URBANITE

Supporting the decision-making in urban transformation with the use of disruptive technologies

Deliverable D5.6

URBANITE DevOps infrastructure

Editor(s):	Iñaki Etxaniz, Gorka Benguría
Responsible Partner:	Tecnalia
Status-Version:	Final – v1.0
Date:	31.03.2021

Distribution level (CO, PU):	PU
Project Number:	GA 870338
Project Title:	URBANITE

Title of Deliverable:	URBANITE DevOps infraestructura
Due Date of Delivery to the EC:	31/03/2021

Work package responsible for the Deliverable:	WP5 - URBANITE ecosystem integration and DevOps
Editor(s):	Tecnalia
Contributor(s):	Iñaki Etxaniz (Tecnalia), Gorka Benguría (Tecnalia)
Reviewer(s):	Giuseppe Ciulla (ENG)
Approved by:	All Partners
Recommended/mandatory readers:	Mandatory: WP3, WP4 Recommended: WP6

Abstract:	This document describes the deliverable D5.6, that is composed by an infrastructure and a set of software tools and procedures to support the continuous integration, delivery and deployment in the project.
Keyword List:	Development environment, integration, testing, deployment, devops, container, git.
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/ The document itself is delivered as a description for the European Commission about the released software, so it is not public.
Disclaimer	This document reflects only the author's views and neither Agency nor the Commission are responsible for any use that may be made of the information contained therein

Document Description

Document Revision History

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	15/01/2021	Table of contents defined	Tecnalia
v0.2	25/01/21	First draft version	Tecnalia
v0.3	10/02/21	Docker section added	Tecnalia
V0.4	15/02/21	DevOps procedures added	Tecnalia
v0.5	26/02/21	Conclusion and Executive Summary	Tecnalia
V0.6	10/03/21	GitLab CI/CD updated, ready for internal review	Tecnalia
V0.7	17/03/21	Internally reviewed	Engineering
V1.0	30/03/2021	Ready for submission	Tecnalia

Table of Contents

List of Figures

List of Tables

Terms and Abbreviations

CI/CD	Continuous Integration/Continuous Deployment
CLI	*Command-line interface*
DevOps	Development and Operation
DNS	*Domain Name System*
DoW	Description of Work
EC	European Commission
GUI	Graphical User Interface
KR	Key Result
QA	Quality Assurance
SCM	Source Code Management
Sw	Software
URL	Uniform Resource Locator
WP	Work Package

Executive Summary

The main key results of URBANITE are software-based components. To implement and manage these components and construct the URBANITE ecosystem, a DevOps strategy and infrastructure have been deployed. The strategy was described in a previous deliverable (D5.3). The infrastructure that implements this strategy is the deliverable D5.6, which is described in this companion document.

The document is delivered as a description of the released software infrastructure for the EU Commission. Apart from that, it is intended that this document facilitates the partner's comprehension of the provided infrastructure and its use, and hence the work in the technical work packages, during the development (WP2, WP3, WP4), integration, deployment (WP5) and validation (WP6) in URBANITE. This deliverable is the result of Task 5.3 - Continuous Integration and DevOps approach.

First, the project DevOps infrastructure that will be used for managing the development process is presented, describing in detail the proposed supporting tools/technologies. The GitLab, provided by Tecnalia, will be used as a version control system, hosting both private and public repositories. A microservice approach will be applied to integrate the outcomes of the different development teams, prioritizing containerization technologies whenever possible. Docker will be used as containerization technology for running the components. Besides, whenever possible an applicable docker container will also take care of the building of the component. Container orchestrations technologies will be used to specify how the containers are configured, including details such as networking, storage, and scalability. The technology that we use for the orchestration is docker-compose, as it requires less infrastructure and other technologies could be adopted in the future such as Kubernetes if the pilots or the project evolution requires to do so. Finally, GitLab CI/CD will be used to run the continuous integration scripts that automate the deployment of some of the environments and their transitions. It greatly integrates with the version control system (GitLab) and the delegation of the compiling of the components to docker technology simplifies the integration making GitLab a practical and powerful solution.

Next, the procedures related to the DevOps approach are presented. These procedures are organized in two categories operation procedures and maintenance procedures. The operation procedures cover the interactions to generate, debug, test and improve the URBANITE platform from the contribution of the different development teams. The maintenance procedures cover the activities to deploy and maintain the DevOps platform in a healthy state.

The most important results of the DevOps methodology and integration tasks in URBANITE will be the DevOps infrastructure (to be released in M12), described in this document and the sequential-iterative URBANITE Ecosystem (M15 and following).

1 Introduction

1.1 About this deliverable

This report is a companion document of the deliverable *D5.6 – URBANITE DevOps infrastructure*, that is composed by infrastructure, a set of software tools and procedures to support the continuous integration, delivery and deployment in the project. The document itself is delivered as a description of the released software for the European Commission.

The strategy to be followed in the integration of URBANITE ecosystem was already described in deliverable D5.3 [1]. The current document explains how this strategy has been finally implemented, provides more details about the infrastructure, which tools are being used, how they are configured and managed, and the DevOps procedures used by the operators during the integration.

This deliverable is a result of *Task 5.3 - Continuous Integration and DevOps approach.*

1.2 Document structure

The document is organized into three (3) main sections plus a conclusion chapter, with the first section presenting the deliverable's objective and structure.

The second section -DevOps Framework- describes the DevOps infrastructure that will be used to support the software implementations in URBANITE, including the mechanism used to continuously build, run, test and deploy the URBANITE platform from the items developed by the different teams in the URBANITE project, the tools adopted and the technologies used.

The third section –DevOps Procedures- is devoted to the common activities carried out by the users over the infrastructure, to integrate the different components of the URBANITE solution, to dispose of the result to the pilots, to manage the feedback from the integration, etc. It will be split into operational procedures and maintenance procedures. The operation procedures cover the interactions to generate, debug, test and improve the URBANITE platform from the contribution of the different development teams. The maintenance procedures cover the activities to deploy and maintain the DevOps platform in a healthy state.

Finally, the conclusion section resumes the most relevant points of the document. The document ends with the references and appendixes.

2 DevOps framework

The DevOps approach in URBANITE consists of several stages an application goes through, from development to integration, delivery and production. DevOps integrates development and operations, incorporating practices such as continuous delivery, continuous integration, and collaboration.

The DevOps framework refers to the infrastructure and tools to be used internally to follow this DevOps approach and finally ensure the successful integration of the different components of the URBANITE solution.

In URBANITE, the DevOps approach will be structured in three environments, as depicted in figure 1. These environments are development, integration and production or piloting.

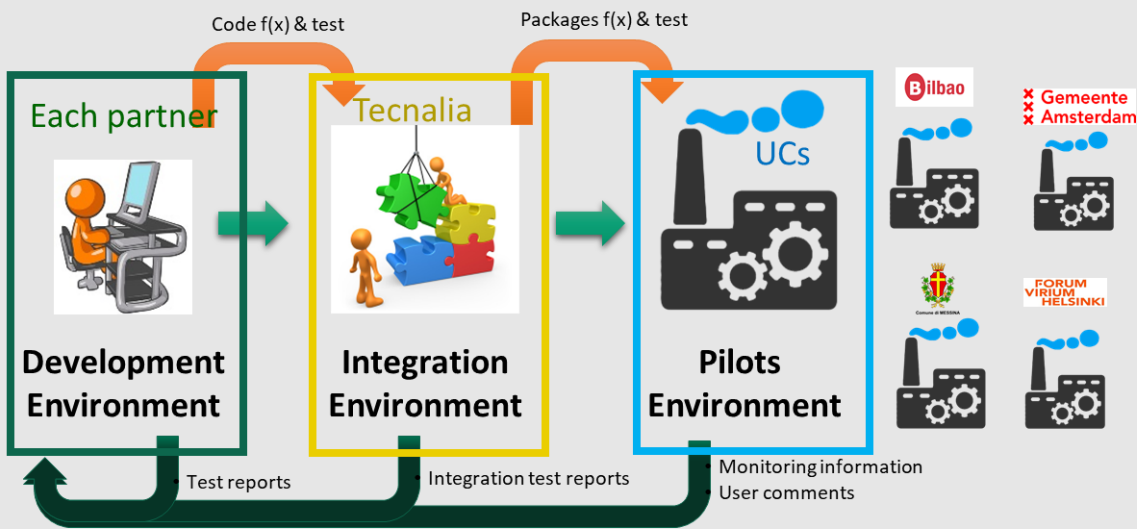


Figure 1. The three environments in URBANITE.

- The **Development environment** is implemented by each work team behind each component of the URBANITE solution. The technologies in the development environment largely depend on the technologies used to implement the component as well as the background of that team. Nevertheless, there are some common requirements at the development, the use of REST (Open API) in defining the interfaces of components.
- The **Integration environment** focuses on compiling the code and performing the unit test and integration test reports. This stage also includes the availability of a common storage mechanism for the binaries created, as well as the assets required to deploy the applications (e.g., configuration files, infrastructure-as-code files, deployment scripts).
- The **Pilots environments** are where the Use Case demos of URBANITE are deployed and implemented. These environments are foreseen to be located under the control of the final users, most probably into their premises.

The DevOps approach relies mainly in the Integration environment. The other two environments defined in the project (development and pilots) can be mentioned but are not the main focus of this document.

The DevOps approach in URBANITE is implemented using several interconnected tools: the version control tool: **GitLab**; the continuous integration tool: **GitLab CI/CD**; the containerization and deployment tool for easier portability and reconstruction of the solution: **Docker**; and the storage for binaries: **Artifactory**.

In the following sections of the chapter, the above-mentioned tools, and how they are organized and configured in URBANITE, is briefly described. This description will provide the reader with an overview of the framework, and will be followed by the description of more detailed procedures to use the framework in chapter *3-DevOps procedures.*

2.1 Gitlab

2.1.1 Software repository

The technical work packages of URBANITE will use **GitLab** [2] to manage source code and for revision control. GitLab is a Web-based Git repository hosting service. It offers all the distributed

revision control and source code management (SCM) functionality of Git [3] and adds additional proprietary features.

A source control tool helps store the code in different chains so one can see every change and collaborate more efficiently by sharing those changes. Rather than waiting on change approval boards before deploying to production, developers can improve code quality and throughput with peer reviews done via pull requests. Pull requests tell the team about changes some developer has pushed to a development branch in your repository. The team can then review the proposed changes and discuss modifications before integrating them into the main code line. Pull requests increase the quality of the software, which results in less bugs/incidents, and faster development.

The URBANITE GitLab[1] is offered by Tecnalia, hosting private and public repositories.

- The **private**[2] repositories are used to host the initial stages of the different components of the project until they are mature enough. The private repositories will also be used to store the pilot-oriented specific source code and any proprietary implementations of the partners that are not intended to be made public.
- Those components to be released under open source license will be deployed in the **pubic**[3] repositories, where they will be publicly available.

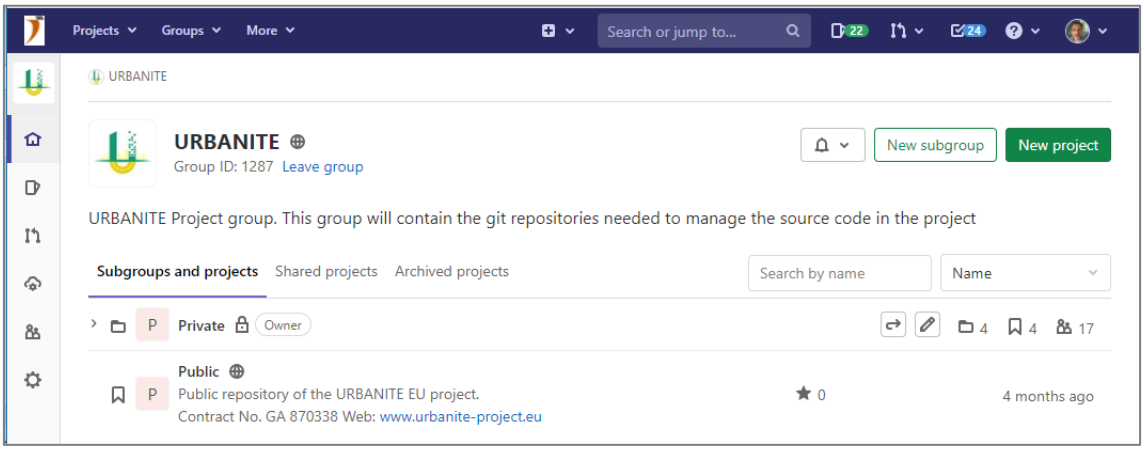


Figure 2. Private and public GitLab repositories of URBANITE.

[1] https://git.code.tecnalia.com/urbanite
[2] https://git.code.tecnalia.com/urbanite/private
[3] https://git.code.tecnalia.com/urbanite/public

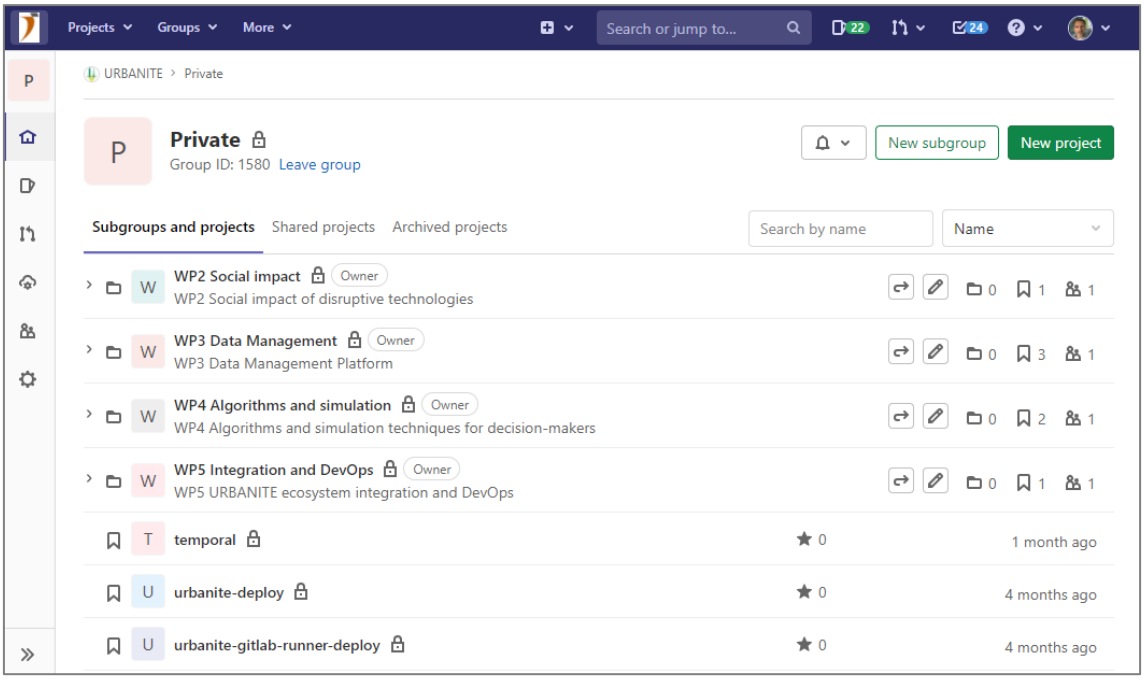


Figure 3. Private GitLab repository of URBANITE, organized by Work Packages.

2.1.2 Development tracking

GitLab issue tracker [4] is a tool for managing the development, for collaboratively developing ideas, solving problems and planning work. Issues are always associated with a specific project (repository), but if there are multiple projects in a group, one can also view all the reported problems collectively at the group level. The issue tracker comes integrated with GitLab, the source code management in use in the project

Issues will be used to track the status of feature proposals, bug reports, new code implementations. Each issue in URBANITE includes the following attributes:

- **Content**: *Title, Description*
- **People**: *Author, Assignee*
- **State**: *open/closed*
- **Planning** and **tracking**: *Milestone, Due date, Weight, Labels*

Labels are part of issue boards and permits categorize issues using colours and descriptive titles, and filter, manage and search the issues. In URBANITE the labels are going to be used to categorize issues according to components and Use Cases. At the time of writing, these are the labels foreseen:

- **Component labels**: *Traffic simulation, Policy simulation & Validation, Recommendation engine, Advanced visualization, Prediction, Correlation discovery, Data clustering, Data projection, Self-organizing map, Controller, Data storage & retrieval, Data fusion/aggregation, Data transformation, Data curation, Data preparation, Data harvesting, Data anonymization*
- **Use Cases labels**: *Helsinki, Bilbao, Messina, Amsterdam*
- **Type labels:** *bug, suggestion, enhancement*

Milestones will be defined as initially planned in the URBANITE Proposal DoW [5], that is, at months M15, M27 and M33.

2.2 GiLab CI/CD

Continuous integration is the practice of making frequent commits to a common source code repository. It's continuously integrating code changes into the existing code base so that any conflicts between different developer's code changes are quickly identified and relatively easy to remediate. If, in addition, the build and test processes are automated, this notably increases deployment efficiency.

In URBANITE, **GitLab CI/CD** [6] is used as the continuous integration and deployment tool. GitLab CI/CD is naturally integrated in GitLab. The setup of GitLab CI/CD for projects hosted on GitLab is easy since it uses the GitLab API for setting up hooks. GitLab CI/CD is a visual management tool, so it can be used as an interactive and operational dashboard for release management.

The automation scripts needed to run the integration and deployment tasks are maintained in Git, the source management tool, so that the integration tasks are included in the configuration management. The elements that configure the URBANITE continuous integration and deployment are listed below.

2.2.1 Branches

In URBANITE, the CI/CD is implemented in a (private) particular repository for integration: *URBANITE-deploy*. Two branches are defined in URBANITE-deploy: *develop* & *master*.

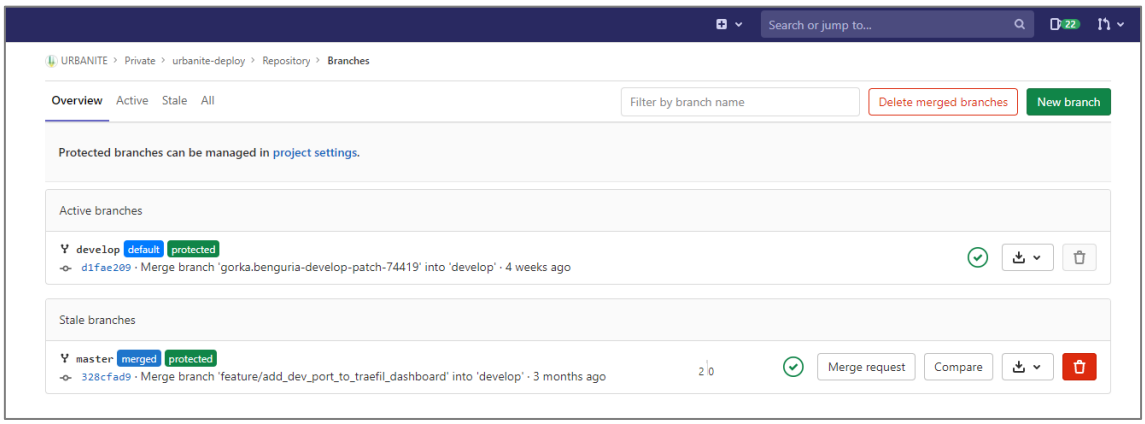


Figure 4. The two branches in the integration repository

Developers are allowed to commit and merge their code in develop branch (default). In the master branch, however, only maintainers are allowed to merge. The master branch is destined to contain the tested, verified releases.

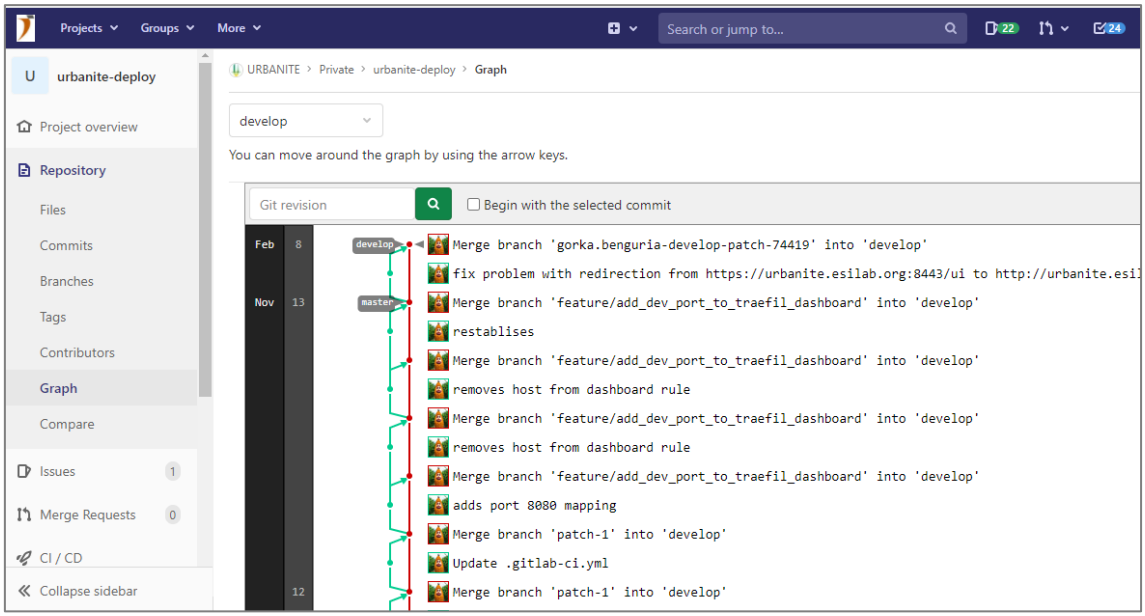
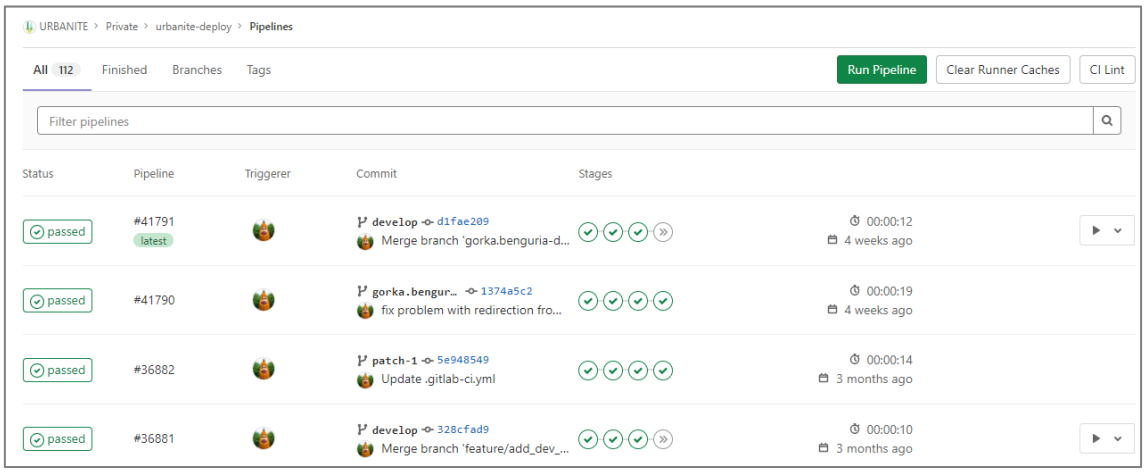


Figure 5. Graph view of the branches in the integration repository

2.2.2 Pipeline

Pipelines[4] are the top-level component of continuous integration, delivery, and deployment. Pipelines comprise Jobs, which define *what* to do (e.g., compile or test), and Stages, which define *when* to run the jobs (e.g., stages that run tests after stages that compile the code). If any job in a stage fails, the next stage is not (usually) executed and the pipeline ends early. If all jobs in a stage succeed, the pipeline moves on to the next stage.

In URBANITE, the pipeline is composed by 5 stages: *build, deploy, tests, stop* and *debug*. The pipeline, as is standard in the tool, is defined in the *.gitlab-ci.yml*[5] file. The pipeline is launched when a merge is accepted and can be controlled and manipulated through a graphical interface.



[4] https://docs.gitlab.com/ee/ci/pipelines/

[5] In GitLab CI/CD, a gitlab-ci.yml file, in the root of the repository, contains the CI/CD configuration.

Figure 6. Some pipelines in the integration repository

The pipeline tasks are based in docker-compose, and are defined and distributed in different yaml files, which are called from the gitlab-ci.yml attending the branch in use and the task to be performed.

2.2.3 Runners

A runner is a process that picks up and executes jobs of the pipeline. Multiple jobs in the same stage can be executed in parallel, provided we have concurrent runners.

In URBANITE, a runner has been defined by now. It runs in one of the machines of the integration environment, *URBANITE.esilab.org*, and runs the GitLab CI/CD jobs defined.

2.3 Docker

Infrastructure as Code (IaC) is a form of configuration management that codifies an organization's infrastructure resources into text files. These infrastructure files are then committed to a version control system like GitLab.

In URBANITE, **Docker** is the IaC technology chosen. Docker allows provisioning to be more consistent and reproducible. Containers allow, using the code, the explicit provision of the configuration of the containers, that can be applied and reapplied many times, to put a server into a known baseline: operating system, packages to include, content, configuration, etc. Docker allows containers creation, instantiation, stopping and deletion, logs communication and persistency definition.

Using Docker technology implies that each “dockerizable” component in URBANITE must have an associated Dockerfile in its repository. A Dockerfile is a text document that contains all the commands to assemble an image automatically. A very simple Dockerfile looks like this:

```
FROM httpd:2.4
RUN mkdir /usr/local/apache2/htdocs/ui
COPY ./public-html/ /usr/local/apache2/htdocs/ui
```

2.3.1 Portainer

Portainer is an open source tool for managing container-based software applications. It can be used to set up and manage environments, deploy applications, monitor app performance and triage problems.

In URBANITE, a Portainer instance has been deployed to make simpler for developers and integrators deploying apps and troubleshooting problems.

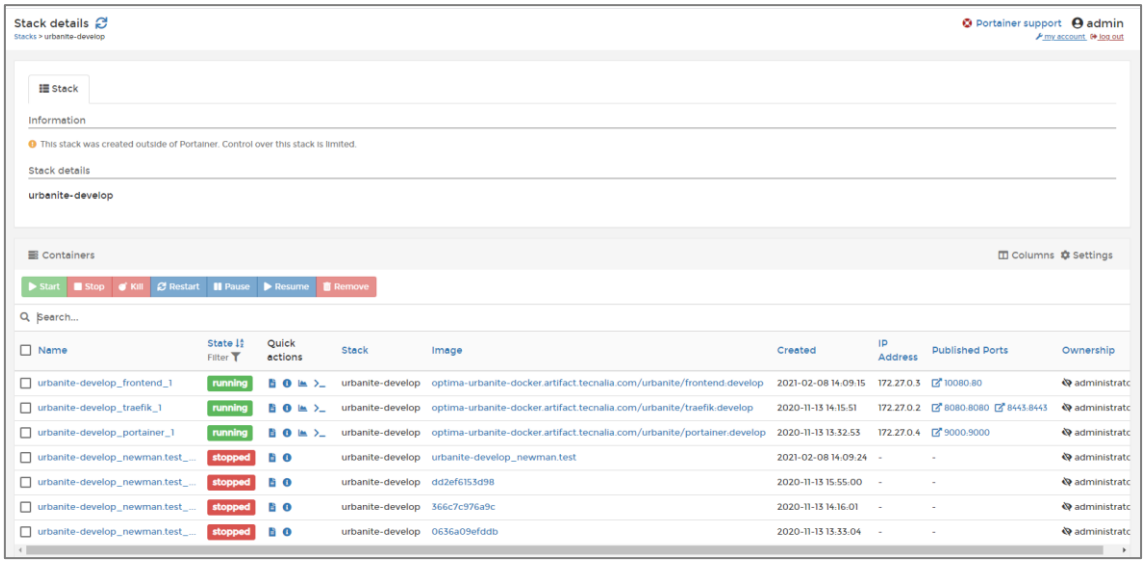


Figure 7. Portainer: view of the URBANITE-develop stack of containers

2.3.2 Artifactory

Some container technologies support the container's registry usage, where the developed containers can be uploaded so that other team members can download, use, and test them with a small set of instructions.

A binary repository manager is a dedicated server application designed to manage binary components needed for the applications that we build. Using a repository manager is one of the best practices for using any build tools.

In URBANITE, we are using **Artifactory** as a repository manager. We offer a registry at the level of the project, where all binary artefacts (docker images) can be stored and downloaded. This provides efficiency, reliability, consistency and facilitates automation using its REST API.

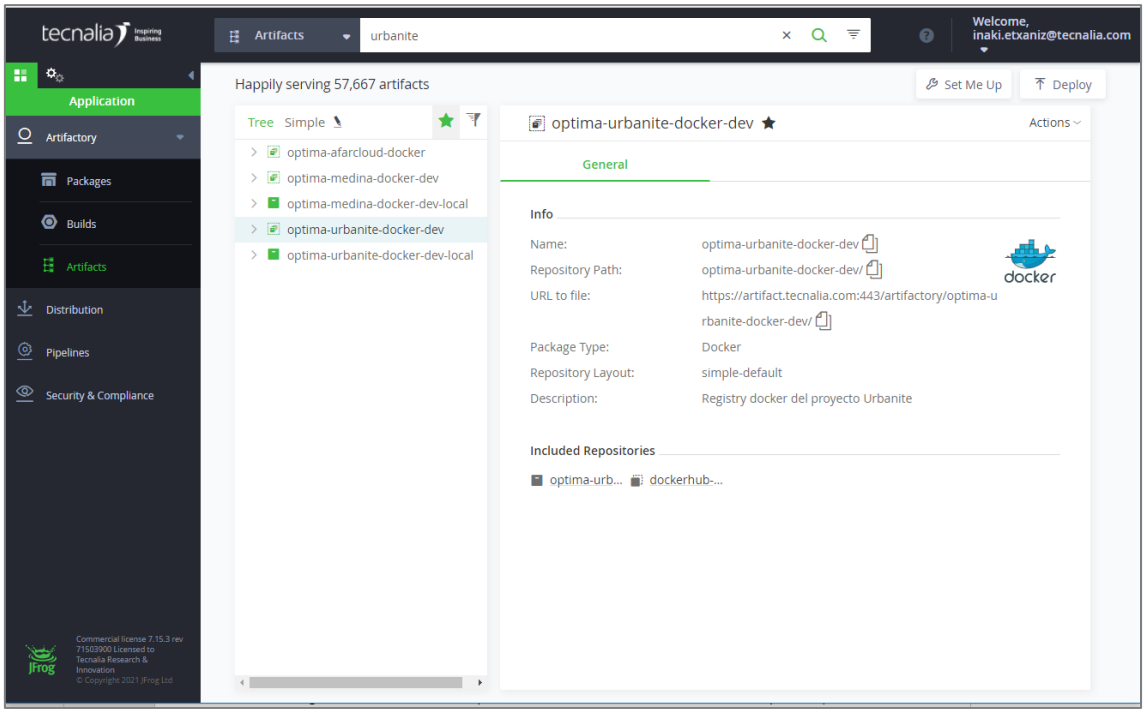


Figure 8. Artifactory for the URBANITE project.

3 DevOps procedures

In this section, we describe the different procedures that are associated with the DevOps infrastructure established in URBANITE. These procedures aim to support the integration of the components coming from the different development teams that contribute to the URBANITE platform and serve to several roles in the project:

- **Development teams**: that implement components and want to test them within the URBANITE platform.
- **Integrators**: in charge of putting all the pieces together
- **DevOps infrastructure maintainer**: in charge of monitoring the health and perform maintenance activities.
- **Release responsible**: in charge of delivering and updating the platform for the project.
- **Pilot Responsible**: in charge of managing, planning, scheduling, and controlling software delivery towards the project pilots.

One person can, and probably will perform more than one of these generic roles in the project. Nevertheless, we have preferred to separate them for the sake of clarity.

3.1 Operation procedures

This section describes the procedures for the development teams, the release responsible and the pilot responsible. A separate section will be provided for each of the roles.

3.1.1 Development team

The development team oversees coding the different parts of the application and configuring their surrounding components and services (such as databases, big data infrastructures, message queues, etc.) when necessary.

In this section, we include procedures related with the testing of changes in already integrated components that do not require to change the integration setup. Changes in the integration setup -such as adding components or modifying the way in which they are integrated- require the supervision and support from the Integrator and are described latter on.

3.1.1.1 Upgrading a component in integration environment

The objective of this procedure is to allow the development teams to upgrade their assets autonomously in the integration environment. Using this procedure, they create a new branch in the deployment repository with the last changes. That branch is tested against the automated tests, and if it succeeds, they will be allowed to merge them into the "develop" branch. As soon as the code is merged, the development environment will be updated with the last changes, and everyone in the project will have access to them for evaluation and interactive testing purposes.

REQUIRED:

Internet connection, Git client.

PROCEDURE:

[NOTE1: The name of the component is indicated through the text as [component_name].]

[NOTE2: in case that the component or any of its associated files requires changes in their start-up configuration, communicate with the *Integrator* to execute the Maintenance procedure "Component integration/update".]

Make sure that the desired changes to the component are committed and pushed into the component repository. Also, be sure that the development tests have been performed, if any.

Take note of the commit id to be integrated with the rest of the parts of the URBANITE platform, for example, 3a661d46.

Clone locally the URBANITE-deploy repo https://git.code.tecnalia.com/URBANITE/private/URBANITE-deploy. In case it has been cloned before, just pull the latest changes from the remote repo. Be sure to be in the "develop" branch

```
git clone https://git.code.tecnalia.com/URBANITE/private/URBANITE-
deploy
cd URBANITE-deploy
git checkout develop
git pull --recurse-submodules
git submodule update --init --recursive
```

Check that we are in the latest version, issuing a git status with a result showing no changes. In case we see any undesired change, we need to reset it or take it into account for the upcoming commit.

```
git status
```

Create a new branch for the changes we are going to introduce. The recommended branch name is "update" plus an identifier (e.g., *update/ui_001*). This will group all the updates in the GitLab user interface. Then go to the folder in which the component is linked (e.g. */[component_name]*) and checkout the desired commit.

```
git checkout -b update/[component_name]_001
cd git/[component_name]
git pull
git checkout 3a661d46
```

Come back to the URBANITE-deploy repo root folder and add and commit the component (it is always recommended to verify the changes before the commit). If everything is ok, push the changes to the remote repo.

```
cd ..
cd ..
git add git/[component_name]
git status
git commit -m "updates ui"
git push –set-upstream origin update/ui_001
```

As soon as you push the new branch, a pipeline will be fired in the integration environment. This pipeline builds, runs, tests and destroys the entire URBANITE platform to make sure that the components can be built and comply with the automated tests. You can check the progress and the result of the pipeline at https://git.code.tecnalia.com/URBANITE/private/URBANITE-deploy/-/pipelines.

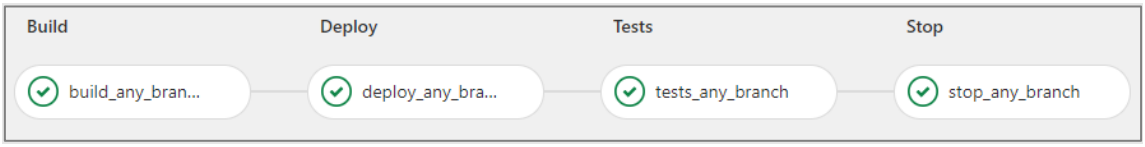


Figure 9. Stages and jobs in the pipeline of any branch.

The push returns a message with a URL to create a request to merge the new branch in the *develop* branch. For this, start a browser and paste that URL. Alternatively, in Windows command line interface (*cmd*), you can issue the following command:

```
start chrome [URL]
```

You will be allowed to merge if the pipeline was correct. Check the messages and, if all is OK, merge the new branch into *develop*. That will start a new pipeline in the integration environment.

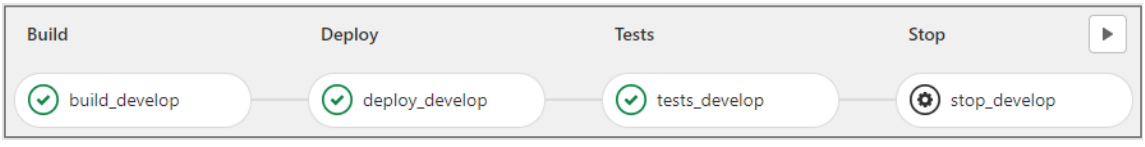


Figure 10. Stages in the pipeline in the "develop" branch.

This pipeline builds, runs and tests the entire URBANITE platform but, in contrast to the previous pipeline, it leaves the platform alive, allowing you to perform additional manual tests. The environment can be accessed through the GitLab GUI, in Operations > Environments.

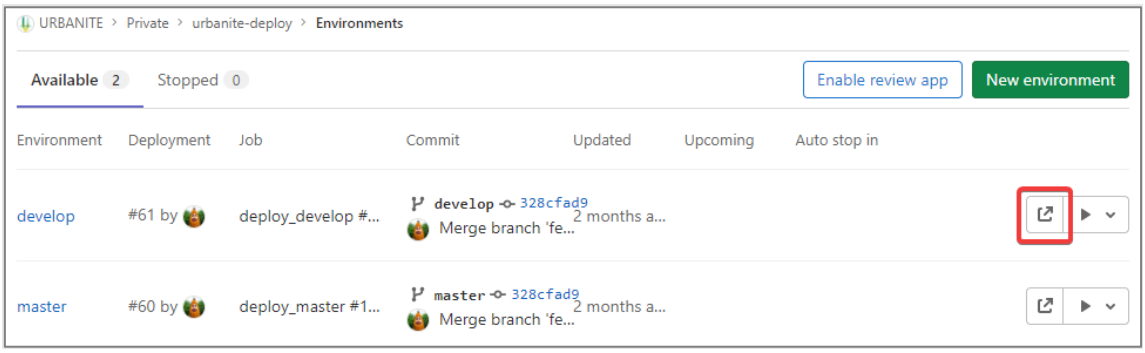


Figure 11. The "develop" and "master" environments.

To finish, you should inform those affected by the change, either by sending an email or by other means.

EXPECTED OUTCOME:

An updated integration environment (in the case of the example, accessible at the following address: https://URBANITE.esilab.org:8443/ui/).

3.1.1.2 Debugging a component deployment

The objective of this procedure is to support the development team to debug and fix issues during the release of their work into the integration environment (for example, if during the previous procedure -"Upgrading a component in integration environment"- the pipeline fails at some point) or even once the containers have been successfully deployed (for example, when some issue is found during the interactive testing).

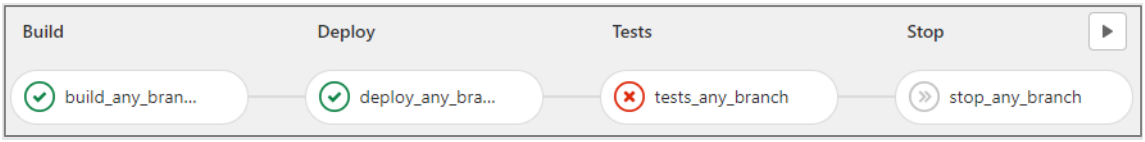


Figure 12. Failure -in "Test" stage- in the pipeline.

REQUIRED:

Internet connection, Portainer.

PROCEDURE:

There are different mechanisms to support the development team in the debugging of issues in the assets running in the integration environment: continuous integration logs, container logs, container status, container command line.

Continuous integration logs are those gathered from the GitLab runners while executing the continuous integration steps. They are accessible in URBANITE deploy project at GitLab, more specifically in the pipelines section (CI/CD > Pipelines). They are relevant when the pipeline fails.

When this occurs, we usually receive an email from the GitLab platform warning us about the problem.

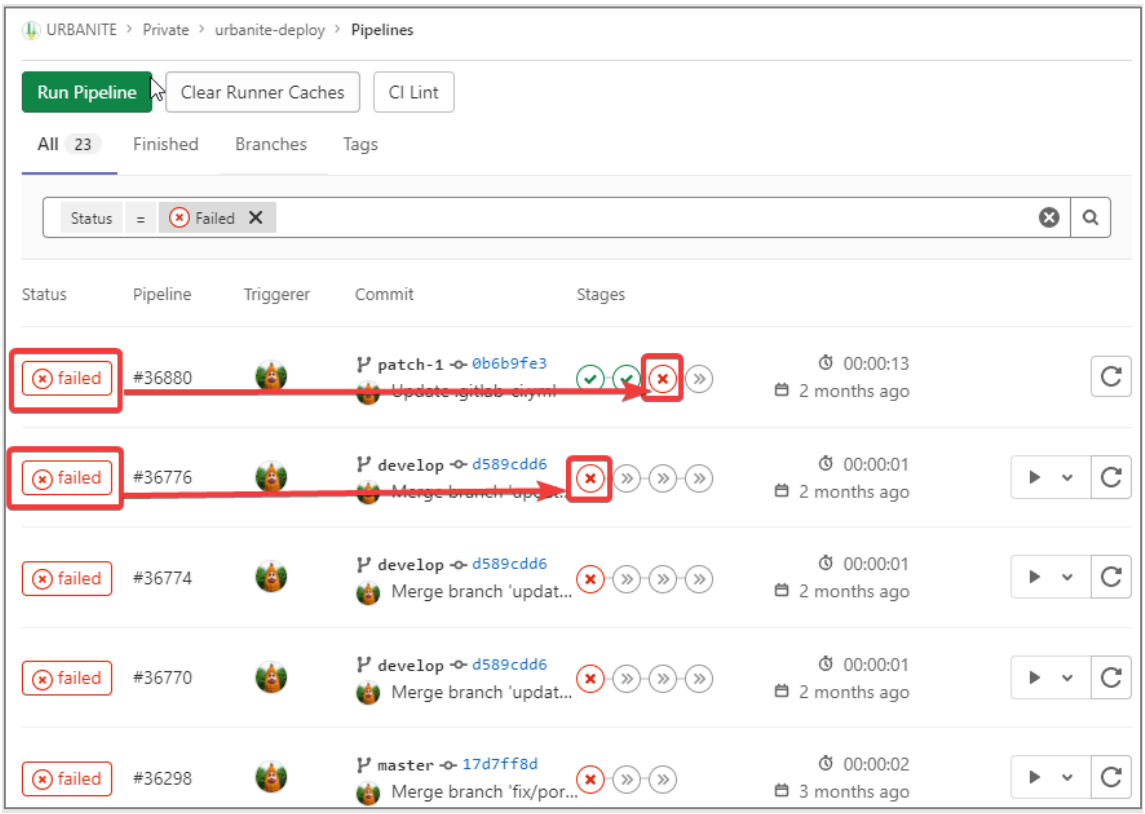


Figure 13. Pipeline page.

On the pipelines page we can see those that have failed. In failed ones, we can check the detail or the pipeline by clicking at the pipeline id.

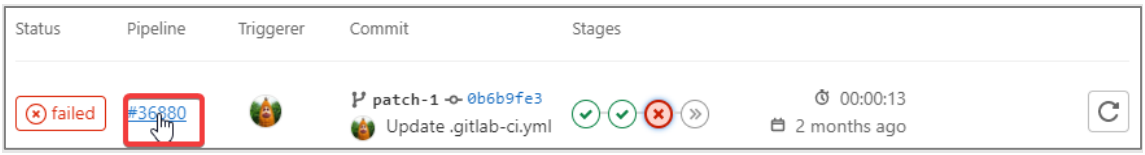


Figure 14. Pipeline ID to access the details.

That will bring us to the detail of the pipeline stages execution.

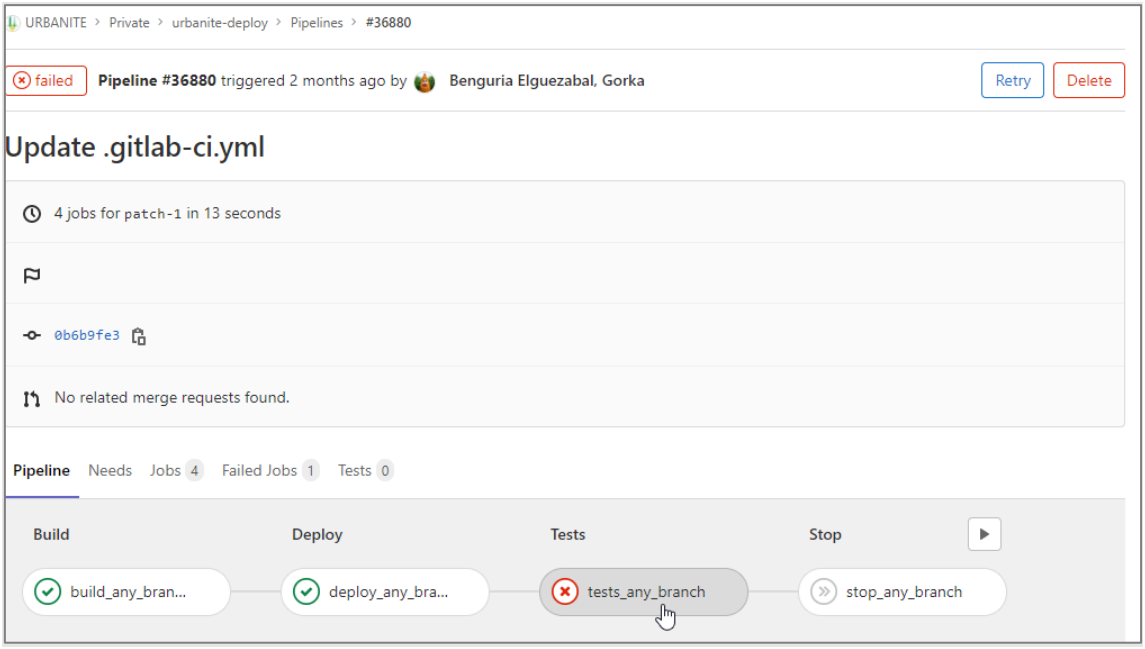


Figure 15. Details page of a pipeline.

You can click on each of the stages to access its details. But beware that job logs are erased periodically (every two months).

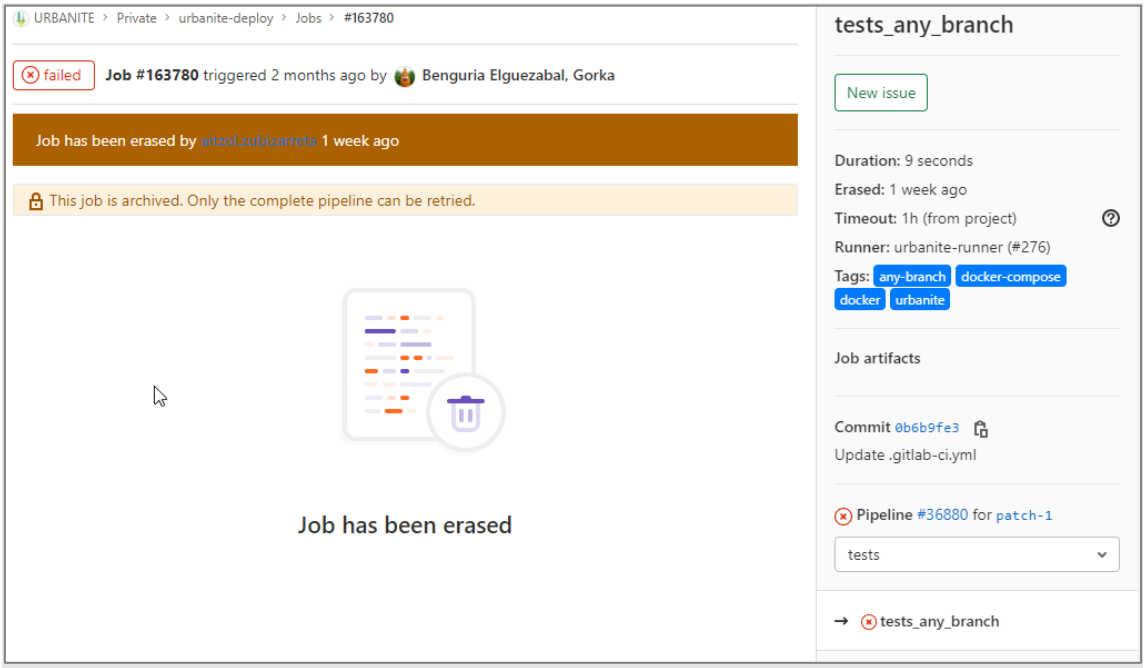


Figure 16. Details of a job that has been erased.

A stage with content would look like the following one.

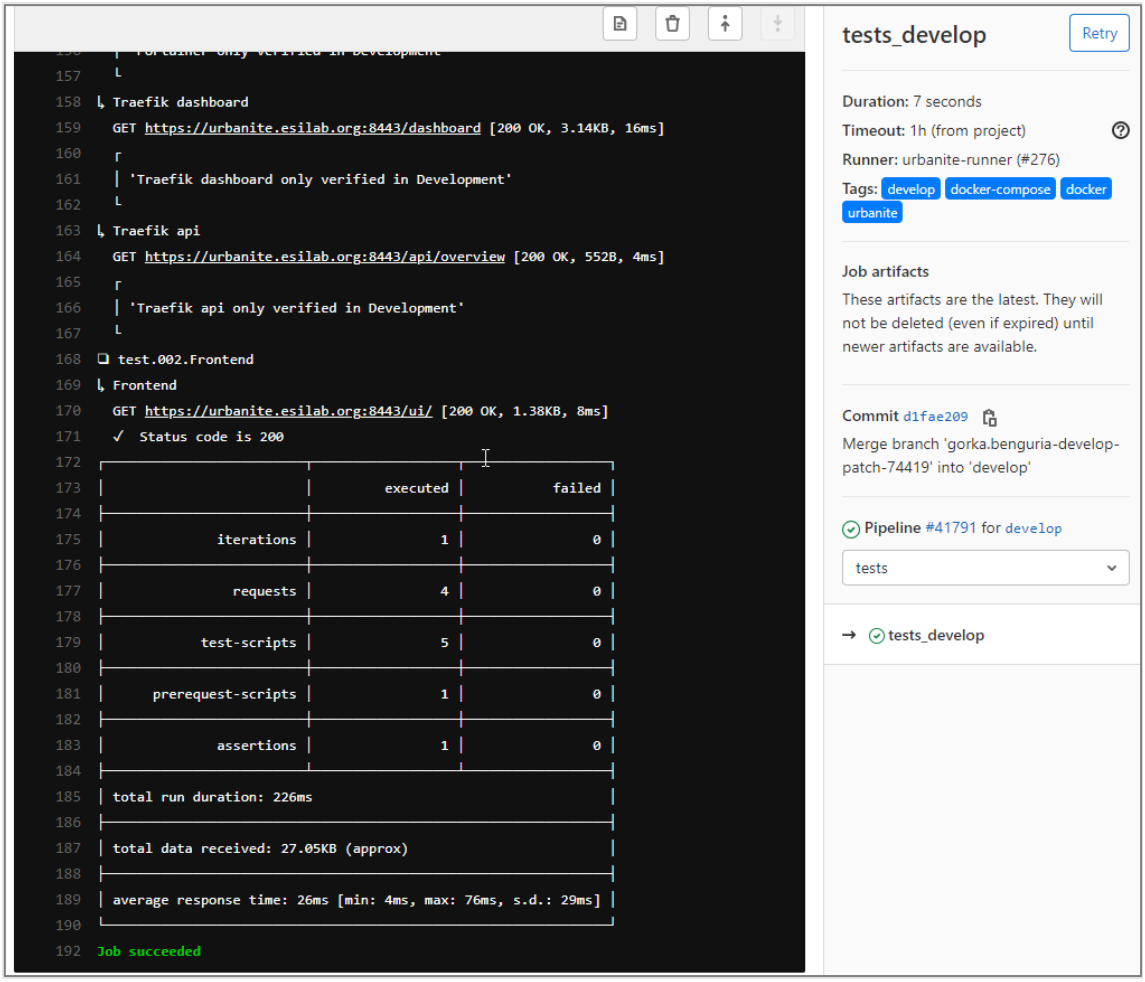


Figure 17. Details of a job.

Another mechanism is the **container status checking**. To check the status of each container in the different container stacks in the integration environment, a Portainer server is provided. The URL of URBANITE Portainer is: [https://URBANITE.esilab.org:8443/Portainer/.](https://URBANITE.esilab.org:8443/Portainer/) It is also possible to filter by stack if we are interested in doing so:

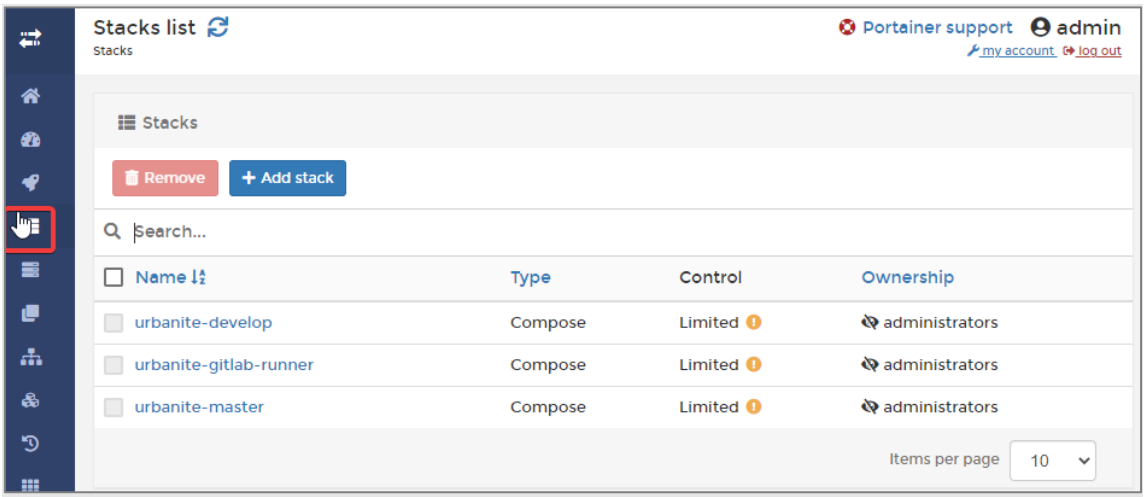


Figure 18. Portainer, stacks list.

The naming approach followed with containers indicates (i) the stack they belong to and (ii) the name of the container. In this case, there are three stacks:

- **URBANITE-develop:** stack that runs the latest commit at the *develop* branch.
- **URBANITE-master:** stack that runs the latest commit at the *master* branch.
- **URBANITE-any-branch:** stack that runs *the rest of branches* as soon as they are pushed.

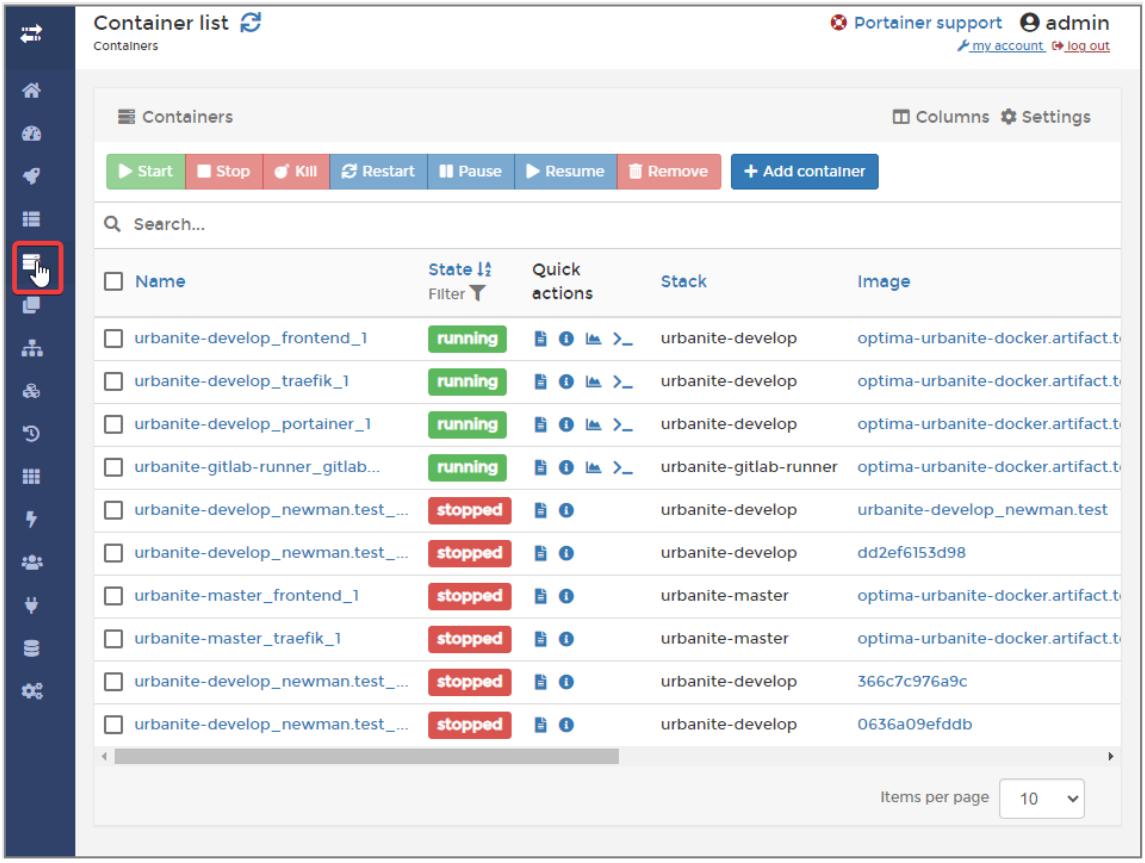


Figure 19. Portainer, container list.

We can then access the details of each container, clicking its name.

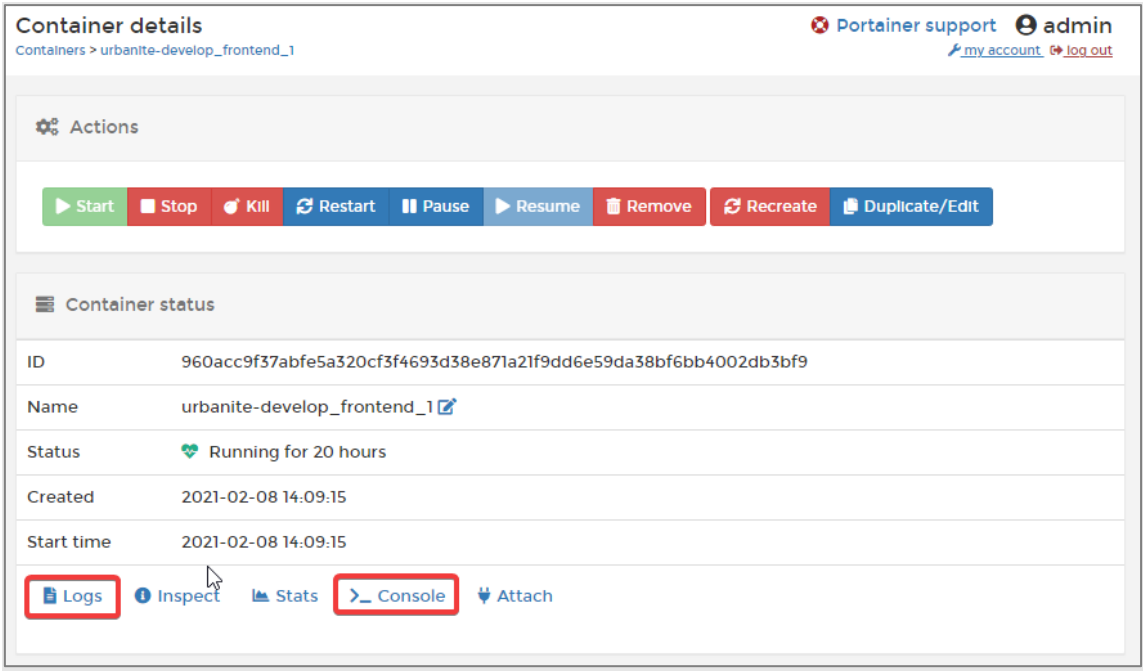


Figure 20. Portainer, container details.

In the container details, we can see information about the variables, the labels, the networks or the volumes. Besides, we have access to the **container logs** and console, which is useful in the odd case that a deployment error takes place during the testing phase of the deployment (container logs show the same output as issuing the "`docker logs -f`" command at the command line).

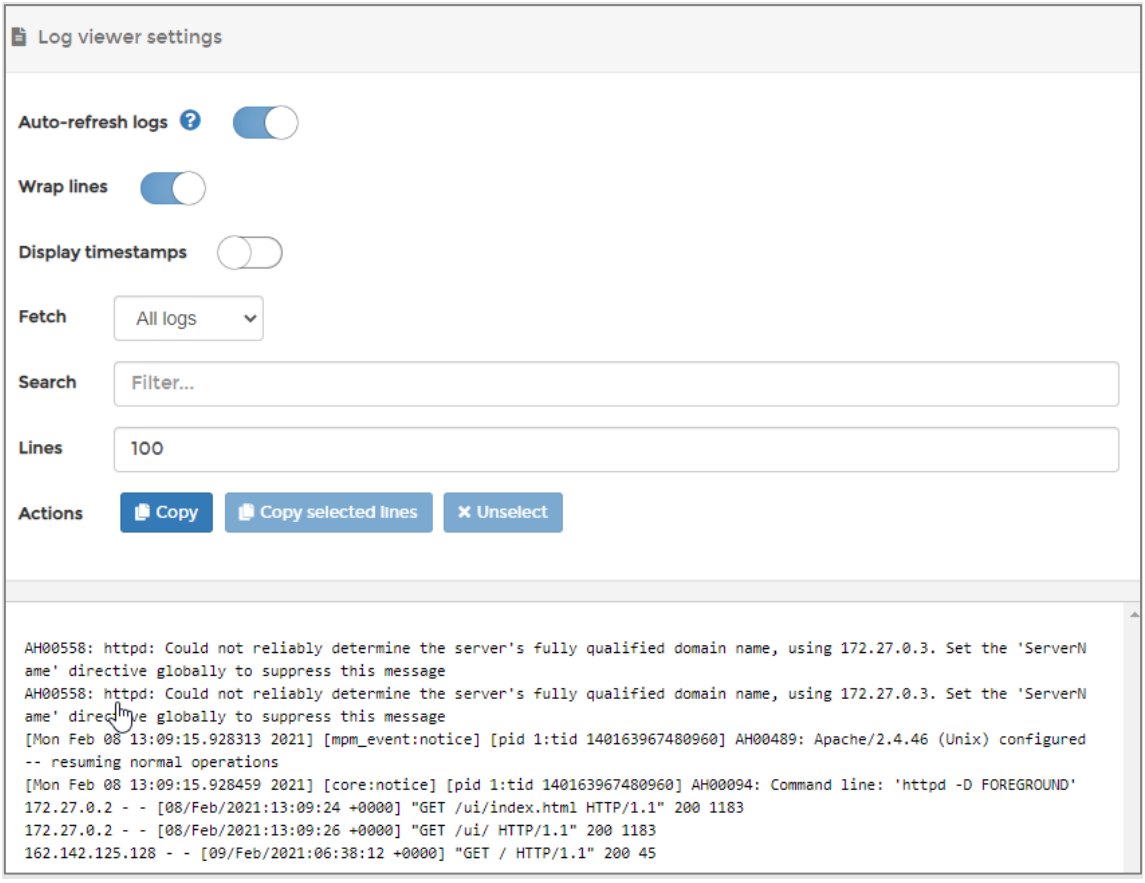


Figure 21. Portainer, container logs.

Container console gives us access to the terminal at the container. As in the command line, you must specify how to access the shell (as it varies depending on the container base: *sh, bash, ash*…) and you can also specify the user (usually *root*).

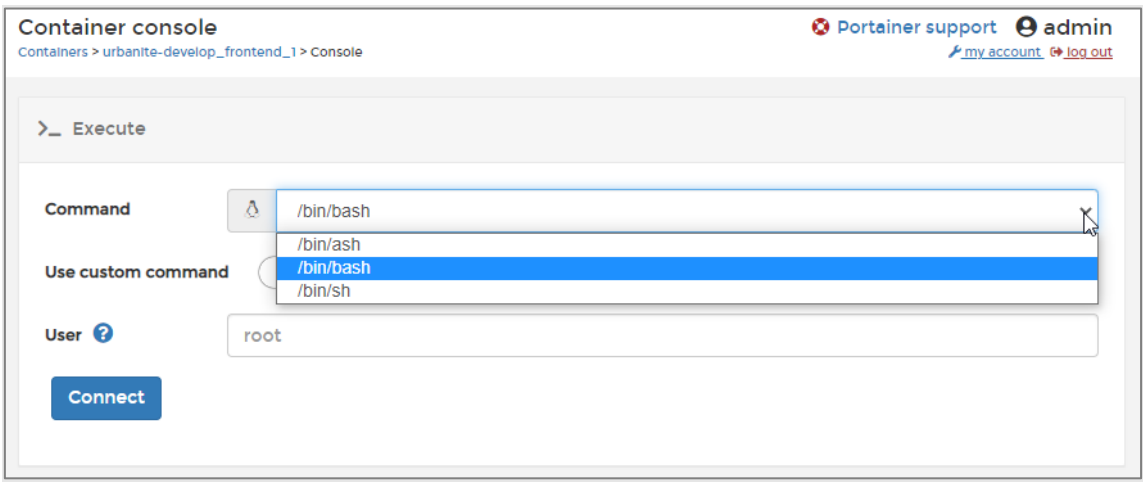


Figure 22. Portainer, accessing container console.

Once we choose the right command, we get access to the container console:

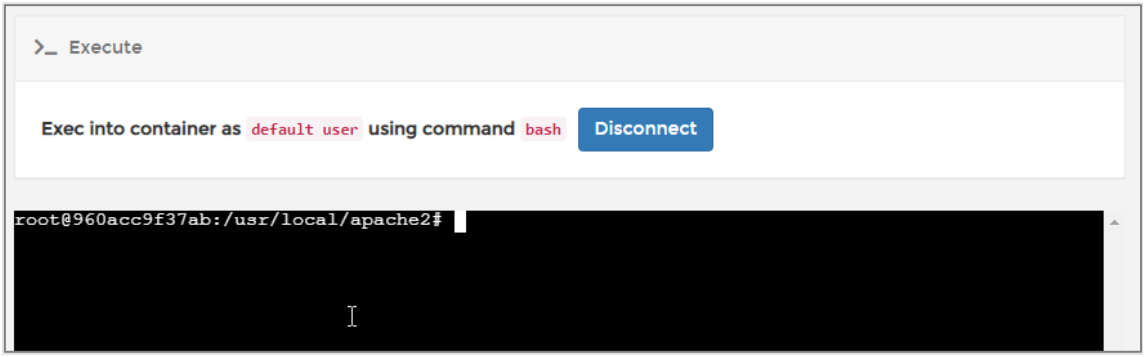


Figure 23. Portainer, container console.

Within the console, we can issue most of the command that we can use in a regular shell. We can install tools, create, edit, read and delete files, etc. It is useful for several purposes like checking connectivity, checking stored data, checking those logs that are not thrown to console but stored in files, or checking configuration files.

Take into account that there are some commands, such as *reboot* or *device* related commands, that are limited by the nature of the docker container technology. For example, in order to reboot a container, the alternative will be to "Restart" the container from the Portainer console.

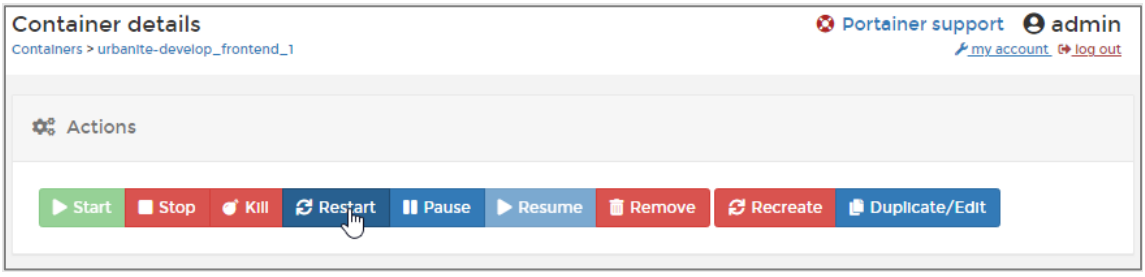


Figure 24. Portainer, actions on containers.

3.1.1.3 Creating an issue

The objective of this procedure is to provide feedback to other development team about some issue at development or integration.

REQUIRED:

Internet connection.

PROCEDURE:

Go to the URBANITE GitLab group https://git.code.tecnalia.com/groups/URBANITE/-/issues, and open an issue. We create an issue associated with the project (repository) associated. The assignment can be adjusted afterwards; therefore, in case it is not clear, you can assign the issue to *URBANITE-deploy*.

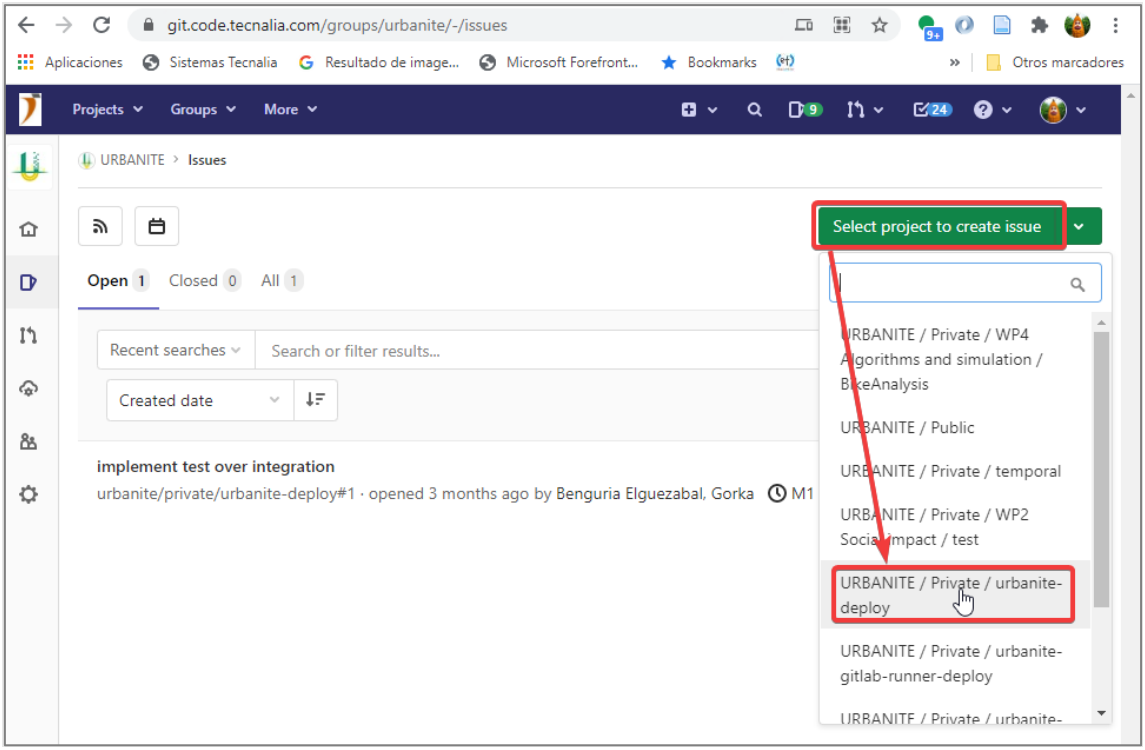


Figure 25. Portainer, selecting project to create an issue.

When we choose the target project the creation button changes to reflect that selection.

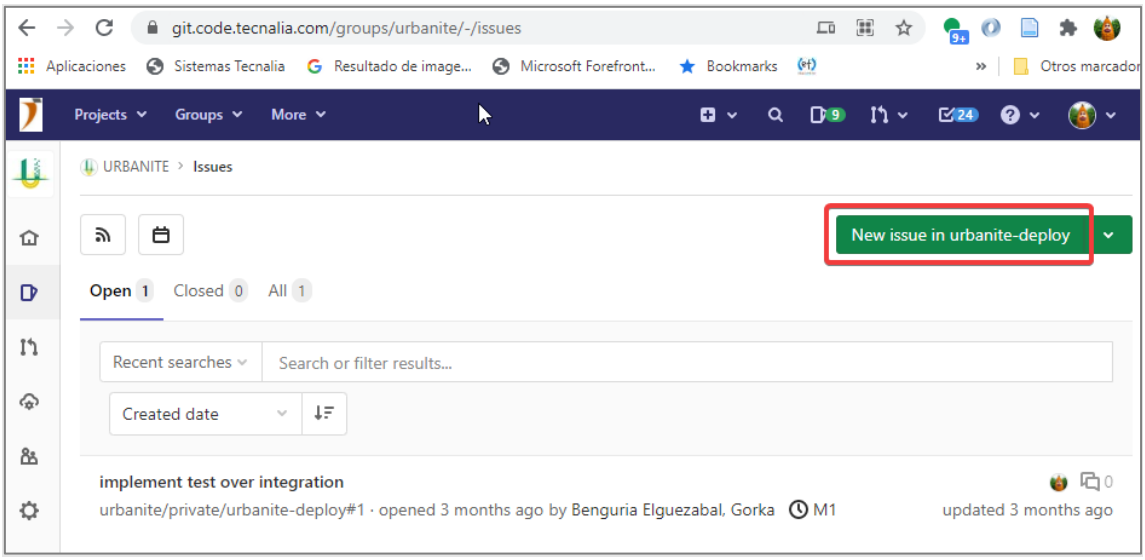


Figure 26. Portainer, creating an issue.

Once created, the issue submit dialog will open. We can fill the relevant fields and label it as “new” so that we can identify it during the planning meetings to discuss, complete, prioritize and schedule appropriately.

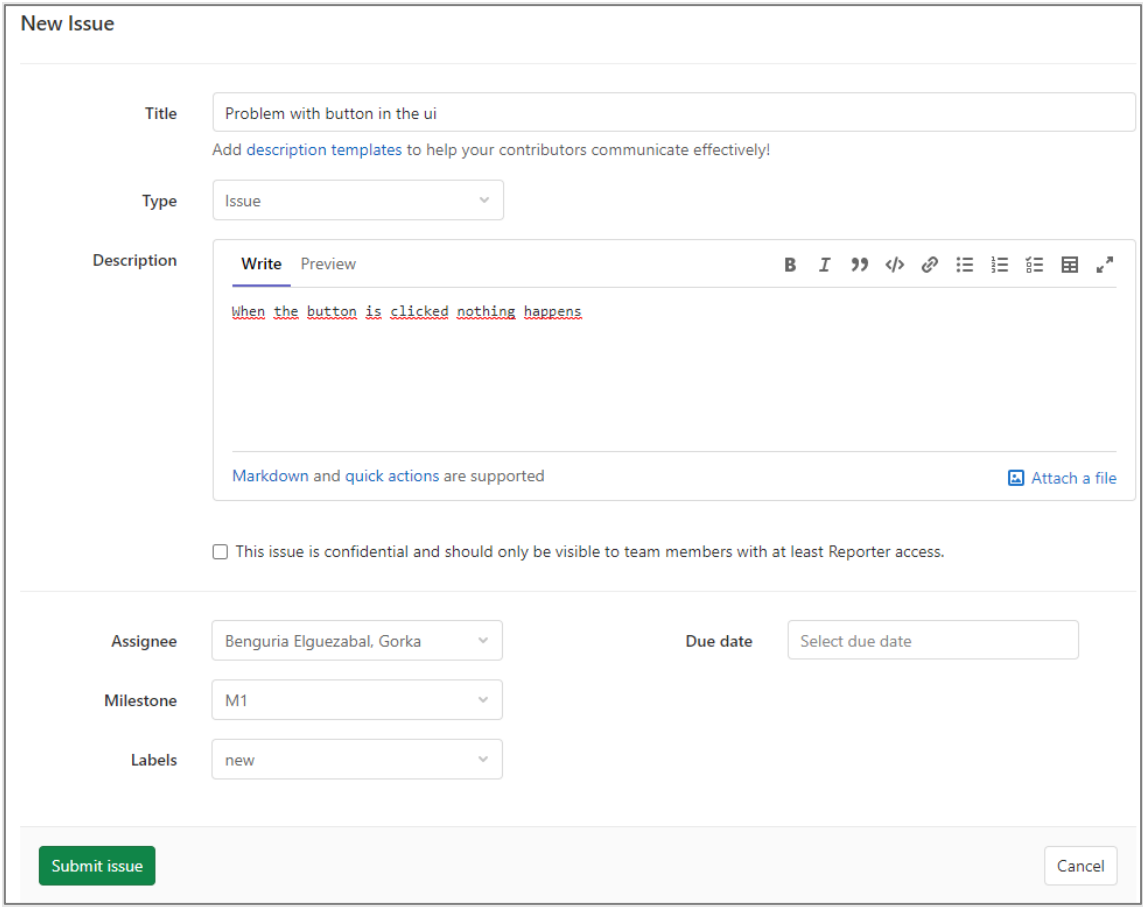


Figure 27. Portainer, new issue details.

EXPECTED OUTCOME:

A new issue in the GitLab URBANITE group to be discussed during the planification meetings.

3.1.2 Release responsible

The release responsible oversees managing, planning, scheduling, and controlling software delivery. The releases will be focused mainly on the project pilots, but there can be other releases related with project milestones, meetings, reviews, etc.

[NOTE: Some procedures performed by the Release responsible in the master environment - e.g., debugging a component, creating an issue- are so equivalent to those followed by developers in the development environment that we decided not to repeat them here.]

3.1.2.1 Promoting a component to Master

The objective of this procedure is to promote the "develop" environment to "master". The master branch holds snapshots of the develop branch. These snapshots can be created in response to different needs: an internal review, a meeting, a milestone, a release, a fix, etc.

REQUIRED:

Internet connection, Git client.

PROCEDURE:

It is assumed that the quality of the component to be promoted has been validated with the required tests that can involve using additional endpoints like REST APIs, message queues, databases, etc.

First of all, communicate to the project team that a component release is going to be fixed. This implies that, during the process, merge requests to the *develop* branch may be constrained. For this, a repo maintainer should temporarily restrict merging, modifying the branch configuration (https://git.code.tecnalia.com/URBANITE/private/URBANITE-deploy/-/settings/repository).

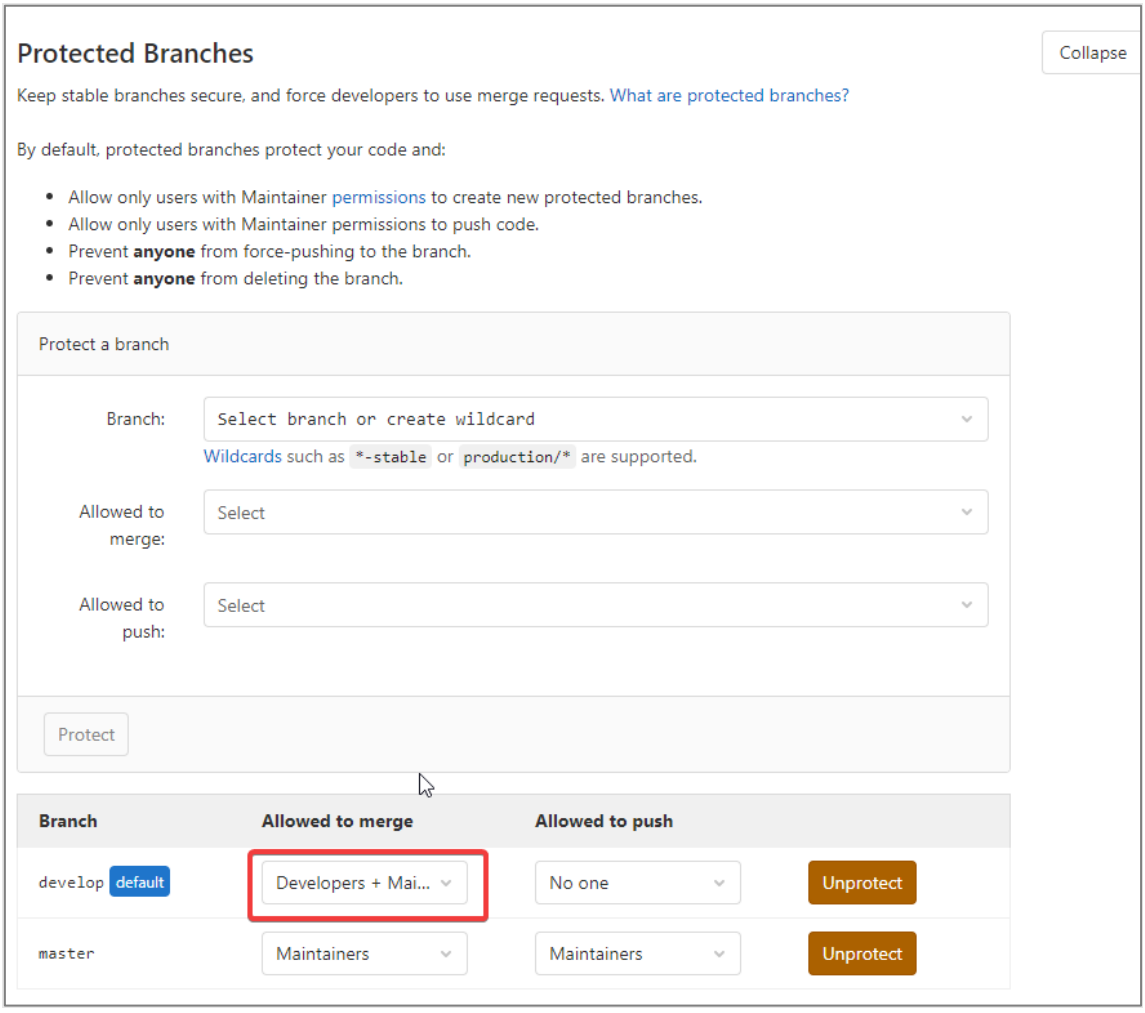


Figure 28. GitLab, branch configuration.

A merge request has to be issued from the *develop* branch. The merge request will clone the develop branch status in the master branch. To do so first, we update the develop branch and all their submodules.

```
cd URBANITE-deploy
git checkout develop
git pull
git submodule update
```

Once updated the *develop* branch, we switch to the *master* branch and reset it to the status of the *develop* branch.

```
git checkout master
git reset --hard develop
```

Make sure that *master* is equal to the current *develop* branch, using the command `diff`.

```
git diff develop
```

This should give no differences. Finally, upload the changes back to the remote repository.

```
git push
```

This will start the master pipeline in GitLab.

Figure 29. GitLab, "master" pipeline.

This pipeline is in charge of updating the master environment that, once finished, will be accessible in our example at https://URBANITE.esilab.org/ui/.

EXPECTED OUTCOME:

A new version of the master environment.

3.1.2.2 Release packaging for sharing

The objective of this procedure is to take a given master commit and generate a release for sharing with other parties, mainly pilots.

[NOTE: this procedure is subject to be improved by extending the continuous integration in the future.]

REQUIRED:

Internet connection, SSH access to the server.

PROCEDURE:

This procedure can be done from the server using SSH or using Portainer. We will describe first the procedure with the SSH and later the same process using Portainer.

With **SSH**, access to the server. Once inside, log into the repository (in our case an Artifactory repository).

```
docker login optima-URBANITE-docker-dev.artifact.tecnalia.com
```

The next step is to identify the images. To do so, we need to get the latest images

```
docker images | master
```

We retag those images with a release version (e.g., v01)

```
docker tag optima-URBANITE-
docker.artifact.tecnalia.com/URBANITE/frontend:master optima-URBANITE-
docker.artifact.tecnalia.com/URBANITE/frontend:v01
```

and push them to the registry

```
docker push optima-URBANITE-
docker.artifact.tecnalia.com/URBANITE/frontend:v01
```

With **Portainer**, we first need to make sure that we have registered the Artifactory registry in Portainer (check at https://URBANITE.esilab.org:8443/Portainer/#/registries).

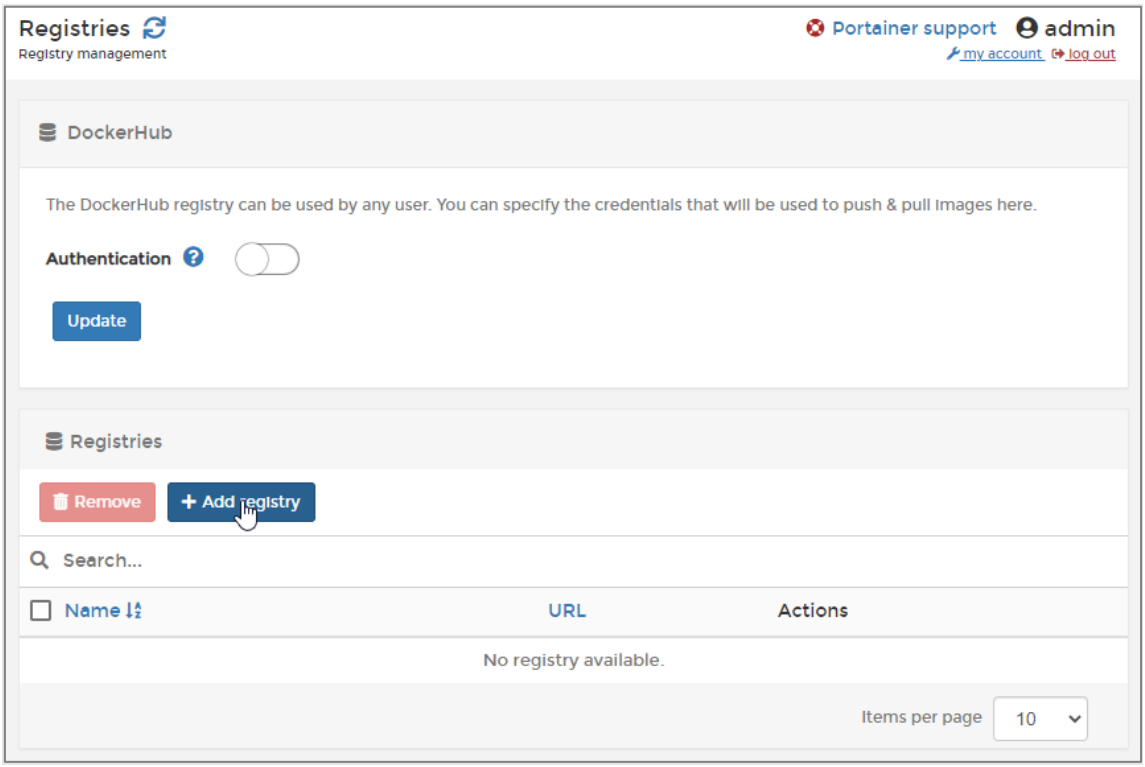


Figure 30. Portainer, adding a register.

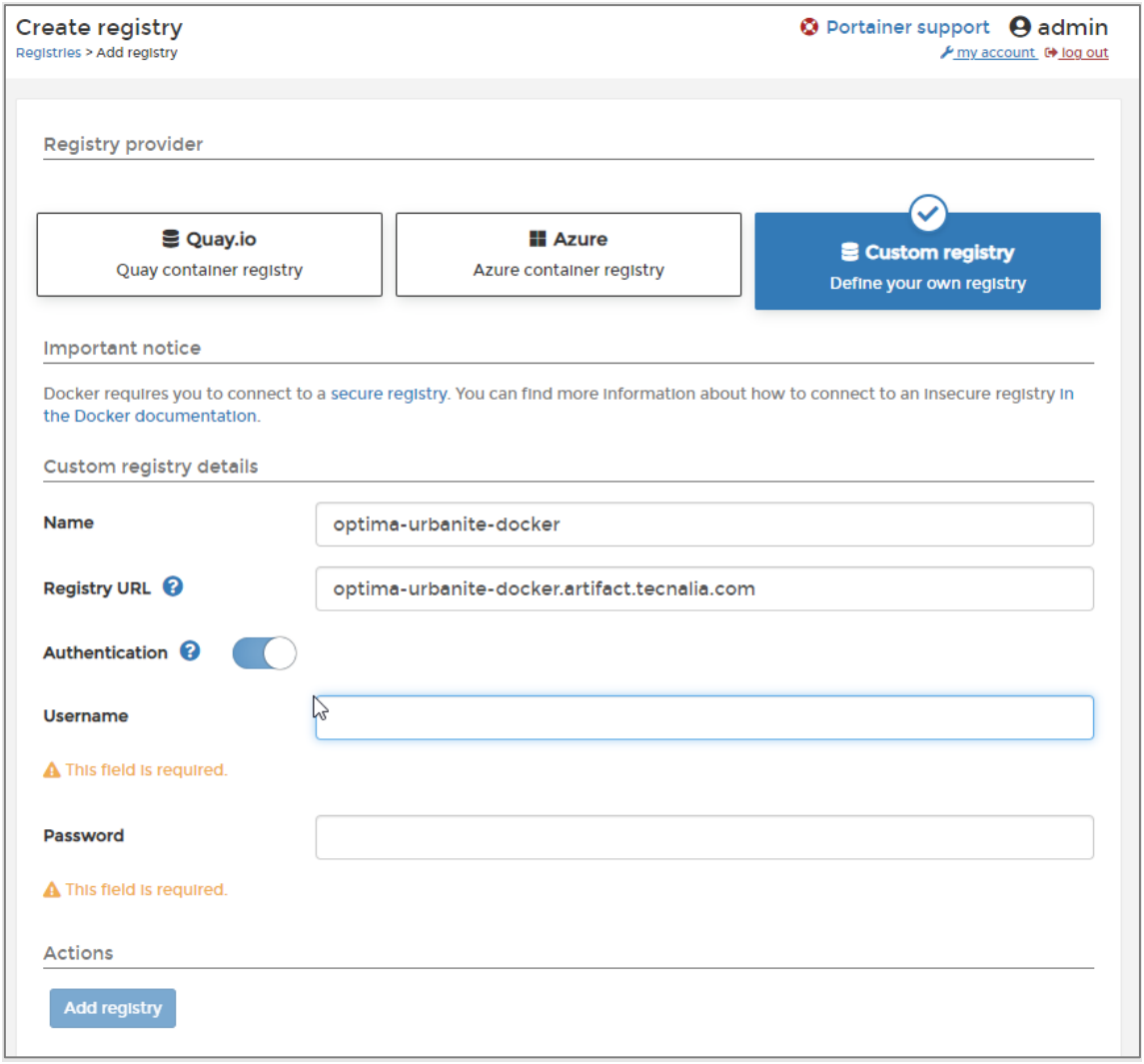


Figure 31. Portainer, registry details.

To publish an image, first, we find the image

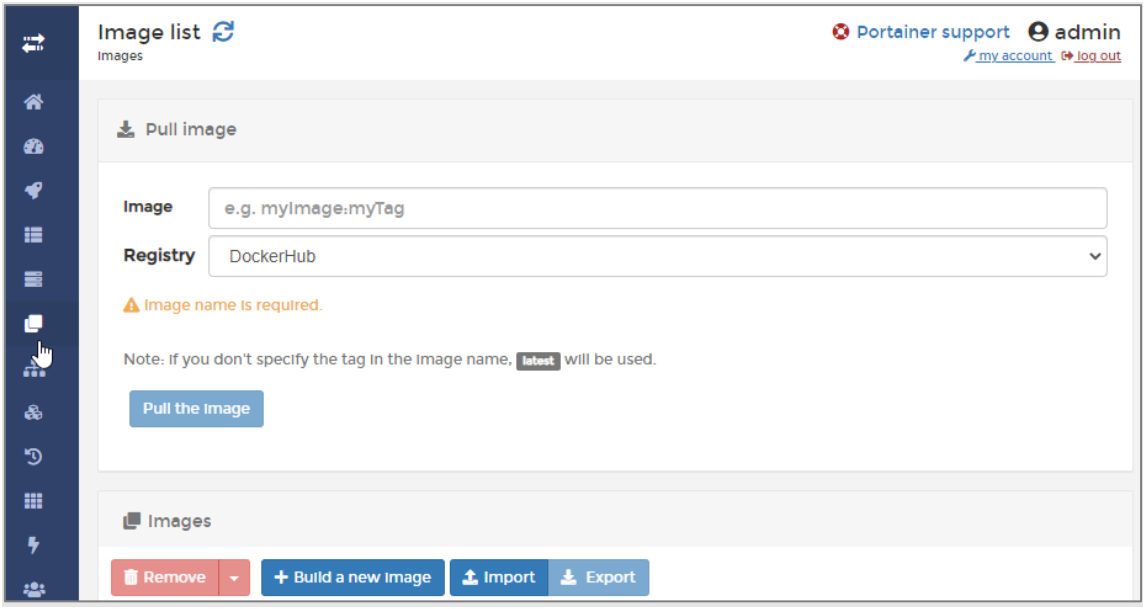


Figure 32. Portainer, image list.

Once we are in the image details, we can tag the image

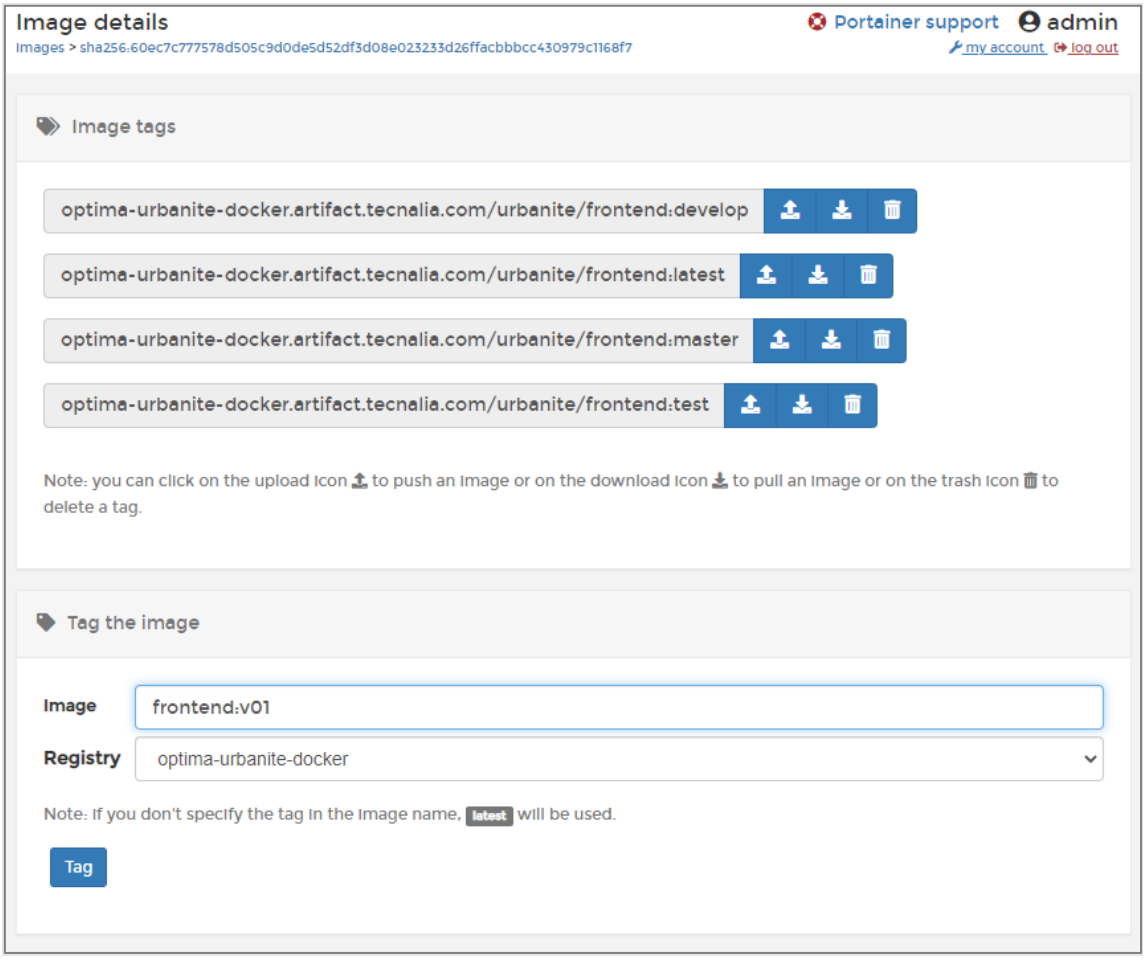


Figure 33. Portainer, creating an issue.

Once tagged, we can upload the image to the registry

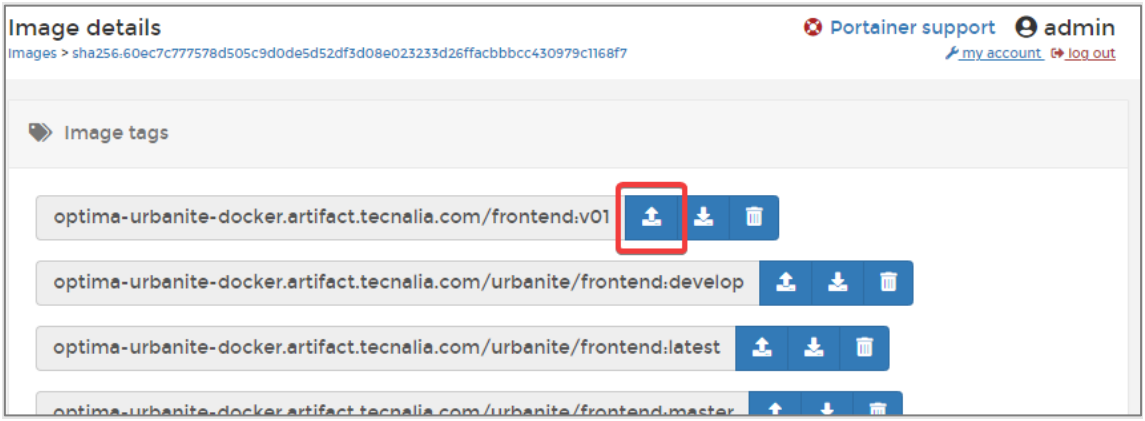


Figure 34. Portainer, image details.

To finish, independently of the procedure followed, we can find the image in the Artifactory (for example, https://artifact.tecnalia.com/ui/repos/tree/General/optima-URBANITE-docker-dev%2FURBANITE%2Ffrontend).

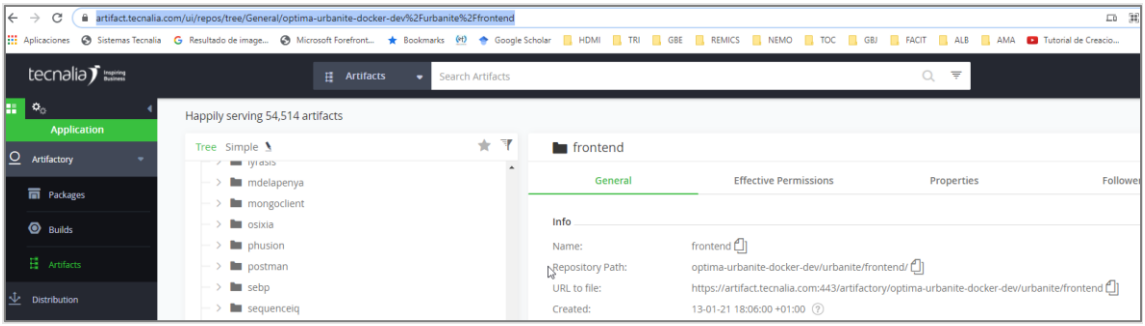


Figure 35. Artifactory.

EXPECTED OUTCOME:

A set of docker images, with the version number, in the Artifactory.

3.1.3 Pilot responsible

The Pilot Responsible is in charge of managing, planning, scheduling, and controlling software delivery towards the project pilots.

[NOTE: Some procedures performed by the pilot responsible in the master environment -e.g., debugging a component, creating an issue- are so equivalent to those followed by developers in the development environment that we consider not to repeat them here.]

3.1.3.1 Deploying/Updating a release to Pilots

The objective of this process is to instantiate the URBANITE platform/update the platform to new versions in the pilots.

[NOTE: This procedure covers the simplest scenario, where migrating the data structures to new schemas is not required. In case a data transformation is needed, a more complex, ad-hoc procedure with the close support of the integrator and the development team will be needed.]

REQUIRED:

Internet connection, a Linux server with a domain configured in a DNS, Git client, Docker server and CLI, Docker compose CLI.

PROCEDURE:

Clone the *URBANITE-deploy* repository (for a deploy, only needed the first time), or simply pull it (for an update, in case it has been already downloaded).

```
git clone https://git.code.tecnalia.com/URBANITE/private/URBANITE-
deploy
cd URBANITE-deploy
git pull
```

Next, edit the environment file (*.env,* in the root folder) to configure the system according to the specificities of the pilot. These can be details like:

- Version to deploy
- Details of external services to use, if any
- Credentials, if required
- The domain name
- Location of persistent storage in case it is configurable

```
vi .env
```

Once the repo has been downloaded and configured, we force the download of the images of the docker compose:

```
export COMPOSE_FILE=docker-compose.yaml:docker-compose-
expose.yaml:docker-compose-redirect-http.yaml
docker-compose pull
```

Once the images are downloaded, we run the docker compose:

```
export COMPOSE_FILE=docker-compose.yaml:docker-compose-
expose.yaml:docker-compose-redirect-http.yaml
docker-compose up -d
```

This should produce a pilot platform instance up and running. You can check it accessing to the corresponding URL.

EXPECTED OUTCOME:

An instance of the URBANITE platform running in the pilot.

3.2 Maintenance procedures

This section describes the procedures for the Integrator and the Maintenance responsible roles. A separate section will be provided for each of these roles.

3.2.1 Integrator

3.2.1.1 Component integration/update

The objective of this activity is to add an additional component to the URBANITE platform or update it afterwards.

[NOTE: The procedure for updating a component is very similar, only excepting the steps to include additional submodules and dockerfile definition (because they should be included since the first integration of the component).]

REQUIRED:

Internet connection, Git client software, Docker CLI, Docker compose.

PROCEDURE:

Download the *URBANITE-deploy* repo (with "clone") or, in the case is already downloaded, update it (with "pull").

```
git clone https://git.code.tecnalia.com/URBANITE/private/URBANITE-
deploy
cd URBANITE-deploy
git pull --recurse-submodules
```

(- - Only-first-integration - -)

Next step differs, depending on if the component to be added contains the docker container definition or not:

a) Component with Dockerfile: Add it as a submodule under the git folder.

```
git checkout develop
git pull --recurse-submodules
git checkout -b feature/integrate_[component_name]
git submodule add [url_of_the_repo] git/[component_name]
```

b) Component without Dockerfile:
   1. Create the subfolder for the component, and
   2. Add the source under *git/[component_name]/source*

```
mkdir git/[component_name]
explorer .
(copy the source files)
```

3. Together with the developers of that component, design the dockerfile. This may involve adding additional components.

```
vi git/[component_name]/Dockerfile
```

Besides, we can edit the *.gitmodules* file of the *URBANITE-deploy* repository to make the repo relative, if the domain of the repository is the same.

```
vi .gitmodules
```

For example, in the case a), the addition to the file should be:

```
[submodule "git/[component_name]"]
        path = git/[component_name]
        url = ../ [component_name].git
```

(- - End-first-integration - -)

Together with the development team, we need to identify all the configuration parameters, the endpoints and the persistence requirements. The configuration parameters include parameters to be modified in each pilot or platform instance and parameters oriented to the integration with other components. The first ones should be placed in the *.env* file, while the others can be placed in the *docker-compose* file directly. We add the docker-compose configuration in the main *docker-compose* file.

```
vi docker-compose.yml
```

```
frontend:
  depends_on:
    - traefik
  image: ${DOCKER_REGISTRY_PREFIX}urbanite/frontend:${URBANITE_VERSION:?err}
  build:
    context: git/urbanite-ui
    dockerfile: Dockerfile
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.frontend.rule=Host(`${SERVER_HOST:?err}`) && PathPrefix(`/ui`)"
    - "traefik.http.routers.frontend.entrypoints=websecure"
```

And we include the default values for the environment parameters at .env file

```
vi .env
```

```
# Reference documentation
# https://docs.docker.com/compose/environment-variables/

DOCKER_REGISTRY_PREFIX=optima-urbanite-docker.artifact.tecnalia.com/

# https://docs.docker.com/compose/reference/envvars/#compose_file#compose_project_name
COMPOSE_PROJECT_NAME=urbanite-develop

# variables for urbanite.dc

URBANITE_VERSION=latest

SERVER_HOST=urbanite.tri.dev
CERTIFICATE_SIGNING_KEY_PASSPHRASE=
ACME_CONFIG=
ADD_DEFAULT_CA=true

HTTPS_PORT=8443
```

We can also include a testing component to verify that the deployed component is working properly. This component should be implemented together with the development team. This component will be placed in a different file.

```
vi docker-compose-tests.yml
```

```
  newman.test:
    depends_on:
      - traefik
      - frontend
    build: tests/newman
    environment:
      - ENVIRONMENT=docker
      - SERVER_HOST=${SERVER_HOST}
      - HTTPS_PORT=${HTTPS_PORT:?err}
```

The resulting configuration can be tested locally.

NOTE: we can customize environment variables to test different configurations: master, develop or any-branch. For example, for master:

```
export URBANITE_VERSION=master
export SERVER_HOST=URBANITE.localhost
export HTTPS_PORT=443
export COMPOSE_FILE=docker-compose.yaml:docker-compose-
expose.yaml:docker-compose-redirect-http.yaml
docker-compose build
docker-compose up -d
```

EXPECTED OUTCOME:

A new component/updated component in the URBANITE platform.

3.2.1.2 Component removal

The objective of this procedure is to remove a component from the URBANITE platform. This is achieved by removing the submodule and updating the dockerfile to delete all its references. This may also require modifying the tests.

REQUIRED:

Internet connection, Git client, Docker CLI, Docker compose.

PROCEDURE:

Download the URBANITE deploy repo, or update in case it is already downloaded

```
git clone https://git.code.tecnalia.com/URBANITE/private/URBANITE-deploy
cd URBANITE-deploy
git pull
```

Delete the relevant section from the *.gitmodules* file.

```
vi .gitmodules
```

```
1 [submodule "git/urbanite-ui"]LF
2     path = git/urbanite-uiLF
3     url = ../urbanite-ui.gitLF
```

Stage the *.gitmodules* changes

```
git add .gitmodules
```

Delete the relevant section from .git/config.

```
vi .git/config
```

Run git rm --cached path_to_submodule (no trailing slash).

```
git rm –cached git/frontend
```

Run rm -rf .git/modules/path_to_submodule (no trailing slash). Commit the changes.

```
rm –rf .git/modules/ path_to_submodule
git commit –m "Removed submodule "
```

Delete the now untracked submodule files rm -rf path_to_submodule (in our example, “frontend”)

```
rm –rf git/frontend
```

Remove the relevant sections from docker-compose and .env

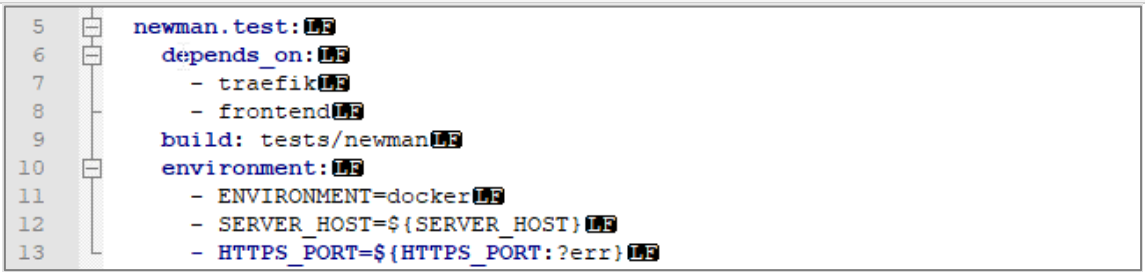
```
vi docker-compose.yml
```

```
26  frontend:CRLF
27    depends_on:CRLF
28      - traefikCRLF
29    image: ${DOCKER_REGISTRY_PREFIX}urbanite/frontend:${URBANITE_VERSION:?err}CRLF
30    build:CRLF
31      context: git/urbanite-uiCRLF
32      dockerfile: DockerfileCRLF
33    labels:CRLF
34      - "traefik.enable=true"CRLF
35      - "traefik.http.routers.frontend.rule=Host(`${SERVER_HOST:?err}`) && PathPrefix(`/ui`)"CRLF
36      - "traefik.http.routers.frontend.entrypoints=websecure"CRLF
```

```
vi .env
```

We can also be required to adjust tests.

```
vi docker-compose-tests.yml
```



The resulting configuration can be tested locally.

NOTE: we can customize environment variables to test different configurations: master, develop or any-branch. For example, for master:

```
export URBANITE_VERSION=master
export SERVER_HOST=URBANITE.localhost
export HTTPS_PORT=443
export COMPOSE_FILE=docker-compose.yaml:docker-compose-
expose.yaml:docker-compose-redirect-http.yaml
docker-compose build
docker-compose up -d
```

EXPECTED OUTCOME:

Running URBANITE platform without the deleted component.

3.2.1.3 Environment migration

The objective is to migrate the integration environment to a different server. This implies installing the required CI/CD tools in the new server and install and configure there a runner.

REQUIRED:

Internet connection, a new Linux server.

PROCEDURE:

Login into the server and install Git, Docker and Docker-compose. Then, download a dockerized runner configured for URBANITE by cloning the prepared repository in the URBANITE GitLab.

```
git clone https://git.code.tecnalia.com/URBANITE/private/URBANITE-gitlab-runner-deploy.git
cd URBANITE-gitlab-runner-deploy
```

In GitLab, go to the *URBANITE-deploy* project and check the runner registration code (in Settings > CI/CD > Runners, https://git.code.tecnalia.com/URBANITE/private/URBANITE-deploy/-/settings/ci_cd).

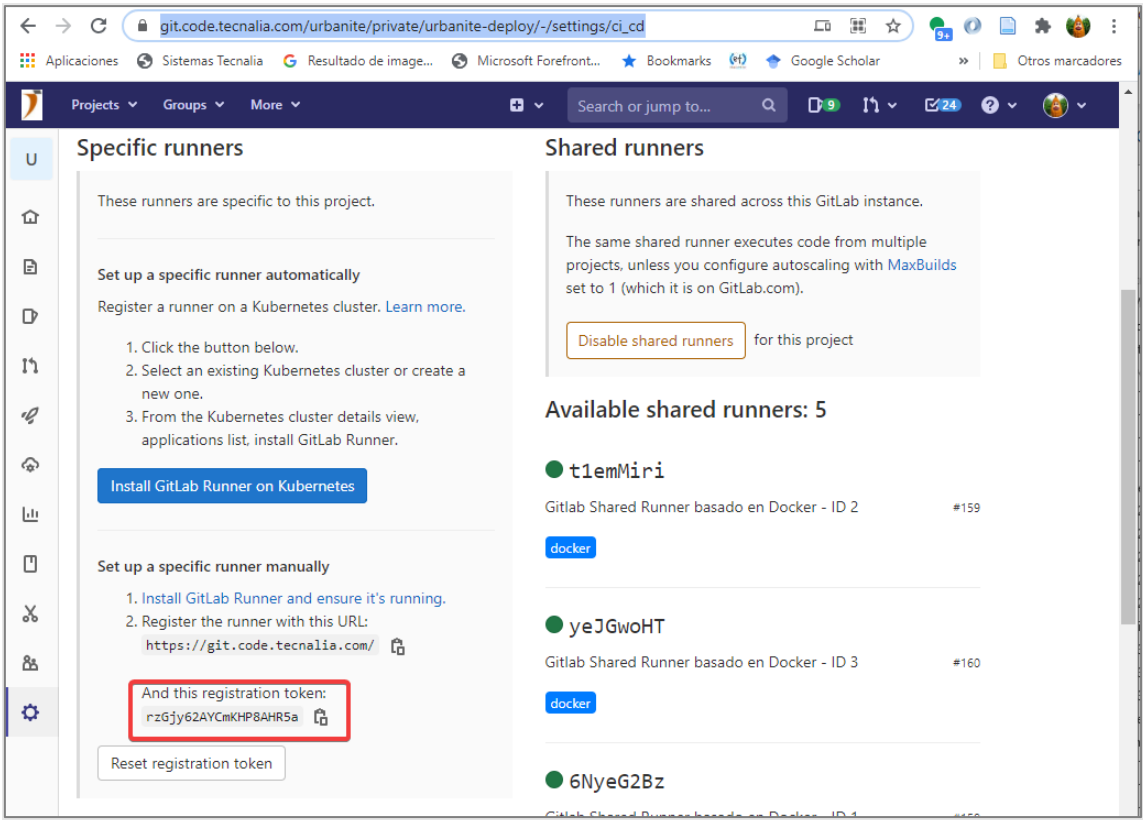


Figure 36. GitLab, runners.

On the server, configure the registration code and the registration tags in Linux environment variables (`REGISTRATION_TOKEN` and `REGISTRATION_TAGS` respectively), and launch the following docker-compose order to install and run the runner's docker:

```
export REGISTRATION_TOKEN=rzGjy62AYCmKHP8AHR5a
export REGISTRATION_TAGS= any-branch,develop,docker,docker-compose,master,URBANITE,CHANGE_THIS_TO_IDENTIFY_THE_SERVER
docker-compose up -d
```

After that, configure the domain to point to the new server. This can imply either:

a) the creation of a new domain and pointing it to the new server (This implies adjusting the "SERVER_HOST" variable of the URBANITE-deploy project.)

b) changing the IP to which the old domain is pointing to the IP of the new server.

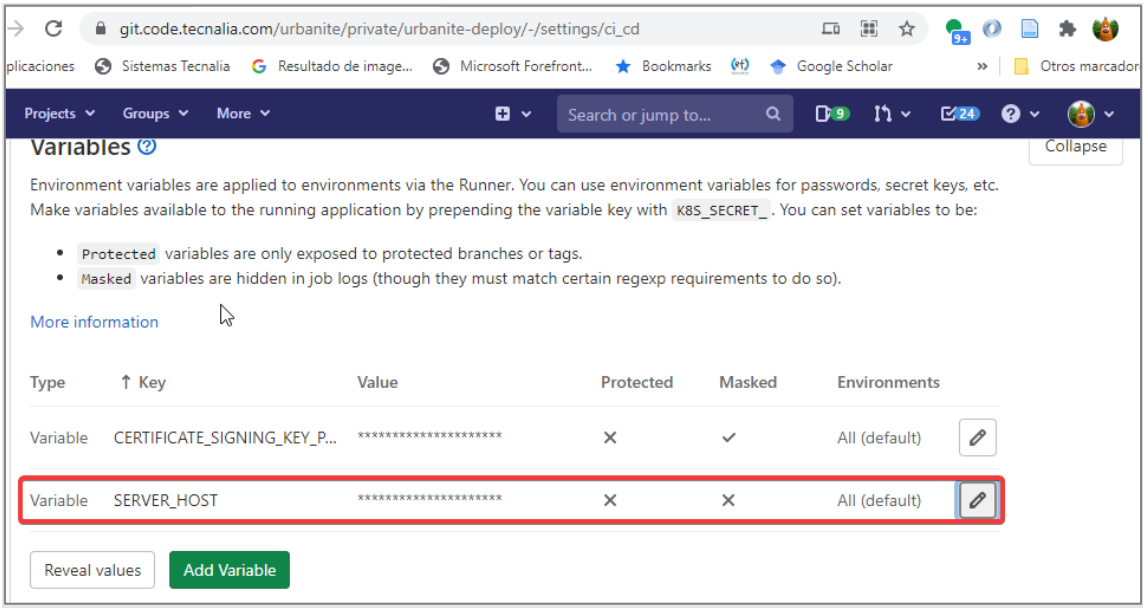


Figure 37. GitLab, variables.

If we want, we can force the pipeline to run, and build and deploy the system (https://git.code.tecnalia.com/URBANITE/private/URBANITE-deploy/-/pipelines)

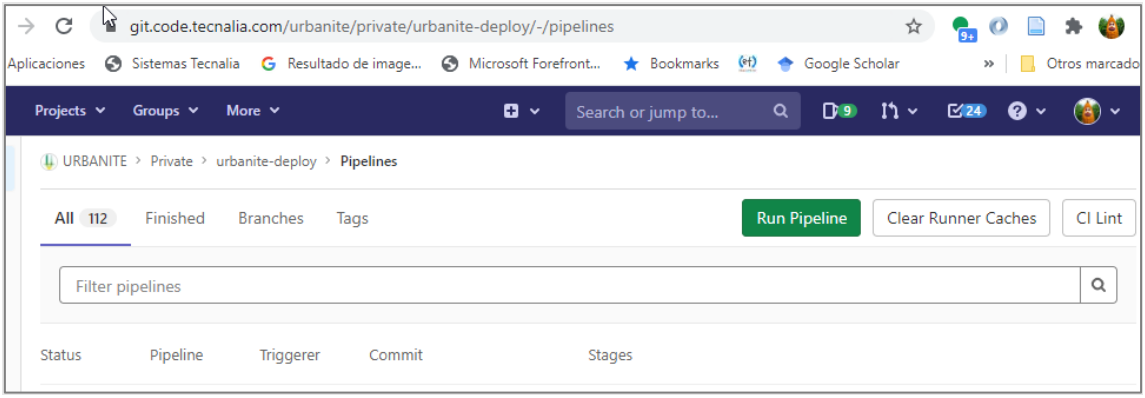


Figure 38. GitLab, forcing a pipeline to run.

Once the new runner is active, we can remove the old one

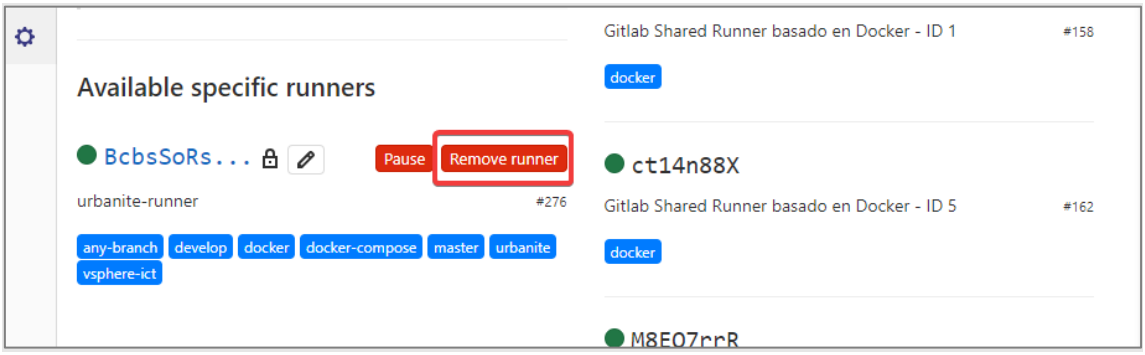


Figure 39. Portainer, removing a runner.

EXPECTED OUTCOME:

A running runner and a new integration environment that will be waiting for the pipelines from GitLab.

3.2.1.4 Version control system migration

The objective is to migrate the version control system from the Tecnalia GitLab to another one.

REQUIRED:

Internet connection, Git client.

PROCEDURE:

The procedure is not exhaustive, because it heavily depends on the new system. And if the new system is GitLab based or not. If not, it implies changing the continuous integration procedure.

The first step is to decide the new system to be used. In case it is a service, formalize the contract; In case it is software, install it on a server.

The next step is to migrate the repos to the new system.

Then we need to configure the continuous integration. If it is GitLab, we only have to configure it as we have done previously the original system. If it is not GitLab, then we will have to identify the new CI system: Jenkins, circle-ci, etc. Some will imply to contract a service; others will imply to install the selected software in new resources. Then, we will need to migrate the *gitlab-ci.yml* to the new system syntax and approach.

Finally, we need to communicate to the consortium the new repositories URL.

EXPECTED OUTCOME:

A new version control system running and setup along the project.

3.2.2 Maintenance responsible

3.2.2.1 Server monitoring

The objective is to verify that the server(s) supporting the integration environments is not running out of computing resources.

REQUIRED:

Internet connection, SSH client.

PROCEDURE:

Login into the server, and once inside, check the storage, the memory and the processing.

EXPECTED OUTCOME:

Stable server.

3.2.2.2 Endpoints checking

The objective of this procedure is to verify that the platform endpoints are responding.

REQUIRED:

Internet connection.

PROCEDURE:

Check the list of endpoints using the script that will be provided by the integrator.

EXPECTED OUTCOME:

Evidence that the services in the platform are running.

3.2.2.3 Containers monitoring

The objective of this procedure is to check that the containers of the platform are running.

REQUIRED:

Internet connection, SSH client.

PROCEDURE:

Login into the server, and once inside, check the storage.

```
docker ps
```

EXPECTED OUTCOME:

Evidence that the containers of the platform are running.

4 Conclusions

Tecnalia, who leads Task 5.3 - *Continuous Integration and DevOps approach,* provides the DevOps infrastructure and is in charge of setting up the tools and managing the tasks during the integration tasks in URBANITE.

The DevOps infrastructure has been established during the first year of the project. It is described in the document, listing the different tools that compose it, how they are organized and configured to manage the development, integration, and validation stages of the software components to be implemented during the life cycle of the project.

The version control is based on the GitLab tool, with separate environments for development, integration and pilots. Regarding integration and validation, GitLab CI/CD is the tool selected to manage the deployment of the components, whereas Docker is the container technology used to achieve hardware independence.

The provided infrastructure itself is necessary but not enough. Without the procedures and rules that the user must perform to carry out the tasks, it would not be useful. So, the second part of the document presents the most habitual procedures that allow the users of the infrastructure –e.g. developer, integrator, infrastructure maintainer or pilot responsible- to manage the

DevOps tasks. These procedures have been explained, providing examples and code/command snippets to clarify the details when needed. The procedures have been grouped by role. Five roles have been defined, and more than twenty procedures are described to cover the full software life cycle.

The infrastructure provided, here described, along with the associated documentation provided, is expected to allow the adequate and fruitful delivery of the URBANITE software components and help to its deployment in the pilot sites.

5 References

[1] URBANITE consortium, «D5.3 Integration strategy,» 2019.

[2] GitLab, «GitLab,» [En línea]. Available: https://about.gitlab.com/. [Último acceso: July 2020].

[3] Git, "Git," [Online]. Available: https://git-scm.com/.

[4] «GitLab issues,» [En línea]. Available: https://docs.gitlab.com/13.0/ee/user/project/issues/. [Último acceso: July 2020].

[5] URBANITE Consortium, "URBANITE Annex 1 - Description of Action," 2019.

[6] GitLab, «GitLab CI/CD,» [En línea]. Available: https://docs.gitlab.com/ee/ci/. [Último acceso: 09 03 2021].