Rapporti Tecnici INAF
INAF Technical Reports

Number	85
Publication Year	2021
Acceptance in OA@INAF	2021-05-04T12:32:04Z
Title	DevOps at IA2 data centre
Authors	ZORBA, SONIA, URBAN, CRISTIANO
Affiliation of first author	O.A. Trieste
Handle	http://hdl.handle.net/20.500.12386/30951, http://dx.doi.org/10.20371/INAF/TechRep/85

DevOps at IA2 data centre

Issue/Rev. No.: 1.0
Date: April 28, 2021
Authors: Sonia Zorba, Cristiano Urban
Approved by: Massimo Sponza

Contents

The word DevOps comes merging the words "software development" and "IT operations". It consists in a set of methodologies where a professional having both programming and system administrator skills configures a set of tools in order to automatize testing and deploy of applications and services. This report describes how IA2 data centre adopted these techniques using Docker, GitLab CI and Ansible.

1 Docker

Docker is a set of tools and processes used to create containers, a lightweight form of virtualization where some processes live in an isolate space (different init process, isolated root file system and isolated network stack) but they still share the kernel with the host (the machine that created the container).

A container is created from an image, a file conceptually similar to a virtual machine snapshot, but usually generated using a Dockerfile (a script that defines the steps necessary to build the image). Docker images can be tagged and distributed to servers called registries. Registries can be both public and private and the world's largest registry is the Docker Hub[1].

Dockerfile syntax[2] allows inheriting from other images using the `FROM` keyword, as shown in the following example:

```
FROM openjdk:14-jdk-alpine
COPY target/vospace.jar vospace.jar
ENTRYPOINT ["java","-jar","/vospace.jar"]
```

Listing 1: Dockerfile for IA2 VOSpace service. The image inherits from `openjdk:14-jdk-alpine`, an official OpenJDK image available on Docker Hub. The jar file is added to the image and executed at container startup using the `ENTRYPOINT` command.

The image is built from the Dockerfile using the dedicated Docker client command: `docker build`. It can then be used to start a container using the `docker run` command.

By default ports of a service running in a container are not reachable from the host and it is necessary to explicitly expose them using the `-p` option of the `docker run` command. The option also allows to map a port number in the container to a different number in the host.

```
docker run -d -p 8085:8080 vospace
```

Listing 2: Runs a container from vospace latest image. The port 8080 in the container corresponds to the port 8085 in the host. The option `-d` (detach) is used to run the container in background.

Docker also handles virtual network interfaces and a DNS service for network communication between containers. In this way a container can see other containers on the same network by using user-defined hostnames. Docker always creates a default network interface (`docker0`) and then can add other interfaces defined by the user.

By default containers are stateless, meaning that every change made in a running container is lost when the container is stopped. Information can be stored permanently by sharing files between the host and the container. This is achieved defining "volumes" that are mounted on the container (a folder in the host corresponds to a folder in the container).

1.1 Container orchestration

Each Docker container should run a single service or application. It is possible to run multiple services creating custom scripts or using a process manager like supervisord, but it is discouraged. When multiple dockerized services need to communicate together they should be orchestrated using dedicated tools. The simplest of these tools is Docker Compose, that runs multiple containers according to a configuration written in a YAML file. It is very simple and useful for simple scenarios running on a single host. When multiple nodes are needed but the set up is still simple, the best choice is Docker Swarm. More complex cases can be handled using Kubernetes or OpenShift.

[1] https://hub.docker.com/
[2] https://docs.docker.com/engine/reference/builder/

```
---
version: '3.7'
services:
  portal:
    build: .
    ports:
      - "8080:8080"
    environment:
      - JDBC_URL=jdbc:mysql://mariadb:3306/portal
  mariadb:
    image: "mariadb/server:10.3.13"
    environment:
      - MARIADB_ALLOW_EMPTY_PASSWORD=true
    volumes:
      - ./portal-db-init.sql:/docker-entrypoint-initdb.d/portal-db-init.sql
```

Listing 3: docker-compose.yml file defining the interaction between two dockerized services. The portal service is built from a Dockerfile in the current directory, while the mariadb service is run from an image available on Docker Hub. The database is initialized adding a file to the `docker-entrypoint-initdb.d` folder defined by the mariadb image. Portal service connects to the database in the mariadb container and exposes the 8080 port to the host.

1.2 Dockerizing an application

Dockerizing an existing application could be difficult in some situations and it is usually easier to start the development keeping in mind that the application needs to run on a container too. The following list describes some facts to consider based on our experience:

- the application needs to handle the dependencies in an automated way, for example using tools like Maven, pip, npm, Composer and so on. If the application relies on internally developed libraries it could be necessary to set up also an artifact repository (e.g. JFrog Artifactory, Apache Archiva or the package registry integrated in the most recent GitLab versions).
- the general rule is "one service per container", so if the application is modular this usually implies to create a different Dockerfile for each service and then orchestrate them.
- most of the Docker images rely on environment variables for configuring the application at container startup; in some cases external configuration files are mounted inside the container.
- orchestrated containers know each others using names provided by a DNS that is different from the host DNS and this can create issues for services that generate URLs (e.g. an URL like http://fileserver works between orchestrated containers but needs to be translated to http://localhost:port when it is called from the host machine). Section 4.7 describes the workaround we used to overcome this problem.
- modern frameworks usually explain how to use them with Docker, so always check the official documentation before starting to containerize[3].
- running production databases on containers is a debated topic on the Internet and some people reported data corruption issues. At IA2 we are running databases on containers only for testing purposes.
- there are some techniques to optimize Docker images (both in term of build time and resulting image size), like the multi-stage builds[4]. Some of these techniques are specific for the package manager used (e.g. Maven or npm). In general, it is a good idea to keep in mind that each Dockerfile instruction generates an intermediate image layer that is cached, so most frequently changed files should be added in the latest instructions when possible. Next section provides some concrete examples.

[3] Example: official Spring Boot Docker documentation at https://spring.io/guides/gs/spring-boot-docker/
[4] https://docs.docker.com/develop/develop-images/multistage-build/

1.3 Optimization of builds

1.3.1 Multi-stage builds

It's very common to need a different set of dependencies for building and running a dockerized application. For example our TAP service image needs Gradle for the build and Tomcat for running. These two dependencies are available from two different official Docker images and this is a perfect case for a multi-stage builds.

It consists in having a Dockerfile with multiple `FROM` instructions. Each `FROM` instruction starts a new build stage using a different base image and it is possible to copy built artifacts from the previous stage into the new stage using the `COPY --from` instruction.

Following snippet shows part of the Dockerfile we use to build our TAP service image:

```
FROM gradle:jdk8

# war file is built and extracted to the /tap directory
# ...

FROM tomcat:9-jdk14
# Generated files are copied from the previos stage
COPY --from=0 /tap /usr/local/tomcat/tap

# ...
```

1.3.2 Downloading dependencies in the first layers

Another common optimization technique consists in exploiting layer caching. Each command in a Dockerfile produces an intermediate image called layer. Layers are cached and they are rebuilt only if changes are detected. In a software project, code changes frequently but dependencies are updated only sometimes. Dependency retrieval may be time consuming, so downloading them in the first layers and then adding the changed code on the next layers can speed up the building process. When using a dependency manager (Maven, npm, Gradle, Composer, pip and so on) dependencies are usually listed in a specific file (pom.xml, package.json, ...). The optimization trick consists in adding only that file (that shouldn't change very often) to the build, calling the command that download the dependencies and then adding the other files and compiling the project.

Following snippet shows this technique on a npm application (we are using this for IA2 portals):

```
# Only package.json is added to the image
ADD package.json .
# Dependencies are downloaded
RUN npm install
# Other project files are added
ADD src src
# Project is compiled
RUN npm run build
```

1.4 Executing commands at container startup

In a Dockerfile the `ENTRYPOINT` instruction specifies a command that will always be executed when the container starts and the `CMD` instruction specifies arguments that will be fed to the `ENTRYPOINT`. Docker default entrypoint is `/bin/sh -c`. If `CMD` is defined from the base image, setting `ENTRYPOINT` will reset `CMD` to an empty value.

Both instructions have a shell form (`ENTRYPOINT command param1 param2`) and an exec form (`ENTRYPOINT ["executable", "param1", "param2"]`). The second one is preferred because it sends `SIGTERM` to the executable when `docker stop` is called, allowing the application to shutdown cleanly.

In the same way, if the entrypoint is a custom shell script that executes some commands and then starts an application in its last line, it should use the form `exec application-name` instead of executing the application directly, because using exec will replace the parent process, rather than having two processes running, and will let the application to receive Unix signals.

2 GitLab CI/CD

Continuous Integration consists in triggering some automated processes every time a developer pushes a commit to a repository (git or other version control systems). These processes usually include build and automated testing. In this way when members of a team integrate their work it is possible to immediately detect critical errors, resulting in a more reliable software. Continuous Delivery consists in producing also some artifacts that are ready to be deployed through a manual action. Continuous Deployment consists in fully automatizing the deploy phase too. Usually before deploying to production the applications are deployed to a nearly exact replica of the production environment, called staging environment. These practices are usually collectively named CI/CD.

GitLab provides CI/CD functionalities that can be configured adding a file named .gitlab-ci.yml in the root of the repository. The file specifies a set of jobs that are executed on a dedicated machine.

Jobs are grouped in phases called stages. Jobs of the same stage can be executed in parallel and stages are executed sequentially. By default if a job fails the next stage is not called.

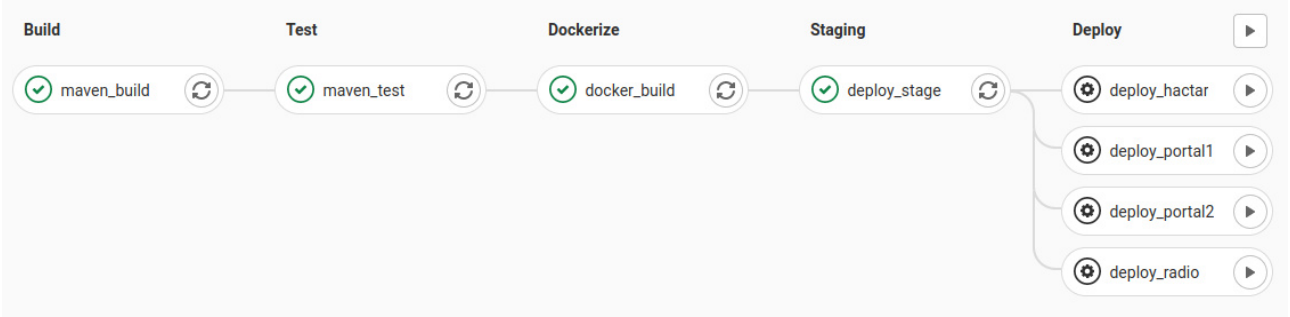

Figure 1: FileService pipeline in the GitLab interface displaying five stages. Each rounded box represents a job. First four stages have been automatically executed, jobs of the deploy stage can be manually triggered by clicking on the play buttons.

2.1 GitLab Runner setup

GitLab Runners are applications that execute CI/CD jobs. On CentOS the package gitlab-runner can be installed using yum after adding the GitLab repositories[5].

To associate a Runner with GitLab project go to repository settings, click on CI/CD, expand the Runners section and copy the registration token. This token can then be used with the command `sudo gitlab-runner register`. Here an example of usage:

```
Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com/):
https://www.ict.inaf.it/gitlab/
Please enter the gitlab-ci token for this runner:
<token>
Please enter the gitlab-ci description for this runner:
[devops.ia2.inaf.it]: Generic shell Runner
Please enter the gitlab-ci tags for this runner (comma separated):
shell
Whether to lock the Runner to current project [true/false]:
[true]:
Registering runner... succeeded                     runner=[...]
Please enter the executor: docker, shell, ssh, docker+machine, docker-ssh, parallels,
    ↪ virtualbox, docker-ssh+machine, kubernetes:
shell
```

[5] https://docs.gitlab.com/runner/install/linux-repository.html

As you can see from the output there are different kind of executors. At the moment we are using shell executors (jobs are executed on a shell of the machine having the runner installed) and docker executors (jobs are executed inside a docker container).

Tags are used to associate a runner with specific jobs, as shown in the following example.

Listing 4: This .gitlab-ci.yml file defines two jobs. The first is executed inside a Docker container created from a Maven image available on Docker Hub, the second is executed on the shell available on the Runner machine. The list of stages can be omitted if the default stage names are used.

```
stages:
  - test
  - deploy

test_job:
  stage: test
  tags:
    - docker
  image: maven:3.6.3-openjdk-14
  script:
    - mvn clean test

deploy_job:
  stage: deploy
  tags:
    - shell
  script:
    - ansible-playbook deploy.yml
```

By default a runner is locked to the project from which its token has been created. This settings can be changed editing the runner settings on GitLab web interface, so the same runner can be used by multiple projects. When runners are installed on different servers used for different purposes it could be useful to add the server name in the tag itself (e.g. shell-testing, shell-staging).

In the previous listing we introduced the .gitlab-ci.yml syntax. In this introduction we are not going to explain all the possible job configuration options, but some of them will be presented together with the concrete examples listed in the next sections. For the complete list of keywords see the dedicated documentation page[6].

2.2 CI/CD Variables

From GitLab interface it is possible to configure some variables that are applied to environments via the runners. Maintainer grant on the repository is required to see or edit these variables.

There are two kinds of variables: variables and files. The expression `${var}` in a .gitlab-ci.yml file returns the variable content for simple variables and the path of a temporary file containing the variable content for file variables.

It is possible to configure these environment variables both at repository level or at group level. For example we configured in the IA2 GitLab group the URLs of some services that can be used by all IA2 applications like RAP, GMS and the internal Maven repository. GitLab also provides some predefined variables[7] that are useful in some situations, for example when accessing protected resources like private git repositories or Docker registries (see next section).

We encountered a known bug that truncates variables containing the dollar symbol[8]. The issue can be solved doubling the dollar.

[6] https://docs.gitlab.com/ee/ci/yaml/
[7] https://docs.gitlab.com/ee/ci/variables/predefined_variables.html
[8] https://gitlab.com/gitlab-org/gitlab-foss/-/issues/45173

2.3 Git strategy configuration

When a CI/CD pipeline starts, GitLab retrieves the updated project data from the repository and it can do it using clone or fetch. By default pipelines are configured to use the fetch strategy, since it is faster. However with some git versions (in particular the one included with CentOS 7) the following error can happen: "fatal: git fetch-pack: expected shallow list". This can be solved going to GitLab CI/CD Settings and selecting "git clone" strategy in general pipeline settings. The strategy can also be defined in the .gitlab-ci.yml using the `GIT_STRATEGY` variable.

2.4 GitLab Docker registry

GitLab also contains an internal Docker registry. Images can be built and pushed to this registry relying on GitLab predefined environment variables as shown in the following snippet:

```
deploy:
  stage: dockerize
  tags:
    - shell
  only:
    - master
  script:
    - docker login -u "${CI_REGISTRY_USER}" -p "${CI_REGISTRY_PASSWORD}" "${CI_REGISTRY}"
    - docker build -t "${CI_REGISTRY_IMAGE}" .
    - docker push "${CI_REGISTRY_IMAGE}"
```

Since INAF Docker registry uses self-signed certificates this works if in the runner machine the repository is explicitly allowed in the `/etc/docker/daemon.json` file:

```
{
    "insecure-registries" : [ "git.ia2.inaf.it:5050" ]
}
```

GitLab imposes a naming convention to the images pushed on its registry: the name of the image must start as defined in the `CI_REGISTRY_IMAGE` variable. If multiple Docker images need to be generated from the same repository it is possible to append to the name a slash and an additional specifier.

2.5 Running jobs only if certain conditions are met

Many examples available on the Internet use the `only` keyword (and its negated form `except`) to define when to add jobs to pipelines. Recently GitLab developers announced that these two keywords will be replaced by the new and more powerful `rules` syntax.

Rules are evaluated in order until the first match. If a rule matches and it has `when: never` as the attribute the job is not added to the pipeline.

```
  rules:
    - if: '$CI_COMMIT_BRANCH != "master"'
      when: never
    - changes:
        - repo_subfolder/**/*
```

Listing 5: In this example the job is executed when some commits are pushed to the master branch and they modify files inside a specific subfolder

In some cases it could be necessary to forcibly run a job even if it has been excluded by the `rules` settings. This can be done going to the Pipelines page on GitLab and clicking on the "Run Pipeline" button.

The `when: manual` option is used to require a manual intervention in order to run the job (a "play" button appears on the job graphical representation). We use this setting for jobs that deploy on production.

2.6 Job artifacts

In some cases it is necessary to keep some of the files generates by a job after its completion. The list of these files and their expiration time can be defined using the `artifacts` keyword.

```
  artifacts:
    paths:
      - target/mypackage.jar
    expire_in: 7 days
```

By default, jobs in later stages automatically download all the artifacts created by jobs in earlier stages. Using the keywords `dependencies` it is possible to limit a job to download only artifacts from certain earlier jobs. More specific settings are available using the `needs` keyword. In artifacts, paths are always relative to the project directory.

2.7 Jobs inheritance

When multiple similar jobs need to be run it could be useful to define a template and inherit from it (you can think of it as an abstract class). The following example uses only YAML anchors (that are part of standard YAML syntax) to define inheritance:

```
.my_abstract_job_definition: &my_abstract_job
  stage: build
  variables: &my_common_vars
    COMMON_VARIABLE: value1
  script:
    - my_command ${COMMON_VARIABLE} ${JOB_SPECIFIC_VARIABLE}

concrete_job1:
  <<: *my_abstract_job
  variables:
    <<: *my_common_vars
    JOB_SPECIFIC_VARIABLE: value2
```

Listing 6: `concrete_job1` inherits from the abstract job previously defined. The variables section is also inherited, so that the values are merged and not overwritten. Jobs starting with dot are treated by GitLab as hidden jobs and are not executed.

An alternative to YAML anchors is the `extends` keyword. It is more powerful since it can be used also to reuse configuration from included configuration files combining it with the `include` keyword. The `extends` keyword merges hashes but not arrays (so arrays are overwritten) and it is possible to extend multiple parent jobs.

2.8 Child pipelines

It is possible to define a job that triggers an entire new pipeline, defined in an external YAML file:

```
run_child1_pipeline:
  stage: build
  trigger:
    include: .gitlab-ci-child.yml
```

Child pipelines run concurrently and they don't depend on each other, so they are very useful to run in parallel sequence of jobs without having to wait the completion of unrelated jobs on the same stage. Variables defined in the job that starts the child pipeline are available to the child pipeline. It is possible to exploit this fact to define a kind of "template pipeline" and execute it multiple times with different variables. This is useful when it is necessary to define multiple similar sequences of jobs.

It is also possible to execute a child pipeline on a different project (multi-project pipeline) or even dynamically generating a pipeline using a generator job that produces a pipeline descriptor as its artifact.

Examples of these techniques are described in section 4.3.

3 Ansible

Ansible is a tool for automatizing installation and configuration processes. Ansible commands are started from a controller server to controlled nodes. Ansible needs to be installed only on the controller, while the nodes only need SSH and Python. This agentless approach is usually cited as one of the main advantages of Ansible compared to other similar tools like Chef or Puppet. At IA2 we rely on Docker Compose for deploying on the staging environment and on Ansible to deploy in production.

3.1 Inventory

Ansible selects the hosts to control from a list called inventory[9]. If no inventory is passed as command line parameter it checks the content of the `/etc/ansible/hosts` file. Ansible executes commands on a group of hosts identified using a label. In this way the same command is executed on multiple nodes. At IA2 we don't have such redundancy, so most of the groups are composed by a single node or a quite small set of nodes.

SSH users and keys to use can be specified in the inventory too. The configuration provided by ~/.ssh/config is used by default. For testing if SSH connection used by Ansible is properly configured the ping module can be used:

```
ansible mygroup -m ping
```

3.2 Playbooks

Ansible executes commands based on YAML files called playbooks. A playbook is composed by a list of plays, that are a list of tasks in turn. A play defines on which hosts the tasks are executed and a task defines how to call the Ansible modules that execute the commands. The following example should clarify the purpose of this hierarchy:

```
- name: Example play
  hosts: group1

  tasks:
  - name: Copy a file
    copy:
      src: "/tmp/local-file.txt"
      dest: "/tmp/remote-file.txt"
```

Listing 7: This simple playbook contains one play that is executed on the hosts belonging to group1. The play has one task that calls the Ansible builtin copy module to copy a file from the controller node to the other nodes.

Most Ansible modules perform their actions only if they are necessary. For example the copy module doesn't copy a file if it detects that a file with the same name and the same content is already in the desired location. In this way a service is not restarted if its configuration file is not changed. Ansible behaves to reach a desired state rather than simply executing a sequence of tasks.

Ansible provides many built-in modules, then the Ansible Galaxy portal[10] provides additional modules developed by the community.

Variables can be used in playbooks using the Jinja2 syntax[11]. In the simplest case a variable is retrieved putting it between doubled curly braces (`{{ myvar }}`). The syntax also provides many filters to manipulate data and others built-in functions.

[9] https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html
[10] https://galaxy.ansible.com/
[11] https://docs.ansible.com/ansible/latest/user_guide/playbooks_templating.html

Another important feature available in playbooks are handlers. They are special tasks that only run when notified. By default they are executed after all the tasks in a particular play have been completed. Handlers are particularly useful for restarting services only when their configuration is changed.

3.3 Ansible Vault

Ansible Vault[12] is the mechanism provided by Ansible for encrypting sensitive data.

Listing 8: Following command encrypts a string using the password stored in the given password file

```
ansible-vault encrypt_string --vault-password-file vault-pass.txt
```

Listing 9: Strings encrypted with Ansible Vault can be used inside YAML files and it is considered safe to put them under version control. It is important to avoid spaces at the end of the vault string, otherwise errors in the decrypting process could happen.

```
client_id: myid
client_secret: !vault |
          $ANSIBLE_VAULT;1.1;AES256
          393462386164336437383561346233666564393431393132633738336163343639336533366162356
          663564373432386230393263336430623435393761373135 0a653031386337616136306661366338
          353364646531303661623139363130336265303132376139393764346136356231613962626563 39
          616435633136333837 0a6336353136386561306532363932623861646131303434326332626166373
          3634
```

A playbook can then be executed using those variables with the password file for decrypting them:

```
ansible-playbook my-playbook.yaml" --extra-vars "@vars.yaml" --vault-password-file pass.txt
```

3.4 Security considerations

For performing administrative tasks like installing new software or restarting services the controller node must have root access on the controlled hosts. This means that the controller node is a sensitive machine and had to be properly protected.

It is possible to execute Ansible tasks with a user different than root and use the `become` directive to use sudo. This might make you think that it would be possible to limit Ansible to execute only certain commands configuring /etc/sudoers, however this doesn't work, since most of Ansible commands are executed through Python calls[13] and Ansible documentation clearly states that "you cannot limit privilege escalation permissions to certain commands"[14]. Moreover we automated also tasks like the creation of new systemd services and unrestricted root access is required for that kind of actions.

3.5 Calling Ansible from GitLab CI/CD

Calling Ansible from GitLab CI is pretty straightforward: it's just a matter of invoking the `ansible-playbook` command in the job script directive. However there are several ways to handle the variables Ansible needs.

In the simplest scenarios the variables can be stored as GitLab CI variables and then passed to the playbook using the `--extra-vars` option:

```
ansible-playbook "${CI_PROJECT_DIR}/deploy.yml" \
    --extra-vars "my_ansible_var1=${MY_GITLAB_VAR1}" \
    --extra-vars "my_ansible_var2=${MY_GITLAB_VAR2}"
```

Also GitLab CI file variables could be used. In that case the `-extra-vars` option value has to start with a "@" symbol (that indicates to load a file).

[12] https://docs.ansible.com/ansible/latest/user_guide/vault.html
[13] https://github.com/ansible/ansible/issues/5712
[14] https://docs.ansible.com/ansible/latest/user_guide/become.html#privilege-escalation-must-be-general

When the configuration is more complex, the Ansible variable files can be added to repository, encrypting the sensitive values using Ansible vaults. Then the vault password can be stored in a GitLab CI variable and used to decrypt the values during the job execution:

```
deploy_job:
  stage: deploy
  before_script:
    - echo "$ANSIBLE_VAULT_PASSWORD" > pass.txt
  script:
    - ansible-playbook "$CI_PROJECT_DIR/deploy.yml"
        --extra-vars "@~/my_ansible_env_file.yml"
        --vault-password-file pass.txt
  after_script:
    - srm pass.txt
```

The temporary file pass.txt containing the password is removed using the `after_script` directive. This ensures that the file is removed also if the scripts fail. Notice the usage of the `srm` command (secure remove) instead of the classic `rm`, to avoid the recovery of the sensitive content. The `srm` utility needs to be installed on the runners machine or container.

3.5.1 Verifying required variables

By default, Ansible fails if a variable in the playbook is undefined. However, if the variable is passed using the `--extra-vars` option but it is not defined in GitLab, Ansible will receive an empty string variable, that is considered valid.

To avoid issues caused by empty variables it is possible to add a dedicated check task at the beginning of the playbook:

```
- name: Check required variables
  assert:
    that:
      - my_var1 != ''
      - my_var2 != ''
```

4 Other project-specific details and examples

This section collects some additional details about how we handled the configuration of some of our services and how specific issues have been solved (only non trivial cases are presented).

4.1 IA2 staging environment and job runner server

A staging environment (stage) is a nearly exact replica of a production environment for software testing. Staging environments are made to test codes, builds, and updates to ensure quality under a production-like environment before application deployment. At IA2 a dedicated server has been set up to be used as a staging environment. It hosts several services that run using some Docker Compose configurations. A simple httpd installation has been configured as a proxy to all the services.

IA2 staging server runs also a GitLab Runner instance that is used to simplify the deploy of the staging applications, but the primary runners (the ones that build most of the software and deploy on production) are located on a different dedicated server. That server doesn't host services accessible from the outside and it is used only for running GitLab jobs. This level of isolation ensures an higher security, that is desirable for such a critical machine.

4.2 Shared Maven local repository for Docker runner

Most of IA2 Java projects rely on Maven build tool. Maven downloads the dependencies a project requires and stores them in a folder named .m2 (the local repository) located in the home directory. This acts as a cache and avoid that a dependency is downloaded again every time it is needed. When a Maven build is run inside a

Docker runner, a clean image is used and all the dependencies would be downloaded again, slowing down the build. To avoid this issue a shared runner has been configured with the runner's user .m2 directory mounted on the root .m2 directory of the container. This can be achieved editing the `/etc/gitlab-runner/config.toml` file and adding the following string to the list of Docker volumes configured for that runner:

```
"/home/gitlab-runner/.m2:/root/.m2:rw"
```

Sharing the .m2 directory brought also an advantage in deploying Maven artifacts to our self-hosted repository. Indeed the credentials for deploying on it are stored in .m2/settings.xml configuration file. Without mounting this directory in the containers we would had to pass the credentials as a CI/CD variable.

Note: from GitLab 13.3 a Maven repository is available in the Package Registry section and it is possible to deploy directly there[15]. At the moment we are using a self-hosted repository using Apache Archiva[16].

4.3 Pipelines for IA2 portals

IA2 astronomical portals share a common core composed by three modules: a query library that performs SQL queries on a TAP service, a backend service (REST API) and a frontend (user interface). Each portal is built adding a customization on the core query library and on the core UI. This modular structure brings a lot of flexibility in customizing each portal but also increases the complexity of the building process. We ended up in splitting all the modules in separate repositories belonging to a common parent group. At the moment each portal has a "customization" repository containing its query library and its UI. A "devops" repository containing pipeline definitions and configuration files has also been set up to handle the composition of the various modules and to keep the deployment details separated from the rest of the source code.

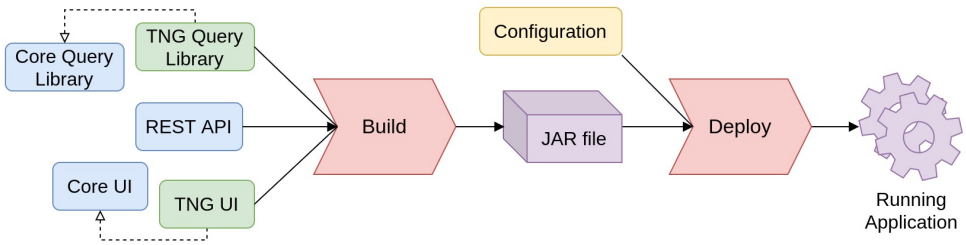

Figure 2: The diagram shows the elements that are combined in order to generate a portal and deploy it (TNG customization is used in this example). When one of the green elements is updated the related portal needs to be rebuilt but when one of the blue element is updated all the portals need to be rebuilt.

4.3.1 First attempt: monorepo approach and child pipelines as templates

This subsection describes the pipeline we set up when all the query libraries were located on the same repository (monorepo approach). Even if now the code is split in multiple repositories, our first pipeline configuration can still be an useful example of using a child pipeline as a template. Notice that GitLab has a feature named "template"[17], however we are not referring to it, but simply to repeating some similar sequences of jobs.

As said before, each portal has its own query library and all these libraries inherit from a base dependency. All these libraries were located inside the same repository, under different folders, but a similar setup could be achieved using git submodules. Both the base dependency and the specific libraries are Maven modules that require a process consisting of the same build, test and deploy commands. For this use case we created

[15] https://docs.gitlab.com/ee/user/packages/maven_repository/
[16] https://archiva.apache.org/
[17] https://docs.gitlab.com/ee/development/cicd/templates.html

a .gitlab-ci-child.yml file that defined the sequence of jobs needed by all the libraries. A `${LIB}` variable was used to specify a different build folder each time the child pipeline was invoked. In this way the single file that defined the child pipelines acted as a template for repeating multiple times the same sequence of jobs inside different folders.

However, portal-specific child pipelines needed to run after the core child pipeline had been completed, so the option `strategy: depend` was used for the core child pipeline. This option forced the trigger job to wait for the child pipeline completion. Following screenshots display what happened.

Figure 3: The `run_core_pipeline` job triggered a child pipeline defined in .gitlab-ci-child.yml file and waited for its completion, according to the `strategy: depend` setting. The child pipeline compiled the core library and deployed it on a Maven repository.

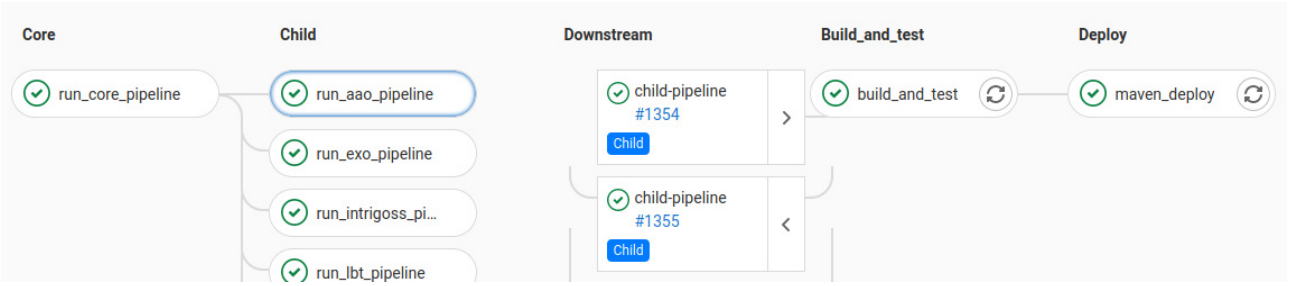

Figure 4: After the child pipeline triggered by the `run_core_pipeline` job completed, all the other child pipelines (that used the same .gitlab-ci-child.yml file) run concurrently to build and deploy all the dedicated libraries.

4.3.2 Current strategy: multi-project pipelines and generated child pipelines

Looking at the first version of the .gitlab-ci.yml file we defined for IA2 portals it was clear that even if it used inheritance it contained a lot of repetition, indeed it defined a job for each portal:

```
run_lbt_pipeline:
  <<: *run_child_pipeline
  variables:
    LIB: LBTQueryLib

run_prisma_pipeline:
  <<: *run_child_pipeline
  variables:
    LIB: PrismaQueryLib

# ...
```

This decreased the readability but also implied that every time a new portal was implemented it was necessary to add a new entry in the file.

Fortunately recent versions of GitLab allow to define dynamic child pipelines[18]. They consist in a generator job that generate a pipeline definition and a trigger job that calls it:

```
pipelines_generator:
  stage: generate
  script:
    - ./generate-pipeline.sh > generated-pipeline.yml
  artifacts:
    paths:
      - generated-pipeline.yml

generated_pipeline_trigger:
  trigger:
    include:
      - artifact: generated-pipeline.yml
        job: pipelines_generator
    strategy: depend
```

The pipeline generator script can be a simple bash script or use any other language.

A portal needs to be rebuilt and redeployed both when something changed in the shared core and when something changed in its own customized code. In the first case all the portals need to be rebuilt (using the generator script described in the previous snippet) but in the second case only a single portal needs to be rebuilt.

Each portal customization repository has its own .gitlab-ci.yml that in most of the cases simply looks like the following snippet:

```
include:
  - project: 'ia2/portals/devops'
    file: 'portal-default-pipeline.yml'
```

Listing 10: This configuration includes a pipeline defined in a different repository (the "devops" repository). All the customization repositories have the same structure, so the code is built using the same commands. Each repository contains a config.properties file that defines some variables used during the build. Other deployment-specific variables are defined in the "devops" repository itself.

This ensures that a portal is rebuilt every time its customization changes. Let's examine the content of the pipeline generator script to see how it works in the other case (when the core changes):

```
#!/bin/bash

for portal in ${PORTALS//,/ }; do
    cat <<EOF
${portal}_build:
  trigger:
    project: ia2/portals/customizations/$portal
    branch: master
EOF
done
```

Listing 11: Content of the generate-pipeline.sh script. `$PORTALS` variable is stored as a CI/CD variable and its value is the list of all the portal identifiers separated by commas. In this way it is possible to loop through all the portals and generate a dynamic list of jobs. The jobs trigger a new child pipeline for each customization repository (multi-project pipeline).

4.4 Periodic cleanup of unused Docker images

We observed that both on server running CI/CD jobs and on staging machine disk space is rapidly filled by unused Docker images. To avoid this issue we setup the following cron job:

```
0 4 * * *  docker rmi $(docker images --filter "dangling=true" -q --no-trunc) > /dev/null 2>&1
```

[18] https://docs.gitlab.com/ee/ci/parent_child_pipelines.html#dynamic-child-pipelines

4.5 A Docker image for testing on embedded PostgreSQL database

In some Java projects we are using some testing libraries[19][20] that include an embedded PostgreSQL database. The database is started at the beginning of the tests and shut down when they end. We encountered several issues in running these libraries inside a Docker container using the standard Maven image. Theoretically these libraries should include all the dependencies needed to run the database, however inside the container we encountered some errors about missing Linux libraries. This leaded us to use a different approach: we attempted to create an image for running these tests inheriting from a postgres image and then installing Maven on it. This fixed the libraries issue but it was necessary to fix other two problems in order to have a working environment: a locale error and the fact that initdb refuses to run as root.

The final Docker image is this:

```
FROM postgres:12

RUN echo 'deb http://ftp.de.debian.org/debian sid main' >> '/etc/apt/sources.list' && \
    apt-get update && apt-get install -y maven openjdk-14-jdk -o APT::Immediate-Configure=0

# To avoid error 'initdb: invalid locale settings; check LANG and LC_* environment variables'
RUN echo en_US.UTF-8 UTF-8 > /etc/locale.gen
RUN locale-gen en_US.UTF-8

# initdb cannot be run as root
USER postgres
```

4.6 Timezone issues

When running jobs inside Docker containers it is important to know that the container could be set to a different time zone compared to the host machine. This caused a failure in a test that checked the format of a date. To solve the issue (in a Debian-based image) the following lines were added to the Dockerfile:

```
ARG TZ
RUN echo $TZ > /etc/timezone
```

Then the image has been called passing the timezone build argument:

```
docker build -t "${CI_REGISTRY_IMAGE}" --build-arg "TZ=Europe/Rome"
```

4.7 Issues with generated URLs and Docker Compose

IA2 VOSpace project is composed by various services that can be run together using Docker Compose. In particular, we encountered an issue in the interaction of the following 3 services:

- file service: the web service used for downloads and uploads;
- VOSpace REST service: the service that handles files metadata and knows how to build URLs for downloading files from the file service;
- VOSpace UI: the web application that calls the other two services.

All the three services must be available to the final users, so their ports are exposed in Docker Compose configuration. However, in a testing environment, the user accesses all of them calling localhost, while each container knows the others using the names defined by Docker Compose (for example the file service could be seen as http://localhost:8080/files by the user but as http://file_service:8080/files by the VOSpace UI container).

VOSpace REST service needs to generate URLs accessible both by the user and the VOSpace UI container so we ended up forwarding the file service port inside the VOSpace UI container using a simple utility called socat. The Dockerfile calls an entrypoint script that checks for a variable defining the file service port and forwards it to the container named as defined in the Docker Compose configuration:

[19] https://github.com/opentable/otj-pg-embedded
[20] https://github.com/zonkyio/embedded-postgres-binaries

```
if [ ! -z "$FILE_SERVICE_PORT" ]; then
    socat TCP-LISTEN:$FILE_SERVICE_PORT,fork TCP:vospace_file_service:$FILE_SERVICE_PORT &
fi
```

In this way the container can use the same URL generated for the user.

After this setup the entrypoint script proceeds as usual.

4.8 Running a Docker container as a systemd service

Following systemd unit starts a Docker container as a Linux service:

```
[Unit]
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop %n
ExecStartPre=-/usr/bin/docker rm %n
ExecStartPre=-/usr/bin/docker pull <image-name>
ExecStart=/usr/bin/docker run --rm --name %n <image-name>

[Install]
WantedBy=multi-user.target
```

The idea has been taken from a blog post[21] but a dash has been added also to the last `ExecStartPre` directive. A dash before the command allows it to fail and we don't want that a service can't be restarted if our private docker registry is unavailable for some reasons.

Pulling a Docker image from a private registry requires authentication and this can be setup on a server generating a deploy token from GitLab repository settings. Then `docker login <registry-url>` command can be used with it. That will add the credentials inside `~/.docker/config.json` file. Notice that this file stores only one username and password for each registry and the deploy token is related to a single project or a group. To pull images from different project groups it is necessary to use the `--config` option of the Docker client to store multiple credentials into different paths[22]. In that case the proper configuration file needs to be passed to `docker pull` command too.

5 Conclusions

Adopting DevOps techniques reduced the time we spend in redeploying our services and leaded to a more homogeneous and less error-prone configuration. Starting to dockerize an application from the earlier stages of its development resulted in writing more configurable and modular software.

Having an automatized deploy process also reduced issues caused by outdated documentation. It happened several time that a documented installation procedure was not usable anymore due to its obsolescence. In the same way, committing the configuration files to git repositories and then let that the CI/CD process copies them to production servers is much better than modifying those files on the servers and then having to remind to update the related documentation on Redmine.

There are still some operations that we could automatize and make more reliable, like database structure updates (the usage of tools like Liquibase or FlyWay might be investigated).

Moreover we will probably experiment the usage of Kubernetes. At the moment we are running containers or jar files as systemd services deploying them with Ansible and this makes sense since some of our services are deployed to virtual machines that are external to the data center (e.g. LBT and radio portals), but we might consider the usage of Kubernetes at least for services located on the machines that we directly manage.

[21] https://blog.container-solutions.com/running-docker-containers-with-systemd
[22] https://stackoverflow.com/a/55635346/771431