HORIZON 2020
The EU Framework Programme for Research and Innovation

CURE Application Solution Brochure

Deliverable D4.4

DATE
30 June 2022

ISSUE
1.0

GRANT AGREEMENT
no 870337

DISSEMINATION LEVEL
PU

PROJECT WEB-SITE
http://cure-copernicus.eu/

AUTHORS
Johannes Schmid (GeoVille)
Samuel Carraro (GeoVille)
Mario Dohr (GeoVille)

CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CCSI	Copernicus Core Service Interface
DAG	Directed Acyclic Graph
GIS	Geographic Information Systems
GEMS	GeoVille MicroService
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
REST	Representational State Transfer
REGEX	REGular EXpression
SQL	Structured Query Language
Stdout	standard output
VM	Virtual Machine
WP	Work Package

1 Introduction

1.1 Purpose of the document

Deliverable 4.4 is an essential delivery of Task 4.4 Integration of Services in WP4 Cure System Development.

CURE Application Solution Brochure serves as an introduction on how to implement a new application into the CURE system. It consists of two main parts, the creation and the incorporation of applications into the GeoVille MicroService (GEMS) architecture.

2 Implementation of CURE Services

2.1 Service-ready Applications

Before a new service can be implemented, it needs to fulfill several requirements. This includes requirements on the code, its detailed documentation, version control and a container virtualization.

2.2 Application-Code

Whether the application is written in Python, C++, Java, Julia or another language, the code needs to be style conform. Most languages have their own style guide, such as PEP8 for Python. Those guides ensure a nice looking and readable code structure.

Moreover, a lot of comments need to be inserted so that everyone else can easily understand the application steps for further improvements or debugging. This also includes logs to stdout. The more logs, the easier it is to find unexpected behavior of the code and monitor the current processing status.

Besides code styling and logs, it is also important to include command line arguments. This enables the user to execute the code while being able to easily change important variables such as paths to input files, output directories or other configuration values. Predefined folder and file structures might cause problems when combining the application code with other components such as the input data download using the CCSI API.

Ideally, the code was developed test-driven. Hence, unit-tests guarantee the absence of errors and verify that new features and changes do not lead to errors elsewhere.

Even if the code needs to be compiled before it can be run, binary data should not be delivered. The compilation will be done by Docker (see Chapter "Container Virtualization").

Now that the application is well structured, includes a sufficient number of logs and is tested, it still needs to be documented outside of the code. Therefore, a README file by using

Markdown is recommended. First and foremost, this file needs to include the detailed description of the service and the execution command with all its parameters. An example command is also very helpful.

2.2.1 Container Virtualization

Only because the application code runs at the developer's environment, it does not necessarily run on any other machine. Since the code needs to run on a virtual machine that serves as an Airflow Worker, the entire environment with all its modules and requirements needs to be installed. To avoid this procedure and make the code system agnostic, a Docker container can be built.

The first step is to create a Dockerfile. This requires special knowledge of the Docker syntax. Although creating a working example might seem easy in the beginning, it is more demanding to take care that the resulting Docker image is not too big. There are slim base images that can be used as a foundation for the newly created Docker image.

The second step is building the Docker image using the Dockerfile. This can be done by using the following command:

```
docker build -t <name_of_the_new_docker_image> .
```

Besides the Docker image name that needs to be provided with the flag "-t", the location of the Dockerfile is required. In case of the example, the Dockerfile lies at the present working directory (.).

After the Docker image was successfully built, the execution of the code can be tested to ensure that all requirements were considered and everything runs as expected. As soon as a Docker image is used to execute code, a temporary Docker container gets created in which the code actively runs.

In case no entry point was defined within the Dockerfile, the command to run a Docker image as a Docker Container can look as follows.

```
docker run <name_of_the_new_docker_image> python3 main.py
```

In this example, a Python script called "main.py" gets executed by using the Docker image. As soon as this command gets executed, the respective Docker container is created and runs the code.

However, this command has many command line arguments such as adding volumes that mount local directories into the container or the definition of environment variables. The following example call shows how a local directory can be mirrored within the container by using the "-v" flag. As a result, this mounted directory can be used as an input for the script main.py.

Imagine a local directory (local_directory) that includes a GeoTiff called "input.tif". If the directory gets mounted into the container (container_directory), it can be used as input data for the application inside the container.

```
docker run -v /local_directory:/ container_directory <name_of_the_new_docker_image> python3 main.py -i /container_directory/input.tif
```

As a summary, the following figure shows the entire Docker process, from creating a Dockerfile to a running Docker Container.

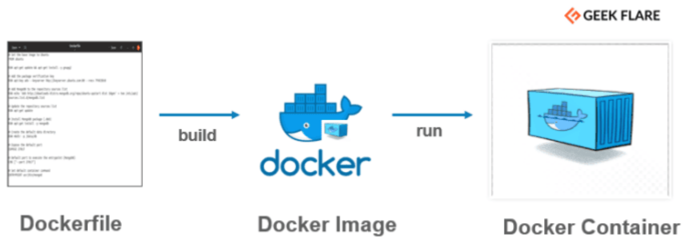

Figure 2-1 Docker Process

Both, the Dockerfile and the Docker image can be shared. While the Dockerfile can be simply provided alongside the application code, the Docker image can be uploaded to a Docker registry. The most common and free-to-use registry is the Dockerhub.

Finally, the README file mentioned in Chapter "Application-Code" should also include an example docker run command in case there are special requirements of the code such as environment variables.

2.2.2 Version Control

Primarily, a version control system manages code changes. This means that changes can be monitored, revoked and tagged. Hence, the code can have several versions. The most famous version control system is called "Git" which is used by several user-friendly web-based providers such as GitHub, GitLab or Bitbucket.

The app developer can use a provider of his or her choice to create a repository and store the application code and the Dockerfile inside.

Most providers also offer pipelines that enable the repository owner to execute commands every time a new commit gets pushed.

Pipelines can include automatic docker builds and uploads to an online docker registry, the execution of unit-tests or the analysis of the code quality to control new commits.

Now that the code is well documented, tested, dockerized and finally stored and managed within a Git repository, the link to the repository can be shared with the engineers who will incorporate the service into the microservice architecture.

3 INCORPORATION INTO GEMS

The incorporation starts by ordering and setting up a virtual machine as an Airflow Worker machine. Furthermore, a Directed Acyclic Graph (DAG) needs to be created for the Airflow Scheduler, since the service does not only consist of the application code alone but also of the input data download and the output data handling. Finally, an API endpoint needs to be created to allow service orders by customers.

3.1 VM setup

In the beginning of the CURE project, it was decided that WEkEO is the DIAS platform that shall be used for ordering virtual machines on which the applications will run. The machines should be based on an actual Ubuntu distribution. The only software that needs to be installed is Docker. As a result, the dockerized Airflow worker service can be run daemonized as a Docker container. Moreover, Docker is required to build or pull the Docker Images of the CURE applications. It is highly recommended that each service or application gets its own worker machine.

3.2 Application-Workflow

Apache Airflow supports the creation of workflows. A workflow is formulated as a Directed Acyclic Graph (DAG). Usually, a DAG is a collection of tasks and each DAG is represented by a Python script.

In the following Figure, an example DAG of a CURE application is visually presented. Each rectangle represents a task which can be either a Python function, a Bash command or a Docker container.

In case an application is more complex and includes more than just the data download using the CCSI API in the beginning and a product upload in the end, the application developers need to provide a flowchart that visualizes the relationships between the different tasks.

Moreover, defining the correct CCSI requests needs a lot of collaboration between the CCSI developer, the application developer and the software engineer who creates the DAG to ensure that the correct data gets downloaded.

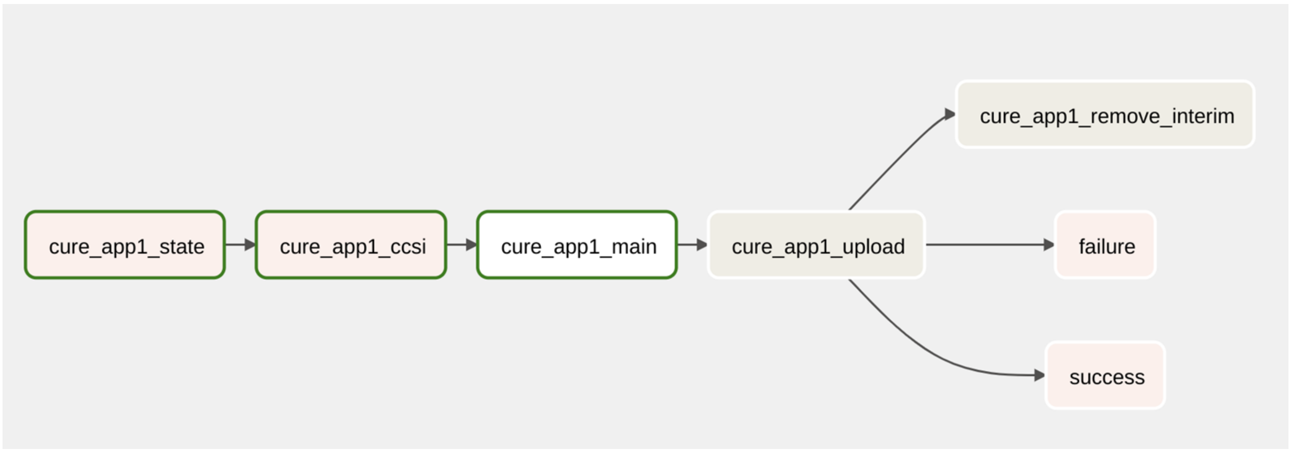

Figure 3-1 Example DAG

3.3 API Endpoint

In conclusion, the new service needs to get an API endpoint so that customers can request the respective application at the CURE portal. As soon as a service request has been sent successfully, the API triggers the Airflow DAG and the code runs asynchronously in the background. Hence, the customer does not need to wait for an immediate response and get notified as soon as the requested product is available. As part of the endpoint creation, a REGEX check can be integrated to validate the request payload. This might require further information from the application developers. To give an example, a user cannot request a date that lies in the future.

Finished with the implementation of a new CURE service, the entire integration can be tested before it deployed in production.

The main reason why these system components have been dockerized is to enable simple reusability and scaling. When a system component needs to be migrated to another virtual machine or when another Airflow worker needs to be deployed, the Docker image can be reused instead of spending hours on installation and configuration. This can be done by either pulling the Docker image from a private or public Docker registry or by building the Docker image locally.

Another big advantage of Docker is the independence from system updates. While the upgrade of a program or module by a system update could cause dependency problems if the system component runs as a daemonized system service, the installation and execution within a Docker container is not affected.

4 CONCLUSION

This document serves as a guide how new services can be implemented into to CURE System. It must be considered that, given the variety of Applications, and their possible special need for input data, a close cooperation with the CCSI developers could be needed to guarantee a bug free performance.