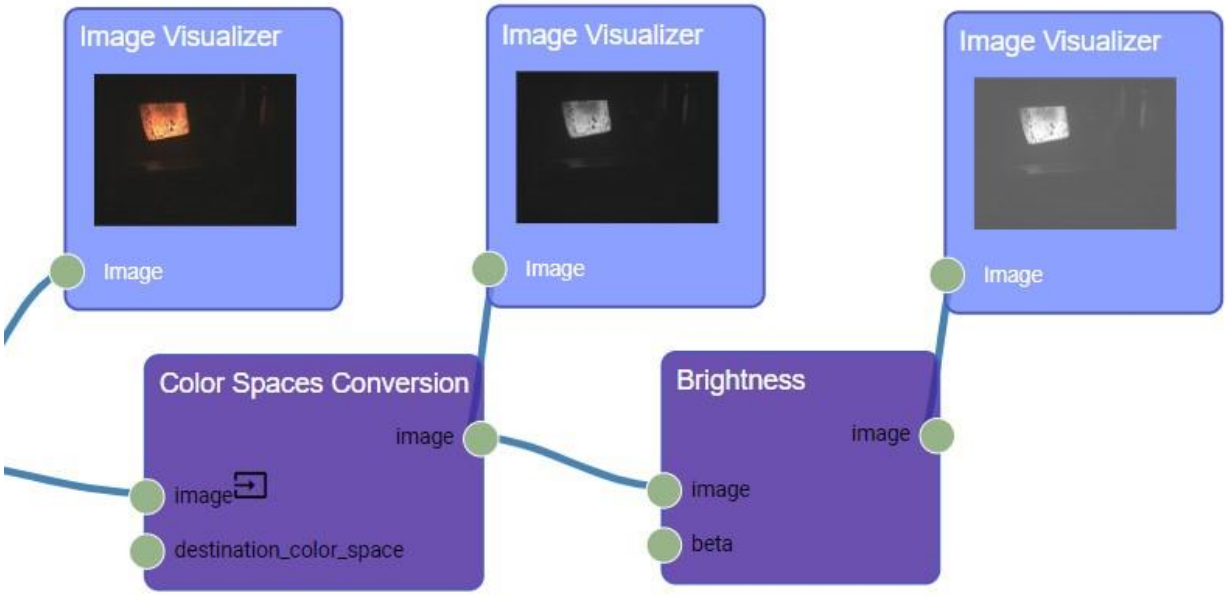


Figure 2. Use of Image Visualizer block as a debugging tool inside the Designer

The component is fully modular:

- A module is a package that contains a list of functions
- A module should run specific algorithms and have custom dependencies.

This way developers can create their own modules if they want to add some custom functions

Every module's function will appear in the Designer; in the image of Figure 2 it is possible to see two functions (Colour Spaces Conversion and Brightness) from the module Machine Vision.

There many Machine Vision algorithms and a specific use case most probably will not need all of them. That is why there is a Builder, which lets the developer choose which modules they want to enable.

The Builder also allows the developer to upload a custom module if they want to add a specific functionality. The Builder then generates an instance of the Runtime component with all the selected modules and their dependencies. Once the component is started, every function of the selected modules will be dynamically available in the Designer and of course the Runtime.

3DScan

Up until now, only the backend of the subcomponent has been developed.

The idea behind this component is to try to minimize the user interaction as much as possible in the obtention of a final suitable 3D model. So, for now, a zero-parameter approach will be implemented (more info in "how to use").

Summing up, a 3D model in a specific folder (a dummy model is provided for testing) is simplified and a final suitable file is created.

1.4 System Requirements

All the software dependencies are specific of every module and resolved via Docker, so they are not listed here.

Images can have a significant impact on the RAM usage, especially if the throughput is very high. 16GB is the minimum required and the user should run some tests to see the real-world usage in a specific situation.

Some modules may require an nVIDIA CUDA capable GPU, for example for Neural Network algorithms. Some more demanding scenarios need more than one GPU.

Software	Version
Docker	19.03.5+
Hardware	
8+ CPUs	
16GB+ RAM	
100GB+ disk	
x nVIDIA CUDA Capable GPU (if any module needs them)	

3DScan

All software dependencies are defined in the Dockerfile, so there is no need to specify them. As for hardware, output files of 3D scanning operations are quite heavy. They can easily size to 300 Mb. That is why a minimum RAM memory of 8 Gb is recommended.

Software	Version
Docker	18.09.7+
Hardware	
8GB+ RAM	

1.5 Installation

To build and run the Builder component simply needs to execute these steps:

```
cd orchestration/
docker-compose build && docker-compose up -d
```

This will start a Web Server and an API server.

3DScan

To test the subcomponent run:

```
sudo docker exec -it ObjectDefinition bash
```

and

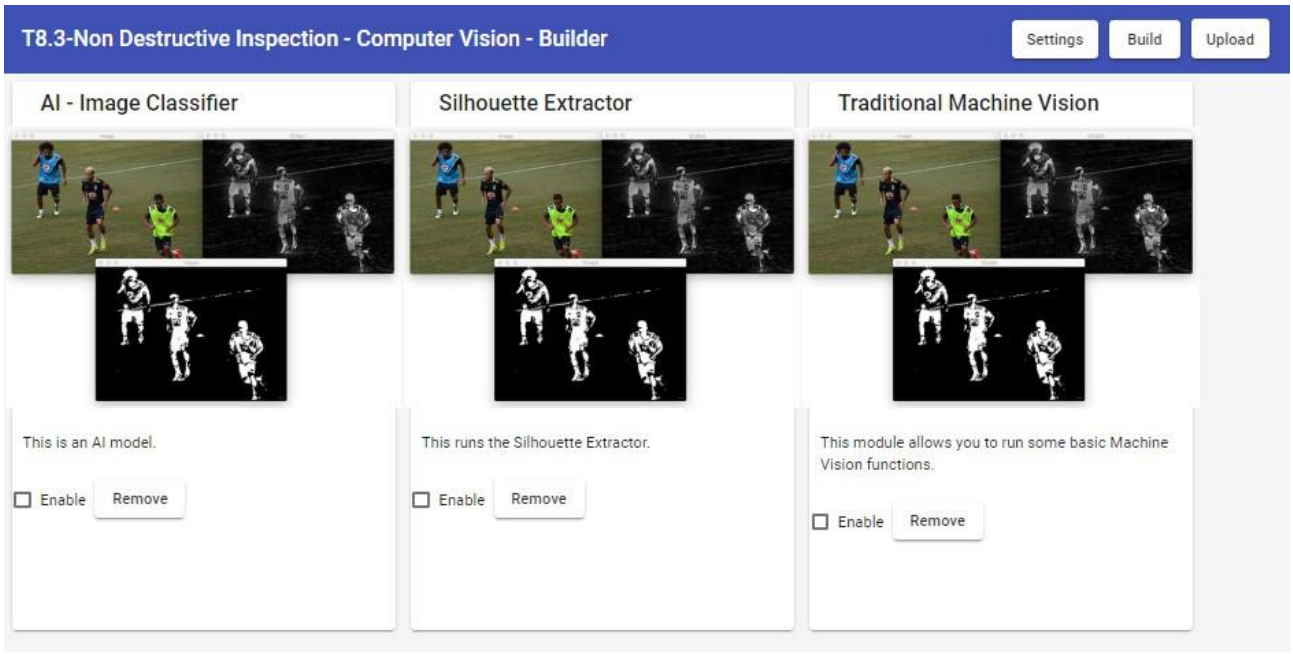
```
sudo docker exec -it ObjectDefinition bash
```

to get CLI within the container.

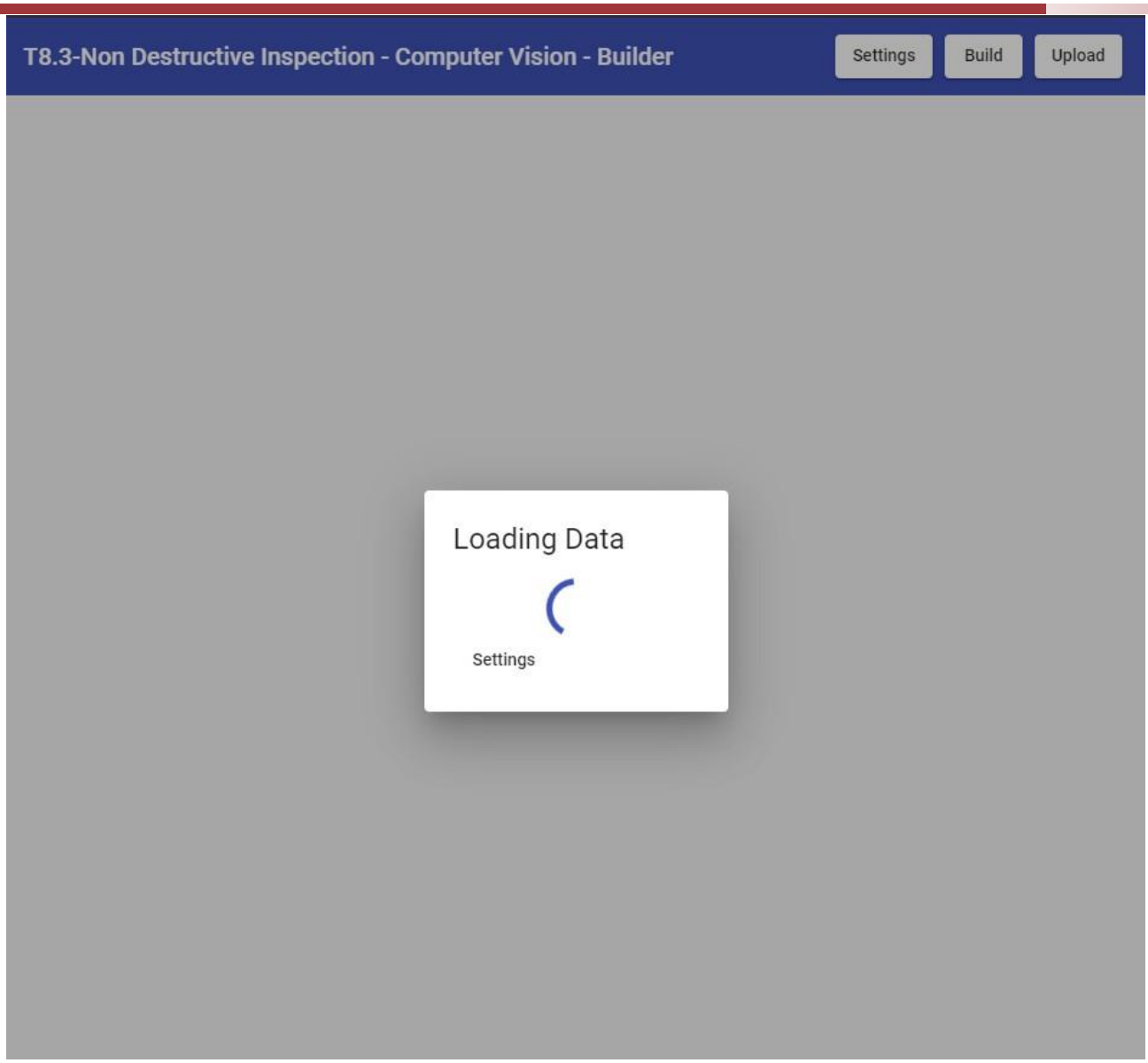
1.6 How to use

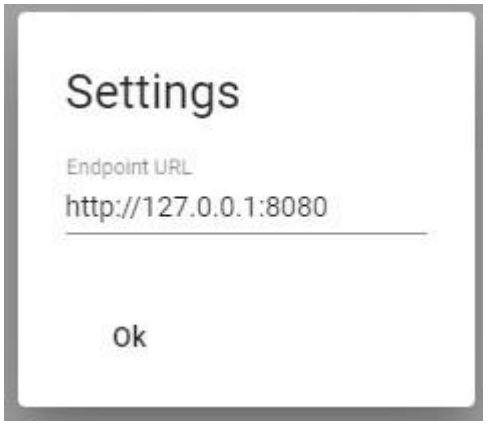
Once the Builder component has been built and started, the user can reach the Web UI at this address, assuming it is running it on a local machine:

http://localhost:4200/



If a user is stuck with the following, this means that the Endpoint URL of the backend is wrong and they can click on *Settings* to change it.

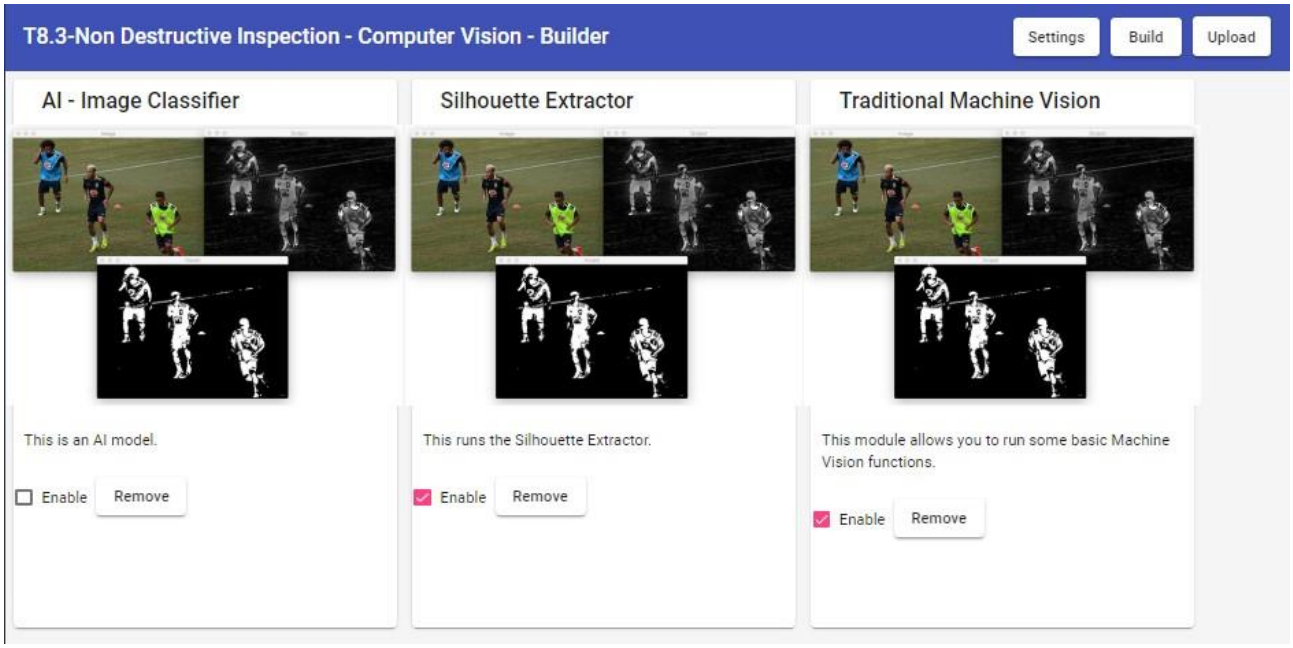




When running it on a local machine, this should be the correct URL: http://localhost:8080/

This Web UI allows uploading custom modules and choose which ones are needed for a custom component. Some modules are already included by default. Every module has a description text box that gives an idea of what it does.

Each component can be enabled by checking the *Enable* checkbox. For this example, check the Traditional Machine Vision and the Silhouette Extractor modules which should look like this:



To build the final component containing all the selected modules and their dependencies, click on the *Build* button, which downloads a .zip file containing all the necessary resources. This .zip file allows building and running the component by independently, but in the final version the Application Runtime will do this automatically.

From now on, the Builder component is not needed anymore and can be killed via:

```
docker-compose down
```

To build the newly generated component (the Runtime component), extract the zip file and then execute the same commands as before:

```
tar -xf ARCHIVE_NAME.zip
cd orchestration/
docker-compose build && docker-compose up -d
```

This will build and start the Runtime component, which is made of the Designer, the API Server, and any other dependencies.

The features to achieve the functionality of this component are itemised below and explained there after:

- Runtime Component
- Integration with Other Components
- Writing own Modules

1.6.1 Runtime component

The Runtime Component allows developers to build a Computer Vision algorithm with the help of the Designer and run that algorithm at runtime with real-time or historic data.

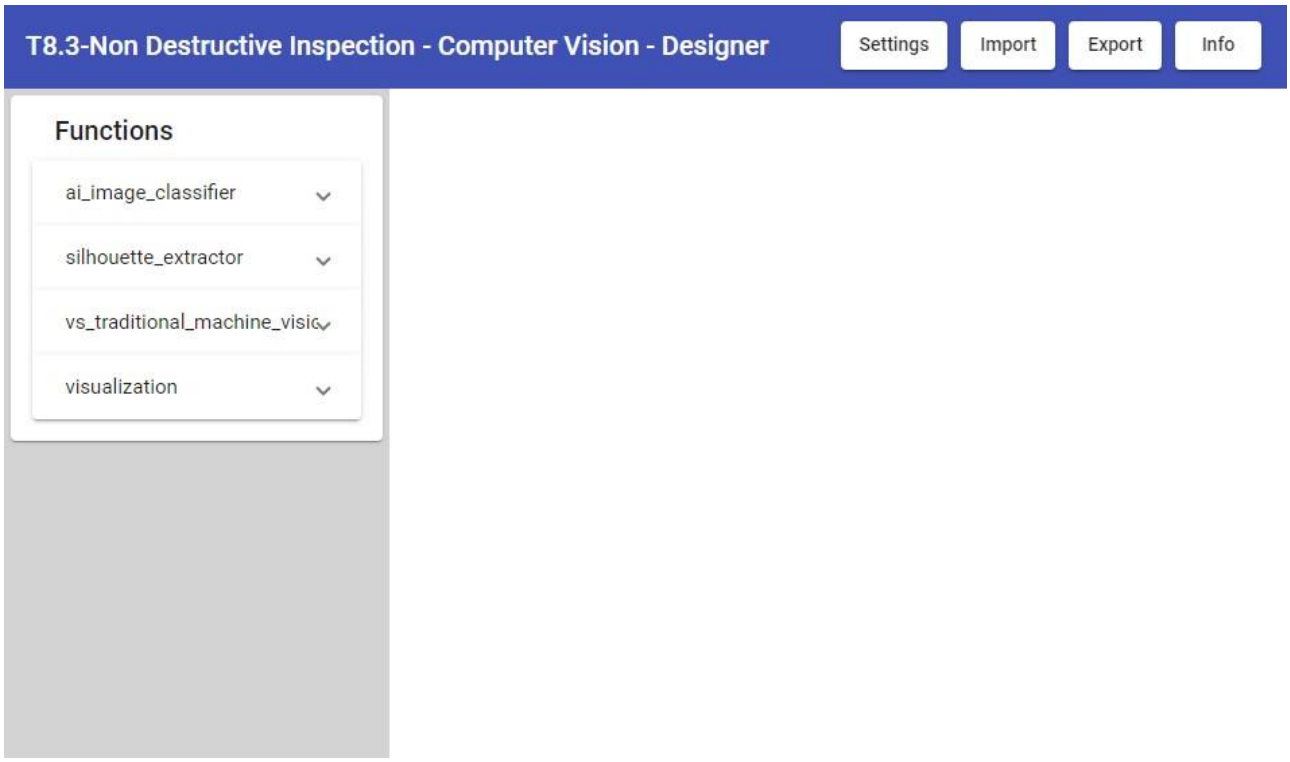
It is made of two parts:

- Designer: Web interface
- Backend: processor of multiple algorithms

1.6.1.1 Designer

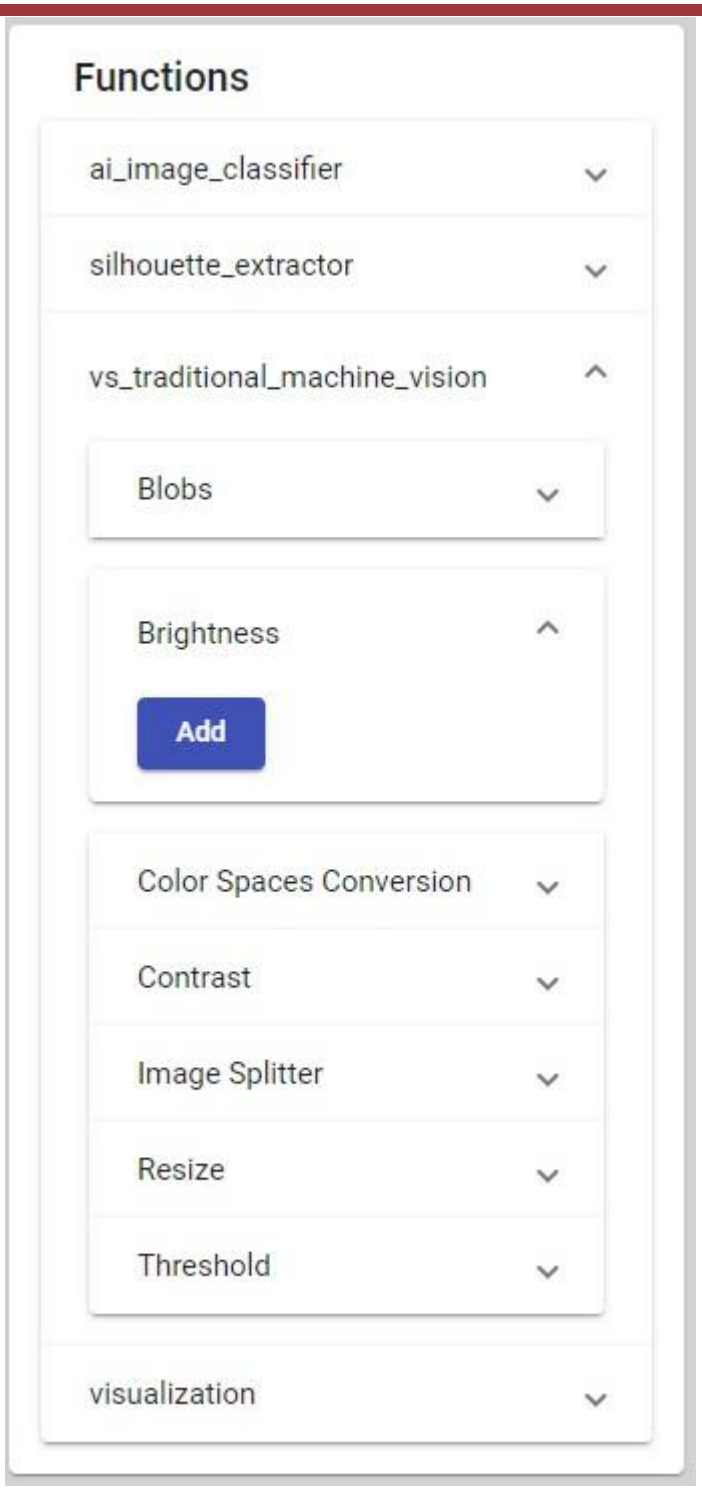
The Designer is a Web interface where a developer can visually build Computer Vision algorithms. This tool facilitates the development of an algorithm even if the user does not know much about Computer Vision. They can build a complex and specific flow using graphic debugging tools and the documentation of each module.

It is particularly useful for pre-processing purposes since there is an infinite number of cameras and sensors in the market, with various characteristics. The light condition is a very important factor too. This tool allows testing an algorithm with different images from different cameras and this way its easy to check if the algorithm is stable or tweak it as necessary.

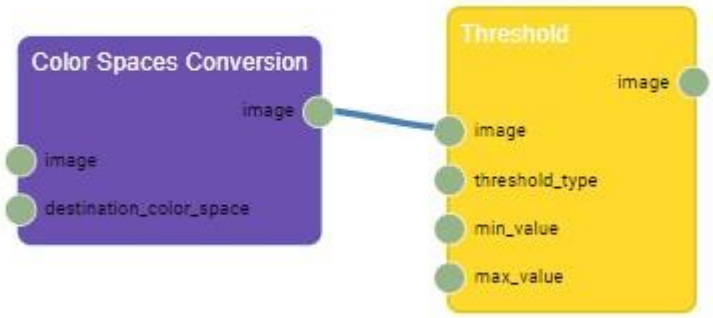


When the page is loaded, information about all the installed modules and their relative functions are downloaded from the API Server.

On the left side can be seen the list of functions grouped by modules. To add it to the flow, click on it and then on *Add*.

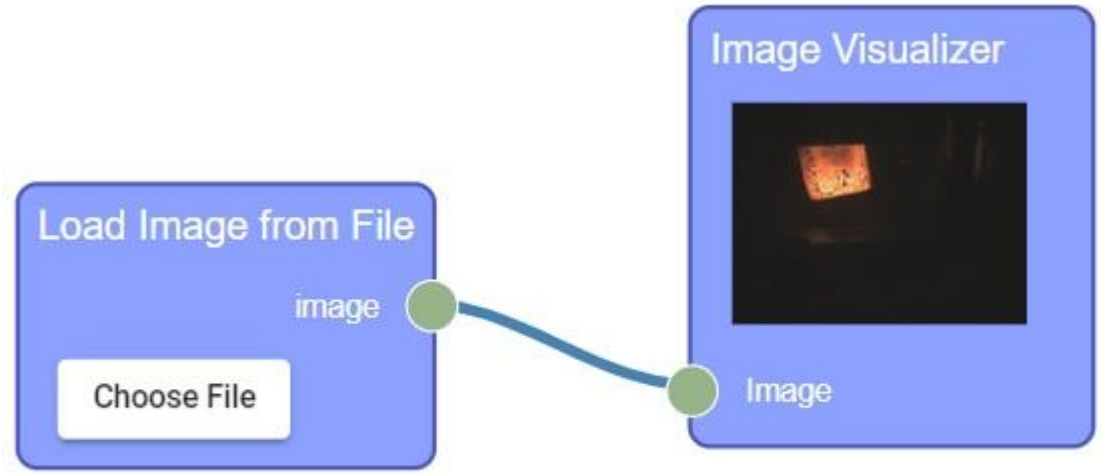


Every function has input and output parameters. An output parameter can be connected to an input parameter. There are several types of parameters: it can be an Image, an Integer, a File, etc. Only parameters that have the same type can be connected.



There are several functions used to debug a specific algorithm. For example, the "Load Image from File" function is used to load an Image from a user PC and connect the output Image parameter to another function.

The "Image Visualizer" function accepts an Image as an input parameter: when connected, it will show the relative image. This is useful for debugging purposes.

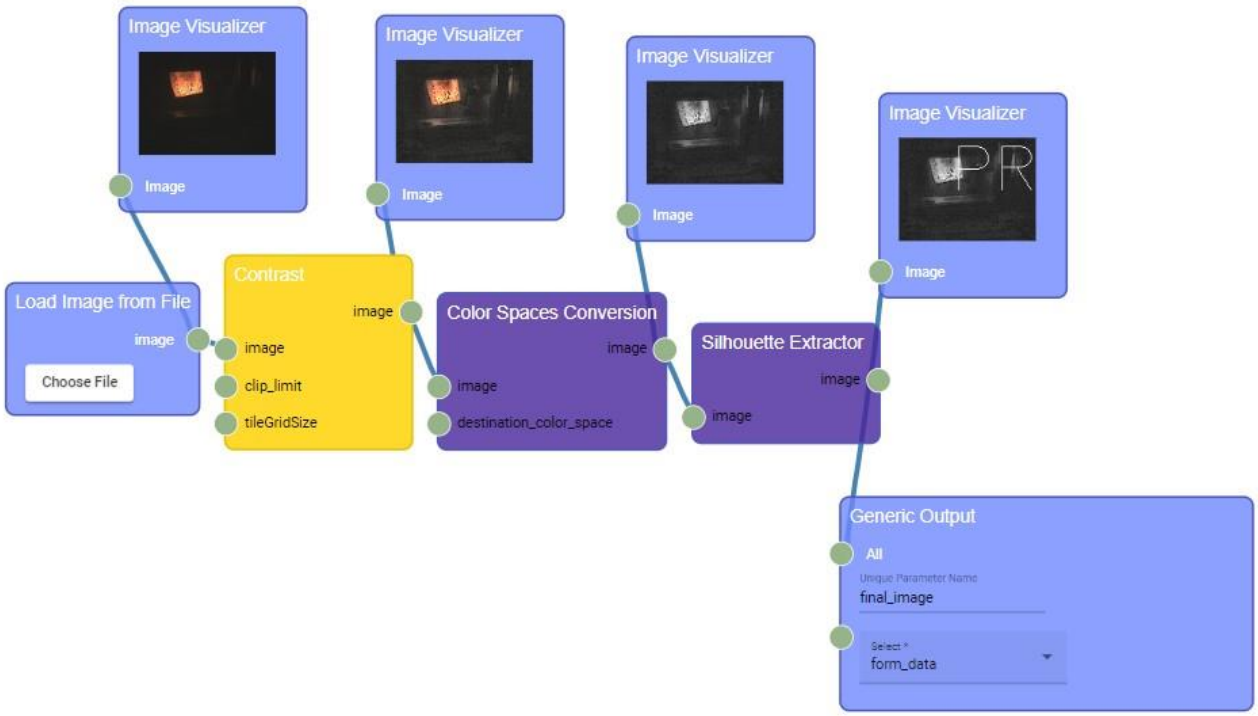


A complete description of all functions will be provided in later iterations.

Click on the *Export* button to download a .json file, and save this file to a local database, and then use it later during runtime to execute the flow.

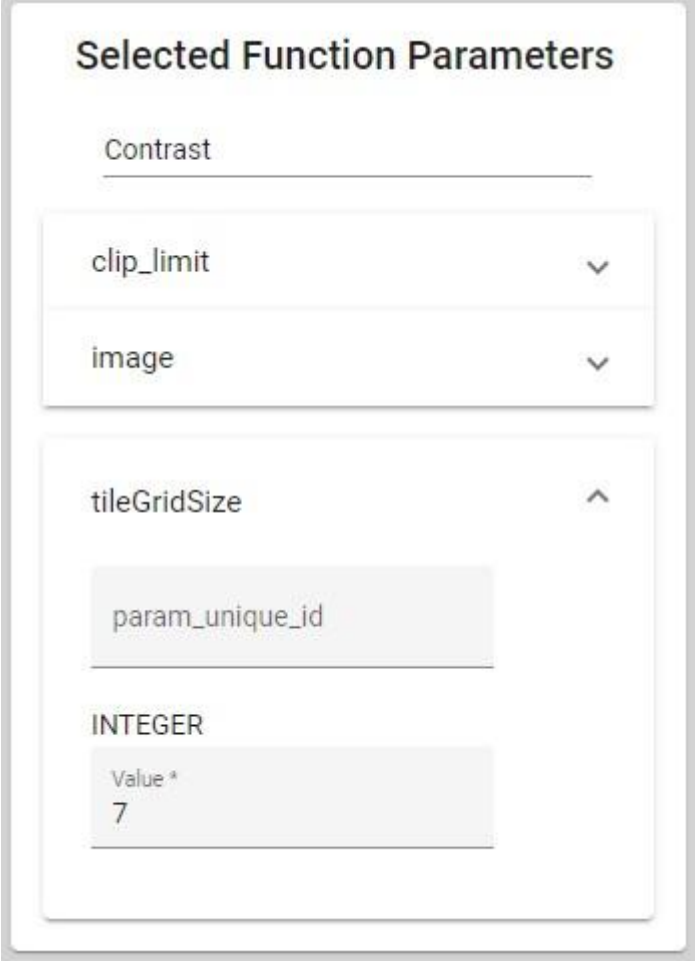
If it is need to later edit the saved flow, it is possible to click on *Import* and load .json file.

An example of a simple algorithm is now shown:



If necessary, the created algorithm from *files/export.json*. can be imported

When a function is selected, on the left side a screen like this will appear:



On the top, can be seen a text box, used to customize the name of the function. Then it is possible to see the list of input parameters of the selected function. Click on each parameter to be able to edit its value.

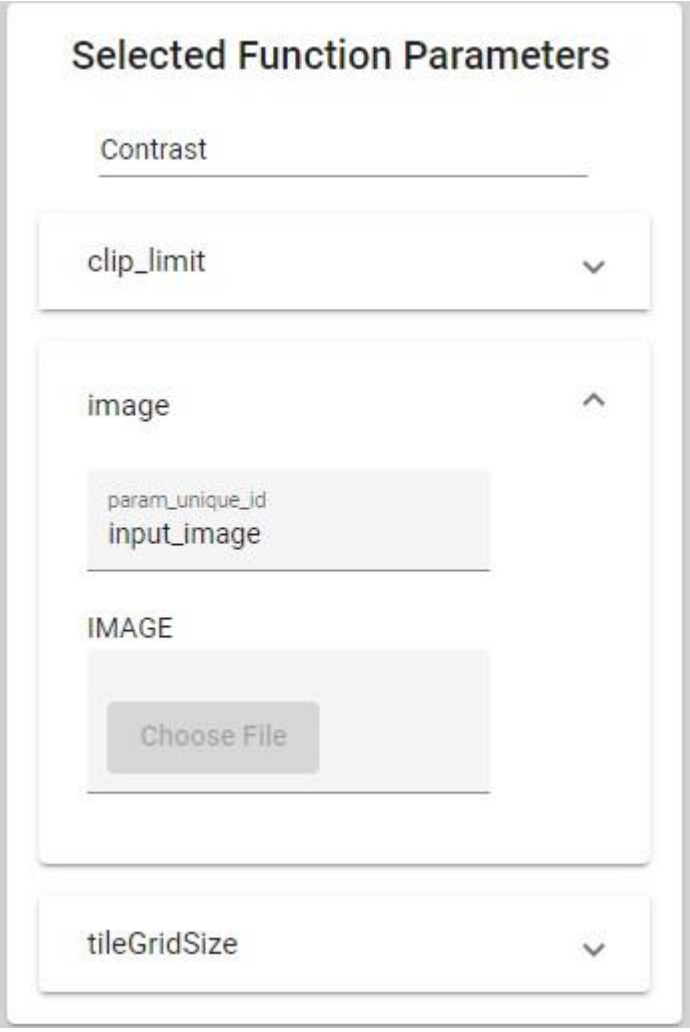
Before the value, can be seen the type of the parameter. Different input methods are available depending on the type of the parameter. If a parameter is connected to another, changing its value is not possible as it would not make sense. The parameters set in the Designer will be saved in the .json file which will be exported later.

Provision of some parameters at runtime may also be necessary.

The image that needs to be processed is of course one of those.

To mark a parameter as a runtime one, it should be given a unique ID.

Suppose that the input image is marked as a runtime parameter:

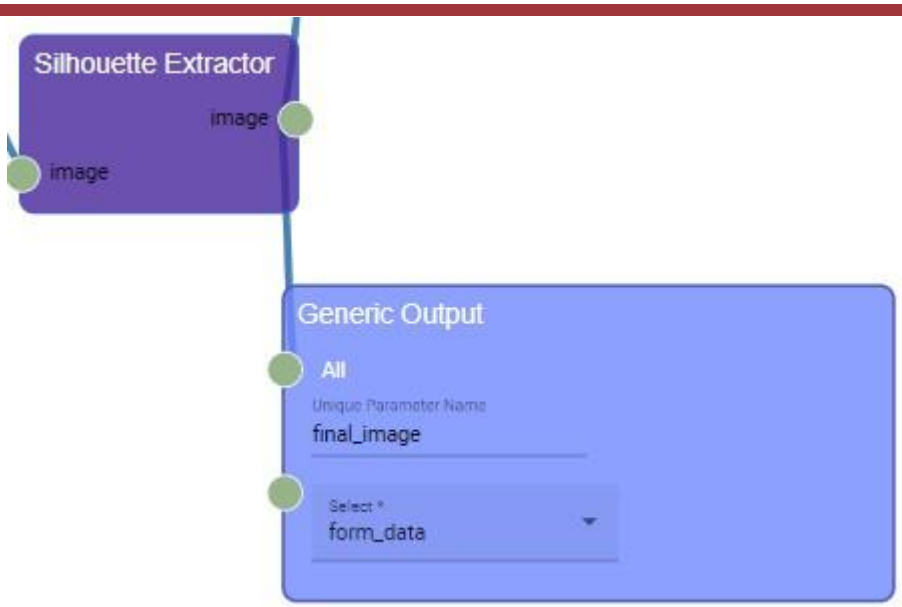


Here the *image* parameter is given the unique ID: *input_image*.

The same operation must be performed for the runtime parameters to be exported.

The results of an algorithm are needed, but also some other parameters for diagnostics might be needed.

To do this, connect the desired parameter to a *Generic Output* function; here the last image in the flow is marked as a runtime output parameter:



The unique ID *final_image* has been given to the parameter.

Now the result is a simple example of an algorithm. The recipe can be exported by clicking on the *Export* button.

To use it at runtime, another component or zApp must call the API Server, which is described below.

1.6.1.2 Backend

The Backend is the API Server of the generated component, accessible via API and the Service & Message Bus. So, it can be accessed by HTTP requests or the AMQP.

Starting with the HTTP APIs, two web pages can be accessed that can help to understand the APIs:

http://localhost:50051/docs/

http://localhost:50051/redoc/

Of course, needing to call the HTTP endpoints directly should not be performed and instead an OpenAPI generator is used. The .json OpenAPI definition can be retrieved here:

http://localhost:50051/openapi.json

For these examples, Python will be used to simulate some calls to the component. The code should not be very different in other languages.

Once generated the OpenAPI client code with a favourite generator, two APIs files should be noted:

```
flow_api.py
process_api.py
```

1.6.1.3 Flow API

The Flow API is used to execute the flow exported from the Designer. Its APIs allow executing multiple functions within one API call, to reduce useless traffic and serialization. When the Flow process is called, additional parameters may also need to be sent. This has already been explained in the Designer, where an input parameter can be given a unique ID.

Calling a Flow process is simple. The .json file is needed which can be exported from the Designer. Optionally, a list of input parameters, identified by the unique ID.

It should be remembered that in the Designer documentation, some output parameters could be connected to a function called "Generic Output", and the parameters could be given an ID.

The Flow process returns a list of those output parameters identified by the output runtime ID.

Below is an example to run a Flow process:

```
import openapi_client

# Defining the host is optional and defaults to http://localhost
# See configuration.py for a list of all supported configuration parameters.
configuration = openapi_client.Configuration(
    host="http://localhost:50051"
)

# Enter a context with an instance of the API client
with openapi_client.ApiClient(configuration) as api_client:
    # Create an instance of the API class
    api_instance = openapi_client.FlowApi(api_client)

    # Loading the exported file from the Designer
    with open("./export.json", "r") as f:
        flow_object = json.loads(f.read())
    # Loading and Serializing the input image
    with open("./sample_image.png", "rb") as image_file:
        encoded_string = base64.b64encode(image_file.read())

    # Adding the input image to the parameters (base64 encoding)
    param = openapi_client.Parameter(
        name="input_image",
        param_unique_id="input_image",
        type="IMAGE",
        value= "" + encoded_string.decode()
    )

    input_params = [param]

    # Creating the FlowRequest object
    flow_request = openapi_client.FlowRequest(
        flow_object=flow_object, input_params=input_params)

    # Calling the Flow
    flow_response = api_instance.run_flow_flow_post(flow_request=flow_request)

    # Iterating the output parameters
    i = 0
    for output_param in flow_response.output_params:
        if(output_param.type == "IMAGE"):
            # Saving all output images to disk
            msg = base64.b64decode(output_param.value)
            buf = io.BytesIO(msg)
            img = Image.open(buf)
            img.save("/tmp/res"+str(i)+".png")

            i = i + 1
```

All the modules' properties, as with the Parameters, are explained below.

1.6.1.4 Process API

As stated earlier, there are multiple modules; each one can contain multiple functions.

A list of these functions can be retried as can the list of default input and output parameters of each one. A function can be executed specifying the input parameters; that will return the processed output parameters. This is also the way the Designer works.

Then a Designer-like approach can be used from a user's code (without the Designer Web UI).

Follows is an example to retrieve the list of modules and functions:

```
import openapi_client

# Defining the host is optional and defaults to http://localhost
# See configuration.py for a list of all supported configuration parameters.
configuration = openapi_client.Configuration(
    host="http://localhost:50051"
)

# Enter a context with an instance of the API client
with openapi_client.ApiClient(configuration) as api_client:
    # Create an instance of the API class
    api_instance = openapi_client.ProcessApi(api_client)

    list_modules = api_instance.installed_modules_and_operations_process_installed_modules_post()
    for module_name, module_operations_list in list_modules.items():
        print(module_name + " : " + str(module_operations_list))
```

Here follows an example to get the default input and output parameters of a function:

```
operation_module = "vs_traditional_machine_vision"
operation_function = "traditional_machine_vision/base_image_processing/resizing"
io_params = api_instance.get_params_process_get_params_post(operation_module, operation_function)

print(io_params.input_params)
print(io_params.output_params)
```

Follows is an example to run a function:

```
operation_module = "vs_traditional_machine_vision"
operation_function = "traditional_machine_vision/base_image_processing/resizing"

# Getting the default IO parameters
io_params = api_instance.get_params_process_get_params_post(
    operation_module, operation_function)

input_params = io_params.input_params

# Getting the input image from file
with open("./easy.png", "rb") as image_file:
    encoded_string = base64.b64encode(image_file.read())

# Editing the default input_params' values
input_params["image"].value = encoded_string.decode()
input_params["scale"].value = 0.2

# Calling the Process
generic_request = openapi_client.GenericRequest(
    input_params=input_params, operation_module=operation_module, operation_id=operation_function)
generic_response = api_instance.run_process_process_post(
    generic_request=generic_request)

# Saving output image to disk
i = 0
for output_param_name, output_param_value in generic_response.output_params.items():
    print("Output Param Name: " + output_param_name)

    if (output_param_value.type == "IMAGE"):
        msg = base64.b64decode(output_param_value.value)
        buf = io.BytesIO(msg)
        img = Image.open(buf)
        img.save("/tmp/res"+str(i)+".png")

        i = i+1
    else:
        print(str(output_param_value.value))
```

This way single functions can be executed.

However, when it is needed to execute multiple functions in sequence, this method is very slow: the data needs to be sent back and forth between the application and the server. The Flow APIs are always recommended when there is a need to call multiple functions.

1.6.1.5 Models description

As shown here, all APIs return a list or a dictionary of *Parameters*. It is not important to show all the different request or response models, because all of them contain this *Parameter* schema:

Field	Mandatory	Type	Description
name	true	string	The name of the parameter. This is unique in the list of input or output parameters of a function.
value	true	any	The value of the parameter. This depends on the type of the parameter itself.
type	true	enum	The type of the parameter.

Field	Mandatory	Type	Description
			allowed_values = ["ALL", "NONE", "INTEGER", "LIST_INTEGER", "FLOAT", "LIST_FLOAT", "STRING", "LIST_STRINGS", "IMAGE", "LIST_IMAGES", "FILE", "LIST_FILES", "BOOLEAN"]
min	false	float	The minimum value allowed for the parameter. Used for INTEGER and FLOAT types.
max	false	float	The maximum value allowed for the parameter. Used for INTEGER and FLOAT types.
step	false	float	The min step value allowed for the parameter. Used for INTEGER and FLOAT types.
param_unique_id	false	string	Unique ID used for overriding a parameter when calling the Flow APIs or used for exporting a parameter after a Flow API call.
file_type	false		Internal use
additional_params	false		Internal use
File_	false		Internal use

As can be seen, the *type* field indicated the type of the parameter. If a parameter has a type "INTEGER", its value will be an integer, if a "FLOAT", its value will be a floating-point number, etc.

It is important to focus on the "IMAGE" and "FILE" types whose value will be a base64 encoded string. The output image encoding will always be a PNG base64 encoded image; PNG is a lossless compression format. It is recommended to use it for input images as well. From the examples above it is clear how to convert an IMAGE to a base64 string and the other way around. The same thing can be performed on any a FILE.

1.6.2 Integration with Other Components

The Builder component may get modules from the Marketplace. Modules, in fact, should be uploaded to the Marketplace and should only be eligible to download if a user has bought it.

When a developer wishes to setup the Builder component, they should be the only one working on that, because they can upload their custom modules or download them from the Marketplace. So, it should be a private instance that only they can access.

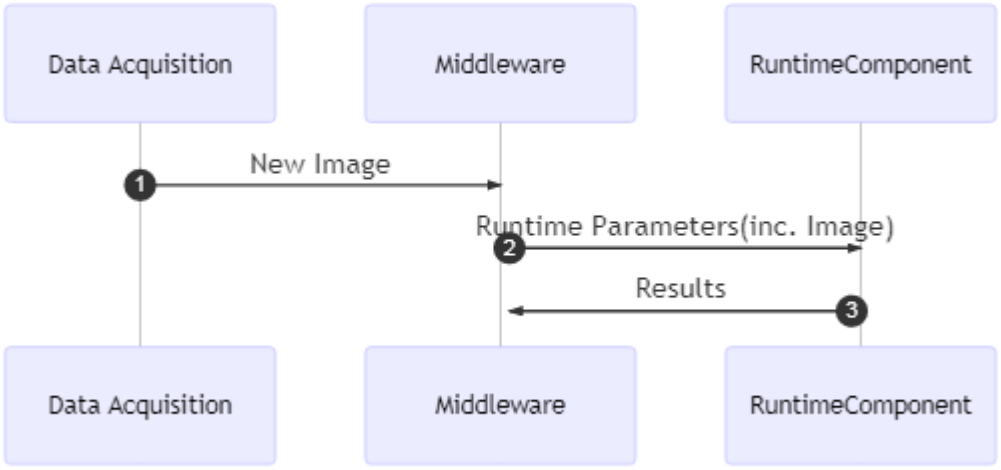
The Builder component generates a dynamic Runtime component; this needs to be built and started by the Application Runtime. This contains custom modules, so the platform should make sure the Runtime component is accessible only from that user or zApp instance.

The Service & Message Bus is needed for the communication between the Runtime component and a zApp or another component.

The Data Acquisition component can send runtime data to be elaborated to this component. As described before, there can be some runtime parameters; however, the Data Acquisition component does not have these parameters because usually they depend on the product or a custom configuration.

The advice is to write a custom program that stays in the middle. It could receive real time data from the Data Acquisition component, and then call the Runtime component with custom parameters. This can be a standalone component or can be something integrated in the zApp directly.

This diagram describes this concept:



1.6.3 Writing a Module

A module is a Python package which contains a list of functions. To be able to expose all the functions to the Designer and Runtime component correctly there are some rules.

Some Computer Vision algorithms are very fast and require only a few dependencies. Neural Network based algorithms, however, are heavy and require custom dependencies, Self-Organizing Structure/Space, and computational resources. That is why there are two types of modules.

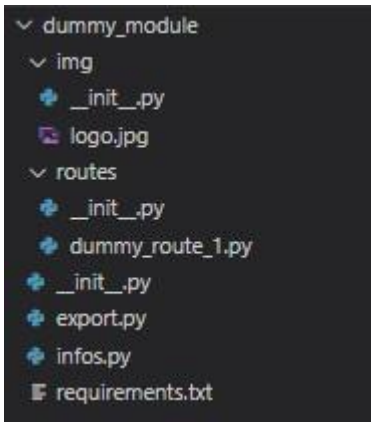
The first one is the simpler as the functions are executed directly on the Backend. A list of Python packages to be installed is provided

The second one is more complex where one or more Docker containers can be created. These will be started with the Runtime component. When a function is called, the Backend routes the request to the external image, which will execute the function. Then the result will be returned to the Backend. This is a little slower because the data needs to be sent back and forth between the two containers, but this is the only solution.

One example for each type of module is provided.

1.6.3.1 Simple Module

A dummy module is provided: *files/dummy_module.zip*. This contains a simple example of a module that can be used as a mock-up. Once extracted it presents:



It is possible to see a folder containing some files and subfolders. Its name is the ID of the module, in this case *dummy_module*. Remember that modules' IDs are unique - it is thus recommended to put the name of a company to avoid collisions.

The *requirements.txt* file contains the list of all the dependencies to be installed.

The *infos.py* file contains some information about the module. It is necessary to replace the module_id with the desired one; the rest is straightforward.

```
INFOS = {
    "module_id": "dummy_module",
    "version": "1.0",
    "name": "Dummy Module",
    "description": "This is a dummy module that shows you how a module works.",
    "thumbnail": "img/logo.jpg"
}
```

The *export.py* file contains the routing logic:

```
from .routes import dummy_endpoint_1

ARE_THERE_EXTERNAL_IMAGES = False

EXPORT_ROUTES = {
    dummy_endpoint_1.OPERATION_ID: dummy_endpoint_1.Process
}

EXPORT_ROUTES_PARAMETERS = {
    dummy_endpoint_1.OPERATION_ID: dummy_endpoint_1.GetParams
}

EXPORT_ROUTES_INFOS = {
    dummy_endpoint_1.OPERATION_ID: dummy_endpoint_1.INFOS
}
```

First it can be seen that *ARE_THERE_EXTERNAL_IMAGES = False*, appears as discussed previously.

Then some dictionaries can be noted; these are needed to map an OPERATION_ID to a function:

- EXPORT_ROUTES: dictionary that maps the OPERATION_ID to a processing function
- EXPORT_ROUTES_PARAMETERS: dictionary that maps the OPERATION_ID to a function that returns the default input and output parameters
- EXPORT_ROUTES_INFOS: dictionary that maps the OPERATION_ID to an object that contains some information about the function.

It is possible to create multiple functions with a unique OPERATION_ID and then add those to these dictionaries.

Following explains how to write a function. In the examples, all the functions are defined inside the *routes* folder: to make a module cleaner, it is recommended to do that too.

The *routes/dummy_endpoint_1.py* is split into pieces to explain the code.

This first part is straightforward:

```
# Unique ID of the function inside the module.
OPERATION_ID = "dummy_endpoint_1"

# Contains some infos about this function.
INFOS = {
    "name": "Dummy Endpoint 1",
    "description": "The Dummy Endpoint 1 does nothing."
}
```

The following shows the GetParams method. This returns two dictionaries, the input and output parameters with their default value. These are the ones shown in the Designer, which defines all the function's possible connections to other components.

In this example, there is an IMAGE input parameter and an IMAGE output parameter.

```
async def GetParams():
    '''
    This method returns a dictionary containing the input
    parameters and the output ones with their default value.
    '''

    input_parameters = defaultdict(dict)
    input_parameters["image"] = {
        "name": "image",
        "value": 0,
        "type": LogicalConnectionEnum.IMAGE,
        "min": 0,
        "max": 0,
        "step": 0
    }
    output_parameters = defaultdict(dict)
    output_parameters["image"] = {
        "name": "image",
        "value": 0,
        "type": LogicalConnectionEnum.IMAGE,
        "min": 0,
        "max": 0,
        "step": 0
    }
    return input_parameters, output_parameters
```

Here follows an example of a Process function. The input of this function must be the input_params dictionary, and it must return the output_params dictionary.

```python
async def Process(input_params):
    '''
    Contains the function's processing logic.
    From the input parameters, this method runs the algorithm and
    returns the output parameters.
    '''

    # Getting an input parameter
    img = input_params["image"]["value"]

    # PROCESSING
    # Here you will implement your own logic.
    cv2.putText(img, "PROCESSED", (30, 30), cv2.FONT_HERSHEY_SIMPLEX, 2, 255)

    # Building output_params dictionary
    _, output_params = await GetParams()
    output_params["image"]["value"] = img

    # Returning the processed output parameters
    return output_params
```

Please note these functions have the *async* keyword.

It is important to specify parameters at this stage are not formatted as seen previously in Section 1.6.1.5. There it was explained how to serialize parameters for an API call. Here parameters are ready to be used. The following table describes how parameters are formatted:

LogicalConnectionEnum	Python type
INTEGER	int
LIST_INTEGER	int[]
FLOAT	float
LIST_FLOAT	float[]
STRING	str
LIST_STRINGS	str[]
IMAGE	cv2 image
LIST_IMAGES	Cv2 image[]
FILE	str (file path)
LIST_FILES	str[] (file paths)
BOOLEAN	bool

The most important are the IMAGEs and FILEs parameters; the first one is an OpenCV image and the second one is a string that represents the file's path.

In the *Process* function above can be see an example of an IMAGE parameters: an input IMAGE, which is an OpenCV image, is processed using an OpenCV method. The same OpenCV image is returned as an output parameter.

1.6.3.2 Module with External Image

It is possible to find this dummy module here:
files/dummy_module_with_external_image.zip.

The structure of the module is similar to the previous one. Of course, the module's ID is different and in the *export.py* module the *ARE_THERE_EXTERNAL_IMAGES* variable is set:

```python
ARE_THERE_EXTERNAL_IMAGES = True
```

In the *routes/dummy_route_1.py* there is the logic to call an external image because the processing does not run in the Backend.

The *images/* folder contains a list of subfolders, which are Docker images to be built and run at Runtime. Only one image is visible: *dummy_external_image*.

An image must have:

- Dockerfile
- compose.yaml

The contents of the *compose.yaml* file will be copied to the final *docker-compose.yml* file, such as the one below. Open ports, set a restart policy, set environment variables, etc.

```
version: "3"
services:
  dummy_external_image:
    #
    #   compose.yaml is
    #   this object
    #
```

The Docker image will be built and started at Runtime and a communication can be created between a Backend function and this Docker image. Any IPC protocol can be used, and it is not important which.

A RabbitMQ server instance, however, is present on the Backend as the *dummy_module_with_external_image* module uses that to communicate.

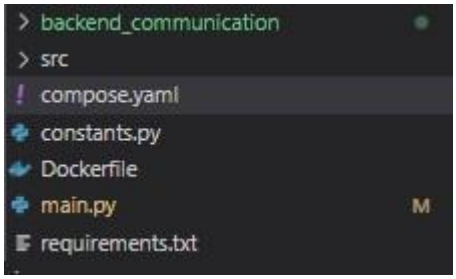
If there are no specific needs, it is recommended to also use RabbitMQ. The *dummy_module_with_external_image* shows how to implement the communication from the Backend and the Docker image, so this sample module can be used as-is.

That is why the *dummy_module_with_external_image* is further explained.

Of course, if it is necessary to write a script in another programming language, it will also be necessary to port the communication code in the specific language, although it is not so difficult.

The *dummy_module_with_external_image* is a Python script that listens to messages in a RabbitMQ queue, it processes the message, and sends the results.

This creates an RPC-like communication; more can be learnt about this in the RabbitMQ documentation.



The *backend_communication/* folder contains the communication logic from and to the Backend component. Using RabbitMQ queues specify the routing key. The *constants.py* file contains this routing key. Of course, the routing key must be unique to avoid listening and writing to other queues. Make sure it is unique by using the module's ID.

```
MODULE_NAME = "dummy_module_with_external_image"
ROUTING_KEY = MODULE_NAME
```

In the *main.py* file there is the initialization of the RabbitMQ channel:

```
from backend_communication import rabbitmq_server, rabbitmq_parameter_serializer

def CustomCallback(body):
    req_obj = jsons.loads(body.decode())
    # etc.

rabbitmq_server.custom_callback = CustomCallback
rabbitmq_server.Start()
```

In the Backend implement the call logic. In the *dummy_route_1.py* file it is shown how it is made:

```
from app.rabbitmq_client import rabbit_instance

ROUTING_KEY = INFOS["module_id"]

# Serializing parameters for RabbitMQ
input_params = rabbitmq_parameter_serializer.SerializeParameters(
    input_params)

# Calling the external image
obj = defaultdict(dict)
obj["request"] = "process"
obj["input_params"] = input_params
output_params = await rabbit_instance.Call(data=jsons.dumps(obj), timeout=3000, routing_key=ROUTING_KEY)
output_params = jsons.loads(output_params)

# Deserializing parameters from RabbitMQ
output_params = rabbitmq_parameter_serializer.DeserializeParameters(
    output_params)
```

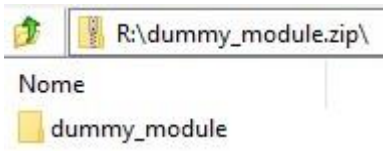
The routing key must be the same for the communication to work.

Set the timeout time in milliseconds: if the response does not arrive in the specified period, the method will return.

It is possible to send whatever is needed to the Docker image. The important thing is that the output_params must be returned to the dictionary for the Process and the input_params, output_params tuple for the GetParams function, exactly as in the simple module described before.

1.6.3.3 Uploading a Module

The module's folder needs to be zipped, which looks like this:



Upload the zip file to the Builder component, and once done correctly it should be possible to see the module appear in the Builder's modules list.

It can then be enabled to build the Runtime component.

Once the Runtime component is run, the module and its functions will be available in the Designer.

3DScan

Once in the CLI, a quick check of the dummy file to simplify and its properties can be made:

```
cd data
ls
stat HorseHead_simp.stl
```

As can be guessed by its size (~600 Mb) it has already been simplified not to copy a 300 Mb file to the container to re-simplify this STL file,

```
cd ../app
python3 app.py
```

And check the output:

```
cd../data
stat HorseHead_simp2.stl
```

1.7 Functional Requirements Implementation Status (M18)

The actual implementation status vis-à-vis the functional requirements implementation at M18 is provided in the annex of the D006 Technical Management Overview Report. This represents the general software status of the project and this WP/Task including information on commits and WP5-8 Risks and mitigations. Below is shown a dummy example for a security component.

Functional requirement	Description	Status	Progress	Comments
T52A013 - Issue New certificates	New client certificates are created. These certificates include the details that permit the identification of the subject (physical device, gateway, or server).	Working	90%	Beta version, requires integration API with security command centre for credentials tokenization

2 Conclusions

This deliverable is the first deliverable in the reporting series for T8.3 Production: Non-Destructive Product Inspection. The deliverables for this task, and all WP5-8 tasks, are software and are of EU type “OTHER”. The software and accompanying material (eg description, instructions) is available on the ZDMP software repository which is updated dynamically. However, for EU formal reporting purposes, this brief cover document provides a formalised pointer to the downloadable software and related content.

This deliverable should read in conjunction with the D006-D020 deliverables which document the software process/status for each WP/Task vs its content. This deliverable represents the status as at M18 with further living editions at M18 and M48 and an informal iteration at M24.

ZERO DEFECTS
Manufacturing
Platform
ZDMP

www.zdmp.eu