DXC.technology

CDP Documentation

Release 1.4.4

DXC.technology

Mar 11, 2019

CONTENTS

Latest update: 2019-03-11T12:30

CHAPTER

ONE

PREFACE

This is the official documentation for the DXC Container DevOps Platform (CDP) solution. If you are reading a printed copy, it might contains outdated information. The latest up-to-date documentation is always available on your CDP console in the *Information –> Documentation* tab.

The CDP documentation is currently composed by the following documents:

1. CDP User Guide
2. CDP System Administrator Guide
3. CDP Technical Documentation

The user guide contains information about how to use the system and how to publish your projects to the CDP catalog to start working within the CDP CI/CD/DevOps ecosystem.

The System Administrator Guide contains information about the overal architecture, infrastructure setup, CDP components installation and configuration, etc.

The Technical Documentation contains information about the system architecture, hardware and software requirement, instance installation and configuration, system maintenance, etc.

This is the “CDP User Guide”.

CHAPTER

TWO

CDP HIGH LEVEL OVERVIEW

The DXC Container DevOps Platform (formerly ECaaS) or CDP it is a solution that fully exploits the containers technology to enable:

- DevOps
- CI/CD
- Fast deployment of IT Services
- Reuse of application components
- Micro-services based application architectures
- Independence from the infrastructure
- Optimized consumption of IT resources
- Enterprise grade software defined infrastructure
- Improved quality of IT services
- Infrastructure as code
- Documentation as code

It is designed and built as an integrated platform to manage the complete lifecycle of IT Services using containers technology.

2.1 Guiding principles

The principles that are driving the creation and evolution of CDP are:

- The unit of delivery is the application service, with its all included applications components (Docker images)
- Use of a full integrated communication and interaction across development, deployment and operations (based on *ChatOps* model)
- Fully based on software defined model with pervasive use of automation and orchestration
- Modular architecture with dynamic API driven integration of system services
- Full independence from the underneath infrastructure (compute, storage, networking)

2.2 The high level architecture

The architecture of CDP is logically divided into three levels or *planes*, each of which provides services to the plane above and consume services from the plane below. The upper plane provides services to end users and applications.

More in detail:

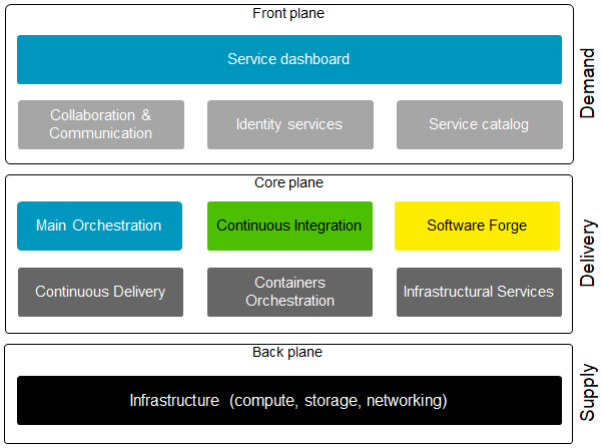

Fig. 1: Logical architecture

- The **back plane** provides the basic infrastructure resources needed for a CDP instance to work. The back plane interacts with one or more IaaS solutions providing resources using a software defined provisioning model
- The **core plane** provides the fundamental functions required to deliver *Continous Integration*, *Continuous Delivery* and *Delivery Management*. Moreover there is an global orchestration/coreography service for internal use that coordinates the activation of the various services
- The **Front plane** allow the end users and the applications to consume CDP services as well as the application stacks deployed on an CDP instance, using a console or the API set through an full fledged API Gateway (with full access control and API call metrics)

The various architectural components are built using specific software products/solutions, almost open source (some of them offer also a commercial support). They are effectively running as CDP services and integrated using API intermediation/aggregation.

The picture below shows the physical architecture of CDP.

User acces is granted from the Internet or corporate LAN through an external load balancer who routes selected traffic to UCP, DTR and Engine nodes (workers). All CDP infrastructure and application containers run on engine nodes, apart from cluster management and *Docker Trusted Registry* containers. Shared storage is provided by *Portworx* and *Minio* technologies.

UCP and DTR are configured in a hi-availability configuration in CDP, as shown in the picture below.

2.2.1 Everything runs in containers

A relevant characteristic of CDP is that all the components inside it run as containers, so the CDP itself is built up on containers technology using the same DevOps approach and through the iterative CI/CD model. In this way CDP is constantly updated with new functionality over the time.

2.2.2 A non-opinionated PaaS

CDP can also be seen as a non-opitionated PaaS in the sense that you can design, develop, build, test and run application stacks using whichever software technology you prefer that can run with Docker. In comparison other platforms there are no limits on the software development language or runtinme you can use.

2.2.3 Loosely Coupled integration across the CDP components

The integration of the various components that constitute CDP is done using a *loosely coupled* model using an API Gateway integration. This is a fundamental decision because the fast pace at which the software defined and container technology evolves requires CDP to evolve as well by adapting at those changes.

The integration of the various CDP services is performed dynamically using an API layer providing a practical way to swap in and out components, to add new ones and remove old one withouth a relevant effort and without impact

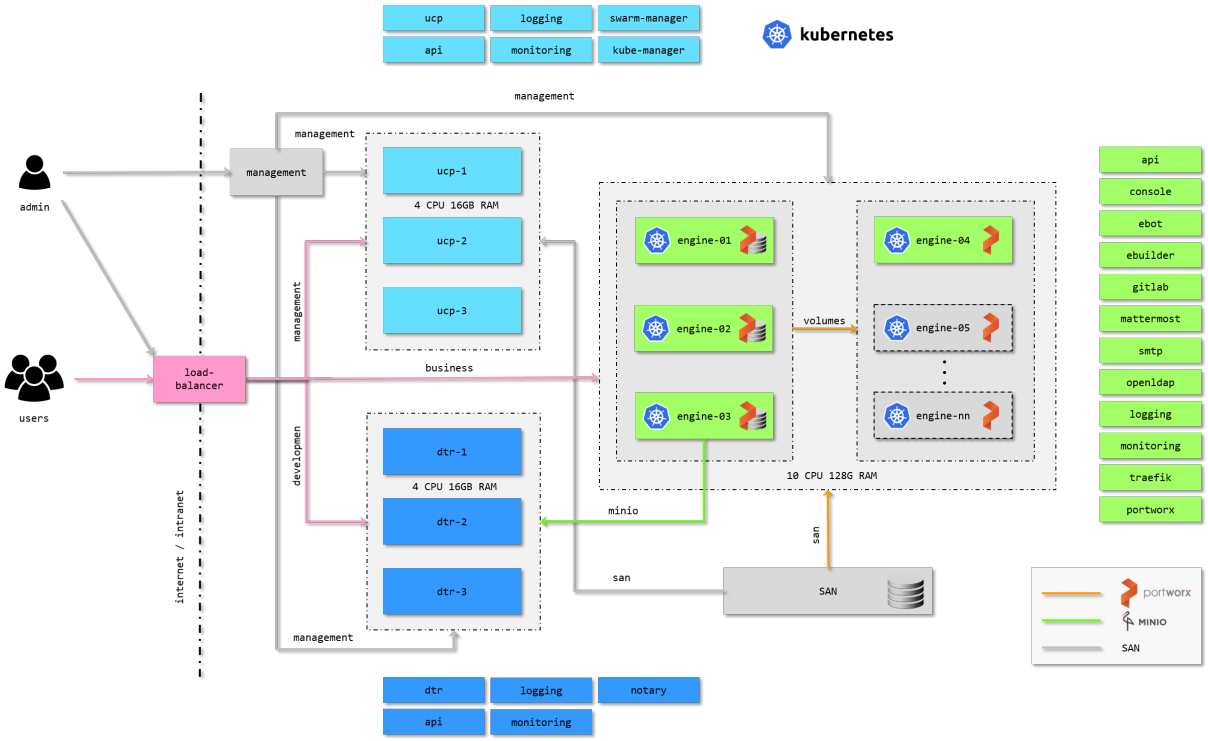

Fig. 2: CDP Physical Architecture

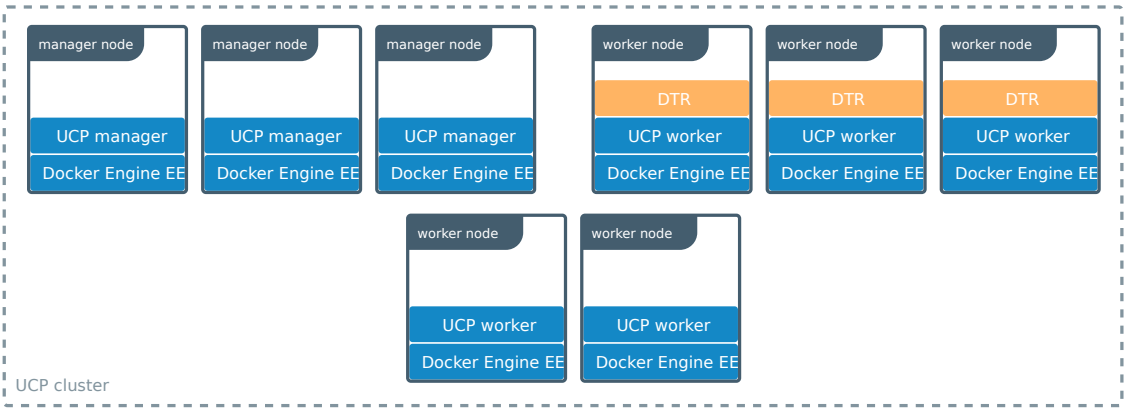

Fig. 3: UCP and DTR running in HA

on already deployed applications. This allow also the possibility to use in parallel components that are devoted to the same task. As an example it is possible to stand up two different platform as software forge environment like GitLab[1] and Github[2] or maybe to use at the same time Jenkins[3] and Drone[4].

The flexibily of this model allows to meet the specific needs of many clients in the enterprise world by adapting the platform to their unique requirements.

2.3 How CDP can be provided to clients

From a practical point of view CDP it can be provided following two possible model:

- **Platform as a Service**: fully managed - at the moment available only on Italy's SPC-Cloud (OpenStack Cloud) and Microsoft Azure although it can be easily deployed also on AWS Cloud as other Cloud supported by Terraform and Docker.
- **Platform on-prem**: on a client owned virtual (VMware, OpenStack) or bare metal infrastructure, fully configured and ready to use.

2.3.1 Accessing and using the platform

The access to the platform for end users and admninistrators is provided, through level of fucntionality, by:

- A web access console
- A specific set of API
- A command line interface tool

[1] https://gitlab.com/
[2] https://github.com/
[3] https://jenkins.io/
[4] https://drone.io/

CHAPTER

THREE

THE CDP CONSOLE

As descrived in the previous section, CDP is a comprehensive development platform which encompasses many technologies. The platform performs most of the work *under the hood*, usually triggered by events, either synchronously of asynchronously, APIs calls or explicit actions performed directly on the platform or on any of the many tools included in the solution.

A complete view on CDP is provided by the **Console**, a web based application which lets you see the systems status, the resource usage, the containers and application stacks running, perform management and administrative tasks if you have the required grants, see your pipeline's status and log messages, etc.

Thi section descrives the Console from a user perspective. The administrators view of the Console is described in the "CDP System Administrator Guide".

3.1 Accessing the platform instance

In order to work with CDP you must have suitable account. With it you can start your work by simply performing the login process. Just obtain your CDP instance URL and perform the login:

If the login is succesful, you will access the Console main page. Depending on your role and grants given you will see only the functions you can access. If you think you need access to administrative functions, please ask your system administrator.

After the login you will presented with the CDP web based console. It is mainly a web portal that has main dashboard that provides a view about the current state of the CDP instance:

- number of running application stacks
- number of running application services
- number of running containers
- number of Docker images
- number of software defined networks
- number of storage volumes
- number of cluster nodes

by clicking on each of the badges you can get the full list of resources. In the picture below you can see the details of the services currently running on the instance. Using the controls available in the window you can sort, paginate and filter results.

There are also three main gauges in the Console home page that show the current status of the cluster, used/available capacity of RAM, CPUs and storage Storage.

In the left hand side of the window there is a menu which you can use to access all Console functions:

The menu is divided into sections and contains only the items you have access to. The menu is structured as a tree: by expanding intermediate items you will reveal other functions.

Within the Console, all links to external tools are marked with a square containing an arrow pointinig outwards. You will se examples later in this section.

Fig. 1: User login

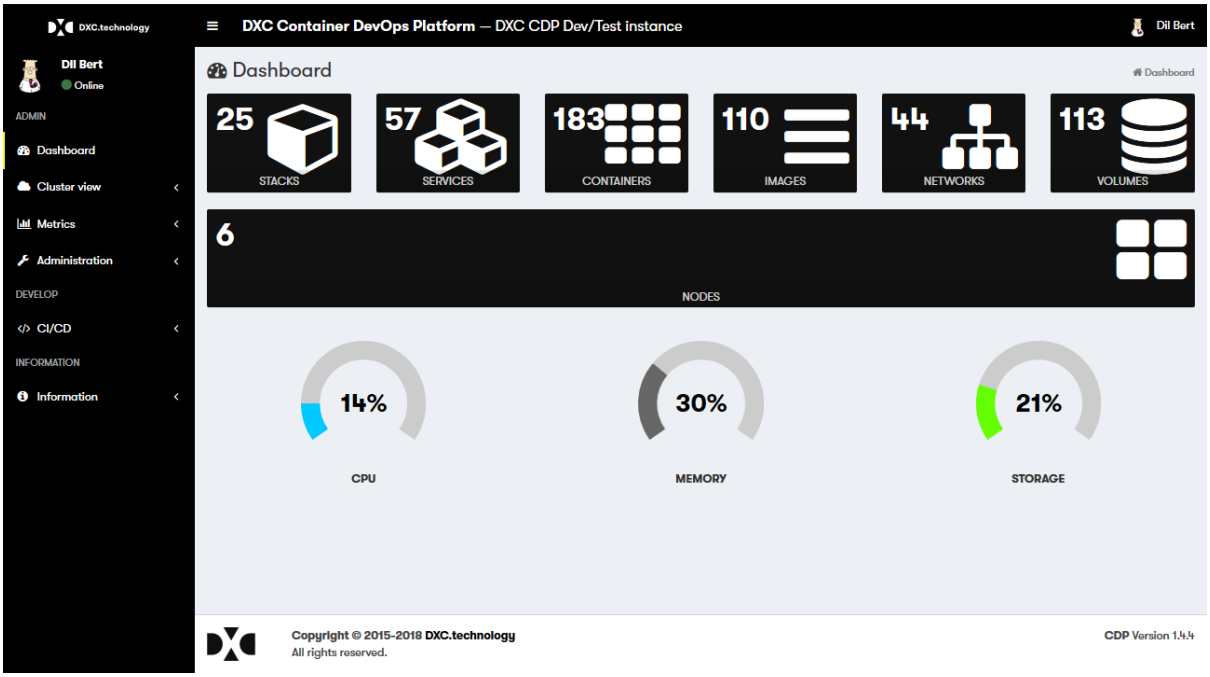

Fig. 2: CDP console and dashboard

Services

Show 10 entries Search:

Id	name	stack
p0w6f9ewnl	elk_elasticsearch	elk
pn5zd4jdpi	elk_cerebro	elk
r6nk3hpwa7	api_apidb	api
rj4wse1bgw	ecaas_demo_master_demo_demoapp	ecaas_demo_master_demo
rkzclxfcz2	ecaas_docint_master_docint_cdp-doc-solution	ecaas_docint_master_docint
rufjhprk25	smtp_smtp	smtp
s4nsfcpepx	ebot_ebot	ebot
sdsbyuyrka	api_api	api
smbp1oguce	openldap_ldap	openldap
t1zr67x17g	monitoring_prometheus	monitoring

Showing 31 to 40 of 54 entries Previous 1 2 3 4 5 6 Next

Fig. 3: Services detail

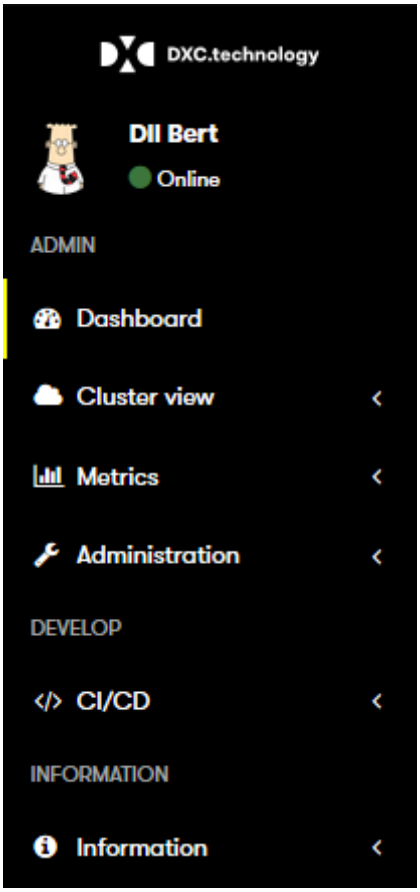

Fig. 4: Console menu

3.2 Cluster view and network topology

If you click on the **Cluster view** you can get access to the **Network topology** diagram, that provides a real-time representation of the Docker networks.

This view is useful to troubleshooting connectivity problems between your containers and to understand how all services cooperate each other in the bigger architecture.

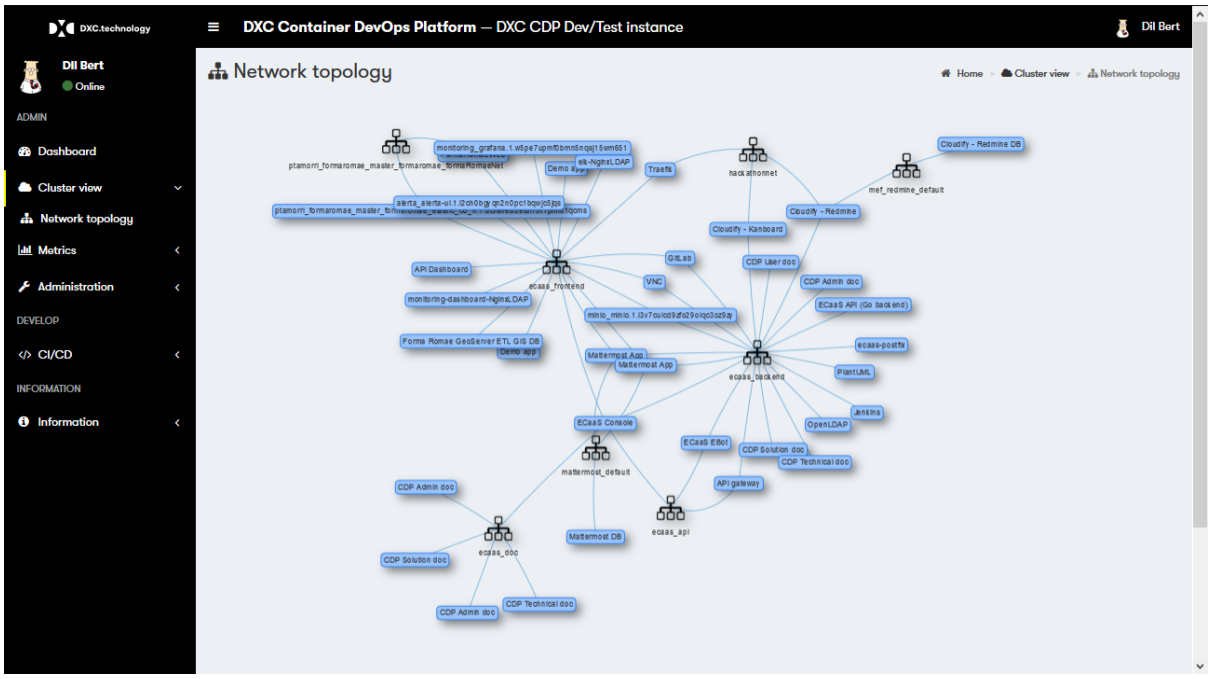

Fig. 5: Network Topology

You can interact with the diagram by panning, zooming, moving the entire diagram or a single node or network in order to see better the details.

For example, in the picture below there is a magnification on the Mattermost application stack: as you can see, the `Mattermost DB` container is only connected to the `mattermost_default` internal network and therefore only `Mattermost App` containers, connected on that network, can access the DB. The `Mattermost App` containers, in turn, are connected to the wider `ecaas_frontend` network.

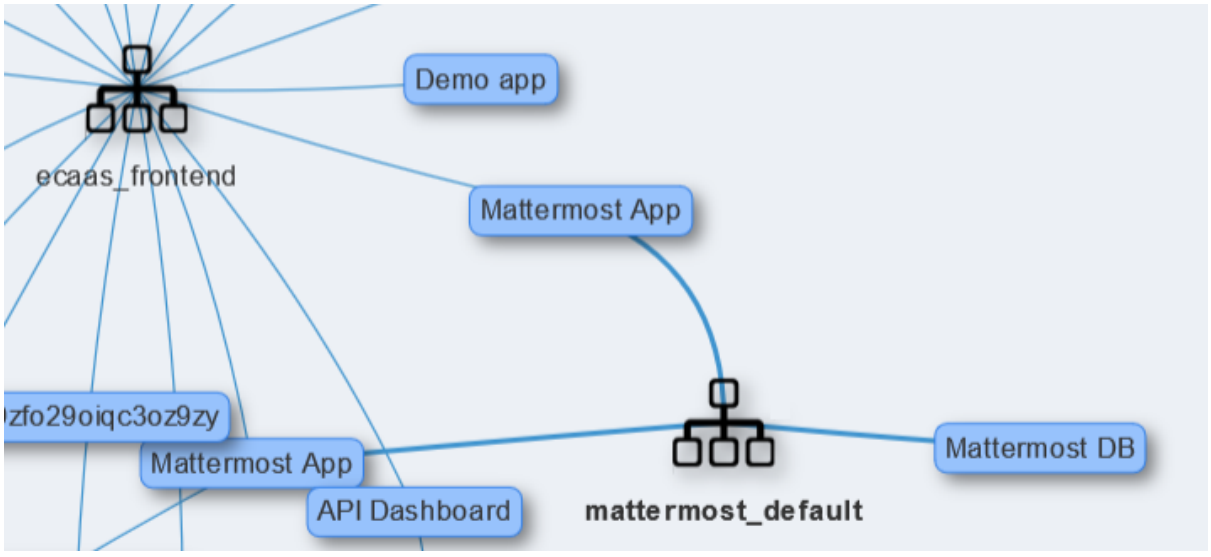

Fig. 6: Network Topology - details

A good design of your internal software defined networks helps you to reduce security breach risks.

3.3 Usage trends

If you click on the **Metrics** entry (still on the left side menu) you will get access to the Cluster usage trend. A 3D diagram will be brought up that can shows the resources used in a specific range of days and in the range of hours in a day by selecting them in the selection list: Containers, Images, Networks, Services and cluster Nodes.

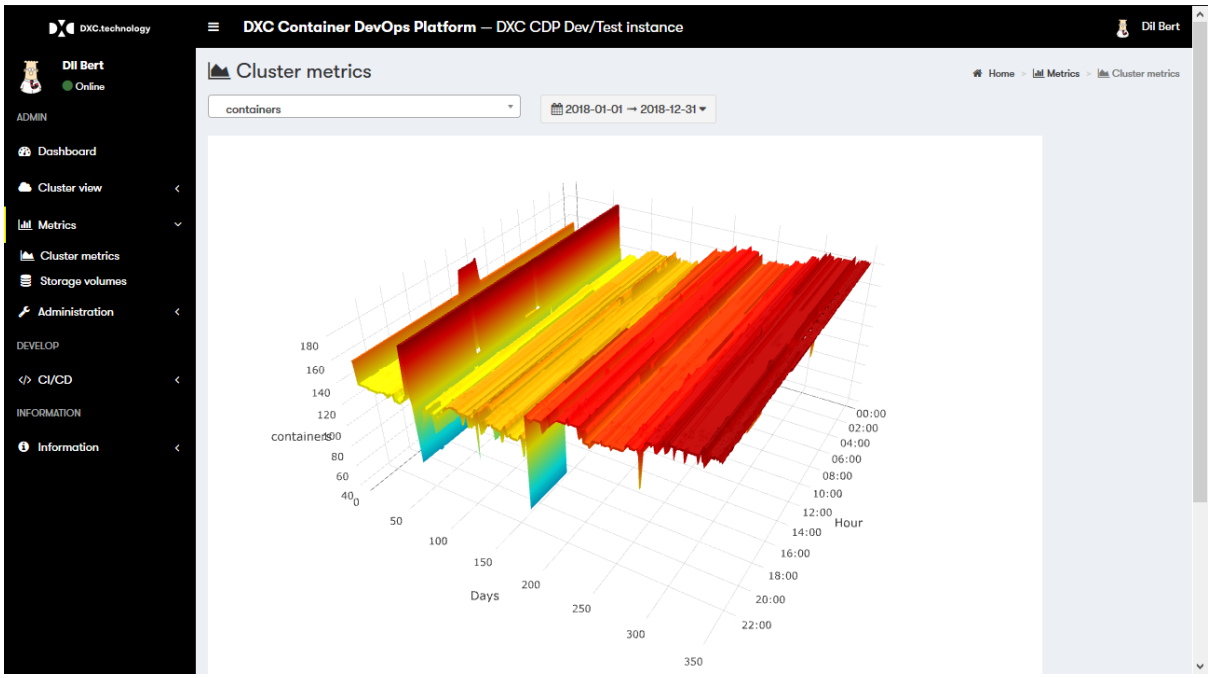

Fig. 7: CDP usage trend

By selecting the time range interval you can reduce or increase the data exploration period: last 7 days, last 60 days, last 180 days or a custom date range.

With the mouse the graph can be moved, zoomed and rotated for better readability.

These trends can be useful to determine the optimal capacity of the cluster so that you can plan for additional nodes or just to consider reducing the number of nodes in the cluster. However CDP can provide also the functionality to implement a cluster auto-scaling thanks to the specific API for this purpose.

3.4 Storage volumes

Cliccing on the **Storage volume** entry brings you to a page showing the state of storage volumes.

For each volume defined on the CDP instance, the following informations are reported:

- **volume name** as it can be referenced in your application stacks
- **driver** if either *local*, for local storage or NFS mounted volume mounted with *bind* option, or *portworx* for enterprise-grade distributed storage; more drivers could be present if configured in your instance
- **usage %** of total space used
- **used/available** size metrics in human readable format

As seen for the dashboard metrics detail windows, the result can be sorted, paged and filtered by using the controls available in the table header and footer.

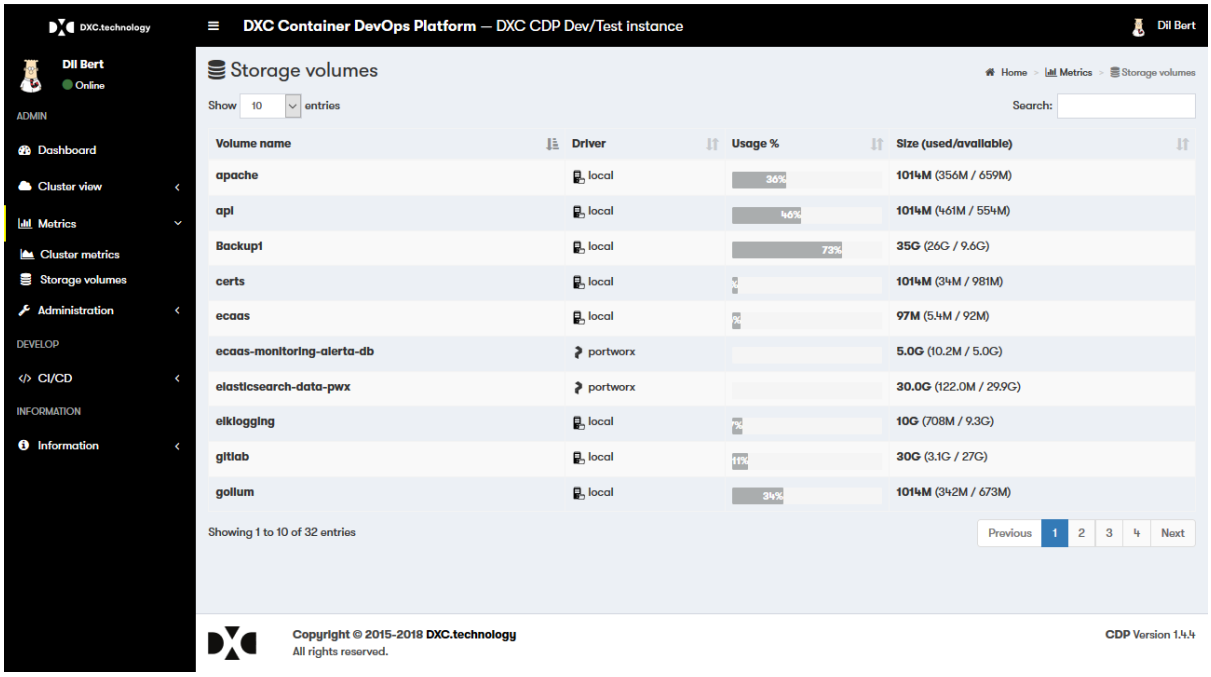

Fig. 8: Storage volume statistics

3.5 Administration

You can easily administer CDP from the lef side menu by accessing the **Administration** entry. However the access to the administration functionality depends on the *grants* a specific user has. The CDP instance administrator can give full access to the console functions or limit that access on the basis of the role that a specific user plays in the CDP instance. For example the administrator has full access while maybe a developer could only work with the **CI/CD** functionality.

The complete administration menu is showed below, as visible to an user with `admin` grant:

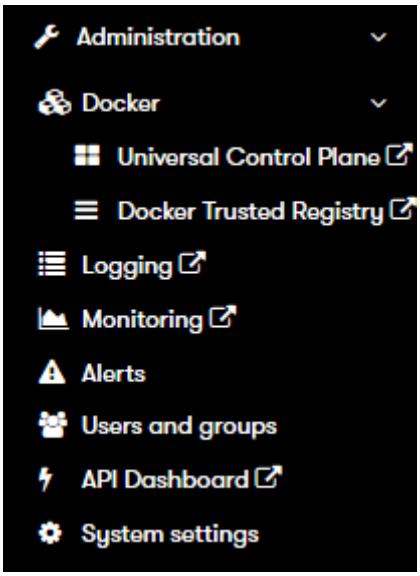

Fig. 9: System Administration menu

Administration tasks are described in the "CDP System Administrator Guide", this guide only contains a high level overview of some functionalities.

3.5.1 Docker

This menu section contains links to the following Docker Enterprise Edition applications and tools:

- **Universal Control Plane (UCP)**
- **Docker Trusted Registry (DTR)**

This section is visible only to users with `ucp` grant. By clicking on those links you will get redirected to the selected tool.

The resources showed in those tools will vary depending on your role. For example, if you are not an admin you will see only the resources (i.e. nodes, containers, networks, volumes, images, etc.) you created or which someone gives you a specific grant, for example based on the teams you are included in.

Universal Control Plane

The UCP (Universal Control Plane) is the enterprise-grade cluster management solution from Docker. It provides an integrated web based console and can be also controlled through a well-defined set of API calls.

With Docker, you can join up thousands of physical or virtual machines together to create a container cluster that allows you to deploy your applications at scale. Docker Universal Control Plane extends the functionality provided by Docker to make it easier to manage your cluster from a centralized place all of the computing resources you have available, like nodes, volumes, networks, configurations, secrets, etc.

Docker UCP has its own built-in authentication mechanism and integrates with the CDP LDAP service (either internal or external); therefore access to the UCP is done using the same CDP user credential. However the specific UCP organization, team, roles and collections are managed exclusively within UCP being them very specific to the UCP RBAC model and applicable only to Docker Enterprise Edition (with Swarm or Kubernetes orchestration).

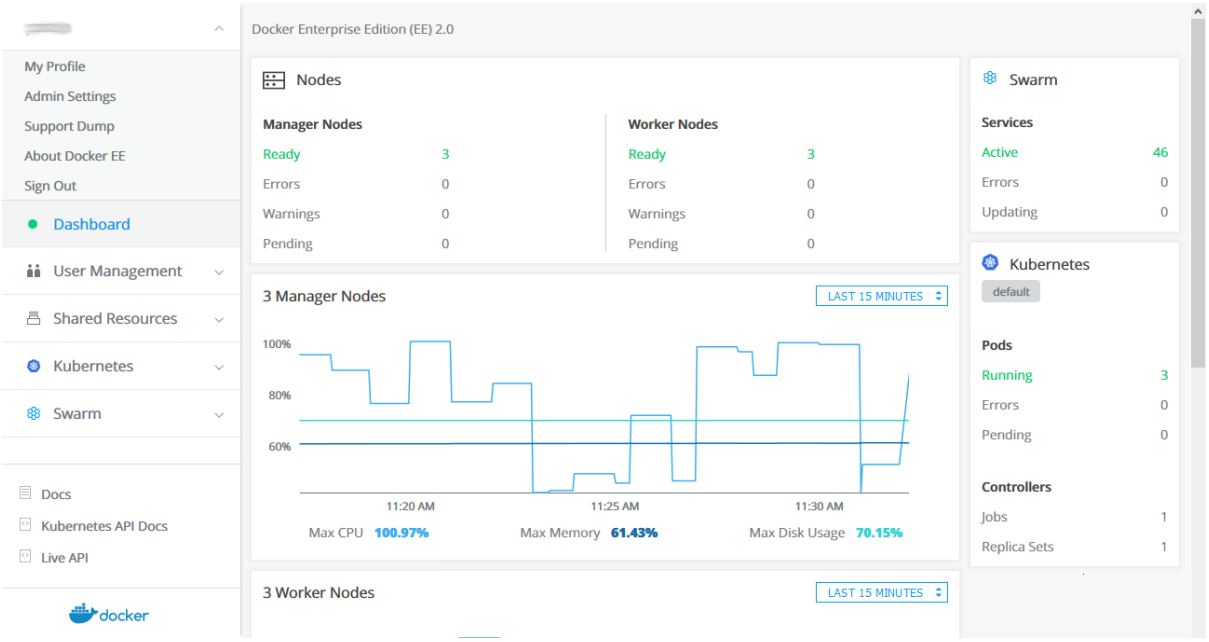

Fig. 10: Universal Control Plane (UCP)

With the UCP you can:

- create and download your *bundle*; a bundle is an archive containing certificates and your private keys needed to use the cluster
- see your deployed services
- create/destroy resources, for example application stacks, containers, etc.
- perform many other tasks

For more information about UCP, its functionalities and DEE RBAC model, please see the Docker documentation in the *References* section at the end of this guide.

Docker Trusted Registry

Docker Trusted Registry (DTR) is the enterprise-grade image storage solution from Docker. It is at the core of the CDP platform to provide a secure store to manage the lifecycle of Docker images you will use in your application stacks.

The main characteristics of the DTR are the following:

- **High availability:** DTR is highly available through the use of multiple replicas of all containers and metadata such that if a node where the DTR container run fails, DTR continues to operate and can be repaired.
- **Efficiency:** DTR has the ability to cache images closer to users to reduce the amount of bandwidth used during docker pulls and image build operations. It also has the ability to clean up unreferenced manifests and layers.
- **Built-in access control:** DTR uses the same authentication mechanism as Docker Universal Control Plane. Users can be managed manually or synched from the CDP LDAP. DTR uses Role Based Access Control (RBAC) to allow you to implement fine-grained access control policies for who has access to your Docker images.
- **Security scanning:** DTR has a built in security scanner that can be used to discover what versions of software are used in your images. It scans each layer and aggregates the results to give you a complete picture of what you are shipping as a part of your stack. Most importantly, it co-relates this information with a vulnerability database that is kept up to date through periodic updates. This gives you a consistent insight into your exposure to known security threats.
- **Image signing:** DTR ships with Docker Notary built in so that you can use Docker Content Trust to sign and verify images. For more information about managing Notary data in DTR see the DTR-specific notary documentation available from the Docker web site.

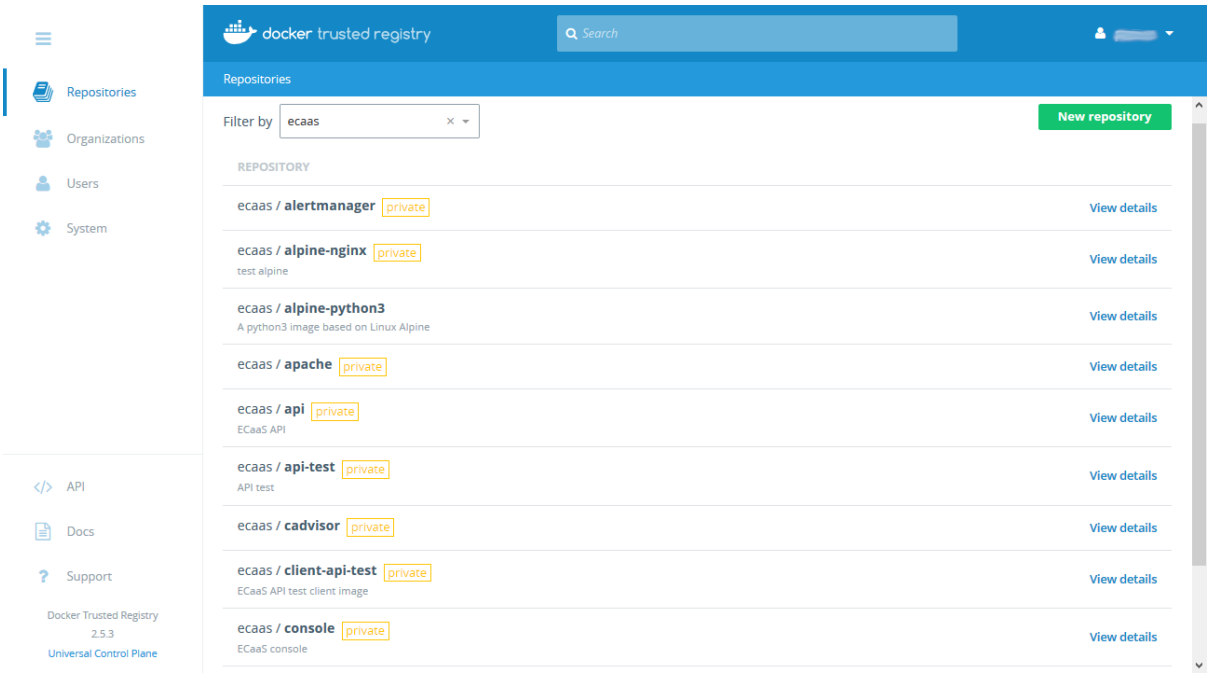

Fig. 11: Docker Trusted Registry (DTR)

With the DTR you can:

- see your images and read deploy tag names
- analyse your images, for example view the layers or start a vulnerability scan if configured in your instance

- remove your images repositories
- perform many other tasks

For more information about DTR and its functionalities model, please see the Docker documentation in the *References* section at the end of this guide.

3.5.2 Logging

Every container deployed on CDP automatically got standard output and standard error logged on the central logging system which at the time of writing is **Kibana**. Access to logging is not limited to admins, if you have console access you can also access the logging system.

The Kibana dashboard is swown in the picture below:

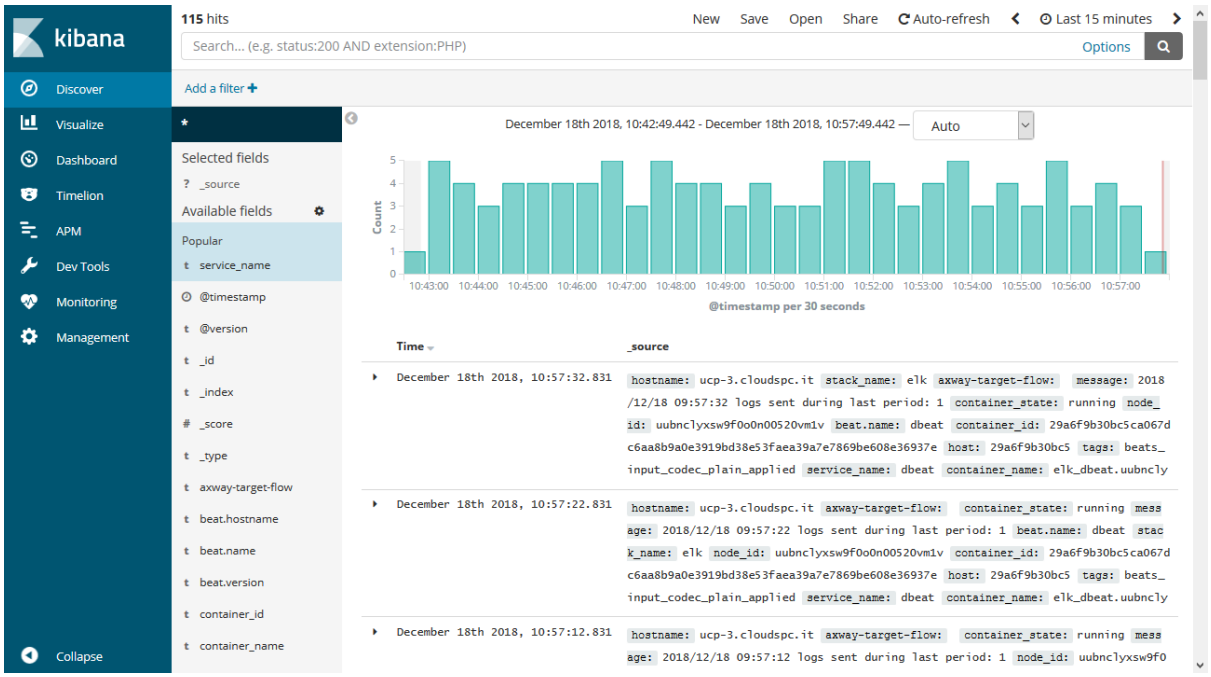

Fig. 12: Kibana discover dashboard

For information about Kibana query language and other functionalities, please read Kibana documentation.

3.5.3 User Administration

There is a specific function, **Users and Groups** (available only to the users with the `admin` grant) that provides a way to add, remove, update users and giving them a specifc grant.

CDP Access Control model

The CDP provides a fine-grained access control mechanism that is used to control the access to the various internal services as well as to the CDP API. In this way when a specific CDP user can see and access only the specific set of services for which she/he has a grant.

Grants can only be assigned to users by a system administrator or by a user with access to the authorization section.

If you need to access a section/service of CDP, please ask your system administrator to give you the specified grant.

User access privilege control

The *grants* enable the access to the various services within an CDP instance. The number of available *grants* is not a static list but can change over time because CDP is continuously updated and can be extended in terms of functionalities. At the time of writing the following grants are available:

- `admin`: this provides administrator privilege to the user (it must used with great care as the user can change everything with relevant consequences: *"With great power comes great responsibility"*)
- `console`: this grant give a user access to the CDP console
- `gitlab`: with this grant a user can get access to GitLab
- `ucp`: with this grant the user can access UCP and DTR consoles that are the key components of Docker Data Center platform

User profiles and credentials

All user profiles and credentials are expected to be maintained on a LDAP server. CDP provides an internal LDAP server for this purpose but it also possible to use an already available external LDAP providing the schema expected by CDP is available.

3.6 CI/CD

The CI/CD menu brings you to the core of application development and it contains the following items:

- **Code repository**, which can be GitLab or another Git-based source control versioning system if configured in the instance
- **CI/CD pipelines** where you can see your pipeline's status and buind logs
- **ChatOps**, which can be Mattermost or Slack, to interact with other developers, sysadmins and receive automated (ChatBot) event notifications

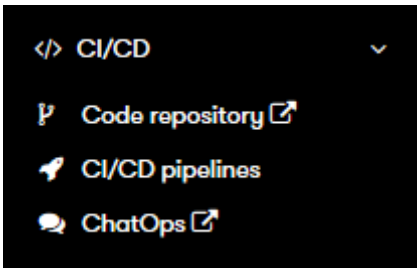

Fig. 13: CI/CD menu

For more information on GitLab, Mattermost and Slack, please read the documentation referenced at the end of this guide.

The CI/CD pipelines menu items opens a page containing a list of all the pipelines started either explicitly by triggering a webkook in GitLab, or automatically at specific configured events (i.e.: push, tag push, etc.).

The picture below shows a typical pipelines list during development:

For each started pipeline the following information are available:

- **status badge** indicating the status of the pipeline:
  - pipeline running the pipeline is still running
  - pipeline interrupted the pipeline has been interrupted
  - pipeline test failed the pipeline failed the test phase
  - pipeline passed the pipeline ended without errors
  - pipeline failed pipeline failed with an error
- **project** name including namespace, which can be the group or the user the project belongs to
- **branch** being pushed and built

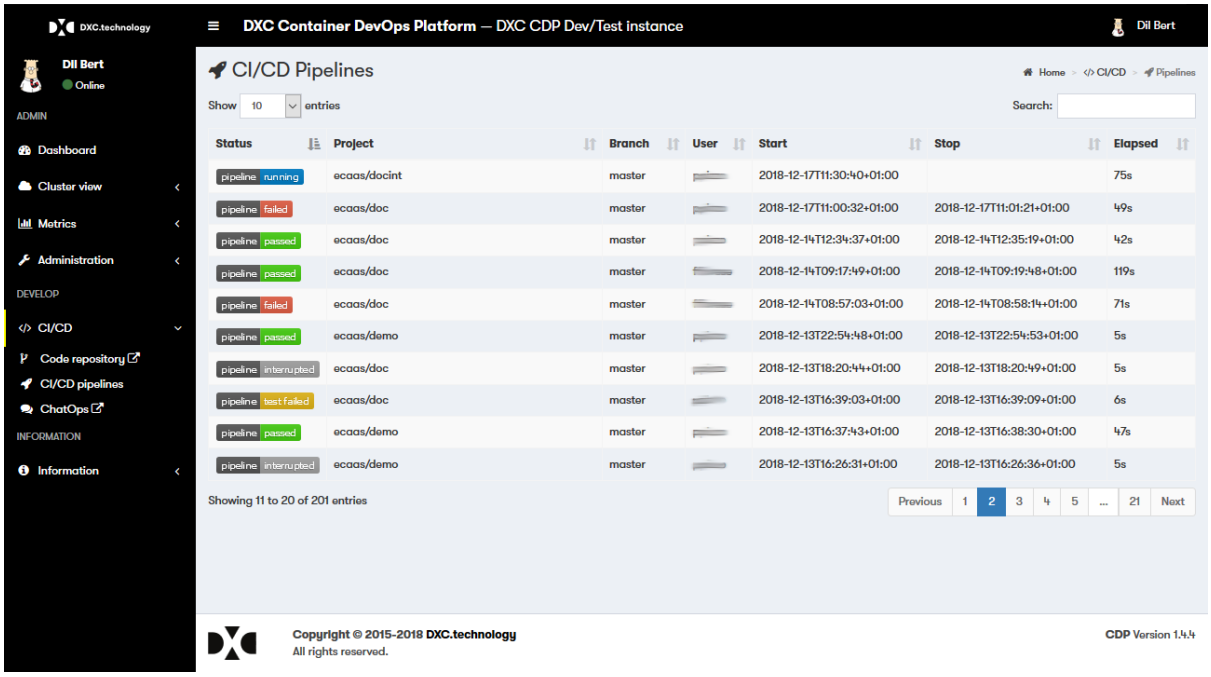

Fig. 14: CI/CD pipelines list

- **user** who triggered the pipeline
- pipeline **start** datetime
- pipeline **stop** datetime, empty if pipeline is still running
- **elapsed** time in seconds

The list can be sorted, paged and filtered.

A pipeline can be interrupted either for an error in the build pipeline itself or because of an external event (i.e.: container killed). If the pipeline ends without errors, the project will be deployed and started. A rolling update will be performed if the application is already up and running.

A click on a status badge opens a page with the details of the specific pipeline selected, where you can see all phases of the pipeline.

You can click on each phase you can see the details of the phase, as shown in the picture below, which contains an example of what you can find in the build phase, i.e. Docker image build messages, compilers output, etc.

If the pipeline stops prematurely, you only see completed phases, the last phase usually containing the error which caused the pipeline to stop.

If all went well, in few seconds, after the reverse proxies, DNS and internal rules have been automatically updated, you should see your updated application running.

3.7 Information

The information menu item contains links to the documentation page, where you can access all available documentation in various formats, including HTML and PDF, and links to the *About* page containing information regarding the DXC Container DevOps Platform, including version information, license, disclaimer and open source technologies used in the solution.

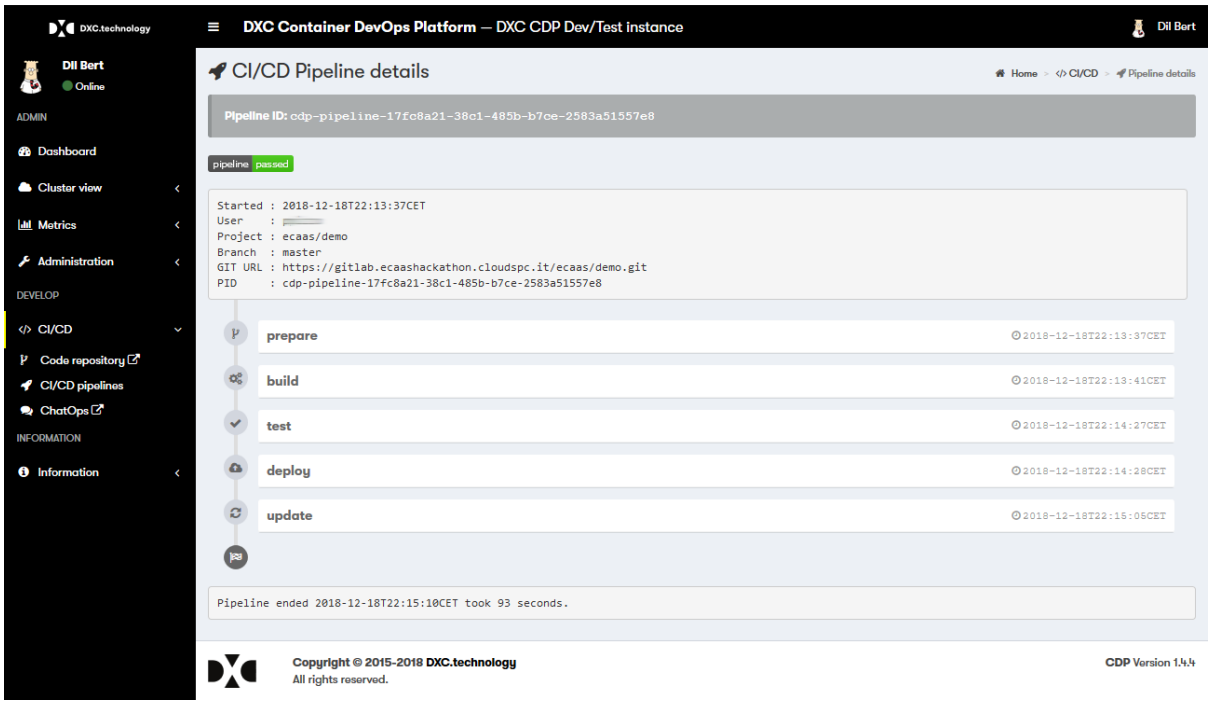

Fig. 15: CI/CD pipeline details

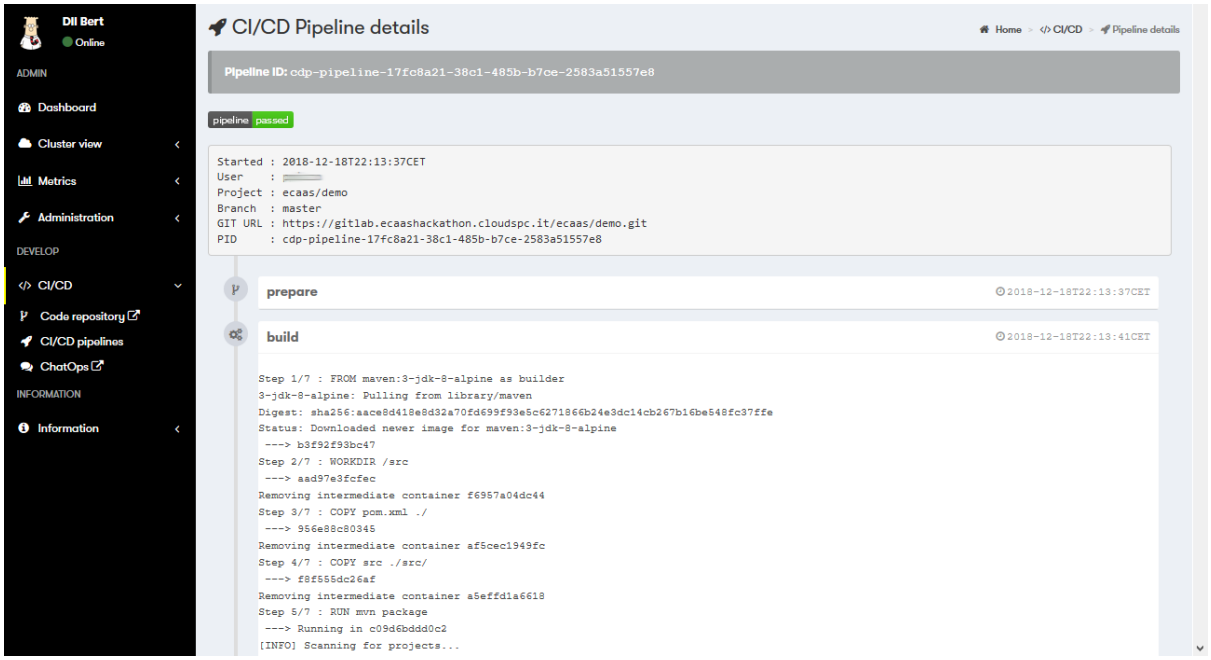

Fig. 16: CI/CD pipeline details

CHAPTER

FOUR

DEVELOPMENT OF APPLICATIONS USING CDP

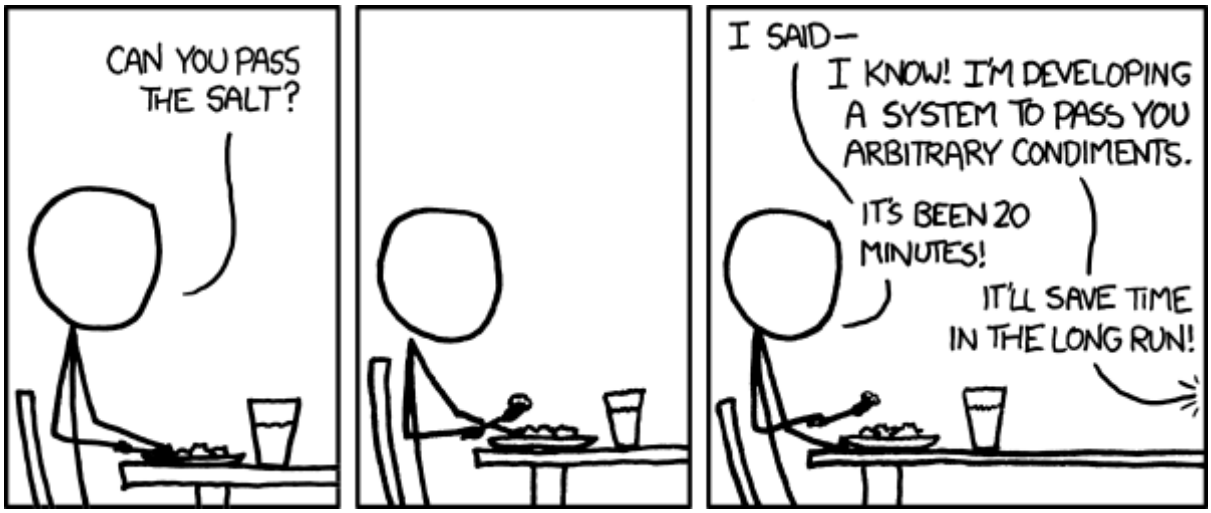

Fig. 1: © *Randall Munroe, XKCD*

In order to start developing an application or better a Stack in Docker terminology, you do not need to have specific knowledge/skills on the platform. CDP does provide an already configured and fully automated environment to allow you to "register" an application stack with a minimal configuration and to publish (automatically) it on Internet at the end of the build process.

Being CDP a non-opinionated platform you are free to use any software technology (language, runtime, middleware, etc.) which is fully compatible with Docker. You can write application components that run on Linux or Windows or both and you could use Swarm or Kubernetes orchestration or both.

It would be however beneficial to know how Docker works and in general to have a good understanding of basic DevOps and CI/CD methodologies.

CDP, like all the DevOps enabling technologies, is in continous evolution and new updates will include additional automations. it.

4.1 CDP CI/CD pipeline

CDP has its own CI/CD internal solution that is fully optimized and make use of Docker containers to executed CI/CD jobs. It uses any GitLab project to trigger the start of a new pipeline after a push to a remote repository or after other events.

Using specific GitLab triggers (like push events, tag push events, comments, etc.) the CI/CD pipeline is activated through an API call to the CDP api service after a push to the repository. The API endpoint called by GitLab in this context is this one:

```
http://api:8000/pipelines
```

A security token must be added to the API call. Please ask to your CDP system administrator wich is the right endpoint to use and the token to add. The API is called with `PUSH` HTTP method.

To trigger the CDP CI/CD pipeline on specific GitLab events, access the GitLab console, navigate to your project home page and in the sidebar in the right hand side select "Settings" –> "Integrations". In the trigger definition windows:

- enter the CDP API endpoint URL
- enter the secret token
- select all events you want to trigger the pipeline (usually *Push events* for development CI/CD, *Tag push events* for production CI/CD)
- deselect *Enable SSL verification* since the CDP API will be accessed internally on CDP overlay network behind the firewall and SSL terminator

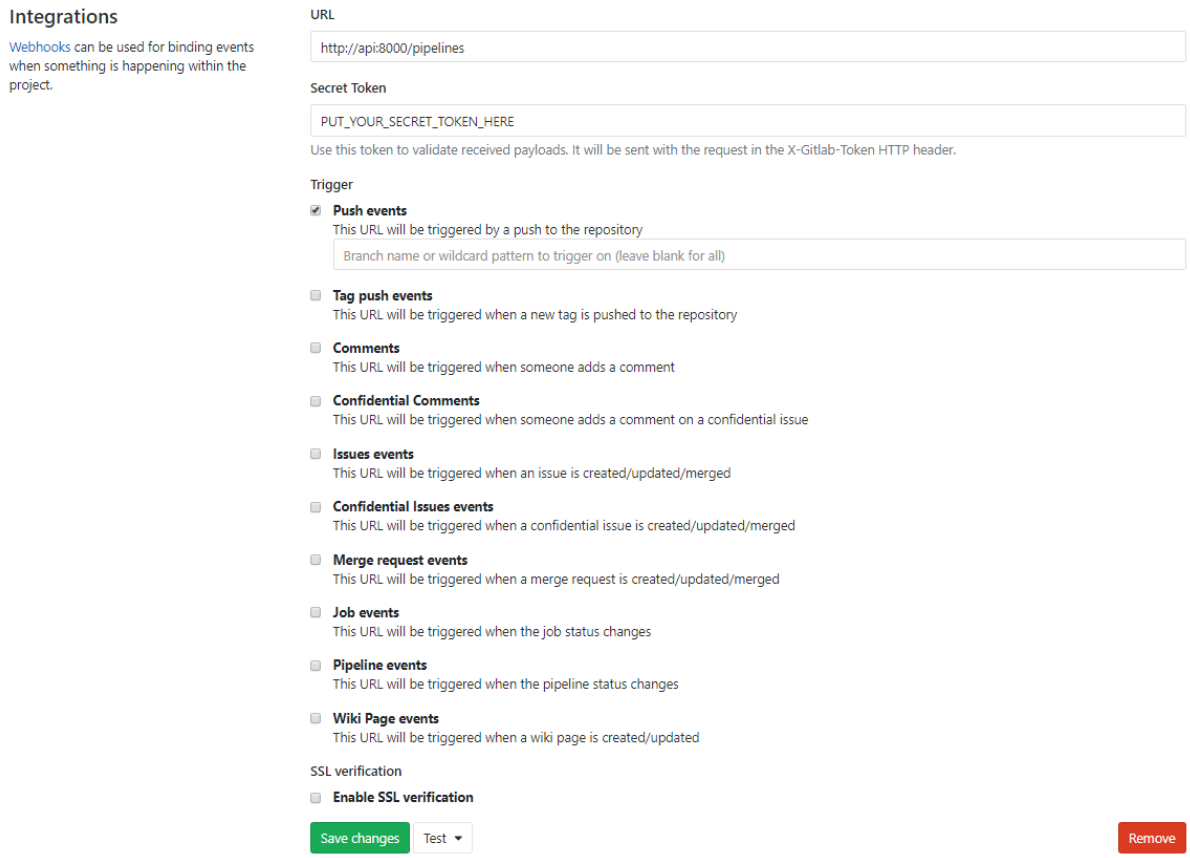

Fig. 2: GitLab integration

Under the hood, the CDP Builder reads the JSON information coming from GitLab integration trigger and invokes the CDP pipeline job with the required parameters (i.e.: project name, user which triggered the build, etc.)

The outcome of this API call is the execution of a set of scripts that performs, by default, the following actions:

1. prepare the build environment and clone the project `git` repository
2. process and generate the final Docker compose file using `ecaas_compose` pre-processor
3. using `bolero` (another CDP tool) performs all pipeline steps: build, deploy and run
4. at the end of the pipeline, a notification is sent to the chatops/collaboration service (Mattermost) in the CI/CD pipelines channel

The following diagram shows the various components involved into the CI flow:

and the diagram below shows how a typical CI/CD pipeline might look like:

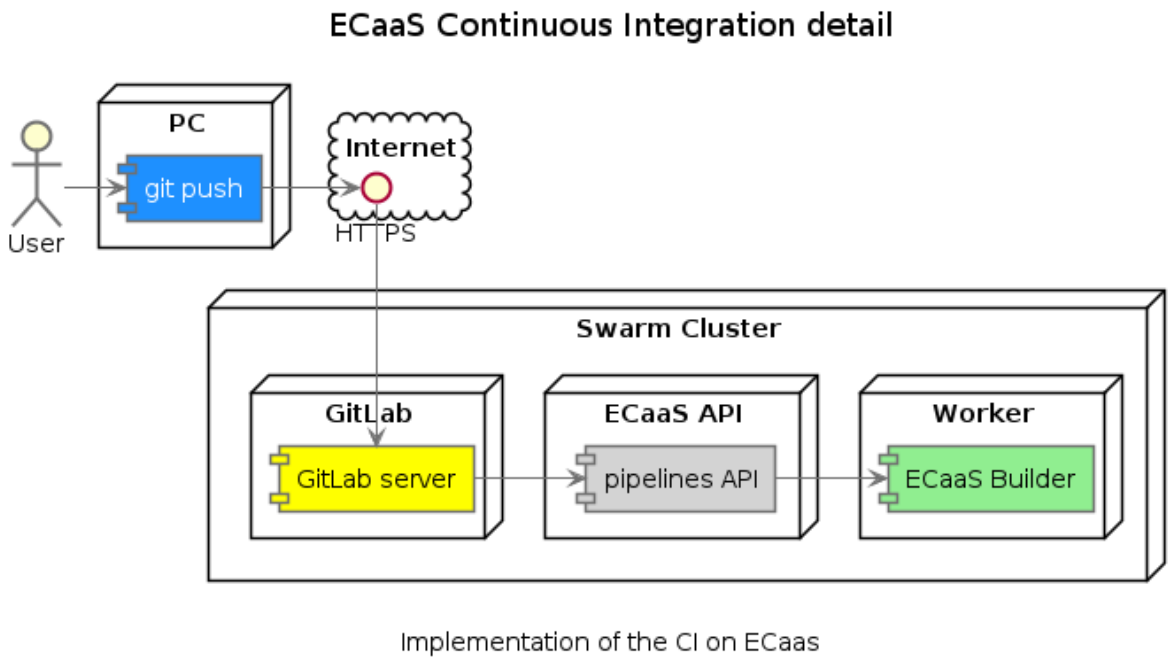

Fig. 3: The CDP CI flow

4.2 Details of the CI/CD execution stages

Following the principle to avoid vendor lock-in and to make the CI/CD process generally usable with several CI/CD engines, the execution of the pipeline is performed through 4 stages:

1. preparation
2. orchestration generation
3. build, deploy and run
4. notify

in this way there is a more granular control over the CI/CD process and error handling can be conveniently managed at the right time.

In the following sections all stages will be described in details.

4.2.1 Preparation

In this stage a suitable environment to run the CI/CD execution is created. It usually consists of the setup of some environment variables, creation of required directories and performing a clone of the project repository from GitLab.

The project repository must contains all artifacts needed to automatically build, deploy and run your application in complete autonomy. A typical project must contain at least the following:

- your application source code in any language of choice (i.e.: Java, Python, Go, C, etc.) and all required artifacts (i.e.: images, configuration files, resources, etc.)
- one or more `Dockerfile` needed to build your application services
- one `docker-compose.yml` file which contains all the information needed to build and run your application

For information about the Docker file formats, please read the references section at the end of this document.

Every CI/CD execution always starts from a scratch environment in a way to avoid any pollution across many CI/CD executions.

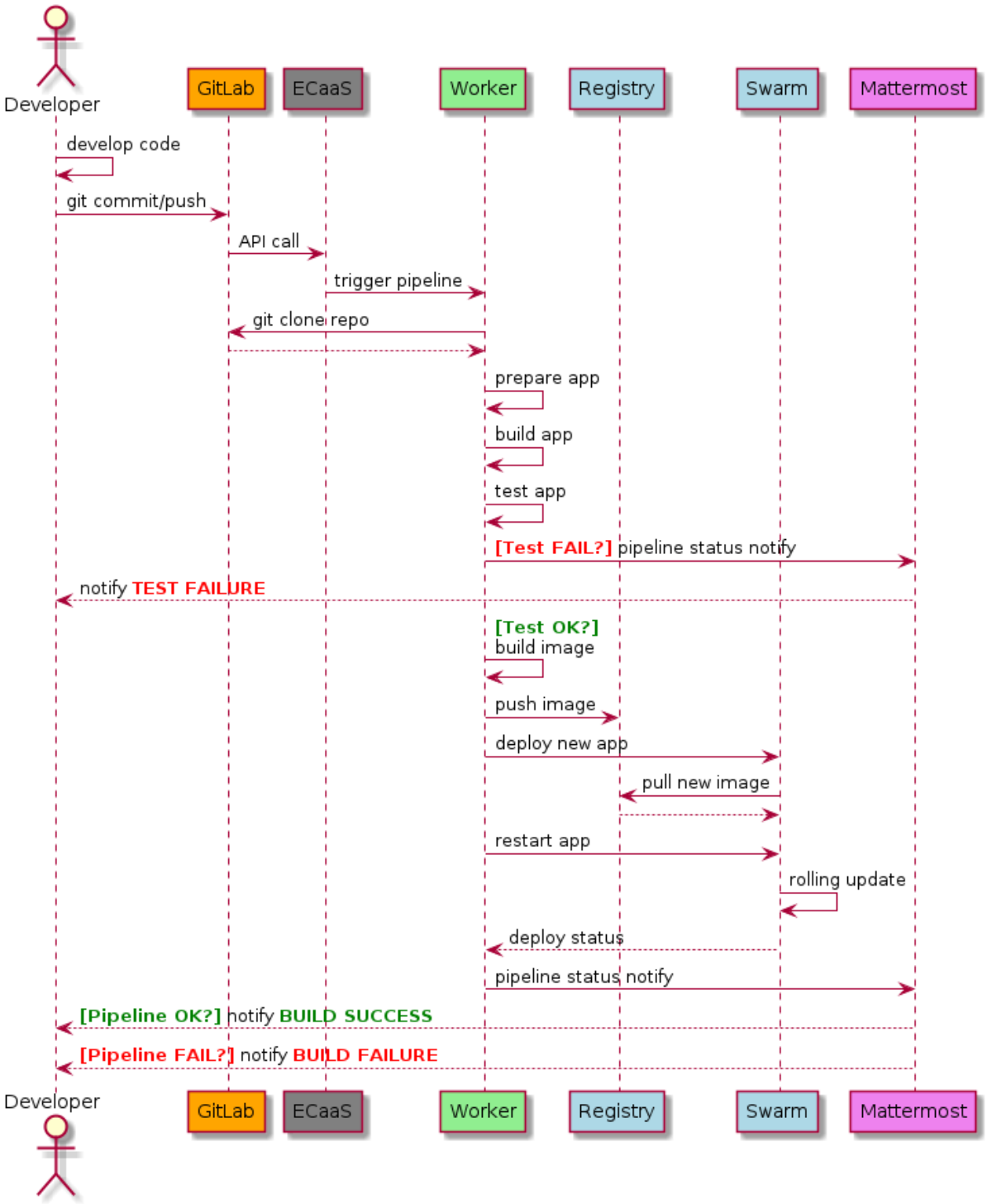
Developer
GitLab
ECaaS
Worker
Registry
Swarm
Mattermost
develop code
git commit/push
API call
trigger pipeline
git clone repo
prepare app
build app
test app
[Test FAIL?] pipeline status notify
notify TEST FAILURE
[Test OK?]
build image
push image
deploy new app
pull new image
restart app
rolling update
deploy status
pipeline status notify
[Pipeline OK?] notify BUILD SUCCESS
[Pipeline FAIL?] notify BUILD FAILURE
Developer
GitLab
ECaaS
Worker
Registry
Swarm
Mattermost

Fig. 4: image

4.2.2 Orchestration generation

This stage takes care of generating the required final orchestration file(s) starting from the one present in the application repository. The current implementation is based on the use of Docker Enterprise Edition and makes use of the Docker Compose orchestration tool. The orchestration file (`docker-compose.yml`) is processed by a tool, CDP compose, developed for the purpose of:

1. setting required Docker parameters needed by the underlying infrastructure (i.e.: metadata labels, deploy information, HA replicas, etc.)
2. creating application domain spaces (using project and user/team names)
3. creating networks, data volumes, etc. (software defined infrastructure)
4. allowing parallel deployments based on project branches
5. security policy enforcement

CDP Compose is a tool that accept in input a standard Docker compose file, processes it and produce a sanitized standard compose file by adding the required infrastructure directives not usually added by developers. In this way the developer is freed from all internal infrastructure details and can concentrate only on the development activities.

For completeness of information, this is the CDP compose usage help:

```
ECaaS-Compose
usage: ecaas_compose.py [-h] [-v] [-a API_HOSTPORT] [-s] -n NAMESPACE -p
                        PROJECT -b BRANCH -u USER -d DATASPACE -t TARGET_NET

Compose processor

optional arguments:
  -h, --help       show this help message and exit
  -v, --version    show program's version number and exit
  -a API_HOSTPORT  the ECaaS API endpoint {host:port}
  -s               request system level processing

required named arguments:
  -n NAMESPACE     the namespace
  -p PROJECT       the project
  -b BRANCH        the branch
  -u USER          a valid ECaaS username
  -d DATASPACE     the dataspace
  -t TARGET_NET    the target front-end network

process the original Docker compose to allow an easy deployment of an
application service on ECaaS
```

The following diagram show how CDP compose acts in the context of the final Docker compose file creation:

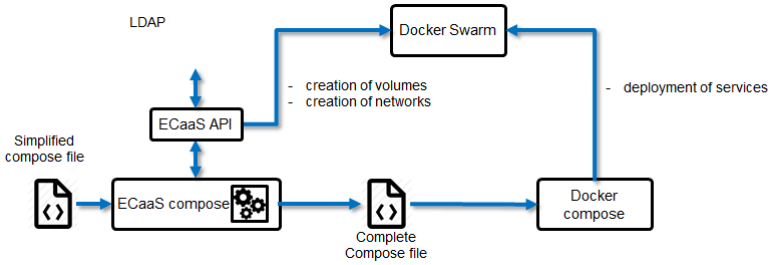

Fig. 5: CDP Compose use flow

4.2.3 Build, Deploy and Run

In this stage the application components are built using a predefined standard process that is based on a specific CDP tool: Bolero. **Bolero** is a simple execution automation engine which reads tasks definition from a simple **YAML** file and execute them with the underlying shell. Detailed information on Bolero usage and file format is in the Bolero section in this document.

In this specific stage there are 5 standard phases managed using Bolero tasks:

1. **build**: perform the build of the application components (Docker images) if required; in the default implementation it performs a `docker-compose build`
2. **test**: this is a phase useful for performance, unit and integration test; by default no task is executed in this context (can be overridden in the projecs as explained later in this document)
3. **deploy**: perform the deployment of the components onto the components repository (Docker Registry); it performs a `docker-compose push` followed by a `docker-compose pull`
4. **stop**: stop the application services; it performs no action providing virtually no downtime on application update
5. **start**: start the application service if it isn't already started, otherwise it performs a rolling update; it performs a `docker stack deploy -c docker-compose.yml`

For each of those tasks there is a corresponding default Bolero YAML file.

It is possible to alter the default behaviour by providing a set of alternative Bolero YAML file for each specific phase. To do so it is sufficient to create an `.ecaas` subdirectory into the root of the project repository and within that `.ecaas` directory place any of the build, test, deploy, stop or start YAML files.

The pipeline behaviour is therefore the following (in pseudocode):

```
for each STAGE in { build, test, deploy, stop, start}:
   if exists .ecaas/STAGE.yaml then
      bolero .ecaas/STAGE.yaml
   else
      bolero /lib/STAGE.yaml
```

The best way to modify a specific stage is to copy the corresponding default CDP Bolero library file under your `.ecaas` directory and modify it to suit your needs.

For your convenience, default Bolero files are shown below:

build.yaml:

```
- task:
    name: build
    run:
      exec: docker-compose build
```

test.yaml:

```
- task:
    name: test
    run:
      exec: echo "No test specified"
```

deploy.yaml:

```
- task:
    name: push
    run:
      exec: docker-compose push
    goto:
      - "ret!=0": end
```

(continues on next page)

(continued from previous page)

```
- task:
    name: pull
    run:
      exec: cd $ENV_DIR && . ./env.sh && cd $OLDPWD && docker-compose pull
    goto:
      - "ret!=0": end

- task:
    name: end
```

stop.yaml:

```
- task:
    name: stop
    run:
      exec: echo "Performing a rolling update"
```

start.yaml:

```
- task:
    name: run
    run:
      exec: cd $ENV_DIR && . ./env.sh && cd $OLDPWD && docker stack deploy -c␣
↪docker-compose.yml $STACK
```

Example:

You have a Java project that uses Maven for the build process but you don't want to install Maven on the target image. Albeit you can use a multi-stage Docker build for this goal, for the sake of argument in this example we will personalise the build phase using a ephemeral container to build the WAR file which will then be added to the target image.

In this situation the `build.yaml` file, created in the project's `.ecaas` directory, can be something like the follwing one:

```
- task:
    name: build package
    run:
      exec: >-
        docker run -ti --rm
        -v /etc/localtime:/etc/localtime:ro
        -v ${project_dir}/src:/src
        -v ${data_basedir}/master/maven/.m2:/root/.m2
        -w /src maven:3-jdk-7-alpine
        mvn package -o
    goto:
      - "ret!=0": error

- task:
    name: copy artifacts
    run:
      exec: "[ -f src/target/application.war ] && cp -f src/target/application.war␣
↪appserver/deploy/"
    goto:
      - "ret!=0": error

- task:
    name: build image
    run:
      exec: docker-compose build
    goto:
```

(continues on next page)

(continued from previous page)

```
      - "ret==0": end

- task:
    name: error
    run:
      exec: export ERROR=1

- task:
    name: end

- task:
    name: clean
    run:
      exec: >-
        docker run -ti --rm
        -v ${project_dir}/src:/src
        busybox rm -rf /src/target

- task:
    run:
      exec: exit $ERROR
```

4.2.4 Notify

At the end of the pipeline, the user is notified through the CDP Mattermost ChatOps service in a specific channel, as shown below:

The notification channel used by the pipeline depends on your local CDP instance configuration but it is usually a public channel which can be joined by every people in the team.

The notification message convey some information about the build, including the build status badge, the project name, the branch and the user who triggered the build. To see the build log just click on the status badge icon.

This channel tends to be very chatty, so to avoid being continously notified for pipelines not triggered by you, it is advisable to mute this channel: in this way you will not be notified when someone else triggers a pipeline but you will still receive notification for your pipelines. This can happen because CDP puts a mention in the message with the username which triggered the build and Mattermost mute setting is bypassed when a message there is a mention for you.

4.3 How to use the CDP CI/CD pipeline with an application project

Assuming that you have an application project repository you just need to make sure to have at least:

1. one or more Dockerfiles required by the Docker services referenced in the project or just refer to existing Docker images available from Docker registries
2. a proper Docker compose file where you define relationship across the services of the application to be deployed, the networks to be used and how are related to the application services as well as the storage volumes, the application internal ports and the name of the exposed applicaiton service from a domain name point of view
3. if needed you can create a specialized set of Bolero's tasks used to manage the 5 phases of the "Build, Deploy and Run" stage under the `.ecaas` directory mentioned previously

For most situations you can write a very simple compose file and let CDP manage automatically all the provisioning of networks, volumes, etc.

Let us assume to use the CI/CD with a simple project, like a Wordpress CMS. The project uses two standard application components, the wordpress engine and the database deployed as a Docker Stack on the CDP.

The logical diagram of the application service is shown here:

The related, simplified, `docker-compose.yml` file is the following one:

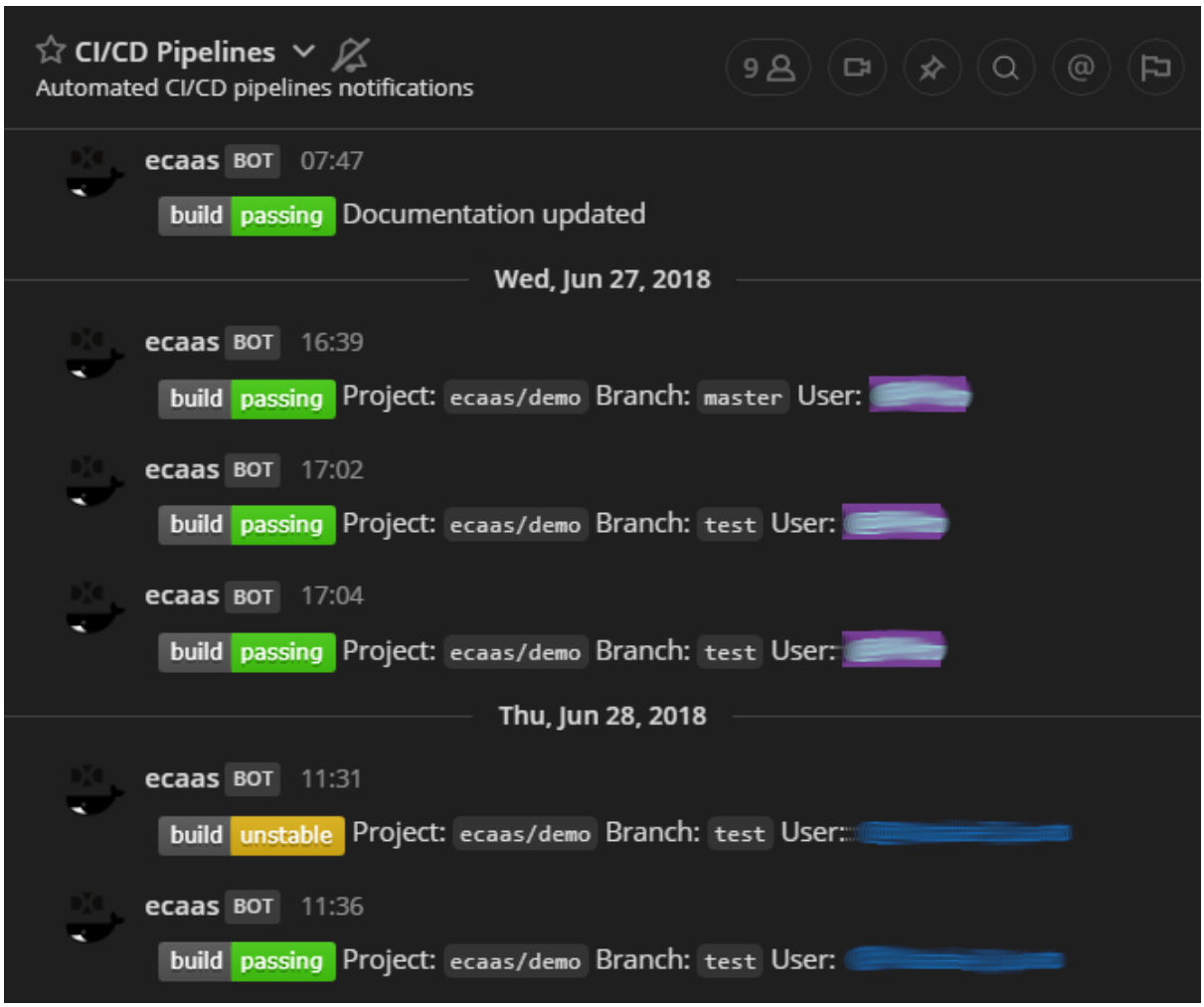

Fig. 6: Mattermost CI/CD

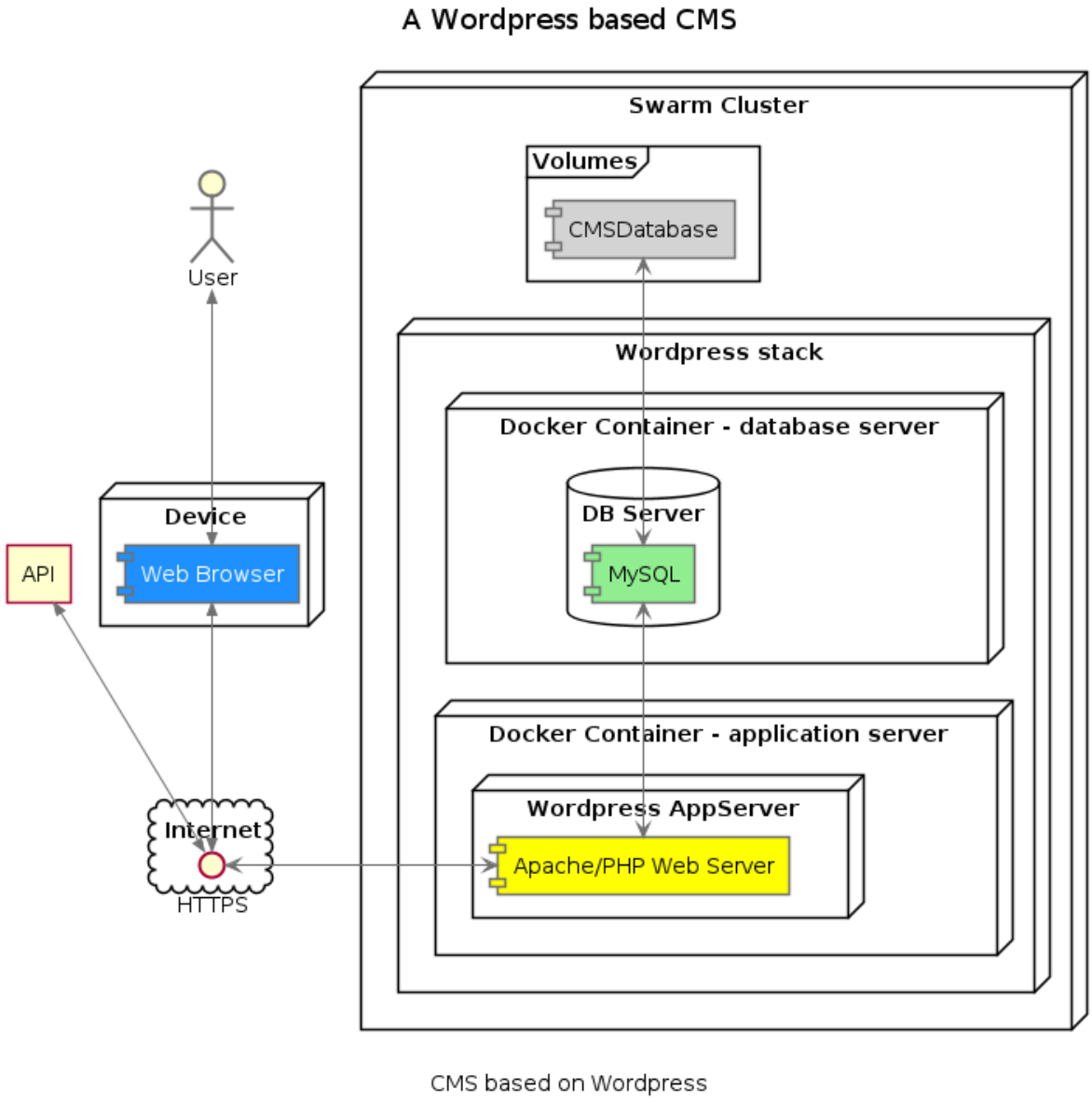

Fig. 7: Wordpress

```
version: '3.2'

services:
  wpdb:

    image: mysql:latest
    networks:
      - default
    volumes:
      - db_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password

  mywordpress:
    depends_on:
      - wpdb
    image: wordpress:latest
    networks:
      - default
    environment:
      WORDPRESS_DB_HOST: wpdb:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
    labels:
      - "com.dxc.ecaas.friendlyname=My Wordpress"
    secrets:
      - db_password
    deploy:
      labels:
        - "ecaas.external.port=80"
        - "ecaas.external.host=mywordpress"

secrets:
   db_password:
     file: ./db_password.txt

volumes:
  db_data:
```

Altough you can be familiar with Docker compose syntax let us go through the compose file in order to understand the various sections and see how with CDP certain features can be automatically managed by the platform.

The volumes

For the volumes you can leave to CDP to take care of the creation, you just need to define the used volumes in the specific section:

```
volumes:
  db_data:
```

In this case there is just one volume used to store the database instance data.

CDP will create for you all the volumes needed by the application stack but you can in any case override this by specifying explicitly volume characteristics.

The name of the application from a DNS point of view

CDP makes use of `labels` to define the name of the application service in the `deploy` section of the service exposed to the public network:

```
deploy:
  labels:
    - "ecaas.external.port=80"
    - "ecaas.external.host=mywordpress"
```

In particular the `ecaas.external.host` label it is used for this purpose, so if your CDP instance is reachable at this URL `https://console.ecaassite.cloudspc.it` when this wordpress application stack is deployed it can be accessed with this URL: `https://mywordpress.ecaassite.cloudspc.it`. Of course you are free to link this stack to a specific registered domain name.

The `ecaas.external.port=80` it is used to define the port on which the application stack is reachable with the CDP local Docker network. The applicaiton is reachable on the public network using the HTTPS protocoo thanks to the routing mesh functionality of the Docker Swarm cluster (by performing load balancing and application level protocolo routing).

You can also use branches: if you create and push your changes on a different branch, CDP automatically allocates a new environment for build and deploy. If you're using a branch, the basename of the web URL will be APPNAME-BRANCHNAME.

For example, if you push your changes to the `devel` branch, a new Wordpress application stack is started and at the end of the pipeline it will be published at the following URL: `https://mywordpress-devel.ecaassite.cloudspc.it`.

If you do not set the `ecaas.external.host` label, a dynamic URL is generated in the following way:

```
https://SERVICE-BRANCH-PROJECTNAME-NAMESPACE.ecaassite.cloudspc.it
```

where `NAMESPACE` is the GitLab group your project is created under, or your username if it is a personal project.

Overriding the default case

Although you will probably be fine for most of the cases to use the simplified compose definitions there can be situation where you could need to use specific compose definitions. You can do that without any problem. In this case CDP will pass your definitions as they are written in the compose file apart for few exceptions like for example if you will try to map any host file-system path as the Docker a volume to the Docker `/var/run/docker.sock` or the host `root`. In order to do this the user that run the stack must have high privilege (system user).

Output of the processing of the docker-compose file

CDP compose processes the docker compose file and complements it with additional information needed to build, deploy and run the application on the CDP platform. Just for example purpose here is the output of this process applied to the docker compose files described previously:

```
networks:
  agid_frontend:
    external:
      name: agid_frontend
  wpress_backend:
    external:
      name: wpress_backend
secrets:
  db_password:
    file: ./db_password.txt
services:
  mywordpress:
    depends_on:
    - wpdb
    deploy:
      labels:
      - ecaas.external.port=80
      - ecaas.external.host=wpdemo
      - traefik.port=80
```

(continues on next page)

(continued from previous page)

```
      - traefik.docker.network=agid_frontend
      - traefik.frontend.entryPoints=https
      - traefik.backend.loadbalancer.sticky=true
      - traefik.frontend.rule=Host:wpdemo.ecaasagid.cloudspc.it
      placement:
        constraints:
        - node.labels.engine==worker
    environment:
      WORDPRESS_DB_HOST: wpdb:3306
      WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
      WORDPRESS_DB_USER: wordpress
    image: wordpress:latest
    labels:
    - com.dxc.ecaas.friendlyname=My Wordpress
    networks:
    - default
    - wpress_backend
    - agid_frontend
    secrets:
    - db_password
  wpdb:
    deploy:
      placement:
        constraints:
        - node.labels.engine==worker
    environment:
      MYSQL_DATABASE: wordpress
      MYSQL_PASSWORD_FILE: /run/secrets/db_password
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_password
      MYSQL_USER: wordpress
    image: mysql:latest
    networks:
    - default
    secrets:
    - db_password
    volumes:
    - ecaas-user_wpdemo_master_db_data:/var/lib/mysql
version: '3.2'
volumes:
  ecaas-user_wpdemo_master_db_data:
    external: true
```

You should have now all the information needet to start developing and deploying your project using CDP CI/CD pipeline. For information not covered in this document, please ask to your CDP instance administrator.

Happy development!

CHAPTER

FIVE

ADVANCED DEVELOPMENT OF APPLICATIONS USING THE ECAAS PLATFORM

This section contains advanced development topics not covered in the previous section.

5.1 ECaaS shell

TO BE DOCUMENTED

5.2 Bolero execution engine

TO BE FINISHED

5.2.1 Introduction

Bolero is a simple execution automation engine developed with simplicity in mind. It reads tasks definition from a simple **YAML** file and execute them with the underlying shell.

By design it does not include any module to perform the tasks, for example an ssh module, a docker module, etc. This because every added module is a potential source of bugs and it must be maintained and updated to the latest

protocols version. Therefore, Bolero uses native shell commands and tools installed on the system to execute tasks.

5.2.2 Main concepts

An execution definition is composed by an ordered list of **tasks** stored in a YAML file.

Each **task** is composed by a single **run** statement which contains the shell command to run. The shell command can be as complex as you want and it can also be composed by a sequence of commands separated by shell metacharacters `;`, `&&`, `||`, etc.

A task can be labeled with a **name** which can be used to reference the task from other tasks. Names cannot be duplicated among tasks.

A task can contain a list of **goto** statements which can move the execution flow to another task. A **goto** statement is a tuple composed by a valid Python expression which must evaluate to a `Boolean` type and the **name** of the task to move the execution if the corresponding expression evaluates to `True`. Expressions are evaluated in the order they are stored in the YAML file and the evaluation stops after the first `True` statement found. If no expression evaluates to `True` the execution flow is moved to the next task in the list. The goto condition can be based on the exit status of the shell command executed in the run statement, which is available in a variable named `ret`. Please see the *YAML syntax* section for more details.

Environment variable are available in all tasks but each task is executed in a dedicated environment: this means that variables set in a task cannot be referenced by other tasks.

The execution stops after reaching the last task in the list. Please note that using goto statements you can end up in an infinite loop.

5.2.3 Install and run

To run the engine you need Python 3 and a few requirements, all listed in `requirements.txt` file. You can install them using the method of your choice, but we recommend using a `virtualenv` environment as follows:

```
virtualenv -p python3 env
. ./env/bin/activate
pip install -r requirements.txt
./bolero.py example.yaml
```

There's also an install script you can use to create the virtualenv in a standard location and copy the command line wrapper in the standard PATH. If you use the installer, you can then use the simple `bolero` command line tool to use the tool.

Dockerfile and docker-compose.yml have also been provided if you want to run bolero inside a container.

From now on, we assume you're using the installed version.

To see all command line options please run:

```
bolero --help (or bolero -h)
```

5.2.4 YAML syntax

The syntax of YAML task definition file is very simple, you can find an example in the `example.yaml` file included in this directory. The YAML file can contain `&` labels and `*` references, as per YAML syntax.

The best way to describe the syntax is by examples.

A simple execution which prints an "hello world" message is showed below:

```
- task:
    run:
       exec: echo "Hello Bolero World!"
```

A complete execution containing all **Bolero** features:

```
# Simple example workflow

# testing the cond + CLI argument retrieval
- task:
    name: 1st task
    cond: args[0] == "hello"
    run:
      # testing the embedding of python inline code using microtemplating
      exec: echo "1st arg is {%emit(args[0])%}"

- task:
    name: 2nd task
    run:
      exec: >-
        if  [ "{%emit(args[0])%}" == "test" ];then echo "1st arg is [test]"; fi

# Since the workflow starts at the first task in the file, we jump to the main␣
↪task avoiding to execute library tasks
- task:
    goto:
      - "True": start

# Subroutine task: the task can be referenced later using '*'
- task: &printsuccess
    run:
      exec: echo "SUCCESS"

# First task: the real work begins here (a simple placeholder task to mark the␣
↪starting point)
- task:
    name: start
    # testing conditional execution by querying an environment variable
    cond: env["TZ"] == "Europe/Rome"
    run:
      exec: echo "TZ is Rome!!"

- task:
    name:
    run:
      # testing the microtemplating by executing a python snippet of code that␣
↪includes an 'import' of a python package
      exec: echo "Current time is {%import time;emit(time.strftime('%H:%M:%S'))%}"

# Loop example: demonstration of environment persistence and how to implement a␣
↪simple loop using shell commands in different tasks
- task:
    run:
      exec: export i=1

- task:
    name: loop
    run:
      exec: echo "Iteration $i"

- task:
    run:
      exec: export i=$(($i+1))

- task:
    run:
      exec: test $i -gt 3
    goto:
```

(continues on next page)

(continued from previous page)

```
      - "ret!=0": loop

# Test timeout: it should exit after 2 seconds instead of 5 seconds
- task:
    run:
      exec: sleep 5
      timeout: 2
    goto:
      - "ret==0": ls
      - "timeout==True": timeout

# Timeout notification
- task:
    name: timeout
    run:
      exec: echo "The previous task timed out"

# Performs a simple listing of a notexistent directory in order to test failure␣
↪condition
- task:
    name: ls
    run:
      exec: ls /notexistent
    goto:
      - "ret==0": success
      - "ret!=0": end

# This should never be executed
- task:
    run:
      exec: echo "You should not see this"

# Placeholder for success goto
- task:
    name: success

- task: *printsuccess

# Last task of the workflow: execution stops after this task (again, a simple␣
↪placeholder)
- task:
    name: end
```

CHAPTER

SIX

WORKING WITH GIT

Fig. 1: © *Randall Munroe, XKCD*

Git is a wonderful tool with a lot of features, but for day-by-day developments only few commands are used. In this section we will describe the most frequently used Git commands and GitLab features.

6.1 Git configuration

The global Git configuration is stored in the `${HOME}/.gitconfig` file, but you can also set per-repository (local) configuration. Please note that local settings take precedence over global ones.

Global configuration can be managed using `--global` command line option, to access the local configuration instead, please use `--local` option issuing the `git` command inside an already cloned repository directory.

6.1.1 User's email

```
git config --global user.email user.email@dxc.com
```

if you want to set the email on a per-repository basis, use `--local` instead of `--global` in the repository diectory after it has been cloned.

Using `--global` or `--local` is up to you: you are free to adapt the commands below based on your preferences.

6.1.2 Credential cache

To avoid entering your username/password each time you interact with a remote, you can set up the credential helper to cache your credential for the specified amount of time in seconds:

```
git config --global credential.helper "cache --timeout=86400"
```

The above example caches the credentials for 24 hours.

6.1.3 Fancy prompt

It is advisable to set up the following prompt in your `.bashrc` in order to see the current branch you're working on:

```
PS1='${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;
↪34m\]\w\[\033[00m\]\[\033[01;33m\]$(__git_ps1)\[\033[00m\]\$ '
```

The above prompt is suitable for a Debian based OS (i.e.: Debian, Ubuntu, etc.), you might have to adapt it if you're using other operating systems.

6.2 Remote repositories

6.2.1 Clone a repository

```
git clone https://gitlab.ecaashackathon.cloudspc.it/ecaas/demo.git
```

After cloning the repository you can enter the newly created directory and start working.

6.2.2 Rename a repository

Sometimes it's necessary to rename a repository on the remote server (i.e.: GitLab, GitHub, ecc.). When this happens, you have to modify the local repository URL to point to the new URL.

You can either modify the configuration file `.git/config` by hand, or (better) issue the following command on a shell console in the repository's directory:

```
git remote set-url origin NEWURL
```

Example:

```
git remote set-url origin https://gitlab.ecaashackathon.cloudspc.it/ecaas/efaas.git
```

In the above command we assume your remote is named `origin`: if this is not the case, change the name accordingly.

6.2.3 Working with multiple remotes

Sometimes it could be useful to work with two or more remotes, for development reasons or just to have a backup of your project on a secondary Git server. Fortunately, Git offers you a simple way to deal with multiple repositories without the need to duplicate your project.

You just have to add a new remote to your cloned repository with the following command:

```
git remote add REMOTENAME REMOTEURL
```

where `REMOTENAME` is an arbitrary name of your choice which identifies your new remote. After the remote has been added you can perform all standard Git operation by using `REMOTENAME` instead of `origin`:

```
git push REMOTENAME
```

For example, if you want to clone a repo from a remote and push to another remote, you can issue the following commands:

```
git clone https://gitlab.ecaashackathon.cloudspc.it/ecaas/demo.git
cd demo
git remote add dxc git@github.dxc.com:cdp/demo.git
git push dxc
```

To see the list of configured remotes, issue the following comomand:

```
git remotes -v
```

6.3 Branches

6.3.1 Creating branches

```
git branch newbranch
git checkout newbranch
```

or

```
git checkout -b newbranch
```

6.3.2 Setting upstream

A newly created branch is local to your repository unless you issue the following command:

```
git branch --set-upstream-to=origin/newbranch newbranch
```

which sets the upstream needed to push the branch to the remote repository.

6.3.3 Switching branches

To switch on devel branch and then on master again:

```
git checkout devel
git checkout master
```

6.3.4 Delete branch

To delete a branch you must first exit the branch and then delete it:

```
git checkout master
git branch -d oldbranch
```

6.3.5 Get branch information from all remotes

```
git fetch --all
git branch -r
```

6.3.6 Switch to a new remote branch

```
git checkout --track origin/devel
```

or

```
git checkout -b devel origin/devel
git branch -u origin/devel
```

6.4 Tags

A *tag* is an information used to identify a specific point in the history of the project, i.e. a snapshot. It is usually used to identify production releases or other events which are important for the project.

There are two types of tags:

- *lightweight* tags: just a pointer to a specific commit without any other information
- *annotated* tags: can contain a message, the tagger name, email, date, they are checksummed and can be signed and verified

Production release tags should be annotated.

6.4.1 List tags

```
git pull
git tag
```

6.4.2 Add a tag

Lightweight:

```
git tag TAGNAME
git push origin TAGNAME
```

Annotated:

```
git tag -a -m "New production release" TAGNAME
git push origin TAGNAME
```

you can also issue `git push --tags` to push all tags at once.

6.4.3 Remove a tag

```
git tag -d TAGNAME
git push origin :TAGNAME
```

6.4.4 Move a tag to another commit

```
git tag -d TAGNAME
git push origin :TAGNAME
git tag TAGNAME
git push origin TAGNAME
```

or (annotated tag):

```
git tag -a TAGNAME COMMITID -f
git push origin TAGNAME -f
```

after the above *force* (`-f`) operations, every developer must do the following:

```
git tag -d TAGNAME
git fetch origin --tags
```

6.4.5 Checkout a tag

```
git checkout TAGNAME
```

WARNING: do not work on a checked out tag! If you plan to make changes, please open a branch otherwise your changes will be unreachable.

6.5 GitLab

6.5.1 Change GitLab email

It could happen that you might want to change your primary email in GitLab but you noticed that there's no way to perform the change in GitLab. Don't worry, there's always a way!

Two things you must know before you start:

- Your primary email is managed by LDAP and it's retrieved by GitLab during the login
- If you made some commit with your old email, you might want to maintain your old email as secondary rather than to replace with the new one

Said that, you can follow the steps below to change your email:

1. Login into GitLab
2. Go to `Profile --> Edit Profile --> Emails` and add your new email
3. Wait for the confirmation email and follow the istructions to confirm your new email
4. Go to the `ECaaS console --> Profile` and change your profile email
5. Logout from GitLab and login again

And you're done! To check if the change has been applied, go into the `Emails` section referenced in step 2 above and you should see your newly added email as "Primary email" and, if necessary, change your notification and public email preferences.

NOTE: your profile email is managed by ECaaS and therefore if you change the email in your ECaaS profile without first adding the new email in your GitLab profile, when you login in GitLab you will get a new username because your email does not match the one stored in your local GitLab account.

Based on your needs, you can keep the old email address in your GitLab profile to identify your old commits or decide to remove it. It's up to you, but unless your old email could be assigned to another person, we recommend you to keep your old email(s) so your old commits will still be associated with your user.

Don't forget to change the email on already pulled GIT repositories! Just issue the command below under the main directory of all cloned repositories:

```
git config --local user.email newemail@example.com
```

or

```
git config --global user.email newemail@example.com
```

based on where you want to manage your email, at repository (local) level or at global level.

6.6 References

- https://git-scm.com/book/en/v2

CHAPTER

SEVEN

DOCKER BEST PRACTICES AND GUIDELINES

In this document we will describe the general guidelines and best practices to be used when deploying a Docker-based architecture and when moving applications to Docker containers.

7.1 Docker environment

A minimal Docker environment is composed by a Docker engine running on a physical or virtual host: the engine is the only mandatory component needed to run applications inside containers. However, this setup is not suitable for an enterprise production environment composed by many Docker engine hosts.

The minimal recommended setup for a production environment is composed at least by the following components:

- **Docker Engine**: the Engine should be installed on at least three hosts in order to guarantee a high availability service. Merely installing more than one engine is not sufficient to have a HA environment: the engines must be controlled by a clustering engine called "Docker Swarm".
- **Docker Swarm**: the native clustering solution for Docker. It turns a pool of Docker hosts into a single, virtual Docker host which can be used for high availability or for load scaling.
- **Docker Kubernetes**: the new industry standard clustering solution based on the Kubernetes open source scheduler/orchestrator for Containers.
- **Docker Compose**: a fundamental tool for defining and running multi-container applications in a micro-service architecture. Even if your application is composed by only one container, it is worth to use Docker Compose anyway because it is the best place to put "docker run" command line options and because your application will be "ready" to be extended with other services or to be included in other multi-container applications.
- **Universal Control Plane**: the administration console for a Swarm-controlled Docker cluster. With the Universal Control Plane (UCP) you can asses the health and resource usage of the cluster, monitor containers, define networks, etc. It gives a high level view of the entire Docker cluster.
- **Docker Trusted Registry**: a secure repository which allows users to store and secure their images on-premises or within their virtual private cloud.

7.1.1 Networking

When starting applications using Docker Compose, if you do not declare a network, Docker-compose will define a temporary network for you, but on production setup you might want to define your own networks.

As a rule of thumb, you should define at least an overlay network, that is a network which spans to the entire cluster. When a new container joins an overlay network, the internal Docker DNS is updated with the internal IP of the new container, and all containers in the same network, even running on a different engine, can reach the new container by name. In this way you don't have to care about the changing IP addresses: just use the services using their symbolic name and you're done.

If you need to implement isolation security rules at networking level, you can define as many networks as your application design requires: each container will then be attached only to the networks it needs to perform its job, as showed in the example below.

Overlay Network example

Let us first create the front-end tier network, we will call it `web-net`:

```
$ docker network create web-net
```

Docker will reply with this network id:

```
e2d66481217817caa7a1f3dcac5401fba82cd0cd0ef701b1039e95fbc1185c8a
```

Now let us create the application tier network, we will call it `app-net`:

```
$ docker network create app-net
```

Docker will give back to us this network id:

```
46e367055757ef5961a4271e28237a7f563a073398e0d2072e7fb288fd07fe8d
```

Now if we list the available networks with this command:

```
$ docker network ls
NETWORK ID          NAME                DRIVER
7c0387a493e6        bridge              bridge
d09a5980b3ca        none                null
a0248fe8612f        host                host
e2d664812178        web-net             bridge
46e367055757        app-net             bridge
```

As you can clearly see we can find our two created networks. Now let us start two Containers, for simplicity we are going to use a simple Ubuntu image and start an interactive shell. Let us start with the first one. We will give to it the `web_server` name and we position it into the `web-net` network:

```
$ docker run -it --rm --net=web-net --name web_server ubuntu /bin/bash
```

The second Container will have the name app_server and will be positioned into the `app-net` network:

```
$ docker run -it --rm --net=app-net --name app_server ubuntu /bin/bash
```

Now we have two distinct Containers, one, named `web_server` running within the `web-net` network and the other, named, `app_server` running within the `app-net`. Let us now connect the `web_server` Container to the `app-net` network with this command:

```
$ docker network connect app-net web_server
```

If we try now to contact the app_server from the web_server Container, for example using a simple ping command:

```
$ ping app_server.app-net
```

As you can see the `app_server` host name must use a canonical format with the network postfix. The expected result is:

```
PING app_server.app-net (172.19.0.2) 56(84) bytes of data.
64 bytes from app_server (172.19.0.2): icmp_seq=1 ttl=64 time=0.195 ms
64 bytes from app_server (172.19.0.2): icmp_seq=2 ttl=64 time=0.103 ms
64 bytes from app_server (172.19.0.2): icmp_seq=3 ttl=64 time=0.112 ms
64 bytes from app_server (172.19.0.2): icmp_seq=4 ttl=64 time=0.110 ms
64 bytes from app_server (172.19.0.2): icmp_seq=5 ttl=64 time=0.113 ms
64 bytes from app_server (172.19.0.2): icmp_seq=6 ttl=64 time=0.108 ms
```

If you want to create a network which spreads across all Docker engine hosts in the cluster, you must use the `overlay` network driver when you create the network:

```
$ docker network create --driver overlay cross-net
```

The newly created `cross-net` network will be immediately available to all Docker hosts in the cluster with a consistent address space. All containers connected to that network can talk each other even if running on different hosts.

So this is very short summary about the basic functionality of the SDN capability now available in Docker 1.9.

7.1.2 Shared storage

By design, Docker images are immutable: if you write something on the file system inside a running container, if you don't commit the change (creating another image) your data are lost when the container is removed.

In order to persist the data, you must export the relevant portions of the file system on external volumes. All volumes resides on the host where the engine is run and therefore if you want to move a container on another engine you must copy the data or using an external shared storage which can be attached to the container using a specific plugin.

The list of currently available plugins is available at the following URL: https://docs.docker.com/engine/extend/plugins/.

In alternative, you can mount an external distributed file system on every host and use the shared space to store containers data and configuration.

For example, you can mount a GlusterFS volume on each host at operating system level and use the shared volume to store containers configuration (i.e. docker-compose.yml file, environment vars, etc.) and application data. Then you can run the containers using traditional volume mount command line option (`-v host_dir:container_dir`) to mount the shared Gluster file system inside the container.

7.2 Microservices architecture

When deploying a complex multi-tier business platform in a container architecture, you should avoid to put everything in a single container. You must identify the basic software components (i.e.: database, front-end web server, web bricks, etc.) which cannot be decomposed anymore and put every atomic application in its own container.

All containers should be defined in a `docker-compose.yml` file and activated using Docker Compose.

In a multi-layer application only the user-facing layer must be exposed to the host using Docker port mapping. For example, in a typical three-tier application composed by a database, an application server and a web server, you must expose to the host only the web server. The web server will contact the application server using the internal Docker network and the same will do the application server with the database.

7.3 Development life cycle

From the users' point of view, an application server like WebLogic, JBoss or Tomcat running in a container is not different from the same application server running on a physical or virtual host. The users still access the software using APIs or a web interface as they're used to do in a traditional installation, including the manual deploy of the web application.

However, to get the most from a container-based architecture, this approach must be avoided. Instead the developers should "extend" the base image of the application server and create a new container with the the application deployment. The new image creation must be completely automated. Only with a full automated build you can implement a true CI/CD pipeline: the image can be automatically build at every check-in, on a daily basis (nightly builds) or on demand.

7.4 Things to avoid with Docker containers

There several advantages in using Docker containers, just to mention the most relevant ones:

1. Containers can implement the *immutability* paradigm. The OS, library files, configuration files, directories, and of course the application are all copied inside a container image. You have the guarantee that the image that was tested in QA will be the same in the production environment exhibiting the the same behavior.
2. Containers are lightweight: the memory footprint of a container is very small. Instead of hundreds or thousands of MBs, the container will only allocate the memory for the main process.
3. Containers are fast: a container is fast like any typical linux process takes to start. Instead of minutes, you can start a new container in fraction of seconds.

However, very often the temptation is to treat containers just like typical virtual machines. This is a very bad error. Containers are not like virtual machine, they are just a wrapper around an application and its needed run-time accessories, required to run that application. Moreover containers have another relevant characteristic: they are fully *disposable*. Considering these aspect it is important to avoid some wrong behaviors:

1. **Do not storage data in containers**: the internal content of a running container is preserved until the container runs. As soon as the container will terminate everything inside it is removed and this is for a purpose. To persist data for an application there is the functionality of `volumes`.
2. **Do not try to change the content of the file system of containers**: while they are running as this is a really wrong practice. The right approach is to update the container image and re-deploy it.
3. **Do not create large images**: creating images with a large size make them more difficult to be used (transferred). Images should contain just the required files for the containerized application to run. With a specific tuning (that can be learned after some practice on this) it is possible usually to slim considerable down the size of an image
4. **Do not use only the "latest" tag of an image**: the `latest` tag it is usually the most recent version of a specific image that sometime could not be the most safe and stable. So it is better to refer to a specific tag that can be considered stable.
5. **Avoid at all to run more than a process in a single container**: technically speaking there are no problems to run multiple processes inside a single container, however that is not the way they were conceived. Multiple processes make difficult to manage the life-cycle of different processes and even more create hard dependencies that do not help exploit the useful aspect of containers, like independent updates of application component. Containers are not Virtual Machines. There can be very few exceptions to this but still this is really something to avoid as much as possible.
6. **Do not store sensitive data within container images**: like password, credential, etc. The most convenient approach is to use `secrets`. They were designed just for this purpose as they make easy to separate injection of sensitive data into application component without any need to pass that information across developers team or to store on unsafe areas.
7. **Do not run process as root**: By default Docker containers run as root, however a more convenient way is to use the `USER` statement when creating a `Dockerfile`.
8. **Avoid using IP addresses of containers**: this is a very bad approach. Docker creates local DNS and container can be referenced by using their name.

CHAPTER

EIGHT

CDP APIS

All main CDP functionalities are exposed through RESTful APIs. The API stack is composed by the following services:

- API gateway (Kong + Postgres)
- API implementation (written in Go)
- API DB (CouchDB)
- API Dashboard (Konga, used just for troubleshooting, not a core component of the architecture)

All APIs are protected by the API gateway with security tokens. Each token is associated to a specific role. Currently there are the following roles (and tokens) defined:

- `admin` which grants access to administration APIs
- `dev` which grants access to APIs used by users and developers

Please contact your system administrator to get a `dev` token.

The high level architecture is depicted below.

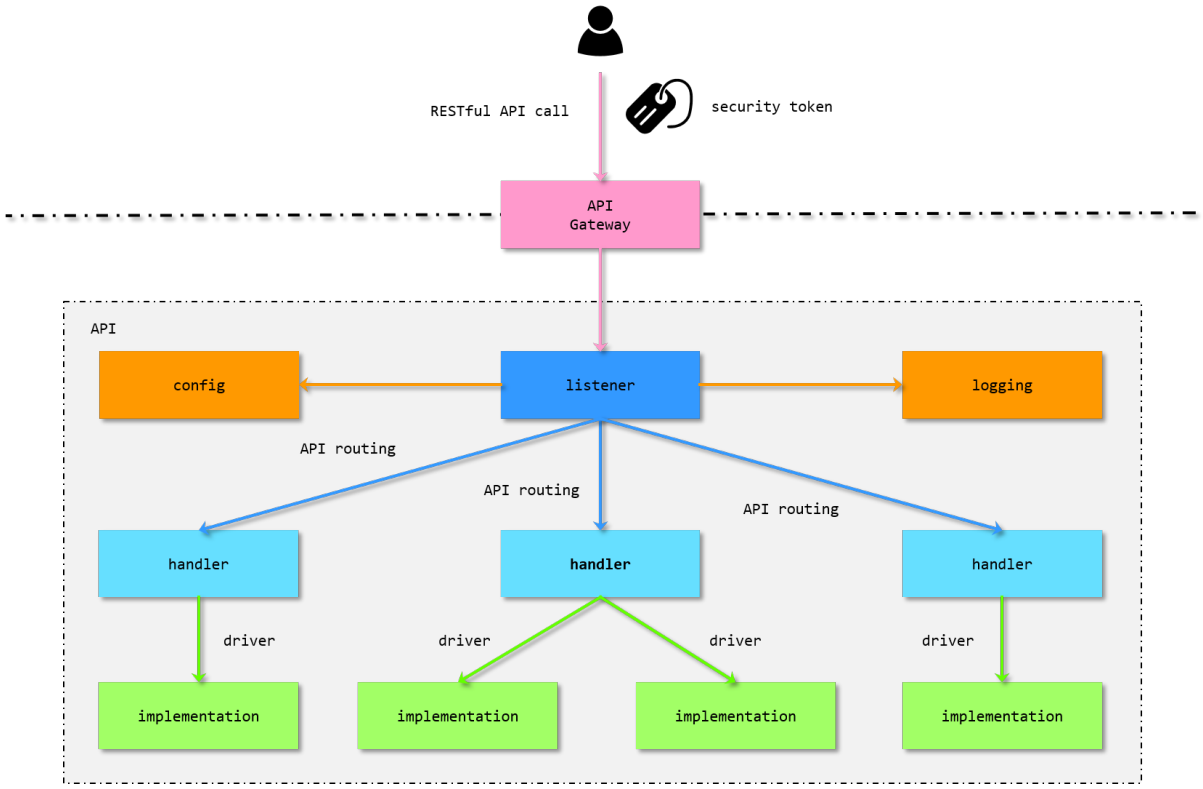

Fig. 1: API high level architecture

The implementation is very modular: a set of *handlers* manage incoming and outgoing communication in HTTP+JSON and, after a request has been received, istantiate an implementation *driver*. The driver perform the work and return the result to the handler which, in turn, converts the result in JSON format and send to the client.

Drivers are used to avoid vendor lock-in and to decouple front-end APIs from specific implementation. For APIs which provide multiple implementations, the specific driver to use is usually configured through the system setting page on the CDP console. An example of such API is the ChatOps system, which can be configured among a choice of providers, i.e. Mattermost or Slack.

8.1 API Overview

CDP APIs are documented below, divided in sections corresponding to specific areas. All interactions must be performed using standard RESTful API methods and JSON format for input and output data.

To test and invoke APIs you can use the command line utility and Swiss Army knife of networking `curl` inside a container with access to the API network, by providing the access token in the `apikey` header. If the apikey is not provided, if it is invalid or if it does not match the security level required, the HTTP status code `401` is returned along with an error message as explained below.

In case of a successful execution, the status code `200` is returned in the HTTP header, along with the specific JSON expected for that particular API. In case of error, a `4xx` or `5xx` status code is returned in the HTTP header depending on the specific error. Along with the HTTP status code, an informative error message is returned with the following JSON format:

```
{"message":"An error message"}
```

Example:

```
$ curl -i -X GET -H apikey: APIKEY' http://api:8000/config/get?key=notexistant/key
HTTP/1.1 404 Not Found
Content-Type: application/json
Date: Sun, 02 Dec 2018 20:29:43 GMT
Content-Length: 59

{"message":"Config value for 'notexistant/key' not found"}
```

The JSON message structure is the same for all APIs and it's used not only for errors but also for informative success messages where a structured JSON output is not required (i.e.: to notify the user the status of an operation wich does not return data).

The base URL for API calls is: `http://api:8000`, only available on the internal api network.

Working examples with test input parameters and sample output results will be provided for all APIs.

8.2 API summary

8.2.1 System Information

GET */version*

GET */healthcheck*

8.2.2 Config

GET */config/get*

POST PUT */config/set*

DELETE */config/delete*

8.2.3 CI/CD

POST */pipelines*

PUT */pipelines*

GET */pipelines*

GET */pipelines/{pid}*

DELETE */pipelines/{pid}*

GET */pipelines/{pid}/log*

POST PUT */pipelines/notify*

8.2.4 Messaging

POST */messages/email*

POST */messages/chatops*

8.2.5 Authentication / Authorization

GET */auth/users*

POST */auth/users/{uid}*

GET */auth/users/{uid}*

PUT PATCH */auth/users/{uid}*

DELETE */auth/users/{uid}*

PUT PATCH */auth/users/{uid}/password*

GET */auth/checkpermission*

PUT */auth/checkpassword*

POST */auth/users/{uid}/bundle*

GET */auth/users/{uid}/bundle*

PUT */auth/users/{uid}/bundle*

DELETE */auth/users/{uid}/bundle*

8.3 API details

8.3.1 GET `/version`

Description

Return current API version.

Input

None

Output

JSON data with `version` attribute holding the CDP version information.

Example

```
$ curl -X GET -H "apikey: APIKEY" http://api:8000/version
{"version":"1.4.2"}
```

8.3.2 GET /healthcheck

Description

Internal call for api container healthcheck. Not intended to be called from outside the API stack.

Input

None.

Output

None.

Example

```
$ curl -X GET -H "apikey: APIKEY" http://api:8000/healthcheck
```

8.3.3 GET /config/get

Description

Read a specific system configuration item.

Input

`key` request parameter with the config item to read.

Output

JSON data with `key` and `value` attributes. An error message if the configuration item is not found or if some problem occurred.

Example

```
$ curl -X GET -H "apikey: APIKEY" http://api:8000/config/get?key=docker/api/version
{"key":"docker/api/version","value":"1.30"}

$ curl -X GET -H "apikey: APIKEY" http://api:8000/config/get?key=not/existing/key
{"message":"Config value for 'not/existing/key' not found"}
```

8.3.4 POST PUT /config/set

Description

Set a specific system configuration item.

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
       -d '{"key":"test/key","value":"testvalue"}' http://api:8000/config/set
{"key":"test/key","value":"testvalue"}
```

8.3.5 DELETE /config/delete

Description

Delete a specific system configuration item.

Input

`key` request parameter with the config item to read.

Output

The following JSON message in case of success:

```
{"message":"Key deleted"}
```

otherwise a message depending on specific error.

Example

```
$ curl -X DELETE -H "apikey: APIKEY" http://api:8000/config/delete?key=unused/key
{"message":"Key deleted"}
```

8.3.6 POST /pipelines

Description

Start a CI/CD pipeline. This API is normally called directly by the software forge application (i.e.: GitLab), usually triggered by a repository push or other events through an outgoing webhook.

The picture below shows the configuration on GitLab:

available in the project's settings –> integrations. In the example above a push trigger is configured, therefore at each push on the remote repository, the pipeline is started and the status reported in the console in CI/CD –> CI/CD Pipelines page.

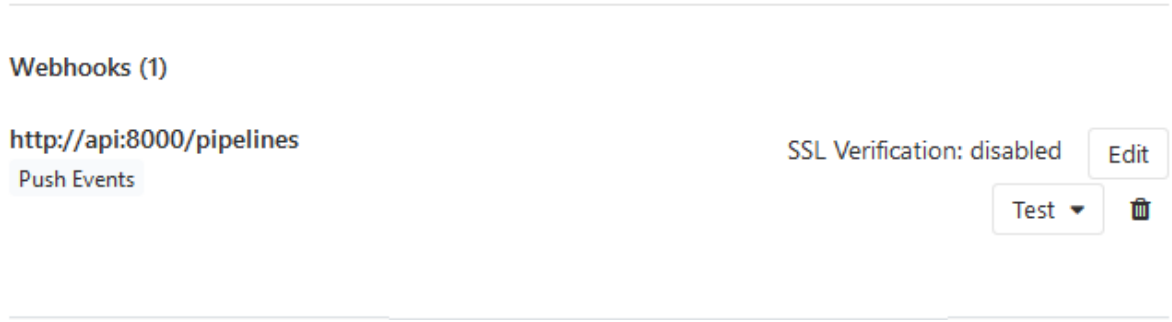

Fig. 2: GitLab push event webhook

Input

Since this API is invoked directly from software forge, the input JSON follows outgoing webhooks standards for your configured software forge application. Not all fields are required, below an example of the minimum JSON request for GitLab:

```
{
    "object_kind":"push",
    "ref":"refs/heads/master",
    "user_name":"Pietro Pizzo",
    "user_username":"ppizzo",
    "user_email":"pietro.pizzo@dxc.com",
    "project":{
        "name":"Demo",
        "git_http_url":"https://gitlab.ecaashackathon.cloudspc.it/ecaas/demo.git",
        "namespace":"ECaaS",
        "path_with_namespace":"ecaas/demo"
    }
}
```

For more information about GitLab JSON format used for webhooks and the meaning of each field, please read the official documentation at the following URL: https://docs.gitlab.com/ce/user/project/integrations/webhooks.html

Output

JSON message with the auto-generated pipeline identifier. An error message if some problem occurred.

The pipeline ID returned is the name of the Docker service started to execute the CI/CD pipeline. The service hangs out in the system for 10 minutes after pipeline ends, in order to give CDP developers time for troubleshooting in case of problems.

Example

```
$ curl -i -X POST -H "apikey: APIKEY" -d '{
    "object_kind":"push",
    "ref":"refs/heads/master",
    "user_name":"Pietro Pizzo",
    "user_username":"ppizzo",
    "user_email":"pietro.pizzo@dxc.com",
    "project":{
        "name":"Demo",
        "git_http_url":"https://gitlab.ecaashackathon.cloudspc.it/ecaas/demo.git",
        "namespace":"ECaaS",
        "path_with_namespace":"ecaas/demo"
    }
}' http://api:8000/pipelines
{"message":"Pipeline running. ID: cdp-pipeline-ee530ab8-dc8f-4228-955e-c14365624a61
↪"}
```

8.3.7 PUT /pipelines

Description

Store or update pipeline information. This API can be called during the pipeline execution in order to sync the internal pipeline information with the current status.

Input

Even if this API must be called with a PUT method, to ease API calls from shell scripts, pipeline attributes are in the URL, like any GET call, and pipeline log is in the HTTP body.

Parameters in the URL:

- `pid` the pipeline ID as returned from the *POST/pipelines* api described above
- `project` project path, as per `path_with_namespace` parameter of *POST/pipelines*
- `branch` project branch being built
- `user` user who triggered the build
- `status` numeric definition of pipeline status (see below for admitted values)
- `start` pipeline start time
- `stop` pipeline stop time; if not present it means the pipeline is still running

The `status` parameter must be one of the following:

- `0`: pipeline passed, succesfull completition
- `1`: pipeline failed, an error arose during execution
- `2`: pipeline test failed
- `3`: pipeline running
- `4`: pipeline halted, if an external signal caused the pipeline to stop prematurely

`start` and `stop` dates must be full timestamps with timezone information as returned by the command `date "+%FT%T%Z"` (i.e.: `2006-01-02T15:04:05MST`)

Along with URL parameters, the body must contain the entire pipeline log so far, in `text/plain` format (not JSON).

Output

The following JSON message in case of success:

```
{"message":"Pipeline stored"}
```

otherwise a message depending on specific error.

Example

In the example below we assume that the pipeline build log is stored in the file named by the variable `${logfile}`:

```
$ curl -X PUT -H "apikey: APIKEY" -H "Content-Type: text/plain" --data-binary @$
↪{logfile}\
       'http://api:8000/pipelines?pid=cdp-pipeline-ee530ab8-dc8f-4228-955e-
↪c14365624a61&project=ecaas/demo&status=3&branch=master&user=ppizzo&start=2018-09-
↪23T15:04:05CEST'
{"message":"Pipeline stored"}
```

8.3.8 GET /pipelines

Description

Read the list of pipelines triggered.

Input

None

Output

JSON array, each element corresponding to a triggered pipeline. Each element of the array has the following attributes:

- `_id` the pipeline ID as returned from the *POST/pipelines* api described above
- `_rev` an internal revision information identifier; you don't have to care about it
- `project` project path, as per `path_with_namespace` parameter of *POST/pipelines*
- `branch` project branch being built
- `user` user who triggered the build
- `status` numeric definition of pipeline status as described in *PUT/pipelines*
- `badgeurl` url of pipeline status badge image corresponding to the status attribute
- `start` pipeline start time
- `stop` pipeline stop time; if not present it means the pipeline is still running

Example

(Output of curl command formatted for better readability)

```
$ curl -X GET -H apikey: APIKEY' http://api:8000/pipelines
[
  {
    "_id":"cdp-pipeline-3380dfd0-e8ef-4bdf-8fef-9b0ca8a4ce15",
    "_rev":"12-ac23fd7e4fb080cde41bb370f82f6884",
    "project":"ppizzo/demo",
    "branch":"patch-1",
    "user":"dilbert",
    "status":0,
    "badgeurl":"/admin/img/pipeline-passed-brightgreen.svg",
    "start":"2018-08-03T18:44:58+02:00",
    "stop":"2018-08-03T18:45:16+02:00"
  },{
    "_id":"cdp-pipeline-4c2e8080-c0f2-415e-baa1-7513ce347a10",
    "_rev":"6-2ae7eea2b20b3a270cb18a991eb9eafb",
    "project":"ppizzo/demo",
    "branch":"test",
    "user":"ppizzo",
    "status":1,
    "badgeurl":"/admin/img/pipeline-failed-red.svg",
    "start":"2018-08-03T18:43:04+02:00",
    "stop":"2018-08-03T18:43:21+02:00"
  },{
    "_id":"cdp-pipeline-638807d1-2639-41cd-8978-62ac14b2d907",
    "_rev":"12-6717db0d0cb03d75234410763a44ee05",
    "project":"ppizzo/demo",
```

(continues on next page)

(continued from previous page)

```
    "branch":"master",
    "user":"ppizzo",
    "status":0,
    "badgeurl":"/admin/img/pipeline-passed-brightgreen.svg",
    "start":"2018-08-03T18:42:13+02:00",
    "stop":"2018-08-03T18:42:26+02:00"
  }
]
```

8.3.9 GET /pipelines/{pid}

Description

Read a specific pipeline.

Input

- `pid` the pipeline ID as returned from the *POST/pipelines* api described above

Output

JSON data with the following attributes:

- `_id` the pipeline ID as returned from the *POST/pipelines* api described above
- `_rev` an internal revision information identifier; you don't have to care about it
- `project` project path, as per `path_with_namespace` parameter of *POST/pipelines*
- `branch` project branch being built
- `user` user who triggered the build
- `status` numeric definition of pipeline status as described in *PUT/pipelines*
- `badgeurl` url of pipeline status badge image corresponding to the status attribute
- `start` pipeline start time
- `stop` pipeline stop time; if not present it means the pipeline is still running

Example

(Output of curl command formatted for better readability)

```
$ curl -X POST -H apikey: APIKEY' http://api:8000/pipelines/cdp-pipeline-638807d1-
↪2639-41cd-8978-62ac14b2d907
[
  {
    "_id":"cdp-pipeline-638807d1-2639-41cd-8978-62ac14b2d907",
    "_rev":"12-6717db0d0cb03d75234410763a44ee05",
    "project":"ppizzo/demo",
    "branch":"master",
    "user":"ppizzo",
    "status":0,
    "badgeurl":"/admin/img/pipeline-passed-brightgreen.svg",
    "start":"2018-08-03T18:42:13+02:00",
    "stop":"2018-08-03T18:42:26+02:00"
  }
]
```

8.3.10 DELETE /pipelines/{pid}

Description

Delete a specific pipeline.

Input

- `pid` the pipeline ID as returned from the *POST/pipelines* api described above

Output

The following JSON message in case of success:

```
{"message":"Pipeline deleted"}
```

otherwise a message depending on specific error.

Example

```
$ curl -X DELETE -H "apikey: APIKEY" http://api:8000/pipelines/cdp-pipeline-
↪638807d1-2639-41cd-8978-62ac14b2d907
{"message":"Pipeline deleted"}
```

8.3.11 GET /pipelines/{pid}/log

Description

Read the pipeline build log as stored by *PUT/pipelines*.

Input

- `pid` the pipeline ID as returned from the *POST/pipelines* api described above

Output

Build log so far in `plain/text` format. The build log, despite the content-type returned, contains an HTML fragment which can be easily embedded in web pages (i.e.: the CDP console).

The build log has the following structure:

```
<pre>
    header and pipeline information
</pre>

<pipeline-phase name="Prepare">
    build log of the specific pipeline stage
</pipeline>
<pipeline-phase name="Build">
    build log of the specific pipeline stage
</pipeline>
<pipeline-phase name="Test">
    build log of the specific pipeline stage
</pipeline>
<pipeline-phase name="Deploy">
    build log of the specific pipeline stage
</pipeline>
<pipeline-phase name="Stop">
    build log of the specific pipeline stage
</pipeline>
```

(continues on next page)

(continued from previous page)

```
<pipeline-phase name="Start">
    build log of the specific pipeline stage
</pipeline>

<pre>
    pipeline end time and duration
</pre>
```

Example

(Output reduced for better readability)

```
$ curl -X POST -H apikey: APIKEY' /pipelines/cdp-pipeline-638807d1-2639-41cd-8978-
↪62ac14b2d907/log

<pre>
Started : 2018-08-03T18:42:13CEST
User    : ppizzo
Project : ppizzo/demo
Branch  : master
GIT URL : https://console.ecaashackathon.cloudspc.it/gitlab/ppizzo/demo.git
PID     : cdp-pipeline-638807d1-2639-41cd-8978-62ac14b2d907
</pre>

<pipeline-phase name="Prepare">
Cloning into 'demo'...
</pipeline-phase>

(...)
other pipeline stages
(...)

<pipeline-phase name="Start">
<span style="color:green">[2018-08-03T18:42:21.815018] - BEGIN execution -␣
↪WORKFLOW 1</span>
<span style="color:green">[2018-08-03T18:42:21.815054] - TASK 1.0 (run) - RUN</
↪span>
Ignoring unsupported options: build

<span style="color:green">[2018-08-03T18:42:26.769288] - END execution - WORKFLOW␣
↪1 - Elapsed: 4.95427 seconds</span>
Updating service ppizzo_demo_master_demo_demoapp (id: vloxoyguqs8m2umyrmr75jlo3)
</pipeline-phase>

<pre>
Pipeline ended 2018-08-03T18:42:26CEST took 13 seconds.
</pre>
```

8.3.12 POST PUT /pipelines/notify

Description

Notify user about pipeline status when the pipeline ends. The user notified is the one who triggered the build. The notification is routed through the configured channel (i.e.: Mattermost, Slack, etc.) and contains a link to the pipeline detail page on CDP console.

Input

JSON data with the following attributes:

- `pid` the pipeline ID as returned from the *POST/pipelines* api described above
- `project` project path, as per `path_with_namespace` parameter of *POST/pipelines*
- `branch` project branch built
- `user` user who triggered the build
- `status` numeric definition of pipeline status as described in *PUT/pipelines*

Output

The following JSON message in case of success:

```
{"message":"Notification sent for pipeline PID}
```

where `PID` is the pipeline ID.

Example

```
$ curl -i -X POST -H "apikey: APIKEY" -d '{
    "pid":"cdp-pipeline-638807d1-2639-41cd-8978-62ac14b2d907",
    "project":"ppizzo/demo",
    "branch":"master",
    "user":"ppizzo",
    "status":0
}' http://api:8000/pipelines/notify
{"message":"Notification sent for pipeline cdp-pipeline-638807d1-2639-41cd-8978-
↪62ac14b2d907"}
```

8.3.13 POST /messages/email

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.14 POST /messages/chatops

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.15 GET /auth/users

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.16 POST /auth/users/{uid}

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.17 GET /auth/users/{uid}

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.18 PUT PATCH /auth/users/{uid}

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.19 DELETE /auth/users/{uid}

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.20 PUT PATCH /auth/users/{uid}/password

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.21 GET /auth/checkpermission

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.22 PUT /auth/checkpassword

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.23 POST /auth/users/{uid}/bundle

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.24 GET /auth/users/{uid}/bundle

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.25 PUT /auth/users/{uid}/bundle

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

8.3.26 DELETE /auth/users/{uid}/bundle

Description

TBD

Input

JSON data with the following attributes:

- `key` the config item to set
- `value` the value to set

Output

JSON data with `key` and `value` attributes. An error message if some problem occurred.

Example

```
$ curl -X POST -H apikey: APIKEY' \
```

CHAPTER

NINE

REFERENCES

Docker: https://www.docker.com/

Docker Access Control model: https://docs.docker.com/ee/ucp/authorization/

Docker Enterprise Edition Architecture: https://docs.docker.com/ee/docker-ee-architecture/

Dockerfile reference: https://docs.docker.com/engine/reference/builder/

Docker compose file reference: https://docs.docker.com/compose/compose-file/

GIT: https://git-scm.com/book/en/v2

GitLab: https://about.gitlab.com/product/

The Go programming language: https://golang.org/

Kibana: https://www.elastic.co/products/kibana

Kubernetes: https://kubernetes.io/

Markdown: https://daringfireball.net/projects/markdown/

Mattermost: https://mattermost.com/

Minio: https://www.minio.io/

Portworx: https://portworx.com/

Python programming language: https://www.python.org/

Slack: https://slack.com/

YAML language specification: http://yaml.org/